Reduction as a Transition Controller for Sound Synthesis Events

Jean Bresson, Raphaël Foulon, Marco Stroppa

Reduction as a Transition Controller for Sound Synthesis Events

Jean Bresson
UMR STMS – IRCAM/CNRS/UPMC
Paris, France
jean.bresson@ircam.fr

Raphaël Foulon
Sony CSL
Paris, France
foulon@csl.sony.fr

Marco Stroppa
University of Music and Performing Arts
Stuttgart, Germany
stroppa@mh-stuttgart.de

Abstract

We present an application of reduction and higher-order functions in a recent computer-aided composition project. Our objective is the generation of control data for the Chant sound synthesizer using OpenMusic (OM), a domain-specific visual programming environment based on Common Lisp. The system we present allows to compose sounds by combining synthesis events in sequences. After the definition of the compositional primitives determining these events, we handle their sequencing, transitions and possible overlapping/fusion using a special fold operator. The artistic context of this project is the production of the opera *Re Orso*, premiered in 2012 at the Opera Comique, Paris.

Categories and Subject Descriptors J.5 [*Computer Applications*]: Arts and Humanities—Performing arts; D.1.1 [*Software*]: Programming Techniques—Applicative (Functional) Programming; D.1.7 [*Software*]: Programming Techniques—Visual Programming

Keywords Computer-aided composition; Sound synthesis; Visual programming; Functional programming.

1. Introduction

Functional programming has had a strong influence in the development of music technology and compositional systems, from pioneering domain-specific languages such as Common Music [29], Arctic [10] or Haskore [13], to more recent projects such as Faust [20] or Euterpea [14]. Most current music programming environments today have a significant, more or less explicit functional orientation. Visual programming is also a common practice in contemporary music creation, be it for real-time signal processing with Max [21] or PureData [22], or in "symbolic" compositional contexts with Patchwork [18] and its descendants OpenMusic [4] and PWGL [19].

OpenMusic (OM) is a functional visual programming language based on Common Lisp, used by musicians to experiment, process and generate musical data [7]. This language has been used by contemporary music composers in the last fifteen years [1] and is today one of the main representatives of a branch in computer music systems called "computer-aided composition" [3]. Computer-aided composition is concerned with the formal development of abstract and complex compositional models embedding programs, data structures, notation and graphical representation in the conception of generative or transformational processes. This approach further emphasizes a high-level functional orientation in the specification and processing of harmonic material, of time structures, and more recently for the control of sound processing and spatialization [6].

OM connects with varied external sound synthesis and processing software, generally via command line or client-server communication, for which formatted control files or commands are to be generated from high-level compositional specifications [5]. In this paper we present a recent project carried out with the control of the Chant synthesizer [25], which puts forward the use and relevance of functional programming concepts.

Chant has been embedded in different musical software along its history [16, 26, 27] and recently reintegrated in compositional frameworks with OM-Chant [8], a library for the OpenMusic environment. As we will show further on, this synthesizer presents an original (pseudo-) continuous control paradigm, which highlighted the idea of synthesis sound *phrases* in the compositional processes. Our objective in this project was to extend the characteristics of this paradigm in order to achieve a level of musical expressivity required by composers working with this synthesizer. In particular, we developed a mechanism for sequence reduction allowing to consider the synthesis event's transitions as key elements in the construction of the sound. Our system is inspired by standard fold mechanisms and abstracts the handling of transitions from the main procedure generating the primitive events.

The motivation and main artistic context of this project was the production of the Opera *Re Orso*[1] by Marco Stroppa, who used the OM-Chant framework intensively in this work, and participated in its conception and development.

After a presentation of the synthesizer and its control in OM (Section 2), we will present strategies for dealing with transitions between synthesis events (Section 3), and how these strategies can be extended to the generation of phrases as higher-order functions using the sequence reduction mechanism (Section 4).

2. The Control of the Chant Synthesizer

Chant is a reference implementation of the FOF synthesis technique (*fonctions d'ondes formatiques*, or formant-wave functions [28]). This technique was originally designed to synthesize realistic singing voices, but is also used to generate varied other kinds of

[1] *Re-Orso*, by Marco Stroppa (musical assistant: Carlo Laurenzi), world premiered at the Opéra Comique, Paris on May, 19th, 2012 [23].

FARM '13, September 28, 2013, Boston, MA, USA.

http://dx.doi.org/10.1145/2505341.2505342

sounds. It consists in generating periodic impulses made of precisely enveloped sine waves, which produce the equivalent of vocal "formants" in the sound spectrum. The parameters of the formant-wave functions control the characteristics of the formants (central frequency, amplitude, band and skirt width) and the frequency of the FOF generator's impulses determines the fundamental frequency of the synthesized sound.

2.1 Control Paradigm

A synthesis "patch" in Chant is a configuration of several sound generation or processing units or "modules" (e.g. FOF generators, filters, noise generator, file reader) defined prior to every run of the synthesizer. The parameters controlling these units are all considered as functions f : *Time* $\rightarrow \mathbb{R}$ from time to single values (*Time* represents an interval on $\mathbb{R}^+$ defining the overall duration of the synthesis process). We call the "state" of the synthesizer at time t the set of parameter values $f_p(t)$ where f_p is the function associated to parameter p.

This state of the synthesizer is not set for every point in time: Chant performs an "off-line" rendering and systematically interpolates between successive user-specified values of f_p in order to compute $f_p(t)$ for for each parameter p and for all $t \in$ *Time*. A notable attribute of this synthesizer is therefore to enable the design of sound synthesis processes involving smooth continuous variations of its states and parameters.

From the composer point of view, this is a quite specific situation as compared to usual synthesis systems. As Risset pointed out [24], it corresponds to Laliberté's "archetype of voice" [17] (continuous control, as in bowed or wind instruments), where the performer is required to continuously attend to the note being produced, as opposed to the "archetype of percussion", where the note is specified only when striking it. The former allows for a greater expressivity and subtler phrasing, but it is monophonic, while the latter is polyphonic, since the performer's attention at playing a note can be immediately focused on the following ones. This conceptual distinction can also be found in the interface of sound synthesis systems: most of them are polyphonic "event-based" systems (e.g. Csound or MIDI-based synthesizers), where every event is independent and unconnected – or artificially connected – to the other ones. On the other hand, fewer systems such as Chant provide no means to express polyphony, but a better control on the overall phrase defined as a succession of connected events (this feature is extremely important when trying to synthesize vocal sounds).

2.2 OM-Chant and Synthesis Events

In a previous paper [9] we presented a framework for the structured control of Chant synthesis processes in OM, extending the OM-Chant library with the notion of *synthesis events*.[2] Different types of synthesis events are defined corresponding to the different modules and controls of a Chant synthesis patch: FOF banks, fundamental frequency, filter banks and noise generators. An event can be either a matrix of parameters (e.g. for the FOF banks: the different parameters of a number of parallel FOF generators) or a scalar controller (e.g. for a fundamental frequency controller) extended with temporal information (onset, duration). It embeds a set of timed values determining the evolution of one or several parameters during the event interval.

This framework allows the user to control sound synthesis processes combining "continuous" specification features with musical constructs defined as sequences of timed events, and draws an intermediate path where continuous control can be associated with the expressive specification of time structures.

[2] This idea and the concrete data structures used for the representation of events were inspired by earlier works on the OMChroma system [2].

2.3 From Events to Control Points

A sequence of events is specified by the user as an input musical structure. This sequence is then processed by OM-Chant to write a control file containing timed values determining the different functions f_p.[3] Each f_p is described by a time-ordered sequences of values (which we will call "control points" from here on), interpolated by the synthesizer to compute $f_p(t)$ for all $t \in$ *Time*.

We consider two principal cases: (1) events that specify a constant value for the parameter(s) and (2) events that represent variable or "continuous" controllers. In case 1 (constant value) OM-Chant duplicates the parameter value and generates control points at the temporal bounds of the interval (see Figure 1a). The parameter is therefore stable inside the event. In case 2 (continuous controller) the values can change at any time (and at any rate or precision) between the beginning and the end of the event (Figure 2a). The variations can come from specified modulations of an initial value (e.g. a vibrato or jitter effect as available in the OM-Chant library), or from other programmed or manually designed control.

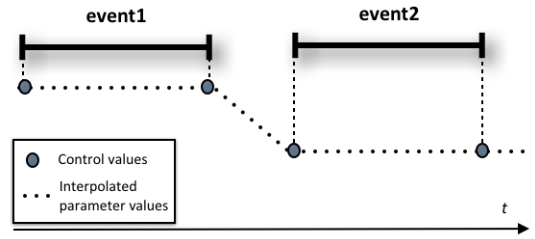
Under-specification in the input sequence (intra- or inter-event) is supported thanks to the synthesizer's interpolations (see Figures 1a and 2a).[4] Over-specifications however (when several events overlap or are superimposed – see Figures 1b and 2b) are not handled by the system and are the main focus of the present work.

2.4 Overlapping Events as Over-Specification

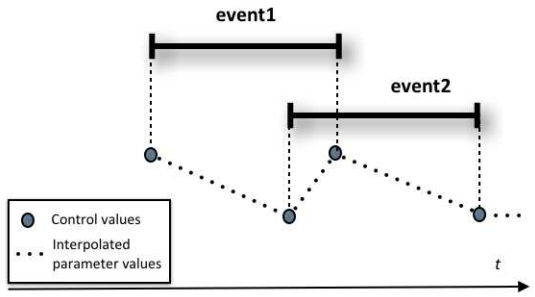
Over-specifications in the control of the synthesizer (when several events of a same kind overlap) could simply be detected as a mistake or contradiction in the parameters specification, and forbidden or corrected by the system. Technically this task amounts to de-

[3] Chant is then invoked by a command line call referring to this control file.

[4] Fade-in/fade-out options allow to generate silence between successive events: the activation of these attributes sets the amplitudes of the generators or filters to zero at a determined interval, respectively before and after the stated beginning/end times of the event – see [9].

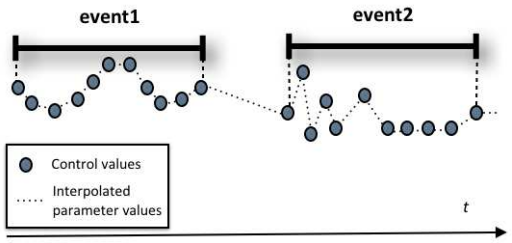


(a)

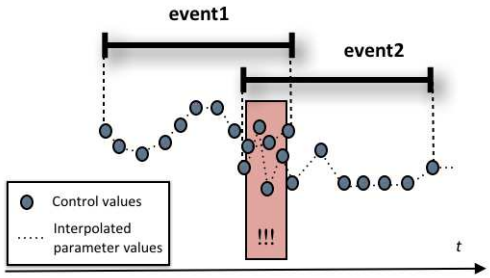


(b)

Figure 1: Events with constant values: control points and sampled/interpolated synthesis parameter (a) without overlapping and (b) with an overlapping interval.



(a)



(b)

Figure 2: Events with continuous values: control points and samples synthesis parameter (a) without overlapping and (b) with an overlapping interval.

termining rules to unambiguously define one or several function(s) from a time interval to the type of control values required by a number of parameters, starting from a configuration of input synthesis events.

In earlier work [9], a solution was proposed to handle the specific case of Figure 1b by interchanging the time of the control points at the end of event 1 and at the beginning of event 2. Then, the parameter values remain constant on the non-overlapping intervals, and an interpolation is performed by the synthesizer during the overlapping interval (an implementation for this mechanism is proposed in Section 3.2). In the case of continuous controllers (Figure 2b), overlapping time intervals are more problematic. The interpreter process (and the synthesizer) by default merge the two sequences of control points, and the resulting description of f_p loses continuity, or even becomes inconsistent if more than one value are specified for $f_p(t)$. To cope with this situation it is also possible for instance to play with the event's time properties, to cut or rescale the control-point sequences in order to make them fit within the non-overlapping intervals, or to establish priorities between superimposed events. However, these solutions may have a serious musical consequences, due to time distortions, poor transitions computed by linear interpolations or abrupt discontinuities in the parameters specification functions.

Numerous strategies can actually be envisaged to musically and efficiently handle these situations, and each synthesis parameter may require particular strategy and processing (we have shown in [11], for instance, how specific voice transitions – consonants – could be simulated by carefully shaping the frequency and amplitude evolutions of specific formants in the synthesis process). We will discuss in Section 3 a number of such strategies.

2.5 Overlapping Events as Implicit Polyphony

Overlapping or superimposition are common and intuitive polyphonic constructs that are very important in musical structures. From a compositional point of view, our issue is therefore not necessarily a problem. On the contrary, it is an exciting opportunity to explicitly control the intermediate states between successive events structuring a continuous control stream, and to grasp the notions of sequencing and *transitions* in the compositional processes.

It is important to realise how overlapping and polyphony are connected in music, and why this requires that a general solution be envisaged. Let us illustrate this with the case of the *portamento* as a particular example. In a normal *legato* singing voice, even if the notes written in score appear as separated events, the singer connects them by performing a short *portamento* between them (a *portamento* is a fast *glissando*, often with a half-sinusoidal shape). Albeit not notated in the score, the *portamento* is an essential feature of expressive singing. Practically, this means that the steady part of a note (which can be modulated with a vibrato or other kinds of effects) is shortened to leave place for the *portamento* without affecting the overall duration of the phrase. During this short phase (in the order of a few hundred milliseconds), no vibrato or modulations are performed on the note. A much longer *portamento* is called *glissando* and is usually notated in the score with a trait linking the two notes. In this case, the singer leaves the steady state earlier and ends the *glissando* at the beginning of the next note. It is however critical to grasp, that the difference between a *portamento* and a *glissando* is only a difference of time (where the process starts) and, perhaps, of shape. By no means it is a structural difference, even though the musical perception is dissimilar. Where the *portamento* exactly starts (before or at the end of the previous note) and ends (at or after the beginning of the following note) is a question of musical interpretation, which lies outside the purpose of this paper. Our goal here is to design a system that would allow for all kinds of *portamenti* and *glissandi* (to follow up with this particular example) to be implemented.

This point was an important part of the *Re Orso* project: the composer wanted to create and precisely control hybrid musical structures, made of a portamento lasting the whole duration of a musical event event (sometimes several minutes long) and including all sorts of modulations. One way to represent the exact place where this state ought to be computed, is to "visually" organise events during this phrase[5] and focus on the transitory states to perform important modulations (see Section 5).

3. Programming Transition Processes in OM

In the OM-Chant library tutorial, a recurring example (reproduced in Figure 3) presents a sequence of Chant events generated from a score. In this example the duration of the notes in the score at the top of the figure produce overlapping intervals.

For the sake of simplicity, we will consider here the events of type *ch-f0* in this sequence (the controller responsible for the fundamental frequency of the FOF synthesis process). A vibrato effect is applied to each *ch-f0* event in order to produce a more "realistic" voice sound. The resulting specification for $f_{\textit{fund-freq}}(t)$ merges contradictory values on the overlapping intervals. This is not a good musical interpretation of events overlapping, even though in this example the undesired behaviour can be masked by adequate control of the FOF generators.

In this section we will discuss a number of ad-hoc possibilities for controlling the transitions between two *ch-f0* events using the visual programming tools available in OM, with this example in mind. Section 4 will then propose a generalization of these possibilities and an application to longer event sequences.

[5] This approach was already present in the design of the Diphone interface to Chant [26], but transitions were limited to a linear interpolation between steady states.

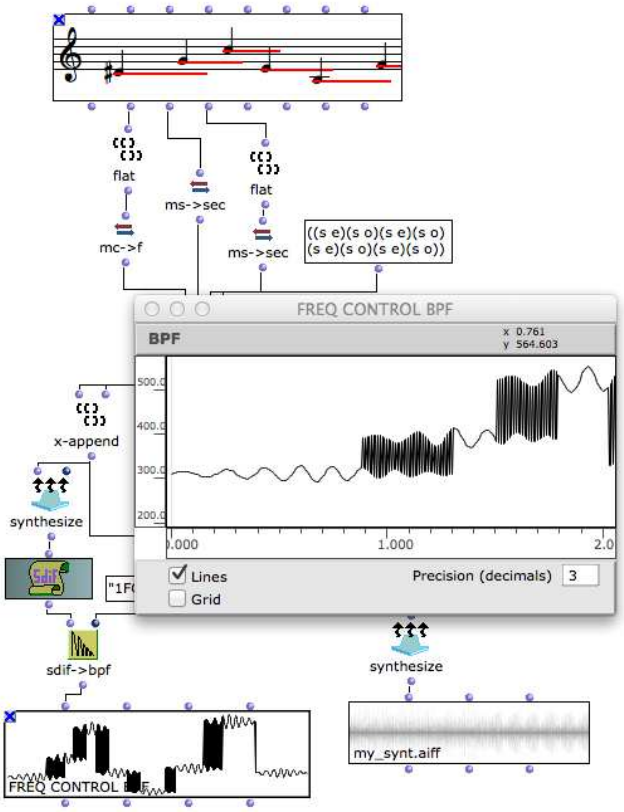


Figure 3: Control of a Chant sound synthesis process from a sequence of events in OM-Chant. The sequence of events is derived from the score at the top of the figure. The curve corresponding to the mereged fundamental frequency controller is visible at the bottom-left, and the editor window open at the center of the figure displays a detail on this curve.

3.1 Visual Programming in OM: A Quick Introduction

An OM visual program is a directed acyclic graph made of boxes and connections. Boxes represent function calls: they use upstream connected values as arguments (inputs are always situated at the top of a box) and yield computed values to their output connections (outlets are at the bottom of the boxes). Arcs (or connections) connect these boxes together and define the functional composition of a visual program. The visual programs are evaluated *on-demand* at specific downstream nodes of the graph, triggering a chain of calls to upstream-connected boxes, and recursively set new values to the evaluated boxes.

Some special boxes are class instance generators implementing local state features and allowing to store, visualize and edit musical objects and data. In Figure 3 for instance, we can see a score object box at the top (input of the program), and two other objects at the bottom (results): a *BPF* (for "break-point function") containing the sequence of fundamental frequency control points as computed by OM-Chant, and a sound file (produced by the synthesis process). The editor window corresponding to the BPF box is visible at the middle of the figure.

The other boxes in the figure, e.g. *flat* (flattening of a parameter list), *ms→sec* (time conversion from milliseconds to seconds), *synthesize* (external call to the Chant synthesizer), *sdif→bpf* (conversion of formatted synthesis parameters to the *BPF* object), are simple functions used to process and transform data in the program.

3.2 Transition (a): Modification of the Time Intervals

A first possible solution to deal with overlapping events consists in modifying the temporal intervals of the synthesis events. With simple arithmetic operations and a little bit of logic, the onset and duration of the events can be changed to fit within the non-overlapping intervals (see Figure 4).

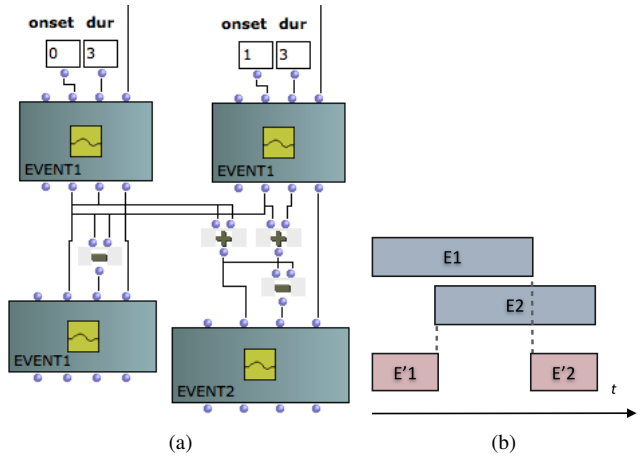


Figure 4: Modification of the temporal attributes of the Chant events. (a) Implementation in OM. (b) Representation of the time intervals.

In the case of continuous values, the contents of the events might need to be modified to fit within the new durations. Different tools can be used for this purpose, such as *bpf-scale* (rescales a curve to a specific duration) or *bpf-extract* (cuts a selected part of a curve). In some cases these tools can be sufficient to design the transition. In the previous example however (Figure 3), the vibrato modulation applied to the fundamental frequencies would be shrunk (and therefore accelerated) or abruptly cut to fit in the new intervals, which is not an acceptable solution.[6]

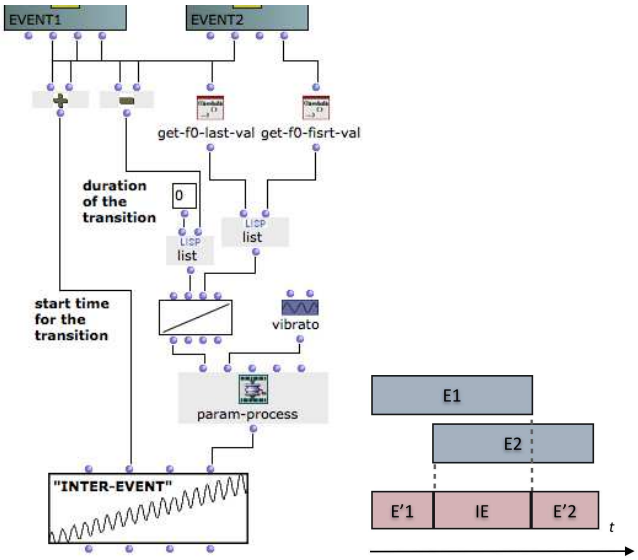
The main problem in this solution is actually that the synthesis parameters remain explicitly controlled only during the non-overlapping intervals. An alternative solution could be to modify the duration of only one of the events, and to transform the values in the other one so as to implement a particular transition process. Other options are proposed below.

3.3 Transition (b): Instantiation of the Transition

In order to precisely control the behaviour of the synthesizer during the transition, it is possible, as a complement to the time intervals modification, to instantiate a third intermediate event localized in this transition interval and implementing the evolution of the different parameters. In our fundamental frequency example, this event could implement a vibrato controller taking into account the values of the two surrounding events (see Figure 5).[7]

[6] The application of time modifications to a vibrato is a well-known issue in computer music [12]. A better solution to this particular case is to apply the vibrato modulation at a later stage of the process, when issues of time intervals are solved. This implies taking this constraint into account at the time of creating the events and passing the vibrato parameters downstream.

[7] Note that this solution still remains problematic in this specific example for it separates the transition from the steady states and can generate phase discontinuities at the limit of the corresponding intervals. Here also, when the initial pitch values are constant, delaying the application of the vibrato to a later stage in the process sensibly improves the results.



(a) (b)

Figure 5: Generation of an intermediate event implementing a vibrato (continuation from the resulting events of Figure 4). (a) Implementation in OM. (b) Representation of the time intervals.

Tools have been developed to help for an easier generation of intermediate events, including in the case of more complex, matrix-based synthesis events. The function *gen-inter-event* :

$Event \times Event \times (\mathbb{R} \times \mathbb{R} \rightarrow Time \rightarrow \mathbb{R})^* \rightarrow Event$ generates a new event with adequate time properties from a pair of events of a same type and an optional set of rules. By default, the generated event parameters are linear interpolations from the end value(s) of the first event to the start value(s) of the second one. Additional rules can then be added for the setting of targeted parameters using static or functional specifications (see Figure 6).

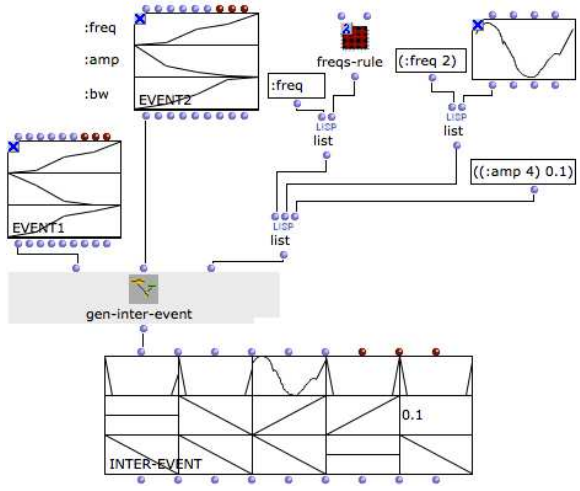


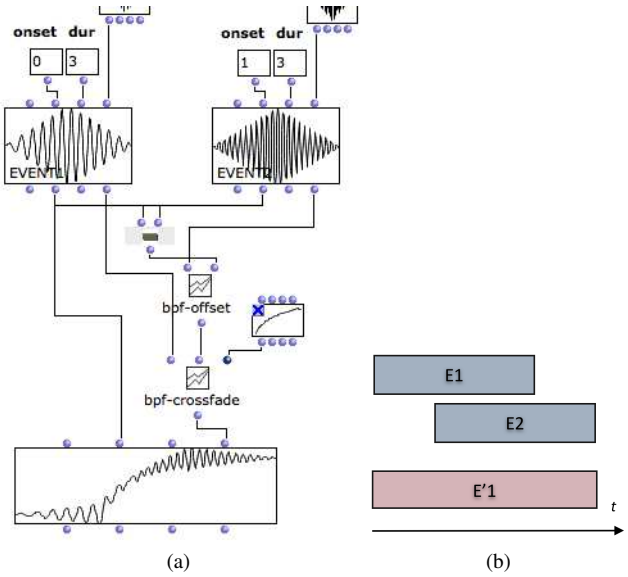
Figure 6: Using *gen-inter-event* to compute an intermediate event between two FOF events. Three rules are applied. (From right to left:) the amplitude of the fifth formant (*:amp 4*) during the transition is set to 0.1.; the transition for the frequency of the third formant (*:freq 2*) follows a specific hand-drawn curve; and the other frequency transitions (*:freq*) are defined in the "freqs-rule" box.

This example is a first demonstration of higher-order functions in OM: the box labelled "freqs-rule" in Figure 6 is an abstraction (it contains another visual program). Moreover, it is in a special "state" denoted by a small λ icon, which turns it into a lambda expression in the evaluation of the main visual program.[8] This box can therefore be interchanged with any other function of two arguments (begin, end) defining the evolution of a parameter value.

Experiments have been carried out using this mechanism for the setting of FOF parameters in the case of voice sound synthesis and the simulation of consonants [11]. Specific data structures were added to the OM-Chant objects library to substitute the set of rules with more compact/graphical transition "profiles" applying to the different parameters. A given transition profile can then match any pair of successive events of a same kind and produce the corresponding intermediate event(s). The creation of databases with these objects allowed to define transition dictionaries (virtual "singers") reusable in different contexts.

3.4 Transition (c): Merging the Events

Another type of structural modification which can be chosen to handle overlapping events and transitions is to merge the successive events, and replace them with a single one where continuity is easier to control and maintain. The OM function *bpf-crossfade* is of a particular help in this kind of process: as shown in Figure 7, two localized and modulated events can produce a new one with a fairly good continuity.



(a) (b)

Figure 7: Crossfading synthesis events. (a) Implementation in OM. (b) Representation of the time intervals.

Another example is proposed in [11] with the idea of "FOF morphing". In this case, two or three overlapping or superimposed events, plus a 2- or 3-D morphing profile were combined to produce a single hybrid structure on the total time interval.

[8] OM fully supports higher-order functions and provides easy ways to turn functional components or programs into lambda expressions (or anonymous functions) to be used in other visual programs. See [7].

4. Generalization using Higher-Order Functions

The previous examples were provided as an attempt to illustrate some possible directions for the implementation of a transition process as a function of two successive events. In fact, there exist numerous ways of dealing with overlapping and superimposition of events, depending on particular situations, technical or musical requirements, underlying data and synthesis patches. As one can imagine, the implementation of transitions can quickly turn into complex problems, and be subject to a number of obstacles, such as:

Scalability: if the transition programming strategy has to be applied to "real" (longer) sequences;

Modularity: since the (visual) code responsible for the transition is currently interleaved with the event generation process.

4.1 General principles

In order to address the previous issues we use a system based on the concept of sequence reduction (or *fold*). A fold operates on a recursive data structure (e.g. a list or a tree), and builds up a return value by combining the results of the recursive application of a function onto the elements of this structure [15]. In our case, the folding function is combined with a concatenation. We defined an operator called *ch-transitions* :

$Event^* \times (Event \times Event \rightarrow Event^*) \rightarrow Event^*$, which performs a left-fold process applying on a list of Chant events and producing a new list of events. In this process a transition-control function $Event \times Event \rightarrow Event^*$ is applied to the successive elements of the sequence, each time pairing the last element of the current result and the next element of the processed list of events, in order to substitute or append a new event (or set of events) at the end of the result. This function can be any of the previous transition strategies turned into a lambda expression, which transforms two events into a new sequence of events (of a variable length). Applied in the reduction process, this new sequence will iteratively replace the current head in the result sequence.

Figure 8 shows the stepwise process operated by *ch-transition* using the three types of transition control described in Sections 3.2, 3.3 and 3.4. In the first case (a) the two events processed by the transition function are converted into two new events with modified time intervals. In (b) three events are generated for each transition and replace the last element in the result sequence (note that only the last of these three is used for the next transition). In (c) the transition produces a single event: in the result sequence, one single event is recursively used and extended to integrate the successive events of the original sequence.

A simplified Common Lisp implementation of this mechanism is listed below:[9]

```
(defun ch-transitions (init-seq transition-function)
  ; the head of the result is initialised with
  ; the first element in <init-seq>
  (let ((result (car init-seq)))
    (mapcar
      ; for each successive element the transition-function
      ; is applied and merged with the head of the result
      '(lambda (event)
        (setq result (append
                      (butlast result)
                      (funcall transition-function
                               (last result)
                               event))))
    (cdr init-seq))
  ))
```

[9] For simplicity and readability in the code listing, the bits of code handling the input and return types of the transition function and their combination with the result sequence have been omitted.

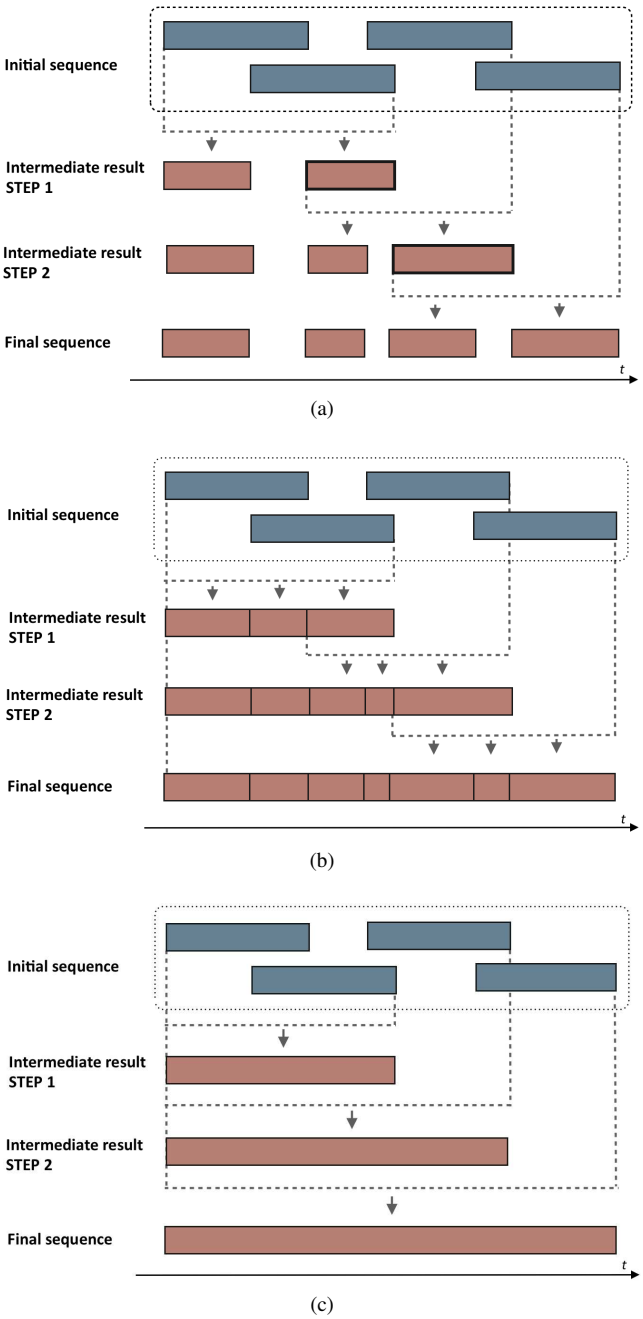


Figure 8: Illustration of the *ch-transitions* left-fold processing with a sequence of 4 events using several transition-control strategies. (a) Modification of the time intervals. (b) Instantiation of the transition. (c) Merging the events.

4.2 Application in OM-Chant Synthesis Processes

In Figure 9 the example from Figure 3 is now extended and includes the *ch-transitions* processing between the list of events generated out of the initial chord sequence, and the Chant synthesis process. The transition-control function (labelled *crossfade-transitions*) is an abstraction box in the "lambda" mode. Its contents is visible in the window at the right of the figure.

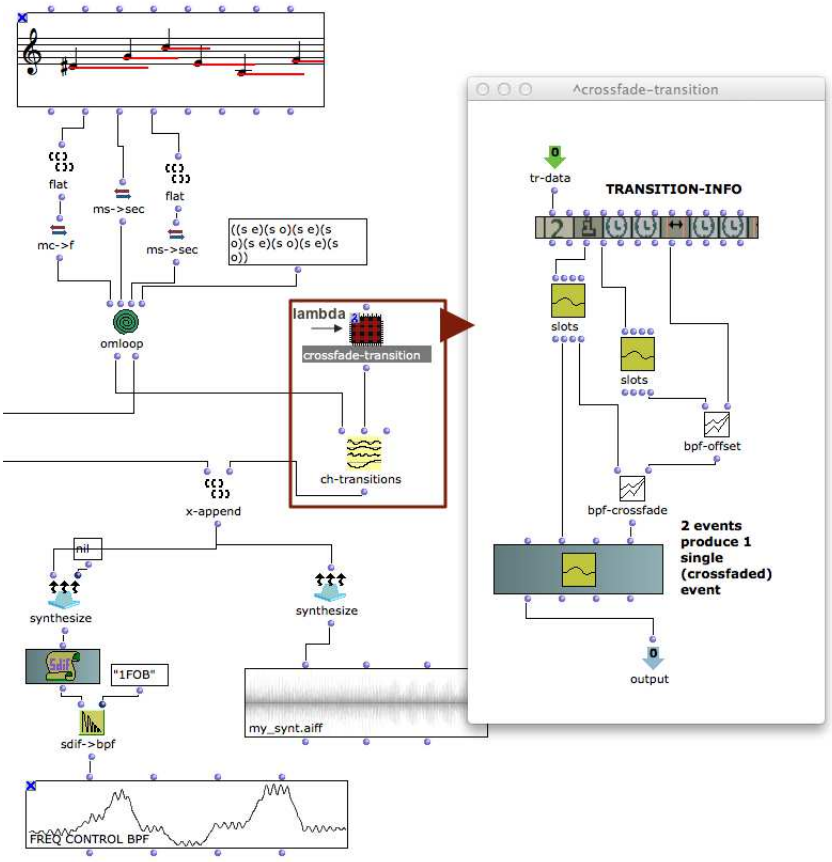


Figure 9: Extending the Chant synthesis patch from Figure 3 with the transition-control mechanism.

We can note that *crossfade-transitions* has only one argument (or input): in order to make explicit some of their relations and context to the user, a structure called *transition-info* replaces the two actual input events. The *transition-info* is instantiated using a standard box in the transition-function visual program. It provides a direct reading access to the main data related to a transition (internal values, temporal information of the events) and its external context; in particular:

- The (full) initial sequence;
- The two events in the current transition;
- The position of the transition in the original sequence (this is a useful information allowing to implement dynamic transition controllers varying along the processing of the sequence);
- The temporal information: onset and end times of the events, duration of the non-overlapping / overlapping intervals.

In this example (Figure 9) the transition-control consists of a cross-fade between the *ch-f0* events: one "merged" event is produced for each pair of events in the fold mechanism, which corresponds to case (c) in Figure 8.

An inspector window can be displayed at calling *ch-transitions* and allows to visualize and debug the sequence processing (see Figure 10).

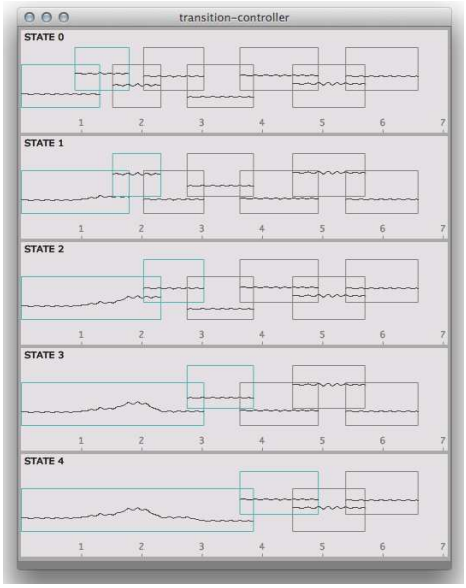


Figure 10: Diplaying/debugging the *ch-transitions* process from Figure 9 using the inspector window. In this example the fundamental frequency curves of the successive events are recursively merged with the beginning, as in Figure 8c. The vertical shifts in the events display is for visibility and has no specific signification.

5. Application in *Re Orso*

Commissioned by Ircam, the Ensemble Intercontemporain, the Opera Comique in Paris and La Monnaie in Brussels, *Re Orso* (2012) is an opera merging acoustic instruments and electronics. Voice and electronics are both key elements in the dramaturgy and the composition of the work, and a major attention has been paid to their expressive connection and integration.

OpenMusic, and OM-Chant in particular, played an important role in this project, both for the composition of the score and the generation of some synthetic sounds that the composer calls "imaginary voices".

One of the most spectacular usages of the OM-Chant controlled transition process in this work is the "death of the king". This passage, which lasts about 1'30", is a morphing between a synthetic countertenor voice (imitating the dying king's voice and surreptitiously "sneaking" into the singer's real voice) and the ominous sound of a knell. FOF events are progressively transformed from the 5 formants corresponding to a sequence of sung vowels to the numerous and narrow harmonics (or "partials" in terms of spectral analysis) of a bell. The fundamental frequency is a glissando from the original sung pitch (D5) to sub-audio frequencies, which are eventually perceived as a sequence of pulses exciting the bell resonators. During this process, the bandwith of the formants gets gradually narrower, which provokes an increasingly longer resonance.

The precise tuning of the synthesis parameters and the joint controlled transitions in this overall process, crucial to the generation of a musical and captivating result, is performed in an OM patch whose structure is similar to the one seen in the previous example (see Figure 11).[10]

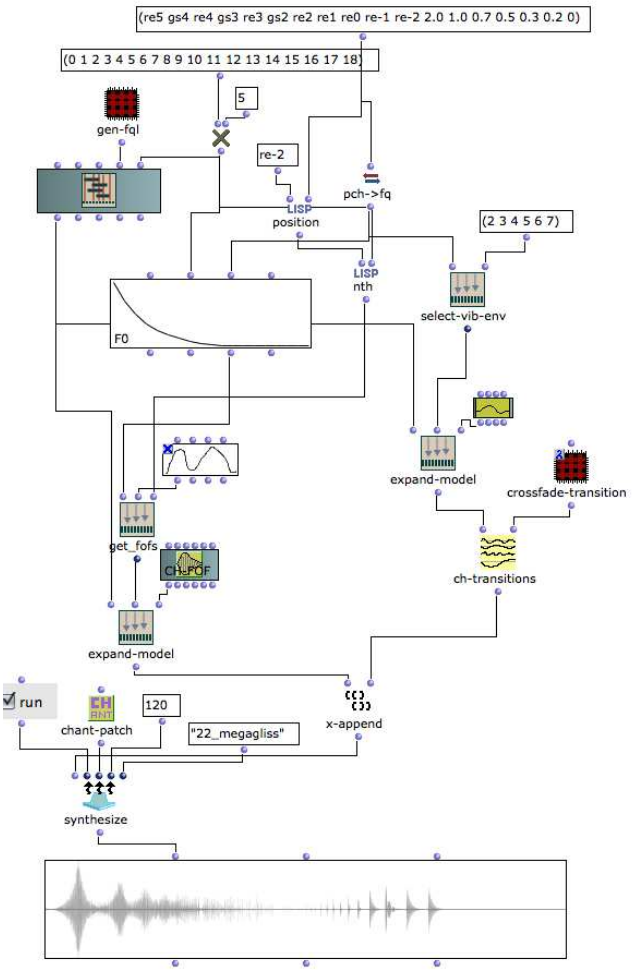


Figure 11: OM patch implementing the sound synthesis process in *Re Orso*'s death of the king passage.

6. Conclusion

We presented a system for the generalised control of transitions between synthesis events in the OM-Chant library, based on higher-order functions and fold mechanisms. The proposed framework provides a powerful and flexible way to deal with computer-generated sequences of control events for the Chant synthesizer. It allows to maintain both a high-level abstraction in the creation of sequences and time structures, and a precise control of the continuous aspects of the internal or inter-event parameter values.

This framework is available in OM-Chant 2.0.[11]

Acknowledgments

This work was partially funded by the French National Research Agency with reference ANR-12-CORD-0009.

References

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

[2] C. Agon, J. Bresson and M. Stroppa. OMChroma: Compositional Control of Sound Synthesis. *Computer Music Journal*, 35(2), 2011.

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

[4] 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), 1999.

[10] Sound extracts available at `http://repmus.ircam.fr/bresson/projects/om-chant/examples`.

[11] Distributed by Ircam, see `http://forumnet.ircam.fr` – A user manual is available at `http://support.ircam.fr/docs/om-libraries/om-chant/`.

[5] J. Bresson. Sound Processing in OpenMusic. In *Proceedings of the International Conference on Digital Audio Effects - DAFx-06*, Montral, 2006.

[6] J. Bresson and C. Agon. Musical Representation of Sound in Computer-Aided Composition: A Visual Programming Framework. *Journal of New Music Research*, 36(4), 2007.

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

[8] J. Bresson and R. Michon. Implémentations et contrôle du synthétiseur CHANT dans OpenMusic. In *Actes des Journées d'Informatique Musicale*, Saint-Etienne, 2011.

[9] J. Bresson and M. Stroppa. The Control of the Chant Synthesizer in OpenMusic: Modelling Continuous Aspects in Sound Synthesis. In *Proceedings of the International Computer Music Conference*, Huddersfield, 2011.

[10] R. Dannenberg, P. McAvinney and D. Rubine. Arctic: A Functional Language for Real-Time Systems. *Computer Music Journal*, 10(4), 1986.

[11] R. Foulon and J. Bresson. Un modèle de contrôle pour la synthèse par fonctions d'ondes formantiques avec OM-Chant. In *Actes des Journées d'Informatique Musicale*, Paris, 2013.

[12] H. Honing. The Vibrato Problem: Comparing Two Solutions. *Computer Music Journal*, 19(3), 1995.

Figure 12: *Re Orso* at the Opera Comique, Paris (May, 2012). Photo: Elisabeth Carecchio

[13] P. Hudak, T. Makucevich, S. Gadde and B. Whong. Haskore Music Notation – An Algebra of Music. *Journal of Functional Programming*, 6(3), 1996.

[14] P. Hudak. *The Haskell School of Music – From Signals to Symphonies*. 2013.

[15] G. Hutton. A Tutorial on the Universality and Expressiveness of Fold. *Journal of Functional Programming*, 9 (4), 1999.

[16] F. Iovino, M. Laurson and L. Pottier. *PW-Chant Reference*. Paris: Ircam, 1994.

[17] M. Laliberté. Archétypes et paradoxes des nouveaux instruments. In *Les nouveaux gestes de la musique*, Marseille : Parenthèses, 1999.

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

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

[20] Y. Orlarey, D. Fober and S. Letz. Faust: an Efficient Functional Approach to DSP Programming. In G. Assayag and A. Gerzso (Eds.) *New Computational Paradigms for Computer Music*, Delatour France / Ircam, 2009.

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

[22] M. Puckette. Pure Data: Another Integrated Computer Music Environment. In *Proceedings of the Second Intercollege Computer Music Concerts*, Tachikawa, 1996.

[23] Re Orso, 2013. Retrieved July 25, 2013, from Opéra Comique: http://www.opera-comique.com/fr/saisons/saison-2011-2012/mai/re-orso.

[24] J.-C. Risset. Le timbre. In J-J. Nattiez (Ed.) *Musiques, Une encyclopédie pour le XXIe siècle. Les savoirs musicaux.* Arles: Actes Sud/Cité de la musique, 2004.

[25] X. Rodet. Time-domain Formant-wave Function Synthesis. *Computer Music Journal*, 8(3), 1984.

[26] X. Rodet and A. Lefevre. The Diphone Program: New Features, New Synthesis Methods and Experience of Musical Use. In *Proceedings of the International Computer Music Conference*, Thessaloniki, 1997.

[27] X. Rodet and P. Cointe. Formes: Compostion and Scheduling of Processes. *Computer Music Journal*, 8(3), 1984.

[28] X. Rodet, Y. Potard and J.-B. Barrière. The CHANT Project: From the Synthesis of the Singing Voice to Synthesis in General, *Computer Music Journal*, 8(3), 1984.

[29] H. Taube. Common Music: A Music Composition Language in Common Lisp and CLOS. *Computer Music Journal*, 15(2), 1991.