Improvising Formalisation — Conversational Programming and Live Coding

Julian Rohrhuber and Alberto de Campo

Abstract

In the present article we aim to contribute to the discussion of new computational paradigms for computer music from the specific angle of conversational sound programming. Over the last decade, sound programming has seen a turn toward including the programming activity into the dynamic unfolding of sound, both within performance (most strongly in *live coding*) and in sound research. If we understand programming as a method of reflection on the possible, as a procedure of interpreting the rules of reformulation, this turn changes the relation between formalisation and process. Rather than one serving the other, formalisation becomes audible as a dialectical interplay between plan and action. Where a given plan can lead us is at least partially unforeseen (unless the plan is trivial), and consequently formalisation is by necessity improvisational in nature.

The idea of (re-)programming music or sound programs while they run introduces a number of constraints, which suggest creating extensions to programming languages that support these activities. On the other hand, it opens new possibilities for communication, both between partners (such as co-musicians, or co-researchers in other contexts) and audiences. A particular approach developed by the authors chooses to introduce even more constraints than live coding in general, and leverages some features of the programming language of choice for even more artistic and communicative possibilities.

1 Background

Sound is a peculiar modality for at least two simple reasons. First, it is experienced exclusively as a change over time; its properties are to be found only in its dynamic unfolding. Stillness is silent, as is constant linear movement; only a change of movement is audible. Second, we usually interpret sound as the effect of an event in the world, and relate it to some physical source which we look for as a plausible cause for its transmission through space. This is for instance why a vinyl record can be both a frozen window through which we hear past physical events, and at the same time a physical structure which can be scratched with a needle to make sounds. Electronic music, however immaterial it may appear, is usually taken to originate from electrons moving in circuits, and from the interference through the human intervention by turning knobs, plugging, bending, touching, playing. The fact that causality of sound remains enmeshed in the situation makes it call for improvisation in real time.

1.1 Real time and conversational programming

The moment when it became possible to run a sufficiently complex sound program efficiently enough on affordable computers to be able to change its parameters at runtime was a big step which immediately extended its domain [10] —the computer became a viable instrument for improvised music. Just as in experimenting with electrical circuits, parameters are the channels for real time intervention. The sound source being essentially specified by the sound synthesis algorithms, such improvisation plays with the consequences of a formal description of a process (the electrical circuit or its numerical simulation) and its control inputs. Yet here, the programming language serves mainly as a tool for creating the blueprint of a machine, which, after the program that 'is' the instrument has been correctly written, becomes involved in the acoustic situation. The formal script is only the static preliminary to the dynamic presence of a sound machine. Formalisation itself, having been a great topic for algorithmic composition, shifts out of focus.

This notion of real time, predominant in the 1990s, coincided with the tendency in mainstream computer science to regard programming languages first and foremost as tools for building applications. Approaches of interactive programming coexisted, but remained largely confined to educational and 'theoretical' contexts. However, as soon as computers became capable of compiling higher level sound programs that run in real time, it also became possible to reconnect to these older conversational approaches. This possibility was not self-evident at first; for instance, the moment when the author of the SuperCollider language decided to keep the interpreter running during sound synthesis was not specifically mentioned in the release notes, even though he was certainly aware of the new possibilities.

Generally speaking, from this moment, the read-eval-print-loop became more than just a way of either specifying a score for the parameters of a synthesis machine, or the testing ground for scripts in building an application or finished score; it rather became the basis for improvised sound programming. Over the last decade, such shifts have led to the development of a broader movement of experimental sound programming.[1]

1.2 Ideal Types of Musical Objects and Processes

> Milhaud: "And even sometimes, as Mr. Duchamp said—and I agree completely—the work guides you. Often a creator, in another work, contradicts himself completely. And thanks be to God! Otherwise, he remains under one label. But he is led, not only by his thought, but by—call it what you like, inspiration, if you are not afraid of such a word. There is the work that guides you too of course, but if you don't have a responsive technique, then it begins to get pretty bad."
>
> Ritchie: "Then, in other words, Mr. Milhaud, the work might run away with itself."

[1]It was not before the first publications and conferences on live coding, such as *changing grammars* and the foundation of TOPLAP in 2004, that such undercurrents received wider attention. As terminology, we find these paradigms under different names, such as "just in time programming", "explorative programming", "on-the-fly programming". For musical performance, the more specific term "live coding" has become widely accepted. The terms of conversational programming and interactive programming fully retain their more general applicability.

Milhaud: “Certainly it might. And so much the better, because if it runs away, probably it should not have been started in the first place.”

Duchamp: “It is a kind of race between the artist and the work of art.”

(MacAgy, *The Western Round Table on Modern Art* [3])

Instrument and Composition

Let us consider for a moment the setting of a traditional musical performance on stage. Even when the sound is amplified, it is always related to the point where it is called forth. This point of induction can be thought as between performer and the instrument. In this context, the instrument is a predictable, deterministic machine, allowing a performer with highly sophisticated motor skills to elicit very precise responses (e.g., a piano) —thus translating physically mediated performer decisions into artistically meaningful sound. The instrument is an ideal as well as conventional type of musical object, which as metaphor underlies many sound programming architectures.

Tracing back the chain of causation, the written score, a proxy for the composer’s artistic intentions, comes into view. In electroacoustic music (foreshadowed by older music machines and automation in the arts), the composition has moved toward the inclusion of the interpretation process. Composition in the tradition of electroacoustic music becomes a fully realized version of a larger-scale musical structure—all decisions are made before the performance, and the piece can only be played back as is, optionally with spatial diffusion as an additional layer of interpretation. Complementing the notion of a sound-generating program as an instrument, this form of composition corresponds to the notion of an algorithm as a complete and closed entity.

Improvisation and Formalisation

From this perspective, it seems intuitive that improvisation and formalisation are separate things: Either we decide beforehand, composing a future to be, or we react within streams of change now. Separating thought and action, these ideal extremes have their benefits—they structure time by delegating complex actions to bodily or technical automatisms.[2] While they might be mixed in the process of composing, and in practising on an instrument, they are separated as far as their aim is concerned. As a result, what is considered difficult and what counts as easy is bound to this separation of concerns.

Yet, we have to acknowledge that within programming languages, as well as within musical practice, these separations have been systematically shifted, and not only recently. Improvisation is inspired by rules and constraints, and the notion of composition has increasingly become a hybrid between automatised rule systems and explicit schemes such as game rules, protocols and contracts. This comes as no surprise, because in a sense, improvisation is deeply entangled reflection on prearrangement and algorithm. In programming languages, the temporal separation between end use and development has for long been systematically blurred.[3] Still, intuitively, program text is taken as static while the active program is represented by other graphical elements following the machine

[2]The opposing concepts of ‘abstract time’ and ‘lived time’ have a longstanding tradition in modern thought. Bergson’s *temps* vs. *durée* is one important example [2].

[3]e.g., in Lisp, Smalltalk, Forth, and other languages.

or instrument metaphor. But when what was a 'tool-bench' for manufacturing programs becomes the program itself, it is necessary to mix these levels.

From an epistemological point of view we can consider a program as a measuring device; the sliders, knobs and buttons become measured parameters (usually with floating point representations of real numbers or integer representations of natural numbers). The program itself can then be seen as the solidified theory, the underlying assumptions behind the observational process. While standardisation of measurement is one essential condition for objective observation and conversation, the modification of the experimental setting is equally necessary [12]. In research (be it artistic or scientific), formalisation is implicated in the experimental situation [11, 7]. We search as much for the right questions as for the right anwers, we search as much for the right means as for the right end. So long as the problem is non-trivial, what is outside and what is inside can only be decided in the process of development.

Therefore the construction of an experimental environment has to allow processes in time to be intercalated in the process of their iterative formalisation, which has consequences regarding the underlying concepts and paradigms of conversational programming and live coding: now we have to take into account the whole situation, with its entanglement in aesthetics, conversation and inspiration. Because the necessary combination of timeless algorithm and algorithmic process impedes phantasms of immediate total control, the common notion of the virtuoso as the prototypical artist becomes but one possibility in a broader field. Neither the machine needs to be virtuosic, nor the human performer. Finally, in the domain of situated algorithms, constraints are an immanent part of improvised interventions, not only their precondition [15].

2 Constraints and Extensions

The aim to program programs while they run introduces constraints. One type of constraint comes from the simple requirement of making provision for meaningful intervention in the static representation of the program, and in its runtime equivalent at the same time. Other constraints similarly have to do with time passing, but concern the programming situation: the limited time available for typing, and the cognitive complexity of the algorithms. These constraints call for what can be called *creative extensions* [16], which are discussed below in more detail within one particular context, the *Just In Time Library* in the programming language and sound synthesis environment *SuperCollider*.

2.1 Complexity

Algorithms specify both processes and how to change them. Algorithmic sound is so interesting because its source is a formalism. Listening to improvised formalisation[4] is interesting, because one can never be sure whether a change came from the algorithmic process or from the performer's intervention [13]. Very complex algorithms may create compelling behaviour, but they may not be the right choice for conversational approaches. On the one hand, instead of causing an irritating ambiguity between automatism and intervention, they form a quagmire in which changes in code seem unrelated to the unfolding sound. Both for a performer as well as for the audience, the intellectual and

[4]For recorded sound examples, see *A prehistory of live coding.* CD released by TOPLAP, 2007.

sensual challenge to grasp algorithmic sound is enhanced by a formalisation that reveals sonically incisive modifications. A discussion by Brown and Sorensen (who are also probably the most experienced live coding duo, *aa-cell*) emphasises that changing complex algorithms (such as neural nets and agent-based systems) meaningfully may take too long and introduce too much risk of errors. On the other hand, descriptions of processes can be too compact: then they allow fewer interesting points of change than a longer, but more openly phrased script [4]. Clearly, the double readability of programs (readable for machine and human interpreters), turns out to be intensified in the situation of conversational programming. By consequence, computational complexity as well as semantic complexity are limited and intertwined.

2.2 Time again

In a conversation, working out answers over longer times while a conversation partner is waiting is at least uncomfortable, if not impolite. Discussing research questions with a partner, then writing little programs that experiment with the question may take somewhat longer times. If the partner also takes part in formalising aspects of the question as a program, there will be no perceived waiting time; if s/he is not, a limit of patience will be reached sooner. In a traditional musical performance setting, time is even more constrained: the musical style chosen may require transitions at specific time points, and missing them will sound ‘wrong’.

Transitions in overall direction can be even more difficult. While experienced improvising acoustic musicians tend to be able to change direction in a split-second, a programming musician (live coder) either needs readily accessible prepared material to go to (which can be less than satisfying for being ‘too prepared’), or must decide to introduce a change in the future, prepare for it, and introduce the change when preparation is done. While this may only take some seconds, it may be too late for being part of a musical conversation. Nevertheless, the tension within improvised music, both as a process of finding interesting rules and as a process of immediate reactions opens a wide range of possible approaches.

2.3 Criteria

While it is notoriously impossible to find valid criteria for the artistic value of any artifact (including algorithms), pragmatically considering which criteria might help to find types of algorithms that seem useful and flexible in live coding contexts is helpful. Brown and Sorensen propose the following criteria for generative processes that in their experience lend themselves well to live coding:

> “Generative processes should be:
>
> - succinct and quick to type;
> - widely applicable to a variety of musical circumstances;
> - computationally efficient, allowing real-time evaluation;
> - responsive and adaptive, minimising future scheduling commitments;
> - and modifiable through the exposure of appropriate parameters.”

Succinctness would depend on programming infrastructure that allows very terse expression of the algorithms in question, and depends on the underlying domain; wide applicability may simply mean avoiding culture—or style—specific approaches; appropriate parameters could be generalized to exposing details that allow interesting changes of direction. Our suggested additions to this list would include:

- clear in communicating the algorithm's behavior (so it can be changed);
- easy to reformulate (rephrase) for variations;
- handling errors gracefully, possibly integrating errors;
- as free of administrative coding as possible.

This list could easily be extended taking into account different domains. More generally, live coding systems should be open to different kinds of communication, and help the improvisers remember and understand the overall state of the system and its structure of causation.

To see how one implementation of a live coding-suitable environment addresses these preliminary criteria, and in particular, which creative extensions it introduces to the basic language it is based on, we turn to the *Just In Time Library (JITLib)*, an extension of the SuperCollider3 language and system.

3 Just In Time Library

The SuperCollider language takes its name from the idea of combining audio DSP (synthesis and processing) and high-level symbolic descriptions of musical events (patterns) in a single language [8]. It modularises DSP algorithms into unit generators, which are represented by the language as functions that take signal processes as arguments. The resulting graph specifies a type of *synth*, which is not so much an instrument, but rather a lightweight abstract sound object or sound event, which may be very short and can be created efficiently at high rates. Synths may communicate via a system of busses. The stateless ugen graph is assembled by the object oriented SuperCollider language, which supports multiple programming paradigms. It also provides a library of *patterns*, which is a stateless functional system describing streams of events, some of which may be sound events. Such *streams* (routines, tasks) may run concurrently, sharing a description, either formulated in terms of patterns, or by any other language construct (such as imperative statements).

3.1 Proxies

Toward the end of the 1990s, it became evident that what is required for just in time sound programming is a combination of timeless functional structures and imperative modifications to the latter. The JITLib, which is part of the SuperCollider language, developed this idea by introducing abstract placeholders for modification (*proxies*), which add an abstract notion of state into the stateless graphs of patterns and unit generators.

Essentially, they simply add the following concepts:

- abstract from the order of execution, so that preparation and action are indistinguishable;
- provide a local context which specifies how and when modifications affect running processes;
- maintain perceptual continuity;
- provide simple schemes of access.

Syntactically, a definition of a new proxy is indistinguishable for an observer from a reference to it. Modifying the specification of a proxy is as similar as possible to its embedding in other processes:

```
Ndef(\x).play; // listen to an as yet undefined sound proxy named \x
Ndef(\x, { SinOsc.ar(Ndef(\y).ar * 200 + 300) });//...specify the sound x
Ndef(\y, Ndef(\x) * 2.4); // and specify what it depends on (y)
```

Proxies for Synths - Ndef

The class NodeProxy is a proxy for continuous synth processes, allowing to easily combine a large number of interdependent graphs while maintaining perceptual continuity (crossfade and sync). In terms of implementation, it isolates from resources such as signal busses. Node proxies (accessed e.g., by Ndef and a unique name) play in the background, so that every node in the system may equally become a source for playback, and may be mapped to an arbitrary number of audio channels. The control parameters and interconnections are maintained across code modifications, and supported with GUIs for overview (what is playing, mixing levels) and tuning sound parameters in detail (ProxyMixer).

Proxies for Tasks - Tdef

Tasks may represent any programming intervention by imperative or functional statements in a coroutine, and is therefore the most general way to do things unfolding in time (structured waiting). A TaskProxy / Tdef provides an environment (internal name space) for local state that may either be shared between parallel instantiations, or may remain independent. Changes are timed according to a synchronisation scheme that can be meter-based, or specific conditions. Again, an overview GUI for all named TaskProxies (Tdefs) is provided, which allows retrieving all TaskProxy source code for rewriting.

```
(
Tdef(\a, { |env|
 10.do {|i| i.postln; (freq: i * 200 + 300).play; 0.5.wait};
}).play;
)
(
Tdef(\a).set(\dt, 0.2);
Tdef(\a, { |env|
 10.do {|i| i.postln; (freq: i * 200 + 300 + 500.rand).play; env.dt.wait};
});
)
```

Proxies for Patterns - Pdef

SuperCollider includes an extensive library of patterns, abstract specifications for processes; A PatternProxy / Pdef represents a point of modification in the stateless graph of patterns that implicitly propagates changes in the specification to the multiple processes. It also encapsulates a player for independent starting, stopping and runtime modifications a single stream. Like Tdef, it has a local environment and a mechanism for timed modification.

```
(
Pdef(\g, Pbind(
   \instrument, \grain,
   \freq, Pexprand(200, 2000),
   \sustain, 0.003,
   \dur, Pgeom(0.01, 1.1, 30)
)).play;
)
```

3.2 Extended Conversation

Networked code exchange, distributed sound

Sometimes circumstances suggest new approaches, just because they facilitate pursuing an idea. SuperCollider3 consists of a client which runs the interpreter, and does realtime scheduling of events, and a separate server which does synthesis and processing [9]. They communicate via the network protocol OpenSoundControl, which immediately allows running client and server on separate machines; but also other topologies become accessible with little effort, e.g., distributing synthesis commands over a list of servers running on different laptops becomes a simple form of spatialisation.

```
(
Pdef(\g, Pbind(
   \freq, Pexprand(200, 2000),
   \dur, 0.1,
   \where, Pwhite(0, 6) // maps to index in a list of servers.
)).play;
)
```

But why only distribute the sounds themselves? One can also send the code that was just evaluated to the clients on all the laptops, and thus communicate by sounds and their algorithmic descriptions. Because proxy assignments are atomic, they can be introduced into a different program context by a conversation partner. Because they represent changes to possibly multiple processes, their effect propagates into various threads of the running system.

Performance History

The recent past is an interesting resource for performances; in improvised acoustic music, going back to recent material is a common way to create longer evolving structures. In live coding, the simplest form would be straight repetition by re-starting previously used patterns, which is rather limiting; it would be better to be able to retrieve past elements as code and adapt them for the present moment. As SuperCollider is based on Smalltalk, it supports the concept of reflection, which extends to the interpreter. The JITLib class History can obtain the last evaluated string and keeps a log of all the code strings evaluated, their time of evaluation, and a name tag. It can actually record history when it occurs in setups with multiple networked clients, with proper code attribution by name. Going back in History and searching by name tags or substrings allows quite complex joint motivic development by realtime open source code exchange.

Shouting

Writing code can absorb one's attention very much, and it can be difficult to attract each other's attention in group live coding. The pragmatic solution we developed is visual shouting: comments starting with a special tags get displayed in very large font on top of all other windows.

```
// a loud message begins with prefix //!!
//!! Get noisier and land in ca 2 min?
```

Deliberate Constraints

The ensemble *powerbooks unplugged*[5] has emerged from a workshop exploring joint conversational programming of granular synthesis variants in networks. It can well serve as an example for the strategy of adopting deliberate constraints in order to explore the remaining possibilities in more depth.

In this ensemble, we attain very localised sound diffusion, as we use only the laptop and its built-in speakers. The resonances of the laptop box create a very physical presence of the sound source. The audio quality also is quite special, with little activity below 250 Hz, early distortion, and rather low maximum level. As a simplification, we also use almost exclusively sounds that do not require accessing control parameters while they run—with granular textures, the individual sound is long gone before one can change a control value; and by extension, layers of longer sounds can also be constructed algorithmically without individual control access.

Because the individual programming activity is decoupled from the sound source —every process may result in sound on every computer, yet the sound sources are brought as close as possible to the programming activity, there is a paradoxical ambiguous agency, a distributed personhood typical for network music [14].

3.3 Artistic conversational programming strategies

Nick Collins explores the notion of live coding as musical practice in [6], including lists of possible exercises to improve live coding skills. As an alternative, we propose a number of artistic strategies that can be experimented with in conversational programming situations; the aim here is rather to attain mental flexibility in moving in the very large possibility space of which iterative transformations of existing code may produce interesting changes in the unfolding sound.

[5] See http://pbup.goto10.org

Some strategies could be:

destructive / analytical programming – Remove more and more parts of an existing script to explore what each removed part contributed to the whole process. Generally speaking, conversational programming can be a useful tool for learning how existing code works.

proxy refactoring – Reorganize an existing script: if it is a single proxy, turn its components into individual proxies, so they are exposed for independent modification. If the script consists of multiple small proxies, merge them into fewer large proxies. Because of a different structure of dependency, in each stage, modifications to the parts result in different sonic shifts.

time-warping – Use a single parameter representing time (or count) to parameterise various processes, which may either be stateless functions or stateful routines. By consequence, each process is a direct function of time.

switchboard programming – Start multiple continuous processes, then control which ones become audible by switching algorithms.

parameter modification – Start a continuous process, then change each of its parameters at least once.

mixed mode parameter tweaking – Switch between parameter modification and the rewriting of the sound processes they control. The same parameter may mean something different in a revision. Tasks or external input may continually vary the parameters, while their interconnection is rewritten.

purloined letters – Choose a script by a co-musician and modify it; multiple players can decide to create swarms of parallel variations of the same starting codelet. The network music band *the hub* used to play the piece *borrowing and stealing*, which is a precursor of this type of shared motivic development.

topic exploration – Agree on a topic to explore, and create sounds around that center.

tandem programming – In a style reminiscent of terminal programming, rewrite one single sound process over the network. Alternatively, multiple participants may program variations of a single input stream.[6]

conversational sonification – Choose some data source as a topic for sonification; try creating different sonic representations for the same data source, while discussing domain and the epistemological implications.

chat model – Terminal conversation, iterative refinement/development by dialogue with program. This can cross the border to multi user dungeons (cf. Craig Latta's *Quoth*).

ex nihilo - blank text editor/canvas – Begin with emptiness and add things in the smallest possible steps. In multi-player situations, begin with an empty document and write code to thin out the sounds of the entire ensemble.

[6]Dan Stowell's mass radio coding used this approach.

4 Instead of a Conclusion: some observations on conversational programming

- The distance between what one may expect an algorithm to have as effect and what the algorithm actually brings about is irreducible; this gap is the thread along which conversational sound programming is kept moving.
- Differential hearing is a condition for every intervention in sound; a system must provide perceptual continuity to allow for it.
- The syntax of a language gives access to possible points of intervention. Recursion is essential [1].
- A program is doubly readable, by machine and other humans. Trying to understand an algorithms changes how we hear the sound.
- The unforeseen requires expectation, even chance needs a wager. Therefore semantic fiction is as necessary as syntactical correctness.
- The more interesting the unfolding of a given algorithm is, the more time may pass before it becomes a burden and we feel an urge to change it. The more revealing and interesting the code is, the quicker we know what would be an appropriate modification.
- Because a sound event is not reducible to the sound wave, but implies both the perception and the situation where it happens, programming languages for sound become a true mathematics of sound. This mathematics of sound is a creole.
- There is no such thing as an immediate access to a sound event—it can be conceived only in relation to its unfolding over time. Each programming language gives a specific access to this unfolding, and only by this constraint is any intervention possible.
- Live coding on stage is drawn between the necessity between trying to convey what is happening (and what is causing what sound) and the abstract quality of the domain sound itself.
- Live coding: What the performer does and what the algorithm is ambiguous. Lured by this question, the observer is drawn into a game of possible causes, and as a result the error becomes an ally of irritation. In a different but analogous way, the programmer becomes an observer. In a state of irritainment, both become interpassive.

Bibliography

[1] Assous R., “Le réel est-il récursif?” In *Le feedback dans la creation musicale*, Lyon 17-18 mars 2006.

[2] Bergson H., *Matter and Memory*, Zone Books, U.S., 1991.

[3] MacAgy D., *The Western Round Table on Modern Art*, transcript, 1949.

[4] Brown A. R. and Sorensen A., “Interacting with Generative Music through Live Coding”, *Contemporary Music Review*, 28:1, pp. 17-29, 2009.

[5] Collins N., McLean A., Rohrhuber J. and Ward A., “Live Coding Techniques for Laptop Performance”, *Organised Sound* 8(3), pp. 321-30, 2003.

[6] Collins N., “Live Coding Practice”, *Proceedings of NIME 2007*, New York, 2007.

[7] de Campo A., *Science by Ear. An interdisciplinary approach to sonifying scientific data*, Ph.D. thesis, University for Music and Dramatic Arts Graz, Graz, Austria, 2009.

[8] McCartney J., “SuperCollider: A new real-time sound synthesis language”. In *Proceedings of ICMC 1996*, pp. 257-258. San Francisco: International Computer Music Association.

[9] McCartney J. “Rethinking the computer music language: SuperCollider”. *Computer Music Journal*, 26(4), pp. 61-68, 2002.

[10] McCartney J., “A Few Quick Notes on Opportunities and Pitfalls of the Application of Computers in Art and Music”, *Proceedings of Ars Electronica*, 2003.

[11] Rheinberger H.-J., *Toward a History of Epistemic Things: Synthesizing Proteins in the Test Tube (Writing Science)*, Stanford University Press, Stanford, California, 1997.

[12] Rohrhuber J., “Das Rechtzeitige. Doppelte Extension und formales Experiment”. In Axel Volmar (ed.), *Zeitkritische Medienprozesse*, Kadmos, Berlin, 2008.

[13] Rohrhuber J. and de Campo A., “Waiting and Uncertainty in Computer Music Networks”, *Proceedings of ICMC*, 2004.

[14] Rohrhuber J., de Campo A., Wieser R., Van Kampen J. K., Echo H. and Hölzl H., “Purloined letters and distributed persons”. In *Music in the Global Village Conference*, Budapest, December 2007.

[15] Rohrhuber J., Hall T. and de Campo A., “Dialects, Constraints, Systems within Systems”. In *The SuperCollider Book*, Nick Collins, Scott Wilson and David Cottle (Eds.), MIT Press, 2009 (forthcoming).

[16] Van Roy P. “Programming Paradigms for Dummies: What Every Programmer Should Know”, *New Computational Paradigms for Computer Music*, Assayag G. and Gerzso A. (eds.), IRCAM/Delatour France, 2009.