A Reactive Extension of the OpenMusic Visual Programming Language

Jean Bresson . Jean-Louis Giavitto
UMR 9912 "Sciences and Technologies of Music and Sound"
IRCAM – CNRS – UPMC – INRIA
1, place Igor Stravinsky 75004, Paris, France

This is an updated and slighly corrected version of the article *A Reactive Extension of the OpenMusic Visual Programming Language* published in *Journal of Visual Languages and Computing* (2014) – `http://dx.doi.org/10.1016/j.jvlc.2014.03.003`**.**

This research is funded by the French National Research Agency projects with reference ANR-12-CORD-0009 and ANR-13-JS02-0004-01.

Abstract

Objectives: OpenMusic (OM) is a domain-specific visual programming language designed for computer-aided music composition. This language based on Common Lisp allows composers to develop functional processes generating or transforming musical data, and to execute them locally by demand-driven evaluations. As most historical computer-aided composition environments, OM relies on a transformational declarative paradigm, which is hard to conciliate with reactive data-flow (an evaluation scheme more adequate to the development of interactive systems). We propose to link these two evaluation paradigms in a consistent visual programming framework.
Methods: We establish a denotational semantics of the visual language, which gives account for its demand-driven evaluation mechanism and the incremental construction of programs. We then extend this semantics to enable reactive computations in the functional graphs.
Results: The resulting language merges data-driven executions with the existing demand-driven mechanism. A conservative implementation is proposed.
Conclusions: We show that the incremental construction of programs and their data-driven and demand-driven evaluations can be smoothly integrated in the visual programming workflow. This integration allows for the propagation of changes in the programs, and the evaluation of graphically-designed functional expressions as a response to external events, a first step in bridging the gap between computer-assisted composition environments and real-time musical systems.

1 Introduction

OpenMusic (OM) is a domain-specific visual programming language designed for music composers, considered as a reference in the *computer-aided composition* domain [6]. It is used to build compositional processes that generate or transform musical data. The goal of this paper is to specify the semantics of an OM program and of its incremental construction from the specific perspective of its visual programming model, and to propose a conservative extension of this semantics adding reactivity to the existing paradigm. This extension integrates well with the incremental building of compositional processes. The resulting system combines a demand-driven together with a data-driven evaluation mechanism, which constitutes a first step in bridging the gap between "out-of-time" and real-time programming environments in computer music.

This paper is organised as follows. The next section provides a background in computer music environments and details the distinction between out-of-time (compositional) and real-time (performative) environments. Then, the OpenMusic visual language and system workflow are presented (Section 2). We propose a semantics describing this high-level programming paradigm (incremental program construction and evaluation) from a functional point of view (Section 3), and we extend this semantics toward a reactive model (Section 4). An implementation of this semantics is then presented, which merges in the existing visual programming structures of the language (Section 5). Finally, we review and compare our work to state-of-the-art research and projects in the fields of visual programming and functional-reactive programming (Section 6).

1.1 Programming Languages for Computer Music

The practice of music in the 20th Century has been strongly influenced by computer science. In particular, programming languages have provided efficient means for contemporary music composers to overcome established musical systems and develop new formal and artistic approaches [35, 17]. In this context, several visual programming languages have been developed and adopted by musicians. Visual languages make programming and the access to computer resources more productive and useful to certain user communities, willing to design complex processes but not necessarily attracted to or skilled in traditional textual programming [26, 30]. They are supposed to ease programming activities (*e.g.* limiting syntactic errors), but also contribute to a more interactive relation between the user and the programs. If this idea can be argued [45, 46, 25], it has been clearly validated in the case of computer-aided composition, where this interaction improves a great deal the creative potential of the computer tools [3].

Two main categories of programming languages exist today in computer music: the declarative computer-aided composition languages used for the off-line modelling and generation of musical structures, and the real-time/reactive data-flow languages used for the design of interactive systems.

1.2 Computer-Aided Composition

The goal of *computer-aided composition* systems is to develop concepts and tools dedicated to the processing and generation of musical data (scores, sounds, harmonic, rhythmic and many other kinds of compositional structures) using programming and computational approaches [4].[1]

Computer-aided composition environments are often qualified as "out-of-time" systems, since they make a strong distinction between the time implicitly present in the musical structures and the time that flows during the computation of these structures. The musical time is therefore handled as a "spatial" dimension, which allows for the formalization and generation of sophisticated musical structures with an important abstraction power.

The Patchwork visual language was a pioneering work introducing the use of visual programming for computer-aided music composition [32]. This language provided a diagram-based interface for Lisp program-

[1] A simple example of a task in this area could be, for instance, the generation of all orderings of a given set of notes such that the interval between two successive notes belongs to a given set and appears exactly one time in the sequence: this problem, called the *all-interval series*, is included in the *CSPlib* benchmark for constraint satisfaction systems — see `http://www.csplib.org/`.

ming (hence, related to a functional/declarative paradigm) enhanced with musical functions, data structures and graphical editors for music notation. The main systems for computer-aided composition today (OpenMusic and PWGL [33]) derive from this language and follow similar concepts. In particular, they both rely on the interactive by-need — or *demand-driven* — computation of visual expressions.

1.3 Real-Time Musical Languages

The distinction between the time of the musical data and the time of their computation in computer-aided composition systems contrasts with the assumptions of real-time musical programming languages, where these two aspects of time are practically merged. Designed for performance applications, real-time musical programs react continuously to input streams (audio signals, events, metronomes, etc.). They articulate locally synchronous temporal flows (audio signals with hard real-time constraints) within globally asynchronous event sequences (discrete timed actions that result, for instance, from user interactions).

Max [37], a visual language based on reactive data-flow, became the standard for the design and specification of real-time event and signal processing in music.[2] The reactive data-driven computation makes this environment particularly suited and oriented toward interactive applications and live performance, but not to compute and represent arbitrarily complex compositional structures.

1.4 Bridging the Gap

The division highlighted in the previous categorization (computer-aided composition/real-time systems) actually reflects the distinction between *transformational* and *reactive* systems [27]. It stems from different approaches and needs in the use of 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 interaction during concerts.

This opposition between "compositional" and "performative" aspects of computer music environments has perpetuated along the years, be it from artistic or technical points of view, and musicians presently perceive this fragmented framework as a frustrating limitation at developing creative musical projects. Their convergence would allow for a tighter coupling between composition and performance, and drastically improve for the development of musical systems, for composers but also in other application fields like interactive music, video games, real-time interactive sonification, reactive sound design or multimedia installations. This coupling between composition and performance already appears in several recent trends in computer music, such as *live-coding*, where the programming activity is embedded in concert performances [44], or in computer accompaniment or automatic improvisation systems, where complex and structured musical parts are generated from live musical inputs and played synchronously with them [5]. We consider that these examples are premises for a radical conceptual evolution in computer music, mixing the compositional and interactive approaches.

As a first step in this direction, this paper presents a reactive extension of the OpenMusic computer-aided composition environment. Our objective is not to process audio signals or events in real time with this language, and thereby substitute existing dedicated systems, 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 sufficient to meet with performance constraints.

2 The OM Visual Programming Language

OpenMusic[3] is a full-featured visual programming environment based on Common Lisp, which combines local state with interactive by-need (demand-driven) computation of visual expressions [2]. This environment provides a graphical interface to the main constructions and features of the Common Lisp language

[2] PureData [38] is another popular open-source language inspired from Max.

[3] See `http://repmus.ircam.fr/openmusic/`

(abstraction, higher-order function, iteration, recursion, object-oriented programming, etc.) [10], and enriches the Common Lisp data structures with specific musical objects and dedicated functions. Functional expressions and structures can be built and organised in project-scale *workspaces*, mixing visual programs and Lisp programs: a tight integration with the underlying programming language is maintained in order to benefit from the expressivity and compactness of textual programming when needed, making the programs and libraries developed scalable and modular [21]. Projects and third-party libraries allow programming to be extended to a wide variety of purposes and applications [11].

The OM environment supports an original interactive and iterative workflow mixing functional programming, execution of the program and rendering/editing of the input or output data, which allows for composers to easily and progressively build and get into the complexity of their computational/compositional models.

With hundreds of referenced users worldwide, OM has contributed to numerous important compositional projects and pieces from the contemporary repertoire. *The OM Composers Books* [3] provide a documented overview of such musical applications.

The rest of this section is devoted to the description of OM programming systems and its workflow. Previous publications have described the functional programming features of OM in more detail [10]. We focus here on the features that are specific of the incremental workflow and the notions needed to understand the semantics presented in the next section.

2.1 Structure and Syntactic Aspects of OM Visual Programs

Visual programs in OM (also called *patches*) are directed acyclic graphs representing functional expressions using *boxes* and *connections*. A box refers to a function and represents the corresponding function call when it is embedded in a visual program. It has a number of inlet and outlet ports: inlets, at the top of the box, represent the function call arguments, and outlets, at the bottom, represent the results of the call. A connection between an output of a box a and an input of another box b means that one of b's arguments requires the value of a. Therefore, the connections in OM patches represent at the same time the functional compositions and the flow of data in these programs.[4] Fig. 1 shows an OM patch. We will use it to analyse and describe the main language features discussed in this paper.

The function associated to a box in OM can be either a simple Lisp function or a specific one defined with additional (graphical) attributes[5]. Beware that such function may be impure, so two evaluations of the same box in a patch with the same arguments may return different results. A box has as many inputs as the associated function arguments (with the possibility to add optional/keyword inputs) and outputs corresponding to the possible multiple values produced by the box's function (according to the Common Lisp notion of multiple-valued functions).

Some boxes correspond to the construction of values. The *value boxes* correspond to raw, constant values (numbers, lists, strings, symbols, etc.) and allow the input of such data in the programs (*e.g.*, boxes **N**, **O**, **P**, **Q**, **R**, **S** in Fig. 1). The *factory boxes* refer to classes and produce instances of these classes, that is, calls to the Common Lisp *make-instance* (in Fig. 1, boxes **A**, **B**, **C** and **D** are factory boxes). The input/output ports of these boxes represent the public attributes (or *slots*) of the referred class: the inputs represent the initialization parameters of the constructor and the outputs are accessors to the corresponding slot values[6]. Factory boxes are also generally associated with graphical editors allowing for the visualization and modification of their internal value (that is, their last computed instance). Note that we will make no formal distinction here between the construction of values (constructors) and (multiple-valued) function calls.

Thanks to the reflexive features of Common Lisp, any function or class defined in Lisp can be instantiated as a box in an OM patch, without any additional programming. Note that the boxes in a visual program can also refer to other visual programs considered as functions (*abstractions*) in the top-level graph.

[4]The acyclic nature of the graph does not prevent the implementation of recursive processes, which are defined by including a box referring to itself in its own definition [10].

[5]OM functions are extensions (subclasses) of the CLOS *generic-function* metaclass [23].

[6]The first input/output port of the factory boxes is a special one, used respectively as a copy constructor and as a direct access to the instance value.

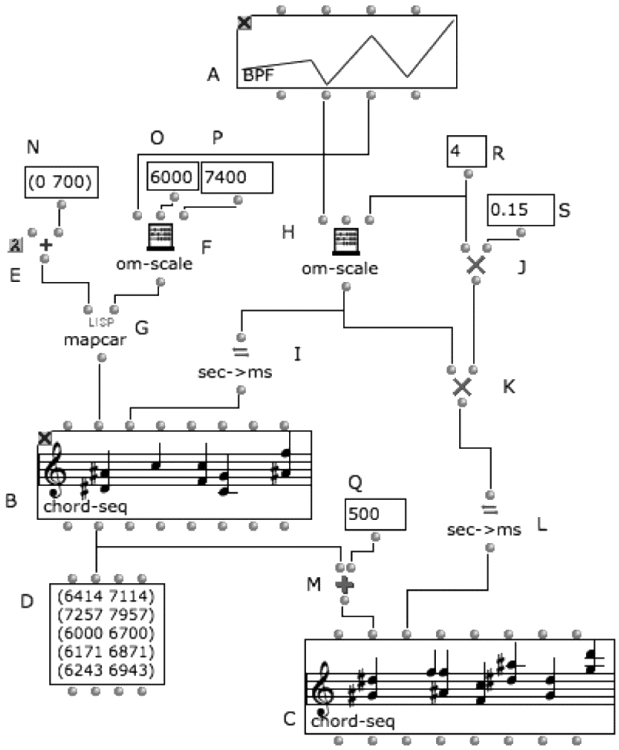

FIGURE 1: A visual program in OM (the box labels **A** through **S** on the figure have been added for easier referencing in the text). Boxes **B** and **C** are score objects whose pitches (in "midicent" units, that is, 1/100 half-tones) and onsets (in milliseconds) are computed starting from the coordinates of the break-points of a piecewise linear curve (defined in box **A**). The *om-scale* function boxes (**F** and **H**) rescale these coordinates to adequate time and pitch value ranges. *Mapcar* (box **G**) is a Lisp function used to apply + to the pitch list. The *sec→ms* functions (boxes **I** and **L**) convert lists of values from seconds to milliseconds. All arithmetic boxes (+, ×) can be applied to scalar values or to lists of values.

2.2 Incremental Programming Workflow

Programming in OM amounts to creating functional expressions using boxes and connections, including possible interrelations and dependencies with other visual programming components or Lisp code. The standard workflow of an OM user/programmer is the incremental building and manipulation of functional graphs, interleaved with data inputs and partial evaluations.

Evaluations are *demand-driven* and propagate following the input connections of the boxes. When *evaluating* a box, the user therefore explicitly requires the re-evaluation of a specific part of the graph. Generally, the result is a musical structure (*e.g.* a chord, a score, a sequence of MIDI events, etc.) but it can also be any kind of data directed to the standard system output or to external streams. It can then be "played" (when relevant), stored, or copied to other parts of the patch, or of the current workspace. We call this "top-level" user-driven evaluation of a box and of the corresponding part of the graph, a *generation*. This notion will be formalized in the semantics of the language in Section 3.

Unlike most functional systems, an OM program does not necessarily have a single and specific output (or *sink*), but it can be evaluated at any time and at any box of the graph. Several graphs can actually coexist, within the same patch or spread through the workspace, and contribute to the user's compositional experiments. Evaluations do not occur at a very high rate (as compared to the signal or control rates involved in reactive and signal processing systems) and can last an undetermined time to be completed — a characteristics of the "off-line" computer-aided composition approach.

Visual programs in OM are therefore not only meant to process or generate data: they actually represent musical structures through generative processes and constitute operative representations of "compositional models" [34] (hence the strong *declarative* emphasis of the language). The environment is considered as a programming/compositional "scratchpad" where musical data are interconnected by functional relations, defining a network of constraints linking the compositional material.

2.3 Box States

Programming in OM is a slightly more a complex process than simply evaluating graphically-designed functional expressions. Composers build and evaluate the programs incrementally, generating data from specific parts of the graph. They regularly edit, modify, store, duplicate them outside the functional context before running further evaluations. Computations are not performed in one single step and a straightforward functional evaluation strategy is not sufficient to support these interactions with the environment. The notion of box *state* supports additional related behaviours in the visual language.

When a box is *locked* (see for instance the boxes **A** and **B** in Fig. 1 and the little ⊠ icon on their top-left corner), the current box value (that is, result of the last evaluation) is stored and reused in the subsequent evaluations. There are several motivations for this mechanism:

- It avoids costly re-evaluations (*e.g.* the synthesis of a sound may take up to several minutes);
- It fits a common compositional workflow, computing data as initial material, which is then processed in subsequent phases of the work;
- In factory boxes, the user can access and modify the stored/last computed values using a graphical editor: in this case, edited data shall not be affected any more by evaluations and must be used as such in subsequent evaluations (see for instance the boxes **A** and **B** in Fig. 1, which have been edited manually after their first evaluation and disconnected from upstream initialization processes).

Another related behaviour (corresponding to an alternative state of the boxes) is called *eval-once*. In this state, the boxes evaluate once during a generation and then temporarily keep the same value for subsequent calls in this same generation. This behaviour is relevant in the case of multiple calls to the same box, either for efficiency issues or when indeterminacy or side effects may change the results between two different calls (*e.g.* a box that generates a random number at each call). We will however not consider this particular behaviour in the present paper, for the sake of brevity and because it will not affect our extension to reactive programming.

2.4 Higher-order Functions

Another state of the OM boxes allows us to use and manipulate higher-order functions. Boxes can be flagged as 'λ' (see for instance box **E** tagged with the ⍰ icon in Fig. 1), making them return a lambda expression on each of their outputs. The parameters of these lambda expressions are the "free" (unconnected) box inputs. For instance, box **E** (in state λ) returns $\lambda x.x + (0\ 700)$ on its first (and only) output. Therefore, user-defined functions can be referred to as a value to be used by other boxes.

Note that special *input* and *output* boxes can also be used in OM visual programs. Connected to other boxes, they allow the inputs of these boxes to be turned into program variables, or to return values out from these programs. Thereby, patches themselves can constitute function definitions likely to be used in other patches, as special boxes called *abstractions*.

3 A Formal Semantics for OM

The semantics we present is focused on the evaluation mechanism at the patch level, and on the incremental construction of programs. So, we assume that the Lisp function associated with a box is a predefined constant whose semantics is defined elsewhere, as in the handling of predefined values in an applied lambda-calculus.

The semantics of multiple patches and the binding of a user-defined box to a patch, is out of the scope of this work.

3.1 Visual Programs

OM visual programs mostly rely on the concept of *box*. Boxes are the nodes of a functional graph and denote function applications, class instantiations, or constant values. We formalize the notion of patch as follows. Let $\mathcal{B}$ be the set of box identifiers (*e.g.* the set of possible nodes in a OM graph) and $\mathcal{E}$ the set of edge identifiers. Variables $b, b', b_1 \dots$ range over $\mathcal{B}$ and variables $e, e', e_1 \dots$ range over $\mathcal{E}$.

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, that is, they satisfy the four conditions:

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).

Because s and t are total functions, conditions (1) and (2) ensure that an edge connects the output of a box to the input of another (no pending edges). Condition (3) ensures that an input of a box is connected at most to one output of another box, and condition (4) ensures that the graph is acyclic.

The strict partial order $\prec_G$ formalizes the *functional dependencies induced by* G. Below, we drop the subscripts G if they can be recovered from the context. We write $b \ {}_k{<}_{k'} \ b'$ iff there exists an edge e such that $\mathsf{s}(e) = (b', k')$ and $\mathsf{t}(e) = (b, k)$. For a given b and a given k, there exists at most one pair (b', k') such that $b \ {}_k{<}_{k'} \ b'$ because $(b', k') \in \mathsf{s} \circ \mathsf{t}^{-1}\{(b, k)\}$ and t is injective.

A *chain* in G is a sequence $(b_1, b_2, \dots)$ of boxes of G such that $b_i \prec b_{i+1}$. Note that 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 $b^- = \{b' \mid b' \preceq b\}$.

3.2 Evaluation

Let $\mathcal{V}$ be the set of values handled in OM: we assume $\mathcal{V}$ to be a domain [36] with least element $\bot$. In addition, we suppose a distinguished element $\star$ of $\mathcal{V}$ used to represent a default value: $\star$ is incomparable with any other element of $\mathcal{V}$ except $\bot$.

For each box $b \in \mathcal{B}$, there are $\mathit{out}(b)$ associated functions

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

giving the semantics of a box. Theses functions are assumed to be continuous in the sense of domain theory [36]. 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

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

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. And it is continuous because it is a composition of continuous functions. Note that the computation of $[\![b]\!]_G(k)$ requires only the boxes of b^+.

Example As an example, we compute the value of $[\![\mathbf{K}]\!]_G(1)$, where G is the patch from Fig. 1. Note that this evaluation may be caused either by an explicit user request, or because it is required by another box. The restriction of G to the nodes in $\mathbf{K}^+$ is displayed in Fig. 2. The semantics of the boxes is given as follows:

- $[\![\mathbf{K}]\!]_1 = [\![\mathbf{J}]\!]_1 = \textit{om-}*$, an extended polymorphic version of the multiplication operator.
- **H** refers to the function *om-scale*. Note that the second input of this box in Fig. 1 is not connected to any other box.
- **R** and **S** are simple value boxes with no inputs.
- **A** is a factory box referring to the class *BPF* describing a 2D piecewise linear curve. Only the second and third outputs are connected in this example. The associated functions return the list of the x coordinates and the list of the y coordinates of the breakpoints in the curve.

The value $[\![\mathbf{K}]\!]_G(1)$ is then computed by solving the set of equations (we drop the subscripts):

$$\begin{cases} [\![\mathbf{K}]\!](1) = \textit{om-}*([\![\mathbf{H}]\!](1), [\![\mathbf{J}]\!](1)) \\ [\![\mathbf{H}]\!](1) = \textit{om-scale}([\![\mathbf{A}]\!](2), \star, [\![\mathbf{R}]\!](1)) \\ [\![\mathbf{J}]\!](1) = \textit{om-}*([\![\mathbf{R}]\!](1), [\![\mathbf{S}]\!](1)) \\ [\![\mathbf{R}]\!](1) = [\![\mathbf{R}]\!]_1() = 4 \\ [\![\mathbf{S}]\!](1) = [\![\mathbf{S}]\!]_1() = 0.15 \\ [\![\mathbf{A}]\!](2) = \textit{BPF}_2(\star, \star, \star, \star) \end{cases}$$

This set of equations can be solved easily by substitutions, which corresponds to a demand-driven evaluation strategy starting from $[\![\mathbf{K}]\!](1)$ and following the functional dependencies displayed in Fig. 2.

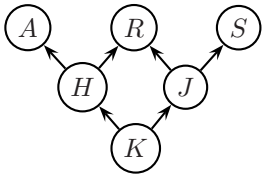

FIGURE 2: Call graph of the evaluation of **K** in Fig. 1.

3.3 Semantics of Flagged Boxes

The previous semantics is adapted to the description of the evaluation of a functional program specified as a first-order data-flow graph. However, it does not reflect the complete behaviour of OM visual programs as described in Section 2.3: in order to take into account the incremental nature of the OM workflow, and to integrate the mechanisms of "locked" box values and of higher-order functions ("lambda-state" boxes), we extend the semantics introducing the notions of *generation*, *box state* and *staggered evaluation*.

Generations and Box States In Section 2.2 we introduced the notion of *generation* to describe the evaluation triggered by the user who requires the value of some arbitrary box output in the visual program. This concept is formalized in the semantics by the state G^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 $G^t \to G^{t+1}$. Between two generations, the program G can change because some edges and boxes may have been deleted or added, some box states may have changed (locked, unlocked, lambda, etc.) and some locked value may have been edited. We model this situation by a sequence of quadruples

$$(G^t, \mathit{flag}^t, e^t, r^t)_{t \in \mathbb{N}} \tag{1}$$

each describing a generation, where:

- $G^t = (\mathsf{B}^t, \mathsf{E}^t, \mathsf{s}^t, \mathsf{t}^t)$ is the patch at generation t;
- $\mathit{flag}^t : \mathsf{B}^t \to \{\square, \boxtimes, \boxed{e}, \boxed{\lambda}\}$ gives the state of the boxes in the patch G^t: $\square$ correspond to a box with the standard behavior, $\boxed{\lambda}$ to an abstracted box, $\boxtimes$ to a locked box and $\boxed{e}$ to a locked box whose output values have been manually edited;
- For boxes $b \in \mathsf{B}^t$ such that $\mathit{flag}(b) = \boxed{e}$, the function $e^t(b, k)$ gives the edited value of its k-th output;
- $r^t \in \mathsf{B}^t$ is the box on which the user requests the evaluation.

Notice that the user requests the evaluation of *all* the outputs of *one* box and that a box is locked for all its outputs (it is not possible to lock only some specific outputs).

Ancestors In Section 3.2, we have outlined that the computation of the outputs of a box b depends only on the outputs of the boxes in b^+. The locking of a box makes the computation of this dependency set slightly less straightforward.

The *ancestors* $\uparrow A$ in a patch G^t of a set of boxes A are the boxes that can be reached from the boxes of A going from inputs to outputs, without having previously passed through a locked box. Let $<_\square$denotes the restriction of the binary relation $<$ to the standard boxes: $b <_\square b'$ iff $b < b'$ and $\mathit{flag}^t(b) = \mathit{flag}^t(b') = \square$. Formally,

$$\uparrow A = \{\, b \in \mathsf{B}^t \mid \exists b' \in A, b' <_\square^* b \,\},$$

where $<_\square^*$ is the reflexive transitive closure of $<_\square$.

Fig. 3 represents the call graph of the evaluation of box **C** in our example patch. The ancestors $\uparrow\{\mathbf{C}\}$ of **C** are coloured in grey. Note that **G**, **E**, **F**, **N**, **O**, **P**, and **I** are excluded from $\uparrow\{\mathbf{C}\}$ because of the locked state of **B**.

Staggered Evaluation The previous framework 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:

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

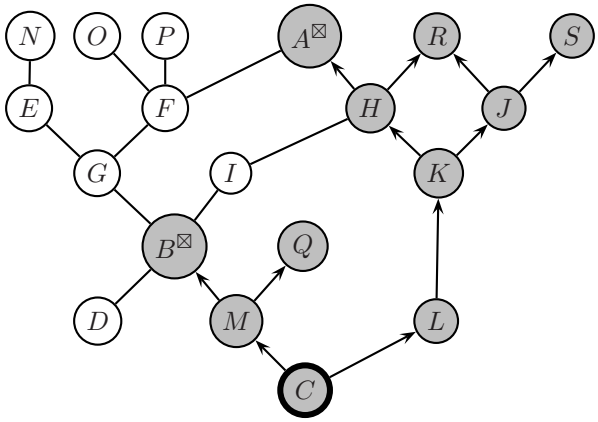

FIGURE 3: Call graph of the evaluation of **C** in Fig. 1.

with

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

and

$$u = \lambda x_{j_1} \dots x_{j_p}.[\![b]\!]_k(w_1, \dots, w_{in(b)}), \tag{2}$$

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

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

In the previous expression, note that if input j is not connected to some box's output, then w_j is a variable bound by the lambda abstraction in Eq. (2). The relations $\cdot \ {}_\cdot{<}_\cdot \ \cdot$ that appear in the definitions of v_i and w_j refer to the current patch G^t. Also note that we dropped the subscript G^t in the notation of the semantic function $[\![b]\!]^t_{G^t}(k)$ and that this function is defined on all boxes $b \in \mathcal{B}$, in order to allow for box additions and deletions between two successive states of a patch.

4 Towards a reactive model

In Section 3, we presented a denotational semantics corresponding to the current OM implementation[7]. OM implements this semantics with a recursive demand-driven process starting at the box evaluated by the user and propagating to the ancestors via the box inputs and connections. Function boxes use their input values as arguments; factory boxes use them to determine the initialisation values for the different slots of the class to be instantiated. If an input is connected to another box, then this box is evaluated recursively in order to yield the value. The evaluation of a box therefore amounts to evaluating a Lisp expression[8] corresponding to the graph rooted at this box. The result of the box evaluation is temporarily saved as the *value* of this box. As mentioned in Section 2.3 the management of multiple calls to the same node or section of the graph is done explicitly by the user using the *eval-once* option.

As a visual language interpreter, OM reacts to the user's inputs; however, the program runs are not reactive processes: they correspond to the evaluation of an expression, processing some input data and

[7]This semantics is valid for OM 6 and all the previous versions of the visual language.

[8]Patches can actually be converted into Lisp expressions, then compiled into functions used for the evaluation of the corresponding *abstraction* boxes.

returning some values without interruption. Visual programs are therefore closed expressions that *cannot react and update in response to external inputs*. By integrating reactivity here, we mean the ability for the OM patches to perform computations as a response to external events or program changes, and to propagate these events through the functional expressions denoted by these patches.

From a certain point of view, the current demand-driven evaluation model can also cause problems of *consistency in the functional expressions*. Computations are triggered exclusively by user requests, and not by any action modifying the patch G^t, so there exists no relation between the edition updates and the evaluations. For instance, the boxes in b^- connected to the output of an edited or evaluated box b are not updated in G^t, since they are not in the upper bounds of r^t. Reactivity in this context would also imply maintaining a set of functional constraints between inputs, outputs and other parts of the visual programs any time these programs may change (that is, if they are edited or if the value of a component is updated).

4.1 Reactivity: Specification and Challenges

The idea of reactivity introduces a "data-driven" conception in the scheduling of the visual program executions: the visual programs should be considered at the same time as functional expressions subject to interpretation or evaluation, and as data-flow graphs propagating data and triggering computations. To do so, an obvious approach is to systematically update all the boxes impacted by an edition, or more generally, related to a new generation. A proper definition of "impacted", that is, of the scope of the different events in the graph, must therefore be established.

At this point, it might be useful to stress the specificity of our objective as compared to a real-time system. Although our targeted system is reactive, time is still driven by a logical clock[9] corresponding to the successive states of the graph G^t. Inside every logical instant, the time elapsed during the computations (even if it can be relatively long) does not matter. Indeed, a design choice of OM is that the possibility of specifying arbitrary complex computations is always privileged over real-time constraints[10]. We therefore want to make no concession regarding the current specificities of the environment, and in particular about its "out-of-time" declarative characteristics, at the core of the computer-aided composition approach.

We need to maintain the incremental, formal and experimental aspects of the program construction in OM, and enhance them with reactive features when relevant. In other words, this language extension must be *conservative* and not interfere with existing programs, semantics, and behaviour. The reactive feature must be optional and potentially applicable *locally* to any existing functional programs. It should not overload the (visual) programming task and be easily switchable on and off for top standard expressions. To some extent, the design goals in this system extension are therefore similar to those of *adaptive functional programs*, as described in [1] (see Section 6).

We propose to use the existing visual programming constructs of OM and extend their behaviour with a finer semantics, mixing data- and demand-driven evaluation, in order to trigger computations in the parts of the functional graphs impacted by incoming events or changes. In the next paragraphs we introduce a reactive extension of the OM semantics presented in Section 3. An implementation will then be proposed in Section 5.

4.2 Events and Requests

Events In the reactive framework, we define an *event* E to be any subset of the boxes in a patch G which leads to an update, propagating following the data-flow connections in this patch. Because we are in an interpreted framework, it is easy to blur the distinction between the building phase and the evaluation phase of a patch. The notion of event therefore 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* (*e.g.* messages sent by other

[9] Aside from real-time considerations, this conception of a reactive system relates more to an event-triggered approach, as opposed to time-triggered systems [29].

[10] Usually, the computations that must respect hard real-time or strong synchrony constraints are delegated to external processes (*e.g.* sound synthesizers with which event-reaction communications are established). Such computations are distinct from higher-level compositional processes executed within the OM environment, where a best-effort strategy is acceptable.

active patches in OM, by an instrument connected via MIDI or by another application via UDP networking). As a matter of fact, the interactions with external processes can be handled as value boxes updated (*i.e.* edited) by these external processes.

Active Boxes For the same reasons that motivate the locking of a box (see Section 3.3), it is not always desirable to update the values of a box in response to an arbitrary event. In addition and for the sake of compatibility, we underlined earlier that the default behaviour of a visual program should be exactly the same as in the standard demand-driven evaluation mode, and that it should be easy for a programmer to turn on and off the reactivity in specific parts of the visual programs.

Thus, we add an additional state flag to the boxes in the form of a predicate $active^t$ specifying if a box must be sensitive to events. Inactive boxes stop the propagation of the updates.

Requests 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 update propagate to its descendants. So, we call a *request* any subset of the boxes of a patch G on which the user requires an evaluation.

The 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, and hence need an explicit request to be updated, and because requests provide a finer control in the case of non-functional boxes (*e.g.* boxes with a mutable state).

4.3 Semantics

In the reactive semantics of a patch, 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 Eq. (1); 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 following 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.

Descendants of a Set of Boxes We can now compute the set of boxes that must be evaluated as a consequence of an event A. The *descendants* of A, written $\downarrow A$, are 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 $<_a$. The 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).

Fig. 4 represents the propagation of the event $\{\mathbf{R}\}$ in our example patch (for instance, caused by a modification of the value in $\mathbf{R}$ by the user). The boxes in $\downarrow\{\mathbf{R}\}$ are coloured in grey.

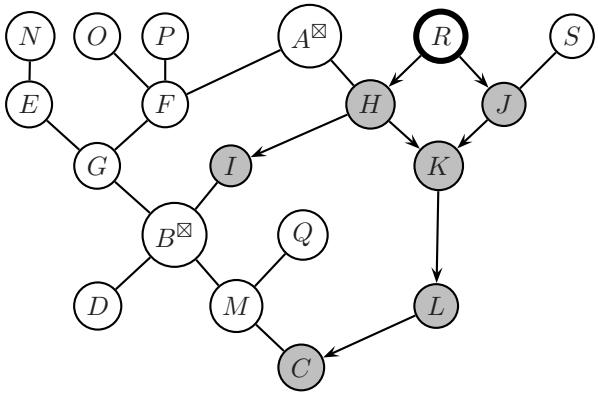

FIGURE 4: Propagation of event $\{\mathbf{R}\}$ in the *reactive* patch from Fig. 1. We suppose that all boxes are active. Notice that $\mathbf{R} \notin \downarrow\{\mathbf{R}\}$: the values associated to $\mathbf{R}$ are obtained by edition, not by evaluation.

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

$$
\begin{aligned}
(\!|\cdot|\!)^t(\cdot) \quad &: \quad \mathcal{B} \times \mathbb{N} \to \mathcal{V} \\
(\!|b|\!)^t(k) \quad &= \quad \begin{cases}
\star & \text{if } b \notin \mathsf{B}^t \\
e^t(b,k) & \text{if } \mathit{flag}^t(b) = \boxed{e} \\
(\!|b|\!)^{t-1}(k) & \text{if } \mathit{flag}^t(b) = \boxtimes \\
u & \text{if } \mathit{flag}^t(b) = \boxed{\lambda} \\
[\![b]\!]_k(v_1, \ldots, v_{in(b)}) & \text{if } b \in \left(\uparrow R^t \cup \downarrow E^t\right) \\
(\!|b|\!)^{t-1}(k) & \text{otherwise}
\end{cases}
\end{aligned}
$$

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

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

The remarks made for Eq. (2) also hold here.

5 Implementation

The semantics presented in the previous sections is independent from the conditions of a transition from a generation to the next one. The implementation in OM must define 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.

5.1 Events

The main conditions for events (that is, the conditions making for an active box b to become part of E^t at the next generation) are the different user edits, modifications, evaluations and other actions performed on the program.

Some of the visual program components might also be turned into event *sources* in order to spontaneously become part of an event E and trigger a new generation. Sources can be passive and simply react to an external process modifying their value, or active and trigger themselves explicit updates at any time during their computation. The production of such an event can be triggered by querying the environment, at the end of a generation or handled through a separate thread. Source boxes can include for instance:

- Widget components (*e.g.* buttons, sliders) reacting to user actions by value changes and corresponding updates in the OM patches;
- Source boxes performing (potentially infinite) loops or iterative processes, during which events are produced and propagated;
- "Receive" boxes waiting for incoming messages from UDP or MIDI ports, and processing/propagating them sequentially upon reception.

5.2 Update Strategy

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. It is reasonable to choose to handle events asynchronously, that is, one at a time. So, requests and evaluations are handled sequentially in a single thread and at each generation, either $\uparrow\{b\}$ or $\downarrow\{b\}$ shall be computed for a given box b in the patch. This strategy serializes the events: every incoming event triggers an update and the next one is processed when the previous update terminates. This strict ordering prevents conflicts, as user edits are also taken into account after any previous event processing is done.[11]

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. In a first approach, we try to reuse these functions in the reactive framework, in order to capitalize as much as possible on the compilation process. This can be achieved easily, if we accept to over-approximate the set of boxes to be updated/evaluated. A finer evaluation strategy will then be proposed.

Over-Approximation The data-flow graph of the OM patch is maintained dynamically during the incremental building of the patch. Every active box registers and updates its output connections as they are added, edited or deleted.

The update strategy for a box b in E can then start by locking b temporarily. The computation of $\downarrow\{b\}$ is then performed by a simple depth-first traversal of the data-flow graph following the registered and active output connections of the boxes. During this traversal, the boxes in $\downarrow\{b\}$ are marked in order to avoid multiple visits to the same sections of the graph. When a *terminal box* b' (a box without descendant) is reached, the standard OM evaluation of b' is triggered .

Locking b at the beginning of the process avoids it (and eventually its ancestors) being evaluated as part of the update process. At the end of the depth-first traversal, b is unlocked.

Exact Approximation The previous strategy is convenient for it allows the reuse the evaluation code in OM. However, it presents two shortcomings:

[11] The actual behaviour of the system might, however, depend on design choices at implementing the serialization. For instance, one could disable or delay any response to user actions while events are being processed, or enable edits that would be taken into account in subsequent updates, while propagating on their own at a later stage.

- Ancestors of boxes in $\downarrow\{b\}$ which are not themselves in $\downarrow\{b\}$ (that is, $\uparrow\downarrow\{b\} - \downarrow\{b\}$) are evaluated and updated during this process. For instance in Fig. 4, boxes **M**, **Q** or **S** will be evaluated on demand of **C**.
- Boxes connected to several terminal nodes, or several times indirectly to a same terminal node, might be evaluated several times as well.

These behaviours shall not impact the result of the computation, provided there are no side effects or indeterminacy and that all boxes in the patch are active. Even if they can be prevented or controlled using the *eval-once* and *locked* states, an exact evaluation strategy can be defined by a slight modification of the default update and evaluation strategies:

- Store the terminal boxes of $\downarrow\{b\}$ in a set instead of evaluate, and evaluate them *after* the marking of $\downarrow\{b\}$ by the depth-first traversal is completed.
- Test before the evaluation of a box if it has been marked (hence, if it belongs to $\downarrow\{b\}$ and needs to be updated), or not (and then just return its current value). This evaluation strategy therefore applies only in the context of the update process (not in the "standard" case of a user request).

6 Related Work

As mentioned in the introduction, visual programming languages are commonplace in music, either for real-time signal processing and interactive systems design (*Max* [37], *Pd* [38], OSW [13]), or for formal composition and generation of musical structures (*Patchwork* [32], *OM*, *PWGL* [33]).

Generally speaking, and except for a few exceptions, data-flow (graph-based) visual programming languages have been designed with a data- or event-driven model. *VPL* is a noteworthy project that presents (like OM) the particularity to be implemented with demand-driven evaluations [31]. This declarative language dedicated to image processing also aims at combining the phases of development, compilation and evaluation of the programs in interactive workflows. As in OM, the demand-driven model allows functional processes to be developed including abstractions and higher-order functions. To some extent, reactive features are even implicitly present, due to some automatic graph computations triggered by the behaviours of "producer" or "consumer" boxes.

Other examples of this data- vs. demand-driven duality can be found in different visual programming languages. Several works, for instance, aim at "simulating" demand-driven, declarative approaches using event-driven data flow (*e.g.* *Cantata* [39] the visual language developed for the *Khoros* image processing environment).[12] The *DeVide* visual language also combines event-driven and demand-driven in a "hybrid" scheduling model [7]. Event-driven computation is performed by default by executing modules in a specific order, determined after a topological sort of the dependency graph. When possible, "streamable" modules are grouped and evaluated together using the demand-driven strategy. These processes, however, are transparent to the user and therefore constitute more an optimization than an actual programming feature of the visual language.

A functional/reactive model close to the one targeted in our work can also be found in form-based visual languages, where cell dependencies can be implemented by hybrid demand-/data-driven mechanisms [12]. *AVS* [41] is another early visual language for scientific visualization where a similar hybrid paradigm is supported: interesting data-driven features are added to the demand-driven model, such as the components' reactivity to global variable changes, and the possibility to "compile" the data-flow network into end-user-oriented interfaces. However, the "flow executive" is an autonomous process handling component dependencies and events in order to automatically request adequate computations, while in the OM reactive extension, the visual program is an autonomous entity managing by itself the internal evaluations and change propagations upon user actions or events.

[12] Although it was based on a data-driven model, this visual language followed similar objectives as OM, concerning for instance the possibility to incrementally interpret boxes and specific parts of the programs in an exploratory, experimental context.

By focusing on the declarative approach in the implementation of reactive features, our project shall also be compared to previous works in the field of functional reactive programming (FRP), a declarative programming framework for the modelling of hybrid reactive systems involving combinations of discrete and continuous components [20]. Several FRP implementations have been proposed, addressing specific domains such as reactive animations [16], real-time systems [42], user interfaces [18] or sound synthesis [24]. *Euterpea* [28] is also a functional/reactive language based on Haskell specifically designed for music representation, algorithmic composition and analysis and low-level audio synthesis. Interestingly, if *Euterpea* can execute some programs in real-time, more complex programs require writing the result to a file: an issue directly related to the distinction we have introduced between "out of time" and "real-time" musical systems. In *Euterpea*, interactions with the external environment take the form of dedicated IO operations and rely on monads to constrain adequately the sequence of evaluations, a usual approach in lazy programming languages.

In the different FRP implementations [16, 18], program executions generally perform the sampling of *sinks* in functional expressions, in order to model the temporal behaviours of a system. This demand-driven approach is well adapted to the design of continuously executing behaviours such as animations, simulations and embedded systems, or sound synthesis processes (see [24]). However, it does not match the idea of reactive expressions as described previously in the motivations of our work (see for instance Section 4.1). In addition, and in certain situations, the demand-driven ("pull-based") reactive model posed efficiency issues, by requiring continuous and complete evaluations of the functional expressions at the system (or best) sample rate. Elliott [19] proposed a formal model for integrating data-driven aspects ("push-based" evaluations) in the FRP framework, principally by introducing the notion of *reactive value*. *Event-Driven FRP* (E-FRP) [43] implements another similar extension to FRP, where event occurrences affect statically decidable parts of the system state. Unlike FRP standard behaviours that can change over continuous time, E-FRP behaviours can change only when an event occurs.

In line with this approach, two other FRP implementations present interesting similarities with our system. *Frappé* [15] is an implementation of FRP in Java, which uses the Java Beans framework to model the reactive aspects of functional programs using the notion of events (conditions notifying other objects of their occurrence) and properties (named mutable attributes that may change over time). This system (initially designed for graphical user interfaces) yields an event-driven model for change propagation. Each FRP combinator is implemented as a Java class, and "listeners" register to be notified whenever one of this class' property value changes in order to compute its own changes and in turn notify its registered listeners via event handler methods. In our visual programming context, the notion of box is used to embed the reactive layer of functional components and the implementation of this mechanism. The "listeners" registration is done automatically when reactive boxes get connected to other boxes; then the change, either caused by the program, or by some user action, propagates to dependencies following a similar model. *FrTime* [14] is another language integrating reactive aspects in Scheme. As in *Frappé*, computation is not driven by a central clock, but by external events. The reactive aspects are handled in a dedicated thread, which builds a graph of data-flow dependencies when the program is executed or evaluated, and then keeps running in parallel to catch events, route them and propagate the changes through this graph. This thread communicates with the standard Scheme REPL via a message queue, and updates components of the Scheme expressions depending on incoming events and changes. *FrTime* is therefore concerned with functional expressions and program manipulation, and thereby presents a number of similarities with the OM-reactive extension, where programs must respond to and process events from external sources, and the arrival of an event triggers a computation that propagates in a tree of dependencies. It also pays interest in that existing programs must remain valid, and possibly extensible (with not significant source transformation) to include the reactive data-flow features.

These characteristics are also present in the *adaptive functional programming* (AFP) framework proposed in [1]. In this framework, ML programs are extended to respond to input changes by re-evaluating portions of code and updating their output accordingly. As in OM, the reactive extension applies as an additional and optional behaviour to any existing programs. Extending programs must not require significant changes, and reactivity can be restricted to some specific portions of these programs. Some similarities also exist in the

implementations, for instance with the embodiment of function calls in specific structures (in our case, boxes) allowing the temporary recording and reading of the results of the computations. Slight modifications of a standard program enable the reactive (adaptive) behaviour, depending on registered dependencies. However, in AFP the adaptive update of the program again relies on a separate process keeping track of dependencies and being responsible for the propagation of changes and the scheduling of evaluations. In contrast, OM dynamically registers the dependencies in the boxes upon building the program graph, and uses the attributes of the language components that are embedded in the visual programs to propagate events and changes in these same programs.

7 Conclusion

Main Contributions. Event-driven computation is not straightforward to embed in a language like OM, based on Common Lisp and relying *a priori* on a demand-driven evaluation strategy. In this paper, we presented a fairly simple and elegant formalism for the integration of reactive processes in the visual programming language.

The first contribution of this paper is to provide a denotational semantics for the visual language, whose workflow and behaviour is not usual as compared to standard functional languages. This semantics and the related implementation rely on the concept of box which embeds and represents the notion of function call in the visual language. It gives account for the original "incremental" programming paradigm of the environment.

We then extended this semantics to match with the reactive features underlined as the motivation of our work. This extension combines well with the incremental building of patches; it defines additional box attributes allowing for the propagation of events and reactive updates. We therefore avoided the definition of new language primitives to encapsulate the reactive concepts, with the objective to provide the language users with simple means to experiment the "activation" of declarative programs by simply turning on or off the reactive behaviour of the different visual program components.

As mentioned in Section 6, an important specificity of this system is that the scheduling is not centralized in a separate process added to the environment, but is spread and handled by the boxes themselves during their evaluations and manipulations by the user/programmer. An original aspect of our work is indeed that the incremental building of the programs is included in the reactive semantics, and is handled by the same mechanism as the evaluation of these programs.

It may be important to precise that this work concerns the core of the OpenMusic visual language: the presented semantics and its reactive extension focus on the basic evaluation mechanism and is independent of alternative existing extensions such as the *maquette*[13].

Impact and Applications. Foreseen applications in the computer music domain are multiple, and introduce new possibilities of compositional interactions with software environments and external devices. From the user/ compositional perspective, OM visual programs are representations of musical processes: making such representations reactive allows these programs to interact and execute in response to actions, modification, or other events occurring in a given framework, and maintain the consistency of the functional expressions dynamically, without explicit requests of boxes (re)evaluations. In compositional contexts, programs developed in interactive systems or languages are indeed likely to be executed both as a—data-driven—response to the composer actions (and by extension, to musical interactions), and as the explicitly requested—demand-driven—result of a formalised process. As stated in the introduction of this paper, the current available systems generally fail in addressing this duality[14].

Reactive visual programs in OM may also be used as symbolic processing components in live musical interactions, for instance to record, analyse, or even learn from incoming musical data, and to produce transformed or dynamically generated musical data on the model of computer-generated improvisation systems [5].

[13]The *maquettes* in OM are hybrid objects at the meeting point of visual programs and musical sequencers. They provide powerful means to control temporal aspects in both the programs and the musical structures—see for instance [6, 9, 8].

[14]These aspects of compositional workflows have also recently been discussed in [40].

Indeed, it is interesting to note that computations in OM are likely to produce events, either destined to other parts or other programs designed in the environment, or to external applications or devices (typically, sending musical data to a player, or to another system running in real-time during a performance). This extension of the programming and computational model in the visual language therefore also introduces a renewed status of computer-aided composition software in larger-scale inter-applications workflows. The programming environment may not be used only for generating musical structures any more, but it can be embedded itself in a musical/temporal flow, responding to incoming events by generating, sending and rendering the data structures produced by processing the incoming data.

Future Work. We mentioned in section 5.1 that the notion of *source* shall be further developed in this new reactive context. Additional language primitives should be provided, for instance to handle the routing or the iterative processing or collection of incoming data. Interesting ideas in this perspective are for instance proposed in *FrTime* [14].

Another aspect to improve concerns the level of reactivity of the components (boxes) in the visual programs. It is indeed possible to imagine situations where an active component might need to react only upon modification or update of one (or several) of its input (or, symmetrically, to update a subset only of the connected "listeners")[15]. This flexibility in the propagation process would involve implementing reactivity at the more precise level of the patch connections, instead of considering it at the level of the boxes as it is currently the case.

Finally, by considering a logical system clock driven by user actions and events, we eluded the issue of computational time in our formalism. However, the system reliability would require more advanced management of time and delays in computation in order to cope with complex calculi in interactive performance situations.

References

[1] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive Functional Programming. In *Principles of Programming Languages (POPL'02)*, Portland, OR, USA, 2002.

[2] C. Agon. *OpenMusic: Un langage visuel pour la composition musicale assistée par ordinateur*. PhD thesis, Université Pierre et Marie Curie, Paris, 1998.

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

[4] G. Assayag. Computer Assisted Composition Today. In *1st Symposium on Music and Computers*, Corfu, Greece, 1998.

[5] G. Assayag, G. Bloch, M. Chemillier, A. Cont, and S. Dubnov. 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, 2006.

[6] G. Assayag, C. Rueda, M. Laurson, C. Agon, and O. Delerue. Computer Assisted Composition at IRCAM: From PatchWork to OpenMusic. *Computer Music Journal*, 23(3):59–72, 1999.

[7] C. P. Botha and F. H. Post. Hybrid Scheduling in the DeVIDE Dataflow Visualisation Environment. In *Simulation and Visualization*, Magdeburg, Germany, 2008.

[8] J. Bresson and C. Agon. Temporal Control over Sound Synthesis Processes. In *Sound and Music Computing (SMC'06)*, Marseille, France, 2006.

[9] J. Bresson and C. Agon. Scores, Programs, and Time Representation: The Sheet Object in OpenMusic. *Computer Music Journal*, 32(4):31–47, 2008.

[15] Similar problems in the domain of signal processing and control are presented in [22].

[10] J. Bresson, C. Agon, and G. Assayag. Visual Lisp/CLOS Programming in OpenMusic. *Higher-Order and Symbolic Computation*, 22(1):81–111, 2009.

[11] J. Bresson, C. Agon, and G. Assayag. OpenMusic – Visual Programming Environment for Music Composition, Analysis and Research. In *ACM MultiMedia*, Scottsdale, AZ, USA, 2011.

[12] M. M. Burnett and A. L. Amber. A declarative Approach to Event Handling in Visual Programming Languages. In *IEEE Workshop on Visual Languages*, Seattle, WA, USA, 1992.

[13] A. Chaudhary, A. Freed, and M. Wright. An Open Architecture for Real-time Music Software. In *International Computer Music Conference*, Berlin, Germany, 2000.

[14] G. H. Cooper and S. Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In *European Symposium on Programming*, Lecture Notes in Computer Science 3924, Vienna, Austria, 2006. Springer.

[15] A. Courtney. Frappé: Functional reactive Programming in Java. In *Practical Aspects of Declarative Languages (PADL'01)*, Las Vegas, NV, USA, 2001.

[16] A. Courtney, H. Nilsson, and J. Peterson. The Yampa Arcade. In *Haskell Workshop*, Uppsala, Sweden, 2003.

[17] R. B. Dannenberg, P. Desain, and H. Honing. Programming Language Design for Music. In Roads et al., editor, *Musical Signal Processing*. Swets and Zeitlinger, 1997.

[18] C. Elliott. Declarative Event-Oriented Programming. In *Principles and Practice of Declarative Programming (PPDP'00)*, Montreal, QC, Canada, 2000.

[19] C. Elliott. Push-Pull Functional Reactive Programming. In *Haskell Symposium*, Edinburgh, Scotland, 2009.

[20] C. Elliott and P. Hudak. Functional Reactive Animation. In *International Conference on Functional Programming (ICFP'97)*, Amsterdam, The Netherlands, 1997.

[21] M. Erwig and B. Meyer. Heterogeneous Visual Languages – Integrating Visual and Textual Programming. In *IEEE Symposium on Visual Languages*, Darmstadt, Germany, 1995.

[22] G. Essl. Playing with Time – Manipulation of Time and Rate in a Multi-rate Signal Processing Pipeline. In *International Computer Music Conference*, Ljubljana, Slovenia, 2012.

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

[24] G. Giorgidze and H. Nilsson. Switched-On Yampa. In *Practical Aspects of Declarative Languages (PADL'08)*, San Francisco, CA, USA, 2008.

[25] T. R. G. Green and M. Petre. When Visual Programs are Harder to Read than Textual Programs. In *European Conference on Cognitive Ergonomics (ECCE)*, San Francisco, CA, USA, 1992.

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

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

[28] P. Hudak. *The Haskell School of Music – From Signals to Symphonies*. http://haskell.cs.yale.edu/euterpea/haskell-school-of-music, 2013.

[29] H. Kopetz. Event-triggered versus time-triggered real-time systems. In *Operating Systems of the 90s and Beyond*, pages 86–101. Springer, 1991.

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

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

[32] M. Laurson and J. Duthen. Patchwork, a Graphic Language in PreForm. In *International Computer Music Conference*, Ohio State University, USA, 1989.

[33] M. Laurson and M. Kuuskankare. PWGL: A Novel Visual Language Based on Common Lisp, CLOS, and OpenGL. In *International Computer Music Conference*, Gothenburg, Sweden, 2002.

[34] M. Malt. Concepts et modèles, de l'imaginaire à l'écriture dans la composition assistée par ordinateur. In *Séminaire postdoctoral Musique, instruments, machines*, Paris, France, 2003.

[35] M. Mathews. The Digital Computer as a Musical Instrument. *Science*, 142(3592):553–557, 1963.

[36] P. D. Mosses. *Handbook of Theoretical Computer Science*, volume 2, chapter Denotational Semantics, pages 575–631. Elsevier Science, 1990.

[37] M. Puckette. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, 15(3):68–77, 1991.

[38] M. Puckette. Pure Data : another integrated computer music environment. In *Second Intercollege Computer Music Concerts*, Tachikawa, Japan, 1996.

[39] J. R. Rasure and C. S. Williams. An Integrated Data Flow Visual Language and Software Development Environment. *Journal of Visual Languages and Computing*, 2(3):217–246, 1991.

[40] H. H. Rutz. A Reactive, Confluently Persistent Framework for the Design of Computer Music Systems. In *Sound and Music Computing Conference*, Copenhagen, Denmark, 2012.

[41] C. Upson, T. Faulhaber Jr., D. Kamins, D. H. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. Van Dam. The Application Visualization System: A Computational environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, 1989.

[42] Z. Wan, W. Taha, and P. Hudak. Event-Driven FRP. In *International Conference on Functional Programming (ICFP'01)*, Florence, Italy, 2001.

[43] Z. Wan, W. Taha, and P. Hudak. Event-Driven FRP. In *Practical Aspects of Declarative Languages (PADL'02)*, Portland, OR, USA, 2002.

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

[45] K. N. Whitley. Visual Programming Languages and the Empirical Evidence For and Against. *Journal of Visual Languages and Computing*, 8(1):109–142, 1997.

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