Reactive Visual Programs in OpenMusic

Jean Bresson

Reactive Visual Programs in OpenMusic

EFFICACe WP-1.2 – ANR-13-JS02-0004-01

Jean Bresson

2014

1 Introduction

OpenMusic (OM) is a visual programming language based on Common Lisp (Bresson et al., 2009). It implements a *demand-driven* computation model, evaluating functional expressions after user requests (see Figure 1a). This approach differs from real-time musical systems which react to internal clocks, external stimuli or data streams following a "data-driven" approach (Puckette, 1991) (see Figure 1b).

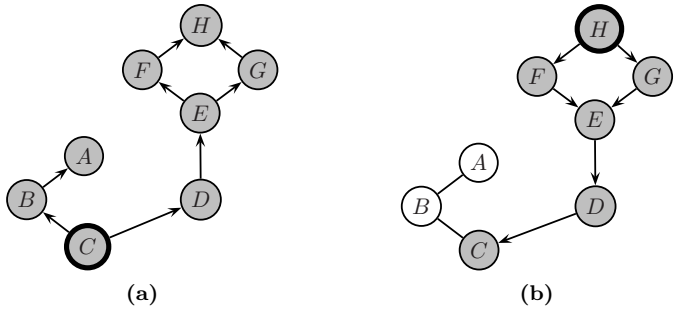

Figure 1: Data-driven vs. Demand-driven execution. (a) *Demand-driven* (the default model in OpenMusic): The user requests the value of node C. This evaluation requires upstream nodes of the graph to evaluate in order to provide C with input values. (b) *Data-driven* (e.g. realtime musical systems): The node H triggers the execution and "send" a value that propagates down in the functional graph.

These two different approaches are usually associated respectively to the domains of composition and performance. They correspond to different ways of using computer systems for music: on the one hand, advanced symbolic manipulation of complex and structured data involved during the composition of a piece of music (scores, but also sounds or any other musical data), and on the other hand, real-time audio processing and live interaction during concerts. Demand-driven computer-aided composition systems generate "off-line" material (event sequences, scores, sounds) that are played, read or interpreted by a performer (if this data is output or converted to a score) or sequenced/rendered by another system (e.g. a player or sound synthesizer). On the other hand, real-time (data- or event-driven) systems react to user inputs and process incoming messages or signals during their activation time.

This long-standing partition has perpetuated along the years, and is perceived by musicians as a frustrating constraint at developing applications mixing interaction and formal compositional processes. One of the objectives of the EFFICACe project is to combine these two approaches and propose a renewed conception of computing and interaction in computer-aided composition.

In our working context, the systems we are considering happen to be programming environments (and in particular, *visual* programming environments). From a technical point of view, the two highlighted approaches (computer aided composition vs. real-time) can be thought in terms of a distinction between transformational and reactive systems (Harel and Pnueli, 1985).

The computing power of standard workstations is today no longer an obstacle at the integration of complex symbolic computations in reactive systems. The relevance of a convergence between the expressive

power of declarative composition-oriented languages and the interactive features of real-time data-flow emerges through several recent trends in computer music, such as *live-coding*, where programming activity is embedded in the concert performances (Wang and Cook, 2004), or in computer accompaniment or automatic improvisation systems, where complex and structured musical parts are generated from live musical inputs (Assayag et al., 2006; Nika and Chemillier, 2012). We believe these are premises for a radical conceptual evolution in computer music, mixing the declarative, compositional and interactive aspects.

Our objective here is to propose a programming framework in OM that will combine these two approaches of music and computation in a common compositional environment. This objective does not necessarily involve processing audio signals or events in real time, but is motivated by the need to develop interactive compositional processes that can be integrated in interactive contexts, assuming that a best effort strategy is acceptable. We propose to integrate the notion of *reactivity* in OM visual programming, without giving up the off-line paradigm that makes the specificity of the computer-aided composition approach.

2 The Reactive Model

This works builds upon a formal semantic model of reactive programming in OM.[1] This model, as well as several aspects of the reflection and tools presented in this report, are published in (Bresson and Giavitto, 2014). In this first section we present a brief overview of the theoretical model introduced in this paper.

2.1 Basic semantics

A semantic basis for the default (demand-driven) evaluation model of the OpenMusic visual programming language is required to introduce the reactive aspects.

Visual Programs

Visual programs in OM are made of boxes and connections, that the user/programmer assembles into directed acyclic graphs representing functional expressions. Every box represents a function (defined in Lisp or graphically as an internal visual program) or a data-structure generator/container (actually, a class constructor or factory, as usually defined in object-oriented systems). A number of inlets (at the top of the boxes) represent the inputs or arguments of the functions, and their outlets (at the bottom of the boxes) represent the returned value(s). The connections between box inputs and outputs therefore determine the functional composition of the programs.

Figure 2 shows and example of an OM visual program. At the moment, we consider first-order functional graphs only for this reactive extension, that is, the "top-level" graphical elements of the visual programs.

Formally, we define a visual program with :

- $\mathcal{B}$ the set of box identifiers (boxes $b, b', b_1 ...$ range over $\mathcal{B}$).
- $\mathcal{E}$ the set of edge identifiers (edges $e, e', e_1 ...$ range over $\mathcal{E}$).

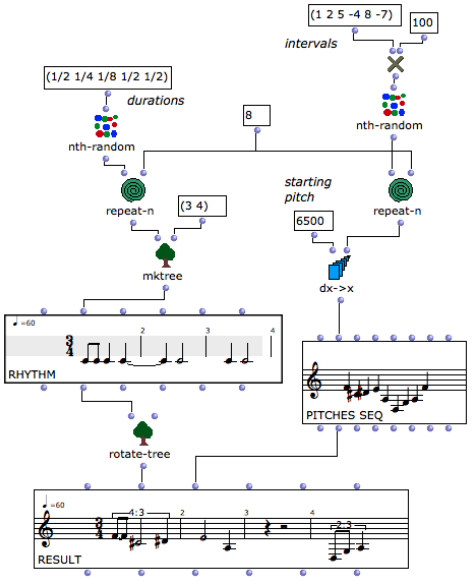

Figure 2: An OM visual program. This program calculates a musical score of 8 notes ; each one with a pitch and rhytmic figure randomly picked among a set of values, respectively for durations and successive intervals.

[1] This theoretical model was initially developed in the context of the INEDIT project (ANR-12-CORD-0009).

The functions *in* and *out* are total functions from $\mathcal{B}$ to $\mathbb{N}$ giving respectively the number of inputs and the number of outputs of a box. A visual program, or patch, is a quadruple $G = (\mathsf{B}, \mathsf{E}, \mathsf{s}, \mathsf{t})$, where $\mathsf{B} \subset \mathcal{B}$ is the finite set of boxes of G, $\mathsf{E} \subset \mathcal{E}$ is the finite set of connections between these boxes, and s (source of an edge) and t (target of an edge) are total functions from E to $\mathsf{B} \times \mathbb{N}$ representing the connectivity in the patch.

The connectivity is guaranteed by the following conditions on s and t:

1. if $\mathsf{s}(e) = (b, k)$ then $1 \leq k \leq \mathit{out}(b)$;
2. if $\mathsf{t}(e) = (b, k)$ then $1 \leq k \leq \mathit{in}(b)$;
3. the function t is injective;
4. let $<_G$ be the binary relation on $\mathsf{B} \times \mathsf{B}$ defined by $b <_G b'$ iff there exists e, k and k' such that $\mathsf{t}(e) = (b, k)$ and $\mathsf{s}(e) = (b', k')$. Let $\prec_G$ be the transitive closure of $<_G$. The relation $\prec_G$ is a strict partial order (*i.e.* an irreflexive, asymmetric and transitive binary relation).

The strict partial order $\prec_G$ formalizes the *functional dependencies induced by* G. A *chain* in G is a sequence $(b_1, b_2, \dots)$ of boxes of G such that $b_i \prec b_{i+1}$. There is no infinite chain because $\prec$ is a strict partial order and B is finite. We define b^+ to be the set upper bounds of b by $b^+ = \{b' \mid b \preceq b'\}$ and the set of lower bounds of b as $b^- = \{b' \mid b' \preceq b\}$.

Evaluation

The construction of a program in OM is an incremental process interleaved with data inputs and partial evaluations. The nodes in the graph can be computed at any time on user request, depending on the values or data structures he/she wishes to calculate or update. Evaluations produce chains of function calls following the connections in the graphs. For instance, the request for the *RESULT* box value in Figure 2 would recursively call *rotate-tree*, *RHYTHM*, *mktree*, *repeat-n*, etc., then *PITCHES_SEQ*, *dx->x*, etc. Upstream terminal box calls return a value and stop the recursive call chain. The returned values then generate a data flow stream down to the initial requested box value.[2]

We call $\mathcal{V}$ set of values handled in OM, and suppose a distinguished element $\star$ of $\mathcal{V}$ used to represent a default value. Each box $b \in \mathcal{B}$, is associated to $\mathit{out}(b)$ functions $[\![b]\!]_k$ giving its semantics:

$$[\![b]\!]_k : \mathcal{V}^{\mathit{in}(b)} \to \mathcal{V}, \quad 1 \leq k \leq \mathit{out}(b),$$

The semantics of a patch $G = (\mathsf{B}, \mathsf{E}, \mathsf{s}, \mathsf{t})$, is a function $[\![\cdot]\!]_G(\cdot) : \mathsf{B} \times \mathbb{N} \to \mathcal{V}$, which associates a value to each output k of a box $b \in \mathsf{B}$ and is defined by the recursive equation:

$$[\![b]\!]_G(k) = [\![b]\!]_k(v_1, \dots, v_{\mathit{in}(b)}),$$

where

$$v_i = \begin{cases} [\![b']\!]_G(j) & \text{if } b \ {}_i{<_j}\ b' \\ \star & \text{otherwise} \end{cases} \quad \text{for } 1 \leq i \leq \mathit{in}(b).$$

This function is well defined because there is at most one pair (b', j) such that $b \ {}_i{<_j}\ b'$ and because there is no infinite chain in G.

The computation of $[\![b]\!]_G(k)$ requires only the boxes of b^+: the value $[\![b]\!]_G(k)$ of a box b can be easily computed by substitutions in the set of equations corresponding to the values v_i of the boxes in b^+. This corresponds to the demand-driven evaluation of a box as done by following the functional dependencies of a patch, and as displayed for instance in Figure 1a.

Note that boxes in OM can be set to special "states" which will impact their evaluation. For instance, a box can be considered as a higher-order function and to produce its own functional definition as value if its state is "lambda". In this case,

$$[\![b]\!]_G(k) = \lambda x_{j_1} \dots x_{j_p}.[\![b]\!]_k(w_1, \dots, w_{\mathit{in}(b)}), \tag{1}$$

[2]Note that not all boxes are pure functions: in Figure 2 for instance, the boxes *nth-random* pick a random element in their respective input lists, and the boxes *repeat-n* collect the results of n successive calls to their first input connection.

where j_k is the index of the kth not connected input of b in G^t and

$$w_j = \begin{cases} [\![b']\!](k) & \text{if } b \ {}_j{<_k}\ b' \\ x_j & \text{otherwise} \end{cases}$$

$\rightarrow$ If input j is not connected to a box output, then w_j is a variable bound by the lambda abstraction.

Generations

Programs change incrementally during an OM session, and are evaluated on request to be partially updated. We call a *generation* an evaluation triggered by the user requiring the value of some arbitrary box in the visual program. This concept is formalized in the semantics by the state $G^t = (\mathsf{B}^t, \mathsf{E}^t, \mathsf{s}^t, \mathsf{t}^t)$ of a patch at a given generation t. The successive generations are identified by integers, starting from 0 and for the sake of the simplicity, we assume that the initial patch is empty:

$$G^0 = (\emptyset, \emptyset, \mathsf{s}, \mathsf{t}).$$

Accordingly, the completion of a user evaluation request invokes the transition $G^t \rightarrow G^{t+1}$.

Between two generations, the program G can change because some edges and boxes may have been added or deleted. Box values in OM visual programs can also be manually edited thanks to graphical editors, and set to a "locked" state in order to prevent reinitializations on future updates (a locked box behaves like a terminal box in the call graph).

Box States

We have seen in the previous paragraphs that the evaluations could actually be influenced by the *state* of the different boxes in G ("locked" state, "lambda" state, etc.). In order to give account for this specificity of the visual language, we introduce $\mathit{flag}^t : \mathsf{B}^t \rightarrow \{\square, \boxtimes, \boxed{e}, \boxed{\lambda}\}$, giving the state of the boxes in the patch G^t:

- $\square$ corresponds to a box with the standard behaviour,
- $\boxed{\lambda}$ corresponds to an abstracted ("lambda") box,
- $\boxtimes$ corresponds to a locked box,
- $\boxed{e}$ corresponds to a locked box whose output values have been manually edited.

We have outlined previously that the computation of the outputs of a box b depends only on the outputs of the boxes in b^+. The different possible states of a box makes the determination of this dependency set slightly less straightforward.

We call *ancestors* $\uparrow A$ of a set of boxes A in G^t the boxes that can be reached from the boxes of A going from inputs to outputs, without having passing through a locked box. Let $<_\square$denotes the restriction of the binary relation $<$ to the standard boxes: $b <_\square b'$ iff $b < b'$ and $\mathit{flag}^t(b) = \mathit{flag}^t(b') = \square$. Then: $\uparrow A = \{\, b \in \mathsf{B}^t \mid \exists b' \in A, b' <^*_\square b \,\}$, where $<^*_\square$ is the reflexive transitive closure of $<_\square$.

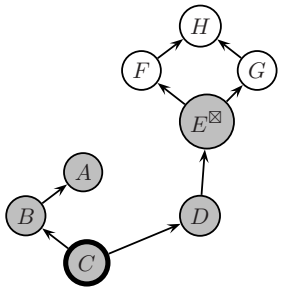

Figure 3: Call graph including a "locked" box.

Figure 3 is similar to Figure 1a with a box in state $\boxtimes$. The ancestors $\uparrow\{C\}$ of C are coloured in gray: F, G and H are excluded from this set because of the state $\boxtimes$ of E.

Staggered Evaluation

We describe the successive generations of a patch by a sequence of quadruples $(G^t, \mathit{flag}^t, e^t, r^t)_{t\in\mathbb{N}}$, each describing a generation, where the function $e^t(b,k)$ gives the edited value of the k-th output of every box $b \in \mathsf{B}^t$ such that $\mathit{flag}(b) = \boxed{e}$, and $r^t \in \mathsf{B}^t$ is the box on which the user requests the evaluation.

The complete semantics leads to a *staggered evaluation*: $[\![\cdot]\!]^t(\cdot) : \mathcal{B} \times \mathbb{N} \to \mathcal{V}$, where only the values of the boxes required to compute the outputs of r^t are updated:

$$[\![b]\!]^t(k) = \begin{cases} \star & \text{if } b \notin \mathsf{B}^t \\ e^t(b,k) & \text{if } \mathit{flag}^t(b) = \boxed{e} \\ [\![b]\!]^{t-1}(k) & \text{if } \mathit{flag}^t(b) = \boxtimes \\ [\![b]\!]_k(v_1, \ldots, v_{in(b)}) & \text{if } b \in \uparrow\{r^t\} \\ [\![b]\!]^{t-1}(k) & \text{otherwise} \end{cases}$$

with

$$v_i = \begin{cases} [\![b']\!]^t(j) & \text{if } b \ {}_i{<}_j \ b' \\ \star & \text{otherwise} \end{cases}$$

Note that we do not include the case $\mathit{flag}^t(b) = \boxed{\backslash}$ in the equation above, which remains as in (1).

2.2 Reactive Extension

The incremental creation of visual programs in OM involves frequent evaluation requests, visualization, data input and modifications. Despite these interactive aspects, programs execute with very limited connection to their external environment, and are triggered exclusively on user requests.

Introducing reactivity in the visual program execution model will enable a number additional behaviours such as the automatic update of downstream parts of a visual programs after changes or user actions, or the reaction to events sent by external applications or input devices.

A number of extensions are added to the semantic framework:

Event. An *event* is the subset of the boxes in G that may lead to an update. This notion abstracts the edition operations on G (*e.g.* user actions or modification of the data in a patch) as well as the response to *external events*.

Request. We call *request* a subset of the boxes of G on which the user requires an evaluation. Requesting an evaluation is not subsumed by the notion of event: the evaluation of a box leads to the evaluation of its ancestors, while the computations triggered by an event apply to its descendants.

This notion of request, which is the default way to trigger computations in OM, is still meaningful in the reactive context, because some parts of the patch may not be reactive (see below), and hence need an explicit request to be updated. It also provides a finer control in case of non-functional boxes (*e.g.* boxes with a mutable state).

Active Boxes. For the same reasons that motivates the locking of a box, it is not always desirable to update the values of a box in response to an arbitrary event. In addition it is important that the OM reactive extension be conservative and do not interfere with existing programs semantics and behaviour: reactivity must be optional and potentially applicable locally to any part of an existing program. Thus, we add an additional attribute to the boxes in the form of a predicate active^t specifying if a box must be sensitive to events.

Updated Semantics

In the reactive semantics, a *run* is a sequence of sextuples $(G^t, \mathit{flag}^t, e^t, \mathit{active}^t, R^t, E^t)_{t\in\mathbb{N}}$, where the first three components G^t, flag^t and e^t are as in the original semantics, active^t is a Boolean function on B^t; R^t and E^t are subsets of B^t representing the request and/or the event at the origin of the new generation t. They satisfy the two conditions:

- $\neg \mathit{active}^t(b) \Rightarrow (b \notin E^t)$: when a box is inactive, it cannot appear in an event (that is, if it appears in an event, it is active);
- $b \in E^t \Rightarrow (\mathit{flag}^t(b) = \boxed{e})$: when a box appears in an event E^t, it behaves like an edited box.

Inactive boxes stop the propagation of the updates. The set of boxes that must be evaluated as a consequence of an event A are the *descendants* of A ($\downarrow A$) defined as the boxes that can be reached from the boxes of A going from outputs to inputs, without passing through an inactive, a locked or an edited box, except those of A.

Let $<_a$ denotes the restriction of $<$ specified by:

$$b <_a b' \iff (b < b') \wedge \mathit{active}(b) \wedge \big(\mathit{flag}(b) = \square\big) \wedge \mathit{active}(b') \wedge \big(\mathit{flag}(b') = \square\big)$$

Then, the descendants of A are defined by: $\downarrow A = \{\, b \in \mathsf{B}^t \mid \exists\, b' \in \mathsf{B}^t, b'' \in A,\ b <_a^+ b' < b'' \,\}$, where $<_a^+$ is the transitive closure of $<_a$. (This definition departs slightly from the definition of the dual notion $\uparrow A$, because the elements of A are not themselves in $\downarrow A$ – they are edited).

The complete reactive semantics is now defined by a generation-indexed sequence of functions:

$$(\!|\cdot|\!)^t(\cdot) \quad : \quad \mathcal{B} \times \mathbb{N} \to \mathcal{V}$$

$$(\!|b|\!)^t(k) \quad = \quad \begin{cases} \star & \text{if } b \notin \mathsf{B}^t \\ e^t(b,k) & \text{if } \mathit{flag}^t(b) = \boxed{e} \\ (\!|b|\!)^{t-1}(k) & \text{if } \mathit{flag}^t(b) = \boxtimes \\ u & \text{if } \mathit{flag}^t(b) = \boxed{\lambda} \\ [\![b]\!]_k(v_1, \ldots, v_{in(b)}) & \text{if } b \in \big(\uparrow R^t \cup \downarrow E^t\big) \\ (\!|b|\!)^{t-1}(k) & \text{otherwise} \end{cases}$$

with

$$v_i = \begin{cases} (\!|b'|\!)^t(j) & \text{if } b\ {}_i{<_j}\ b' \\ \star & \text{otherwise} \end{cases}$$

and

$$u = \lambda x_{j_1} \ldots x_{j_p}.[\![b]\!]_k(w_1, \ldots, w_{in(b)}),$$

where j_k is the index of the kth not connected input of b in G^t and

$$w_j = \begin{cases} (\!|b'|\!)^t(k) & \text{if } b\ {}_j{<_k}\ b' \\ x_j & \text{otherwise} \end{cases}$$

3 Implementation

The semantics presented in the previous section is independent from the conditions of a transition from a generation to the next one. The implementation in OM defines a number of such conditions, as well as adapted solutions to efficiently determine and calculate ($\uparrow R^t \cup \downarrow E^t$) for each new generation G^t.

We mentioned earlier that a user request on a box b triggers a new generation. As a consequence, there can be only one box at a time in R^t. We choose to handle events asynchronously, that is, one at a time, so that requests and evaluations are handled sequentially in a single thread. As a consequence, at each generation, we have to compute either $\uparrow b$ or $\downarrow b$ for a given box b in the patch.

The standard evaluation process in OM associates with each box b a compiled Lisp function that implements the evaluation spanned by a request on b. The implementation of the reactive model on top of this process can be relatively straightforward, provided we add a number of features to the visual programs.

Active Boxes. We add a new attribute of the OM boxes determining their reactive status. Reactivity can be switched on and off by the user with a keyboard short-cut (x). Reactive boxes are highlighted by red frames in the visual programs.

Registration of descendants. While editing the patch (adding/removing boxes and connections), every box registers its downstream connected boxes, i.e. the direct descendants which use the value of this box as an argument, and are likely to propagate updates. (In the current/default OM demand-driven implementation, boxes are only linked to their direct ancestors through input connections.)

Events. The events defined in the system are actions or changes in a visual program likely to disable the functional consistency of a graph (i.e., to require an update on some of the boxes in G^t). Box or input value modifications can occur for instance with keyboard actions (e.g. for numerical or text inputs) or with actions in the graphical editors (for data structure boxes). They can also occur with the evaluation of a box following a request by the user, or with an external event caught by the system.

Therefore, events have no concrete implementation: they simply correspond to the beginning of an update starting at a specific place in the visual program.

Notifications and Updates

Events are propagated to the box descendants through a *notification* mechanism. Below is the *signal-event* function responsible for the appearance of an event:[3]

```
(defmethod signal-event ((self OMReactiveBox))
  (when (active self)
    (setf (state-lock self) t)
    (OMR-Notify self)
    (setf (state-lock self) nil)
  ))
```

Note that the box is "virtually" locked during the process (using the attribute *state-lock*). This will avoid the box at the origin of the event (and eventually its ancestors) to be evaluated as part of the update process. The evaluation method *omng-box-value* is slightly modified accordingly:

```
(defmethod omNG-box-value ((self OMReactiveBox) &optional (numout 0))
 (if (state-lock self)
    (current-box-value self numout)
   (call-next-method) ;;; standard OM evaluation
 ))
```

Anything considered as an event (user edit, value change, etc.) should then simply invoke *signal-event*. Currently, this function is called for instance:

- When the user evaluates a box (request);
- When a "value box" has been edited;
- When a box input has been edited (and after an update of the box value);
- When a modification is made by the user in an object editor and is validated/retuned to the box.

A notification starts at the event ($b \in E$) and is propagated to its descendants in $\downarrow\{b\}$ by a depth-first traversal of the graph following the registered and active output connections. Every reactive descendant in turn propagates the notification:

```
(defmethod OMR-Notify ((self OMReactiveBox))
  (unless (notified-tag self)
    (setf (notified-tag self) t)
    (let ((listeners (remove-if-not 'active (listeners self))))
      (if (and (active self) listeners)
        (mapcar 'omr-notify listeners)  ;; propagate
       (omNG-box-value self) ;; evaluate
  ))))
```

When a terminal box b' is reached (that is, a box with no descendent), the standard evaluation of b' is triggered following the OM demand-driven model (*omNG-box-value*), hence respecting the existing scheduling strategy and semantics for program executions.

Notified boxes in $\downarrow\{b\}$ are marked during the propagation (using the *notified-tag* flag), in order to avoid multiple visits to the same sections of the graph. When the evaluation is done, a global flag clean-up is done (this process hooks on an existing clean-up occurring for other box flags).

[3]The code listings presented in this report are often simplified for easier reading. The complete code is available in *reactif-lib* distribution.

Note that the *notified-tag* flag can also be used as an additional "lock test" avoiding the evaluation of boxes that are not edited and/or not in the reactive chain (that is, $\uparrow\downarrow\{b\} - \downarrow\{b\}$, e.g. boxes B and A in Figure 1b). Such implementation choice can make a difference, for instance when random or other impure components yield a different results at each evaluation (no matter if inputs have changed or not). In our current implementation, this decision is left to the user who can decide to lock relevant boxes and limit the scope of the graph updates and computations.

Discussion

This relatively simple implementation provides all the required features for first-order reactive visual programs. It does not overload programming tasks and is completely transparent and conservative with respect to the OM language design.

The choice of ordering events and handle them asynchronously in a single thread simplifies a great deal the process and is acceptable in our context and applications. Every incoming event triggers an update and the next one is processed when the previous update terminates. This strict ordering prevents any possible conflict, as user edits are also taken into account after any previous event processing is done.

4 Applications

Consistency of the Graph

Changes in a reactive visual program trigger re-computation of descendants and thereby maintain the consistency of the functional graphs. For instance, a change in Figure 2 which would require the explicit evaluation of *RESULT* to make the graph consistent (e.g. modification of th box value $'8'$, or some editing in the the score editor *PITCHES SEQ*) now automatically updates the patch. This reactive mechanism impacts the OM visual programming practices and can facilitate user experimentations with input values.

Interactive Widgets

This mechanism can make particular sense with a number of special components of the OM visual programs such as the "interface boxes": buttons, list/menu selection, sliders, etc. The insertion of a notification after the standard activation procedure of an interactive components makes it very handy to deal with the experimentation of multiple choices or value ranges in the visual programs. Figure 4 shows an example of a melody calculated from a curve, which values are scaled in a pitch range determined by two sliders. The activation of a reactive chain between the sliders and the score box makes this range controllable interactively with immediate feedback.

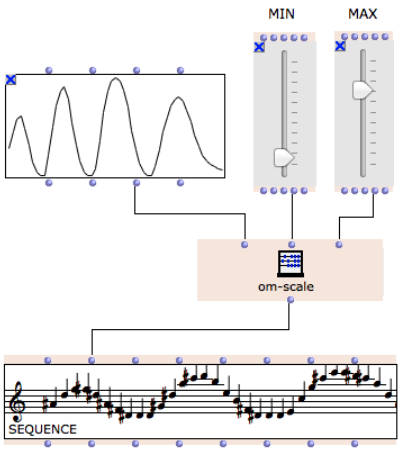

Figure 4: Interactive experiments with input values in a reactive OM visual program using slider boxes. The two slider boxes (*MIN*, *MAX*), *om-scale* and *SEQUENCE* are reactive.

It is important to note that this behaviour is not desirable in every cases and all the time, hence the local and optional aspects of the reactive mechanism. In an experimental environment like OM, visual program construction is an incremental process carried out by trials and errors. Sometimes boxes can be temporarily connected in a certain way by mistake, or for different purposes than immediate evaluation (which could moreover generate chains of errors). Evaluations also sometimes last large amounts of time (e.g. in complex optimisation problems, large data structure processing, or high-precision signal analysis or synthesis), and the computation of the visual program is then usually run once for all at the end of its construction only (the previous example could become problematic in case of heavy computational processes depending on the reactive slider values).

Communication with External Systems

The reactive framework also renews the possibilities for the communication of the computer-aided composition environment with other applications, and more generally with its external context. Networked inter-application communication is supported in the system OSC encoding (Wright, 2005) over UDP and available at the level of the visual programs through the *OSC-receive* and *OSC-send* boxes. When activated, *OSC-receive* runs a server thread receiving incoming UDP messages on a specified port. A reactive behaviour (notification) is also added to this box, updating its value and triggering a notification when a packet is received. Visual programs containing an active *OSC-receive* hence become reactive to incoming messages or events from any application, input device or instrument.[4]

Data Collection

The messages received through receive/reactive boxes are in principle "instantaneous": in order to integrate compositional processes, they generally need to be gathered in more complex data sets and mapped to time structures. A specific box called *coll* has been created for this purpose, which instantiates a local storage for dynamic data collection.

Coll boxes have three inputs and constitute a special case in the notification mechanism. Upon reception of a notification on it's first input (labelled *data-in*), the connected box at the source of the notification is evaluated as a terminal box and the resulting value is added into a list that constitutes the *coll* "memory". On its second input (labelled *push*), *coll* behaves as a standard reactive box, propagates the notification through its descendants, and returns the current contents of its memory (a list of all previous data collected through *data-in*) upon evaluation. Finally, a notification to the third input (labelled *init*) reinitializes the memory and stops the propagation.

Coll behaves as an intermediate step in the notification/evaluation mechanism where notification temporarily stops and inputs are evaluated. The box is then locked and (if requested so depending on the incoming notification) propagates on its own.

Coll can be associated to OSC or MIDI receive boxes to develop reactive programs in OM that structure and process live inputs from real-time systems. Figure 5 shows an example where a Max program (Puckette, 1991) sends a "note" message via UDP every time the user clicks on one of the piano keyboard widget. The OM program containing a running *OSC-receive* (at the right) collects the pitch information in this message into the *coll* box. At every received message the *push* input also receives a notification (the two *coll* inputs are connected), and therefore triggers an update of the downstream part of the OM program. The growing contents of the memory is converted into a data structure that can be further processed in the visual programming environment.

Handling Time and Groups in Data Collection

It is frequent in communication frameworks that several messages must be gathered in a same group and/or be considered simultaneous (if the messages are notes as in the previous example, this idea would correspond to the concept of *chord*). However it is very unlikely that two events belonging to the same group (or notes in the same chord) be sent and received at the exact same time. And even so, they would not be handled simultaneously by the system. Time-tagging events is a solution to deal with this

[4]The same principle applies for MIDI messages and musical instruments or devices communicating with this protocol.

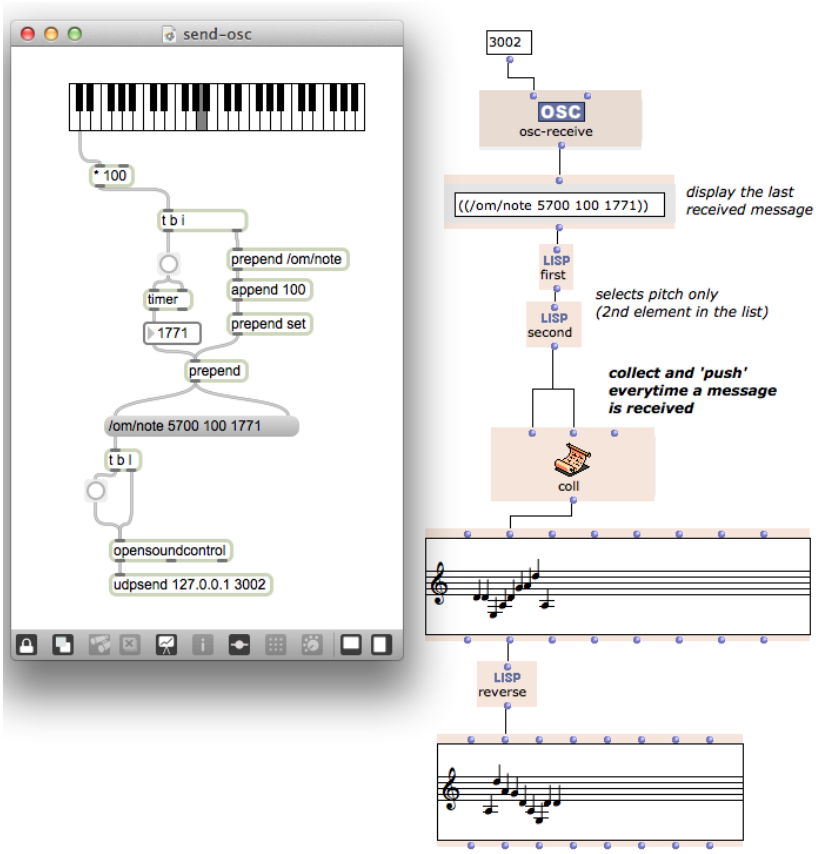

Figure 5: A basic process of reception and collection of incoming data from a Max real-time program (window on the left) in OM.

situations – the OSC protocol also groups simultaneous messages in time-tagged *bundles*, see (Schmeder and Freed, 2008) – but all events are not necessarily time tagged.

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 collected data according to specified intervals. The *group* box implements this behaviour: it has one input collecting data (similar to the *coll* input) and a second input setting the time interval.

The *timed-coll* object extends *coll* with the *group* behaviour and gathers the items collected during a given time interval into separate data chunks.

Depending on the timing of incoming events, the result in Figure 5 using *timed-coll* would be a list of lists of pitches, corresponding to a sequence of chords in the score.

Even-Driven Conception of the Visual Programs

Reactive applications such as the one in Figure 5 tend to orient the design and general conception of the visual programs towards a data-driven paradigm, where the propagation of notifications needs to be controlled as evaluation is (the *coll* mechanism, for instance, can be seen as the event-driven counterpart to the *collect* primitive in standard Lisp or OM loops). Typically, one quickly needs to select for instance between the data collection, push or init commands in the *coll* by sending different messages, instead of instantiating separate OSC sever connections for every command.

Several tools inspired by reactive data-driven systems like Max are being added for this purpose to the OM reactive framework, such as *route*, a utility that filters notifications depending on a set of tests performed on its input. *Route* is also a particular case in the update mechanism in the sense that it requires input evaluation before to chose where to propagate the notification:

```
(defmethod OMR-Notify ((self ReactiveRouteBox))
  (unless (push-tag self)
    (setf (push-tag self) t)
    (omNG-box-value self)  ;;; => CALLS "ROUTE" AS A NORMAL FUNCTION (EVALUATES INPUTS BEFORE)
    (setf (state-lock self) t)
    (when (active self)
       (let* ((routed-outputs (loop for i = 0 then (+ i 1)
                                    for v in (value self)
                                    when v collect i))    ;;; ONLY OUTPUTS WITH A VALUE WILL PROPAGATE
              (listeners (boxes-connected-to-outputs (listeners self) routed-outputs)))
      (mapcar 'omr-notify listeners)))
    (setf (state-lock self) nil)
 ))

;;; METHOD SETTING THE VALUE OF THE BOX
;;; COLLECTS THE DATA IF TESTS SUCCEEDS OR NIL IF IT FAILS
(defmethod! route (data &rest tests)
  (cons data
     (mapcar
        #'(lambda (test) (when (test-match data test) data))
        tests)))
  ))

(defun test-match (data test)
  (if (functionp test)
    (funcall test data)
   (equal test data)))
```

Figure 6 shows an extended version of the example in Figure 5 using *route-osc*, a version of *route* specialized for processing OSC messages. *Route-OSC* tests the message "address" and propagates the rest of the data. The different messages now coming from the Max interface control different behaviours of the OM program, differentiating the phases of initialization, data collection, and activation of the downstream processing for the collected data.

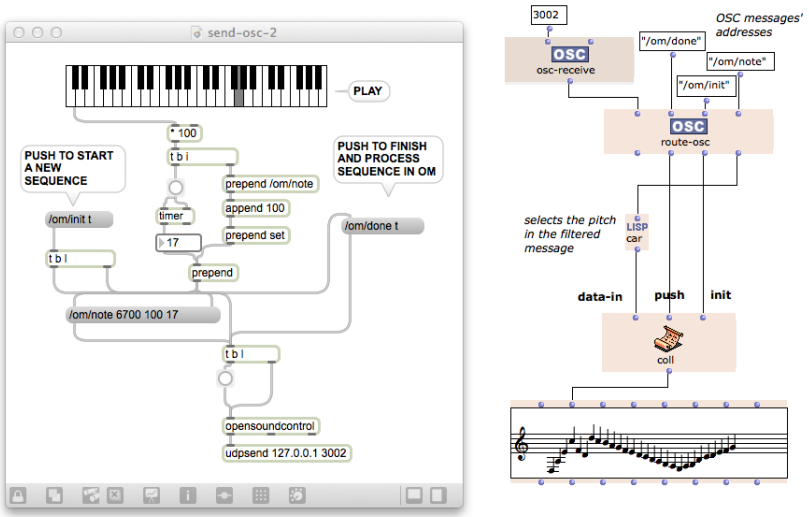

Figure 6: Routing notifications using incoming OSC message addresses.

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

5 Conclusion

The reactive extension of OM introduces a new conception of the computer-aided composition framework and of its relations to the external context. Compositional processes are built and run interactively, but now also "live" on their own and can be driven by external processes and interactions. The fundamental "off-line" characteristic of computer-aided composition systems is preserved and interleaved in larger-scale/reactive temporal contexts.

The main questions raised by this hybrid paradigm are related to time structures and computation.* In Figure 5 for instance, instantaneous data received from a real-time environment had to be converted into timed musical data structures. In this case the mapping to timed structures can be an arbitrary process programmed in OM (as done in a primitive way by the *timed-coll* object), or embedded in the communication protocol (as with an embedded time-tagging protocol).*

The generated musical structures can in turn be re-injected in the "real time" flow, by user request or as a response to a notification in the reactive system, and either via networking (e.g. with *OSC-send*) or by direct synthesis or rendering in OM (e.g. via embedded audio/MIDI players).* Such reactive loop involving computer-aided composition processes (where real-time processing leaves space for advanced formal/compositional computations) opens interesting perspectives for enhanced musical expressiveness in interactive situations.

[*] ⇒ These different points will be addressed in further stages of the EFFICACe project:

- WP 1.3: Audio/MIDI systems
- WP 2.1: Time structures and scheduling
- WP 2.2: Communication protocols

The reactive extension of OM is packaged as a dynamic library (*reactif-lib*) loadable on top of the standard OM version ($\geq$ 6.7). It is included as a native feature in OM 6.9.

References

Assayag, G., Bloch, G., Chemillier, M., Cont, A., and Dubnov, S. (2006). Omax Brothers: A Dynamic Topology of Agents for Improvization Learning. In *Workshop on Audio and Music Computing for Multimedia, ACM MultiMedia*, Santa Barbara, CA, USA.

Bresson, J., Agon, C., and Assayag, G. (2009). Visual Lisp/CLOS Programming in OpenMusic. *Higher-Order and Symbolic Computation*, 22(1).

Bresson, J. and Giavitto, J.-L. (2014). A Reactive Extension of the OpenMusic Visual Programming Language. *Journal of Visual Languages and Computing*. In Press.

Harel, D. and Pnueli, A. (1985). On the Development of Reactive Systems. In *Logics and Models of Concurrent Systems*. Springer Verlag.

Nika, J. and Chemillier, M. (2012). ImproteK, integrating harmonic controls into improvisation in the filiation of OMax. In *Proceedings of the International Computer Music Conference*, Ljubljana, Slovenia.

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

Schmeder, A. and Freed, A. (2008). Implementation and Applications of Open Sound Control Timestamps. In *Proceedings of the International Computer Music Conference*, Belfast, Ireland.

Wang, G. and Cook, P. R. (2004). On-the-fly Programming: Using Code as an Expressive Musical Instrument. In *New Interfaces for Musical Expression (NIME'04)*, Hamamatsu, Japan.

Wright, M. (2005). Open Sound Control: an enabling technology for musical networking. *Organised Sound*, 10(3).