PROCESSING SOUND AND MUSIC DESCRIPTION DATA USING OPENMUSIC

Jean Bresson, Carlos Agon
IRCAM – CNRS UMR STMS
Paris, France
{bresson,agon}@ircam.fr

ABSTRACT

This paper deals with the processing and manipulation of music and sound description data using functional programs in the OpenMusic visual programming environment. We go through several general features and present some toolkits created in this environment for the manipulation of different data formats (audio, MIDI, SDIF).

1. INTRODUCTION

Musical information processing often requires manipulating large musical or sound description data bases. Ordering, sorting, renaming files, automatic indexing, contents browsing or transformations are basic operations in compositional, analytical, or other experimental applications.

Visual music programming environments make it possible to define specialized and personalized programs and to adapt such sound or music description processing to specific aims in their respective domains. Real-time music/sound processing environments such as Max/MSP or PureData [5] are commonly used for this purpose. They provide convenient interactive features and efficient realtime buffering and rendering of audio or other music description data.

In this paper, we try to show how similar applications can be performed in OpenMusic (OM) [1] emphasizing some related features and specificities of this computer-aided composition environment. OM provides a set of predefined structures allowing to perform complex iterative processes on musical data in a visual and symbolic context. Specialized toolkits allow to perform these processes with music description data using standard formats.

2. OM: A VISUAL PROGRAMMING LANGUAGE

OpenMusic (OM) is a visual programming environment based on the Common Lisp language. Programs are created in patch editors, where the user/programmer assembles and connects functional units represented by boxes. Basically, OM boxes are function calls (Lisp functions or user-defined functions), and the OM patches are more or less complex graphs corresponding to functional expressions. The language provides a set of graphical control structures, such as iterations and conditional controls, as well as the possibility to carry out other programming concepts like abstraction, higher-order functions or recursion [4].

OM includes specialized functions and data structures designed for musical applications. The different musical or extra-musical objects are represented by classes (as meant by object-oriented programming) and used in the visual programs by means of factory boxes, i.e. boxes generating instances of these classes and allowing to access their different attributes (called *slots*).

Compared to most existing visual programming environments, OM has the particularity to run a demand-driven execution model. In this model, the user triggers execution by evaluating a box somewhere in the visual program graph, which recursively evaluates the upstream connected boxes and returns a value. This model is very close (semantically equivalent) to the evaluation of functional expressions in Lisp. It also provides time-independent conception of programs and proved to be a relevant model for formal music composition and generation of complex and structured musical data.

3. FUNCTIONAL BATCH PROCESSING IN OM

In order to introduce our subject and emphasize some possibilities provided by this environment, we present an example of how OM visual programming can be used for the design of automated audio file analysis and processing.

Let us first consider the case of a single sound file to be processed. Figure 1 shows an OM patch performing successive operations using sound analysis tools available in OM [2]: 1) Analysis of the sound (*sound* object, top of the figure) with a fundamental frequency estimation (external call to the sound analysis kernel *pm2*). 2) Storage of the analysis results as a temporary SDIF file (*SDIFFile* object). 3) Extraction of the frequency estimation data (“1FQ0”) from the SDIF file. 4) Determination of a pitch value (e.g. in this example, the mean of all estimated frequency values – 186 Hz). 5) Conversion of the pitch value (frequency in Hz) to

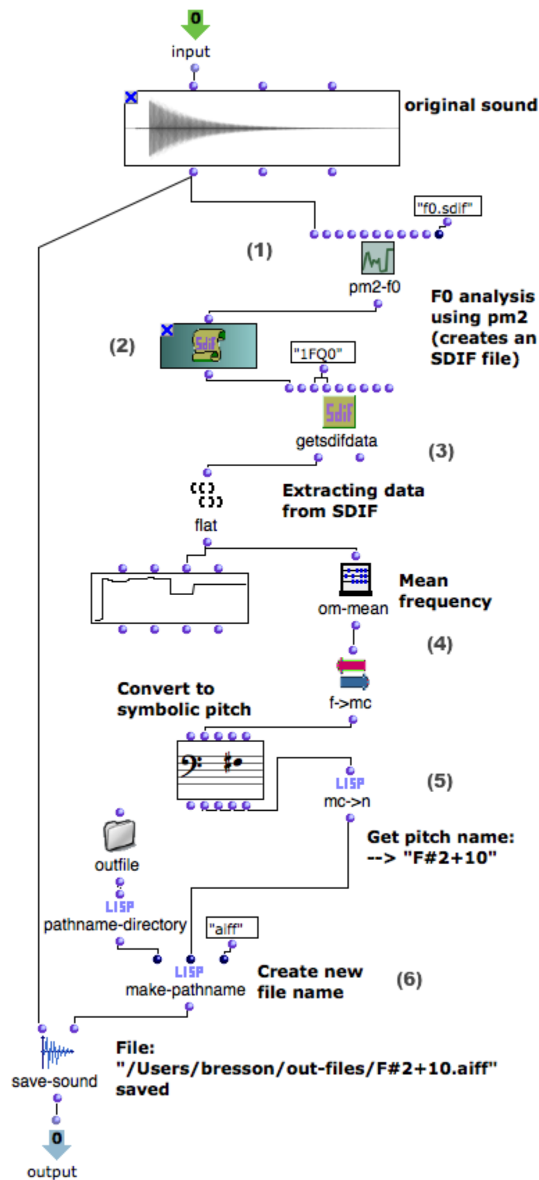

Figure 1. Example of sound file processing in OM: copy and rename a sound file after its inner estimated pitch. This patch will be referred as *auto-rename* in the next figures.

a symbolic (MIDI) value (5410 midicents in the example, i.e. F#2 + 10 midicents) and then to a textual name (e.g. "F#2+10"). 6) Renaming and saving the sound file after the estimated note value (e.g. "∼/mysoundfiles/ F#2+10.aiff").

Visual programs such as the one in Figure 1 can be embedded in abstractions, i.e. autonomous procedures or functional elements usable in other programs, where they communicate by argument passing through some inputs and outputs. These abstractions are represented by "patch" boxes, that is, they are patches used as functions in higher-level patches.

Various batch processing can be derived from our example procedure. In Figure 2, it is applied successively to a list of files contained in a given directory. The program from Figure 1 is now embedded as an abstraction in a patch box, which has one input corresponding to the sound to be analyzed and saved with the appropriate name. This patch box is used in association with the *mapcar* box, a Lisp function applying another function to the successive elements of a list. The function *om-directory* returns the list of all the files present in a given directory. The *mapcar* call will therefore trigger the analysis and renaming of each one of these files.

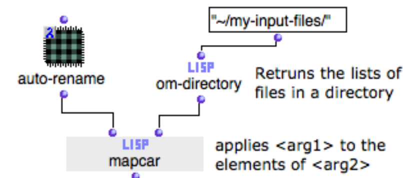

Figure 2. Applying the procedure to the files in a directory.

The process in Figure 2 is known as *higher-order programming*: by setting the *auto-rename* box to a specific mode ('lambda' – see little λ icon on the patch box), its content is compiled and returned as a function (lexical closure) to the *mapcar* box: *mapcar* is a *higher-order function* accepting functions as parameters. It is therefore possible to perform the same iterative process with other functions or patches, by simply connecting them instead of the current one.

Other extensions can be implemented: We could now suppose that our initial sounds are melodic sounds, which should first be segmented in "notes" and then processed as previously. This can be done with preliminary transient detection and segmentation of the initial sound file (also available in OM), and creating other iterations where each segment wound be subjected to pitch analysis. The *omloop* (a patch, i.e. a visual program or abstraction, with specific features adapted to the creation of iterative processes—equivalent to the Lisp *loop* statement) would be for instance an appropriate tool for this purpose. Further on, this new process could also be embedded in a directory iteration, like in Figure 2. The resulting visual program would then build a sound bank of single pitch samples out of a melodic sound directory.

4. ENHANCING INTERACTION

In order to enhance the interaction over the visual program design and execution, a set of special boxes (called *dialog-item* boxes) have been created, which correspond to standard user interface dialog-items connected to the boxes and components of the OM patches. The dialog-item boxes include simple text input and display, check boxes, buttons, pop-up menus, multiple-choice lists, sliders, etc. (see Figure 3).

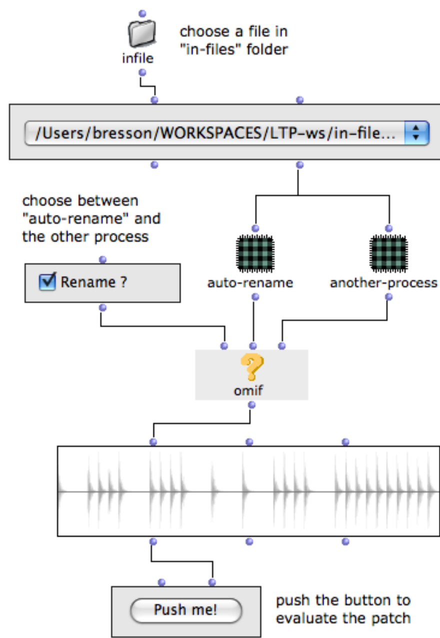

Figure 3. Example of use of some dialog-item boxes in OM patches: *pop-up-menu* (top—used to select a file among the contents of a directory), *check-box* (middle), *button* (bottom—triggers the evaluation of the sound box and the global visual program execution).

5. STANDARD FORMATS TOOLKITS

In the next sections we describe specialized toolkits dedicated to the manipulation of musical data using standard formats, and how they can be used for devising processing algorithms on this data.

5.1. MIDI

Although several new formats tend to propose more complete representations, MIDI is still one of the most widespread platform for the interchange and storage of "symbolic" musical data. A significant amount of information can be stored in MIDI files or transferred through MIDI streams, such as program changes, continuous controllers, tempo, textual data, and so forth.

In OM, MIDI data can be manipulated thanks to a complete toolkit of classes and specialized functions, allowing users to personalize its processing and transfers in particular situations. The primitive in this toolkit is the class *MidiEvent*, whose slots represent the type, date, channels and values of a MIDI event. An instance of this class constitutes a symbolic object which can be represented in OM patches and take part to processing algorithms. Starting from this primitive object, other classes have been defined, representing more complex and structured data such as event sequences, continuous controllers, etc.

A number of OM functions allow to process and convert musical objects and MIDI structures. Eventually the MIDI events sequences can also be stored and saved as MIDI files. Wide ranges of algorithms for the processing and generation of complex MIDI event sequences can therefore be implemented graphically and interactively, using all the available features of the visual programming language.

The function *get-midievents*, for instance, receives an object (e.g. a MIDI file, one of the aforementioned MIDI classes, or traditional musical objects like chords, voices, etc.) and returns a list of MIDI events (instances of the class *MIDIEvent*) corresponding to this object or to its closest description using MIDI events. The higher-order programming features can be used here again and allow for a personalized processing: An optional input parameter of *get-midievents* allows one to specify a function designed to process *MIDIEvents* and determine whether they should be considered or not in the conversion or transfer. That is, the behaviour of *get-midievents* can be let as default or specialized by the user. A set of predefined filtering functions are provided, e.g. *test-date*, *test-type*, *test-channel* etc., which will allow to select events in a given time interval or, respectively, of a given type or a given channel, but more complex filtering processes can be defined by the user as well (in Lisp or as visual programs—see Figure 4).

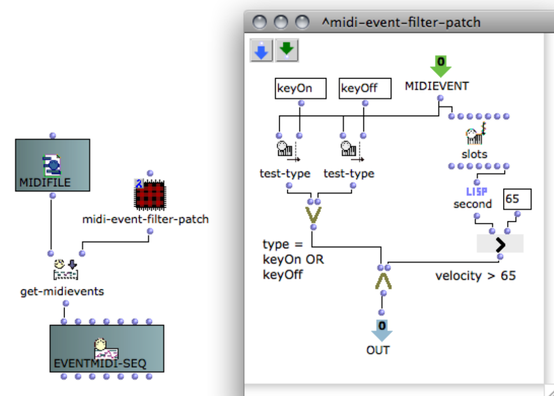

Figure 4. Filtering MIDI events by user-defined process. The patch *midi-event-filter* is used as the test function for *get-midievents*.

5.2. SDIF

SDIF is a standard data format for the codification, storage and transfer of sound descriptions. This format is currently used by an increasing number of composition and sound processing tools and software for varied purposes and data types [6]. An SDIF file basically describes the evolution of data sets of predefined or user-defined types, stored as *matrices* and embedded in time-tagged *frames*.

SDIF is also supported in OM and is used to communicate sound description data with external sound analysis and synthesis tools (see for instance the *SDIFFile* box in Figure 1). It also helps unifying the codification and manipulation of the various possible types of such descriptions inside the environment [3]. Various tools allow to inspect, extract and manipulate the contents of an SDIF file in OM visual programs.

A set of classes have been added to the OM library to represent the components of a sound description in this format: The classes *SDIFBuffer*, *SDIFStream*, *SDIFFrame* and *SDIFMatrix* allow to reconstitute the hierarchical structure of the SDIF data; and the *SDIFType* object stands for a data type definition to be declared. In Figure 5 a set of data generated in OM is processed and formatted as SDIF matrices, then stored in a file.

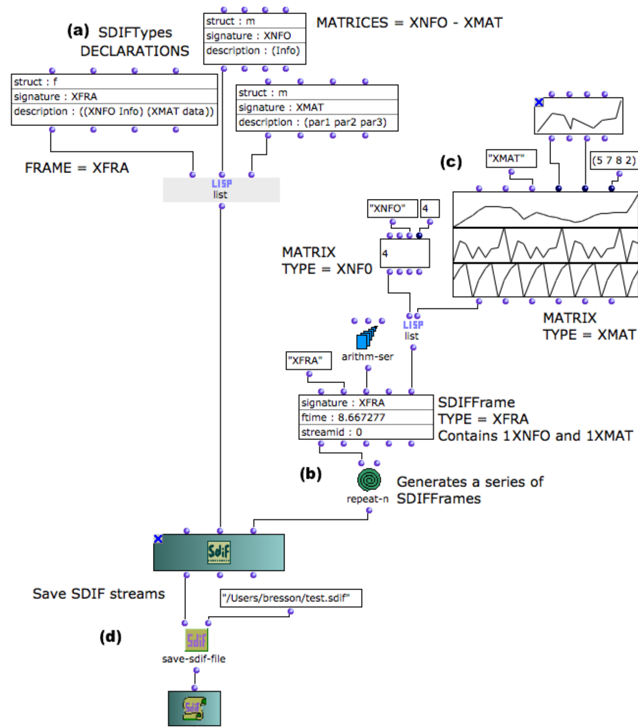

Figure 5. Generating SDIF data in OM: Three new types are declared (a), then frames (b) and matrices (c) of these types are created and collected to constitute the frame stream (d).

6. CONCLUSION

The particular programming paradigm, execution model and other specificities of the OM environment, such as iterative processes or higher-order functions, may lead to original ways of thinking and designing music description data analysis, manipulation or generation processes.

Complementarily to automatic conversions between low-level description data and the high-level musical ones, the visual programming tools presented in this paper enable dealing with this low-level description data in OM visual programs. Even though, the environment provides symbolical representations of the related processes and allow to maintain intuitive and high-level control. Moreover, this integration with computer-aided composition ensures the possibility to connect the different data sets generated or manipulated in these processes to symbolic musical data such as chords, sequences or rhythmic structures.

Even if OM is principally dedicated to music composition, some collaborations in other contexts, such as musicology or experimental data analysis, gave us the opportunity to experiment the use of these tools for extra-compositional purposes in different situations where automatic analysis, processing and/or formatting of music or sound file databases were needed.

This approach could be extended to other data formats and standard protocols as well. For real-time interaction, for instance, the class *OSCEvent* already allows to build, process and send OSC formatted data [7] in OM. However, concrete and real-size applications of OSC messages creation and scheduling in OM visual programs remain to be found and developed.

7. REFERENCES

[1] G. Assayag, C. Rueda, M. Laurson, C. Agon, and O. Delerue, "Computer Assisted Composition at Ircam: From PatchWork to OpenMusic," *Computer Music Journal*, vol. 23, no. 3, 1999.

[2] J. Bresson, "Sound Processing in OpenMusic," in *Proc. Int. Conf. on Digital Audio Effects*, Montréal, 2006.

[3] J. Bresson and C. Agon, "SDIF Sound Description Data Representation and Manipulation in Computer Assisted Composition," in *Proc. International Computer Music Conference*, Miami, 2004.

[4] J. Bresson, C. Agon, and G. Assayag, "Visual Lisp/CLOS Programming in OpenMusic," *Higher-Order and Symbolic Computation*, vol. 22, no. 1, 2009.

[5] M. Puckette, "Combining Event and Signal in the MAX Graphical Programming Environment," *Computer Music Journal*, vol. 15, no. 3, 1991.

[6] D. Schwartz and M. Wright, "Extensions and Applications of the SDIF Sound Description Interchange Format," in *Proc. International Computer Music Conference*, Berlin, 2000.

[7] M. Wright, "Open Sound Control: an enabling technology for musical networking," *Organised Sound*, vol. 10, no. 3, 2005.