Time, Timelines and Temporal Scopes in the Antescofo DSL v1.0

Jean-Louis Giavitto, José-Manuel Echeveste, Arshia Cont, Philippe Cuvillier

Time, Timelines and Temporal Scopes in the *Antescofo* DSL *v*1.0

Jean-Louis Giavitto
CNRS UMR STMS 9912
IRCAM & UPMC & Sorbonne University, Paris
giavitto@ircam.fr

José Echeveste, Arshia Cont, Philippe Cuvillier
Antescofo
Agoranov, Paris
name@antescofo.com

ABSTRACT

This paper presents the model of time developed in the version 1.0 *of the* Antescofo *system.* Antescofo *integrates a listening module with a dedicated reactive and timed real-time programming language used to define the electronic actions to be performed in sync with a human performer. Since its beginnings, the model of time supported by the DSL has been developed and enriched, going from implicit relationships to denotable entities in* v1.0*. This paper focuses on the simultaneity and succession relationships that organize actions on a timeline, the creation of multiple independent timelines and on the notion of temporal scope that defines how time passes on a timeline relatively to the occurrence of events and to the fluence of another timeline. Temporal scopes offer generic and expressive ways to "play in time" the electronics with a human performer.*

1. INTRODUCTION

Antescofo is system coupling a *listening module* and a *domain specific programming language* (DSL). It is used by music composers, and more generally by interactive multimedia designers, to specify and to implement *augmented scores*, *i.e.*, temporal scenarios where electronic musical processes are computed and scheduled in interaction with a live musician performance. Interaction scenarios are expressed at a symbolic level through the specification of musical time in the score (musical events like notes and beats in relative tempi) and the management of the physical time of the performance (with relationships like succession, delay, duration, rate... of events occurrence on stage).

During the performance, human performers "implement" the instrumental part of the augmented score, while the language runtime evaluates the electronic part with the help of the information provided by a listening module, to control and synchronize the electronic actions with the musical environment.

Antescofo bridges the gap between the composition and the performance, which requires the handling of several notions of time. The composer defines *potential temporal relationships* between musical entities in an augmented score. These relationships are expressed through the temporal relationships between *Antescofo* actions arranged in timelines and musical events. During the performance, the potential temporal relationships specified by the composer become *actual* with the realization of musical events by the musicians and the computation of electronic actions by the computer. A unique feature of *Antescofo* is that the composer is able to specify constraints between the potential and the actual temporal relationships.

This paper introduces the various temporal notions at work in *Antescofo*. It presents the dynamic control structures used to specify sequences of actions into *timelines* and how a timeline gathers both event-driven and time-driven relationships.

Since the version *v*1.0, the *passing of time*, *i.e.*, the progression on a timeline, is also a denotable entity in the language: the notion of *temporal scope* defines the advancement on a timeline relatively to another timeline. This progression accommodates the event-driven view and the timed view of time. Temporal scopes make possible to synchronize with external systems that have their own non-deterministic timelines like a human musician.

Organization of the paper. Next section presents the handling of succession and simultaneity in *synchronous languages*, a successful approach developed in the realm of real-time programming for the development of embedded systems. Section 3 compares the *Antescofo* specification of succession and simultaneity with the approach developed by *ChucK* [1], another DSL in computer music that fulfills the *synchrony hypothesis* and provides a strong and coherent time model. Section 4 introduces the notion of temporal scope that defines how the beats on a timeline are converted into physical time. This conversion is driven by the progression on another timeline, a mechanism that provides all features needed for playing "in time".

2. TIME IN PROGRAMMING LANGUAGES

The classical analysis of time in philosophy distinguishes between two temporal entities, *instant* and *duration* that are linked by three temporal modes or relationships: *succession*, *simultaneity* and *permanence*. This analysis can be used to classify programming languages and computer music systems by their handling of instant and duration:

- Dealing with succession and simultaneity of instants leads to the *event-triggered* or *event-driven* view, where a processing activity is initiated as a consequence of the occurrence of a significant event. For instance, this is the underlying model of time in MIDI.
- Managing duration and permanence points to a *time-triggered* or *time-driven* view, where activities are initiated periodically at predetermined date and last. This is the usual approach in audio computations.

These two points of view [2] are supported in *Antescofo* and the composer/programmer can express his own musical processes in the most appropriate style.

2.1. Instants and Succession: Sequential Languages

Sequential programming languages usually deal only with instants (which are the location in time of elementary computations) and their succession. The actual duration of a computation does not matter, nor does the interval of time between two instants: these instants are atomic *events*.

This model is that of MIDI: basic events are *note on* and *note off* messages. There is no notion of duration in MIDI: the duration of a note is represented by the interval of time between a *note on* and the corresponding *note off* and it has to be managed externally to the MIDI device, *e.g.* by a sequencer. In addition, two MIDI events cannot happen simultaneously. So we cannot say for instance that a chord starts at some point in time, because starting the emission of the notes of the chords are distinct sequential events.

There is two "sources of succession" in programming languages: explicit succession, as specified with the explicit sequence of statements in a language like C or the ; operator in Pascal ; and the *causality* which implies for instance that the condition of a conditional must be calculated before computing the selected branch.

2.2. Simultaneity

In a purely sequential programming language, it is very difficult to do something at a given date. We can imagine a mechanism that suspends the execution for a given duration and wakes up at the given date, as in

```
sleep(12 p.m. - now()) ;
computation to do at 12 p.m.
```

or if we have a mechanism that suspends the execution until the arrival of a date

```
wait(12 p.m.) ;
computation to do at 12 p.m.
```

or until the occurrence of an event:

```
wait(MIDI message) ;
process received message
```

Notice that the computation resumes *after* the date or the event. On a practical level, this is usually negligible (*e.g.*, chords can be emulated in MIDI using successive events). However, at a conceptual level, it means that *simultaneity* cannot be directly expressed in the language, which will make the specification of some temporal behaviors more difficult.

To express simultaneity in the previous code fragment, we have to imagine that computations happen infinitely fast, allowing events to be considered atomic and truly synchronous. This is the **synchrony hypothesis** whose consequences have been investigated in the development of synchronous languages dedicated to the development of real-time embedded systems like Esterel [3], Lustre or Lucid Synchrone [4].

2.3. Synchronous Languages

Synchronous languages have not only postulated infinitely fast computations, allowing two computations to occur simultaneously, they have also postulated that *two computations occurring at the same instant are nevertheless ordered*. This marks a strong difference between simultaneity and parallelism (more on this below) and articulates, in an odd way, succession and simultaneity (one relationship does not imply the negation of the other).

However, there are no logical flaws in this idea [5]. Much better, this hypothesis reconciles determinism with the modularity and expressiveness of concurrency: at a certain abstraction level, we may assume that an action takes no time to be performed (*i.e.* its execution time is negligible at this abstraction level) and we may assume a sequential execution model (the sequence of actions is performed in a specific and well determined order) which implies deterministic and predictable behavior. Such determinism can lead to programs that are significantly easier to specify, debug, and analyze than nondeterministic ones [6].

A good example of the relevance of the synchrony hypothesis in the design of real-time systems is the sending of messages, in MAX or PureData, to control some device. To change the frequency of an ugen, the generator must have already been turned on. But there is no point in postulating an actual delay between turning on the generator and changing its frequency default value. The corresponding two messages are sent in the same logical instant but in a specific order. Another example is audio processing when computations are described by a global dataflow graph. From the audio device perspective, time is discretized in instants corresponding to the input and the output of an audio buffer. In one of these instants, all computations described by the global graph happen together. However, in this instant, computations are ordered, *e.g.* by traversing the audio graph in depth-first order from audio sources to sinks.

3. SIMULTANEITY AND SUCCESSION IN *ANTESCOFO*

Action is the name used in *Antescofo* to refer to some computations. *Elementary actions* are basic predefined operations. They can be instantaneous (assignation, sending a message, evaluating an expression) or they can have duration. An example of elementary action that has a non-null duration is the curve, which interpolates between parameters during a definite time interval.

Compound actions organize the temporal relationships of several sub-actions. *Succession* is one of such constructions.

3.1. Succession Operators

They are five succession operators in *Antescofo*: the *juxtaposition*, the *delay*, the *followed-by* and the *ended-by*. These binary operators are associative at the exception of the delay.

Juxatposition. The juxtaposition operator is implicit: two actions a and b that appears one after the other, as a b in the score, are performed simultaneously.

Delay. The delay operator is used to launch an action b after a specified duration has elapsed. The delay begins with the start of action a. The operator is written in infix form using the specified duration as the operation. For example,

"a 5 b" will start b 5 beats after the start of a. Notice that "a 0 b" is equivalent to "a b".

They are several ways to specify a delay. An expression *e* evaluating into a numerical value always refers to the passing of time in the current temporal scope (whose unit is always called a *beat*). The labeling of the expression by s or ms defines a delay in seconds or milliseconds, that is, in the scope of the physical time: a 5s b will start b five seconds after the start of a.

Notice that this may seem similar to "chunking" a delay in *ChucK*. Indeed,

```
a; 5::s => now; b
```

in *ChucK* gives the same results as "a 5s b" in *Antescofo* if we suppose that a and b are instantaneous computations. This is no longer true if a has some duration as discussed in sect. 3.3.

Followed-by and Ended-by. These two operators are similar to the juxtaposition but differs with respect to the reference point used for the succession. Instead of considering the start of a, the `==>` (followed-by) and the `+=>` (ended-by) operators consider the termination of a: a `==>` b will trigger b at the end of a; if a is a compound action that spawns sub-actions, `+=>` will further wait the termination of all sub-actions (recursively) before launching b.

This notion of succession is a *semantic* one, not a syntactic one as for the usual succession operator. For example, if ::P is a process (process identifiers begin with :: in *Antescofo*), then

```
::P()  +=>  print "stop"
```

will print "stop" at the termination of ::P and all of the spawned children, not after the launch of ::P. For this reason, operators `==>` and `+=>` are similar to *continuation* operators [7].

3.2. Control Structures

Succession operators define a local temporal relation between two actions. Additional control structures, like iteration, process call and reaction, can be used to specify more global relationships (others control structures available in *Antescofo* are not addressed in this paper).

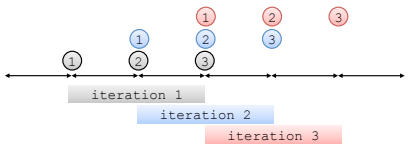
Iteration. The construction "loop p a" iterates the execution of a every p. Expression p defines a time interval. It can be a constant to define a periodic iteration, or its value can change during loop's execution. For instance

```
$p := 1
loop $p {
  $p := $p * 0.95
  a
}
```

defines an exponential *accelerando* for the iterations of action a (in *Antescofo*, variable identifiers begin with $).

Each instance of the loop body runs in its own thread. So instances of loop bodies may overlap in time if the duration of a loop body exceeds the period p. For example the loop at left produces the trace given on the right:

```
loop 1 {
    print 1
  1 print 2
  1 print 3
}
```



Process Call. An *Antescofo* program can be parametrized and abstracted in a process definition. This process can later be called as an action. The call itself is atomic and the process runs in its own thread.

Process definitions are first class values. They can also be recursive. For example

```
@proc_def ::Tic($d) { $d print tic }
@proc_def ::Toc($d) { $d print toc }
@proc_def ::Clock($p, $q) {
    :: $p (0)
    :: $q (1)
  2 ::Clock($p, $q)
}
```

defines three processes. Processes ::Tic and ::Toc are elementary actions performed after a delay given in parameter. The last process ::Clock calls the two processes given as arguments and then calls itself recursively after a delay. This recursion is not bounded but it implies a delay. So there is no accumulation of actions to perform before a given date. The net result is equivalent to

```
loop 2 {
  :: $p(0)
  :: $q(1)
}
```

So the call ::Clock(::Tic, ::Toc) spawns a periodic clock that alternates the triggering of the actions in parameter every beat.

Reaction. The previous control structures initiate an action relative to the beginning (or the end) of another action, or after a delay has elapsed. *Antescofo* introduces several ways to start an action in reaction to an event. They are several kind of events: the performance of a musical event, the reception of a Max, PD, OSC message... and the occurrence of an arbitrary logical condition. Reactions to the last kind of events are handled by the whenever control structure that launches an action a every time its condition *cond* is fulfilled:

```
whenever(cond) a
```

The condition *cond* characterizes a specific instantaneous state in the system, expressed as a logical expression like "$x + $y > 3" (action a is launched each time the sum $x + $y changes its value for a value greater than 3). The condition can also specify a durative state called *temporal patterns* [8]. For instance, the complex event: "variable $pitch takes the same value $x during at least 2 beats and then is assigned to the same value $x before 1 beat has elapsed", is defined by the construction:

```
@pattern_def lap {
  @local $x
         state $pitch value $x during[2]
  before[1] event $pitch value $x
}
```

Temporal Restriction. Without other indications, a loop runs forever and a whenever watches the variables in its condition until the end of the program. *Bounding guards* can be specified to restrict these lifetimes or the lifetime of any other action. The interval on which a loop runs, can be specified by a logical guard: the loop stops its iteration when a logical expression becomes true (but the existing instances of the loop body continue their executions). This interval can also be specified directly as a time interval using the during clause:

```
loop 1 { a } until ($x == 3)
loop 1 { a } while ($x != 3)
loop 1 { a } during [3]
loop 1 { a } during [3s]
loop 1 { a } during [3#]
```

The sharp # in the last during clause is similar to the label s and denotes logical time: the loop body is instantiated exactly 3 times before aborting the loop. Notice that durative actions can also be explicitly aborted.

3.3. Parallelism and Simultaneity

We mentioned that chuncking a delay in *ChucK* is not equivalent to the *Antescofo*'s delay operator. Starting an activity (after a delay) with the start of another one, requires implicitly that all activities are independent and run in parallel. This is apparent in "a 1 b" if we replace a by an action that takes some time like a loop. In

```
    loop 1ms { c }
1ms b
```

the action b is started at 1ms, simultaneously with the second iteration of the loop. In the following *ChucK* code, action b is never performed:

```
while (true) { c; 1::ms => now; };
1::ms => now; b
```

The interpretation of "a 1s b" and "a; 1::s => now; b" differs greatly: the *ChucK* program is a sequential program that is stopped for a given duration when chuncking the delay, while the *Antescofo* program describes two parallel activities that are shifted in time. To achieve the *Antescofo* semantic in *ChucK*, one has to explicitly use *shreds* (*i.e.* ChucK threads) to make the actions independent:

```
spork ~ a;
1::s => now;
spork ~ b;
```

(the spork operator forks a new shred). The previous example points the difference between *Antescofo* and *ChucK*, and more generally, the approach taken by imperative synchronous languages: despite the fact that they all fulfill the synchrony hypothesis, computations happen infinitely fast and sequentially in existing imperative synchronous languages, while computations in *Antescofo* are infinitely fast and occur in parallel.

This formulation may seem absurd until one realizes that, here, parallelism refers to a logical notion related to the structure of the program evaluation and is not related to an operational property of the execution. In this view, a program is parallel if several threads describe the progression of the computation. Here a thread corresponds to an instruction counter that points in a succession of actions. In the case of *ChucK*, all control structures are sequential (*i.e.* takes place in one thread) except the explicit thread creation operation spork. On the other hand, threads are implicit in *Antescofo* and are derived from the simultaneity and the succession structure of compound actions.

Coroutines. Similarly to *ChucK*, *Antescofo* threads of activities are implemented using *coroutines* [9], not processes or posix threads. The concept of coroutines was introduced in the early 1960s and constitutes one of the oldest proposals of a general control abstraction. The notion of coroutine was never precisely defined, but three fundamental characteristics of a coroutine are widely acknowledged:

- the values of data local to a coroutine persist between successive calls;
- the current execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage;
- they are non-preemptive: coroutines explicitly transfer control among themselves with some control-transfer operations (there is no preemption, nor interruption).

In addition, *Antescofo*'s coroutines have several specific features. There is no explicit coroutine creation. They are first-class objects that can be freely manipulated by the programmer. There is only one control transfer operation: waiting for a delay. This operation corresponds to the *yield* operation used to suspend a coroutine execution: the coroutine's continuation point is saved so that the next time the coroutine is resumed, its execution will continue from the exact point where it suspended. But in *Antescofo* there is no explicit *resume* operation: they are implied implicitly by the succession relationships and the passing of time.

A coroutine is a sequence of instantaneous actions interleaved with delay. So each action a in a coroutine has a date that corresponds to the sum of delays that precedes a. Coroutines have a priority, so the actions that must be run at a given date (in the same instant) are unambiguously ordered. For example, in

```
{ 2 a 3 b }
{ 3 c 2 d }
```

actions b and d occur at the same date 5. Their execution is nevertheless ordered deterministically (by their order of appearance in the score). The fact that b and d are simultaneous can be here determined statically, *i.e.*, prior the program execution. But in general, delays are defined by arbitrary expressions and the simultaneity relationship cannot be determined statically.

4. SHARING TIMELINES AND COLLECTIVE PERFORMANCES

The previous operators locate actions on a timeline. To tackle two fundamental problems faced by *mixed music*, *Antescofo* introduces the handling of multiple timelines related by synchronization strategies.

In the context of written music, mixed music is defined as the association in live performance of human musicians and computer mediums interacting in real-time. Mixed music raised two problems to the computer part:

1. music as a performance,
2. and performance as a collective process.

The first point refers to the divide between the score and its realization. Usually, notation does not specify all of the elements of music precisely, which leaves room for *interpretation*. The score can be thought of as a set of constraints that must be fulfilled by the interpretation but many scores' incarnations may answer these constraints. The interpretation matters, conveying some meaning and assigning significance to the musical material. It is the performer's responsibility to choose/implement one of these possible incarnations. In doing so, the performer takes many decisions based on performance practice, musical background,

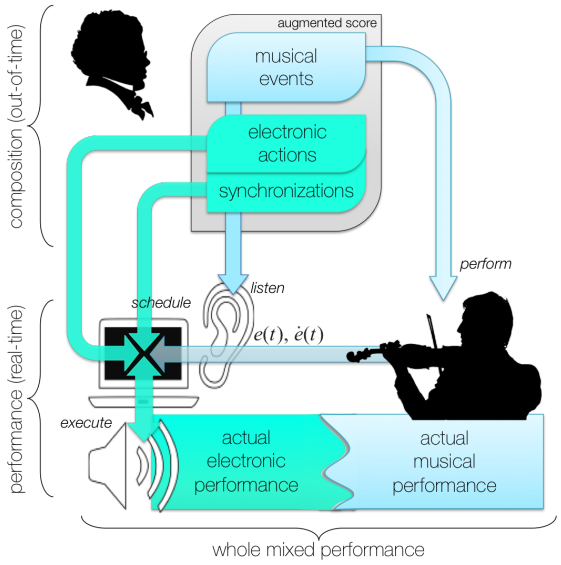


Figure 1. Mixed Music

individual choices and also because he is part of an ensemble: the music is played together with other musicians and the collective will dramatically affect the interpretation.

These two points challenge mixed music: how should various prescriptions of rhythm, tempo, dynamics and so on, be precisely realized by a computer w.r.t. their specification in the augmented score? Computers cannot make these decisions out of the blue and, in addition, have to take the other performers into account.

The *Antescofo* answer is to let the programmer specify electronic actions on a timeline T *but to parametrize the progression of time on* T *relatively to (the progression of time in) another timeline* U.

This approach corresponds to a big shift of paradigm in mixed music and score following: U can be the progression of one musician on its own score and T the schedule of the electronic accompaniment. Electronic actions are not triggered on the occurrence of some musical events, but rather the timelines of the electronic actions are aligned (synchronized) with the timeline of the performer(s).

4.1. Timelines

A timeline is a common temporal reference frame: atomic actions on the same timeline have a date expressed in beats, and these dates can be compared and ordered. A date can be defined by the occurrence of an event (in this case it cannot be located in the physical time before its occurrence) or by the expiration of a delay starting from another date. In addition, the handling of duration requires a notion of rate (or speed), the *tempo*, which drives the beat/time mapping in-between events.

Synchronous languages are based on a discrete notion of time. Physical time is typically handled as an external event, for instance the reception of a periodic signal (*e.g.*, the audio interrupt). This approach presents two drawbacks: *(i)* action launched at a given date cannot be anticipated (in a discrete-event system, the interval of time between events is irrelevant); and *(ii) duration* is not in the domain of discourse. As a consequence, *continuous notions* like "going twice slower", "accelerando" or "playing this phrase from here to there" [10], are difficult to express, if not impossible.

Most real-time computer music programming languages rely on a discrete-event model of time and timed relations are emulated by counting the quantum of time provided by a clock. However, robust discrete-event emulation of continuous time is difficult (think to the accumulation of rounding errors in the iteration of a loop), it is a burden to do it explicitly and it is often expensive (computations occur at each quantum of time).

Antescofo handles both events and continuous duration. The latter are managed through the notion of tempo, which drives the mapping of the relative date on a timeline (expressed in beats) to the physical time (expressed in seconds). The former are shared between timelines: an atomic event is a synchronization point that is simultaneously seen by all timelines

4.2. Temporal Scopes

The way time progresses on T relatively to U is defined by a *temporal scope* $\mathcal{T}_U$. A temporal scope instantiates a *synchronization strategy* which defines the temporal relationships to maintain between T and a referred timeline U. The temporal scope makes explicit what means "playing in time with U" without the timelines being identical.

Each timeline has a unique temporal scope and each action refers to a temporal scope. If two actions share the same temporal scope $\mathcal{T}_U$, they are located on the same timeline and they progress in the same way relatively to U: they see the same events and they progress at the same speed.

The job of the *Antescofo* runtime is to compute as soon as possible these dates and the tempo to schedule the actions at their corresponding date and at the correct rate. Some preliminaries synchronization strategies have been presented in [11]. The novelty with the version $v1.0$ is that synchronization strategies can be the result of a computation and that temporal scopes are explicit, making possible to parametrize a compound action with a temporal scope or to pass a temporal scope as an argument of a procedure.

4.3. Dynamic Definition of Timelines

Primitive timelines are timelines corresponding to the evolution of external processes, like human performers. External processes notify to *Antescofo* the occurrence of events and a tempo extraction algorithm [12] is used to attach to each event e, an associated tempo $\dot{e}$. Physical time is a primitive timeline with no events and a fixed tempo of one beat per second.

By default, child actions inherit the temporal scope of their parent. The programmer/composer may label an action by a synchronization strategy (qualifiers begin with @...) leading to the creation of a new temporal scope. Creation is lazy: temporal scope with the same behavior are shared. The programmer may refer to an existing temporal scope, for instance to specify that a compound action must be performed in sync with another compound action (by sharing the same temporal scope). However *Antescofo* does not introduce a new type of values to represent temporal scope: they are managed through the notion of coroutine: each coroutine refers to one temporal scope

(the temporal scope of the actions performed by the coroutine). Coroutines are first class values and are used to refer to a temporal scope where this is needed.

5. CLASSICAL EXAMPLES

Antescofo provides a whole spectrum of synchronization strategies following the use of the information of position and the information of tempo. In the following examples, we use these synchronization strategies to emulate in *Antescofo* the synchronization capabilities provided by the *Ableton Link protocol* [13]. We suppose that we have a process ::P running and we want to specify its progression relatively to some information coming from the environment through an OSC message /sync at port 43210:

```
oscrecv sync 43210 /sync $s
```

This *Antescofo* statement creates an OSC receiver which dispatches the argument of the message into variable $s.

Tempo synchronization is easy if this information is provided as the argument of the message. The action

```
::P() @tempo = $s
```

calls process code::P and executes it in a new timeline which is defined by a tempo $s and no event. When a new tempo is broadcasted, the progression of the actions spawned by ::P changes accordingly.

Suppose that the tempo changes by increment. A way to smooth the tempo changes is to proceed gradually. We can use a reaction to trigger a loop that will increase or decrease gradually the tempo applied to ::P. This is easily written:

```
whenever ($s) {
     abort $l
     $delta = ($s - $is)/100
     $l := loop 0.1 { $is += $delta }
           until ($is >= $s)
}
::P() @tempo = $is
```

The tempo of ::P is now defined by variable $is. When a new tempo is received, the whenever is activated and trigger a loop which will increase or decrease the ::P's tempo every 0.1 beat until reaching the new value by steps of $delta. The $l variable records the coroutine corresponding to the loop. This reference is used when a new tempo is received to abort the eventual running loop (if there is still one) before to start the new one.

One may want to refer to the ::P's temporal scope, to use it on another process. This is easily done through the coroutine that performs P:

```
$p := ::P() @tempo = $is
```

Then we can launch ::Q on the ::P's timeline:

```
::Q() @sync = $p
```

The @sync attribute specifies the timeline to follow. This example is not very informative because we have explicitly specified the temporal scope of ::P, so we can do the same for ::Q. However, the idea is that $p may result from arbitrary complicated expression evaluated in other part of the score.

Usually, musical applications require beat alignment in addition to tempo synchronization. We suppose that the external process send an osc message every beat. So, before the definition of the osc receiver, we specify that variable $s is used to record the occurrences of event and must compute tempo information:

```
@tempovar $s(60, 1)
```

This definition specifies a variable which expects a periodic assignment. This variables refers to a timeline where each assignment is located one beat after the previous one with a nominal tempo of 60. With this setting, the call

```
::P() @sync = $s
```

will synchronize with the timeline computed from $s. The default synchronization strategy is to synchronize only with the tempo. So here we synchronize actions in ::P with the tempo extracted automatically from the speed of assignment. We can specify that we want to synchronize also on each event (that is, each beat) by saying that the synchro must be tight:

```
::P() $sync = $s, @tight
```

such that, in-between events the progression follows the tempo. When a new event occurs, if this event becomes earlier w.r.t. to the inferred tempo, the time progression "makes a jump" to reach the new date. If the event arises later, the progression is frozen until the actual occurrence of the event. Smoother synchronization strategies are available [11].

6. REFERENCES

[1] G. Wang, P. R. Cook, and S. Salazar, "Chuck: A strongly timed computer music language," *Computer Music Journal*, 2016.

[2] H. Kopetz, "Event-triggered versus time-triggered real-time systems," *Operating Systems of the 90s and Beyond*, pp. 86–101, 1991.

[3] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of computer programming*, vol. 19, no. 2, pp. 87–152, 1992.

[4] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet, "Towards a higher-order synchronous data-flow language," in *Proceedings of the 4th ACM international conference on Embedded software*. ACM, 2004, pp. 230–239.

[5] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity-the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[6] N. Halbwachs, *Synchronous programming of reactive systems*. Springer Science & Business Media, 2013, vol. 215.

[7] C. Strachey and C. P. Wadsworth, "Continuations: A mathematical semantics for handling full jumps," *Higher-order and symbolic computation*, vol. 13, no. 1, pp. 135–152, 2000.

[8] J.-L. Giavitto and J. Echeveste, "Real-time matching of antescofo temporal patterns," in *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*. ACM, 2014, pp. 93–104.

[9] A. L. D. Moura and R. Ierusalimschy, "Revisiting coroutines," *ACM Trans. on Prog. Languages and Systems (TOPLAS)*, vol. 31, no. 2, p. 6, 2009.

[10] C. Trapani and J. Echeveste, "Real time tempo canons with antescofo," in *International Computer Music Conference*, 2014, p. 207.

[11] A. Cont, J. Echeveste, J.-L. Giavitto, and F. Jacquemard, "Correct Automatic Accompaniment Despite Machine Listening or Human Errors in Antescofo," in *International Computer Music Conference 2012*, 2012.

[12] E. W. Large and C. Palmer, "Perceiving temporal regularity in music," *Cognitive science*, vol. 26, no. 1, pp. 1–37, 2002.

[13] "Ableton Link Presentation," https://ableton.github.io/link/, accessed april 2017.