In composing *Continuum*, Jung's use of *jey-noise* transformed into the first generation materials, but then proceeds to ones made in advance of this generation, which are paradoxically turning them into their offspring. Jung then derives these materials out of the same lineage, organizing the sound materials into distinctive sections then concatenating sections in a sequence so that each section contrasts to adjacent ones. Similar to Koenig, who treated "the possible form-sections" as "closely linked … without having a goal-oriented relationship to each other"[3], Jung's *Continuum* also attempts to derive a diachronic narrative from within the relations of the coexisting materials.

4.2 Generative Bootstrapping

Bootstrapping is a process of deriving probability distribution of a population. The law of large numbers states that, when proceeding statistical trials, the greater the number of random samples present the greater the precision of any probability distribution analysis. However, generative bootstrapping pursues to elicit dynamics rather than precision. Seen within a limited number of trials, the analysis data of any population of random numbers correspondingly differs from the given PDF prescription by any degree. As a consequence, an incorrect analysis data becomes the new PDF list and the same recursion repeats itself. Such a feedback implements Koenig's concept of "serial" within statistical trials. Here too, an inexactitude of distribution analysis data paradoxically guarantees change.

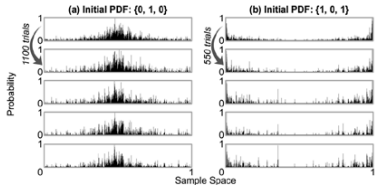

Figure 10. Each family originates from a distinctive list.

The number of trials determines how fast changes in probability data will be. Also, the "interpolation mode of the sample space of noise transformation" and the "resolution of histograms" affect the tendency of change in probability distribution data. This process and *jey-noise* were used in a live-performance entitled *Noise Gallery I*[4].

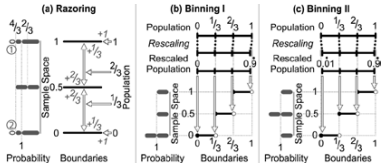

Figure 11. The PDF analysis methods are shown.

Let us now examine how a population was analyzed through the use of generative bootstrapping. The processes in Figure 11 show the analyses of the distribution of a population—including 4 samples {0, 1/3, 2/3, 1}—by the resolution of 3. So far in this paper, the process (a)—that can be understood as feeding samples into a CDF—was used in making histograms. In the case of (a), the direction of the descending samples run parallel to the boundaries, this means that only samples 0 and 1 will fall into the boundaries and the corresponding probability values will increase by 1. The other samples, 1/3 and 2/3, increase the probability values of adjacent boundaries by fractions. In case of the probability values on both ends, incrementing by fractions, only occurs from one side. ① and ② thus show how an incrementing 1/3 affects the probability values that correspond to the sample 0 and 1, which were doubly weighted.

When using *binning*—that uses an inverse CDF—in deriving histograms, the samples of the given population is always rescaled. The process (b) involves "$0 \leq S < 1$ rescaling"—ensuring that the range of samples does not equate to exact 1—and results in the probability plot where the peak is on the leftmost sample. The process (c) involves the rescaling range of $0 < S < 1$ and re-positions the peak in the probability plot to be in the middle of the sample space. Furthermore, successive histograms made by using (c) will show an oscillation of density along the y-axis and an expansion along the sample space.

5. CONCLUSIONS

All processes so far discussed have already been used as part of modules integrated into either a live-performance system or voltage-controlled patches. Additionally, to elicit a variety of behaviors from this approach, it is conceivable to plan to expand the types and kinds of available modules. Also, introducing wave analysis methods will enrich the period-by-period synthesis technique. But at present there are still many possibilities of using noise transformation, especially for creating random number streams with incredibly long repetition period. Given this the authors of this paper admit to an overall curiosity to explore the technical concept of achieving consistency between the PDF prescriptions of noise transformation and the spectra of *jey-noise*. *Gendee* will furthermore be improved, soon able to rotate any given curve. Lastly, given that the histogram (d)—Figure 6—results in weak ripples on the envelope, it is foreseeable that a pseudo inverse CDF of the $sin^2(x)$ function will be used for future filter design.

Acknowledgments

The authors are indebted to the editing of Johan van Kreij.

6. REFERENCES

[1] C. Ames, "A Catalog of Statistical Distributions: Techniques for Random, Determinates and Chaotic Sequences." *Leonardo Music Journal* 1, no. 1 (1992): 55-70.

[2] S. Luque, "The Stochastic Synthesis of Iannis Xenakis." *Leonardo Music Journal* 19 (2009): 77-84.

[3] G. M. Koenig, "Genesis of Form in Technically Conditioned Environments." *Interface* 16, no. 3 (1987): 165-175.

[4] https://soundcloud.com/user-30688271

Recreating Gérard Grisey's Vortex Temporum with cage

Daniele Ghisi
STMS Lab (IRCAM, CNRS, UPMC)
Paris

Andrea Agostini
Conservatory 'G. Verdi'
Turin

Eric Maestri
LabEx GREAM, Université de Strasbourg
CIEREC, Université de Saint-Etienne

ABSTRACT

This paper shows how a significant portion of the first movement of Gérard Grisey's Vortex Temporum *can be implemented using a subset of the* cage *package for Max, aimed at representing and rendering* meta-scores*, i.e., musical scores whose notes or chords represent complex processes, rather than simple musical items. We shall also attempt a solid definition of meta-score, and describe the rationale and workings of the* cage.meta *system.*

1. INTRODUCTION

Vortex Temporum [1] is a highly formalized chamber music composition in three movements, written between 1994 and 1996 by French composer Gérard Grisey, and widely acclaimed as one of the masterpieces of the late twentieth century music. Among the literature discussing various aspects of the piece, a very detailed technical analysis of the musical formalization techniques and mechanisms it employs has been carried out by Jean-Luc Hervé [2]. Starting from this analysis, we decided to try to recreate a substantial portion of the first movement as a case study for the *cage* [3] and *bach* [4] systems, which are two Max packages devoted to computer-assisted composition and musical notation, with *bach* providing a low- and middle-level infrastructure, implementing dedicated data structures and musical notation editors, and *cage*, which depends on *bach*, implementing higher-level musical processes and helper tools. In particular, we based our work on one specific subset of the *cage* package, the *cage.meta* system, aimed at representing and rendering what we call *meta-scores*. By this term, we mean musical scores (in the wide sense of symbolic timelines), each of whose individual events represents a complex musical process, rather than a simple musical item such as a note. In this paper, we describe the *cage.meta* system, and how it has been used for recreating a portion of *Vortex Temporum*. [1]

[1] The patches described in this article can be downloaded at the url `http://data.bachproject.net/examples/icmc2016.zip`, and require Max 6.1.7 or higher (`http://cycling74.com`), bach v0.7.9 or higher, and cage v0.4 or higher (both downloadable at `http://www.bachproject.net`).

2. META-SCORES

2.1 Composition and hierarchies

Most musical compositions can be described, at least partially, as constituted of hierarchically-nested structures [5, 6]. According to the musical style and genre, as well as the features of each individual composition and the analytical approach, these structures can be more or less numerous, and the hierarchy can be deeper or shallower. They can represent a hierarchical subdivision of time, as in the classical sonata form in which three major structures (exposition, development, recapitulation) are in turn composed of smaller structures (e.g., the exposition is often described as being composed by a first thematic area, a modulating bridge, a second thematic area, and a conclusive section), which can be divided further (e.g., a thematic area can be divided into one or more themes, and their elaborations and connections), and further (e.g., a theme can be divided into phrases or periods), and further (e.g., a phrase can be divided into semiphrases), and further (e.g., a semiphrase can be divided into motivic cells); or they can represent musical objects that can overlap in time (e.g., in the exposition of a fugue one can discern a subject and a countersubject, which are meant to be played in counterpoint, one against the other); or both kinds of structure can be discerned in the same composition, at different levels of the hierarchy. Moreover, even in the case of a temporal hierarchy, different sections can overlap, and their boundaries can be blurred. This leads to scenarios whose degree of complexity can be quite high.

On the other hand, this kind of hierarchically-structured representation of a piece of music can be a powerful conceptual tool for musical composition. Composers often take advantage of nested structures, even in programmatically 'frugal' stylistic contexts such as rock music (it is hard to imagine a song like Nirvana's *Smells like Teen Spirit* not having been conceived with a clear architectural project of alternation of verse and chorus, each built upon the repetition of a four-bar melodic pattern, based in turn upon a double iteration of a two-bar harmonic pattern). In more complex scenarios, it may be debated whether structures which can be revealed by means of musical analysis are part of a rational, deliberate structural plan, or are the product of the composer's 'intuition', whatever meaning one wants to assign to the term. In general, though, it can be safe to assume that most musical compositions have been at least partially conceived in the rational terms of some kind of hierarchical structure.

Software systems for computer-aided composition mostly focus on the manipulation of basic musical elements, such

as notes and chords, or small-scale phenomena, such as melodic profiles. However, some of these systems provide tools for dealing with higher-level musical objects, treated as containers or generators of lower-level elements, one notable example being OpenMusic's *maquette* [7].

2.2 Hybrid scores

The concept of musical score is a crucial one. For Lydia Goehr, a score "must specify at least everything that is constitutive of a work" [8] and for Nelson Goodman it is "a notational language" in which "the compliants are typically performances" [9]. In short, the score is a prescriptive means instantiated by performances.

Since the 1950s, the role of musical notation has undergone a number of generalizations (including graphical scores, gestural scores, morphological scores in electroacoustic music). This evolution has been caused, in part, by the development of electronic music. Pierre Schaeffer defined two kinds of score: a descriptive one and an operational one [10]; Charles Seeger proposed the opposition between prescriptive and descriptive scores [11]. This differentiation is reinforced by the development of computer music: as a matter of fact, both computer programming and composing are mediated through notation [12]. Composers use scores to sketch musical ideas, formalize them into a script and communicate them to performers; computer programmers, on the other hand, mostly use symbol-based formal languages to convey instructions to a computer. In both cases, notation is prescription. Both musical notation and programming are systems of prescription of actions, specifically in the case of musicians that must activate a vibrating body, and tasks, in the case of computers, that activate the vibrating body of the loudspeaker mediating the intentionality of the musician: the combination between the two defines a *hybrid* dimension of musical scores as partially suggested by Simon Emmerson [13] and by Andrew Sorensen and Henry Gardner [14].

Hybrid scores have a twofold meaning: on the one hand, they are targeted to performers, to whom they prescribe actions which are typically, although not exclusively, aimed at producing sound; on the other hand, they are targeted to computers ('digital performers' [15]), to which, through information encoded in a programming language, they prescribe the production of sound or symbols, or even more complex tasks. In particular, hybrid scores are capable of prescribing (and hence embedding) other hybrid scores within themselves, which makes them very suitable to represent and process abstract, nested musical structures. Incidentally, such hybrid scores also contribute to the narrowing of the gap between scores and instruments [12].

Within this conceptual framework, we use the term *meta-score* to define a hybrid score whose components are not elementary notational items (typically, single notes or chords), but rather processes which can be further prescribed and described as scores in their own terms. So to speak, we might say that a meta-score is a score *of* scores (i.e., a score containing other scores), or a score *for* scores (i.e., a score containing instructions to build other scores), or a score *about* scores (i.e., a score containing descriptions of other scores), hence expanding the fundamental ideas described in [16].

3. CAGE.META

3.1 The basic idea

From the very beginning of the development of the *bach* library, we were encouraged to develop a system for dealing with meta-scores within the real-time framework of Max. After careful consideration, we came to the conclusion that we would not want to implement a dedicated graphical editor and paradigm (which is what the maquette is, within the OpenMusic environment), but rather to devise a design pattern allowing the usual *bach* notation editors/sequencers (the *bach.roll* and *bach.score* objects, respectively implementing time in a non-measured, proportional fashion, and in a measured fashion, with tempi, meters, traditional note values and rests) to be used for representing meta-scores, rather than scores proper. When we had the opportunity to develop the *cage* library [3], we decided to include in it two modules devoted to this specific problem, and constituting one of the various subsets of the library itself, namely the *cage.meta* subset.

The choice of extending to meta-scores the concepts and tools used for representing traditional scores is motivated by the observation that, somehow, there is no clear boundary between traditional and meta-scores. In fact, more often than not, symbols in any traditional score refer to extremely complex processes and activities, be it the skillful control of the friction of a bow on a string, or the triggering of complex sets of envelopes to be applied to the inputs and outputs of a bank of oscillators. Moreover, in historical musical practices, there exist specific synthetic notations representing complex musical figures, such as trills, mordents, arpeggi, gruppetti and other ornamentation marks, or — even more specifically — the numbers and symbols of figured bass. By not striking a dividing line between scores and meta-scores we aim to focus on the similarities, and the continuum, between the two, rather than on the differences. At the same time, we feel that a graphical interface based upon the traditional notational paradigm can be perceived as more 'neutral' than a custom one, and as such is less likely to suggest specific compositional approaches or practices, and more inviting to be bent to each composer's creative needs.

The basic idea behind *cage.meta* relies upon the fact that scores contained in *bach.roll* or *bach.score* objects are hybrid scores, as each of their notes can be seen as a container of various, heterogeneous parameters: a small, standard set of basic, required data which define the note itself in traditional terms (position in time, expressed in milliseconds in *bach.roll*, in bars and beats in *bach.score*; duration, expressed in the same respective units; pitch; and MIDI velocity), and an optional, user-definable combination of other associated data belonging to a wide array of types (numbers, text, breakpoint functions, lists of file names, and more), contained in specialized data structures called *slots*, with the only constraint that associations between individual slots and their data types are global with respect to the score (e.g., the first slot of all the notes of a score might be a container of a number, or the sixth slot a container of a list of file names). This restriction is meaningful in that it encourages establishing a correspondance between each slot and one parameter (or one coherent and well-defined set of parameters) to be controlled through one slot. On the other hand, when more flexibility is needed, text and list [2] slots may act as dynamically-typed data containers, as they can contain any combination of numbers, symbols and sublists. In any case, the actual data contained in each note, although constrained in type by the global association, are fully independent from all the other notes in the score. Slot data can be edited both graphically and algorithmically, by means of messages sent to the containing *bach.roll* or *bach.score* object, and queried at any time. Moreover, they are always returned at play time. This means that when a note with, for example, a breakpoint function in one of its own slots is encountered during playback, all its associated data are output from one dedicated outlet of the object containing the note itself, including its onset, pitch, velocity, duration, and all the points and slopes of the breakpoint function. These points and slopes can be used, for example, to control the amplitude envelope of the synthesizer responsible to render the score into audio. Indeed, in a typical scenario, slot data contain real-time parameters for DSP, but their scope and potential is much wider.

In the *cage.meta* system, each event of a meta-score is represented as a note (or possibly a chord, as discussed further) whose first slot contains the name of a Max patcher file implementing the process associated to the event itself: we shall say that the note *refers* to said patcher. At initialization time, the patchers referred to by all the notes of the score are loaded and individually initialized. At play time, when a note is met, all its parameters and slot data are passed to the patcher it refers to. Although this is not enforced in any way, the idea is that the patcher itself will activate its intended behavior according to these parameters when they are received. Because the duration of a note is passed as one of its parameters, it is possible for the activated process to regulate its own duration according to it — but, once again, this is not enforced by any means, and it is possible to implement processes whose duration is fixed, or depends on other parameters. The same goes for the pitches and the MIDI velocities: the fact that they are passed to the process does not mean that the process itself must use them in any traditional, literal way — in fact, it can as well ignore them altogether.

In practice, all this is achieved by means of two modules, named *cage.meta.engine* and *cage.meta.header*.

3.2 Building the infrastructure

A meta-score system is built in two distinct phases. The first phase is creating the patchers implementing the processes that will be associated to the meta-score events. Each patcher must contain one *cage.meta.header* module: at play time, parameters from the referring note will not be passed to these patchers through inlets, but by the third or fourth outlet of *cage.meta.header*, according to some settings which will be described in detail below. [3] The fact that one single inlet is passing all the parameters to the patcher is not a limitation, thanks to the ability of the *llll* data structure of representing hierarchically-structured collections of data of arbitrary breadth and depth.

The second phase is setting up the meta-score system, constituted by a *bach.roll* or *bach.score* object (which we shall refer to as the 'score' object from now on) connected to a *cage.meta.engine* object in a 'loopback' configuration, such that the two inlets of *cage.meta.engine* are connected respectively to the leftmost outlet of the score object, from which all the score data can be output as one possibly large *llll*, and its one-but-rightmost outlet, from which the data of each note are output as the note itself is encountered at play time. Also, the first outlet of *cage.meta.engine* must be connected to the leftmost inlet of the score object: in this way, *cage.meta.engine* can perform queries on it and set some of its parameters if required. Indeed, if the *format* message is sent to a *cage.meta.engine* module connected as described to a score object, the first slot of the latter is initialized to the 'file list' data type (although only one file name will be used), and the second slot to the 'integer' data type (so as to store the aforementioned optional instance number). Finally, a different *bach.roll* or *bach.score* object (according to the type of the meta-score object) can optionally be connected to the second outlet of *cage.meta.engine*, so as to collect a rendered score, according to a different usage paradigm which will be discussed below.

3.3 Implementing the meta-score

Now it is possible to write the actual meta-score: as said before, the first slot of each note will contain a reference to the patcher file implementing its process. This poses a problem: what happens if the same process must be triggered by more than one note, over the course of the meta-score? One might want each run of the process to depend on the result of the previous one, or to be completely independent from it. Moreover, polyphony must be taken into account: what happens if one note triggers a process, and before that process ends (which may or may not coincide with the actual end of the note as it is written) another note triggers the same process? Most often, one will want another independent execution of the same process to start, but it is also possible that the process itself is able to manage its own polyphony, in which case the same copy of the process that was already running should receive a new activation message. All these possible scenarios are managed through the optional assignment of an instance number to each note. Two notes referring to the same file with different instance numbers will activate two distinct instances of the same patcher; if the instance numbers are the same, the same instance of the patcher will be activated. If no instance number is provided, the note will activate the so-called 'global instance' of the patch, which is the same for all the notes without an instance number, and different from all the numbered instances. As a helper tool,

[2] The ubiquitous *bach* data type is a tree structure expressed as a nested list with sublists delimited by pairs of parentheses, and called *llll* (an initialism for Lisp-like linked list, hinting at the superficial similarity between the two representations). The *llll* can be somehow seen as a simpler and more general alternative to the Max *dictionary* data structure, without the obligation of associating symbolic keys to data. It should be remarked that the root level of an *llll*, as opposed to the Lisp list, is not enclosed by a pair of parentheses: as such, a flat *llll* is essentially equivalent to a regular Max message, thus allowing seamless exchange of data between standard Max objects and *bach* objects [4].

[3] For the sake of completeness, we will mention the fact that the first outlet of *cage.meta.header* should be connected to a *thispatcher* object, necessary to perform some ancillary operations such as programmatically hiding or showing the patcher window, or disposing the patcher itself; and the second outlet reports the patcher instance number, a concept which will be discussed shortly.

when *cage.meta.engine* receives the *autoinstances* message, it will automatically assign an instance number to each note referring to a patcher file, in such a way that instance numbers for each patch are minimized, with the constraint that overlapping notes (according to their own durations plus a 'release time' which can be globally set) referring to the same patcher will always receive different instance numbers — in a similar fashion to what happens with voice assignment in a polyphonic synthesizer. There is one more parameter influencing the choice of the copy of each patcher to be activated for one specific note: the engine name. Each instance of the *cage.meta.engine* module can be assigned an optional name, to be passed as the object box argument (i.e., one can type *cage.meta.engine foobar* in the object box, thus assigning the *foobar* name to *cage.meta.engine*). By assigning them different engine names, more than one *cage.meta* system can run simultaneously in one Max session, without interfering with each other. This naming also allows building nested *cage.meta* systems: a whole *cage.meta* system, composed of a meta-score, a *cage.meta.engine* module and a set of process patches, can be included in a process patch of another, higher-level *cage.meta* system, as long as the names of the *cage.meta.engine* modules are unique.

After the meta-score has been written, generated or loaded from disk, the *load* message can be sent to *cage.meta.engine*: this causes the score to be queried for all its file names and instance numbers, and loads each referred patch as many times as required by the different instance numbers found in the score. Immediately after having being loaded, each patch is initialized, that is, it is sent three identifiers: the engine name, the patcher's own file name, and its instance number. These three identifiers will be used at play time to route the parameters of each note to the correct instance of its referred patch only, while avoiding conflicts with other possible *cage.meta* systems running at the same time, in the same Max session. Furthermore, depending on *bach.roll*'s or *bach.score*'s attributes, markers and tempi can also be sent to all the patches at play time. Receiving notifications for markers can be useful if, for instance, one needs to adapt the behavior of a process to different sections of the score. As a convenient debug tool, selecting a note in the score editor and pressing the 'o' key causes the corresponding referred patch to be opened.

3.4 Playback and rendering

In principle, the outcome of a run of the meta-score is just data, which can be collected in any possible data structure, directed to any possible device or process, and to which any possible meaning can be assigned. Each process, as implemented in the corresponding patcher, receives data from *cage.meta.header* and produces a result which can be routed, for instance, to a MIDI device, or an audio output: but also, according to a less inherently real-time paradigm, to an audio buffer collecting algorithmically-generated audio samples; or to a new score object which will contain the 'rendering' of the meta-score. In particular, we deemed this last scenario to be so typical and peculiar that it deserved some kind of special treatment. More specifically, we expect most musical processes that users may want to implement with the *cage.meta* system to produce either a real-time audio, MIDI or OSC result, or a symbolic result (i.e., a score) to be built incrementally note by note, or chord by chord. As an example of the former case, each *cage.meta.header* patch could contain a synthesizer, generating a complex audio stream depending on the parameters of the associated note; in the latter case, each patch could produce a complex musical figure (e.g., an arpeggio) built according to the parameters of the associated notes, and meant to be transcribed into a final score resulting from the run of the whole meta-score. The latter case can be seen as a special case of the general one, but the complexity of setting up a system for the very basic purpose of generating a score starting from a meta-score prompted us to implement a specific mechanism allowing a process patcher to return to *cage.meta.header* one or more chords in response to a message coming from *cage.meta.header* itself.

More specifically, when the 'playback' attribute of *cage.meta.engine* is set to 1, events coming from the score object are passed to each process patch through the third outlet of *cage.meta.header*, and can be routed to any generic destination (real-time audio, MIDI, OSC, or anything else): for example, the synthesizer implemented in the process patch would set its own parameters according to the data received from the meta-score, activate itself and produce an audio signal to be directly fed to the audio output of Max.

On the other hand, when the 'render' attribute of *cage.meta.engine* is set to 1, events from the score object are passed to each process patch through the fourth and rightmost outlet of *cage.meta.header*, and one or more chords (that is, *llll*s featuring all the chords and note parameters, formatted in the *bach* syntax) can be returned to the second and rightmost inlet of the same *cage.meta.header* module, in a loopback configuration. [4] The *cage.meta.header* module then returns the received chords to its master *cage.meta.engine*, which formats them in such a way to allow an instance of the appropriate object, connected to its second outlet, to be populated with the score being rendered. All this is preceded by a sequence of formatting instructions sent to the destination *bach.roll* or *bach.score*, and generated only if the *render* attribute is on. If only one of the two mechanisms is implemented in the process patches, it is advisable not to activate the other, so as to avoid unnecessary and potentially expensive operations. At the end of the rendering process, the whole rendered score will be contained in the notation object connected to *cage.meta.engine*'s second outlet. So, for example, a patch building an arpeggio receives the parameters of note of the meta-score referring to it (and containing the parameters of the arpeggio, such as starting pitch, range and speed) from the fourth outlet of *cage.meta.header*, and returns the rendered arpeggio, as a sequence of notes, to the rightmost inlet of *cage.meta.header*. The notes of arpeggio are then sent by *cage.meta.header* to the master *cage.meta.engine*, which in turn formats them as messages for the *bach.roll* or *bach.score* object connected to its second outlet. Through this mechanism, this destination *bach.roll* or *bach.score* is incrementally filled with contents and eventually it will contain the result of the whole rendering process.

As a final note, it is possible to have both the *playback* and the *render* attributes set to 1, in which case each event in the meta-score will be routed to both the third and the fourth outlet of the corresponding *cage.meta.header* object, which in turn will expect to receive a rendered sequence of chords in its second inlet.

[4] This kind of loopback configuration between the rightmost inlets and outlets of a module is a common design pattern in *bach* and *cage*, and because of its practical relation with Lisp's lambda functions it is called, conveniently if not entirely accurately, a *lambda loop* [4].

3.5 Running the system

When the 'play' message is sent to the score object, the playback and/or rendering of the meta-score begins: the score object starts outputting the notes it contains according to their very temporality, and *cage.meta.engine* and *cage.meta.header* cooperate to route the data associated to each note to the correct instance of the patcher the note itself refers to. Another possibility is sending the score object the 'play offline' message: in this case, the score object starts outputting the notes it contains in strict temporal order, but with the shortest possible delay between them, without respecting their actual onsets and duration. This is somehow analogous to the 'offline render' command that can be found in virtually any audio and MIDI sequencer. As hinted above, this is useful to trigger non-realtime rendering processes, such as, typically, the rendering of a score through the *lambda loop* of *cage.meta.header*, but also, for instance, the direct writing of samples in an audio buffer, or any other kind of batch operation.

3.6 Final considerations

Everything that was said so far implied that events are represented as data associated to notes of the meta-score, but a slightly different approach is possible. Both *bach.roll* and *bach.score* have a notion of chords as structures of notes sharing the same onset: indeed, single notes appearing in the scores are actually represented as one-note chords, and *bach.score*'s rests are noteless chords; on the other hand, it is possible for two distinct chords in the same voice of a *bach.roll* object to have the same onset. The key point here is that it is possible to choose whether *bach.roll* and *bach.score* should play back the score they contain in a note- or chord-wise fashion: of course, this choice is applicable to both the real-time and offline playback processes. When a single-note chord is encountered, there is no substantial difference between the two modes. A multi-note chord will be output as a sequence of individual messages in the former case, as a single message containing the data for all the concerned notes in the latter case. [5] By setting the playback to chord-wise, it is possible to define processes taking more than one note as their parameters (e.g., an arpeggiator, or a stochastic process based upon a pitch range). In this case, the patcher name and instance number should be assigned to one note per chord: if more than one note has these data assigned, it is undefined which one will be taken into account for initialization and playback.

[5] As a technical note, it should be stressed that, although in most cases the note-wise playback of all the notes of multi-note chord will happen in one Max scheduler tick, there is no guarantee for that, as this depends on the number of same-onset notes, the other processes possibly running, and the user-definable scheduler settings. On the other hand, the same behavior would apply to the playback of multiple chords all having the same onset. So, the only reliable way of knowing at play time how notes are grouped into chords is by playing back the score chord-wise.

It should also be remarked that each item (that is, each note, chord, measure, or voice) in a *bach.roll* or *bach.score* object can be individually muted or soloed. Through this feature, it is possible to control which events of a meta-score will be actually played back or rendered, thus making it easy to build partial, or alternative, versions of the same score. Moreover, although the *cage.meta* system has been designed with a musical usage in mind, its potential is actually wider: for example, it is easy to imagine it applied to generative video, or complex DMX lighting scenarios, or the automation of physical actuators, or, in general, any device or system implemented in, or controlled by, Max.

As a final note, this system also has some shortcomings. First of all, it relies on the synchronicity of Max *send* and *receive* objects across different patchers, which is not guaranteed if the 'Scheduler in Audio Interrupt' option of Max is active. Also, the playback fully relies on the sequencing system of *bach* editors, which output note data (including all slots) at once whenever the playhead reaches a note onset. Due to this, it is impossible to modify the parameters of a given note while it is playing, as modifications are taken into account only if they happen before the playhead reaches the note onset. For the same reason, scrubbing is not possible. Also, that there is no support for the *transport* system of Max. On the other hand, any sequencing feature supported by the *bach* objects (such as looping, jumping to an arbitrary position, and changing the play rate via a *setclock* object) will be perfectly usable within this system.

4. RECREATING *Vortex Temporum*

As a case study for the *cage.meta* system, we decided to recreate the first 81 measures (corresponding to numbers 1 to 20, according to the numbered sections marked in the score) of Gérard Grisey's *Vortex Temporum* in *bach* and *cage*, basing our work upon Jean-Luc Hervé's analysis [2].

The basic idea behind our exercise is abstraction: we aim at building and manipulating a meta-score featuring operative compositional elements, rather than pre-rendered symbolic processes. For instance, since the pitch choices in *Vortex Temporum* are strictly based upon a spectral paradigm, our meta-score will be solely composed of spectral fundamentals. [6] Every note in our meta-score is hence a fundamental for some process, and the indices of harmonics that are built upon it and used by the process are contained in the third slot of each note. We implemented both an off-line score rendering and a real-time rendition, the latter through an extremely simple synthesis process: for this reason, each note carries in a dedicated slot an amplitude envelope information, in the form of a breakpoint function.

Each note of the meta-score triggers one of three different processes: an arpeggio, an accented chord, or a long note. We shall now describe them in detail.

The patch `vt_arpeggiator.maxpat` is designed to generate all the arpeggio-like figures which characterize and identify *Vortex Temporum* from its very beginning. More specifically, the arpeggiator renders all the 16th-

[6] It might be worth pointing out that all harmonic series used by Grisey in the part of *Vortex Temporum* that we reimplemented are stretched linearly in the pitch domain by a factor of $\pi/3$. This observation does not appear in [2], but it seems pertinent in our implementation.

notes figures, with the notable exception of the accented piano and string chords at the beginning of each numbered section: these figures have a different musical role (akin to attack transients), and will be rendered by a different module, which will be discussed further.

Besides the fundamental note and the list of harmonic indices, the arpeggiator also receives some additional content, contained in further slots of our meta-score: the duration of each rendered note in the arpeggio (it is, in our case, constantly 1/16); the number N of notes composing a single arpeggio period (for instance, for flute and clarinet at measure 1 we get $N = 8$, since the arpeggio loops after 8 notes); and the profile for the arpeggio period, as a breakpoint function representing time on the x axis, and the elements in the harmonics list on the y axis. The final point of this function should always coincide with the starting one (to comply with the looping).

Inside the arpeggiator patch, the arpeggio profile is sampled at N uniformly distributed points, each of which is then approximated to the nearest element in the list of the harmonics, which are uniformly distributed on the y axis, independently of their actual value, and subsequently converted into the pitch derived from the stretched harmonic series (see Fig. 1). All pitches are approximated to the quarter-tone grid, with the exception of piano notes, which are approximated to the semitonal grid. [7]

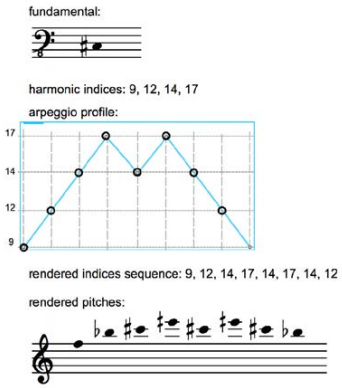

Figure 1. The conversion of the arpeggio profile for the flute (measure 1) into actual pitches. The harmonic indices, relative to the defined fundamental, are uniformly distributed on the y axis, and the profile is uniformly sampled on the x axis. The result is then snapped to the nearest harmonic index. The sequence is then rendered by retrieving the single harmonics of the stretched harmonic series built upon the fundamental.

During real-time playback, harmonics are then output by *bach.drip*, with the appropriate (uniform) time delay between them, depending on the current tempo and the duration of each rendered note. The audio rendering is performed by basic oscillators: the flute is rendered via a slightly overdriven sine tone; the clarinet via a triangular wave; the piano via a rectangular wave. These are of course completely arbitrary choices, only aimed at clearly illustrating the rendering mechanism.

Two accessory processes also need to be described. Firstly, during each of the short numbered sections, Grisey gradually filters out some notes from the arpeggi, replacing more and more of them with rests. This deletion only takes place in the flute and clarinet parts (also enabling the players to breathe between phrases), and it is roughly anticorrelated to dynamics. Since we do not have a model of the exact process employed by Grisey for performing these deletions, we decided to implement it probabilistically, by adding an 'existence probability' for each note, directly correlated to the instantaneous decibel value of the amplitude envelope: at 0dB, the existence probability will be 100%, while at -22dB the probability drops down to 50%. Secondly, starting from number 11, some bichords show up in the piano arpeggi. Bichords are always composed by neighbour harmonics in the harmonics list, thus preserving the arpeggio profile; hence it made sense for us to map a bichord to a half-integer snapping on the y axis of Fig. 1.

The second rendering module, implemented in the patch `vt_hits.maxpat`, deals with the previously mentioned accented chords at the beginning of each section. The offline rendering is trivial: each note is rendered as a chord whose pitches are the specified partials; no additional parameters are needed for this patch. During playback, each note is rendered as a white noise fed into a narrow-band resonant filter centered around the frequency of the note itself, and subsequently artificially reverberated.

The third rendering module, implemented in the patch `vt_tenuti.maxpat`, generates the long viola and cello notes (first appearing at number 3). The viola is rendered via a sawtooth oscillator, the cello via a rectangle wave oscillator.

Figure 2. The first measure of the flute part in the meta-score. The low C♯ (fundamental) and its 9th, 12th, 14th and 17th stretched harmonics are rendered via the arpeggiator (1st instance). The arpeggio loop has 8 notes (each lasting 1/16th) and follows the 'mountain'-like profile shown in light blue. The amplitude envelope is the descending orange line.

In our line of work, the meta-score, that is, all the notes with all their parameters, is completely written 'by hand' in a *bach.score* object. [8] All the slot values are kept visible for each note; tied notes by default share the same slot value and are output as a single longer note. The first measure of the flute part is displayed in Fig. 2, with all metadata visible next to the notehead, as text or breakpoint functions. Depending on our desired operating mode, at any time, we can playback or render symbolically in real-time any portion of score. We can also render the whole meta-score off-line, in order to have it all output form *cage.meta.engine*'s right outlet. A screenshot of the main patch is displayed in Fig. 3.

Although our meta-score might appear bizarre at first (all instruments are notated in F clef, transposed one octave lower), it turns out to be extremely pertinent. For one thing, it is immediately clear that all instruments are somehow playing 'meta-unisons' (except for the right hand of the piano), correspondingly to the fact that all instruments are confined within the same harmonic series. When, at number 10, all fundamentals switch to the G, such important change is immediately readable in our meta-score, while it would take a larger effort of analysis to grab the same information from the score proper. Our meta-score also reveals very clearly the diminished seventh tetrachord (C♯, E, G, B♭) [9] underlying the whole construction, and abstracts the complexity of the arpeggi to a profile shape and a few indices (leaving most of the score made of long tied sequences of 4/4 notes).

5. CONCLUSIONS

Meta-score rewriting operations are a valid analysis tools, allowing significant properties of the musical content to emerge (in our case, harmonic relations and profiles). The fact that each instance of a process is symbolized by a note (or a chord) lets us represent pitch-based operations in a very intuitive way, via their 'most meaningful' pitch (or set of pitches), and shows the pertinence of representing a whole musical process through a single note.

On the other hand, meta-scores have a clear value in musical writing: not only do higher properties emerge, but they can also be easily handled rapidly and efficiently. One can, for instance, change the duration unit of all the notes in the arpeggi from 1/16 to 1/32, or transpose all the clarinet notes down a tritone with a simple Max message (see the *'modify?'* subpatch), or apply standard computer-aided composition techniques on higher-level structures (for instance, it would be fairly easy in our case to retrogradate or stretch all the arpeggio profiles). All these possibilities make *cage.meta* a powerful tool for prototyping, exploring and composing.

Acknowledgments

The *cage* library has been developed by two of the authors within the center of electroacoustic music of the Haute École de Musique in Geneva, supported by the music and arts domain of the scene of the Haute École Specialisée of Western Switzerland.

6. REFERENCES

[1] G. Grisey, *Vortex Temporum*. Milan, Ricordi, 1995.

[2] J. L. Hervé, *Dans le vertige de la durée : Vortex Temporum de Gérard Grisey*. L'Harmattan, L'Itineraire, 2001.

[3] A. Agostini, E. Daubresse, and D. Ghisi, "*cage*: a High-Level Library for Real-Time Computer-Aided Composition," in *Proceedings of the International Computer Music Conference*, Athens, Greece, 2014.

[4] A. Agostini and D. Ghisi, "A Max Library for Musical Notation and Computer-Aided Composition," *Computer Music Journal*, vol. 39, no. 2, pp. 11–27, 2015/10/03 2015. [Online]. Available: http://dx.doi.org/10.1162/COMJ_a_00296

[5] P. Nauert, "Timespan hierarchies and posttonal pitch structure: A composer's strategies," *Perspectives of New Music*, vol. 1, no. 43, pp. 34–53, 2005.

[6] M. Rohrmeier, W. Zudeima, G. A. Wiggins, and C. Scharf, "Principles of structure building in music, language and animal song," *Philosophical transactions of the Royal Society B: Biological sciences. CCCLXX:1664 (March 2015): Biology, cognition and origins of musicality*, vol. CCCLXX:1664, 2015.

[7] C. Agon and G. Assayag, "Programmation Visuelle et Editeurs Musicaux pour la Composition Assiste par Ordinateur," in *Proceedings of the 14th IHM Conference*. Poitiers, France: ACM Computer Press.

[8] L. Goehr, *The Imaginary Museum of Musical Works, An Essay in the Philosophy of Music*. Oxford, Claredon Press, 1992.

[9] N. Goodman, *Languages of Art. An Approach to a Theory of Symbols*. Indianapolis/Cambridge, Hackett Publishing Company, Inc., 1976.

[10] P. Schaeffer, *Traité des objets musicaux, Editions du Seuil*. Paris, 1966.

[11] C. Seeger, "Prescriptive and Descriptive Music-Writing," no. 44 (2), pp. 184–195, 1948.

[12] C. Nash, "The cognitive dimensions of music notations," in *Proceedings of the First International Conference on Technologies for Music Notation and Representation*, Paris, France, 2015.

[13] S. Emmerson, "Combining the acoustic and the digital: music for instruments and computers or prerecorded sound," in *The Oxford Handbook of Computer Music*, roger t. dean ed. New York: Oxford University Press, 2009, pp. 167–190.

[14] A. Sorensen and H. Gardner, "Programming with Time. Cyber-physical programming with Impromptu," in *Proceedings of OOPSLA10 : ACM International Conference on Object Oriented Programming Systems Languages and Applications*. New York: ACM, 2010, pp. 822–834.

[15] M. Mathews, "The Digital Computer as a Musical Instrument," no. 3591, pp. 553–557, 1963.

[16] M. Mathews, F. R. Moore, and J.-C. Risset, "Computers and Future Music," *Science*, vol. 183, no. 4122, 1974.

[7] On occasion, the flute part contains 12-TET tempered notes instead of 24-TET tempered notes. This is the case at measure 19, for instance: the natural C 'should' be a quarter-sharp C according to the harmonic series rendering. Grisey probably adjusted these notes to ease and respect the instrumental technique, but we did not account for these 'manual' adjustments in our work.

[8] By manipulating the schemes given in [2] one might build the meta-score content itself with *bach* via standard algorithmic techniques — this goes, however, beyond the purpose of this paper, which is to exemplify the usage of *cage.meta* via a specific case study.

[9] It should be pointed out that the E does not appear in the portion of score upon which we focused.

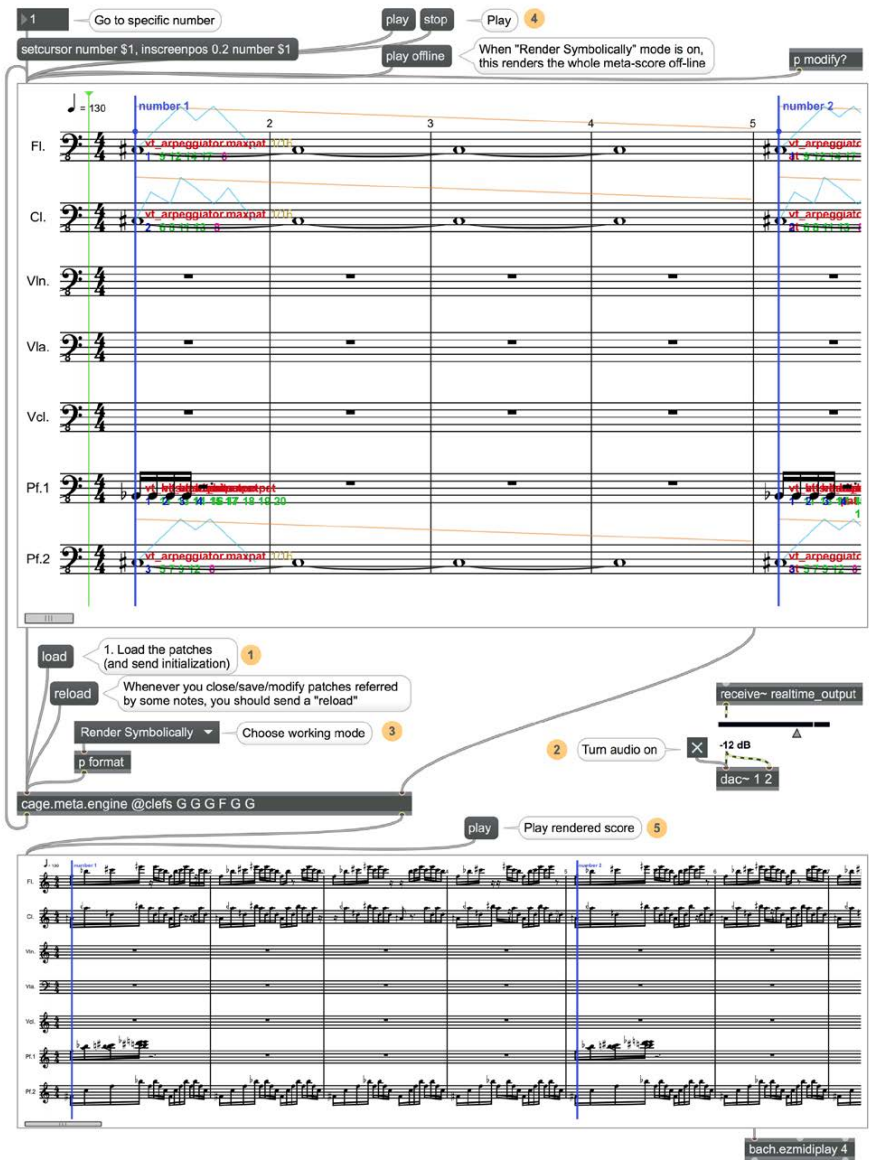

Figure 3. A screenshot of the main patch, showing the first measures of the meta-score modeling *Vortex Temporum*.

Composing in Bohlen–Pierce and Carlos Alpha scales for solo clarinet

Todd Harrop
Hochschule für Musik und Theater Hamburg
tharrop5@gmail.com

Nora-Louise Müller
Hochschule für Musik und Theater Hamburg
welcome@noralouisemuller.de

ABSTRACT

In 2012 we collaborated on a solo work for Bohlen–Pierce (BP) clarinet in both the BP scale and Carlos alpha scale. Neither has a 1200 cent octave, however they share an interval of 1170 cents which we attempted to use as a substitute for motivic transposition. Some computer code assisted us during the creation period in managing up to five staves for one line of music: sounding pitch, MIDI keyboard notation for the composer in both BP and alpha, and a clarinet fingering notation for the performer in both BP and alpha. Although there are programs today that can interactively handle microtonal notation, e.g., MaxScore and the Bach library for Max/MSP, we show how a computer can assist composers in navigating poly-microtonal scales or, for advanced composer-theorists, to interpret equal-tempered scales as just intonation frequency ratios situated in a harmonic lattice. This project was unorthodox for the following reasons: playing two microtonal scales on one clarinet, appropriating a quasi-octave as interval of equivalency, and composing with non-octave scales.

1. INTRODUCTION

When we noticed that two microtonal, non-octave scales shared the same interval of 1170 cents, about a 1/6th-tone shy of an octave, we decided to collaborate on a musical work for an acoustic instrument able to play both scales. *Bird of Janus,* for solo Bohlen–Pierce soprano clarinet, was composed in 2012 during a residency at the Banff Centre for the Arts, Canada. Through the use of alternate fingerings a convincing Carlos alpha scale was playable on this same clarinet. In order to address melodic, harmonic and notational challenges various simple utilities were coded in `Max/MSP` and `Matlab` to assist in the pre-compositional work.

We were already experienced with BP tuning and repertoire, especially after participating in the first Bohlen–Pierce symposium (Boston 2010) where twenty lectures and forty compositions were presented. For this collaboration we posed a few new questions and attempted to answer or at least address them through artistic research, i.e. by the creation and explanation of an original composition. We wanted to test (1) if an interval short of an octave by about a 1/6th-tone could be a substitute, (2) if the Carlos alpha scale could act as a kind of 1/4-tone scale to the BP scale, (3) if alpha could be performed on a BP clarinet, and (4) how would the composer and performer handle its notation? Some of these problems were tackled by computer-assisted composition. The final piece grew out of a sketch which was initially composed algorithmically, described in section 3.2.

Our composition was premiered in Toronto and performed in Montreal, Hamburg, Berlin and recently at the funeral of Heinz Bohlen, one of the BP scale's progenitors; hence we believe the music is artistically successful considering its unusual demands. The quasi-octave is noticeably short but with careful handling it can be convincing in melodic contexts. Since the scales have radically different just intonation interpretations they do not comfortably coalesce in a harmonic framework. This is an area worth further investigation. Our paper concludes with a short list of other poly-tonal compositions albeit in BP and conventional tunings.

Figure 1. Bohlen–Pierce soprano clarinet by Stephen Fox (Toronto, 2011). Owned by Nora-Louise Müller, Germany. Photo by Detlev Müller, 2016. Detail of custom keywork.

2. INSTRUMENT AND SCALES

This project would not have been possible with either a B♭ clarinet or a quarter-tone clarinet since the two scales described in 2.2 have few notes in common with 12 or 24 divisions of the octave. Additionally, our BP clarinet was customized and is able to play pitches which other BP clarinets cannot.