Embedding native audio-processing in a score following system with quasi sample accuracy

Pierre Donat-Bouillud, Jean-Louis Giavitto, Arshia Cont, Nicolas Schmidt, Yann Orlarey

Embedding native audio-processing in a score following system with quasi sample accuracy

Pierre Donat-Bouillud
IRCAM UMR CNRS STMS 9912,
INRIA Paris - MuTant Team-Project
ENS Rennes
pierre.donat-bouillud@ens-rennes.fr

Jean-Louis Giavitto
IRCAM UMR CNRS STMS 9912,
INRIA Paris - MuTant Team-Project
jean-louis.giavitto@ircam.fr

Arshia Cont
IRCAM UMR CNRS STMS 9912,
INRIA Paris - MuTant Team-Project
arshia.cont@ircam.fr

Nicolas Schmidt
Computer Science Dept.,
Pontificia Universidad Catolica de Chile
nschmid1@uc.cl

Yann Orlarey
Grame, Lyon, France
orlarey@grame.fr

ABSTRACT

This paper reports on the experimental native embedding of audio processing into the Antescofo *system, to leverage timing precision both at the program and system level, to accommodate time-driven (audio processing) and event-driven (control) computations, and to preserve system behaviour on multiple hardware platforms. Here native embedding means that audio computations can be specified using dedicated DSLs (*e.g.*, Faust) compiled on-the-fly and driven by the* Antescofo *scheduler. We showcase results through an example of an interactive piece by composer Pierre Boulez,* Anthèmes 2 *for violin and live electronics.*

1. IMS AND EVENT-DRIVEN *VS.* TIME-DRIVEN ARCHITECTURES

Interactive Music Systems (IMS) were promoted in early 1990s in an attempt to enable interaction between human musicians and real-time sound and music computing, initially for applications in *mixed music* defined by association during live music performance of human musicians and computers [15].

One specific challenge of IMS is to manage two "*time domains*": asynchronous event-driven computations and time-driven periodic management of audio processing. It led to the development of real-time graphical programming environments for multimedia such as *Max* [13] and the open source *PureData* [12].

In event-driven systems, processing activities are initiated as a consequence of the occurrence of a significant event. In time-driven systems, activities are initiated periodically at predetermined points in real-time and lasts. Subsuming the event-driven and the time-driven architectures is usually achieved by embedding the event-driven view in the time-driven approach: the handling of control events is delayed and taken into account periodically, leading to several internally maintained rates, *e.g.*, an audio rate for audio, a control rate for messages, a refresh rate for the user-interface, *etc.* This approach is efficient but the time accuracy is *a priory* bounded by the control rate.

An example of this approach is Faust [10], where control events are managed at buffer boundaries, *i.e.* at the audio rate. In Max or *PureData*, a distinct control rate is defined. This control rate is typically about 1ms, which can be finer that a typical audio rate (a buffer of 256 samples at sampling rate of 44100Hz gives an audio rate of 5.8 ms), but control computations can sometimes be interrupted to avoid delaying audio processing (*e.g.* in Max). On the other hand, control processing can be performed immediately if there is no pending audio processing.

The alternative is to subsume the two views by embedding the time-driven computations in an event-driven architecture. As a matter of fact, a periodic activity can be driven by the events of a periodic clock. [1] This approach has been investigated in ChucK [17] where the handling of audio is done at the audio sample level. Computing the next sample is an event interleaved with the other events. It results in a tightly interleaved control over audio computation, allowing the programmer to handle on the same foot signal processing and higher level musical and interactive control. It achieves a time accuracy of one sample. But this approach sacrifices the performance benefits of block-based processing (compiler optimizations, instruction pipelining, memory compaction, better cache reuse, *etc.*).

In this paper we propose a new architecture for the native embedding of audio computation in *Antescofo* [2]. The *Antescofo* system is a programmable score following system. *Antescofo* offers a tight coupling of a real-time machine listening [3] with a real-time synchronous Domain Specific Language (DSL) [7]. The language and its runtime system are responsible for timely delivery of *message passing* to host environments (MAX or *PureData*) as a result of reactions to real-time machine listening. The work presented here extends the *Antescofo* DSL with native audio processing capabilities.

[1] From this point of view, the only difference between waiting the expiration of a period and waiting the occurrence of a logical event is that, in the former case, a time of arrival can be anticipated.

In our approach, we do not attempt to provide yet-another universal language for audio processing but to provide the ability to compose complex architectures, using embedded local codes that employ specialized DSL, on-the-fly compilation and dynamic type-checking for passing between various time-domains. This approach reflects several important considerations: (1) to harness existing and well established audio processing systems, (2) to fill the gap between authorship and real-time performance; and (3) to improve both performance and time accuracy compared to existing IMS.

We start the paper by providing the necessary background on the *Antescofo* approach by focusing on a real-world example, an interactive piece by composer Pierre Boulez, *Anthèmes 2* for violin and live electronics, cf. Fig. 1. Section 3 discusses the main contribution of the paper by providing a time-aware semantics for combining real-time control and signal processing in *Antescofo*. Finally, we showcase results through the example of embedding audio processing in the Antescofo score of *Anthèmes 2*.

2. REAL-TIME COORDINATION OF MUSICAL EVENTS AND COMPUTING ACTIONS

2.1 A Paradigmatic Example

As an illustration, we showcase a mixed music piece that has entered the repertoire, *Anthèmes 2* (1997) by Pierre Boulez for violin and live electronics. The dominating platforms for programming such interactive paradigms are the graphical programming languages *Max* or its open-source counterpart *PureData*. In this section, we focus on a *PureData* implementation taken from *Antescofo*'s tutorial [5].

Programming of Interactive Music pieces starts by a specification of interactions, computing processes and their relations between each other and with the physical world in form of an *Augmented Music Score*. Fig. 1 (left) shows the beginning few bars of *"Anthèmes 2"*, Section 1. The top staff, upper line, is the violin section for human performer and in traditional western musical notation; and lower staves correspond to computer processes either for real-time processing of live violin sound (four *harmonizers* and *frequency shifter*), sound synthesis (two *samplers*), and spatial acoustics (artificial reverberation *IR*, and live spatialisation of violin or effect sounds around the audience). Computer actions in Fig. 1 are ordered and hung either upon a previous action with a delay or onto an event from a human performer. Computer Processes can also be chained (one sampler's output going into a reverb for example) and their activation is *dynamic* and depends on the human performer's interpretation.

Fig. 1 (right) shows the main patcher window implementation of the electronic processes of the augmented score in *PureData (Pd)*. The patch contains high-level processing modules, *Harmonizers*, *Samplers*, *Reverb*, *Frequency Shifting* and *Spatial Panning*, as sub-patchers. The temporal orderings of the audio processes is implicitly specified by a data-driven evaluation strategy of the data-flow graph. For example, the real-time scheduling mechanism in *PureData* system is mostly based on a combination of control and signal processing in a round-robin fashion [14], where, during a scheduling tick, time-stamped actions, then DSP tasks, MIDI events and GUI events are executed, cf. Fig. 2. Scheduling in *PureData* is thus block-synchronous, meaning that controls occur at boundaries of audio processing. Furthermore, in data-flow oriented language, the audio processes activation, their control and most importantly their interaction with respect to the physical world (human violinist) cannot be specified nor controlled at the program level.

2.2 Authorship in *Antescofo*

Real-time coordination and synchronization between human performer's events and computer actions is the job of *Antescofo* [2]. Code 1 shows the *Antescofo* excerpt corresponding to the augmented score in Fig. 1.

```
NOTE 8100 0.1                                  Q7
  Curve c1 @grain := 25ms
  { $hrout { {0} 1/4 {0.8} } }

  $h1 := -1
  $h2 := -4
  $h3 := -8
  $h4 := -10
NOTE 7300 1.0
```

Event

Curve Control

Actions triggered by the event

Code 1. Antescofo score for the first notes of *Anthème 2*

In the *Antescofo* code fragment above, notice the specification of both expected musical events from the physical environment (the NOTE keyword), computing actions (the sampling of a curve every 25ms for the next 1/4 beat) and their temporal relationships, *i.e.*, their *synchronization* (the sampling starts with the onset of the NOTE).

The *Antescofo* language provides common temporal semantics that allow designers and composers to arrange actions in multiple-time frameworks (absolute time, relative time to a tempo, event or time triggered), with multiple synchronization and error-handling strategies [4, 7]. Actions can be triggered simultaneously to an event detection by machine listening $e(t)$, or scheduled relatively to the detected musician's tempo or speed $\dot{e}(t)$. Actions can live in nested and parallel blocks or group, and the user can decide to program synchrony of a block on static or dynamic *target*s in the future (for instance, at the end of a phrase). Block contents can also be "continuous", as for the curve construct that performs a sequence of actions for each sample of a breakpoint function.

Real-time control in *Antescofo* follows a reactive model of computation assuming the *synchrony hypothesis* [1, 8]: atomic actions hooked directly to an event, called *reactions*, should occur in zero-time and in the right order. This hypothesis is unrealistic, however in practice, the system needs only to be quick enough to preserve the auditory perception of simultaneity, which is on the order of 20 milliseconds. This hypothesis is also adopted by ChucK [17] and makes the language a *strongly timed computer music language*:

- time is a first class entity included in the domain of discourse of the language, not a side-effect of the computations performance [9, 16];

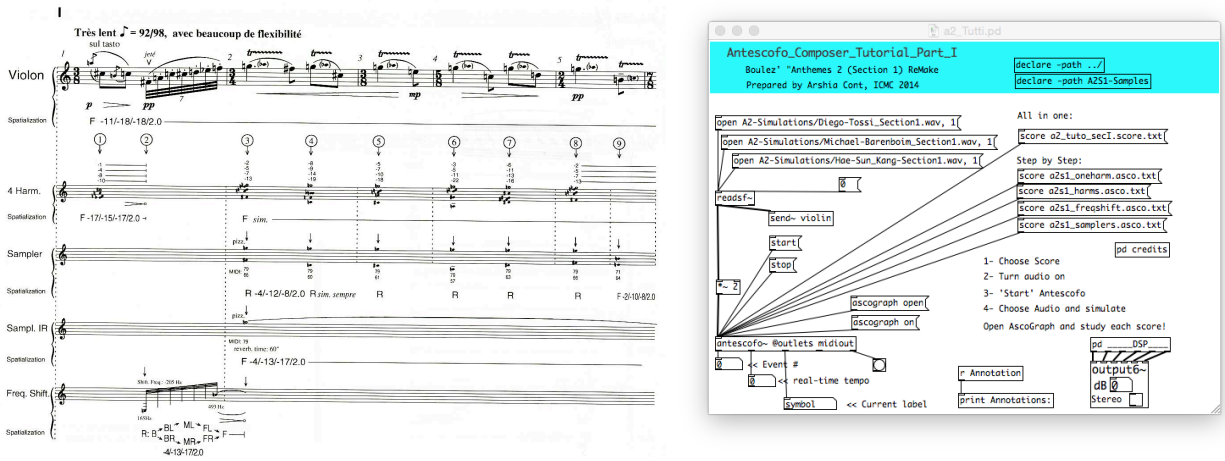

Figure 1. *Left:* Composer's score excerpt of *Anthèmes 2* (Section 1) for Violin and Live Electronics (1997). *Right:* Main *PureData* patcher for *Anthèmes 2* (Section 1) from *Antescofo Composer Tutorial*.

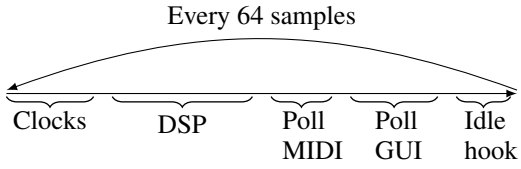

Figure 2. Scheduling cycle in *PureData* (polling scheduler)

- *when* a computation occurs is explicit and formalized in language semantics, ensuring behavior predictability and temporal determinism;
- assuming enough resources, the temporal behavior of a program is free from underlying hardware constraints and non-deterministic scheduling.

Antescofo synchronous programs departs in several ways from synchronous languages: *Antescofo* execution model manages explicitly the notion of duration, so the programmer may for instance trigger a computation after a well-defined delay. It is possible to refer to various time coordinates, including user-defined ones. And the dedicated language offers several constructs corresponding to "continuous actions" that span over an interval of time. Furthermore, *Antescofo* is dynamic and allows the dynamic creation of parallel processes.

The role of the *Antescofo* runtime system is to coordinate computing actions with physical events implementing the specified synchronizations, as shown in Fig. 3.

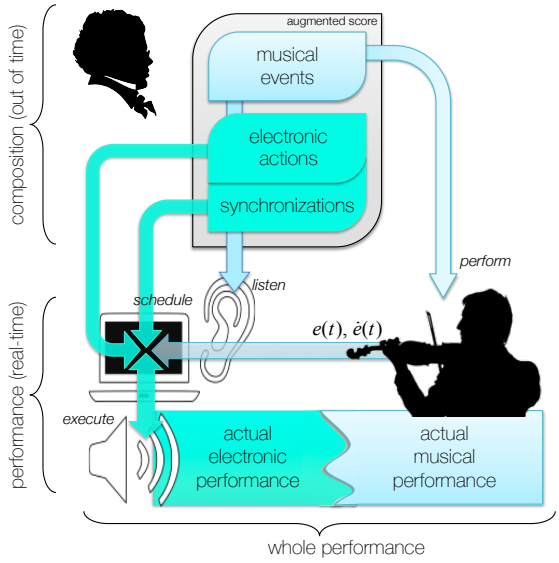

Figure 3. Antescofo Executation Diagram

3. TIME-AWARE COMBINATION OF SIGNAL PROCESSING AND REAL-TIME CONTROL IN *ANTESCOFO*

During run-time execution, the standard *Antescofo* implementation delegates the actual audio computations to the host environment. So, their time-safety and consistencies are subject to real-time scheduling of control, signal processing, and other user interrupts such as external controls of the GUI in the host. *PureData* and Max's capability of combining real-time control and signal processing within the same framework is the major feature of their architecture for end-users, but presents several shortcomings. Time is implicit in the data-flow style, so some temporal constraints between audio and control computation are simply not expressible. And the round-robin scheduling strategy forces a fixed interleaving of control and audio computation, that reduces the temporal accuracy that can be achieved.

Embedding digital audio processing in *Antescofo* is an experimental extension of the language, aimed at driving various signal processing capabilities directly within *Antescofo*, to overcome these drawbacks. The rest of this section presents this extension, sketches the underlying execution model and discusses the resulting temporal accuracy.

3.1 Audio Processing Nodes and Their Types

Signal processors are defined directly in an *Antescofo* program, harnessing several signal processing libraries. Currently, *Faust* [10] and a few dedicated DSP processors can be defined. These examples are enough to validate the approach and its versatility.

Faust processors are defined directly as Faust programs within the *Antescofo* score. They are compiled by the embedded *Faust* compiler when the *Antescofo* score is loaded and the resulting C code is compiled on-the-fly (with the in-core LLVM compiler) into a dynamically linked function that implements the specified computation. A few dedicated DSP processors have been specifically developed, notably a FFT transformation based on the Takuya Ooura's FFT package. The objective is to validate the integration of spectral computations in the *Antescofo* audio chains, an example of time-heterogeneous audio computations.

For efficiency reasons, audio samples are grouped into buffers and all samples of a buffer are handled altogether by a DSP node which therefore performs its computation periodically: a buffer corresponds to a set of values that are supposed to be produced and consumed sequentially in time, but that are all accessible at the same moment because the actual use of the buffer is deferred to a later moment. So, irrespectively of the exact computation it achieves, a DSP processor can be abstracted by a function that processes a *sequence of buffers*. These sequences are characterized by a *buffer type* corresponding to the periodicity and the size of a buffer in the sequence. It represents also the information needed to associates a time-stamp to each element in the sequence once a time-stamp is given to the buffer. Such types make it possible to represent overlapping buffers, which are common when doing spectral processing.

Antescofo distinguishes between two kinds of DSP nodes, as illustrated on Fig. 4. *Isochronous* node, or *effects*, transform a buffer into a buffer of same type, so only the elements of the buffers are modified, but not the size nor its periodicity. They can have ordinary *Antescofo* variables as additional input or output control. Typically, Faust processors are isochronous.

Heterochronous nodes consumes and produces sequences of buffers of different types. A *detector* which takes an audio signal as input and outputs a boolean as the value of a control variable when some condition is satisfied, is an example of heterochronous node. A Fourier transformation is another example of a heterochronous computation.

3.2 Connecting Audio Processing Nodes

DSP nodes are connected by *links*. Links are implicit *buffer type adapters*: they are in charge of converting a sequence of buffers into another equivalent sequence of buffers, leaving the samples untouched. Equivalence means here that we have the same sequence of buffer elements, irrespectively of the buffer boundaries, or that the output sequence is an "(un)stuttering" of the input sequence in case of overlapping buffers. Links represent also input or output channels, that transport the audio signal from and to the soundcard or the host environment. Once the buffer types of DSP nodes are known, the adequate link adaptation can be automatically generated to convert between buffer types.

Links appears as a special kind of variable in *Antescofo*. They are denoted by $$-identifiers where ordinary variables are denoted by $-identifiers. As for ordinary variable, the occurrence of a link in a control expression denotes the "current value" of the corresponding buffer sequence, *i.e.* the sample corresponding to the current instant.

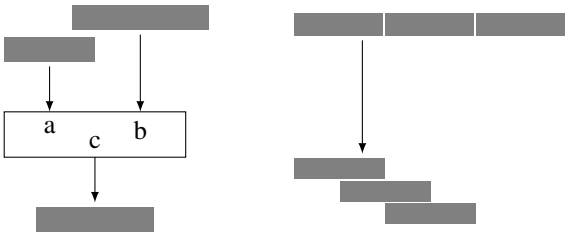

Figure 4. *(Left):* An isochronous node processes buffers, not sequences of buffers. This one has two input ports a and b and an output port c. *(Right):* A link act as an implicit type converter and transforms sequences of buffers into an equivalent sequence of buffers. Here, the input buffers are contiguous and the output buffers overlap.

3.3 Dynamic Patches

DSP nodes and links are declared independently in the score, and can be connected later using a new dedicated *Antescofo* action, *patch*, which represent a set of equations. One equation corresponds with one DSP node. Occurrences of links in the equations materialize the producer/-consumer relationships between DSP nodes. Fig. 5 shows the effects and the links for the DSP graph of the beginning of *Anthèmes 2*.

Patch actions can be launched dynamically, enabling reconfiguration of the audio computation graph in response to the events detected by the listening machine. These dynamic changes can be also synchronized with the musical environment using the expressive repertoire of synchronization strategies available in *Antescofo*.

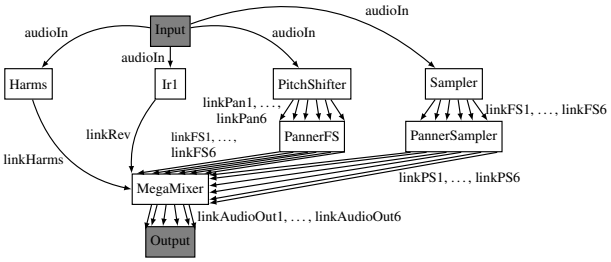

Figure 5. DSP graph at the beginning of *Anthèmes 2* by Pierre Boulez. The audio signal flows from *Input* to *Output*.

If a DSP node or a link channel is not used in an active *patch*, the link and the related DSP nodes are disabled as shown on Fig. 6: removing a link (resp. a node) from the audio graph also removes the subtree rooted by the link (resp. the node). All links and nodes that are not connected to an output channel are also disabled.

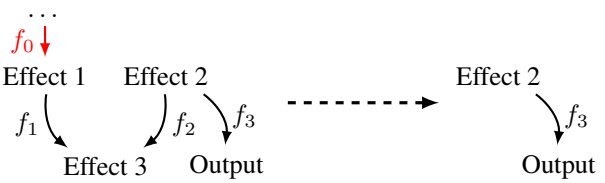

Figure 6. Removing the link f_0 in the DSP graph. As Effect 3 and Effect 3 need buffers traversing f_0, Effect 1, Effect 3 and link f_1 are removed from the graph. The incoming effects to Effect 1 don't have any other outcoming path to the Output, so they are also removed from the Dsp graph.

3.4 Architecture Rationals

Several benefits are expected with this tight integration of audio computations in the language: *i)* An augmented score will be specified by *one textual file* which records the definitions and the control of all the software components involved in the implementation of the piece. *ii)* The network of signal processors is heterogeneous, mixing DSP nodes specified with different tools (*Faust*, *FluidSynth*, *etc.*). *iii)* The network of signal processors can change dynamically in time following the result of a computation. This approach answers the shortcomings of fixed (static) data-flow models of the Max or Pd host environments. *iv)* Signal processing and its scheduling are controlled at a symbolic level and can be guided, *e.g.* by information available in the augmented score (like position, expected tempo, *etc.*). *v)* This tight integration allows concise and effective specification of finer control for signal processing, at a lower computational cost. One example (developed below) is the use of symbolic curve specifications to specify variations of control parameters at sample rate. *vi)* Signal processing can be done more efficiently. For example, in the case of a *Faust* processor, the corresponding computation is highly optimized by the *Faust* on-the-fly compiler.

3.5 A GALS Execution Model

The temporal interactions between audio and control computations within *Antescofo* can be roughly conceived as two autonomous worlds that evolve in parallel and that interact by shared information. Audio computation can be seen as computation on continuous data and control computation as sporadic processes. In a rough approximation, audio processing is done *continuously and in parallel* to the control computations.

To fully understand the interplay between audio and control computation, one has to refine the "continuous-processing-of-audio-signal" notion into the more realistic "sample-processing-of-sampled-audio-signal" implementation. At the end of the day, each sample corresponds to a physical date and moving forward in the buffer corresponds to some time progression.

A control signal (an external event, the exhaustion of a delay, *etc.*) may occur during an audio computation: control computations and audio computation are "asynchronous". But all audio computations correspond to well known time-stamps and control computations are also well-ordered by causality. Thus, locally, the computation appears "synchronous". The term *GALS* for "globally asynchronous, locally synchronous" has been used to describe this situation [6]. The challenge thus lies at the interface of the two models of computation : control computation which is supposed to happens instantly, may be delayed until the end of an audio computation, which decreases the temporal accuracy of the system.

3.6 Temporal Accuracy in a GALS Execution Model

If audio processing and control processing appear as parallel computations, they interact by sharing information or spanning computations at some points in time. In an idealistic world with unbounded computational power, buffer sizes will be reduced to only one sample and control and audio processing will be fully interleaved. This execution model achieves *sample-accuracy*, the greatest possible temporal accuracy between the two asynchronous audio and control worlds. Due to the limited computational power available, buffer sizes cannot be shrunken to one sample. To take advantage of the buffer processing approach, the *Antescofo* execution model takes into account the control variables during DSP processing only at some limited dates, as follows (cf. Fig. 7).

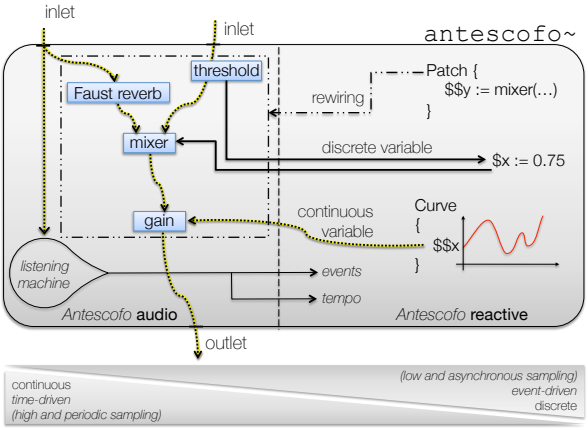

Figure 7. Interaction between audio processing and reactive computations.

i) The dependencies between audio computations are ordered using a topological sort to sequence the buffer computations. If the audio computations are independent of any control parameter, this achieves sample-accuracy while preserving block computation. ii) Musical events are detected by the listening machine in an audio stream and signaled to the reactive engine which eventually triggers control computation at the end of the processing of the input audio buffer. The theoretical constraints of the spectral analysis done to detect the musical events imply that at most one event is recognized within roughly 11 ms (512 samples at 44 100Hz sampling rate). iii) Reactive computations that are not spanned by the recognition of a musical event, are triggered by the elapsing of a delay or by an external event signaled by the environment (*e.g.*, a keyboard event). In these case, the temporal accuracy of the reaction is those provided by the host environment (typically in Max, 1 ms for an external event, usually better for the exhaustion of a delay). We say that these computations are *system-accurate*. iv) Computations in the reactive engine may start, stop or reorganize the audio computations. These actions take place always at the end of a reaction and at "buffer boundaries", that is, between two buffer processing in the DSP network. We say that these computations are *buffer-accurate*. v) The audio computation is controlled by discrete data or symbolic continuous data computed in the reactive engine. Discrete data are read by audio computation at buffer boundaries and are assumed constant during the next buffer computations while symbolic continuous data are handled as if they are read for the processing of each sample because their time-evolution is know *a priori*.

This approach is temporally more accurate than the usual approach where control processing is done after the processing of all audio buffers, for two reasons.

Control computation is interleaved with DSP node processing and is not delayed until the end of all DSP processing. Compared to sample-accuracy, it means that taking into account the change of a control variable is delayed for each audio node only by the time interval corresponding to its own rate. Because the DSP network includes heterogeneous rates, the benefit can be sensible. Furthermore, if the code corresponding to the DSP nodes permits (this is usually the case for *Faust* specified nodes), these rates can be dynamically adjusted to achieve a greater temporal accuracy (at the expense of more overhead) or to lower the computational cost (at the expense of temporal accuracy).

The second benefit of our approach is that control variables managed within the reactive engine can be taken into account during audio-processing at the level of sample-accuracy, when they are tagged "continuous" (this is the case when their identifier starts with $$). Continuous variable can be used as ordinary *Antescofo* control variables. However, when their updates can be anticipated, because for instance they are used to sample a symbolic *curve* construct, this knowledge is used to achieve sample accuracy in the corresponding audio processing. Fig. 8 illustrate the difference; the top plots draw the values of the variable $y in relative and absolute time in the program:

```
Curve @grain 0.2s { $y { {0} 6 {6} } }
```

This curve construct specifies a linear ramp in time relative to the musician tempo. For the implementation, the control variable $y samples the curve every $0.2\,$s (notice that the sampling rate is here specified in absolute time) going from 0 to 6 in 6 beats. There is 3 changes in the tempo during the scan of the curve, which can be seen as slight changes in curve derivative in the right plots (these change does not appear in relative time). The bottom plots figure the value of the continuous variable $$y (the same changes in the tempo are applied) defined by:

```
Curve @grain 0.2s { $$y { {0} 6 {6} } }
```

Despite the specification of the curve sampling rate (used within the reactive engine), the continuous control variable samples the line every $1/44100 = 0.022\,$ms during audio processing.

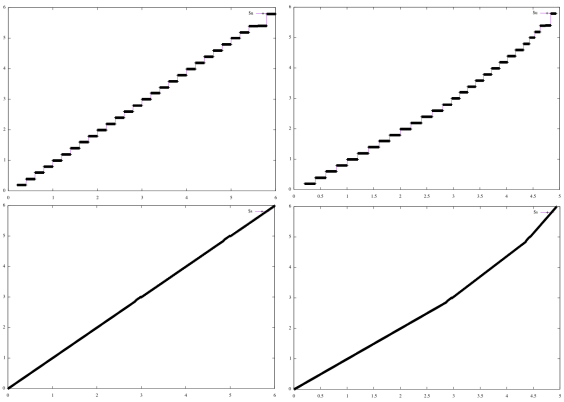

Figure 8. *Top:* plot of the values of the variable $y in relative and absolute time. There is 3 changes in the tempo during a linear ramp. *Bottom:* plot of the value of the continuous variable $$y. The same changes in the tempo are applied.

4. ANTHÈME 2 EXAMPLE

We validated the previous extension in a a strict re-implementation of *Anthème 2* introduced in section 2. For this reimplementation, we aim at embedding all audio computation inside an *Antescofo* program and next to control parameters following extensions. Code listing 2 shows a message-passing implementation where level control and DSP parameter (frequency shift value here) are passed to an outside module corresponding to implementation in section 2; and Code listing 3 shows the new implementation with strictly the same behavior. In Code 3, the level parameter is defined by a Curve control construct, and the DSP node employs faust :: PitchShifter . The definition of PitchShifter is a native *Faust* code included in the same scope, and is compiled on-the-fly upon score load in *Antescofo*. Computation of PitchShifter is handled dynamically during runtime executation of the system (i.e. live music performance) and parameters combine discrete values ($freq), interpolated variables ($psout) as well as audio buffer input ($$audioIn) and an audio buffer output link $$linkFS which is sent later on for spatial panning. Implementation of other modules follow the same procedure, by embedding native *Faust* code (for time-domain signal processing).

```
TRILL ( 8100 8200 ) 7/3    Q25
  ; bring level up to 0db in
       25ms
  fs-out-db 0.0 25
  ; frequency shift value
  fd1_fre -205.0
```

Code 2. Message Passing (old style)

```
TRILL ( 8100 8200 ) 7/3    Q25
  Curve c3    @grain := 1ms
  {; bring level up to 0db in
       25ms
    $psout
    {
      {0}
      25ms {1}
    }
  }

  $freq := -205    ; freq.
       shift value
  $$linkFS := faust::
       PitchShifter(
       $$audioIn, $freq,
       $psout)
```

Code 3. Embedded Audio

Time-profiling analysis of the message-passing example (Figure 1) and its embedded counter-part shows an improvement of 12% overall system utility improvement with the new implementation, corresponding to 46% utility performance on the task itself. This analysis was done on a MacBook and using *XCode*'s *Time Profiler* tool on a sampled period of real-time performance simulations where the code interacts with a human musician.

This improvement is due to several factors: optimisation of local DSP code provided by native hosts (such as *Faust*) and the lazy type conversion approach adopted in section 3 when converting (for example) between Curve and continuous-audio variables.

The approach developped here provides some gain in performance but also preserve both the control structure of the program for designers and its final behavior. This is made possible by explicit consideration of timing control over computations at stake and embedding them into the coordination system of *Antescofo*. The performance improvement has also allowed us to prototype such interactive music pieces on mini-computers such as Raspberry PI and UDOO.

5. CONCLUSION

We extended the *Antescofo* language, already employed by the community in various creations involving musicians and computers worldwide, with the possibility of time-aware combination of signal processing and control during authorship and real-time performance. This is achieved through a *GALS* execution model as shown in section 3, embedding of existing local modules in their native language in the system and compiled on the fly, and assuring their timely inter-communication and constraints inherited from their types or computation at stake. We showcased this study by extending an existing implementation of a music piece in the general repertoire (namely, Pierre Boulez' "Anthèmes 2") using the proposed approach. We showed its potential both for behavior preservation, time precision, ease of programming without significant breakdown for designers, and potential for multi-platform deployment.

This work will be extended in several ways, providing more native embedding services to users based on existing practices in interactive multimedia. The type system can be enriched for adequate description of finer temporal relationships. More studies and benchmark should be undertaken when combining and deploying multiple-rate processing modules and their synchrony in the system. Static Analysis tools should provide feedback to designers when certain timing constraints between computational modules can not be held, as an extension to [11], and could enable further optimizations in the DSP graph, and in the interaction between signal processing and control. The listening module of Antescofo could be reimplemented as an audio effect, opening the way to various specialized listening modules.

Acknowledgments

This work was partially funded by the French National Research Agency (ANR) INEDIT Project (ANR-12-CORD-0009) and the INRIA internship program with Chile.

6. REFERENCES

[1] G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, 1992.

[2] A. Cont, "Antescofo: Anticipatory Synchronization and Control of Interactive Parameters in Computer Music," in *Proceedings of International Computer Music Conference (ICMC)*, Belfast, Irlande du Nord, August 2008.

[3] ——, "A Coupled Duration-Focused Architecture for Real-Time Music-to-Score Alignment," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 6, pp. 974–987, 2010.

[4] A. Cont, J. Echeveste, J.-L. Giavitto, and F. Jacquemard, "Correct Automatic Accompaniment Despite Machine Listening or Human Errors in Antescofo," in *Proceedings of International Computer Music Conference (ICMC)*. Ljubljana, Slovenia: IRZU - the Institute for Sonic Arts Research, Sep. 2012. [Online]. Available: http://hal.inria.fr/hal-00718854

[5] A. Cont and J.-L. Giavitto, "Antescofo workshop at ICMC: Composing and performing with antescofo," in *Joint ICMC - SMC Conference*, Athens, Greece, Sep. 2014, The remake of *Anthèmes 2* is part of the tutorial and it can be downloaded at http://forumnet.ircam.fr/products/antescofo/.

[6] F. Doucet, M. Menarini, I. H. Krüger, R. Gupta, and J.-P. Talpin, "A verification approach for gals integration of synchronous components," *Electronic Notes in Theoretical Computer Science*, vol. 146, no. 2, pp. 105–131, 2006.

[7] J. Echeveste, J.-L. Giavitto, and A. Cont, "Programming with Events and Durations in Multiple Times: The Antescofo DSL," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 2015, (submitted).

[8] N. Halbwachs, *Synchronous Programming of Reactive Systems*, ser. Lecture Notes in Computer Science, A. J. Hu and M. Y. Vardi, Eds. Springer, 1998, vol. 1427.

[9] E. A. Lee, "Computing Needs Time," *Communications of the ACM*, vol. 52, no. 5, pp. 70–79, 2009.

[10] Y. Orlarey, D. Fober, and S. Letz, *FAUST : an Efficient Functional Approach to DSP Programming*, 2009, pp. 65–96. [Online]. Available: http://www.grame.fr/ressources/publications/faust-chapter.pdf

[11] C. Poncelet and F. Jacquemard, "Model based testing of an interactive music system," in *ACM SAC*, 2015.

[12] M. Puckette, "Pure data," in *Proc. Int. Computer Music Conf.*, Thessaloniki, Greece, September 1997, pp. 224–227. [Online]. Available: http://www.crca.ucsd.edu/~msp

[13] ——, "Combining Event and Signal Processing in the MAX Graphical Programming Environment," in *Proceedings of International Computer Music Conference (ICMC)*, vol. 15, Montréal, Canada, 1991, pp. 68–77.

[14] R. V. Rasmussen and M. A. Trick, "Round robin scheduling – a survey," *European Journal of Operational Research*, vol. 188, no. 3, pp. 617 – 636, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377221707005309

[15] R. Rowe, *Interactive music systems: machine listening and composing*. Cambridge, MA, USA: MIT Press, 1992.

[16] A. Sorensen and H. Gardner, "Programming With Time: Cyber-Physical Programming With Impromptu," in *ACM Sigplan Notices*, vol. 45, no. 10. ACM, 2010, pp. 822–834.

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