

Specification of a reactive computation model for OpenMusic

Jean Bresson, Jean-Louis Giavitto

▶ **To cite this version:**

Jean Bresson, Jean-Louis Giavitto. Specification of a reactive computation model for OpenMusic. [Research Report] IRCAM. 2014, pp.17. hal-00959312

HAL Id: hal-00959312
https://hal.science/hal-00959312

Submitted on 14 Mar 2014

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

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

ANR-12-CORD-0009
INEDIT Project
Programme ContInt
http://inedit.ircam.fr/

Specification of a reactive computation model for OpenMusic

Jean Bresson, Jean-Louis Giavitto
IRCAM / UMR 9912 STMS
bresson@ircam.fr, giavitto@ircam.fr

Résumé

This document is a report on task 3.3.b concerning the specification of a reactive computation model in OM. We have defined a denotational semantics of the visual language, that takes into account the evaluation, but also the construction of a visual program. This formal definition is then extended in order to model reactive processes.

The core content of this report will be published in the *Journal of Visual Languages and Computing*, although we give some additional precisions here that do not appear in the article.

1 Introduction

Different kinds of programming languages coexist in computer music. Real-time data-flow languages are the more common ; they are used for the design of reactive systems and are usually oriented toward interactive applications and live performance. Max [1] or PureData [2] are two reference visual languages in this category. On the other hand, declarative languages are used in the computer-aided composition domain for the off-line modelling and generation of musical structures. The Patchwork visual language was a pioneering work in this field [3], providing a diagram-based interface for Lisp programming (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 derive from this language and follow similar concepts [4, 5].
In this report we present a reactive extension of OpenMusic (OM), which merges these two approaches in the general context of computer-aided composition. OM is a domain-specific visual programming language used by composers manipulate musical data [6], which allows to graphically build and evaluate musical processes as functional expressions. Adding reactivity to the off-line computer-aided composition paradigm will allow to connect compositional systems to real-time processes and environments, and bridge the "compositional" and "performative" aspects in musical applications. We believe this bridge can improve 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.
In Section 2, we present a brief overview of the OM visual programming environment and of its relevant aspects related to the present work. In Section 3 we propose a semantics describing the OM high-level programming paradigm (incremental program construction and evaluation) from a functional point of view, and then extend this semantics toward a reactive model (Section 4). An implementation of this semantics is then proposed, which merges in the existing visual programming (Section 5). 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.

2 The OM Visual Programming Language

The OpenMusic visual programming environment provides a graphical interface to the main constructions and features of the Common Lisp language (abstraction, higher-order function, iteration, recursion, object-oriented programming, etc.) [7], and enriches the Lisp structures with specific musical objects and dedicated functions. The computation paradigm extends the Lisp approach by combining local state with interactive by-need (demand-driven) computation of visual expressions [4]. 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. Projects and third-party libraries allow programming to be extended to a wide variety of purposes and applications [8, 9].
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. The rest of this section will describe this programming systems and workflow, focusing on the notions needed to understand the semantics presented in Section 3.

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. Figure 1 shows an OM patch. We will use it to analyse and describe the main language features.

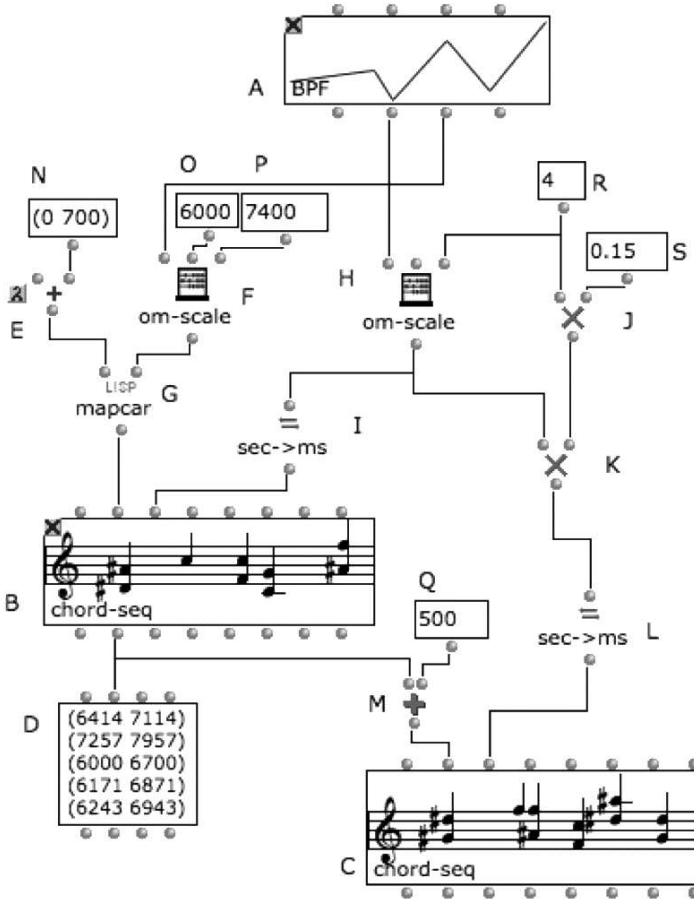


FIGURE 1 – A visual program in OM (the box labels **A** through **S** on the figure have been added for easier referencing in the text). Boxes **B** and **C** are score objects whose pitches and onsets are computed starting from the coordinates of the break-points of a piecewise linear curve (defined in box **A**). The *om-scale* function boxes (**F** and **H**) rescale these coordinates to adequate value ranges. *Mapcar* (box **G**) is a Lisp function used to apply $+$ to the pitch list. (Note that all arithmetic boxes ($+$, $\times$) can be applied to scalar values or to lists of values.)

The function associated to a box in OM can be either a simple Lisp function or a specific one defined with additional (graphical) attributes. 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. 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. 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.

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 a 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. 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" [10]. 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 or manually input 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 case of multiple calls to a 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 λ 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 $b, b', b_1 \ldots$ range over $\mathcal{B}$ and variables $e, e', e_1 \ldots$ 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 $1 \leq k \leq out(b)$;

2. if $\mathsf{t}(e) = (b, k)$ then $1 \leq k \leq in(b)$;
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 [11] with least element $\perp$. 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 $\perp$.
For each box $b \in \mathcal{B}$, there are $out(b)$ associated functions

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

giving the semantics of a box. Theses functions are assumed to be continuous in the sense of domain theory [11]. 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 :

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

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 are given as follows :

- $[\![\mathbf{K}]\!]_1 = [\![\mathbf{J}]\!]_1 = om\text{-}*$, 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) :

$$\begin{cases} [\![\mathbf{K}]\!](1) = om\text{-}*([\![\mathbf{H}]\!](1), [\![\mathbf{J}]\!](1)) \\ [\![\mathbf{H}]\!](1) = om\text{-}scale([\![\mathbf{A}]\!](2), \star, [\![\mathbf{R}]\!](1)) \\ [\![\mathbf{J}]\!](1) = om\text{-}*([\![\mathbf{R}]\!](1), [\![\mathbf{S}]\!](1)) \\ [\![\mathbf{R}]\!](1) = [\![\mathbf{R}]\!]_1() = 4 \\ [\![\mathbf{S}]\!](1) = [\![\mathbf{S}]\!]_1() = 0.15 \\ [\![\mathbf{A}]\!](2) = BPF_2(\star, \star, \star, \star) \end{cases}$$

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.

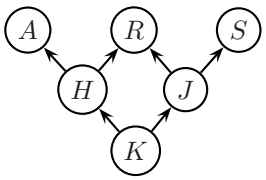


FIGURE 2 – Call graph of the evaluation of **K** in Fig. 1.

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

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

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 ;

- $flag^t : \mathsf{B}^t \to \{\square, \boxtimes, \boxed{e}, \boxed{a}\}$ gives the state of the boxes in the patch G^t : $\square$ correspond to a box with the standard behavior, $\boxed{a}$ 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 $flag(b) = \boxed{e}$, 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 paragraph 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 $flag^t(b) = flag^t(b') = \square$. 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 gray. Note that **G**, **E**, **F**, **N**, **O**, **P**, **I** are excluded from $\uparrow\{\mathbf{C}\}$ because of the locked state of **B**.

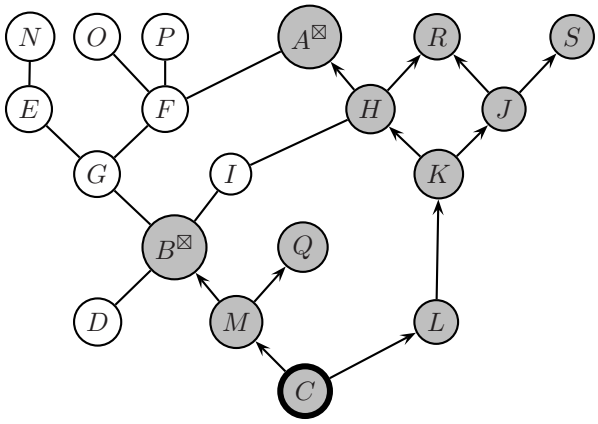


FIGURE 3 – Call graph of the evaluation of **C** in Fig. 1.

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 :

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

with

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

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.

3.4 Abstraction and Recursion

In OM, a box can be bound to a patch G and used in (another) patch. In particular, a box referring to a patch G can be included in its own associated patch achieving a recursive definition at user-level.
The handling of such references in our framework is as follows :

- Specific boxes in_k (with no input and only one output) and out_k (with one input and one output) are used to distinguish the inputs and the outputs of a patch. The semantic function associated with a box out_k is the identity.
- The set of (predefined) box identifiers $\mathcal{B}$ is extended with the set of user-defined box identifiers $\mathcal{U}$. The patch associated with $u \in \mathcal{U}$ is denoted by $G_u = (\mathsf{B}_u, \mathsf{E}_u, \mathsf{s}_u, \mathsf{t}_u)$.
- The semantics of a user-defined box $u \in \mathcal{U}$, with p in boxes and q out boxes, are the q functions $\mathcal{V}^p \to \mathcal{V}$ defined by :

$$[\![u]\!]_k = \lambda x_1 \dots x_p.[\![\mathsf{out}_k]\!]_{(G_u/\mathsf{in}_1 \leftarrow x_1, \dots, \mathsf{in}_p \leftarrow x_p)}(1), \qquad \text{for } 1 \le k \le q.$$

  where $(G_u/\mathsf{in}_1 \leftarrow x_1, \dots, \mathsf{in}_p \leftarrow x_p)$ is the graph G_u, where the constants x_i have been substituted for in_i (a constant is a box with no input and one output).

In the previous equation, the function $[\![_]\!]_(_)$ refers to the function defined in Section 3.2.
Notes :

- The handling of recursive definitions at the level of the Lisp functions defining the semantics of a predefined box does not appear in the semantics, because at this level, such functions are represented by predefined constants.
- A fixed-point operator is not explicitly needed, because we suppose that the binding between box identifiers and patches is implicitly given : at the level of our semantics, there is no binding construction in the language.
- The above semantics of user-defined boxes is defined within one generation and lifts in a straightforward way at the level of multiple generations.

4 Towards the Reactive Model

In Section 3, we presented a denotational semantics corresponding to the current OM implementation. 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 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 a 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 transformational programs that process input data and return values upon user requests. By integrating reactivity in this context, we mean the ability for the OM patches to perform computations as a response to external events or changes occurring in G^t, and to propagate these events through the functional expressions denoted by these patches. Reactivity can also be seen as the a set of functional constraints to maintain 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). These relations can ensure the permanent consistency of the functional expressions represented in visual programs, establishing relations between the edition updates and the evaluations (for instance, updating boxes in b^- connected to the output of an edited or evaluated box b, that are not in the upper bounds of r^t).

4.1 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 [1] 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 [2]. 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

1. Aside from real-time considerations, this conception of a reactive system relates more to an event-triggered approach, as opposed to time-triggered systems [12].
2. 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.

expressions. To some extent, the design goals in this system extension are therefore similar to those of *adaptive functional programs*, as described in [13].
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 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 motivates 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 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

$$(G^t, \mathit{flag}^t, e^t, active^t, R^t, E^t)_{t\in\mathbb{N}}\,,$$

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 two conditions :

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

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

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 **R** by the user). The boxes in $\downarrow\{\mathbf{R}\}$ are coloured in gray.

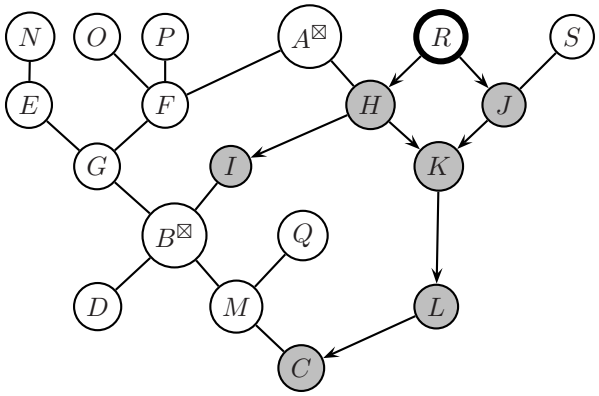


FIGURE 4 – Propagation of event $\{\mathbf{R}\}$ in the *reactive* patch from Fig. 1. We suppose that all boxes are active. Notice that $\mathbf{R} \notin \downarrow\{\mathbf{R}\}$: the values associated to **R** are obtained by edition, not by evaluation.

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

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

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

with

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

and

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

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.

Semantics of a Program Run If we want the previous semantics to give account also for programs that go wrong, we need to add an explicit definition of the semantics of a run.
The semantics of an element of the run (a graph G^t together with its flags, etc.) is a function in $\mathcal{S} = \mathcal{B} \times \mathbb{N} \to \mathcal{V}$. However, this function is evaluated on the boxes of $(\uparrow R^t \cup \downarrow E^t)$. If one of these evaluations does not terminate, the system is stuck and will not respond further to the requests of the environment.
So, the semantics of a run will be an element of $\mathcal{S}^\$$. We follow here the notations of [14] : $\mathcal{S}^\$ \cong \mathcal{S} \otimes \mathcal{S}^\$_\perp$ which differs from the domain of streams $\mathcal{S}^\infty \cong \mathcal{S} \times \mathcal{S}^\infty$ in that the former does not allow $\perp$ components to be followed by non-$\perp$ components. To build an element of this domain from the semantics associated to the elements of a run, we need the function :

$$\begin{aligned} \textsc{smash} \ : \ & \mathcal{S} \times 2^{\mathcal{B}} \to \mathcal{S} \\ & (f, B) \mapsto \begin{cases} \perp_{\mathcal{S}} & \text{if } \exists b \in B,\ \exists k, 1 \le k \le out(b),\ f(b,k) = \perp_{\mathcal{V}} \\ f & \text{elsewhere} \end{cases} \end{aligned}$$

SMASH is a continuous function. Let $s^t \in \mathcal{S}$ be the semantics associated to the t^{th} element of the run $r = (G^t, \mathit{flag}^t, e^t, \mathit{active}^t, R^t, E^t)_{t\in\mathbb{N}}$. The semantics $S = (S^t)_{t\in\mathbb{N}}$ of the run r is then defined as the element of $\mathcal{S}^\$$ defined by :

$$\begin{aligned} S^0 &= \textsc{smash}(s^0, \uparrow R^0 \cup \downarrow E^0) \\ S^{t+1} &= \begin{cases} \perp_{\mathcal{S}} & \text{if } S^t = \perp_{\mathcal{S}} \\ \textsc{smash}(s^{t+1}, \uparrow R^{t+1} \cup \downarrow E^{t+1}) & \text{elsewhere} \end{cases} \end{aligned}$$

Cyclic Programs The theoretical framework we have introduced can support new program structures such as cyclic programs, which are not allowed in the initial design of the visual language. Cyclic graphs can indeed be allowed in the reactive semantics, with some restrictions. A cycle within a patch would correspond to a kind of recursive definition, depending on values computed during the previous generation. Cycles would be handled as follows :

- We assume that each cycle of a cyclic graph $G = (\mathsf{B}, \mathsf{E}, \mathsf{s}, \mathsf{t})$ contains at least one distinguished box $\mathsf{delay}(d)$ with one input and one output, such that the graph $G_{(\mathsf{delay(d)}\leftarrow\star)}$ built from G by substituting the constant $\star$ with the $\mathsf{delay}(d)$ boxes and removing the pending edge, is acyclic.
  Informally, the semantics of a box $\mathsf{delay}(d)$ corresponds to the "d fby ..." construction (operator "followed by") in synchronous programming, which shifts in time a stream of values introducing the initial value d.
  Such a condition, used in Lustre [15] and in Esterel v4 [16], is a syntactic constraint sufficient to avoid the existence of *temporal shortcuts* and to ensure *causal consistency* : the program, viewed as a set of equations, admits a unique solution [17].

- Let G^t be a patch at generation t and let $\mathsf{delay}(d)$ be a delay box in G^t. The sole input of this box is eventually connected to the k^{th} output of a box b : $\mathsf{delay}(d)\ {}_1{<}_k\ b$. We write $in(\mathsf{delay}(d))$ for this box b and $index(\mathsf{delay}(d))$ for k. Then the semantics of G^t is defined as in Section 3.3, except for the delay boxes. Informally, the output value of a delay box at a generation t is the value of its input at generation $t-1$. Formally :

$$[\![\mathsf{delay}(d)]\!]^0_{G^0}(1) = d$$
$$[\![\mathsf{delay}(d)]\!]^t_{G^t}(1) = [\![in(\mathsf{delay}(d))]\!]^{t-1}_{G^{t-1}}(index(\mathsf{delay}(d))) \qquad \text{for } t > 0$$

  These equations hold for all the delay boxes in G^t and these boxes cannot be flagged. If $in(\mathsf{delay}(d)$ does not exist for some delay box (the box is not connected), then its output value is $\star$.

Note that this approach of delays and cyclic patches corresponds to a pure reactive approach : at least one external event is required to trigger a generation. The patch evaluation is not proactive in the sense that a cycle in the patch does not imply a future change of generation (a programmer has to explicitly trigger a new generation from within the patch). This approach fits well with the handling of an infinite source of external events such as an external clock. It is used for example in Lustre, in ReactiveML [18], SugarCubes [19] and in other explicitly stepped synchronous languages, where the compilation produces an evolution function which is iterated. Note however that the use of cyclic patches can be non intuitive : at a given generation, all boxes are not evaluated and so, the delays may refer to actually different shifts in time, which may obscure the understanding of the evaluation.

5 Implementation Directions

The semantics presented in the previous sections is independent from the conditions of a transition from a generation to the next one. A concrete implementation should 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.
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 can be 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. Such strict serialization of the events shall prevent all kind of conflicts, typically if events occur before previous events processing and update is done. The actual behaviour of the system, however, might depend on design and implementation choices.

5.1 Events

Implementations should define the conditions for events (that is, the conditions making for an active box b to become part of E^t at the next generation). Typically, events will be the different user edits, modifications, evaluations and other actions performed by the user on the program.
Some 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 of events 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. 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 ;
- 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

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 wish 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 : We suppose 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 :

- 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 Conclusion

Event-driven computation is not straightforward to embed in a language like OM, based on Common Lisp and relying on a demand-driven evaluation strategy. We presented a fairly simple and elegant formalism for the integration of reactive processes in this environment.
We first proposed a denotational semantics for the visual language, whose workflow and behaviour is not usual as compared to standard functional languages. This semantics relies 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 with reactive features. This extension combines well with the incremental building of patches, and 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, which allows to turn on or off the reactive behaviour of the different visual program components. 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. With these mechanisms, one can design arbitrarily processes reacting upon events, and embed OM visual programs in an interactive context.

References

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

[2] M. Puckette : Pure Data – Another Integrated Computer Music Environment, Second Intercollege Computer Music Concerts, Tachikawa, Japan, 1996.

[3] M. Laurson, J. Duthen : Patchwork, a Graphic Language in PreForm, International Computer Music Conference, Ohio State University, USA, 1989.

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

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

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

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

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

[9] J. Bresson, C. Agon, G. Assayag : OpenMusic – Visual Programming Environment for Music Composition, Analysis and Research, ACM MultiMedia – OpenSource Software Competition, Scottsdale, USA, 2011.

[10] M. Malt : Concepts et modèles, de l'imaginaire à l'écriture dans la composition assistée par ordinateur, Séminaire postdoctoral Musique, Instruments, Machines, Paris, France, 2003.

[11] P. D. Mosses : Handbook of Theoretical Computer Science, Vol. 2, Elsevier Science, 1990, Ch. Denotational Semantics, pp. 575–631.

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

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

[14] C. A. Gunter, P. D. Mosses, D. S. Scott : Semantic domains and denotational semantics, DAIMI Report Series 18 (276).

[15] P. Caspi, D. Pilaud, N. Halbwachs, J. A. Plaice : LUSTRE : a declarative language for real-time programming, ACM symposium on Principles of Programming Languages (POPL '87), 1987, 178–188.

[16] R. Bernhard : Esterel v4 : une extension modulaire d'Esterel. PhD thesis, Ecole des Mines de Paris, 1991.

[17] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, R. De Simone, The synchronous languages 12 years later, Proceedings of the IEEE 91 (1), 2003, 64–83.

[18] L. Mandel, M. Pouzet : ReactiveML, a Reactive Extension to ML, Principles and Practice of Declarative Programming (PPDP), 2005.

[19] F. Boussinot, J.-F. Susini : The SugarCubes tool box - a reactive Java framework, Software Practice and Experience, 28(14), 1998, pp. 1531–1550.