Programming Mixed Music in ReactiveML

Guillaume Baudart
École normale supérieure de Cachan
Antenne de Bretagne
DI, École normale supérieure
Guillaume.Baudart@ens-cachan.org

Louis Mandel
Univ. Paris-Sud 11
DI, École normale supérieure
INRIA Paris-Rocquencourt
Louis.Mandel@lri.fr

Marc Pouzet
Univ. Pierre et Marie Curie
DI, École normale supérieure
INRIA Paris-Rocquencourt
Marc.Pouzet@ens.fr

Abstract

Mixed music is about live musicians interacting with electronic parts which are controlled by a computer during the performance. It allows composers to use and combine traditional instruments with complex synthesized sounds and other electronic devices. There are several languages dedicated to the writing of mixed music scores. Among them, the Antescofo language coupled with an advanced score follower allows a composer to manage the reactive aspects of musical performances: how electronic parts interact with a musician. However these domain specific languages do not offer the expressiveness of functional programming.

We embed the Antescofo language in a reactive functional programming language, ReactiveML. This approach offers to the composer recursion, higher order, inductive types, as well as a simple way to program complex reactive behaviors thanks to the synchronous model of concurrency on which ReactiveML is built. This article presents how to program mixed music in ReactiveML through several examples.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Concurrent, distributed, parallel languages; H.5.5 [*Information Interfaces and Presentation*]: Sound and Music Computing.

Keywords Synchronous Programming; Language Embedding; Mixed Music; Live Coding.

1. Introduction

Technical progresses since the 1950's have led composers to write music mixing live performers with electronic parts. At first, performers simply followed a pre-registered magnetic band. But the advances in computing rapidly allowed real interaction between musicians and electronic parts.

Developed at IRCAM, Antescofo [5][1] is a state-of-the-art score following system dedicated to mixed music. Since 2008, Antescofo has been used in the creation of more than 40 original mixed electronic pieces by world renowned artists and ensembles, including Pierre Boulez, Philippe Manoury, Marco Stroppa, New York Philharmonics, Berlin Philharmonics, and the Radio France Orchestra.

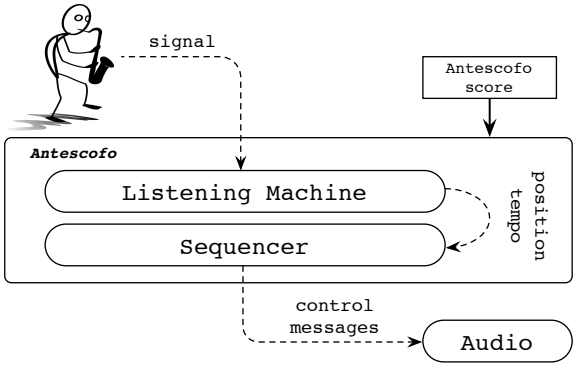


Figure 1. Architecture of the Antescofo system. Continuous arrow represent pre-treatment, and dotted ones real-time communications

Figure 1 describes the organization of the Antescofo system. It is composed of two distinct subsystems: a listening machine and a sequencer. During a performance, the *listening machine* estimates the *tempo* (i.e., execution speed) and the position of the live performers in the score. The role of the *sequencer* is to use these information to trigger electronic actions by sending control messages to a music programming environment. e.g., Max/MSP.[2] Then, the environment uses these messages to handle complex sound synthesis, manage lights, etc.

The language of Antescofo [9] is a descriptive language inspired by classical western music notation: a score is an interleaving of instrumental notes and electronic actions. An original aspect of this language is that it specifies synchronization and error-handling strategies for the electronic actions. It means that during the performance, depending on the tempo variations and the errors made by the musician or the listening machine, the sequencer will react in a way which is defined by the composer.

The sequencer of Antescofo is a *synchronous* reactive system: it continuously reads inputs from the listening machine, produces outputs to the musical environment and is subject to the strong timing requirement related to ear tolerance (typically 30 ms). Synchronous languages [2] have been invented and are used widely[3] for programming a wide range of critical control software: on board control of trains and track interlocking, fly-by-wire and cockpit displays for planes, etc. Therefore, they would have been well suited for programming the sequencer. But, to ensure statically that programs execute in bounded time and memory, synchronous lan-

[1] http://repmus.ircam.fr/antescofo

[2] http://cycling74.com/

[3] http://www.esterel-technologies.com

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

http://dx.doi.org/10.1145/2505341.2505344

guages like Lustre and Esterel intentionally forbid the dynamic creation of processes which is a feature of the Antescofo language. To overcome this limitation, we have implemented an Antescofo sequencer in ReactiveML [1], an extension of the functional language OCaml[4] with synchronous parallelism. Note that we are interested in the language and the sequencer, the link with live musicians still relies on the actual Antescofo listening machine.

Contribution of the Paper The main contribution of this paper is to show, through a collection of examples, the benefit of the embedding of Antescofo in ReactiveML for programming mixed music. This embedding combines the expressiveness of the two. It facilitates the creation process for the composer by using higher-order combinators. It can be used to prototype new programming constructs for Antescofo. In all our experiments, response times were less than the reaction time of human ear, using the current version of the ReactiveML compiler and run-time.

The paper is organized as follows. Section 2 presents ReactiveML through a simple library for electronic music. In Section 3, we introduce the main features of the Antescofo language. Then, Section 4 illustrates our approach on a complete example. In Section 5, we show how this work can be used to design new kinds of interactions between a live performer and electronic parts. Related work is discussed in Section 6 and we conclude on Section 7.

The code presented in the article is available on a companion web site: http://reactiveml.org/farm13. Links to music or video demos are indicated with the symbol ♫.

2. Music in ReactiveML

ReactiveML [17] is a synchronous reactive language built as an extension of the functional programming language OCaml. Therefore, following the work of Hudak [12] we can easily define musical structures, in a functional programming style.

This section presents the language ReactiveML and its concurrency model through a simple music library.

2.1 Music data types

Traditionally, in western music, a melody is a sequence of *notes*, where a note is a sound event characterized by a pitch and a duration. Notes can be defined with the following types.

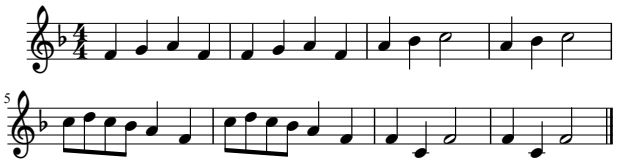
```
type pitch_class =
  | A  | B  | C  | D  | E  | F  | G
  | As | Bs | Cs | Ds | Es | Fs | Gs
  | Af | Bf | Cf | Df | Ef | Ff | Gf

type octave = int
type pitch = pitch_class * octave
type dur = float
type note = dur * pitch
```

The pitch is represented by a pair `(pitch_class, octave)`, where `pitch_class` denotes one of the twelve semi-tones, e.g., A, A#, A♭, B, B#..., and `octave` is an integer. For instance `(A,4)` denotes the famous A 440Hz. Here, as in Hudak's work, `As` stands for A sharp and `Af` for A flat.

A classic melody is just a sequence, or list, of notes. For instance, Figure 2 presents the theme of a traditional French musical round: *Frère Jacques* and the beginning of its representation in our data structure.

It is often convenient to treat pitches as integers. Functions `int_of_pitch` and `pitch_of_int` convert a symbolic pitch, e.g., `(A,4)` to an integer and vice versa (one can, for example, associate the corresponding MIDI note, e.g., A4 = 69).

[4] http://caml.inria.fr



```
let jacques =
  [ 1.0, (F,4); 1.0, (G,4); 1.0, (A,4); 1.0, (F,4);
    1.0, (F,4); 1.0, (G,4); 1.0, (A,4); 1.0, (F,4);
    1.0, (A,4); 1.0, (Bf,4); 2.0, (C,5); ... ]
```

Figure 2. The traditional French musical round *Frère Jacques* and the beginning of its representation in our data structures.

Now that we can represent musical structures, we write functions to manipulate them. For instance, the following defines functions to transpose a sequence.[5]

```
let move_pitch inter p =
  pitch_of_int ((int_of_pitch p) + inter)
val move_pitch: int -> pitch -> pitch

let transpose inter sequence =
  List.map
    (fun (d, p) -> d, (move_pitch inter p))
    sequence
val transpose: int -> note list -> note list
```

The function `move_pitch` transposes a symbolic pitch by an interval `inter` in semi-tones. To transpose an entire melody, we use the higher-order function `List.map` from the standard OCaml library that applies `move_pitch` to every note in a sequence.

2.2 Time in ReactiveML

ReactiveML is a synchronous language based on the reactive model introduced by Boussinot [4]. It relies on the notion of a global logical time. Time is viewed as a sequence of logical instants. Regular OCaml functions are considered to be instantaneous, i.e., when called it returns its result in the very same instant. In addition, we can define *processes* that may last over several logical instants. It is introduced by the keyword `process`.

To start with, consider the programming of a simple countdown. The following program awaits a given amount of time `value` given in seconds.

```
let process countdown value =
  let deadline = Unix.gettimeofday () +. value in
  while Unix.gettimeofday () < deadline do
    pause
  done
val countdown: float -> unit process
```

First, the date of the deadline is computed using the function `gettimeofday` of the `Unix` module[6]. Then, while the current date if less than the deadline, the process `countdown` awaits the next logical instant (`pause`).

Communications between processes are made through signals that are values characterized by a status defined at every logical instant: present or absent. For instance, the following process implement a standard timer similar to the Unix timer. It awaits a first

[5] The type of the functions given in *italic* can be less general than the one inferred by the compiler.

[6] `+.`, `*.`, `/.` are floating-point addition, multiplication and division operators.

duration `value`, and then periodically emits the signal `alarm` with a period `interval`.[7]

```
let rec process timer value interval alarm =
  run (countdown value);
  emit alarm ();
  run (timer interval interval alarm)
val timer:
  float -> float -> (unit, unit) event ->
    unit process
```

The process `timer` is the interface between physical time and synchronous logical time. This process is non-deterministic: the number of logical instants between two emissions of the signal `alarm` is not necessarily constant. Yet, if the duration of every logical instant is much smaller than `interval`, the error between two emission of the signal `alarm` will be negligible with respect to the reaction time of the human ear, i.e., 30 ms [10]. The important point is that we have isolated the source of non-determinism in the program. Now, it is possible to express delays with respect to a signal which is supposed to be periodic.

In the following, `alarm` denotes a global signal. It will be generated by the process `timer` described above, with period `period` defined as a global constant. Using this signal, one can write a process that does nothing but wait for a duration `dur`. For simplicity, we assume in this section that all durations are expressed in seconds:

```
let process wait dur =
  let d = int_of_float (dur /. period) in
  for i = 1 to d do
    await alarm
  done
val wait: dur -> unit process
```

This process waits for `d` occurrences of the signal `alarm` where `d` is the duration `dur` expressed as a number of instants.

In ReactiveML, signals can also carry values. Different values can be emitted during an instant, which is termed multi-emission. In this case, when the signal is declared, a function defining how to combine the multiple emissions must be provided. Here, we declare a signal `perf` which accumulates in a list the multiple simultaneous emissions.

```
signal perf default [] gather (fun x y -> x::y)
val perf: (note, note list) event
```

The default value is set to the empty list, `[]`, and the gathering function `(fun x y -> x::y)` appends the value `x` to the head of the list `y`. This function is used to fold all the values emitted during an instant into a list, starting from the default value.

Now, we write a process `play` which iteratively emits the notes from a sequence `sequence` on the signal `perf`.

```
let rec process play sequence perf =
  match sequence with
  | [] -> ()
  | ((dur, pitch) as note) :: s ->
      emit perf note;
      run (wait dur);
      run (play s)
val play:
  note list -> (note, note list) event ->
    unit process
```

[7] In order to avoid time shifting and floating point errors, the actual implementation is slightly different.

To play a note, the program sends it on the signal `perf`. Then, it waits for the delay corresponding to the note duration before sending the next note.

To produce sound, notes emitted on signal `perf` are sent to the audio environment.

```
let process sender perf =
  loop
    await perf (notes) in
    List.iter (fun n -> send_to_audio n) notes
  end
val sender: (note, note list) event -> unit process
```

This process loops infinitely: it waits for an emission on signal `perf` and sends all received notes to the audio environment, by calling the function `send_to_audio`.

A musical performance is the parallel composition of the two previous processes.♫

2.3 Parallel and Sequential Execution

Thanks to the notion of global logical time inherited from the synchronous model, it is easy to combine processes. Two expressions can be evaluated in sequence (`e1; e2`) or in parallel (`e1 || e2`). In the latter case, they execute synchronously (or *lockstep*). Therefore, they remain synchronous during an execution.

Imagine we want to double a voice one third higher (i.e., four semi-tones). Of course, it is possible to write a function that takes a list of notes and returns a new list of notes containing the two voices. But, we can also use the deterministic parallel execution provided by ReactiveML to play in parallel the voice and its transposition. This construct will be very useful to combine the musical performance with other processes (see for instance Section 5.2).

```
let process double sequence =
  run (play sequence) ||
  run (play (transpose 4 sequence))
val double: note list -> unit process
```

In ReactiveML, processes are first class citizens. Thus, it is possible to write higher order processes. For instance, the following code delays the execution of a process `p` by a duration in seconds.

```
let process delayed p dur =
  run (wait dur);
  run p
val delayed: unit process -> dur -> unit process
```

Finally processes can be iterated with the classical constructs: `loop`/`end` for an infinite loop, `for`/`do`/`done` for a finite sequential loop and `for`/`dopar`/`done` for a finite parallel loop.

With these constructs, we are able to execute the round presented in Figure 2.♫ Four different voices play the theme in loop. Each voice starts two measures, i.e., 8s after the previous one.

```
let process theme =
  loop
    run (play jacques)
  end
val theme: unit process

let process round =
  run theme ||
  run (delayed theme 8.0) ||
  run (delayed theme 16.0) ||
  run (delayed theme 24.0)
val round: unit process
```

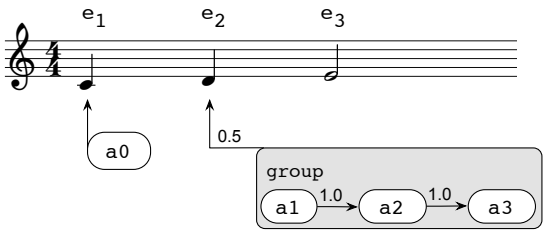


Figure 3. Representation of an Antescofo score. Musical notes correspond to the musician's part and the rest to electronic actions.

This process can also be written with a parallel loop:

```
let process round =
  for i = 0 to 3 dopar
    run (delayed theme (float (i*8)))
  done
val round: unit process
```

or, using a recursive process taking the number of voices as an argument.

```
let rec process round nb_voices =
  if nb_voices <= 0 then ()
  else begin
    run theme ||
    run (delayed (round (nb_voices - 1)) 8.0)
  end
val round: int -> unit process
```

We are now able to describe and execute electronic music in ReactiveML. But what about mixed music?

3. Toward Mixed Music

Mixed music is about interaction between a live musician and electronic parts.♫ The Antescofo language [9] allows a composer to specify electronic parts and how they interact with musicians during a performance. This language is the result of a close collaboration between composers and computer scientists. Figure 3 shows a graphical representation of a very simple Antescofo score.

3.1 Relative Time in Music

In most classical western scores, durations and delays are expressed relative to the tempo i.e., the execution speed expressed in beats per minute (bpm). On the example of Figure 3, the duration of the first two notes is set to 1.0 beat, i.e., a quarter note. If the current tempo is 60 bpm it means 1s, but 0.5s if the tempo is 120 bpm. Musicians are free to interpret a score with a moving tempo. Indeed, in classical western music, tempo is one of the most prominent degrees of freedom for interpretation. It partially explains the huge difference between a real interpretation by a live musician and an automatic execution of the same score by a computer.

One of the main features of the Antescofo system is tempo inference. During a performance, the listening machine decodes both the position in the score and the execution speed. The tempo is not estimated from the last duration alone but rather from all durations detected since the beginning of the performance. In this way, the listening machine adds some inertia to tempo changes which corresponds to the real behavior of musicians playing together [5, 15].

3.2 The Antescofo language

In this framework, the most basic electronic actions, called *atomic actions* are simple control messages destined for the audio environment. Unlike traditional musical notes, atomic actions are instantaneous; they are not characterized by a duration but by the delay needed before their activation. Moreover, delays between electronic actions, like note durations, can be specified in relative time. Thus, electronic parts can follow the speed of the live performer as a trained musicians would.

Sequences of electronic actions are linked to an instrumental event, the *triggering event*. For instance, in Figure 3, action `a0` is bound to the first note with a delay of 0.0. When the first note is detected, action a_0 is sent immediately.

Actions can be regrouped into structures called groups. Groups are treated as atomic actions and are bound to an instrumental event and characterized by a delay. On the example of Figure 3, a group containing a sequence of three actions is bound to event e_2.

Furthermore, groups can be arbitrarily nested. It allows to faithfully capture the hierarchical structure of a musical piece. Actions contained in a nested group are executed in parallel with the actions following them in the embedding group, not in sequence.

A score can be described with the following types:

```
type score_event =
    { event : instr_event_label;
      seq : sequence; }

and sequence =  delay * asco_event list

and asco_event =
  | Action of action
  | Group of group

and action =
  | Message of audio_message
  | Signal of (unit, unit list) event

and group =
      { group_synchro: sync;
        group_error: err;
        group_seq: sequence }

and sync = Tight | Loose
and err = Local | Global | Causal | Partial
```

The type `score_event` represents a sequence of electronic actions `seq` bound to an instrumental event `event`. A sequence is just a list of electronic actions characterized by a delay relative to the tempo (see Section 3.1).

Atomic actions can be either simple control messages destined to the audio environment, `Message`, or classic ReactiveML signals to control other processes, `Signal`. ReactiveML signals are useful to couple the electronic accompaniment with other reactive processes (see Section 5).

The problem is that during a performance, the computer is supposed to act as a trained musician, following the other performers. Thus, the specification of mixed music scores faces two major challenges:

- During a live performance a musician can make a mistake, or worse, the listening machine may fail to recognize an event. In both cases, an expected event is missing. But what happens to actions bound to this event?
- Sometimes, a sequence of actions bound to an instrumental event may last longer than the duration of the triggering event. For instance, on our example, actions `a2` and `a3` should occur after event e_3 although they are triggered by event e_2. In this case, the question is: how should an electronic part synchronize with instrumental event that occur during its execution?

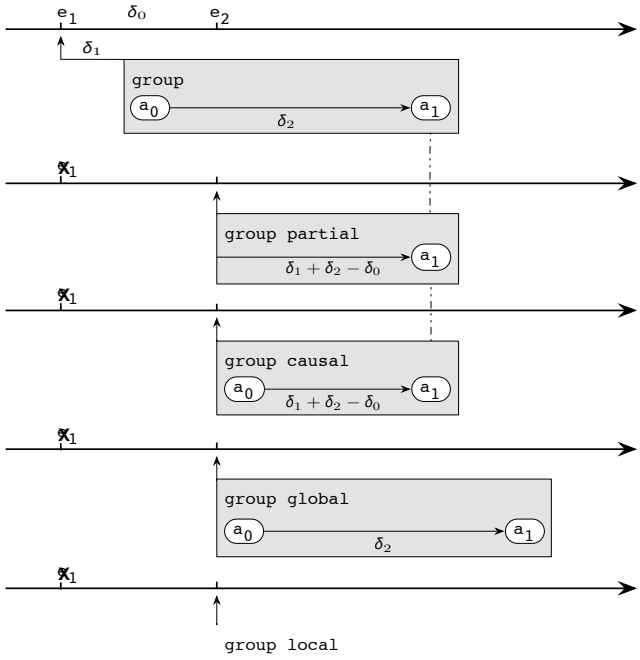


Figure 4. Illustration of the four error-handling attributes on a simple score (on top). e_1 and e_2 represent instrumental events. Suppose that e_1 is missed and e_2 is detected.

Error-handling and synchronization strategies depend very much on the musical context. The Antescofo language proposes solutions to reflect different musical behaviors. Groups are characterized by two attributes [6], a synchronization attribute (see Section 3.4) and an error handling attribute (see Section 3.3).

3.3 Error handling

The error-handling attribute defines the behavior of the group when the expected triggering event is absent. There are several ways to deal with errors depending on the musical context. We present here four exclusive error handling strategies: `Partial`, `Causal`, `Local` and `Global`. Figure 4 illustrates the four different behaviors.

The most basic accompaniment is perhaps a simple harmonization of the musician's voice. In this case, if a triggering event is missed, electronic part must keep on going as if the error never occurred. This is, for instance, the behavior of a pianist accompanying a beginner. Thus, attributes `Partial` and `Causal` aim to preserve a simple property: *the future of a performance does not depend on past errors*.

Now, the question is: what about actions that should already have been launched when the error was detected? If the error-handling attribute is set to `Partial`, these actions are simply discarded. On the other hand, if the error handling-attribute is set to `Causal`, these actions are immediately launched. This strategy could be useful if some actions are used to initialize an external process, e.g., turn on the light.

Moreover, a group can be used to define a complex control unit, e.g, parameters needed to synthesized the sound of a gong. Then, the integrity of the group must be preserved. Thus, when an error is detected, `Global` groups are launched with a zero delay, whereas `Local` groups are completely ignored.

3.4 Synchronization Strategies

A composer is allowed to specify how actions contained in a group will synchronize with instrumental events that occur during the execution of the group. Currently, the language proposes two distinct strategies: `Tight` and `Loose`.

If the synchronization attribute is set to `Tight`, every action contained in the group is triggered by the most recent corresponding instrumental event. The nearest event is computed with respect to the ideal timing of the score regardless of tempo changes. This strategy is ideal when electronic parts and the musician must remain as synchronous as possible, e.g., if an electronic voice is a simple harmonization of the performer's part. In the example of Figure 3, if the group has a synchronization attribute set to `Tight`, actions `a2` and `a3` will be triggered by e_3 even though the entire group is bound to the second instrumental event.

The strategy `Loose` allows the composer to define groups that, once triggered, only synchronize with the tempo. Due to the inertia of the tempo inference, an electronic action contained in such a group and an instrumental event that seems to be simultaneous in the score may be desynchronized during the performance. Indeed, a performer may accelerate or decelerate between two events. This strategy is used to preserve temporal relations between actions contained in the group. For instance, it can be very useful when the electronic part is treated as an independent background sound.

4. A First Example: *The house of the rising sun*

As a first example, let us take a classical folk song, *The house of the rising sun*. The score is presented in Figure 5. In this example, we want to define an electronic accompaniment which follows the main theme played by a real musician.♫

First, we need to create and initialize the `asco` environment that contains the instrumental score. This environment allows us to manage inputs from the listening machine and the outputs destined for the audio environment.

```
let asco = create_asco "rising_sun.asco" 120.
val asco: Ascolib.asco
```

To create the environment, we need the instrumental score of the future performance, here `"rising_sun.asco"` and an initial tempo, typically 120 bpm.

4.1 The basics

The accompaniment is roughly described in the score by a sequence of chords: Am, C, D... bound to instrumental events. For instance, the first chord is related to the second instrumental event, the second to the fourth one and so on.

First, we need a symbolic description of chords. Like pitch describe in Section 2.1, a chord is characterized by a pitch class: A, C, D... and a color: major or minor.

```
type color = Maj | Min
type chord = pitch_class * color
```

We can define the bass line of our example as a sequence of chords characterized by a delay:

```
let bass = [0.0, (A, Min); 2.0, (C, Maj); ...]
val bass: (delay * chord) list
```

Then, we need to turn this sequence of chords into a sequence of electronic actions. Perhaps the most simple accompaniment is to play only the roots of the bass line.

```
let root_of_chord chord octave =
  let (pitch_class, color) = chord in
  (pitch_class, octave)
val root_of_chord: chord -> octave -> pitch
```

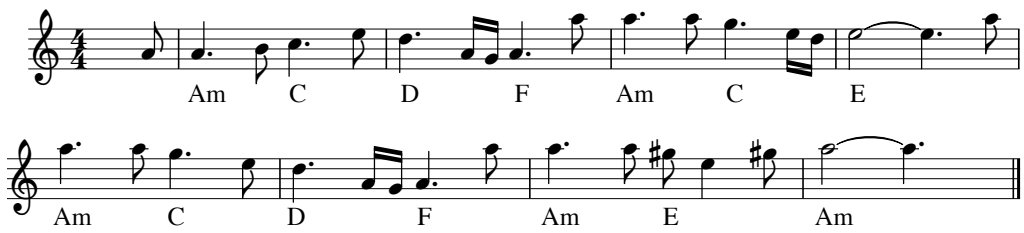


Figure 5. Score of the house of the rising sun (classical folk song)

```
let only_roots chords octave dur =
  List.map
    (fun (delay, chord) ->
       let root = root_of_chord chord octave in
       (delay, action_note (dur, root)))
    chords
val only_roots:
  (delay * chord) list -> octave -> dur -> sequence
```

Given an octave, the function `root_of_chord` returns the pitch corresponding to the root of a chord. Then, the function `only_roots` defines the sequence of electronic actions corresponding to the roots of a sequence of chords, `chords`. Function `action_note` converts a note into an atomic action destined for the audio environment (a value `Action message` where the message contains the characteristics of the note).

Remark that in Figure 5, no octave is specified on the score for the bass. We choose to instantiate the bass with the octave set to 3 (i.e., the first root of the bass is set to A3 = 220 Hz) and the duration to 2.0 beats which corresponds to the delay between two chords.

```
let roots = only_roots bass 3 2.0
val roots: sequence
```

The next thing to do is to build the link between the instrumental part and the electronic accompaniment. The process `link asco evt seq` links a sequence of actions `seq` to an instrumental event `evt`, in the environment `asco`. This process waits for the event to be detected or missed and it triggers the sequence. For instance, on the example of Figure 5, the first chord is bound to the second instrumental event. We can link the sequence `roots` to this event to obtain a basic accompaniment.

```
let process basic_accomp =
  run (link asco 2 roots)
val basic_accomp: unit process
```

If the performer does not play at constant speed, the accompaniment part may become desynchronized at some point. We can easily avoid this behavior if we put the bass line inside a `Tight` group (see Section 3.4). Moreover, it allows us to specify an error-handling strategy for the accompaniment. In our case, if an instrumental event is missed we do not want to hear the associated chord. Therefore we use the `Partial` strategy (see Section 3.3).

```
let process tight_accomp =
  let g = group Tight Partial roots in
  run (link asco 2 [(0.0, g)])
val tight_accomp: unit process
```

Here, the function `group` *sync err* `seq` is a simple constructor for a group with attributes `sync` and `err` containing the sequence of actions `seq`.

Note that the sequence bound to the second instrumental event contains only one zero delay electronic action: the group that defines the entire accompaniment. When this instrumental event is detected, each chord of the accompaniment is linked to its closest preceding instrumental event.

The last thing to do is to launch the listening machine and the reactive accompaniment.

```
let process main =
  run (init_asco asco);
  begin
    run (listener asco) ||
    run (tight_accomp) ||
    run (sender asco)
  end
val main: unit process
```

It first initializes the environment. Then, the process `listener` links the listening machine of Antescofo to the environment. In parallel, it executes the accompaniment part, and sends it to the audio environment `sender asco`.

4.2 Hierarchical structure

The harmonization defined above is a bit minimalist. One solution could be to add several other voices following the previous method. But we can do better.

In the original song, the accompaniment is made with arpeggio over the bass line. Thanks to functional composition, such an accompaniment is very easy to define.

First, we define the arpeggio style we want to apply:

```
let arpeggio chord octave =
  let fond = root_of_chord chord octave in
  let third =
    match color with
    | Min -> move_pitch 3 fond
    | Maj -> move_pitch 4 fond
  in
  let fifth = move_pitch 7 fond in
  let dur = 0.125 in
  group Loose Local
    [0.0, action_note (dur, fond);
     0.625, action_note (dur, third);
     0.125, action_note (dur, fifth);
     0.125, action_note (dur, move_octave 1 fond);
     0.125, action_note (dur, move_octave 1 third);
     0.333, action_note (dur, move_octave 1 fond);
     0.333, action_note (dur, fifth);]
val arpeggio : chord -> octave -> asco_event
```

An arpeggio corresponds to a `Loose Local` group. Thus, an arpeggio related to a missing event is entirely dismissed (`Local`). Besides, the synchronization strategy preserves the rhythm of the arpeggio (`Loose`). The function `move_octave` is similar to `move_pitch` and shift the pitch a given number of octaves.

Then, we can define the accompaniment by applying the function `arpeggio` to the bass line.

```
let arpeggio_bass chords octave =
  List.map
    (fun (delay, chord) ->
       (delay, arpeggio chord octave))
    chords
val arpeggio_bass:
  (delay * chord) list -> octave -> sequence
```

As for the basic accompaniment, we encompass the resulting sequence inside a `Tight Partial` group.

```
let process arpeggio_accomp =
  let chords = arpeggio_bass bass 3 in
  let g = group Tight Partial chords in
  run (link asco 2 [0.0, g])
val arpeggio_accomp: unit process
```

This accompaniments illustrates the possibility of defining hierarchical structures in the score. Indeed, a global `Tight` group contains a succession of arpeggios also defined as groups.

Finally we can replace the process `tight_accomp` in the main process by `arpeggio_accomp`.

4.3 Live Coding

A benefit of the embedding of Antescofo in ReactiveML is the ability to use its read–eval–print loop, called toplevel [16]. For example, if we execute the example of the rising sun in the toplevel, while it is playing, we can define a percussion part (we assume here that the environment `asco` has been properly initialized and that processes `listener` and `sender` are already running):

```
let process drums_accomp =
  let rhythm = [0.0; 0.625; 0.5; 0.5; ... ] in
  let drums =
    List.map (fun d -> d, action_beat) rhythm
  in
  let accomp = group Tight Partial drums in
  run (link asco 2 [0.0, accomp])
val drums_accomp: unit process
```

First, we define a sequence of delays, `rhythm`, to represent the rhythm of the drum part. Then, we turn this rhythm into a sequence of electronic actions, `drums`. Here, `action_beat` is a value `Action message` such that whenever the audio environment receives one of these messages, it plays a drum beat. Finally the result is a `Tight Partial` group containing the entire percussion part bound to the second instrumental event.

Dynamically during the performance, we can ask that this part be played:[8]

```
#run drums_accomp
```

This new voice will synchronize automatically with the others, even if the triggering event already passed, thanks to the error-handling strategy.

The expressiveness of the language and this dynamic aspect allows us to build more complex interactions with musicians during a performance.

Let us start with a simple example: a higher order process `killable` that allows a process given as argument to be stopped when a signal `k` is emitted. This process can be used to control the execution of an accompaniment.

[8] The `#run` is a toplevel directive which executes the process given as argument in background.

```
let process killable k p =
  do
    run p
  until k done
val killable:
  ('a, 'b) event -> unit process -> unit process
```

The construction `do`/`until` is not a looping structure. It stops the execution of its body when the signal is emitted.

Now, we can define a signal `kill_drum` in the toplevel:

```
signal kill_drum default () gather (fun x y -> ())
val kill_drum: (unit, unit) event
```

Instead of simply executing the percussion part, we execute it under the supervision of the `killable` combinator.

```
#run (killable kill_drum drum_accomp)
```

Then, we can stop the drum part during the performance whenever we want.

```
emit kill_drum ()
```

A more interesting example is to dynamically replace an accompaniment. First, let us define a signal `replace` on which we can send the processes.

```
signal replace
  default (process ())
  gather (fun x y -> process (run x || run y))
val replace: (unit process, unit process) event
```

Note that if several emissions occur during the same instant, the result will be the parallel composition of all the emitted processes.

Using the previous signal, we can define another higher order process `replaceable` which executes processes emitted on the signal `replace`.

```
let rec process replaceable replace p =
  do
    run p
  until replace (q) ->
    run (replaceable replace q)
  done
val replaceable:
  (unit process,  unit process) event ->
    unit process ->
      unit process
```

First, the process `replaceable` launches the process `p`. Then, whenever a process `q` is emitted on signal `replace`, the execution of `p` stops and the process `q` is launched in turn.

For instance, we can use this feature to dynamically switch between the different styles of accompaniment: basic, arpeggio or drums.

```
#run (replaceable replace tight_accomp)
emit replace arpeggio_accomp
emit replace drums_accomp
```

The first instruction launches the basic accompaniment (see Section 4.1). After a while, we can evaluate the second or the third instruction to stop the electronic part and start the complete accompaniment, `arpeggio_accomp` or the percussion part, `drums_accomp`.

Thus, it is possible to compose, correct and interact with the score during the performance.

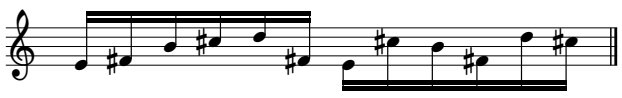


Figure 6. The melodic figure of Steve Reich's Piano Phase

5. Reactive Interactions

The main advantage of embedding the Antescofo language in ReactiveML is that the reactive accompaniment is now able to interact with regular ReactiveML processes. Indeed, atomic actions of the Antescofo language can be ReactiveML signals. This feature can be used to program musical pieces where the interaction between the performer and the accompaniment is more subtle than in the previous examples.

5.1 *Piano Phase*

Piano Phase is a piece written in 1967 by the minimalist composer Steve Reich. In this piece, two pianists, Alice and Bob, begin by playing the melodic figure presented in Figure 6 over and over again in unison. Then, one of the pianists, let us say Alice, begins to play her part slightly faster than Bob. When Alice plays the first note of the melody as Bob is playing the second note, they resynchronize for a while, i.e., Alice slows down to play at the same speed as Bob. The process is repeated. Alice accelerates again and then resynchronizes when she plays her first note as Bob is playing the third, then the fourth, etc.

If playing a simple melody at constant speed is relatively easy (Bob's part), Alice's part which alternatively desynchronizes and resynchronizes is much harder to achieve. Fortunately, we can program this part as an electronic accompaniment of a pianist that always plays at constant speed, here Bob.

The difficulty comes from the fact that we cannot easily compute *a priori* when Alice and Bob will resynchronize. Even in the original score, the composer only specifies bounds on the number of iterations of the sequence during the desynchronization (between four and sixteen times). How can we program this behavior?♫

Let us start with a simple process that plays the twelve note melody at a given speed and sends a signal `first_note` each time the first note is played. This code corresponds to Alice's part.

```
let process melody evt n delay first_note =
  let pattern =
    group Tight Partial
      [0.0, action_note (delay, (E,4));
       0.0, action_signal first_note;
       delay, action_note (delay, (Ff,4));
       delay, action_note (delay, (B,4));
       ...]
  in
  let seq = [0.0, pattern]
  let period = 12.0 *. delay in
  run (link_and_loop asco evt period n seq)
val melody:
  instr_event_label -> int -> delay ->
    (unit, unit) event -> unit process
```

First, we define a simple `Tight Partial` group that contains the melodic pattern presented in Figure 6. The second action, introduced with the function `action_signal` sends the signal `first_note`. Since the delay of this action is set to 0.0, the signal will be emitted simultaneously with the first note of the sequence. Then we compute the period of the pattern which is the duration of the twelve note melody, i.e., `12.0 *. delay`. The process `link_and_loop evt n period seq` links a sequence of electronic actions to an instrumental event `evt` in the environment and loops `n` times over the sequence `seq` where each iteration is separated by a delay `period`. If the period is shorter than the duration of the pattern, the execution of the loop leads to overlapping executions of the pattern.

The electronic part resynchronize when Bob plays the k-th note of the sequence as the electronic part is playing the first note. Thus, we need to track the k-th note of Bob's part.

```
let process track k kth_note  =
  let ie = instr_event asco in
  loop
    await ie (evt) when (evt.index mod 12 = k) in
    emit kth_note ()
  end
val track:
  int -> (unit, unit) event -> unit process
```

The function `instr_event` returns the signal carrying the output of the listening machine. Whenever an instrumental event is detected the index of the event in the score and the estimated tempo is sent on this signal. The construct `await ie (evt) when ...` binds the value received on `ie` to `evt` when the signal is emitted. The execution can then continue only if the condition of the `when` is satisfied. Here, if the detected position `evt.index` corresponds to the k-th note of the sequence, the signal `kth_note` is emitted.

The last thing to do is to compare the emissions of the two signals `first_note` and `kth_note`. If two emissions are close enough it emits a signal `sync`. We can split this process into three steps.

The process `stamp s r asco` associates a date to each emission of signal `s` and emits the result on a signal `r`.

```
let process stamp s r =
  loop
    await s;
    emit r (date asco)
  end
val stamp:
  (unit, unit) event -> (delay, delay) event ->
    unit process
```

The function `date` returns a date relative to the tempo. It corresponds to the elapsed delay since the beginning of the performance. Therefore the process `stamp` is responsive to the execution speed.

Then, the following process emits a signal `sync` each time the last values emitted on signals `r1` and `r2` are approximately equal, i.e., whenever the difference between the two values is less than a constant `eps`.

```
let process spy r1 r2 sync eps =
  loop
    await (r1 \/ r2);
    let t1 = last ?r1
    and t2 = last ?r2 in
    if abs_float (t1 -. t2) < eps then
      emit sync ()
  end
val spy:
  (float, float) event -> (float, float) event ->
    (unit, unit) event -> float -> unit process
```

The construct `await (r1 \/ r2)` awaits for one of the two signals `r1` or `r2` to be emitted. The expression `last ?r1` evaluates to the last value emitted on signal `r1`.

Finally we can combine the previous processes to achieve the desired behavior.

```
let process compare s1 s2 sync eps =
  signal r1 default -.eps gather (fun x y -> x) in
  signal r2 default -.eps gather (fun x y -> x) in
  run (stamp s1 r1) ||
  run (stamp s2 r2) ||
  run (spy r1 r2 sync eps)
val compare:
  (unit, unit) event -> (unit, unit) event ->
    (unit, unit) event -> float -> unit process
```

This process, associates a date to each emission of signals `s1` and `s2` taken as arguments. Results are emitted on signals `r1` and `r2`, respectively. Then, in parallel, we launch the process `spy` on `r1` and `r2`, which emits the signal `sync` whenever an emission of `s1` is close enough to an emission of `s2`. Since the process `stamp` is responsive to the tempo, the tolerance represented by the constant `eps` increases when the tempo decelerates and decreases when it accelerates. It corresponds to the behavior of actual musicians playing together.

Signals `r1` and `r2` are initialized in order to avoid initialization conflicts. Indeed, if a signal has never been emitted, the last value is the default value. Besides, multiple emissions are in this context very unlikely since the entire sequence is played between two emission of `first_note` or `kth_note`. Therefore, the combination function chooses arbitrarily one of the values emitted during the same instant.

Finally the entire piece is described by the following code.

```
let piano_phase sync desync first_note kth_note =
  let rec process piano_phase k =
    let ev = last_event asco in
    run (melody ev 4 0.25 first_note);
    emit desync ();
    do
      let ev = last_event asco in
      run (melody (ev+1) 16 0.246 first_note) ||
      run (track k kth_note) ||
      run (compare first_note kth_note sync 0.05)
    until sync done;
    run (piano_phase ((k + 1) mod 12))
  in
  piano_phase 1
val piano_phase:
  (unit, unit) event -> (unit, unit) event ->
    (unit, unit) event -> (unit, unit) event ->
      unit process
```

First, we play the sequence four times with the duration of the notes set to 0.25 (a sixteenth note) i.e., synchronously with the pianist. During this phase, the loop is linked to the last detected instrumental event: `last_event asco`. Then we emit the signal `desync` and start to play the sequence slightly faster with the next instrumental event (the duration of the notes is set to 0.246). In parallel, we compare emissions of signals `first_note` and `kth_note`. When they are close enough (here `eps` is set to 0.05), the signal `sync` is emitted. Then we increment k and restart the whole process. The number of iterations in the desynchronized mode is set to 16. After that, if the two pianists have not resynchronized, the electronic part stops.

5.2 Synchronous Observer

In ReactiveML, signals are broadcast communication channels. Therefore, one can easily write *synchronous observers* [11], i.e., processes that listen to the inputs and outputs of a process without altering its behavior. For instance, it is relatively easy to program a graphical interface for the previous example. This section is simply a proof of concept. We do not claim that this interface is especially aesthetic.

Let us start with a very simple process that draws a rectangle at a given position, `Left`, `Right` or `Full`, each time a signal `s` is emitted.

```
let process draw_sig s pos color =
  loop
    await s;
    draw_rect pos color;
    run (wait asco 0.2);
    Graphics.clear_graph ()
  end
val draw_sig:
  (unit, unit) event -> position -> color ->
    unit process
```

The process `wait` waits for a duration specified relative to the tempo of the environment. Thus, the interface is responsive to the speed of the performance.

Then, we can write a graphic observer that reacts to the emission of two signals `s1` and `s2` by alternating between two modes triggered by the signals `m1` and `m2`.

```
let process graphic_observer s1 s2 m1 m2 =
  loop
    do
      Graphics.clear_graph ();
      run (draw_sig s1 Full Graphics.green)
    until m1 done;
    do
      Graphics.clear_graph ();
      begin
        run (draw_sig s1 Left Graphics.blue) ||
        run (draw_sig s2 Right Graphics.red)
      end
    until m2 done
  end
val graphic_observer:
  (unit, unit) event -> (unit, unit) event ->
    (unit, unit) event -> (unit, unit) event ->
      unit process
```

In the first mode, we draw a green rectangle on the entire graphic window each time the signal `s1` is emitted. When an emission of signal `m1` occurs, we switch to the second mode. Then, we draw a blue rectangle on the left side of the window each time the signal `s1` is emitted, and a red rectangle on the right side if the signal `s2` is emitted. Then, we restart the whole process when the signal `m2` is emitted.

We can run the process `graphic_observer` with signals `first_note`, `kth_note`, `desync` and `sync` in parallel with the process `piano_phase` to obtain a simple graphical visualization of *Piano Phase*. During the synchronous phase, green flashes indicate that Alice, i.e., the electronic part, is playing the first note of the sequence. During the desynchronization, we can see red flashes on the right each time Bob, i.e., the actual pianist, is playing the k-th note, and blue ones on the left each time the accompaniment part is playing the first note.

6. Related Work

Comparison with the actual Antescofo language We focused here on a subset of the actual language of Antescofo. However, constructs that we left aside can be easily encoded using our library. For instance, an original control structure was added to the

language recently [8]. The construct `whenever pred seq` triggers the sequence `seq` whenever the predicate `pred` is true. The predicate may depend on variables, in this case, its value is re-evaluated each time one of the variables is updated. Adding this feature to our library was relatively easy as we now show.

First, variables become ReactiveML signals. Thus, it is possible to react when a variable is updated.

```
let make_var value =
  signal x default value gather (fun x y -> x)
```

The function `make_var value` creates a variable initialized with the value `value`.

Then, we can easily define classic functions to set and access the value of a variable.

```
let set x v = emit x v
let get x = last ?x
```

To set the value of a variable `x`, one need only to emit the desired value `v` on the corresponding signal. We can access the current value of a variable thanks to the function `last ?` (see Section 5.1).

Then, the following process waits for one of the variables contained in the list `vars` to be updated.

```
let await_vars vars =
  signal moved default () gather (fun x y -> ()) in
  let rec await_par vars =
    match l with
    | [] -> ()
    | x :: l ->
        run (await_par l) ||
        (await x; emit moved ())
  in
  do run (await_par vars) until moved done
```

The recursive process `await_par` waits in parallel for an update of any variable. Whenever, an update occurs, the signal `moved` is emitted. Thanks to the preemptive construct `do/until`, this emission stops the execution of the process `await_par`.

Finally the `whenever` construct can be implemented as follows.

```
let rec process whenever pred vars p =
  run (await_vars vars);
  begin
    if pred (List.map get vars) then run p ||
    run (whenever pred vars p)
  end
```

Whenever a variable contained in `vars` is updated, we re-evaluate the predicate `pred` with the new values of the variables. If the predicate is true we launch the process `p`, meanwhile we start another instance of the process `whenever`. Thus, if the predicate is true again while `p` is still running, we execute another process `p` in parallel. Here a predicate is a function applied to a list of variables.

Thus, our approach shows that ReactiveML is well suited for prototyping new features of the language.

Functional music The idea of composing music in a functional framework is not new. One can cite the Haskore library [14], for instance, which allows the expression and combination of musical structures in Haskell. More recently, Haskore and HasSound (a library for audio processing in Haskell) merged into a single embedded language: Euterpea [13]. Thus, this language handles both high-level music representation and low-level audio processing.

We took a similar approach: embedding a domain-specific language inside a functional language. Yet, our approach is not the same as we embed a reactive language in a more expressive reactive language. The major advantage, with respect to the embedding in a general purpose functional language is that we can rely on the temporal expressiveness, static analysis and runtime of ReactiveML.

Note that, we focused here on the control of a musical performance. Audio synthesis is left to an audio environment such as Max/MSP[9] [20] or its open source counterpart PureData [21]. An interesting development would be to couple our work with a functional language dedicated to audio synthesis like the language Faust [19] or Euterpea.

Live coding There are several languages dedicated to musical live coding. One can for instance cite Chuck [22] and SuperCollider [18]. These languages allow live interactions with an audio environment.

Our approach is completely different. Instead of creating an entire domain specific language, we chose to embed it in a general purpose one that already offers most of the features we need: higher order, recursion, inductive data types... This choice enabled us to implement our library in only a few hundred lines of code. Yet we can already use it to specify relatively complex behavior.

Composing Mixed Music The Iscore project [7] proposes a tool to compose interactive scores by specifying temporal constraints between musical objects. Musical structures can be bound to external events such as those occurring during a live performance. In this approach, relations are computed statically before the performance.

Besides, there exist programming languages dedicated to electronic music. For instance, Max/MSP and PureData are graphical languages that handle both the control treatment and signal processing.

However, in both cases it is relatively difficult to program dynamically changing behavior which evolves with parameters instantiated at performance time. For instance, the `whenever` construct described above would be difficult to simulate. Moreover, these languages are not functional.

7. Conclusion

In this paper we showed that a reactive functional language like ReactiveML is well suited to the specification of reactive processes such as a performance of mixed music. We believe that this embedding is a powerful tool for programming mixed music scores as well as prototyping new features for the Antescofo language.

Here, the link with a live performance is realized through the listening machine of the Antescofo system. However, this is not a real limitation. It could be interesting to adapt our framework to other sensors like a gesture follower [3], a speech recognition system or even combine these different input systems. Such an application with multiple sensors and actuators fits into the kind of systems for which synchronous languages like ReactiveML have been designed.

Acknowledgment

The authors would like to thank Antoine Madet, Timothy Bourke, Robin Morisset and Adrien Guatto for their kind and patient proofreading.

References

[1] G. Baudart, F. Jacquemard, L. Mandel, and M. Pouzet. A synchronous embedding of Antescofo, a domain-specific language for interactive mixed music. In *Thirteen International Conference on Embedded Software (EMSOFT'13)*, Montreal, Canada, Sept. 2013.

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

[9] http://cycling74.com/

[3] F. Bevilacqua, B. Zamborlin, A. Sypniewski, N. Schnell, F. Guédy, and N. Rasamimanana. Continuous realtime gesture following and recognition. In *Gesture in embodied communication and human-computer interaction*, pages 73–84. Springer, 2010.

[4] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software: Practice and Experience*, 21(4):401–428, 1991.

[5] A. Cont. Antescofo: Anticipatory synchronization and control of interactive parameters in computer music. In *International Computer Music Conference*, North Irland, Belfast, Aug. 2008.

[6] A. Cont, J. Echeveste, J.-L. Giavitto, and F. Jacquemard. Correct automatic accompaniment despite machine listening or human errors in Antescofo. In *ICMC 2012 - International Computer Music Conference*, Ljubljana, Slovénie, Sept. 2012.

[7] M. Desainte-Catherine and A. Allombert. Interactive scores: A model for specifying temporal relations between interactive and static events. *Journal of New Music Research*, 34(4):361–374, 2005.

[8] J. Echeveste. The future of the language of Antescofo. Personal communication, May 2013.

[9] J. Echeveste, A. Cont, J.-L. Giavitto, and F. Jacquemard. Operational semantics of a domain specific language for real time musician-computer interaction. *Discrete Event Dynamic Systems*, 2013.

[10] H. Fastl and E. Zwicker. *Psychoacoustics: Facts and models*, volume 22. Springer, 2006.

[11] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology (AMAST'93)*, pages 83–96. Springer, 1994.

[12] P. Hudak. *The Haskell school of expression: learning functional programming through multimedia*. Cambridge University Press, 2000.

[13] P. Hudak. *The Haskell School of Music – From Signals to Symphonies*. (Version 2.5), January 2013.

[14] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation–an algebra of music. *Journal of Functional Programming*, 6(03):465–484, 1996.

[15] E. Large and M. Jones. The dynamics of attending: How people track time-varying events. *Psychological review*, 106(1):119, 1999.

[16] L. Mandel and F. Plateau. Interactive programming of reactive systems. In *Proceedings of Model-driven High-level Programming of Embedded Systems (SLA++P'08)*, Budapest, Hungary, Apr. 2008.

[17] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 82–93, 2005.

[18] J. McCartney. Rethinking the computer music language: SuperCollider. *Computer Music Journal*, 26(4):61–68, 2002.

[19] Y. Orlarey, D. Fober, and S. Letz. FAUST: an efficient functional approach to DSP programming. *New Computational Paradigms for Computer Music*, 2009.

[20] M. Puckette. Combining event and signal processing in the MAX graphical programming environment. *Computer music journal*, 15:68–77, 1991.

[21] M. Puckette et al. Pure data: another integrated computer music environment. *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41, 1996.

[22] G. Wang and P. Cook. Chuck: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *Proceedings of the 12th annual ACM international conference on Multimedia*, MULTIMEDIA '04, pages 812–815, New York, NY, USA, 2004. ACM.