AntesCollider: control and signal processing in the same score

ABSTRACT

We present AntesCollider, an environment harnessing Antescofo, a score following system extended with a real-time synchronous programming language, and SuperCollider, a client-server programming environment for real-time audio synthesis. The environment enables the definition of a centralized executable score *specifying complex timelines driving sophisticated controls, audio synthesis and their synchronization with a performer. The audio processing information is distributed to the SuperCollider servers using the OSC protocol under the supervision of the Antescofo scheduler taking care of the synchronizations with external events. Audio processing can be distributed over several SuperCollider server to maximize CPU efficiency. We showcase the system on a new piece,* Curvatura II *for live electroacoustics.*

1. INTRODUCTION

AntesCollider is a library programmed in the Antescofo language to provide higher level and expressive control on the SuperCollider *scsynth* server. The library is organized around a set of concurrent objects to easily create dynamically audio processes and to implement them using scsynth servers. The motivation is to use the Antescofo language expressiveness to write complex electronic musical processes and synchronize them relying on the score follower capabilities of the Antescofo *meta sequencer* and taking advantages of the optimized and versatile audio synthesis dynamic capabilities of scsynth.

The library has already been used for the production of several electroacoustic, mixed and multimedia works, including sensors, lights and video in real time. Compared to the usual approach where the Antescofo score controls audio processes implemented in Max (or PureData) through messages, the resulting systems are more robust and more CPU efficient.

The paper is organized as follows. Next section gives some background information on SuperCollider and Antescofo and its object system. Section 3 is devoted to the dynamic organization of audio chains, the communication between Antescofo and scsynth, the expressive control of the synthesis, some strategies to achieve load balancing, and the real time monitoring of the system through a dedicated gui. Section 4 presents the use of the system in the development of *Curvatura II*, a real time electroacoustic piece using an HOA spatialization system developed by one of the authors.

2. BACKGROUND

2.1 SuperCollider

SuperCollider [1, 2] is a real-time audio synthesis and algorithmic composition environment and programming language. It is divided into two components, as shown on Fig. 1: a server, *scynth*, and a client, *sclang*, which communicate using the OSC protocol [3]. The client translates the high level object oriented and functional language of SuperCollider, cf. Code 1, into OSC messages sent to the server. SuperCollider processes audio through an ordered tree of unit generators for analysis, synthesis and processing. It can use any number of input and output channels. Unit generators are typically grouped statically in a higher level processing unit, called a *synth*. Another server for SuperCollider, Supernova [4], exploits multicore and multiprocessor architectures by providing a new instruction, `ParGroup`, that parallelizes the unit generators. SuperCollider provides ways to plan and to schedule events, but only refering to physical time or to a fixed musical time that does not follow the dynamic tempo of the interpretation. The specification of complex temporal relationships, especially in an interactive setting where the dynamic timing of external events must be taken into account (synchronization), remains difficult. These shortcomings motivate the coupling with the Antescofo system.

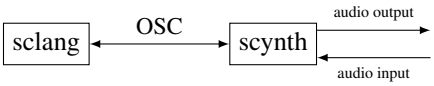

Figure 1. SuperCollider is split into two components: the client, *sclang*, using a object oriented and functional language, and the server, *scynth*, which processes the audio. Both communicate using the OSC protocol.

```
{ SinOsc.ar(440, 0, 0.1) + WhiteNoise.
    ar(0.01) }.play;
```

Code 1. A simple program *sclang* that generates a sine at 440 Hz with phase 0 and an amplitude of 0.1, and white noise at the audio rate (*i.e.* `ar`), adds them, and plays them. Sclang implements the evaluation of this expression by sending OSC messages to the scsynth server.

2.2 Antescofo

Antescofo [5] is a score following system combining a *listening machine* with a *synchronous reactive programming language*. Antescofo is embedded into Max/MSP [6] or Puredata [7] as an external. An Antescofo program is an *augmented score* using the dedicated programming language to define the musical events to follow, the electronic reactions to the events, and the synchronization between

computed actions and human performers. In the following excerpt of Antescofo code, the first line defines a musical event, a note with pitch C4 and duration of half a beat. The following lines are actions that are launched when the event is recognized: `rcvr1...` is a message sent to receiver `rcvr1` in Max or Puredata, followed after a delay of half a beat (relative to the tempo of the performer) by a message to the receiver `print`.

```
NOTE C4 1/2
        rcvr1 harm1 60 87 0.5
        1/2 print HELLO
```

The system decodes an input audio stream to retrieve the musical events specified in the augmented score but may also process sensors input data (e.g., accelerometers, kinect, midi input). Beside audio processing, it has been used to control video displays, light, and mechatronics, using Max messages or OSC messages [3]. Antescofo is used at Ircam and elsewhere for the realization of various mixed music pieces. The underlying technology is also used in Metronaut, [1] an automatic accompaniment system for the general audience.

Antescofo handles complex timelines with musical time (in beats) and physical time (in seconds). Control of audio synthesis can be expressed using curves (piecewise defined functions) that adapt to the tempo. The way the system synchronizes with the human performer is explicitly specified by synchronization strategies [8]. For instance, in the *loose* synchronization strategy, the scheduling of the actions follows the real-time changes of the tempo of the musician, whereas in the *tight* synchronization strategy, actions are triggered taking into account the occurrence of the nearest event in the past. The choice of a relevant synchronization depends on the musical context.

Actors in Antescofo. The notion of *object* is now widespread in programming. This concept is used to organize code by gathering values together into a state and making the possible interactions with this state explicit through the notion of methods. A related and less popular notion is the concept of *actor*. The actor model of programming was developed in the beginning of the '70s with the work of Carl Hewitt and languages like Act [9]. Later, Actor programming languages include the Ptolemy programming language [10] and languages offering "parallel objects" like Scala or Erlang.

While objects focus on code reuse with mechanisms like inheritance, method subtyping, state hidding, etc., actors focus on the management of concurrent activities of autonomous entities. Antescofo provides actors as entities that encapsulate a state and provide concurrent, parallel and timed interactions with this state. [2] An actor definition can be instantiated into an actor instance. Methods can be called on this instance and correspond to instantaneous computations [11]. Processes can also be started from these objects: they correspond to timelines and perform computations that last over time. A method can also be simultaneously executed on all instances of a given actor (e.g., for synchronization purposes), to trigger arbitrary reactions when some logical expression becomes true, or to kill an instance (which may trigger some instance handler). All actor's computations are subject to synchronization with the musician or on a variable, they can be performed on a given tempo, etc. The concurrency between method, process, synchronization, reaction and handler invocations, is managed implicitly and efficiently by the Antescofo run-time system [12].

Audio in Antescofo. Antescofo is usually used in conjunction with Max/MSP or Puredata for the audio synthesis part and is directly embedded into it. Compared to Max/MSP and Puredata, Supercollider can easily modify the audio graph during execution which is a requirement of our target applications. Composers have also used Csound [13] with Antescofo and have created a shallow layer in the Antescofo language to control it [14]. Csound is a well known audio processing language but Supercollider provides a more modern implementation, especially with SuperNova, which is able to exploit hardware parallelism. In [15, 16], the Faust language [17] as well as custom C++ audio effects are natively embedded into Antescofo and audio processing graphs can be created and modified on the spot. Further developments are required to make this extension more robust and usable at a large scale.

3. BRIDGING ANTESCOFO AND SUPERCOLLIDER: ANTESCOLLIDER

AntesCollider uses Antescofo to describe the synchronization strategies and the timeline for a musical piece, and SuperCollider, for the audio synthesis and processing, as shown on Fig. 2. The audio processing routing is defined in the Antescofo score and sent directly to *scynth* using OSC.

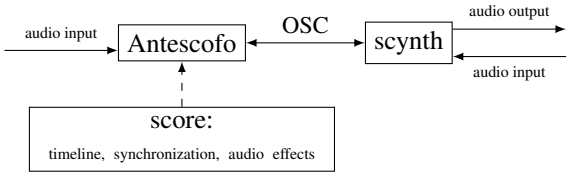

Figure 2. AntesCollider does not use the *sclang* client for audio routing and interacts directly with the scsynth audio server. The Antescofo score describes the timeline, the synchronization strategies and the audio effects, and *scynth* processes the audio.

3.1 Defining Supercollider audio processing in an Antescofo score

AntesCollider uses the paradigm of *tracks* (as in a DAW) for the organization of the different chains and modules of synthesis and treatments. Audio chains are organized in four layers, each layers corresponding to specific scsynth's groups, as shown on Fig. 3:

- *default group*, as in the sclang node representation, the group that embeds all groups and audio processes,
- *mix_group*, the group that encapsulates tracks. Tracks in a module are eventually mixed .
- *track*, the group that contains all the modules. Modules are stacked in a track.

[1] `https://www.antescofo.com`

[2] `http://support.ircam.fr/docs/Antescofo/manuals/Reference/actors`

- *module*, real time audio processing for treatments and synthesis. It corresponds to a *synth*.

This organization allows us a great flexibility on the general controls such as the amplitude of several tracks, for example, the control of a mix_group or the creation/destruction of a *module, track, mix_group* or *server*, both individually or globally.

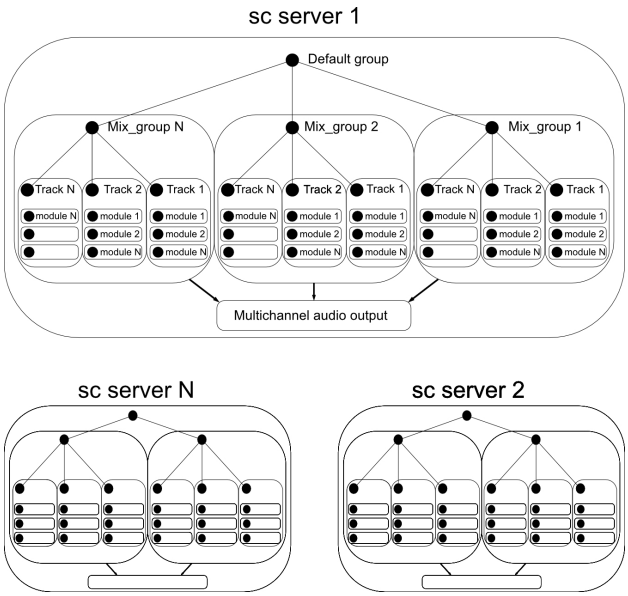

Figure 3. Multiserver configuration, with N scynth servers. Audio processing is organized in a tree fashion inside each server.

To instantiate a *server*, a *mix_group* or a *track*, we use the actor system of Antescofo. The sc_server object represents a scynth server in the Antescofo score. It can be initialized with several optional arguments such as UDP port number, number of input and output ports, sampling rate, etc. For instance, obj::sc_server("server1", 57110) instantiates a scynth server with name *serve1* on port 57110. Methods of the object can be used to control and monitor the server, as shown in Code 2.

```
// add a 24 channels output master to the end of
// all chains to control the general volume
$Servers("server1").master_out(24)
// set the general output master vol. to -20 dB
$Servers("server1").volume(-20)
// return scsynth's current version
$Servers("server1").version()
// return scsynth's current status
$Servers("server1").status(1)
// quit the scsynth server "server1"
$Servers("server1").quit()
```

Code 2. $Servers is an Antescofo dictionary that binds server names to server objects. Here, we control and monitor server1.

Mix groups are instances of obj::mix_group. At creation, parameters provide group name, server name and the number of channels or the HOA order and decoder required for higher order ambisonics [18] outputs. *Tracks* are created inside a mix_group. Tracks automatically adapt to the mix_group type (number of channels or HOA order and decoder). AntesCollider track objects have different methods to control parameters, to change the behavior, to activate different process or to kill the object and his representation in the server side. For instance, the following code snippet:

```
$audioChain := [
    ["BassSynth1", "freq", 35.9, "fmrange",
        1.5, "fmfreq", 7.85, "lpf", 214., "hpf"
        , 51.2, "rq", 0.35, "amp", -10],
    ["TPan8", "pos", 0., "width", 2, "lag", 0,
        "amp", 0]  ]
obj::crea_track("sc_track2_1", 0, 1, 0,
    $audioChain, "group1_s2")
$tracks("sc_track2_1").mod_add(["TFlanger", "
    flangefreq", 1.822], "after", "BassSynth1")
$tracks("sc_track1").rand_lfo("TRingMod1", "
    modfreq", 150, 300, 164.4, "linear", 200)
```

creates a track sc_track2_1 in group group1_s2. The initial audio chain array controled by the track is a module called BassSynth1 followed by a panner TPan8. The variable $tracks refers to the dictionary of all tracks in the system, and is used to retrieve track sc_track2_1, to add a Flanger effect TFlanger after the BassSynth1 module. Then, the method rand_lfo creates an Antescofo process that does continuous random control to modulate the parameter modfreq of TRingMod1 module between 150 and 300 Hz with specific interpolation curve and tempo.

3.2 Communications between Supercollider and Antescofo

The scsynth server is driven by the AntesCollider library relying on the builtin OSC communication capabilities embedded in Antescofo. The management of OSC messages is achieved in Antescofo through 3 primitives: oscsend defines an OSC output channel used to send messages to a set of OSC receivers specified by an IP address, a port number and, optionally, a predefined message header. oscrecv defines an OSC input channel used to handle the OSC messages incoming to a set of specified ports. Optionally a message header can be specified to restrict the message handled to those with this header. These two primitives establish only an unidirectional channel with an external processs. Bidirectional communications relying on these two primitives require two channels (one for sending and one fo receiving), which is not the communication pattern used by scsynth which rely on the UDP metadata of a message to answer to the message sender [19, chap. 8]. This communication pattern is handled by an osc_client declaration. The command uses the following syntax: osc_client id host : port * handler attributes.

Once initiated, the name of the command can be used to send OSC messages to a receiver, called the *server*, while the callback is activated to handle incoming messages from the server. The Antescofo program acts as a *client* in a server-client relationships: the server answers to the requests of the client but do not engage in an interaction which is not initiated by the client. An osc_client does not specify a predefined message header. So when a message is sent, the first argument of the message is used as the header, and the rest of the parameters are the arguments of the OSC message.

For instance, communicating with the scsynth typically involves an initialization phase:

```
// launching the SuperCollider server
// listening the requests on port 57110
@system("scsynth -u 57110")

// Open an osc channel in client mode. Process
// ::Recv is called to process incoming msgs
```

```
osc_client scServer localhost:57110 * ::Recv

// initiate the connection with SuperCollider
// by sendind the "/notify" command
scServer "/notify" 1
// the server will answer /done /notify
```

Here, the command osc_client introduces a new command scServer that can be used to send messages to scsynth.

3.3 Handling the incoming server's messages

Two distinct mechanisms can be used to handle the incoming messages. In the previous example, a process is used as a callback launched each time an incoming messages is received, in parallel with the main computations.

Antescofo follows the synchronous paradigm: computation are carried in a strict sequential mode, but elementary computations are assumed to takes zero time to execute. So some computations are done "in parallel", *i.e.* take place in the same instant, but are scheduled sequentially [11]. This approach articulates in an odd way the relationships of *simultaneity* and *succession* but is logically well founded [20] and presents the benefits of not requiring locks, semaphores, or other synchronization mechanisms (computations that occur simultaneously execute sequentially and without preemption, behaving naturally as *atomic transactions* with respect to variable updates).

The header and the arguments of an incoming message are dispatched on the parameter's of the callback. If there is more arguments than parameters, the last parameter receive a vector gathering the remaining arguments. If there are fewer arguments than parameters, the additional parameters are given the *undef* value.

An alternative mechanism can be used to handle the incoming messages. Instead of a callback, the osc_client command specifies a list of variables. These variables are updated on the reception of the message arguments, following the previous dispatch strategy. They can be used elsewhere in the program, asynchronously, and their value reflect the information brought by the last received message.

3.4 Controlling Audio synthesis from Antescofo

Using the previous communication mechanisms, the control of the scsynth audio processing is interleaved in the Antescofo augmented score, through OSC messages instead of using MAX messages. Compared to a MAX-based implementation, the only additional specification is the sequence of messages required to dynamically create the audio graph (in MAX this audio graph is not computed at run time but statically patched by the programmer at programming time). This is not a drawback but a benefit of the approach: only the audio transformation that are really needed are allocated on once an audio treatment is no longer required, the corresponding resources can be released by scsynth, thus optimizing audio buffers, CPU processing, etc.

There are several benefits in controlling the audio processing from Antescofo, even if the score following features are not used and Antescofo has been instrumental in the development of several purely electronic pieces, as well as mixed music pieces. As a matter of fact, the Antescofo programming language offers:

- expressive constructions (breakpoint functions that are data, higher order functions and processes, iteration, temporal recursion, reaction to arbitrary conditions, temporal patterns [21], ...) help to control arbitrary processes through time;
- an actor system (autonomous objects that can be instantiated and run in parallel) which helps to define and parameterize autonomous musical voices;
- sophisticated synchronization primitives on events but also on tempi [12] that make possible to gather these voices in a relevant polyphony.

The ability to start an Antescofo program at an arbitrary point proved also especially useful during composition phase and during rehearsals.

3.5 Load-balancing ugen allocation

The pieces, especially when using many channels and many ugens, can be computationally expensive and overload the capacity of one *scynth* server, even if CPUs are not saturated. If a server becomes overloaded, the sound is degraded and we can hear clicks. The parallel-execution server, Supernova, can be used but requires to explicitly mark the parts of the audio graph to parallelize using the ParGroup function. We tried instead to exploit multicore or multiprocessor architectures by launching multiple scsynth servers and to transparently balance the load by distributing the groups and synthesizers on the servers, as shown in Fig. 4. The load-balancing strategy is however constrained by the data dependencies induced by the audio graph.

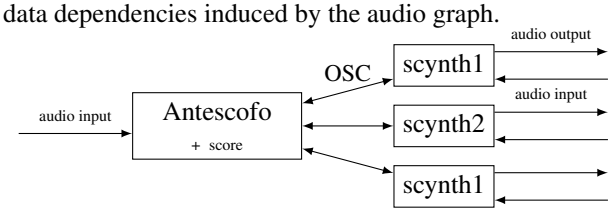

Figure 4. AntesCollider launches several *scynth* servers, for instance, one per core on the processor. It watches the load on the server and tries to place new ugens on the least loaded servers.

We have observed that the load on a server is roughly constant if the number of groups, tracks and modules remains the same, as shown on Fig. 5. Therefore, we load-balance our set of servers only when adding a new group, track or module. When creating a new *group*, we place it on the server with the minimum load. We cannot choose as freely where to put tracks or modules as they are embedded into a group. For *tracks*, we just put them in the server of their group. For a new *module*, if the load on the the server of its group is too high (given a threshold α), we migrate the whole group on the server with the minimum load. If all the servers have the same high load close to the threshold, we just put the module on its group without migrating and warn the user. The migration of a group consists of the removal of all the tracks and modules in the group in the origin server and its creation on the destination server. To prevent the audio to click, we fade it out on the origin server and fade it in on the destination one. Fig. 5 shows the peak and average CPU load for one server.

Not all audio graphs can benefit from this simple parallelization approach. For example, a chain of effects (*i.e.*

one group) will be placed on the same server, irrespective of the number of ugens in the group. A more sophisticated approach in this case would be to cut the group into several ones and transport audio between the servers using the jack audio server [22] or a multiple input/output soundcard (but that would also increase latency).

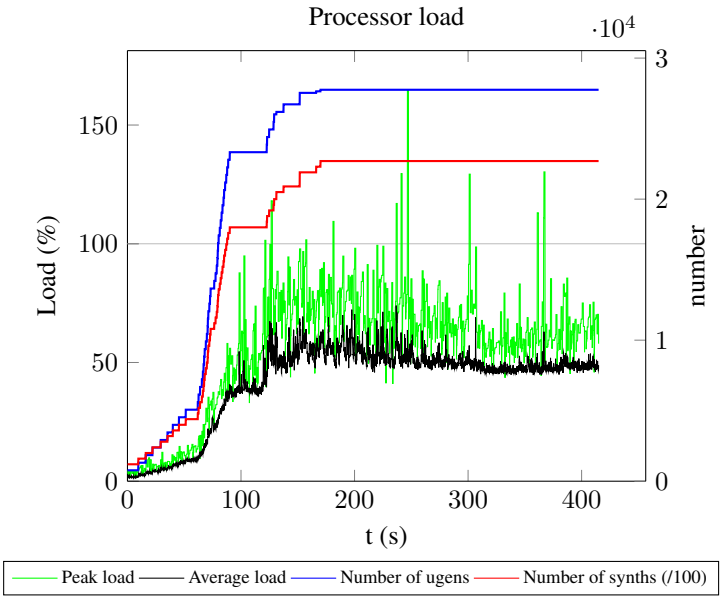

Figure 5. The CPU load (calculated as a moving average), peak CPU load, number of unit generators and number of synths, during an execution of the piece Curvatura II.

3.6 Graphical interface

In complex pieces, the current status of the audio chain is difficult to grasp. And even if Antescofo provide a textual interface for live coding, the textual control of audio chains can be painful. So we decide to develop a graphical component for the creation and the manipulation of audio processing directly in a user interface. This component is also used to visualize the content of tracks and modules dynamically. This GUI is implemented in SuperCollider sclang using the Qt embedded interface, as shown on Fig. 6.

To send the data parameters from the graphical interface to AntesCollider the user can rely on two methods. The user can create presets directly in the GUI and then import it in AntesCollider via a JSON file. Another option is to send the parameters data of a track directly from the GUI to AntesCollider via OSC. At reception, Antescofo create a new dictionary of parameters (called a MAP in Antescofo). The MAP can be saved on disk to be read latter. This MAP represents a snapshot of the node tree and can be used to create or restore an audio chain later, playing the role of a preset.

Figure 6. Overview of the GUI of AntesCollider.

4. CASE STUDY: *CURVATURA II*

Several pieces of mixed music have already been composed with preliminary versions of AntesCollider. The first one was*Dispersion de trajectoires* (2014) for baritone saxophone and electronics using the Antescofo score follower.

Curvatura II is a real time electroacoustic piece made with AntesCollider for an HOA system. The whole composition is entirely written in AntesCollider, validating the notion of *centralized executable score* where all data needed to perform the piece are gathered into a single document (an Antescofo score). The piece is performed in real time. Because of the use of SC_HOA library [23], it can be decoded in different loudspeaker configurations.

Curvatura II is first and foremost an exploration and research on the creation of different superimpositions of sound layers and on the transformation of sound material through time. The "curvature of time" corresponds to continuous deformation and distortion of the tempi (accelerando, rallentando, modulation, as shown on Fig. 7). These distortions are easily specified in the Antescofo language where all processes can be parameterized with arbitrary tempo expressions. It enables the creation of complex rhythms and simplify the computation of continuous controls on parameters for the sound synthesis as well as their superimpositions. The superposition of all these elements in constant transformation and mutation gives rise to sound aggregates to produce a global perceptual effect which, at the same time, constructs the directionality and course of the musical discourse. Antescofo processes are heavily used to generate both the micro (sound, timber, texture) and the macro structure (gesture, phrase, form) of the piece. The flexibility and CPU performance of multi server SuperCollider scsynth setup allow to create also complex texture that can be spatialised in the HOA environment.

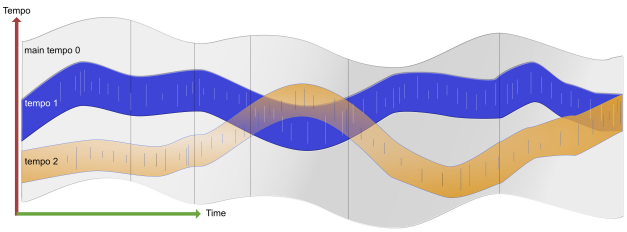

Figure 7. Graphical representation of tempi modulation for process and music events; tempo1 and tempo 2 have his own modulation tempo in time and tempo 0 is the main tempo which in turn embedded and modulated both tempo 1 and 2.

The following techniques are used in the piece, showing the versatility of the architecture:

- Different audio synthesis techniques (additive, subtractive, frequency modulation, amplitude modulation)

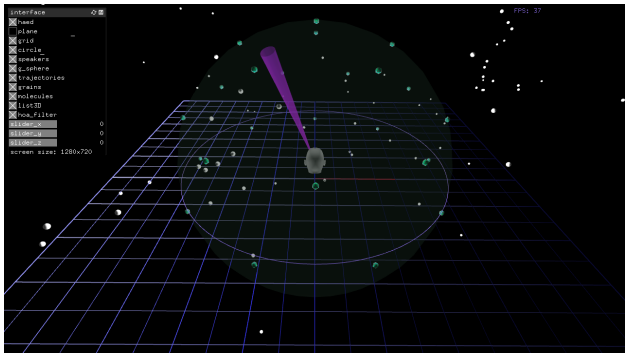

Figure 8. Sound sources used for the *Curvatura II* piece placed in space and position of the listener, thanks to HOA.

- Spatial granular synthesis: AntesCollider can generate hundreds of grain simultaneous, each grain with his own 3D position, envelope, rate, etc. This allows to create a kind of spatial grain clouds in the space.
- Concatenative synthesis with different enveloppes.
- Physical models for synthesis and control.
- Control models (3D Mass Spring, Boids, orbitals) are programmed directly in the Antescofo language using objects.

5. CONCLUSION

We created an integrated environment, *AntesCollider* for mixed music, with complex timelines and synchronizations handled by Antescofo, and advanced audio synthesis by SuperCollider. We showcase our system on a new piece for electronic music, *Curvatura II*. The system is a preliminary implementation of the notion of *centralized executable score*. A centralized score gathers in one document all the information needed for the definition of the temporal media (electronics, performer score, interactions and audio constructions) within the same language. The notion of "executable score" is motivated by the development of more dynamic, precise and musically expressive electronic scores, enabling new couplings between computers and musicians, and renewing the problem of interpretation both at the level of the composer and the instrumentalist.

The pieces developed so far with AntesCollider show that the centralized executable score is relevant for the execution of the performance. This paradigm supports the explicit writing of instrument/computer interactions and favors a dynamic approach in the management of audio processing and synthesis: audio chains are easily created and destroying "on the fly" in direct relation to internal and external events. However, more abstract representation are certainly needed, e.g. for the conductor or the performers.

Future work includes the distribution of the AntesCollider library together with the Antescofo system and several enhancement of the GUI to monitor Antescofo events and actions. We develop also several physical models in Antescofo to control synthesis.

6. REFERENCES

[1] S. Wilson, D. Cottle, and N. Collins, *The SuperCollider Book*. The MIT Press, 2011.

[2] J. McCartney, "Rethinking the computer music language: SuperCollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[3] M. Wright, "Open Sound Control: an enabling technology for musical networking," *Organised Sound*, vol. 10, no. 3, pp. 193–200, 2005.

[4] T. Blechmann, "supernova, a multiprocessor-aware synthesis server for SuperCollider," in *Proceedings of the Linux Audio Conference*, 2010, pp. 141–146.

[5] A. Cont, "ANTESCOFO: Anticipatory Synchronization and Control of Interactive Parameters in Computer Music." in *International Computer Music Conference (ICMC)*, 2008, pp. 33–40.

[6] D. Zicarelli, "An extensible real-time signal processing environment for Max," 1998.

[7] M. Puckette, "Using Pd as a score language," in *Proc. Int. Computer Music Conf.*, September 2002, pp. 184–187. [Online]. Available: http://www.crca.ucsd.edu/~msp

[8] J. Echeveste, J.-L. Giavitto, and A. Cont, "A Dynamic Timed-Language for Computer-Human Musical Interaction," https://hal.inria.fr/hal-00917469/document, Tech. Rep., 2013.

[9] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artificial intelligence*, vol. 8, no. 3, pp. 323–364, 1977.

[10] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," 1994.

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

[12] J.-L. Giavitto, J.-M. Echeveste, A. Cont, and P. Cuvillier, "Time, Timelines and Temporal Scopes in the Antescofo DSL v1. 0," in *International Computer Music Conference (ICMC)*, 2017.

[13] R. C. Boulanger *et al.*, *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming*. MIT press, 2000.

[14] J. Blondeau, "Espaces compositionnels et temps multiples: de la relation forme/matériau," Ph.D. dissertation, Paris 6, 2017.

[15] P. Donat-Bouillud, J.-L. Giavitto, A. Cont, N. Schmidt, and Y. Orlarey, "Embedding native audio-processing in a score following system with quasi sample accuracy," in *ICMC 2016-42th International Computer Music Conference*, 2016.

[16] P. Donat-Bouillud and J.-L. Giavitto, "Typing heterogeneous dataflow graphs for static buffering and scheduling," in *ICMC 2017-43rd International Computer Music Conference*, 2017.

[17] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of Faust," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[18] J. Daniel, S. Moreau, and R. Nicol, "Further investigations of high-order ambisonics and wavefield synthesis for holophonic sound imaging," in *Audio Engineering Society Convention 114*. Audio Engineering Society, 2003.

[19] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming: The Sockets Networking API*. Addison-Wesley Professional, 2004, vol. 1.

[20] E. A. Lee, "CPS foundations," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*. IEEE, 2010, pp. 737–742.

[21] 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.

[22] S. Letz, D. Fober, and Y. Orlarey, "Jack audio Server for Multi-Processor Machines." in *ICMC*. Citeseer, 2005.

[23] F. Grond and P. Lecomte, "Higher Order Ambisonics for SuperCollider," in *Linux audio conference 2017 proceedings*, 2017.