ALGORITHMS TODAY

NOTES ON LANGUAGE DESIGN FOR JUST IN TIME PROGRAMMING

Julian Rohrhuber
University Cologne, Academy of Media Arts Cologne

Alberto de Campo
Institute for Electronic Music and Acoustics, University for Music and Dramatic Arts Graz

Renate Wieser
University for Fine Arts Hamburg

ABSTRACT

Modifying the code of a program at runtime has been made possible by quite a number of programming languages, such as *Scheme, Objective-C, Smalltalk, Self*, and others. Be it for allowing different development and prototyping methods, or be it for performative code improvisation (*live coding* [1]), these languages have gained a wider use in algorithmic sound synthesis.[2] Consequently, the temporal delimitation between development (preparation) and application (action) has become less rigid - even more, instead of making only the parameters of an application accessible, its source code can be modified at runtime. It is this more immediate interrelation between changes in the code and changes in the sound that brings the text of the computer language into play more evidently as a direct description of sound.
This article presents an overview of such a system implemented in a dynamic programming language and discusses some implications and problems of this approach. It concludes with examples of interactive programming in sonification research, in film sound and music performance, areas where this approach is used with interesting results.

1. INTRODUCTION

Algorithmic sound sources, being the actualization of rules, often seem to have their beauty in the more or less indirect perceptibility of these artificial causal relationships. Programming languages allow the formulation of such algorithms, not only for the computer to actualize them, but at the same time, to maintain a discourse with a model, a portrait of some world with its own rules.
But if we ask how this formulation takes place, it may, quite similar to literary text production, turn out not to be simply the notation of pre-formed ideas or intentions. Despite being deterministic we often can't be entirely sure what sound an algorithm will yield. In many cases its complete anticipation, precluding anything new from happening, would not be all that desirable. Suppose we search for a specific sound, say the creaking of a closing bus door heard from the other side of the road. We would begin by writing an initial algorithm that captures a rough imagination, a conjecture of how the sound could be characterized. Then we would modify this description until it became, possibly in a surprising moment, a sudden realization of something that evokes a memory of that particular sound. The surprising moment is not so much the result of a random coincidence, but of the way in which program-text, synthesis process, sound and perception interact.[3]

2. TIME TROUBLES

"Back in the 1920s, the nuclear physicist Niels Bohr said, "Predictions are hard, especially if they concern the future." Of course that's still true today."[4]

A program obviously is a plan of how something is supposed to happen, an anticipation of future events. If the program text is used as the representation of algorithmic processes with their causal relations, one encounters the problem that the process is happening in time while its description has been made in advance. This becomes apparent as soon as one tries to change the plan when it is already in the process of realization. It is interesting that it is not so much its predictive quality that makes this difficult than the fact that the algorithm, as it happens, operates on its own past states. Its iterative[5] character causes the algorithm to stick to its own history, so that as a process, it is always something else than its rules.
Even if it is described in a declarative way, the programmer's (and the sound's) *"temporal existence [...] imposes state on the system."*[6] But even more, for the same reason it is also difficult to relate two processes

[1] See e.g. *"temporary organisation for the proliferation of live algorithmic programming"* (toplap), http://toplap.org, with references to developers and artists like Fredrik Olofsson, Nick Collins, Ge Wang, Dave Griffiths, Craig Latta, Amy Alexander, Adrian Ward, Alex McLean

[2] See e.g. Collins, McLean, Rohrhuber, Ward 2004.

[3] Here, we try to describe the implications of interactive programming not so much as a problem of control, but emphasize the involvement in a process of distributed agency. A very good general description of the interaction between human and non-human agents (actants) can be found in Latour 1996a [8]

[4] *New World* talking to Claus Weyrich, head of Corporate Technology at Siemens)

[5] Derrida notes that the sanskrit root of the word *'iter'* is *'itara'*, *'otherness'*, which connects otherness with repetition.

[6] While Abelson and Sussman were talking about the *"user"* imposing time on the system, it is the programmer and the sound generation in our context. [1], p.291

to each other. When one replaces a running algorithm by a new one, the new development might be similar to the old, but it is nevertheless a new enfolding which, by itself, has no memory of the previous one. This results in a situation where there is no general equivalence between the present and a modified version of an algorithm.

Interactive programs that provide a graphical interface to control their behaviour seem to avoid this rigid character. But, quite obviously, this type of immediate control happens within the space that the programmer has chosen before. In this sense, a graphical user interface leads us to the same basic problem: The algorithm implicitly presumes a fixed delimitation between what can be changed at runtime and what is the prevailing context of this change. In a language for interactive programming, this delimitation can be made explicit (syntactically) as well as dynamic (semantically), forming a porous signification space of what is part of the operation and what is its parameters.

Whatever part of the process is exchanged at runtime, it is always a new part in an ongoing context (or, taken the other way round, a changed context of an ongoing part). Defining (and modifying) this structure therefore should be seen primarily as working on a temporal delimitation creating „islands in history“[7]. It will become clear that there are different possible ways of interaction between these islands and their contexts, depending on the situation.

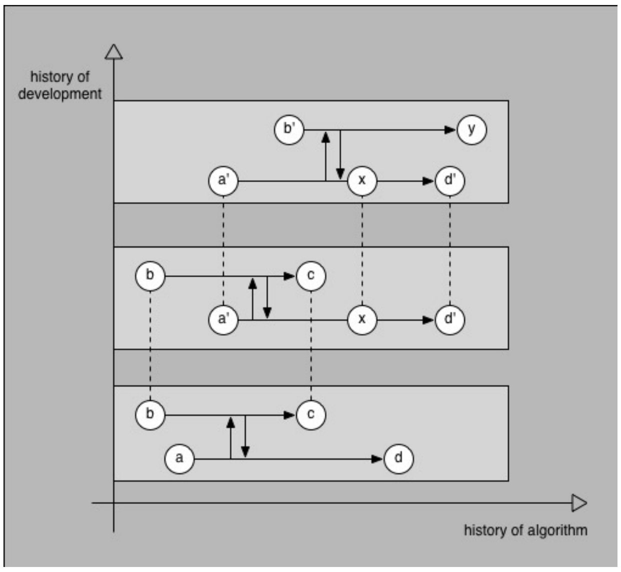

Figure 1: *When modularizing the algorithm, a part can be exchanged without changing the temporal development of the other. Here, a pair of algorithms (visually connected by grey zones) is developed together: in a first step, x is inserted in (ad), while keeping (bc) the same. In a second step, (bc) is changed to (b'y), while keeping (a'xd').*

To recapitulate, we can see that working on an active program means being involved in two time structures: On the one hand, there is an algorithm that goes its own way determined by its inner set of rules, and on the other hand, this very same process is modified, rethought, rewritten. Obviously we have to state a basic disagreement between the programmer's time and the time of the program - each trying to reach their own aims, before they are ready again to take each other into account. This is evident in the iterative character of software development cycles. One can safely predict that this incongruousness cannot be resolved once and for all.[8] Here, we will show an attempt to find a clear way to express this relation for a sound synthesis language.

3. REFERENCE, ASSIGNMENT, DEFINITION

In the process of writing a program, there is normally a clear temporal order: first some entities are defined (variables, parameters), then some value (object) is assigned to them and only then are they referred to or operated on. Because this order is part of the temporal delimitation which we want to restructure at runtime, we need some placeholder for a later algorithm which can be used in a running program already before it is known (sometimes referred to as a *lazy proxy)*. The system discussed here (the *Just In Time Programming Library*[9]) provides such placeholders within the *SuperCollider*[10] language by extending its system of higher-order functions. By making assignment, definition and reference syntactically very similar, implications of evaluation order are avoided. The fact that one can refer to an element before it exists as well as change it when it is already in use allows the refactoring of a sound algorithm at runtime (here a very basic example in three steps - note that all code examples given are evaluated line by line):

```
Pdefn(\x, Pseq([Prand([0, 1], 4), 1, 1, 0, 1]));
```

```
Pdefn(\x, Pseq([Pdefn(\y), 1, 1, 0, 1]));
Pdefn(\y, Prand([0, 1], 4))
```

```
Pdefn(\x, Pseq([Pdefn(\y), Pdefn(\z)]));
Pdefn(\y, Prand([0, 1], 4));
Pdefn(\z, Pseq([1, 1, 0, 1]));
```

An alternative syntax for this placeholder system is provided using a modified environment access (in this case, a system of synthesis nodes). Here, an environment variable such as *~x* returns a proxy when referred to, and thus can be played before an algorithm is assigned to it.

```
ProxySpace.push;
~x.play;
~x = { SinOsc.ar([440, 540]) * 0.1 };
~x = { SinOsc.ar([440, 540]) * ~y.ar * 0.1 };
~y = { LFPulse.ar([0.3, 0.34]) };
```

[7] Note that this "monadic" structure is not necessarily identical to concurrency, but modularity seen from a temporal perspective. The exchangeable part may be a process that runs independently, but also may be e.g. a stream, a single value or a functionally defined algorithm. To be exact, the text itself should be considered such an "island" as well, in so far as it is an algorithm evaluated by the interpreter to construct a program.

[8] Sussman/Abelson seem to agree with us on this point: *"As far as anyone knows, mutability and delayed evaluation do not mix well in programming languages, and devising ways to deal with both of these at once is an active area of research."* [1], p.288

[9] *JITLib* is written in SuperCollider language and has been evolving since 2000, an introduction is to be found e.g. in Collins et al. 2004. It is part of the SuperCollider distribution.

[10] *SuperCollider* combines a dynamic programming language and a real time sound synthesis server. It is available under GPL at http://supercollider.sourceforge.net (see also [10, 11])

One proxy can play its role in any number of contexts, so that a change of one part may affect the whole system in very heterogenous ways. As a consequence, the resulting graph also has no predefined output, so that one can use any appropriate node to listen to. Different types of processes (tasks / patterns, synthesis nodes) demand different implementations of this structure: A (quasi-continuous) synthesis proxy (*NodeProxy*) takes advantage of the flexible bus architecture of the *SuperCollider server* so that its single signal can be read by any other node simultaneously independent of their node order. On the client side, *SuperCollider* implements a system of higher order stream descriptions, called patterns, that are used for tasks (sequences of evaluations) and algorithmic generation of values or sound events alike. Being streams with late evaluation, they represent a discrete and encapsulated model of time, unlike the server-side synthesis nodes. As one pattern can create multiple streams, a placeholder for such a pattern should change all streams that derive from it. The library of *PatternProxy* is used to provide such descriptions, which get "threaded into" an already existing stream that uses them. Together, these placeholders allow us to write and modify networks of interdependent temporal (or "historic") structures. By their relation, they define what a change in the program text means for the resulting algorithmic process.

4. "STATELESS" STATES

What is supposed to happen when one changes the textual description of one single node of the network? When modifying a sound algorithm at runtime it is not always that easy to recognize the difference - the old sound is gone and the new one takes over our perception. In order to grasp the effect of the modification, a certain perceptual consistency must be given. For this, we need to maintain some identity of the process, a continuity between the old and the new behaviour, which helps to understand their difference. As previously mentioned, the new text can't be taken as the description of the very same process, simply because this process is part of history.
One could suggest that we should "fast forward" the new process to the point in which the old is operating at the moment of exchange. But it is not at all trivial to know what the corresponding step or local state would have to be. Without doubt, there are cases in which such analogy can be found, but such solutions are necessarily contingent. Even more, what makes a sound is not only its present state, but its immediate past, its dynamic change in time being its only apparent quality.
Another suggestion would be to use a psychoacoustic model to know what we would perceive as a consistent transition between two slightly different versions of the sound algorithm. Apart from the fact that this might be infinitely complex, each part of the program may play very specific roles in the sound. Therefore, such a transition between the two versions wouldn't have a consistent effect.
The only solution that is left, as a kind of desperate attempt to preserve some sort of elementary identity, is to interpolate the nodes' outputs during a phase of transition while maintaining their "causal meaning" in the relations of the algorithmic network. In order to be able to synchronize textual changes at runtime, the system provides grids of reference time (allowing e.g. beat synchronicity) and functions that postpone the change until a certain condition is fulfilled. In order to be able to distribute and dislocate the change of an algorithm in a collaborative environment with several people, a small system of network dispatchers is part of the library. This makes various types of collaborative situations possible, in which the program text is part of the conversational process.

5. SOME APPLICATION FIELDS

Sonification

Sonification design and tuning is an activity where interactive programming is particularly appropriate: In the ideal design session, there is a lively discussion between domain experts and audio designers on the current version of some algorithm that transforms data into sound. Here, being able to change the scaling of some mapping of data to synthesis parameter is practically a minimum requirement; in fact, interactive sonification has been the topic of conferences and there is general agreement in the auditory display community that this is a very promising direction [12].
Being able to exchange synthesis processes while playing (which often means iterating over some subset of the data) turns the design into a much more communicative process. In the project *SonEnvir* [3,4,5], all prototypes are written in *JITLib*. Here, the acceptance by the domain experts has been very good so far; this is both useful for prototype development and for user access to make meaningful experimental changes by themselves. In fact, being able to store results of a design/tuning session as text has proven extremely valuable already; even long-term reproducibility of specific solutions on future audio programming platforms seems quite realistic.

Music Performance

A series of seminars called *Warteraum* has led to the formation of a band, *PowerBooks_UnPlugged*, with Alberto de Campo, Echo Ho, Hannes Hölzl and Jankees van Kampen. *PB_UP* employs the laptop as a complete instrument, by using internal speakers only, speech synthesis as supplied by the operating system, and live coding in a more literal sense. We use a common pool of text files with code snippets collected in rehearsals that serve as shared performance material. Every file contains one or several little scripts that create sound textures; these can be streams of note- or grain-like events, complex evolving synthesis processes, or mixtures of both.
Some of these textures have fixed durations, so that some layers end by themselves, while others are being rewritten and modified during runtime. Modifications that deem interesting are sent to the other players, along with other chat messages (such as discussions what to do next).
These variations and other ideas developed while playing can be added to the pool. The implicit working model is as democratic and symmetrical as the spatial disposition of the music: everyone can make sounds on her own laptop as well as (simultaneously or sequentially) on everyone else's. We find that the resulting uncertainty [13] is one of the most interesting and enjoyable side effects of the new possibility space created by the *JITLib* approach described here.

Film Sound

Sound synthesis for film is engaged in an interesting area between the musical (and often psychological) sound track and the atmospheric sounds that, together with the images, form the physical texture of the narrative. The sound of the experimental documentary *"Alles was wir haben"* [7] operates on the border of artificial and natural impression of atmospheric sounds, and on the expectation of "fidelity". In the development of this soundtrack almost all "real" sounds were created in a process of interactive programming, where the two artists tried to find ways toward a certain sound impression from their memory. The collaborative process was only possible in this way because the textual description of this purely synthetic, algorithmic sound could be modified while active. The story of the film evolves around the attempt to construct an (in the end, rootless) identity of a home land ("Heimat"), which may correspond quite well to the idea of a program of which we do not know if it will ever come to halt.

6. CONCLUSION

Text is often taken to be something stable, unchangeable, something that is projected into the future of a reader. A program text, seen as a description of a tool, an application, might be part of this "future-precluding" character even more [9]. But as the activity of programming reveals, neither code nor the deterministic algorithm is created in this way - iteration after iteration the written language shows its influence on the thinking and world of the programmer as much as the code changes with its use. Interactive programming can't fulfill the desire for complete and immediate control of a sound process. Clearly, this immediacy must constantly escape, thwarted by the temporal structure of the symbolic system. Nevertheless, for someone who is interested in getting involved in such problems rather than avoiding them, it can be rewarding to experiment along these lines. For us these considerations offer the promise of something like a poetic language of code to find its way into programming and sound research.

REFERENCES

[1] Abelson, H. and Sussman, G.J., *"Structure and Interpretation of Computer Programs"*, The MIT Press, Cambridge, 1996

[2] Collins, et al: *"Live Coding in Laptop Performance"*, Organized Sound, Cambridge, 2004.

[3] Dayé, C. et al, *"Sonification as a Tool to Reconstruct Users' Actions in Unobservable Areas"*. submitted to International Conference for Auditory Display, Limerick, Ireland, 2005.

[4] De Campo, A., Frauenberger, C., Höldrich, R., "*Designing a Generalized Sonification Environment"*, Proceedings of the International Computer Music Conference, Miami, USA, 2004.

[5] De Campo, A. et al, *"Sonification of Quantum Spectra"*. Submitted to International Conference for Auditory Display, Limerick, Ireland, 2005.

[6] Derrida, J, *"Signatur, Ereignis, Kontext"* (*"Signature Event Context"*), in: Jacques Derrida, Limited Inc., Wien: Passagen Verlag 2001.

[7] Kamensky, Volko: *"Alles was wir haben"* ("All That We Have"), experimental documentary film, 2004, real sound synthesis: Julian Rohrhuber. http://swiki.hfbk-hamburg.de:8888/MusicTechnology/491

[8] Latour, Bruno: *"On actor-network theory, A Few Clarifications"*, in: Soziale Welt 47/4 (1996), p. 369–381.

[9] Lyotard, Jean-Francois. *"Time Today" The Inhuman: Reflections on Time.* Trans. Geoffrey Bennington and Rachel Bowlby. Stanford: Stanford UP, 1991.

[10] McCartney, James. 1998. *"Continued Evolution of the SuperCollider Real Time Synthesis Environment."* Proceedings of the International Computer Music Conference", Ann Arbor, Michigan, 1998.

[11] McCartney, James. 2002. *"Rethinking the Computer Music Language: SuperCollider."* Computer Music Journal, 26:4, 61-8.

[12] http://www.interactive-sonification.org/

[13] Rohrhuber, J, De Campo, A, *"Waiting and Uncertainty in Computer Music Networks"*, Proceedings of the ICMC 2004, Miami, USA.