Computer-aided Composition of Musical Processes

Dimitri Bouche, Jérôme Nika, Alex Chechile, Jean Bresson

Computer-Aided Composition of Musical Processes

Dimitri Bouche[1], Jérôme Nika [1], Alex Chechile[2], Jean Bresson[1]

[1] UMR 9912 STMS: IRCAM - CNRS - UPMC, Paris, France

[2] CCRMA - Stanford University, CA, USA

[1] **{bouche,jnika,bresson}@ircam.fr**

[2] **chechile@ccrma.stanford.edu**

Abstract

We present the formal model and implementation of a computer-aided composition system allowing for the "composition of musical processes". Rather than generating static data, this framework considers musical objects as dynamic structures likely to be updated and modified at any time. After formalizing a number of basic concepts, this paper describes the architecture of a framework comprising a scheduler, programming tools and graphical interfaces. The operation of this architecture, allowing to perform both regular and dynamic-processes composition, is explained through concrete musical examples.

> "Music, then, may be defined as an organization of [...] elementary operations and relations between sonic entities or between functions of sonic entities."
>
> Iannis Xenakis (Xenakis, 1992)

1 Introduction

Formalization of compositional processes is the cornerstone of computer-aided composition (CAC). CAC software allows composers to design computer processes generating musical structures and data. The execution of such processes is generally qualified "deferred-time" for it does not overlap with the process of rendering the produced data. We propose to extend this view of CAC with a model allowing composers to create pieces at a "meta" level, that is, producing musical structures made of running compositional processes.

Joel Chadabe's experiments (Chadabe, 1984) are inspiring pioneering works in this direction. Using the CEMS (Coordinated Electronic Music Studio – an analog-programmable music system built by Robert Moog) he could divide the composition process in two stages: first, the composer designs a compositional process; then, s/he interacts with the machine playback according to the initial design using analog inputs. In this context user actions induce long-term reactions in the music generation, and the computer outputs may in turn affect instant or future decisions of the composer. In this paper, our stance is to go beyond this idea: not only allowing real-time control over processes, we aim at providing tools to arrange and connect multiple music-generation processes to build dynamically evolving musical structures. We envisage a score whose content is never "frozen," but rather consists of a program which output can evolve dynamically through time while being played, according to actions or controls from performers, from the external environment, or following predefined rules (see Figure 1). In other words, a program that can be monitored and modified in its entire temporality at any time. We call this idea "composing processes"; to implement it, we propose a dynamic music scheduling architecture embedded in a computer-aided composition environment.

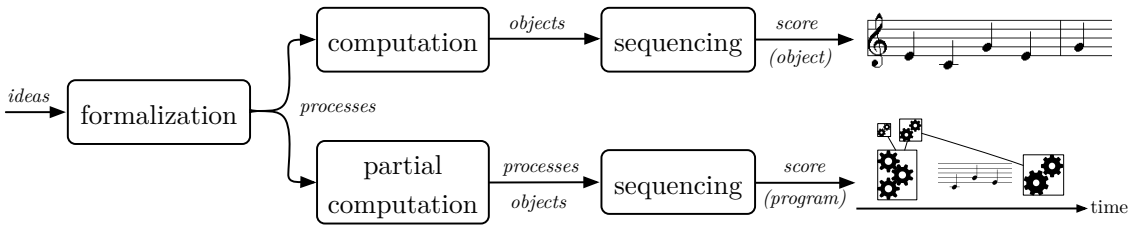

Figure 1: *Top branch: score as musical data (result of a process) – Bottom branch: score as a program (set of data, processes and rules)*

In section 2 we describe musical examples illustrating the objectives of our work. Section 3 proposes a theoretical description relating formally this work to a scheduling problem. In Section 4 we present the architecture we developed to address this problem, and Section 5 describes its implementation, as well as the realization of the aforementioned examples. Finally, Section 6 compares our system to previous related works.

2 Examples

In this section we present two examples that illustrate this idea of "composition of processes", and possible applications of a dynamic music scheduling architecture. Both examples are based on recent collaborations and ongoing musical research projects.

2.1 Dynamic music generation with formal specifications

This first example takes place in the context of automatic music generation systems combining formal specifications of temporal structures and interactivity. Such systems find applications for instance in computer improvisation (Assayag et al., 2006). The objective is to embed agents generating musical material in high-level, formal while interactive time structures. We consider the generation engine of ImproteK (Nika et al., 2016, to appear), an interactive music system dedicated to guided human-computer improvisation. This system generates improvisations by guiding the navigation through a musical "memory" using a "scenario" structure. It is constituted by a chain of modular elements: a guided music generation model; a reactive architecture handling the rewriting of musical anticipations in response to dynamic controls (Nika et al., 2015); and synchronization mechanisms to adapt MIDI or audio rendering to a non-metronomic pulse during a performance.

Figure 2 depicts a possible integration of this generation engine in a process-composition context. We consider two improvisation agents (*voice 1* and *voice 2*), playing short solo phrases in turn (question/answer alike) on a given chord progression. These solos overlap and have variable durations. The idea behind their integration is that each one is determined by the specified chord progression (the "scenario") but also by some contextual parameters depending on the recent past. The voices are aware of each other so that the beginning of each solo follows the end of the other one to ensure the musical continuity. In other words, every sequence played by an agent should influence and dynamically schedule the next sequence played by the other agent.

We also imagine real-time user control over some parameters, such as the frequency of the trade between voices.

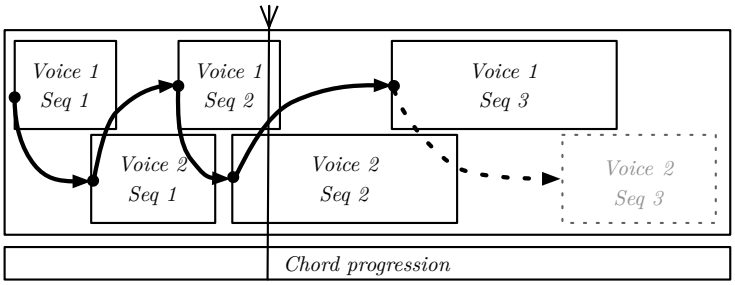

Figure 2: *Two agents generate on-the-fly question/answer matching a specified chord progression.*

2.2 Control of auditory distortion product synthesis software

In this second example, we consider a score driving the long-term control of an external system synthesizing sound in real-time. It is based on Alex Chechile's composition research on synthesis techniques for evoking auditory distortion products—sounds generated within the listener's ears from acoustic primary tone combinations (Chechile, 2015). In this context, multiple oscillators are to be controlled simultaneously at specific frequencies with precise ratios between each voice.

Here also, we consider a framework for the specification of the control process and its evolution, as depicted for instance in Figure 3. In this example, one initial control curve (corresponding to one voice) is pre-computed (at the top of the figure). User input triggers the computation and the integration of the other control curves in such a way that the result resembles the shape of an upside down pyramid: the duration of a control curve depends on the time it was created and scheduled. To produce the auditory distortion products, each added control curve must also be precisely determined and scheduled according to the main pre-computed control curve, and to the current rendering time of the process (to respect the phase and frequency ratios between the different oscillators). Finally, each control curve continuously oscillates back and forth on the time axis, producing timing variations in the resulting material.

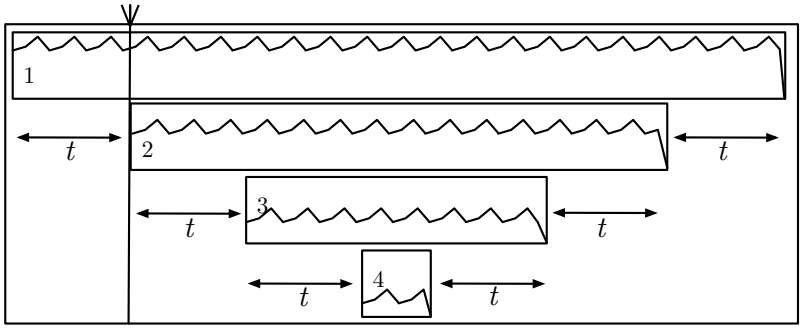

Figure 3: *Computation and integration of control curves for the control of auditory distortion product software. Curves 2 to 4 generated at time t are determined by both t and by the initial curve. Curves' onsets are slightly and continuously changing at rendering time.*

As in the previous example, here we have to deal with a dynamic temporal structure, whose content can not be fully known or predicted at the beginning of the rendering, but still implies a significant need for precise and relatively long-term schedules.

3 "Composition of processes" as a scheduling problem

From the implementation of musical ideas in compositional processes to playing musical data, computer music software has to perform the successive operations of *computation*, *scheduling* and *rendering*. We call *computation* the execution of a sequence of operations leading to the generation or transformation of musical *objects* (containers for musical data). The *scheduling* operation consists of turning this data into a sequence of timed *actions* (atomic operations) called a *plan*[1] which, when rendered, accurately charts the content of the data. For instance, a NOTE can be considered as a simple musical object; its scheduling results in creating a timed sequence of two actions to perform: one *note-on* event sent via MIDI at the note start date and one *note-off* event sent at its start date + duration. Finally, the *rendering* of a plan is performed by a process that triggers each node of the action queue on due time. We name this process a *rendering loop*.

While these successive steps are common in most musical software, they are not performed the same way according to the different types of environments. In composition-oriented software, the scheduling of musical structures is "demand driven," that is, it happens when the user wants to play the data, and rendering starts when the plan is available. In real-time software however, anything – internal or external to the system – can happen during the rendering phase to modify scheduled objects. The two stages of scheduling and rendering are therefore interleaved: if compositional processes are to be controlled in real-time (as for instance in the previous examples), the system needs to react and produce the new output (objects and plans), while the older planned output is being rendered.

Extending CAC systems to deal with dynamic compositional processes and process composition therefore requires addressing dynamic re-secheduling issues. In the remainder of this section, we define a number of terms, notations and operators which will help to better understand and design our framework from this perspective.

3.1 Definitions

Actions: Let $(\mathcal{A}, +_{\mathcal{A}}, null)$ be the set of instantaneous atomic operations to be executed by our system with a commutative and associative composition operation $+_{\mathcal{A}}$. Thanks to associativity and commutativity, a sum of actions represents another instantaneous action corresponding to the multiset of actions (the terms of the sum) that must be performed in parallel.

Dates: Let $(\mathcal{D}, <)$ be the set of dates equipped with a total order $<$. A date is a value characterizing timing informations (symbolic, absolute, relative ...). It can be converted to real time (in this paper, $\mathbb{N}$) using $eval : \mathcal{D} \mapsto \mathbb{N}$.

Objects: Let $(\mathcal{O}, +_{\mathcal{O}}, null)$ be the set of objects. An object is a set of dated actions $\{(d_i, a_i)\}_{i\in\mathbb{N}} \subset (\mathcal{D} \times \mathcal{A})$. It can be specified through its characteristic function $\mathbb{1}_o \in \mathcal{O} = (\mathcal{D} \times \mathcal{A}) \mapsto \{true, false\}$. The formal associative and commutative operation $+_{\mathcal{O}}$ represents the aggregation of two objects. $+_{\mathcal{O}}$ is required to distribute over $+_{\mathcal{A}}$: $(d, a_1) +_{\mathcal{O}} (d, a_2) = (d, a_1 +_{\mathcal{A}} a_2)$. This property simply expresses that performing two actions a_1 and a_2 at the same date d is equivalent to performing the parallel action $(a_1 +_{\mathcal{A}} a_2)$ at d.

[1] In computer science, this phase is often divided between the sub-phases of *planning* and *scheduling*, where planning consists in finding an optimal path of operations to follow among numerous ones. However, there is often only one possible way to render a set of musical data. We consider that planning is implicitly performed during the computation of musical structures.

Compositional processes: A compositional process is a sequence of operations computed to transform musical objects,[2] relying on any parameter, internal or external to the environment it belongs to. Processes can be part of objects through a special "compute" action $a_c : \mathcal{O} \mapsto \mathcal{O}$. The execution of a_c is a simple instantaneous and atomic trigger to a computation that is handled outside of the rendering loop. Thus, by definition, $a_c \in \mathcal{A}$.

3.2 Scheduling & Rendering with interleaved computations

We write $\sum_{i=0}^{n} t_i.a_i$ an execution (rendering) trace of an object, where $t_i.a_i$ means that action a_i was triggered at time $t_i = eval(d_i)$. As rendering simply consists in triggering planned actions on due time, a trace is a plan (result of a scheduling operation). Therefore, scheduling an object can be modeled as the following function:

$$\begin{aligned} [\![\ ]\!] : \mathcal{O} &\mapsto (\mathbb{N} \times \mathcal{A}) \\ \{(d_i, a_i)\}_{i \in \mathbb{N}} &\to \sum_{i=0}^{n} t_i.a_i \end{aligned} \tag{1}$$

However, when dealing with an object embedding compositional processes (computations that modify its content), scheduling cannot be performed in one step, since triggering a_c is assumed to change the plan. We define the function to schedule one action at a time, which returns a singleton plan along with the continuation of the object:

$$\begin{aligned} [\![\ ]\!]_1 : \mathcal{O} &\mapsto (\mathbb{N} \times \mathcal{A}) \times \mathcal{O} \\ \{(d_i, a_i)\}_{i \in \mathbb{N}} &\to (t_0.a_0, \{(d_i, a_i)\}_{i \geq 1}) \end{aligned} \tag{2}$$

This function allows to partially schedule objects and control the duration of plans. When reaching a "compute" action while scheduling an object, we have:

$$\begin{aligned} [\![\ ]\!]_1 : \mathcal{O} &\mapsto (\mathbb{N} \times \mathcal{A}) \times \mathcal{O} \\ \{(d_i, a_i)\}_{i \in \mathbb{N}} &\to (t_0.a_c,\ a_c(\{(d_i, a_i)\}_{i \geq 1})) \end{aligned} \tag{3}$$

These new definitions introduce a loop from the actions of the plans (real-time domain) back to the objects. This feedback loop has an important impact, for it includes the compositional process inside the rendering process. The computation and scheduling phases, which were out of the rendering loop in the traditional scheme (and therefore out of any temporal/real-time constraints) are now to be integrated in a comprehensive computation flow. The scheduling architecture described in the next section allows us to tackle these challenges.

4 System architecture

In this section we present an architecture allowing the implementation of compositional processes as described in the previous section.

4.1 Components

Our architecture, depicted in Figure 4, is modeled as a reactive system comprising three main components. The computation and scheduling operations presented in the previous section are performed respectively by the *Engine* and the *Scheduler*. These go along with a *Renderer* in charge of rendering plans.

[2] It also implicitly includes generation of new objects, since it can be modeled as the transformation of the *null* object.

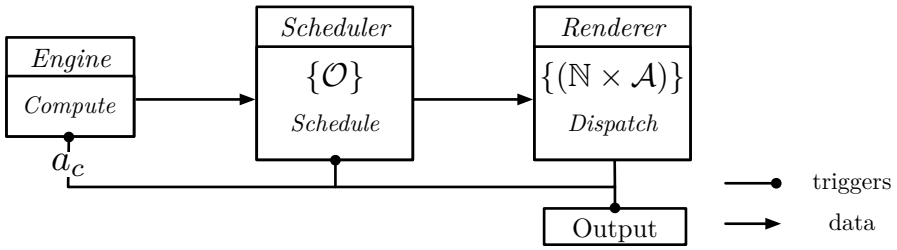

Figure 4: *Complete architecture.*

Engine: Computes compositional processes triggered by the user (e.g. to compute or modify the score from the GUI), by the external environment, or by the renderer. These computations are enqueued and processed sequentially. Their results (objects transformations or data) are sent to the scheduler. Note that the actions that trigger the engine can be executed by the renderer either on due time (as other musical actions), or ahead of their actual date (this anticipation, which can be controlled in time, is not detailed in this paper).

Scheduler: Has a register to store references to all playing objects and produces plans for them on demand (from the renderer). These plans can be time-bounded (see 4.2). When it receives new data (from the engine) for currently playing objects, the scheduler can decide whether or not to produce new plans for them. To do so, the bounds of the last produced plans are compared to the bounds of the new data.

Renderer: Runs a "rendering loop" for each plan. Triggers planned actions and requests (triggers) the scheduler for new plans before depletion (see 4.2). The rendering loop checks plans periodically to trigger actions on due time. Actions to "compute" (*i.e.* a_c) trigger the engine.

4.2 Scheduling policy

In order to be reactive to changes, the scheduler in our architecture must remain active during rendering in order to perform rescheduling operations when required. Scheduling however is not performed in zero-time: the duration of this operation is proportional to the number of actions to be planned. In order to perform low-latency scheduling, we control the duration of the produced plans dynamically following an *adaptive behavior* that we describe in this section. Instead of scheduling objects, the scheduler will perform series of on-demand scheduling operations generating *sub-plans* for portions of the musical objects defined by time intervals (noted $\mathcal{I}$).

We first introduce algorithm 1, which allows to schedule elements (d, a) of an object o with the restriction $t = eval(d) \in \mathcal{I}$.

Algorithm 1 $S(o, \mathcal{I}) \equiv$ scheduling an object o in the interval $\mathcal{I}$

$(p, o') \leftarrow [\![o]\!]_1$
while $t \leq max(\mathcal{I})$ **with** $(t.a, o') \leftarrow [\![o']\!]_1$ **do**
  $p \leftarrow p + t.a$
end while
return p

When the renderer receives a sub-plan from the scheduler, it starts rendering it and triggers

the scheduler to prepare the next one: the scheduler and renderer operate in parallel. Therefore, if we note $\Delta S(o, \mathcal{I}_k)$ the duration of the scheduling operation $S(o, \mathcal{I}_k)$, the following condition has to remain true to avoid latency issues:

$$\forall k \in \mathbb{N}^*,\ \Delta S(o, \mathcal{I}_k) < length(\mathcal{I}_{k-1}) \tag{4}$$

The main idea of the *adaptive behavior* is that the interval $\mathcal{I}$ progressively grows while the object does not change, in order to limit the resources spent on scheduling operations during the static portions of rendering. To do so, we first partition the real time domain (here $\mathbb{N}$) into disjoint intervals of variable, increasing size ($\mathbb{N} = \mathcal{I}_1 \cup \mathcal{I}_2 \cup \dots$). The lengths of the intervals are empirically defined so that they grow fast enough to limit the number of scheduling requests, and slowly enough to hold the previous condition true. They are currently implemented as a geometric progression:[3]

$$\begin{aligned} length(\mathcal{I}_0) &= L_{min} \\ length(\mathcal{I}_{k+1}) &= length(\mathcal{I}_k) * r, \\ r &\in]1; +\infty[\end{aligned} \tag{5}$$

L_{min} is the lowest possible duration for a plan, and r is the ratio of the geometric progression.[4]

If the engine ends a computation at a time t_c (relative to the object rendering start time) that modifies an object while one of its sub-plans is being rendered, it notifies the scheduler which decides, depending on the temporal scope of the modification and the length of the current interval, if a rescheduling operation is necessary. We call this strategy (algorithm 2) a *conditional rescheduling*. In this case, the interval progression is reset, assuming that further short-term modifications are likely to happen again.

Algorithm 2 Conditional rescheduling

▷ $t_c \leftarrow$ computation's end date
▷ $\mathcal{I}_k \leftarrow$ interval of currently rendered plan
▷ $\mathcal{I}_M \leftarrow$ temporal scope of the modification

if $(\mathcal{I}_k \cap \mathcal{I}_M \neq \emptyset) \wedge (length(\mathcal{I}_k) > L_{min})$ **then**
  $\mathcal{I}_{k+1} \leftarrow [t_c + L_{min}; t_c + 2 * L_{min}]$
  $S(o, \mathcal{I}_{k+1})$
end if

This reactive architecture allows for quick computation of new plans when a perturbation occurs: the modifications are taken into account in the rendering with a fixed latency of L_{min}. Then the intervals' length progression starts over to reduce the number of scheduling operations. Figures 5 and 6 summarize possible interval progressions, respectively without and with a reaction to perturbation of the plan.

Figure 7 shows an execution timeline for the three main components of the architecture corresponding to the rendering of an object, with one "dynamic" computation occurring in the system. The object is scheduled at (1), which produces a first partial plan for $\mathcal{I}_0$ that the *renderer* starts to render at (2) (instantly once the plan is available). During the interval $\mathcal{I}_0$ the *scheduler* prepares a second, longer plan to be rendered for $\mathcal{I}_1$, and so on. A new computation is triggered in the *engine* at (3) (for instance, coming from a user control). In the middle of $\mathcal{I}_2$ this computation

[3] This progression is bounded by the object duration: $\forall k \in \mathbb{N}^*,\ sup(\mathcal{I}_k) = min(sup(\mathcal{I}_k), max(eval(d_n), n \in \mathbb{N}))$.

[4] In our current implementation, $L_{min} = 10$ ms and $r = 2$.

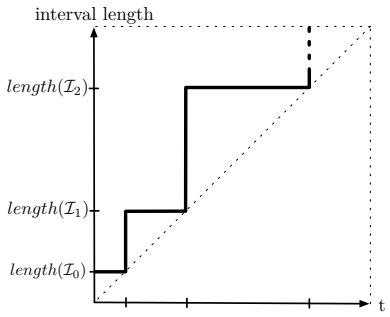

Figure 5: *Interval duration – idle behavior.*

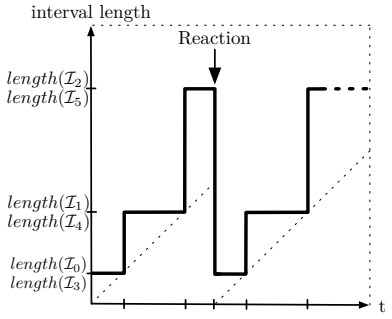

Figure 6: *Interval duration – reactive behavior.*

terminates: the *scheduler* restarts the adaptive loop by generating a new short plan for $\mathcal{I}_3$, replacing the planned end of $\mathcal{I}_2$.

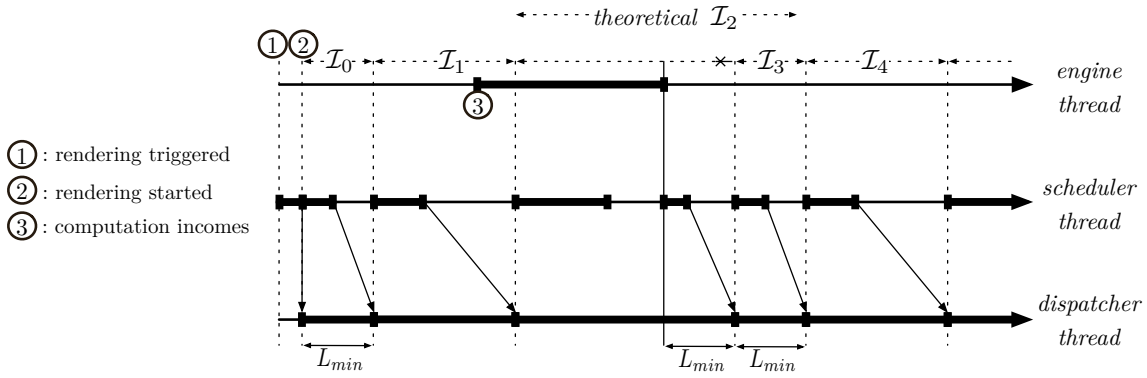

Figure 7: *Behavioral timeline of the components of the architecture.*

5 Implementation

5.1 Computer-Aided Composition environment (OpenMusic/*maquette*)

Our system is integrated in the OpenMusic computer-aided composition environment (Assayag et al., 1999). OpenMusic is a visual programming language used by composers to generate musical structures (following the traditional/static paradigm described at the beginning of this article). This environment provides a number of interesting aspects that will allow us to carry out the "composition of processes" as described in the previous section:

- A visual programming framework allowing composers to easily design the processes;
- Graphical user interfaces and editors;
- Advanced temporal structures (*maquettes*) used as a formal model for integrating our system;
- New *reactive* features allowing the user to jointly program and execute offline musical processes and react to user controls and external events (Bresson and Giavitto, 2014).

In OpenMusic, a *maquette* is a two-dimensional container for compositional processes, embedding both musical data (score extracts, sounds, etc.) and processes (visual programs), which

can be functionally connected to each other. The horizontal coordinate is the time dimension and determines a temporal layout for the objects and data resulting from the computation of the processes. The *maquette* was therefore defined as a hybrid, unifying environment between a (visual) program and a score. The evaluation of a maquette occurs prior to its rendering. It executes all internal processes and collects their results (objects) in a comprehensive timed data structure, which can be played (rendered) or further processed in the programming environment.

In (Bresson and Agon, 2006) an extension of the *maquette* was proposed, including a *synthesis patch* called prior to rendering and responsible for the computation of the musical structure ($o \in \mathcal{O}$) given the contents of the *maquette* (that is, gathering the results of all processes in order to build a global structure – for instance, to create a sound from a set of synthesis parameters).

In terms of our previous formalism, the *maquette* corresponds to an object $o = \{(d_i, a_{c_i})\}_{i \in \mathbb{N}}$ (container for *compositional processes*). Its evaluation is the sequential execution of a set of "compute" operations a_{c_i}, which correspond to the programs contained in the maquette, and its rendering implies a single, preliminary scheduling operation $[\![o]\!]$ for the objects it contains and/or resulting from program executions. Although made of processes, the musical structure is therefore a static one (it will not change at rendering time, after these initial phases of execution and scheduling are performed).

5.2 Application Programming Interface (API)

Our architecture comes along with a programming interface. We provide a small set of operations which allow the manipulation of objects and processes. Note that in the following definitions, $o = \{(d_i, a_i)\}_{i \in \mathbb{N}}$ and a score s is an object $\in \mathcal{O}$ aggregating other objects.

- **add(s, o, d)**: add an object o in a score s at date d;
  $\equiv s \leftarrow s \cup \{(d + d_i, a_i)\}_{i \in \mathbb{N}}$
- **remove(s, o)**: remove an object o from a score s;
  $\equiv s \leftarrow s \setminus o$
- **move(s, o, Δd)**: move an object o in a score s by adding Δd to its date.
  $\equiv \forall (d_i, a_i) \in s \cap o,\ (d_i, a_i) \leftarrow (d_i + \Delta d, a_i)$

As we aim to provide tools for automating the behavior of the score content, we also propose a set of objects to trigger user defined actions (subsequently denoted λ):

- **Clock = <$p, \lambda(i), d$>**: calls $\lambda(i)$ with i incrementing from 0 each p dates while $n * p < d$;
  $\equiv o = \{(p \times i, \lambda(i))\}_{i \in \mathbb{N},\ p \times i < d}$
- **BPF = <$\{d_i, v_i\}_{i \in \mathbb{N}}, \lambda(v)$>**: calls $\lambda(v_i)$ at date d_i;
  $\equiv o = \{(d_i, \lambda(v_i))\}_{i \in \mathbb{N}}$
- **Interface = <$type, \lambda(v), v_{min}, v_{max}$>**: creates a visual interface of *type* (slider, knob, button etc.) to manually trigger $\lambda(v)$ following user events, with $v_{min} < v < v_{max}$.

5.3 Real-time control over compositional processes

The notion of process composition extends the concepts found in the *maquette* (see section 5.1) by bringing the computation and rendering steps of its execution together in the same process.

We are working in a prototype environment derived from OpenMusic, designed to embed this research. The core scheduling system of this environment is implemented following the architecture described in Section 4.1. It allows the user to play musical objects concurrently in the

graphical environment. The rendering of the prototype *maquette* interface used in our examples, currently represented as a superimposition of separate tracks, is implemented as a set of calls to the scheduling system as described in section 4.2.

Objects in the *maquette* interface are actually visual programs generating data. They can be set to compute this data either "on-demand", prior to rendering (as in the standard *maquette* model), or dynamically at runtime (typically when the playhead reaches the corresponding "box" in the sequencer interface). These objects can be linked to their context (and in particular to their containing maquette), so their computation can modify it using the different available calls from the system API. The *synthesis patch* – here called *control patch* – is active during the rendering and can also dynamically modify the contents of the *maquette* (for instance through a reactive OSC receiver thread, triggering some of the modification methods from the API listed in Section 5.2).

5.4 Applications

In this section, we propose an implementation of the two examples from Section 2 using the presented system.[5]

5.4.1 Improvisation agents

In order to build the example described in section 2.1, we first implement the musical *agents* as visual programs (patches) in OpenMusic (see Figure 8). The ImproteK generation model implemented in OpenMusic is used to produce musical content (here, a sequence of notes) by navigating through a musical memory (here, a collection of recorded jazz solos) to collect some sub-sequences matching the specification given by the scenario (here, a chord progression) and satisfying secondary constraints. Each *agent* embeds a reactive handler and is able to produce such musical sequences on demand.

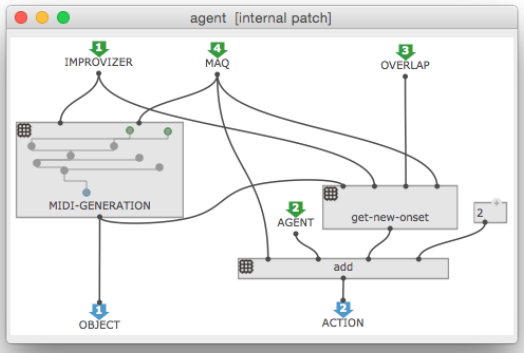

Figure 8: *Improvisation agent. Generates a MIDI sequence and schedules a generation of the other agent at the end.*

The process is computed by the *engine* dynamically, when an agent is reached by the score playhead. It performs two operations:

1. It generates some data (a MIDI sequence) according to the scenario (chord progression) and to the solo previously generated by the other agent – the scheduling of this data is

[5] Videos can be viewed at http://repmus.ircam.fr/efficace/wp/musical-processes.

automatically handled by the system.

2. It launches a new computation generating another agent in the other voice (user-defined behaviour, implemented using the **add** function of the API).

The two agents are roughly the same, but they use two distinct instances of the generation engine with different generation parameters (simulating two different performers).

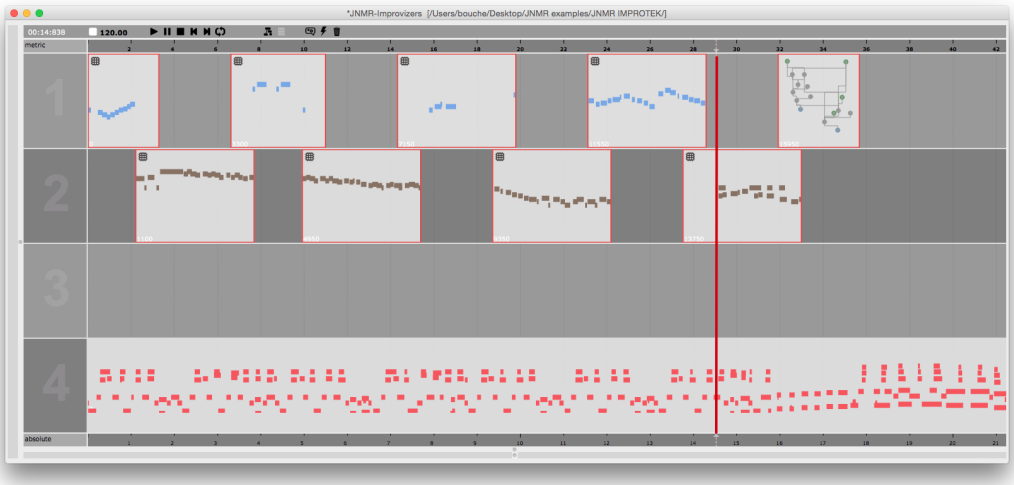

Figure 9: *Maquette rendering the improvisation example.*

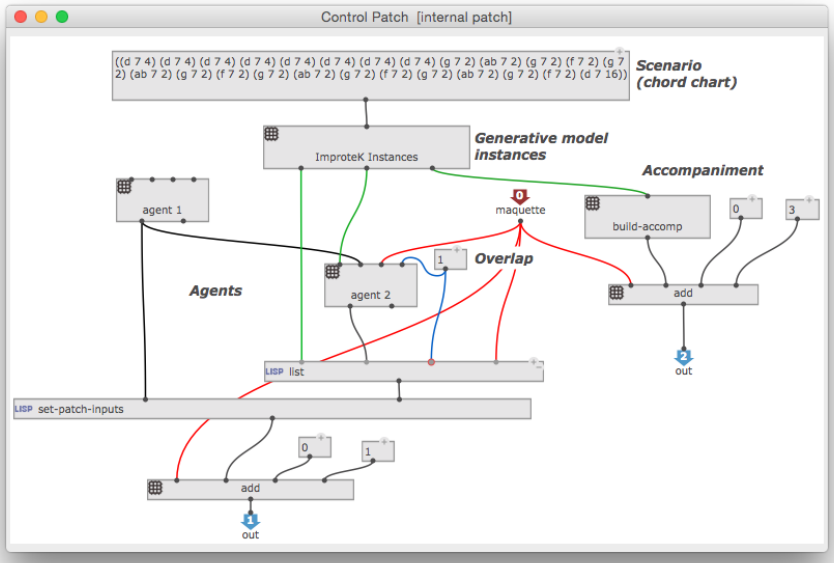

Figure 10: *Control patch implementing the improvisation example.*

Figure 9 shows the main score interface (*maquette* prototype) and Figure 10 shows the attached *control patch*. The preliminary evaluation of this *control patch* builds the two instances of the ImproteK generation engine, includes them within the two interconnected agents, and adds the first agent on track 1 to start the sequence. The rest of the process will unfold automatically at rendering time, computing the sequences in alternation from the two improvisation agents. To increase the musicality of this example, an accompaniment track is pre-computed (using a third instance of the generative model).

5.4.2 Control of the auditory distortion product synthesis software

This example makes use of the **add** and **move** operators from the API, and the **clock** and **interface** objects. In the *control patch*, an initial object is set up with a list of frequencies, a duration, a tempo, a port and an address for OpenMusic to communicate with Max (Puckette, 1991), which runs custom software and a bank of oscillators.

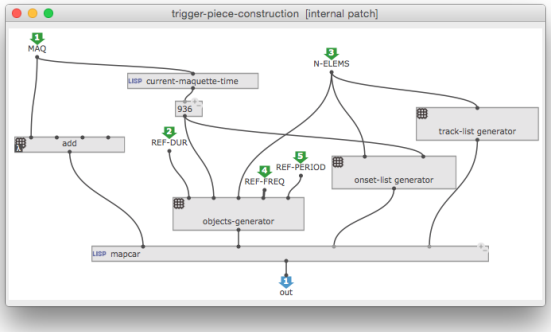

Figure 11: *Piece constructor patch.*

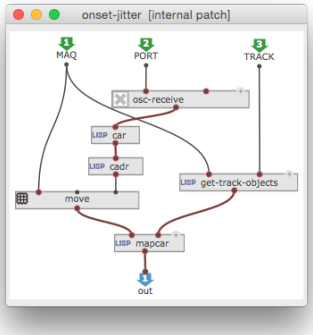

Figure 12: *Reactive jitter agent patch.*

During playback, mouse input (implemented using the **interface** instantiated in the *control patch*) triggers the generation and scheduling of additional curves in the *maquette*, each of which are multiples of the original sequence. The durations of the additional tracks are inversely proportional to the duration between the start of playback and the trigger input, and therefore decrease at each subsequent track (see Figure 11).

Through OSC, OpenMusic sends frequency information to the Max oscillators, and Max sends timing information to OpenMusic. In addition to the sequences of frequency data sent to Max, the **clock** object examines the number of currently active tracks in the *maquette* and outputs a variable used in Max to determine the ratio between the additional pitches. Conversely, four continuous streams of data from Max change the position of tracks 2 to 5 in the *maquette* using four similar agents (see Figure 12), resulting in sequences slightly moving ahead and behind time during playback. Figure 13 and 14 show the *maquette* and the *control patch* after the curve construction was triggered.

6 Related Works

6.1 Computer Music

The idea of "composition of processes" was used in early computer music systems such as *Formes* (Rodet and Cointe, 1984), a language explicitly designed to manipulate "processes" (*active*

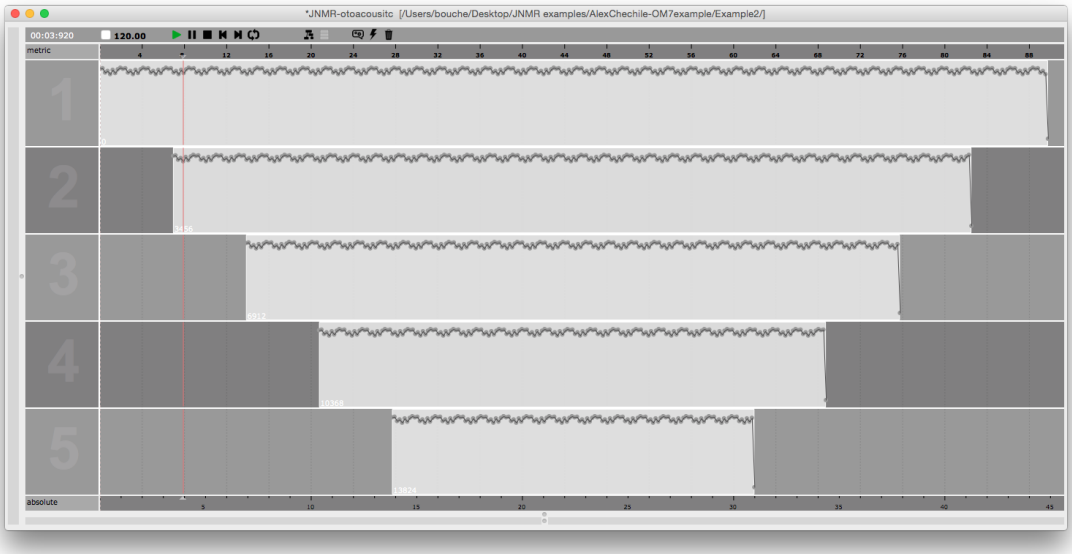

Figure 13: *Maquette rendering the auditory distortion product synthesis example.*

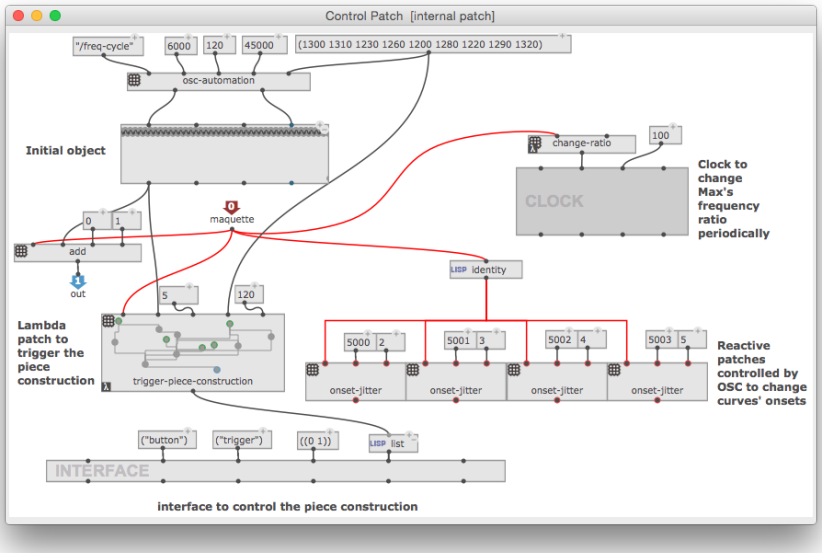

Figure 14: *Control patch implementing the auditory distortion product synthesis example.*

objects (Cointe, 1983)) for controlling a sound synthesizer. As in our architecture, concurrency and asynchrony in *Formes* were handled by message passing and through a scheduling system. While *Formes* processes depict long-term structures, the system uses a "step by step" computation model. Therefore, the computation time is actually strongly linked to the real time and the high-level composed material resulting from processes execution can only be observed *a posteriori* by

monitoring the output of the system. A similar idea was more recently put forward in the real-time extension of the *Elody* music composition environment (Letz et al., 2000), where processes applying transformations on a real-time input can be manipulated as static objects on a timeline. Such system therefore interleaves composition and rendering phases. This specificity is also present in projects like the *Bach (automated composer's helper)* library for Max (Agostini and Ghisi, 2013). *Bach* aims at offering real-time control over compositional processes, implementing list notation and formal structures in the Max real-time, performance-oriented environment. Therefore, the user can monitor long-term musical structures while modifying the parameters of their computation. Although sharing comparable general objectives, the presented architecture lays in an opposite view the structure and execution of compositional processes and aims at programming scores as processes with erratic computations and an global musical timeline.

Interleaving rendering and composition can be used to build dynamic/interactive scores. Nowadays, two main computer music systems also share this outlook:

– *i-score* is an interactive sequencer for intermedia creation (Baltazar et al., 2014). This software allows the user to define temporal scenarios for automations and events, with chronological relationships and constraints between them. The sequencer is designed in a multi-linear way: each object has its own temporality, and may have its own playhead according to execution delays introduced by the scenario or interactions. This representation differs from the one we presented, which is simply linear, using mobile boxes instead of playhead split. While *i-score* does not provide composition-oriented tools, it allows the definition and manipulation of time relations between predefined control objects.

– *Antescofo* is a score following system coupled to a synchronous programming language for musical composition (Echeveste et al., 2013). It is used to synchronize an instrumental performance and electronic/computer-based actions. One can program processes with their own temporal scope, relying on any variable they can listen to (as detected tempo for example). This results in an electronic score that goes along with the instrumentalist's score, allowing for any complex temporal relationships to be implemented between these two. The produced score can be termed interactive since its timeline is linked to the performer's timing information. Computation and scheduling are different from our work by the synchronous nature of the language, made for an event-based time flow.

6.2 Computer Science

The new CAC techniques we introduced are modeled as a scheduling problem, which involves multiple notions across general computer science.

The issues introduced by frequent and non-deterministic changes in a rendering scenario can be found in different fields. It is formally studied in general computer science and termed as replanning/rescheduling. Multiple strategies exist to tackle these issues, the most widely used method relies on reactive scheduling. Notably used in robotics (Hernandez and Torres, 2013), reactive scheduling involves planning only one action at a time, thus offering the best possible reactivity. However in our case, the prior knowledge provided by the composer and the score s/he writes allows the scheduler to anticipate. Coupled with the dynamic data changes taken into account in the rendering, an intermediate strategy between static and reactive scheduling is established, with particular effort in avoiding too frequent computations.

Our architecture is based on a model where the execution system (the renderer) and the scheduling system run in parallel to allow frequent plan revising. Previous work, especially applied to video games, established such framework. For instance, while following an initial guideline, Vidal and Nareyek (2011) allow story changes to occur according to player actions

during a game execution. They also introduce "partial plans" that could be related to our notion of sub-plans. However, a plan here represents a set of high-level actions (potentially made of multiple lower-level actions) to reach some state in the story, while in our case, plans depict a time horizon until a point when the score can be considered as static. As a result, our system can be considered as time (and event) driven while Vidal and Nareyek's system is state/goal driven (thus closely related to classical planning (Rintanen, 2013)).

Finally, dynamic time horizon of the plans described in this paper is similar to dynamic buffer sizing (Raina et al., 2005) used in data networking. However, the key features of this technique are to reduce latency during re-scheduling operations and to maintain musical coherence at the output. In data networking, it is mostly used to avoid data overflow and to adapt resource usage.

7 Discussion and conclusion

We presented a formal framework and a musical system scheduling architecture that supports the high-level specification, execution and real-time control of compositional processes. The examples used throughout this work illustrate how composers can use such system to compose at a "meta" level, that is, to create a score including compositional processes running during the execution of the piece. Implementing the system in OpenMusic allows the user to take advantage of the capabilities of one of the most flexible CAC packages for expressing many types of musical ideas. While the architecture aims at being used in computer music software, it could be implemented in any software that requires the programmation of evolving scenarios, with chronological representation of mixed data.

The asynchronous nature of our system leads to non-deterministic score rendering: we do not consider or check the actual feasibility of processes defined by composers. Thus, a score may produce non-relevant musical output if, for instance, computations involved in its execution do not terminate on time.[6] Therefore, one has to pay attention to the complexity of its development, and experiment by running the system. In this respect, we are currently investigating techniques to efficiently anticipate and organize computations. We are also working on robustness against time fluctuations due to lower-level OS-controlled preemptive scheduling (as the computer music systems we consider are deployed on general audience machines), and on a single threaded version of the architecture which in order to enable a "synchronous" rendering mode. This would lead the system towards more deterministic behaviors, and for instance let computations delay the score execution, a useful feature for composers wanting for instance to gradually compute scores using physical input (with no need for precise timing, not being in a performance context). Finally, some field-work with composers will help to refine the system and develop tools enabling better control over the introduced dynamic and adaptive scheduling features.

References

Andrea Agostini and Daniele Ghisi. Real-time computer-aided composition with bach. *Contemporary Music Review*, 32(1):41–48, 2013.

Gérard Assayag, Georges Bloch, Marc Chemillier, Arshia Cont, and Shlomo Dubnov. Omax Brothers: A Dynamic Topology of Agents for Improvization Learning. In *Workshop on Audio and Music Computing for Multimedia, ACM MultiMedia*, Santa Barbara, CA, USA, 2006.

[6] Here, "musical output" may not only refer to sounds or control messages, but also to the score display.

Gérard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon, and Olivier Delerue. Computer-assisted composition at ircam: From patchwork to openmusic. *Computer Music Journal*, 23(3): 59–72, 1999.

Pascal Baltazar, Théo de la Hogue, and Myriam Desainte-Catherine. i-score, an Interactive Sequencer for the Intermedia Arts. In *Joint Fortieth International Computer Music Conference / Eleventh Sound and Music Computing Conference*, Athens, Greece, 2014.

Jean Bresson and Carlos Agon. Temporal Control over Sound Synthesis Processes. In *Sound and Music Computing*, Marseille, France, 2006.

Jean Bresson and Jean-Louis Giavitto. A Reactive Extension of the OpenMusic Visual Programming Language. *Journal of Visual Languages and Computing*, 4(25):363–375, 2014.

Joel Chadabe. Interactive composing: An overview. *Computer Music Journal*, 8(1):22–27, 1984.

Alex Chechile. Creating spatial depth using distortion product otoacoustic emissions in music composition. In *International Conference on Auditory Display*, Graz, Austria, 2015.

Pierre Cointe. Evaluation of Object oriented Programming from Simula to Smalltalk. In *Eleventh Simula Users'Conference*, Paris, France, 1983.

José Echeveste, Arshia Cont, Jean-Louis Giavitto, and Florent Jacquemard. Operational semantics of a domain specific language for real time musician-computer interaction. *Discrete Event Dynamic Systems*, 23(4):343–383, 2013.

Josue Hernandez and Jorge Torres. Electromechanical design: Reactive planning and control with a mobile robot. In *Proceedings of the 10th Electrical Engineering, Computing Science and Automatic Control (CCE) Conference*, Mexico City, Mexico, 2013.

Stephane Letz, Yann Orlarey, and Dominique Fober. Real-time composition in elody. In ICMA, editor, *Proceedings of the International Computer Music Conference*, Berlin, Germany, 2000.

Jérôme Nika, Dimitri Bouche, Jean Bresson, Marc Chemillier, and Gérard Assayag. Guided improvisation as dynamic calls to an offline model. In *Sound and Music Computing (SMC)*, Maynooth, Ireland, 2015.

Jérôme Nika, Marc Chemillier, and Gérard Assayag. Improtek: introducing scenarios into human-computer music improvisation. *ACM Computers in Entertainment, special issue on Musical Metacreation*, 2016, to appear.

Miller Puckette. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, 15(3):68–77, 1991.

Gaurav Raina, Donald F. Towsley, and Damon Wischik. Part 2: control theory for buffer sizing. *Computer Communication Review*, 35(3):79–82, 2005.

Jussi Rintanen. Tutorial: Algorithms for classical planning. In *23rd International Joint Conference on Artificial Intelligence*, Beijing, China, 2013.

Xavier Rodet and Pierre Cointe. Formes: Composition and scheduling of processes. *Computer Music Journal*, 8(3):32–50, 1984.

Eric Cesar Jr. Vidal and Alexander Nareyek. A real-time concurrent planning and execution framework for automated story planning for games. In *AAAI Technical Report WS-11-18*, 2011.

Iannis Xenakis. *Formalized Music: Thought and Mathematics in Composition (revised)*. Harmonologia series. Pendragon Press, 1992.