Reactive Visual Programs for Computer-Aided Music Composition

Jean Bresson

Reactive Visual Programs for Computer-Aided Music Composition

Jean Bresson
STMS Lab: IRCAM-CNRS-UPMC
1 place Igor Stravinsky, Paris, France
jean.bresson@ircam.fr

***Abstract*—We present a new framework for reactive programming in OpenMusic, a visual programming language dedicated to computer-aided music composition. We highlight some characteristics of the programming and computation paradigms, and describe the implementation of a hybrid system merging demand-driven and event-driven evaluation models in this environment.**

I. Introduction

Numerous computer systems for music composition have been designed as end-user programming languages. Computer-aided composition environments today integrate flexible tools that allow composers to create music and to develop creative processes embedding algorithmic approaches, musical notation, graphical representations, i/o devices and communication in (visual) programs and programming languages [1].

In this paper we present a new framework for reactive programming in the OpenMusic computer-aided composition environment. After a brief presentation of OpenMusic (Section II), we will single out some specific aspects of this environment that bear important comparisons with other musical systems (Section III). We describe the implementation of a model merging demand-driven and event-driven evaluations in OpenMusic (Section IV), and discuss a number of possible applications (Section V).

II. OpenMusic

OpenMusic (OM) is a visual programming environment dedicated to music composition. The visual language is based on Lisp and extended with musical tools, data structures and editors [2][3]. This environment is used by composers worldwide and has contributed to numerous compositional projects and pieces from the contemporary music repertoire [4].

Visual programs in OM are made of boxes and connections: the user/programmer incrementally builds and evaluates directed acyclic graphs representing functional expressions.

Fig. 1 shows and example of an OM visual program. Each box in this program represents a function,[1] a value or an object generator/container.[2] Inlets, at the top of the boxes, represent the inputs or arguments of the functions, and outlets, at the bottom of the boxes, correspond to returned value(s).

OM implements a *demand-driven* evaluation model: the nodes in the visual program graph are computed on user request, depending on the values he/she wishes to calculate or update (see Fig. 2a). Typically in Fig. 1, the user would evaluate the boxes RHYTHM, PITCHES_SEQ or other downstream

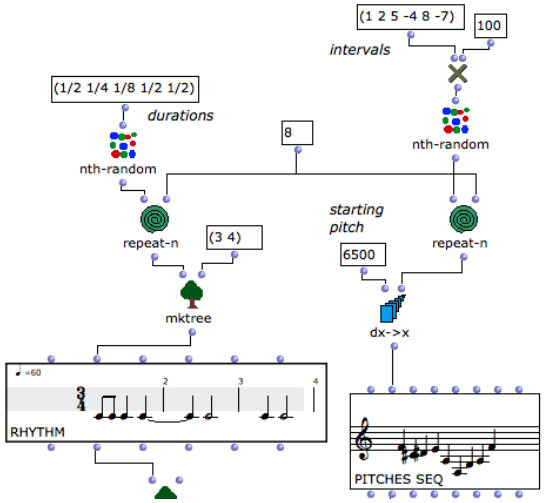

Fig. 1: Example of an OpenMusic visual program.

musical structures (not visible in this figure). Such evaluation produces a chain of function calls following the box input connections. Upstream terminal box calls yield values that are then processed downwards until the box from which the request originated.

Evaluations, data input and rendering are generally interleaved as part of the composers' programming work-flow. Intermediate values can be edited using graphical editors, and locked in order to prevent reinitializations on future updates (locked boxes behave as terminal nodes in the call graph).

III. Computation Models in Music Programming Systems

Among the different computer tools proposed to musicians today, an important distinction exists between "deferred-time" and "real-time" systems. These two approaches correspond to different ways of using computer systems for music: on the one hand, the symbolic manipulation of complex and structured musical data involved in the composition of a piece of music (scores, but also sounds or other types of data), and on the other hand, real-time audio processing and live interaction in concerts or performance contexts.

Executions in real-time musical systems are reactive processes driven by internal clocks, external events, data streams or signals that are processed through data-flow graphs.[3] In these systems music is produced "on the fly" by the program

[1] Functions can written in Lisp or graphically as internal visual programs.

[2] Object boxes are actually class constructors or factories, as usually found in object-oriented systems.

[3] Max [5] is a popular visual language in this category. We will use it to illustrate inter-application communication processes in Section V (Fig. 4).

(each step of the program run generates a bit of music: an event, a sequence of audio samples, etc.). Therefore, the temporal dimension in the musical structures is practically merged with the "run-time" that flows during computation, which limits the possibility to "compose" with musical time.[4]

In contrast, compositional systems like OpenMusic generate "off-line" musical data in a *deferred time* context. If these systems are interactive (in the sense that they include user interactions), the programs created by the user are *transformational*: they process inputs, produce an output and terminate.[5] The output musical structures (scores, sounds, etc.) are played, read or interpreted by a performer or sequenced/rendered at a later stage. Such systems therefore make a distinction between computation time and musical time: the musical result is produced when the overall process is over, and its temporal structure is independent from the "run time" of the program. This paradigm allows for the formalization and generation of more sophisticated musical structures, without time-related restrictions or constraints (e.g. limitations of the accessible time domain, or constraints on complexity when results are required within strict time delays).

The deferred-time paradigm also have a number of drawbacks. While advantageous in its full and precise control over program execution, it can be impractical in that it forces frequent explicit updates after modifications or data input. The transformational paradigm of OM's model also has the disadvantage that it tends to isolate compositional processes from interactions with the external environment.

IV. Reactive Visual Programs in OM

In a recent paper [8], we have proposed an extended semantics of OM merging off-line/demand-driven and reactive/event-driven evaluation models (see Fig. 2). In this new situation, events or changes that occur in specific boxes in the visual programs propagate and update downstream branches and boxes, thus introducing more interactive aspects in compositional processes.

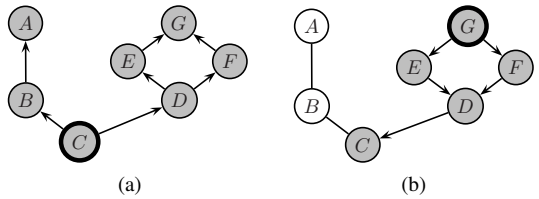

Fig. 2: Control flow in demand-driven vs. data/event-driven evaluation. (a) *Demand-driven*: the evaluation of C propagates up in the graph to request input values. (b) *Data-driven*: G activates an execution that propagates down to C.

At the moment, we consider the first-order functional graph only, that is, the "top-level" graphical elements of the visual programs. The implementation of this system can be relatively straightforward but needs to respect a number of constraints. Most importantly, we view reactivity as an optional and local feature and so this new functionality must not interfere with the environment's pre-existing semantics and behaviour. A number of features and concepts are added to the visual language for this purpose, that we list in this section.

1) Active Boxes: In order to define reactive chains in the functional graphs, there is a new attribute of the OM boxes that determines their reactive status. Reactivity can be switched on and off by the user with a keyboard short-cut (the frame of the box is highlighted so that reactive objects are easily distinguishable from non-reactive ones). A reactive box systematically registers boxes downstream from it, i.e. following boxes that would use its value as an argument, and eventually propagate updates.

2) Events: Events are actions or changes on a box that are likely to require an update of other boxes in order to restore the consistency of functional relations in the program. Events can occur for instance with user actions such as the re-evaluation of a box yielding a new value (as in the standard evaluation model), the modification of a "value" box or of a box input, or specific editing actions in object editors that are validated and applied to the associated box. Events can also come from external processes or interruptions caught by the system.

3) Updates: Events are propagated and update the box descendants through a *notification* mechanism. A notification starts at an event (that is, a modification of a reactive box value), and is propagated to its descendants by a depth-first traversal of the visual program graph following the output connections. Every reactive descendant in turn propagates the notification. When a terminal box is reached (a box with no direct reactive descendent), this box is evaluated following the standard OM demand-driven model, hence respecting the existing scheduling strategy and semantics for program executions.

The box at the origin of the event is temporarily "locked" during the process, so that it is not re-evaluated as part of the update. Notified boxes in this process are also marked during the propagation, in order to avoid multiple visits to the same sections of the graph.

In this system, events are ordered upon occurrence and are handled asynchronously in a single thread. Every incoming event triggers an update and the next one is processed when the previous update terminates. This choice in event-handling simplifies the update process and is acceptable in both compositional contexts and applications.

This process successfully implements first-order reactive visual programs in OM. It has the advantages that it does not overload programming tasks and is completely transparent and conservative with respect to the initial language design.

V. Extensions and Applications

The reactive mechanism implemented in OM performs automatic computations and provides direct feedback for the different user/programmer actions. In so doing, it alters programming practices and needs in the environment and enables a variety of new applications. We provide a number of examples below.

A. Interactive Widgets and Components

The reactive mechanism facilitates experimentation with input values and, in certain situations, can be useful when combined with OM programs' special interface components such as buttons, list/menu selectors, sliders, etc. The insertion

[4]Recent works in real-time musical environments such as *bach* [6] shall be mentioned here, for they bring higher-level musical constructs and processes in this constrained temporal paradigm. *bach* can be seen as a symmetrical counterpart to the present work, for it aims at bridging computer-aided composition and real-time/reactive visual programming frameworks, from the real-time systems perspective.

[5]See [7] for a discussion about transformational vs. reactive systems.

of an *event* after the standard activation procedure of an interactive component makes it handy to use at setting the values for the inputs of a visual program. Fig. 3 shows an example of a melody calculated from a curve, whose values are scaled in a pitch range determined by two sliders. The reactive chain between the sliders and the score box makes this value range controllable interactively with immediate feedback.

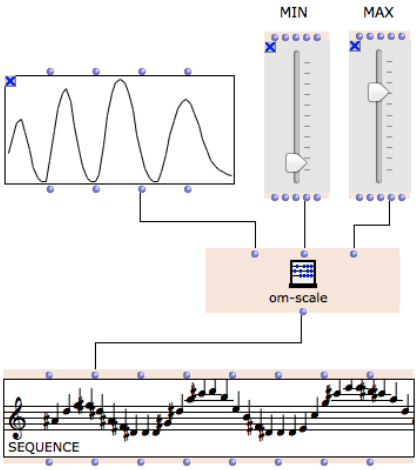

Fig. 3: Interactive control of input values in a reactive visual program using slider boxes. The two sliders (MIN, MAX), *om-scale* and SEQUENCE are reactive boxes.

It is important to note that immediate reactivity is not always desirable, hence the importance of local and optional control of the reactive mechanism. OM fosters an experimental approach, allowing for visual programs to be constructed incrementally through a process of trial and error. Sometimes boxes can be temporarily connected in a certain way by mistake, or for different purposes than immediate evaluation. Additionally, evaluations can sometimes take a long time to complete (e.g. in complex optimization problems, or in programs involving large data structures or high precision signal processing). In these cases of computationally-intensive processes, the visual program is usually run only once at the end of its construction, and the use of reactive sliders such as shown in the previous example is not an adequate means for triggering executions.

B. Communication with External Systems

The reactive framework creates new possibilities for communication between the computer-aided composition environment and other applications, and more generally with its external context. Networked inter-application communication is supported via MIDI or UDP/OSC[6] and available in the visual programs respectively through the *MIDI-in/out* and *OSC-receive/send* boxes.

When activated, MIDI or OSC "receive boxes" run a server thread receiving incoming messages on a specified port. A reactive behaviour (*event*) is added to these boxes, which updates their value and triggers a notification when a message is received. Visual programs containing active receive boxes hence become reactive to such events or messages sent by external applications, input devices or instruments that communicate with these protocols.

[6]OSC is a format for UDP packets created for musical applications [9].

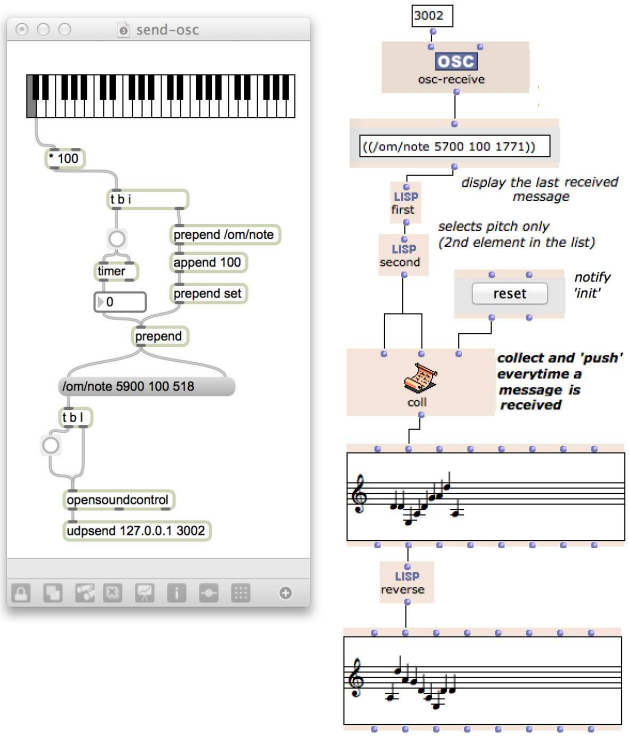

Fig. 4: Reception, collection and processing of incoming data. The Max program (window on the left) sends a "note" message via UDP every time the user clicks on the piano keyboard widget. The OM program (at the right) collects information from the messages into the *coll* box. At every received data the *push* (second input of *coll*) also receives a notification and triggers an update of the downstream part of the program.

C. Data Collection

Reactive boxes and notifications handle "instantaneous" data and events. In order to integrate compositional processes, these data generally need to be gathered in more complex data sets and mapped to time structures. A specific box (*coll*) has been created for this purpose, which instantiates a local storage for dynamic data collection in OM visual programs.

The *coll* box constitutes a special case in the update mechanism. It behaves as an intermediate step where notification temporarily stops and inputs are evaluated. The box has three inputs: when the first one (labelled *data-in*) collects some data, this data is added into a list that constitutes the *coll* "memory". The second input (*push*), makes *coll* behaving as a standard reactive box, propagating the notification through its descendants. The third input (*init*) reinitializes the memory. When it is evaluated (typically, after the propagation has triggered an evaluation), *coll* returns the current contents of its memory as a list of all previous collected data.

Coll can be associated to OSC or MIDI receive boxes to create reactive programs in OM that structure and process live inputs from real-time systems, players or instruments. Fig. 4 shows an OM program collecting and processing data from the Max software using *OSC-receive* and *coll*. In contrast to Fig. 1, evaluations here are driven by incoming messages in *OSC-receive*. The growing contents of the *coll* memory is converted into a data structure (a score) that can be further processed in the visual programming environment.

D. Handling Groups and Time in Data Collection

It is frequent in communication frameworks that several messages need to be considered as being simultaneous or otherwise "grouped" together. (If the messages are musical notes then this idea would correspond to the concept of a *chord*.) However, it is very unlikely that two events belonging to the same group would be sent and received at exactly the same time (and even so, they would not be handled simultaneously by the system). Time-tagging and/or structuring events provides a solution for sender applications to deal with this situation [10], but all events are not necessarily formatted so.

When no bundle or time-tagging is present in the communication protocol (e.g at receiving simple UDP or MIDI messages), internal timing can be used to gather the collected data according to specified time intervals. The *group-in* box implements this behaviour: it has one input collecting data (similar to the *coll* input) and a second input setting a time frame during which events are gathered and considered as part of the same data set. This box generates a slight delay and propagates collected data at once after this delay is over.

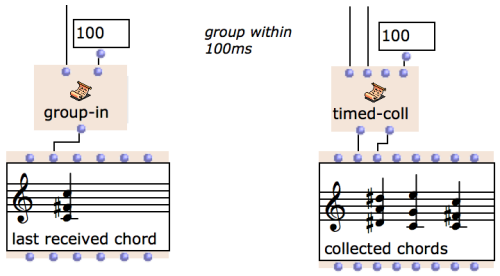

Fig. 5: Grouping incoming data. A delay of 100ms is set to gather and transmit or collect notes as chords.

The *timed-coll* object also extends *coll* with the *group-in* behaviour and gathers the items collected during a time frame into separate data chunks. Depending on the timing of incoming events, the result in Fig. 4 using *timed-coll* would be a list of pitch lists, corresponding to a sequence of chords in the score (see Fig. 5).

E. Event-Driven Conception of the Visual Programs

Reactive applications such as the one in Fig. 4 tend to emulate and orient the conception of the visual programs towards the data-driven paradigm (Fig. 2b), where the propagation of notifications needs to be controlled in the same way as evaluation. (The *coll* mechanism, for instance, can be seen as the data-driven counterpart to the *collect* primitive in standard Lisp or OM loops).

Several tools inspired by reactive systems like Max are being added to the OM reactive framework to help it expand beyond its original paradigm. One such tool is *route*, a utility that filters notifications using on a set of tests performed on its input. *Route* is also a particular case in the update mechanism as it requires stopping the notification and evaluating inputs before selecting where to propagate an update.

Other "data-driven" utilities include for instance an internal send/receive mechanism, also inspired from the Max *send* and *receive* tools, allowing to propagate notifications and updates in the visual programs without passing through the whole path of intermediate connections.

VI. Conclusion

In this paper we have summarized the significance of deferred-time computation in computer-aided composition systems and outlined a reactive framework in OM where this characteristic is combined with interactive processes. The reactive extension of OM introduces a new conception of the computer-aided composition framework and of its connection to the external context. Visual programs and compositional processes are built and run interactively, but also "live" on their own and can be driven by external processes and interactions. The generated musical structures can in turn be re-injected into a "real" time flow, either via networking or direct rendering, as a response to an event in the reactive system. This kind of reactive loop, in which real-time processing leaves space for advanced formal/compositional computation, opens interesting perspectives for enhanced musical expressiveness in interactive situations.

Our first experiments with the system show that OM users can easily shift to the reactive paradigm and reuse their tools and libraries in this new computational framework.

The main questions raised by this project are now related to the notions of time involved in reactive musical systems: how to deal with the asynchronous occurrence of events; with the relations between the duration of computations, the real time of interactions, and the duration of the computed time structures. We have only briefly touched on these questions in this paper, and plan to address them in more detail in future works.

OpenMusic is a free and open-source software.[7] The reactive extension can be loaded dynamically as an external library, and shall be included as a native feature in future versions.

Acknowledgements

This work is done with the support of the French National Research Agency projects with reference ANR-13-JS02-0004-01 and ANR-12-CORD-0009.

References

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

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

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

[4] C. Agon, G. Assayag, and J. Bresson, Eds., *The OM Composer's Book (2 volumes)*. Editions Delatour / IRCAM, 2006-2008.

[5] M. Puckette, "Combining Event and Signal Processing in the Max Graphical Programming Environment," *Comput. Music J.*, vol. 15, no. 3, 1991.

[6] A. Agostini and D. Ghisi, "Bach: An Environment for Computer-Aided Composition in Max." in *Proc. Int. Comp. Music Conf.*, Ljubljana, 2012.

[7] D. Harel and A. Pnueli, "On the Development of Reactive Systems," in *Logics and Models of Concurrent Systems*. Springer Verlag, 1985.

[8] J. Bresson and J.-L. Giavitto, "A Reactive Extension of the OpenMusic Visual Programming Language," *J. Visual Lang. Comput.*, 2014.

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

[10] A. Schmeder and A. Freed, "Implementation and Applications of Open Sound Control Timestamps," in *Proc. Int. Computer Music Conf.*, Belfast, 1998.

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