A Synchronous Embedding of Antescofo, a Domain-Specific Language for Interactive Mixed Music

Guillaume Baudart[*]

Florent Jacquemard[†]

Louis Mandel[‡]

Marc Pouzet[§]

ABSTRACT

ANTESCOFO is recently developed software for *musical score following* and *mixed music*: it automatically, and in real-time, synchronizes electronic instruments with a musician playing on a classical instrument. Therefore, it faces some of the same major challenges as embedded systems.

The system provides a programming language used by composers to specify musical pieces that mix interacting electronic and classical instruments. This language is developed with and for musicians and it continues to evolve according to their needs. Yet its semantics has only recently been formally defined. This paper presents a *synchronous semantics* for the core language of ANTESCOFO and an alternative implementation based on an embedding inside an existing synchronous language, namely REACTIVEML. The semantics reduces to a few rules, is mathematically precise and leads to an interpretor of only a few hundred lines. The efficiency of this interpretor compares well with that of the actual implementation: on all musical pieces we have tested, response times have been less than the reaction time of the human ear. Moreover, this embedding permitted the prototyping of several new programming constructs, some of which are described in this paper.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Concurrent, distributed, parallel languages; D.3.1 [**Formal Definitions and Theory**]: Semantics; H.5.5 [**Information Interfaces and Presentation**]: Sound and Music Computing.

---

[*]ENS de Cachan, Antenne de Bretagne, DI École normale supérieure. Email: Guillaume.Baudart@ens.fr

[†]IRCAM Paris, INRIA Paris-Rocquencourt.
Email: Florent.Jacquemard@inria.fr

[‡]Univ. Paris-Sud 11, DI École normale supérieure, INRIA Paris-Rocquencourt. Email: Louis.Mandel@lri.fr

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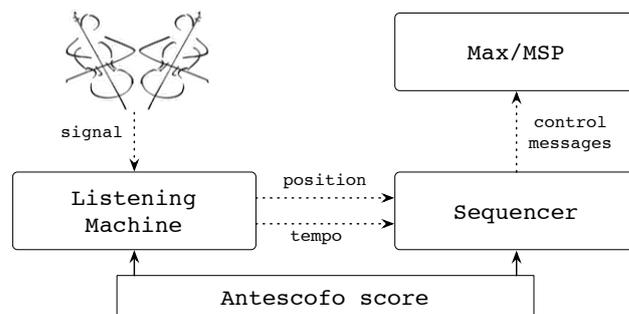
.

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

Keywords

Synchronous Programming; Language Embedding; Error Handling and Synchronization Strategies; Mixed Music.

1. INTRODUCTION

Since the 1950s, composers have shown a growing interest in new kinds of music mixing electronic parts and live musicians. Thus, computer music research and industry have focused on real-time interaction between musicians and computers. These considerations have led to the development of programming languages dedicated to musical interaction. Examples include Max/MSP, which resulted from the collaboration of Miller Puckette and the composer Philippe Manoury [13], and James McCartney's SuperCollider [12].

In the same vein, ANTESCOFO[1] is a system and dedicated programming language developed for the synchronization and control of interactive parameters in computer music. It allows a composer to manage a computer interacting with other musicians at performance time. Using this system, composers can jointly specify both instrumental and electronic parts on the same score.

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 illustrates the global behavior of the system. It is composed of two distinct subsystems: a listening machine and a sequencer. During a performance, the *listening ma-*

---

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

chine 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 this information to trigger electronic actions by sending control messages to a music programming environment: Max/MSP.[2] Max/MSP use these messages to handle complex sound synthesis, manage lights, etc. More details can be found in [3].

ANTESCOFO faces some of the major challenges of embedded system design and implementation: the synchronization of all the electronic instruments that play in parallel with the score follower itself; the mix of logical and physical time with slow and fast time scales; the design and implementation of an expressive language where programs are compiled to target code guaranteed to run in real-time. Yet the relationships between ANTESCOFO and existing models and languages for embedded systems have been little studied.

In this article we focus on the sequencer, a typical example of a reactive system that continuously receives inputs from the listening machine. When the listening machine detects an event, the sequencer reacts to produce the corresponding accompaniment. Moreover, the dedicated language includes constructs typical of languages for embedded systems—like a logical global time scale, synchronous parallelism and instantaneous broadcast—but also some original ones for synchronization and error handling strategies.

Among programming languages for embedded systems, *synchronous languages* [1] are used in the most critical applications including airplanes, trains, and automotive subsystems. They incorporate a mathematically precise model of concurrency and advanced features for communication and code generation. The language of ANTESCOFO can benefit from this research and, more fundamentally, links with this trend of research be investigated.

Contributions of the paper.

This paper presents a new semantics and implementation for the core language of ANTESCOFO. An originality of our approach is to implement the semantics as an interpreter inside an existing synchronous language. In this way, the precise semantics leads directly to an implementation.

For the implementation platform, we chose REAC-TIVEML [11] which appeared to be the best candidate, though we also experimented in the LUSTRE-like language defined in [9]. Indeed, the ability to define inductive datatypes, higher order processes and recursion greatly simplify programming. The implementation is only a few hundred lines of REACTIVEML code (see Appendix C) and it competes with the current implementation. In all of our experiments, response time were less than the reaction time of the human ear.

By embedding ANTESCOFO inside REACTIVEML, we were able to add and experiment with novel programming constructs and synchronization strategies with little effort. We illustrate this with several examples. Our framework is thus a powerful tool for prototyping new constructs before possibly integrating them into the core language.

The paper is organized as follows. Section 2 defines the core language of ANTESCOFO. Its semantics is defined in Section 3. Section 4 presents the REACTIVEML implementation. In Section 5, we presents several applications. Related work is discussed in Section 6 and we conclude in Section 7.
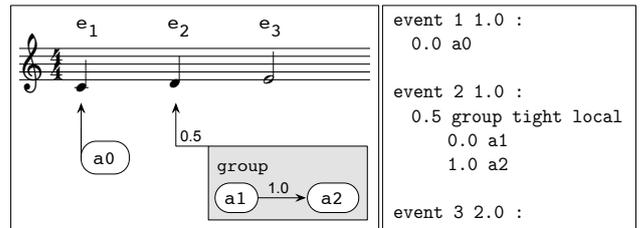

Figure 2: Representation of an Antescofo score; on the right its textual representation. ♩

Throughout the paper, we present several examples; they are available to the reader together with the REACTIVEML source code at http://reactiveml.org/emsoft13.

2. A LANGUAGE FOR MIXED MUSIC

We now describe the kernel of the ANTESCOFO language, a language dedicated to the writing of mixed music scores [5]. This language was developed as a language for coordinating events performed by humans and electronic actions controlled by a computer. It allows a composer to specify both electronic and instrumental parts in the same score. Figure 2 shows a simple example of one such score (the symbol ♩ indicates a link to a demonstration video available online).

The language permits expressing delays relative to a tempo expressed in beats per minute (bpm). For instance, a duration of 1.0 means 1.0 *beat*. During a performance, the listening machine estimates both the position in the score and the tempo of the performer. This allows the sequencer to follow the speed of the performer as a trained musician would. Indeed, 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 [3]. This feature explains some of the success of ANTESCOFO with composers, as synchronizing different parts using a common tempo is standard practice when writing polyphonic music (in many other environments for mixed music, delays can only be expressed in milliseconds).

2.1 The Core Language

The main idea is to bind electronic actions to instrumental events. During a performance, actions related to an instrumental event are executed when the event is detected. Thus, a *score* is a sequence of instrumental events and for each such event an associated sequence of electronic actions. It is described by the following grammar (the empty sequence is denoted ε).

$$
\begin{array}{rcl}
score & ::= & \varepsilon \mid (event : seq)\ score \\
event & ::= & \texttt{event}\ i\ t \\
seq & ::= & \varepsilon \mid (\delta\ ae)\ seq \\
ae & ::= & action \mid group \\
group & ::= & \texttt{group}\ synchro\ error\ seq \\
synchro & ::= & \texttt{tight} \mid \texttt{loose} \\
error & ::= & \texttt{local} \mid \texttt{global} \mid \texttt{partial} \mid \texttt{causal}
\end{array}
$$

An instrumental event (e.g., a note, chord, trill, etc.) is denoted by $\texttt{event}\ i\ t$, where $i \in \mathbb{N}$ is the index of an event in the score and $t \in \mathbb{Q}$ its duration relative to the tempo.

Indeed, for the sequencer, instrumental events are only triggers for electronic actions. Therefore, the only useful information is the position and the duration of an event. An electronic action (ae) is either an atomic action taken from a finite set ($action \in \mathbb{A}$) or a group of electronic actions. A group is a sequence of electronic actions characterized by a synchronization strategy (described in Section 2.2) and an error handling strategy (described in Section 2.3). A sequence (seq) is a list of pairs, each associating an electronic action (ae) with a delay (δ) relative to the tempo ($\delta \in \mathbb{Q}$).

The most basic actions in a score, called *atomic actions*, are simple control messages destined for the audio environment (e.g., UDP messages for Max/MSP). Each is bound to an instrumental event, the *triggering event*, and characterized by a delay. When the listening machine detects the triggering event, the sequencer waits for the specified delay and then sends the corresponding control message. In the example of Figure 2, action a_0 is bound to the first note e_1 with a delay of 0.0. Thus, when the first note is detected, the message a_0 is sent immediately.

Atomic actions can be grouped into control structures called *groups*. Like an atomic action, a group is triggered by an instrumental event and characterized by a delay. When the triggering event is detected, the sequencer waits for the corresponding delay and then launches the actions contained in the body of the group. In the example, a group is bound to the second instrumental event with a delay of 0.5 *beat*. When this event is detected, the sequencer waits 0.5 *beat* and then launches action a_1, after another delay of 1.0 *beat*, the message a_2 is sent. Groups can be nested arbitrarily. Actions contained in a nested group are executed in parallel with the actions following them in the embedding group, not in sequence. An electronic voice can be split into two parallel voices which can in turn be split and so on. Thus, a score can faithfully capture the complexity of a musical piece.

There are two kinds of parallelism in the language. First, two sequences bound to different instrumental events are executed in parallel. Second, a nested group inside a sequence is executed in parallel with the rest of the sequence. Note that when two sequences are executed in parallel, it is important that they share the same global time: the time of the performance. That is, they must be synchronous. Otherwise the performance would not reflect the musical score. To this similarity with synchronous languages, ANTESCOFO adds two original features: synchronization and error handling strategies.

2.2 Synchronization Strategies

Groups are characterized by two attributes (see [5]). The first one defines a synchronization strategy. 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. There are several ways to achieve this synchronization depending on the musical context. Currently, the language proposes two distinct modes of synchronization: `loose` and `tight`.

Once a `loose` group is launched, delays are computed according to the current value of the tempo, regardless of instrumental events that may occur during its execution. 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
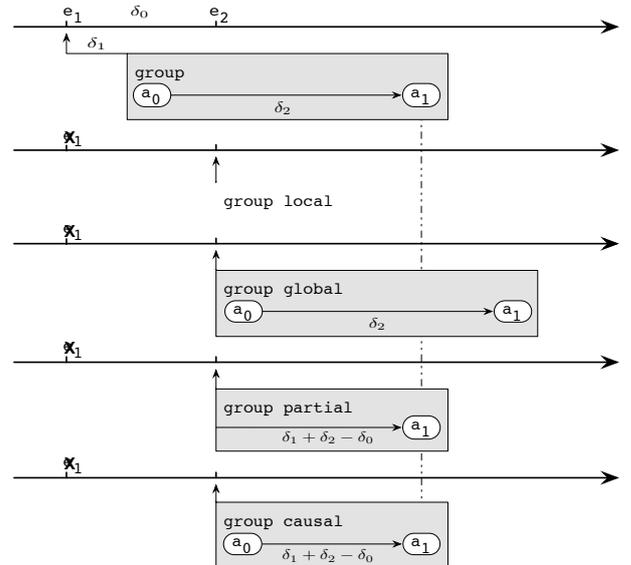
Figure 3: Illustration of the four error handling attributes on a simple score (at top). e_1 and e_2 represent instrumental events. Suppose that e_1 is missed and e_2 is detected.

or decelerate between two events. Typically, this strategy is used in those parts of a score where a performer follows the electronic voices.

In a `tight` group, every action is triggered by the most recent corresponding instrumental event. In the example of Figure 2, the group has a synchronization attribute set to `tight`. Thus, although the entire group is bound to the second instrumental event, action a_2 will be triggered by e_3. Here, the nearest event is computed with respect to the ideal timing of the score regardless of tempo changes. This strategy is ideal when the electronic voice must accompany the interpreter's voice.

2.3 Error Handling Strategies

ANTESCOFO is designed to accompany real musicians and, thus, errors may sometimes occur. By error we mean an instrumental event which is expected and missing, either because it was not played by the musician or not detected by the listening machine. The second attribute of a group defines the error handling strategy which should be taken when an expected triggering event is absent. There are several ways to deal with errors depending on the musical context. Here, we present four exclusive error handling strategies: `local`, `global`, `partial` and `causal`. Figure 3 illustrates the four different behaviors.

The `local` and `global` strategies preserve the integrity of a group. Indeed, if the triggering event is missed, the group is either completely ignored (`local`) or launched with zero delay (`global`) as soon as a later event is detected. In both cases, delays between actions within the group are unchanged. The group can thus be seen as a single block with a certain duration.

The other two strategies aim to preserve a simple property: *The future of a performance does not depend on past errors*, i.e., the show must go on! When an error occurs, the corresponding group is split into two parts: actions that

should already have been launched when the error was detected, termed the *past*, and actions that should occur after the detection, termed the *future*. The attributes `partial` and `causal` differ only in their treatment of past actions. For `causal` groups, past atomic actions are launched immediately. For `partial` groups, past atomic actions are simply discarded. In both cases, future actions are launched as if they were bound to the next detected event. They are performed as if an error never occurred.

3. BEHAVIORAL SEMANTICS

In this section we describe the semantics of the core of ANTESCOFO. The remaining features, like loops and continuous groups, are easily expressed in our kernel (see Section 6). The score given in Figure 4 will serve as a running example. In the following, *sync* and *err* stand, respectively, for any of the synchronization and error handling attributes already presented. The semantics rules are given in Figure 5 and explained below.

3.1 Execution Rules

The semantics specifies, given a set of detected events and a score, the intended performance, that is, the desired output of the sequencer relative to the tempo. It is defined by the predicate:

$$D \models^{exec} sc \Rightarrow p$$

It relates a set of detected events $D \subseteq \mathbb{N}$, a score sc, and a performance p, which is a set of triplets (i, δ, a) where $i \in \mathbb{N}$ is a detected instrumental event (e.g., the index of the position in the score), $\delta \in \mathbb{Q}$ is the delay to wait after the detection of i and $a \in \mathbb{A}$ is an atomic action.

A score is a sequence of score events of the form (`event` i t : seq) where $i \in \mathbb{N}$ denotes an instrumental event of duration $t \in \mathbb{Q}$ and seq the associated sequence of electronic actions (see Section 2.1). If the score is empty (rule (EMPTY SCORE)), the associated performance is empty. Otherwise (rule (EXEC SCORE)), we collect in parallel the performances generated by every score event contained in the score. This is done by iterating the predicate:

$$D \models^{exec} (\text{event } i\, t : seq) \rightarrow p$$

which relates a score event with a performance.

Now, to obtain the performance associated to such score events (`event` i t : seq), the first thing to do is to check whether the corresponding instrumental event i is detected or missed:

- Rule (DETECT): i is detected, that is, $i \in D$. We apply a predicate *detected* to the sequence seq. Thus:

$$D, i, \delta \models^{detected} seq \Rightarrow p$$

means that sequence seq, bound to the detected instrumental event i with a delay δ, leads to the performance p.

- Rule (MISS): i is missing, i.e., $i \notin D$. We apply a predicate *missed* to the sequence seq. Thus:

$$D, i, \delta \models^{missed} seq \Rightarrow p$$

means that sequence seq related to the missing event i with a delay δ leads to the performance p.

In order to deal with sequences of actions, we introduce two administrative rules (EMPTY SEQUENCE) and (EXEC SEQUENCE). These rules allow us to apply a predicate, *detected* or *missed* on a sequence of electronic actions while computing the correct delay for each action. Here, *generic* can be instantiated by *detected* or *missed*.

For example, let us consider a simple sequence of electronic actions $[(\delta_0\ a_0)\ (\delta_1\ a_1)]$ related to an event i with a delay δ. For both actions, the triggering event is i, and the associated delays are $\delta + \delta_0$ for a_0, and $\delta + \delta_0 + \delta_1$ for a_1. More generally, the delay associated to an action in a sequence is the sum of previous delays.

In the following, we will detail the behaviors of the different electronic actions (atomic actions and groups). But first, we need to define some auxiliary functions.

3.2 Notations

Let $\mathcal{E}$ be the function that returns the delay between an instrumental event i and the beginning of the score, i.e., the *date* of i relative to the tempo. This is just the sum of the durations t_k of all instrumental events k between the beginning of the score and i:

$$\mathcal{E}(i) = \sum_{k=1}^{i-1} t_k$$

Besides, by definition in Section 2.3, it is impossible to detect a missing event before the next detected instrumental event. Indeed, if the next event is not yet detected, it could simply be a deceleration of the tempo. For a missing event $i \notin D$, $\mathcal{M}(i)$ denotes the next detected event. Formally: $\mathcal{M}(i) = \min\{j \in D \mid \mathcal{E}(j) > \mathcal{E}(i)\}$.

3.3 Atomic Actions

Now we define the predicates of the form:

$$D, i, \delta \models^{generic} ae \rightarrow p$$

They explain the behavior of an electronic action ae bound to an event i with a delay δ.

First, let us consider the case of an atomic action a. If the event i associated to a is detected, we apply the rule (DETECTED ACTION) which leads to the performance (i, δ, a). Indeed, after the detection of i we just have to wait out the delay δ and then send the message a.

Now, if i is missing, the error will be detected with the detection of $j = \mathcal{M}(i)$. The behavior of an atomic action when the triggering event is missed is described by the rule (MISSED ACTION). If a should have been launched before the event j, the message is immediately sent. Otherwise we wait the remaining delay: $(\mathcal{E}(i) + \delta) - \mathcal{E}(j)$ before sending the message.

Example. *In our example, there is only one atomic action related to e_4. Thus, if e_4 is detected, the corresponding performance is the triplet:* $(4, 0.5, a_{41})$.

3.4 Synchronization Strategies

Now, recall that actions can also be structured into groups whose synchronization with the instrumental events can be specified with the attributes presented in Section 2.2.

To interpret a group declared as `loose`, we just apply the detected version of the rule (EXEC SEQUENCE) to the body of the group (rule (DETECTED LOOSE GROUP)). The triggering event i remains the same for all these actions.

```
event 1 2.0 :
  0.0 group sync partial
    1.0 group sync partial
      0.0 a11
      1.5 a13
    1.0 a12

event 2 2.0 :
  1.0 a21
  0.5 group sync err
    0.0 a22
    1.0 a23

event 3 1.0 :

event 4 1.0 :
  0.5 a41
```
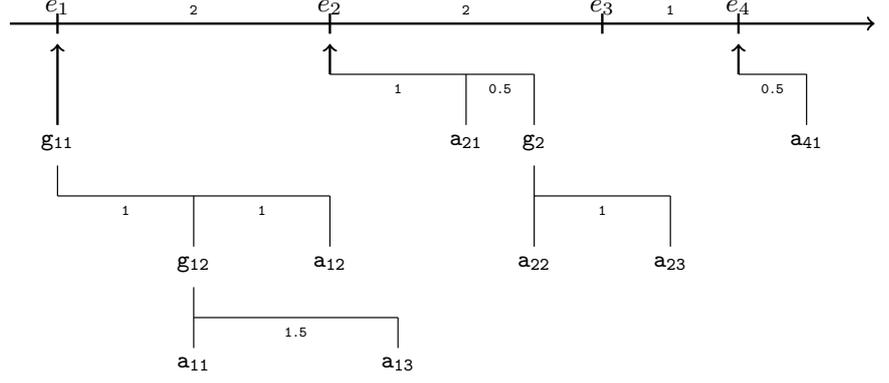
Figure 4: A simple score and its graphical representation where the e_i's denote instrumental events.

(Empty Score)
$$D \vdash^{exec} \varepsilon \Rightarrow \varnothing$$

(Exec Score)
$$\frac{D \vdash^{exec} (\mathtt{event}\ i\ t : seq) \to p_1 \qquad D \vdash^{exec} sc \Rightarrow p_2}{D \vdash^{exec} (\mathtt{event}\ i\ t : seq)\ sc \Rightarrow p_1 \cup p_2}$$

(Detect)
$$\frac{i \in D \qquad D, i, 0.0 \vdash^{detected} seq \Rightarrow p}{D \vdash^{exec} (\mathtt{event}\ i\ t : seq) \to p}$$

(Miss)
$$\frac{i \notin D \qquad D, i, 0.0 \vdash^{missed} seq \Rightarrow p}{D \vdash^{exec} (\mathtt{event}\ i\ t : seq) \to p}$$

(Empty Sequence)
$$D, i, \delta \vdash^{generic} \varepsilon \Rightarrow \varnothing$$

(Exec Sequence)
$$\frac{D, i, \delta + \delta' \vdash^{generic} ae \to p_1 \qquad D, i, \delta + \delta' \vdash^{generic} seq \Rightarrow p_2}{D, i, \delta \vdash^{generic} (\delta'\ ae)\ seq \Rightarrow p_1 \cup p_2}$$

(Detected Action)
$$D, i, \delta \vdash^{detected} a \to (i, \delta, a)$$

(Missed Action)
$$\frac{\mathcal{M}(i) = j}{D, i, \delta \vdash^{missed} a \to (j, \max(0.0, \mathcal{E}(i) + \delta - \mathcal{E}(j)), a)}$$

(Detected Loose Group)
$$\frac{D, i, \delta \vdash^{detected} seq \Rightarrow p}{D, i, \delta \vdash^{detected} \mathtt{group\ loose}\ err\ seq \to p}$$

(Detected Tight Group)
$$\frac{D \vdash^{exec} \mathrm{Slice}(i, \delta, (\mathtt{group\ tight}\ err\ seq)) \to p}{D, i, \delta \vdash^{detected} \mathtt{group\ tight}\ err\ seq \to p}$$

(Missed Local Group)
$$D, i, \delta \vdash^{missed} \mathtt{group}\ sync\ \mathtt{local}\ seq \to \varnothing$$

(Missed Global Group)
$$\frac{\mathcal{M}(i) = j \qquad D, j, 0.0 \vdash^{detected} \mathtt{group}\ sync\ \mathtt{global}\ seq \to p}{D, i, \delta \vdash^{missed} \mathtt{group}\ sync\ \mathtt{global}\ seq \to p}$$

(Missed Causal Group)
$$\frac{\mathcal{M}(i) = j \qquad (past,\ future) = \mathrm{Split}(i, j, \delta, seq) \qquad D, i, \delta \vdash^{missed} past \Rightarrow p_1 \qquad D, j, 0.0 \vdash^{detected} \mathtt{group}\ sync\ \mathtt{causal}\ future \to p_2}{D, i, \delta \vdash^{missed} \mathtt{group}\ sync\ \mathtt{causal}\ seq \to p_1 \cup p_2}$$

(Missed Partial Group)
$$\frac{\mathcal{M}(i) = j \qquad (past,\ future) = \mathrm{Split}(i, j, \delta, seq) \qquad D, i, \delta \vdash^{missed} \mathrm{Extract}(past) \Rightarrow p_1 \qquad D, j, 0.0 \vdash^{detected} \mathtt{group}\ sync\ \mathtt{partial}\ future \to p_2}{D, i, \delta \vdash^{missed} \mathtt{group}\ sync\ \mathtt{partial}\ seq \to p_1 \cup p_2}$$
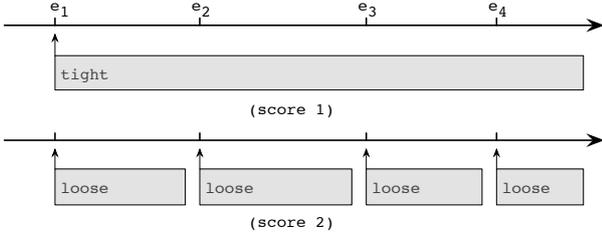
Figure 5: Behavioral Semantics

Figure 6: Slicing of a `tight` group.

Example. *Imagine that g_2 is a group `loose` and e_2 a detected event. Then, the execution of g_2 leads to the performance:* $\{(2, 1.5, a_{22}), (2, 2.5, a_{23})\}$.

For a group declared as `tight` a bit more treatment is needed. Intuitively, the body of the group must be sliced according to an instrumental event that should occur during the execution of the group. Each action in the body is associated to the nearest instrumental event in the past. Figure 6 illustrates this transformation: the result of the slicing of the group in (score 1) is a fresh score (score 2), containing one `loose` group bound to each instrumental event that should occur during the execution of (score 1).

Formally, let g be a `tight` group bound to event i with a delay δ and body $seq = [(\delta_1\ x_1)\ (\delta_2\ x_2)\ ...\ (\delta_n\ x_n)]$. Here x denotes any kind of electronic action, group or atomic action. First we associate a date d_k, relative to the tempo, to each element x_k.

$$d_k = \mathcal{E}(i) + \delta + \sum_{p=1}^{k} \delta_p$$

Then, for each instrumental event $j \geq i$, we compute the subsequence s_j of seq such that the date of electronic actions is between the date of j and the date of $j + 1$:

$$s_j = [(\delta_{j_0}\ x_{j_0})\ ...\ (\delta_{j_1}\ x_{j_1})]$$
$$\text{where } d_{j_0-1} < \mathcal{E}(j) \leq d_{j_0} \leq d_{j_1} < \mathcal{E}(j+1) \leq d_{j_1+1}$$

Each subsequence s_j is associated to the corresponding event j instead of i. Therefore, we need to update delays such that the dates of the electronic actions remain the same: $\delta'_{j_l} = d_{j_l} - \mathcal{E}(j)$. Finally, the resulting sequence: $s'_j = [(\delta'_{j_0}\ x_{j_0})\ (\delta'_{j_0+1}\ x_{j_0+1})\ ...\ (\delta'_{j_1}\ x_{j_1})]$ becomes the body of a fresh `loose` group with the same error handling strategy as g. Thereby we preserve the error handling strategy for the future: $g_j = \text{group loose } err\ s'_j$. The result of the slicing is a fresh score containing one score event for each group g_j: $se_j = (\text{event } j\ t_j : (0.0\ g_j))$.

$$\text{Slice}(i, \delta, g) = [se_i\ ...\ se_{i+k}] \text{ where } \mathcal{E}(i+k+1) \geq d_n$$

The last thing to do is to execute the resulting score. This behavior is described by the rule (DETECTED TIGHT GROUP).

Example. *Suppose that g_2 is declared as `tight` and e_2 is detected. The result of the slicing is (see Figure 4):*

$$\begin{aligned}
\text{Slice}(2, 1.5, g_2) &= [(\text{event } 2\ 2.0 : (0.0\ g_2^1)) \\
&\qquad (\text{event } 3\ 1.0 : (0.0\ g_2^2))] \\
\text{where } g_2^1 &= \text{group loose } err\ (1.5\ a_{22}) \\
\text{and } g_2^2 &= \text{group loose } err\ (0.5\ a_{23})
\end{aligned}$$

Assuming that e_3 is detected, the corresponding performance will be: $\{(2, 1.5, a_{22}), (3, 0.5, a_{23})\}$.

3.5 Error Handling Strategies

Groups are also characterized by an error handling strategy: `local`, `global`, `partial` or `causal`. These attributes allow a composer to specify how the group will behave if the triggering event is missed (see Section 2.3). When the triggering event is missing, a `local` group is completely ignored (rule (MISSED LOCAL GROUP)). Conversely, a `global` one is launched with zero delay, as if the next detected event was the triggering event (rule (MISSED GLOBAL GROUP)).

Example. *For example, assume that g_2 is declared as `global`, e_2 is missing and e_3 is detected. Then, the corresponding performance will be:* $\{(3, 0.0, a_{22}), (3, 1.0, a_{23})\}$. *Similarly, if e_2 and e_3 are missing and e_4 is detected, then, the performance corresponding to g_2 will be:* $\{(4, 0.0, a_{22}), (4, 1.0, a_{23})\}$. *If the group is declared `local`, the performance will be empty.*

For the two other attributes, the first thing to do is to split the body of the group into two subsequences (see Section 2.2): actions that should have been launched before the error detection (*past*), and actions that should occur after the error detection (*future*).

Formally, let g be a group declared as `causal` or `partial`, with body $seq = [(\delta_1\ x_1)\ (\delta_2\ x_2)\ ...\ (\delta_n\ x_n)]$. We suppose that g is bound to a missing event $i \notin D$ such that $\mathcal{M}(i) = j$. First, we associate a date d_k, relative to the tempo, to each element x_k (see the slicing of `tight` groups in Section 3.4). Then we can split the sequence seq into the two subsequences, *past* and *future*. This is done by the function Split:

$$\begin{aligned}
\text{Split}(i, j, \delta, seq) &= (\textit{past, future}) \\
\text{where } \textit{past} &= [(\delta_1\ x_1)\ ...\ (\delta_p\ x_p)] \textit{ with } d_p < \mathcal{E}(j) \leq d_{p+1} \\
\text{and } \textit{future} &= [(\delta_{p+1}\ x_{p+1})\ ...\ (\delta_n\ x_n)]
\end{aligned}$$

Attributes `partial` and `causal` only differ in the treatment of past actions. The difficulty comes from the hierarchical structure of the score. Indeed a group that appears in the past could contain an action that should be launch in the future. Hence, nested groups must be split as well. We solve this problem by recursively applying the *missed* version of the rule (EXEC SEQUENCE) on past actions sequences.

If the past contains an atomic action this action will be launched immediately. This is described by the rule (MISSED ACTION). This is the desired behavior for a `causal` group. On the other hand, to achieve the `partial` behavior, we need to ignore past atomic actions. In other words, the past of a `partial` group only contains past nested groups.

Thus for a `partial` group, we need to compute a new sequence which contains only past nested groups. The past is a sequence of actions, therefore the delay associated to each action is relative to the previous one. Hence we must re-compute the delays associated to each group while extracting them. Let x_{g_1} be the first group of the past, x_{g_2} the second one and so on. Since x_{g_1} is the first group, actions $x_1, x_2, ..., x_{g_1-1}$ are atomic actions. Thus in the new sequence, the delay associated to x_{g_1} is $\delta_1 + \delta_2 + ... + \delta_{g_1}$. More generally, the new delay of a group x_{g_i} is the sum of delays between x_{g_i} and the previous group $x_{g_{i-1}}$: $\delta_{g_{i-1}+1} + \delta_{g_{i-1}+2} + ... + \delta_{g_i}$.

$$\begin{aligned}
\text{Extract}(\textit{past}) &= [(\delta'_{g_1}\ x_{g_1})\ (\delta'_{g_2}, x_{g_2})\ ...\ (\delta'_{g_p}\ x_{g_p})] \\
&\text{where } \forall 1 \leq i \leq p\ \ \delta'_{g_i} = \sum_{k=g_{i-1}+1}^{g_i} \delta_k
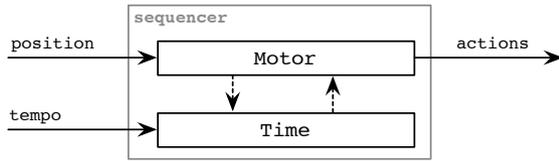\end{aligned}$$

Figure 7: Architecture of the sequencer.

Finally we wrap the future into a fresh group with the synchronization attribute and the error handling attribute of the original group. We thereby preserve both the synchronization strategy and the error handling strategy for the future. The last thing to do is to execute the missed version of the (EXEC SEQUENCE) rule on the past and the detected version of this rule on the future. Thus, the future is launched as if it were related to the detected event j, and the rest of the score is executed as if the error never occurred. These behaviors are described by the rules (MISSED CAUSAL GROUP) and (MISSED PARTIAL GROUP).

Example. *To illustrate the case of nested groups, let us assume that the groups g_{11} and g_{12} have been declared* **partial**, *and that e_1 is missed and e_2 is detected, i.e., $1 \notin D$ and $\mathcal{M}(1) = 2$. When g_{11} is split, a_{12} is in the future and g_{12} appears in the past.*

$$\begin{aligned} \text{Split}(1, 2, 0.0, g_{11}) &= ((1.0\ g_{12}), (0.0\ a_{12})) \\ \text{Extract}(1.0\ g_{12}) &= (1.0\ g_{12}) \end{aligned}$$

However, a_{13} which is bounded to g_{12}, must be launched after e_2 (see Figure 4). By recursively applying a predicate missed, g_{12} is split in turn. Thus, a_{13} appears in the future and will be launched after e_2.

$$\begin{aligned} \text{Split}(1, 2, 1.0, g_{12}) &= ((0.0\ a_{11}), (0.5\ a_{13})) \\ \text{Extract}(0.0\ a_{11}) &= \varepsilon \end{aligned}$$

The resulting performance is: $\{(2, 0.0, a_{12}), (2, 0.5, a_{13})\}$

4. A SYNCHRONOUS EMBEDDING

The implementation of the sequencer is divided into two parts (see Figure 7). The module `Time` interfaces the abstract time relative to the tempo (in beats) with the physical time (in ms) (Section 4.2).★[3] The module `Motor` receives the current position in the score and triggers atomic actions (Section 4.3).★

4.1 Survival Kit for ReactiveML

REACTIVEML[4] is a synchronous language which combines features found in ML-like typed functional languages with synchronous features *à la* ESTEREL [2]. It borrows the basic principles and constructs of OCAML[5] on which it is built and compiled to. Like other synchronous languages, it provides a notion of global logical time on which all processes can synchronize. Semantically, processes execute in lock step and they communicate with each other in zero time.

---

[3]The symbol ★ denotes source code available online.
[4]A tutorial is available at http://reactiveml.org/tryrml.
[5]http://caml.inria.fr/ocaml/index.en.html.

Time and Signals.

In REACTIVEML, a *process* is a function that lasts for several logical instants. It is introduced with a special keyword `process`. The following program defines the process `emit_clock` which takes two arguments `period` and `clock`. Its purpose is to emit a signal `clock` every `period` second.★

```
1  let process emit_clock period clock =
2    let next = ref (Unix.gettimeofday () +. period) in
3    loop
4      let current = Unix.gettimeofday () in
5      if current >= !next then begin
6        emit clock ();
7        next := !next +. period
8      end;
9      pause
10   end
```

The variable `next` always contains the desired date for the next emission.[6] This variable is initialized with the current time (in milliseconds), using the function `Unix.gettimeofday` from the `Unix` module, plus the duration of a period. Then the process loops infinitely. At every instant, it computes the current time (line 4); compares it to the value of the desired date `!next`; if `current` is greater than this date, it emits the signal `clock` (line 7) and update `next`. Finally, it awaits for the next instant (line 10). We assume that logical steps are much smaller than `period`, which is the case in practice. Typically, we simulate a clock with a period from one to ten milliseconds.

Using the signal `clock`, one can write a process that does nothing except waiting for a duration `dur` in seconds:

```
let process wait_abs dur period clock =
  let d = int_of_float (dur /. period) in
  do
    for i = 1 to d do pause done
  when clock done
```

The `do`/`when` construct executes its body only when the signal `clock` is present. The only thing to do is to wait for `d` instants: `for i=1 to d do pause done` where `d` is the duration `dur` expressed as a number of instants.

Synchronous Parallelism.

Now, REACTIVEML allows for the definition of processes that run in parallel. For instance, the following process waits for a duration `dur1` and then emits a signal `a`. In parallel, it waits for a duration `dur2` and then emits a signal `b`:

```
let process ab dur1 dur2 a b =
  signal clock in
  let period = 0.001 in
  run (emit_clock period clock) ||
  (run (wait_abs dur1 period clock); emit a ()) ||
  (run (wait_abs dur2 period clock); emit b ())
```

We declare a local signal `clock`. This signal is emitted by the process `emit_clock` with a period `period`. Operator `||` is for parallel composition and `run` denotes the execution of a process. Thus, the process `emit_clock`, and the two calls to processes `wait_abs` are executed in parallel and communicate through the local signal `clock`.

---

[6]`+.`, `*.`, `/.` the floating-point addition, multiplication, division. If `x` denotes a reference, `!x` is its content.

4.2 Following the Tempo

Remember that, in our version of ANTESCOFO, all delays are expressed relative to the tempo in beats. In practice, the tempo is computed by the listening machine: for each detection, the listening machine sends the label of the detected instrumental event and the estimated tempo.

We need to compute the elapsed delay, relative to the tempo, since the beginning of the performance. This is just like calculating the distance an object travels by integrating its speed. It is a simple fixed-step integrator of a piece-wise constant function `bps` (the tempo only changes when an instrumental event is detected). In the following, `clock` denotes a global signal. It will be generated with the process `emit_clock` above, with period `period` defined as a global constant.

The process `elapsed` integrates the value of tempo since the beginning of its execution. Here, `bps` is a signal that carries the value of the tempo in *beats-per-second*. For each step of the integration, we send the result on a signal `date`.★

```
let process elapsed bps date =
  let x = ref 0.0 in
  do
    loop
      x := !x +. last ?bps *. period;
      emit date !x;
      pause
    end
  when clock done
```

The variable `x` contains the current value of the integrator, initialized to 0.0. Thus, at the n-th occurrence of `clock`, the signal `t` is emitted with a value $x(n)$ such that:

$$x(n) = \sum_{i=0}^{n} \texttt{last}(\texttt{bps})(i) \times \texttt{period}$$

The construct `last ?bps` denotes the last value of the signal `bps`. It only changes when a new `bps` value is emitted. Placing the `loop`/`end` construct within a `do`/`when` means that the loop is only executed when the signal `clock` is present.

Now that we can measure time relative to the tempo, we can write a process that waits for a duration `delta` relative to the tempo.

```
let process wait date delta =
  let ending = last ?date +. delta in
  while last ?date <= ending do pause done
```

The signal `date` will be produced by the process `elapsed`. The process `wait` first computes the future date of the end of the waiting: `ending`. We compare the last computed value of `date` with the deadline `ending`, and wait until `last ?date` reaches the deadline. Note that delays ($\delta \in \mathbb{Q}$) relative to the tempo are implemented using floating-point numbers.

In practice, the waiting process is implemented using signals and a priority queue. The process `wait` sends a signal and a deadline to the priority queue. Then the only thing to do is to wait for the return of this signal. Therefore, only the scheduler requires computation at each instant.★

4.3 Translating Semantic Rules

REACTIVEML extends the language OCAML. Therefore, the grammar presented in Section 2.1 can be represented by the declaration of an inductive type (given in Appendix A).★ For every predicate defined in Figure 5, (i.e., *exec*, *detected*, and *missed*), we define a corresponding process in REACTIVEML.★

Execution Rules.

To execute a score, we launch in parallel a process `exec_score_event` for each score event. Therefore, all electronic actions stay synchronous during the performance. In particular, when an event is detected, the execution of missed electronic actions remains synchronous with the execution of detected ones.

```
let rec process exec score =
  match score with
  | [] -> (* rule (Empty Score) *) ()
  | se::sc ->
      (* rule (Exec Score) *)
      run (exec_score_event se) ||
      run (exec sc)
```

Note the similarity with rules (EMPTY SCORE) and (EXEC SCORE) of Figure 5.

The process `exec_score_event` is the implementation of the two rules (DETECT) and (MISS). The parameter `se` is a structure which denotes a score event where `se.seq` is the corresponding sequence of electronic actions and `se.event` is the associated instrumental event. The process `wait_event` awaits an instrumental event. Then, if this event is detected, it returns the value `Detected`. On the other hand, if the event is missed, the process returns the value `Missed(j)` where $j = \mathcal{M}(i)$ is the next detected event (see Section 3.2).

```
let rec process exec_score_event se =
  let status = run (wait_event se.event) in
  match status with
  | Detected ->
      (* rule (Detect) *)
      run (exec_seq (detect i) 0.0 se.seq)
  | Missed(j) ->
      (* rule (Miss) *)
      run (exec_seq (missed i j) 0.0 se.seq)
```

Each score event is related to one instrumental event. Thus, during the performance the only thing to do is to wait for the detection of this event. In REACTIVEML, such waiting requires no computing at all. Indeed, one important characteristic of the REACTIVEML implementation is the absence of busy waiting: nothing is computed when no signal is present.

The process `exec_seq generic delta s` implements the behavior described by the rules (EXEC SEQUENCE) and (EMPTY SEQUENCE) (see Section 3.1). This process computes the delay corresponding to each action in the sequence `s` and runs a process `detected` or `missed` (see below) in parallel for each of these actions with the computed delay. All electronic actions are executed in parallel. Thus nested groups can be treated as atomic actions.

```
let rec process exec_seq generic delta seq =
  match seq with
  | [] -> (* rule (Empty Sequence) *) ()
  | (dae,ae)::s ->
      (* rule (Exec Sequence) *)
      run (generic (delta +. dae) ae) ||
      run (exec_seq generic (delta +. dae) s)
```

Detected and Missed Rules.

The next process is the encoding of the predicate *detected* of Figure 5. It matches the type of an electronic action `ae`, bound to an instrumental event `i` with a delay `delta`, and executes the corresponding behavior. When an atomic action `a` is reached, the corresponding triplet (`i,delta,a`) is

sent on the global signal `perf`. In parallel, another process listens for this signal and sends control messages to the audio environment (Max/MSP).

Here `date` is the signal produced by the process `elapsed` described previously. The function `slice` is the implementation of Slice defined in Section 3.4.★

```
val slice : label -> delay -> group -> score
```

It returns a fresh score where each score event contains one zero delay `loose` group. Then, the process `detected` is implemented as follows:★

```
and process detected i delta ae =
  match ae with
  | Action(a) ->
      (* rule (Detected Action) *)
      run (wait date delta);
      emit perf (i,delta,a)
  | Group(g) ->
      begin match g.group_synchro with
      | Loose ->
          (* rule (Detected Loose Group) *)
          let bg = g.group_seq in
          run (exec_seq (detected i) delta bg)
      | Tight ->
          (* rule (Detected Tight Group) *)
          let gs = slice i delta g  in
          run (exec gs)
      end
```

In the same way, the process `missed` is the transcription of the *missed* rules (see Appendix B).

4.4 Evaluation and Limitations

While the original system is developed as an object in the Max/MSP programming environment, our interpreter communicates with this environment via UDP sockets in a local network. The communication latency between the two applications is negligible (around 10ns). The original sequencer, embedded in the ANTESCOFO object, is disabled as the synchronous one replaces it. Hence, during the performance, Max/MSP sends the output of the listening machine. Then, the synchronous sequencer treats this information and sends control messages back to Max/MSP.

Currently, there is no proper benchmark to evaluate the sequencer part of the system. Each modification is tested with real scores written by composers. Nonetheless, we exercised our application on several toy examples (traditional songs, violin concertos, etc). In these examples, the accompaniment is a set of groups bound to the first note of the performer. They are relatively simple because the electronic part does not have a complex hierarchical structure. However, they are still meaningful for an experimental validation because the sequencer handles a realistic number of electronic actions. For instance, in the second and third movements of the Tchaikovsky's violin concerto, there are 3705 instrumental events for 11062 electronic actions.

The synchronous sequencer is used to control a MIDI synthesizer which accompanies a solo musician. We test these examples with random faults and a randomly changing tempo. On these examples, our sequencer compares well to the original one. Indeed, the latency between the two applications remains under 30ms, the reaction time of the human ear [8].

Nonetheless, there is one major difference between our implementation and the original one. ANTESCOFO is embedded in Max/MSP which provides precise timers and the means to wake-up processes on timers. When ANTESCOFO is activated by Max/MSP, it possibly emits some control messages to Max/MSP, and then computes the next deadline to be activated. Max/MSP wakes ANTESCOFO up when the deadline expires or because an instrumental event is detected. As our implementation is not embedded in Max/MSP, we have experimented with a different approach. The REACTIVEML compiler generates a step function. It is sampled using an (imprecise) UNIX timer on which a regular signal `clock` is built (see Section 4.1). For any sampling value between 1ms (the refresh frequency of Max/MSP) to $30/2 = 15$ms (half of the reaction time of the human ear), the implementation is fast enough. However, if the sampling value is greater than 15ms, one can hear a noticeable delay between electronic and instrumental parts. Besides that, when executed at full speed, the implementation keeps respecting deadlines but monopolizes the CPU uselessly.

Moreover, if the period of the `clock` signal is too small or if the sequencer requires a lot of computations, the sequencer may fail to produce the signal on time. This is of no importance because it does not change the next deadline for the signal `clock`. Thus, if there is nothing to do during the next steps, the sequencer will quickly catch up. In practice, the sequencer waits during most of the steps and catches up very quickly. On the other hand, the period of the `clock` signal is limited by the score. It can not be higher than the minimal delay between two actions.

5. APPLICATIONS

Our implementation provides a powerful tool for prototyping. After defining the semantics of the core of ANTESCOFO, we were able to implement it directly in REACTIVEML. Therefore, new features of the language can be added with little effort. For instance, we proposed two new synchronization strategies `partial` and `causal` which fit perfectly in the semantics framework and have been implemented in REACTIVEML and tested on some examples.

5.1 Prototyping New Features

In the same way, we can rely on REACTIVEML features to implement more complex structures. Assume that we want to introduce a new preemption construct to stop the execution of a sequence when a particular instrumental event is detected. Typically, this construct is difficult to add to an implementation that only uses priority queues. But, thanks to the preemptive construct `do`/`until`, the integration of this new feature in the language is immediate. The only thing to do is to add a case in the processes `detected` and `missed`. Here, `u` is an instance of our new construct, `u.until_seq` denotes a sequence of electronic actions and `u.until_event` denotes the control event.★

```
and process detected i delta ae =
  match ae with
  ...
  | Until(u) ->
      signal kill in
      do
        (run (exec_seq (detected i) delta u.until_seq);
         emit kill)
        ||
        (run (wait_event u.until_event);
         emit kill)
      until kill done
```

5.2 Towards New Interactions

Moreover, coupling the sequencer with REACTIVEML via signals is another valuable asset for prototyping. Indeed, in our application, actions consist in either sending a control message to Max/MSP or emitting a REACTIVEML signal (see Appendix A). Therefore, it is easy to link a score with a REACTIVEML program. One of the main advantages of our approach is that the program and the sequencer remain synchronous. We can use this property, for instance to write a performance simulator which runs in parallel with the sequencer and sends messages containing the index of an instrumental event and the value of the tempo, instead of the listening machine.

This technique can also be used to write pieces with complex interactions between the performer and the accompaniment. For example, *Piano Phase* is a piece written in 1967 by the minimalist composer Steve Reich. In this piece, two pianists begin by playing a rapid twelve-note melodic figure over and over again in unison. Then, one of the pianists begins to play his part slightly faster than the other. When the first pianist plays the first note of the sequence as the second is playing the second note, they resynchronize for a while. The process is repeated, so that the second pianist plays the third note as the first pianist is playing the first, then the fourth, etc.

We implemented this piece using both our sequencer and REACTIVEML.♩ The real musician plays the part with the constant tempo, while the accompaniment alternates between desynchronization and resynchronization. First, the two pianists play at the same speed. Then, the electronic accompaniment begins to play its part slightly faster and emit a signal `first_note` each time it plays the first note of the sequence. Meanwhile, we track the position of the performer and emit a signal `kth_note` each time the performer plays the second note of the sequence. When the two signals are close enough we resynchronize the accompaniment and the two pianists play at the same tempo again. Then, we restart the desynchronization but, this time, we track the third note of the performer, and so on. The difficulty comes from the fact that we do not know a priori when the pianists resynchronize (in the original score, the composer only specifies bounds on the number of iterations of the sequence during the desynchronization). Programming such a behavior is not possible to achieve with the language described in [5]. In ANTESCOFO, the electronic part only synchronizes using the output of the listening machine and a static score. Here, the sequencer has to synchronize with both the listening machine and an external program running in parallel and sharing the same notion of time. A very recent extension of the ANTESCOFO language, developed independently from the present work, allows the simulation of this new feature [4].

5.3 Toplevel and Live Coding

Finally, REACTIVEML has a toplevel [10], and it is thus possible to dynamically write, load and execute REACTIVEML code. This feature allowed our implementation to be extended to support live coding. It is now possible to write and/or correct an accompaniment during the performance using our kernel of the ANTESCOFO language.♩

Dedicated languages and applications already exist for live coding (i.e., real-time synthesis and control during a performance) [14, 12]. The main advantage of our proposal is

that composers can rely on the features of the ANTESCOFO language during live coding, e.g., for synchronization and error handling strategies. However, it only works in interaction with Max/MSP. Therefore it is much more limited than languages dedicated to, and optimized for, live coding.

6. RELATED WORK AND DISCUSSION

Differences with the original language.
The most notable differences between this core language and ANTESCOFO as defined in [5] are the specifications of group attributes. In the original language, there are only two error handling attributes: `global` (`g`) and `local` (`l`) whose meaning depends on the synchronization strategy. To make synchronization and error handling attributes independent of each other, we have changed the meaning of the `local` and `global` attributes and introduced the new attributes `partial` and `causal`. The 2×4 following combinations are now possible whereas the original system only provided 4. The combination of `tight` (`T`) and `partial` was named `tight local` (`T l`) and the combination of `tight` and `causal` was named `tight global` (`T g`):

	local	global	partial	causal
tight(T)	.	.	T l	T g
loose(L)	L l	L g	.	.

Furthermore, the semantics proposed in [5] is based on a normalized subset of the language: a pre-processing converts `tight` groups into `loose` groups, and `tight` groups nested inside a `loose` group are interpreted as `loose` groups. There is no such pre-processing in our proposal. Our semantics gives a precise meaning to arbitrary nestings of any kind of group and directly reflects the hierarchical structure of a score.

We focus here on a subset of the language, but the omitted control structures can be easily expressed in our kernel. For instance, in the original language, a loop is just a succession of groups characterized by the same attributes. In the same way, the original language allows the use of *continuous* groups [5]. Messages contained in a continuous group correspond to the sampling of piece-wise linear functions specified by a sequence of control vectors. Thus, it can be easily translated into a simple group containing all the desired actions. As explained above, we can attain sampling values down to 1 ms which is comparable to the current ANTESCOFO system. This is satisfying enough for control, e.g., of amplitude, but not for audio synthesis for which a sampling period of 0.02 ms is needed.

The original language offers the possibility to specify delays in seconds instead of delays relative to the tempo. This feature can easily be added to our interpreter. The only change required is to use the process `wait_abs` defined in Section 4.1 instead of `wait` for delays expressed in seconds.

Finally, a denotational semantics in term of a datation function of actions can be found in [6]. We believe that the deduction rules of Section 3 are easier to understand, permit some reasoning and a natural translation into a synchronous implementation. Moreover, this formalism is also more open in the sense that it is easy to extend with new constructs like new group attributes. It is not clear for instance whether the error handling attributes `partial` and `causal` can be described in the semantics of [6].

Modeling of real-time.

Recent work in the Berkeley PTIDES project [15] extends the PTOLEMY[7] system with real-time features. Basically, signals are tagged with time-stamps as they move through a distributed data-flow network: input tokens are stamped at occurrence, dates are incremented as tokens pass through delays within the system and finally actuator tokens are queued to take effect when their time stamps expire. In our context, time-tags are not simply a real-time date but a couple (i, δ) where i is a detected event and δ a delay relative to the tempo. Tags are completely ordered with the following relation:

$$(i_1, \delta_1) < (i_2, \delta_2) \Leftrightarrow \mathcal{E}(i_1) + \delta_1 < \mathcal{E}(i_2) + \delta_2$$

Thus, our semantics rules can be viewed as a way to compute tags associated to each atomic action.

Language Embedding.

This work is influenced by Conal Elliott's shallow embedding of FRAN [7], a language for defining reactive animations in HASKELL. We took a similar approach by embedding a domain-specific language inside a functional language. Yet, our approach is novel as we embed a reactive language into a more expressive one. By reusing the synchronous parallelism of REACTIVEML, we reduced much of the work of compilation. Moreover, the implementation closely follows the semantics rules.

7. CONCLUSION

In this paper, we proposed a semantics for the core language of ANTESCOFO, a system dedicated to mixed music. This semantics highlights its synchronous nature: all running processes execute in lock-step according to a global time scale. Moreover, this semantics naturally yields a synchronous implementation. We used REACTIVEML for its expressiveness and efficiency.

By embedding ANTESCOFO in an existing synchronous language, much compilation work is avoided. The implementation is small (a few hundred lines of REACTIVEML) and efficient enough so that no perceptive difference has been noticed with respect to the current implementation. We experimented with it on several musical pieces and computation delays were always below human ear tolerance (30ms).

This embedding allows the writing of complex reactive programs in REACTIVEML capable of interacting with the rest of the musical score. We believe therefore that our application is a powerful tool for prototyping all kinds of new features such as new synchronization or error handling strategies, but also for other types of new interactions between live performers and computers.

Acknowledgement

---

[7] http://ptolemy.berkeley.edu/index.htm

8. REFERENCES

[1] 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.

[2] G. Berry and G. Gonthier. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

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

[4] J. Echeveste. Modeling Steve Reich's Piano Phase with the new version of Antescofo. Personal communication, April 2013.

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

[6] J. Echeveste, A. Cont, F. Jacquemard, and J.-L. Giavitto. Formalisation des relations temporelles entre une partition et une performance musicale dans un contexte d'accompagnement automatique. In *Modélisation des systèmes réactifs*, volume JESA45:1-3, pages 109–124, Novembre 2011.

[7] C. Elliott and P. Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273, 1997.

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

[9] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet. A modular memory optimization for synchronous data-flow languages. In *Languages, Compilers and Tools for Embedded Systems*, June 12-13 2012.

[10] L. Mandel and F. Plateau. Interactive programming of reactive systems. In *Proceedings of Model-driven High-level Programming of Embedded Systems*, 2008.

[11] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, 2005.

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

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

[14] G. Wang and P. Cook. Chuck: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *Proceedings of the international conference on Multimedia*, pages 812–815. ACM, 2004.

[15] J. Zou, S. Matic, E. A. Lee, T. H. Feng, and P. Derler. Execution strategies for PTIDES, a programming model for distributed embedded systems. In *Real-Time and Embedded Technology and Applications Symposium, 2009. 15th IEEE*, pages 77–86, 2009.

APPENDIX

A. STRUCTURE OF THE SCORE

We present here the translation of the grammar presented in Section 2.1 in terms of OCaml types.

```
type delay = float
type instr_event_label = int
type tempo = float

(* Asco action: control message or RML signal *)
type action =
  | Message of string  (* Max/MSP control message *)
  | Signal of (unit, unit list) event  (* RML signal *)


(* Synchronization strategies *)
type sync = Tight | Loose

(* Error handling strategies *)
type err = Local | Global | Causal | Partial

(* Asco group *)
type group =
    { group_synchro : sync;
      group_error : err;
      group_seq : sequence; }

(* Generic asco action *)
and asco_event = Group of group | Action of action

(* Sequence *)
and sequence =  (delay*asco_event) list

(* Electronic score event *)
type score_event =
   { event : instr_evnet_label;
     seq : sequence; }

(* Score *)
type score = (score_event list)

(* Instrumental score *)
type instr_score = delay array

(* Performance element *)
type perf_elem = instr_event_label * delay * action
```

B. THE MISSED RULES

The process `missed` is the transcription of the *missed* predicate (see Figure 5). Here `ae` is an electronic action related to a missing event i, with a delay `delta`, and $\mathcal{M}(i) = j$ (see Section 3.2). The function `split` is the implementation of Split (see Section 3.5). It returns two sequences of actions, the *past* and the *future*.★

```
val split :
  label -> label -> delay -> group -> sequence * sequence
```

The function, `extract_groups` is the implementation of Extract (see Section 3.5). It returns groups contained in a sequence.★

```
val extract_group : sequence -> sequence
```

Finally `instr_sc` is a map structure that contains dates of instrumental events. It plays the role of the function $\mathcal{E}$ (see Section 3.2). Then, the process `missed` is implemented as follows:★

```
and process missed i j delta ae =
  let dj = instr_score.find j in
  let di = instr_score.find i in
  match ae with
  | Action(a) ->
      (* rule (Missed Action) *)
      let d = (max 0.0 (delta +. di -. dj)) in
      run (wait d);
      emit perf (j,d,a);
  | Group(g) ->
      begin match g.group_error with
      | Local -> (* rule (Missed Local Group) *) ()
      | Global ->
          (* rule (Missed Global Group) *)
          run (detected j 0.0 ae)
      | Partial ->
          (* rule (Missed Partial Group) *)
          let past, future =
            Groups.split instr_score i j delta g in
          let gpast = Groups.extract_group past in
          let gfuture =
            Group({group_synchro = g.group_synchro;
                   group_error = g.group_error;
                   group_seq = future;})
          in
          (run (exec_seq (missed i j) delta gpast) ||
           run (detected j 0.0 gfuture))
      | Causal ->
          (* rule (Missed Causal Group) *)
          let past, future =
            Groups.split instr_score i j delta g in
          let gfuture =
            Group({group_synchro = g.group_synchro;
                   group_error = g.group_error;
                   group_seq = future;})
          in
          (run (exec_seq (missed i j) delta past) ||
           run (detected j 0.0 gfuture))
      end
```

Note that processes `exec`, `detect` and `missed` are mutually recursive. Indeed, the process `exec` launches `detect` or `missed`, `detect` sometimes executes `exec`, and `missed` sometimes launches `detect`. Thus the three processes must know each other.

C. SOURCE LINES OF CODE

We computed the number of lines of code with `ocamlwc`.[8]

File	sloc
types.rml	35
time.rml	60
motor.rml	97
groups.rml	65
input.rml	87
output.rml	33
ascolib.rml	210
asco.rml	10
parser.mly	104
lexer.mll	84
reactive_queue.rml	92
reactive_map.rml	70
utils.rml	41
config.rml	104
network.rml	20
Total	**1111**

---

[8] https://www.lri.fr/~filliatr/software.en.html