Live coding in laptop performance

NICK COLLINS,[1] ALEX McLEAN, JULIAN ROHRHUBER and ADRIAN WARD

[1]St. John's College, Cambridge, CB2 1TP
E-mail: nick@sicklincoln.org, alex@state51.co.uk, rohrhuber@uni-hamburg.de, adrian@signwave.co.uk
URL: http://www.sicklincoln.org, http://www.slab.org, http://swiki.hfbk-hamburg.de:8888/MusicTechnology/6, http://www.slub.org

Seeking new forms of expression in computer music, a small number of laptop composers are braving the challenges of coding music on the fly. Not content to submit meekly to the rigid interfaces of performance software like Ableton Live or Reason, they work with programming languages, building their own custom software, tweaking or writing the programs themselves as they perform. Often this activity takes place within some established language for computer music like SuperCollider, but there is no reason to stop errant minds pursuing their innovations in general scripting languages like Perl. This paper presents an introduction to the field of live coding, of real-time scripting during laptop music performance, and the improvisatory power and risks involved. We look at two test cases, the command-line music of slub utilising, amongst a grab-bag of technologies, Perl and REALbasic, and Julian Rohrhuber's Just In Time library for SuperCollider. We try to give a flavour of an exciting but hazardous world at the forefront of live laptop performance.

1. INTRODUCTION – LIVE PATCH BUILDING AND PROGRAMMING

There is a continuum of changes of state possible in computer music performance that stretches from simple play and stop buttons to the building of a low-level DSP engine from scratch. On the level of least interest to this paper is the manipulation of graphical interfaces built for you by a third party. In graphical programming languages, where one has access to an infinite grammar of possible constructions, the capacity for live rejigging and construction is more readily apparent. Whether in the process of creation and experimentation, or to correct an error spotted at the last minute or temporarily bypass some processing alley, Reaktor, PD or MAX/MSP users have edited the structure of the signal graph as their patches run. Moving beyond graphical programming languages to the command-line antics of interpreted text-based programming languages, the abstract potential of the system, along with its difficulty of use, continues to grow. It is the domain of text-based, scripted and command-line control of audio that fits most with the investigations in this article.

Whilst it is perfectly possible to use a cumbersome C compiler, the preferred option for live coding is that of interpreted scripting languages, giving an immediate code and run aesthetic. We do not formally set out in this article the choice between scripting languages like Perl, Ruby or SuperCollider, believing that decision to be a matter for the individual composer/programmer. Yet there is undoubtedly a sense in which the language can influence one's frame of mind, though we do not attempt anything so ambitious as to track the influence on artistic expression of a language's representational mind set. No program can be free of *a priori* strictures and assumptions and this has been discussed in detail in the literature (Flores and Winograd 1995). There is a cult to coding itself, evidenced even in popular culture by movies about hackers and virtual reality or designer's appropriation of computer iconography for record sleeves or T-shirts, and we may even tackle the issue of an aesthetic to generative coding (Cox, McLean and Ward 2001).

We are not advocating a situation in which the programmer/composer rewrites man- years' worth of support libraries. It is hard to imagine beginning entirely from scratch to write a driver or DSP engine unless you're working in the background in a venue over a number of nights, before finally emerging with a perfect heartfelt bleep on Sunday evening. Some custom coders will want to write their own libraries for standard DSP functions well before they get on with the compositional algorithms at a gig. Many will work with an established language for computer music, which comes ready specialised to the audio domain. There have been many computer music languages over the years (Loy 1989; Roads 1996: 781–818; Lyons 2002) and we shall not enter into a discussion of their relative merits for live coding, most honestly because many of them are strictly non-real-time, and others remain virgin territory for live coding experiments, at least to the authors' knowledge. Our two test cases, however, will contrast the use of general programming languages and work with the audio-specific programming language SuperCollider.

2. THE PRACTICE OF LIVE CODING

There are readers who are no doubt wondering why they would ever wish to attempt live coding in

Organised Sound **8**(3): 321–330 © 2003 Cambridge University Press. Printed in the United Kingdom. DOI: 10.1017/S135577180300030X

performance, and we might riposte immediately that live coding allows us to keep a sense of challenge and improvisation about electronic music-making. With commercial tools for laptop performance like Ableton Live and Radial now readily available, those suspicious of the fixed interfaces and design decisions of such software turn to the customisable computer language. Why not explore the interfacing of the language itself as a novel performance tool, in stark contrast to pretty but conventional user interfaces? We certainly contend that music-making is more compelling with elements of risk and reference to the performance environment, and that the human participant is essential to social musical activities. Yet we do not wish to be restricted by existing instrumental practice, but to make a true computer music that exalts the position of the programming language, that exults in the act of programming as an expressive force for music closer to the potential of the machine – live coding experiments with written communication and the programming mind-set to find new musical transformations in the sweep of code.

We confront this issue by looking at some of the power and the drawbacks of this inchoate artform in the table.

These cons and pros are in no way a final say on the matter. For the glitch aesthetic, the sense of danger is probably quite appealing, and destructive reworking of patches and the entering of illegitimate code is in itself the process of the composition. For testing before running, one could imagine using separate audio outs, or two laptops in the style of DJ auditioning. There are some tests that can be applied to code during interpretation to check validity of structures and reduce the risks of setting up an impossible unit generator network. The science of computability places limits on this capacity, especially in spotting such errors as infinite loops. In the worst case, the computer will crash or lock, though often mistakes are simple syntax errors, which just pop up error messages and allow one to make immediate corrections without any loss of sound.

Because of the danger of running very complicated new code live, in practice most composers would content themselves with modifying pre-tested snippets. They'd go to a gig with a library of prefabricated and tested code. Certain functional aspects of their engine will already be in place; for instance, some sort of time server or scheduler, and shortcuts for running, managing and killing sound agents. This is not to say that in advanced code one cannot make a massive change to the sound synthesis structure just by changing a few characters.

An area of further great potential is collaborative or competitive coding. Performers can pass code to each other to modify, allowing a very abstract sense of musical transformation, and even work in a Chinese whisper style remix circle. Games might be set up where coders have a fixed time limit to complete some goal with a restricted set of tools. One could even imagine a live coding version of the rap 'dis' battle where coders compete to aurally insult one another, or a hacker style 'root war' in which they subvert each other's computer systems.

The live sharing of information between computer music performers is already familiar from the experiments of ensembles like The Hub. That network band's programmer/composers usually wrote their sound processing programs before performance to an interfacing protocol restricted to three control streams output and input (Dean 2003: 88–93). There was,

Table. Pros and cons of live coding performance.

Pros and potential	Cons and dangers
Flexibility; the next section can be anything	You forget the current audio or just take too long while you prepare the next section
A great intellectual challenge	Your concentration halves with the adrenalin/stress/beer of a gig
Arbitrarily complex changes of structure at performance time	It's risky to just run code! No debugging or testing is available
A new form of improvisation	You must accept some failures and ugly moments of sound; there is a trade-off between preparation and gig specificity
Normal performance programs like Reason look dull if the screen is projected – but the arcane text coding systems have allure	Deliberate obfuscation is no great criteria for art!
It's rewarding to see real efforts translated into sound	You apply lots of effort for little payback; you must have back-up code in case inspiration is not forthcoming
Computer languages are immensely rich infinite grammars	Typing is hardly the most visually exciting interfacing method – you'll be bent over a screen for the night unless you code in further external controllers

however, an element of live coding in their rehearsal preparations, and an occasional attempt to modify the running code in live performance. Tim Perkis (2003) notes that:

> There was certainly quite a bit of command-line control going on in performance in the early days. Several of us, including myself, were running FORTH, and from time to time I remember putting together new little word definitions defining particular behaviours and/or sequences and letting 'em fly in performance.
>
> Things were usually pretty stable by the time we got to an actual performance, but rehearsals most generally consisted of continuous hacking by everyone involved, without shutting down the musical network interaction, everyone working in new pieces of code that would modify the overall network behaviour on the fly. I remember that Phil Stone, as the only one using a compiled language, – C – would take a certain amount of flak from everyone about going dead from time to time as he ran a compile. The rest of us just continuously modified our FORTH or LISP programs as we sort of 'sculpted' the piece.
>
> . . . in particular on the piece 'Waxlips', my machine's behaviour was controlled by a hunk of code newly auto-generated on my command at the beginning of each section. There was also a certain amount of fooling around with self-modifying code; at the time we were much more interested in finding weird and uncontrollable behaviour than in clarity, reliability, maintainability and other such outmoded concepts!

Ensemble complexities leading to emergent behaviour were very much a part of The Hub's explorations, though even in limited laptop jams one encounters the interesting tendency to lose the sense of who is contributing what. Aside from involved multiple-player scenarios with information exchange, basic collaborative performance is still extremely helpful to live coders: a partner in creation can take over the focus while you prepare your next sequence of actions. A powerful situation occurs when multiple performers are rhythmically synchronised, duos usually working extremely well, though more complex group arrangements can suffer the 'too many cooks' syndrome without sympathetic handling of the rhythmic aggregate desired.

3. TEST CASE 1 – SLUB

Slub are Alex McLean and Adrian (Ade) Ward, a London-based laptop music duo who write custom audio software in languages like Perl and REALbasic, and perform with great adaptability live, synced via a TCP/IP tick server. Alex is well known for projecting his laptop screen to show a torrent of bash shell command-line antics (McLean 2001, McLean 2003) as he starts and kills all manner of subroutines and controls the running audio processes. Ade has subverted idiomatic music interfaces to his own creative ends. The slub sound ranges from relentless techno to meditative sonic studies. The common thread throughout their music is a human, home-made quality, borne by the expressive control they have over their software.

3.1. A tour of the slub system

Here we will first describe the slub system, starting with the user interface and going down into the compositional processes and network protocol, and finishing with the sound synthesis.

Slub control their music using user interfaces created by and for themselves. These vary from the apparently conventional to the abstract, and from graphical to entirely textual. We'll skip this for now, as two of the slub interfaces will be described in detail later in this section. Figure 1 is a screenshot of some of the messier slub applications in action on Alex's laptop.

Behind the slub interfaces lie the 'compositional' or 'musical' processes – many separate pieces of code written as explorations of musical ideas. Each piece of code describes an experiment in such areas as combinatorial mathematics, chordal progressions, sonified models of dancing people, morphing metres, algorithmic breakbeats, and so on. In any case, the code engine is creating patterns, melodies and stranger musical components.

These compositional processes send messages to one another across a TCP/IP network using a line-based protocol. The messages travel via a central server, which also manages time sync between all the slub processes. This architecture allows many hundreds of compositional processes to run in parallel, and new processes can be introduced at any time, and may run from either of the laptops.

The network protocol solves a problem which might otherwise be unsolvable: Adrian and Alex often take very different approaches to making music. However, they don't have to argue about how the music is made. Because they agreed upon and implemented a network protocol between their programs, they are free to make music however they like, knowing that their programs will synchronise with each other. In this way disparate ideas may slot together seamlessly, and lock together to make music.

Finally, a hybrid software sampler and synthesizer (named 'MSG') receives messages that are either turned into sound, or affect sounds that are currently playing. At this point the distinctions between composition and synthesis blur, as experiments in synthesis lead to unexpected but reproducible effects.

This whole network of software was written by slub to fulfil their individual needs, forming an environment ideal to their methods of working. This brings us to an important point – that *the code is not just running in the computer*. To explain; when programmers are

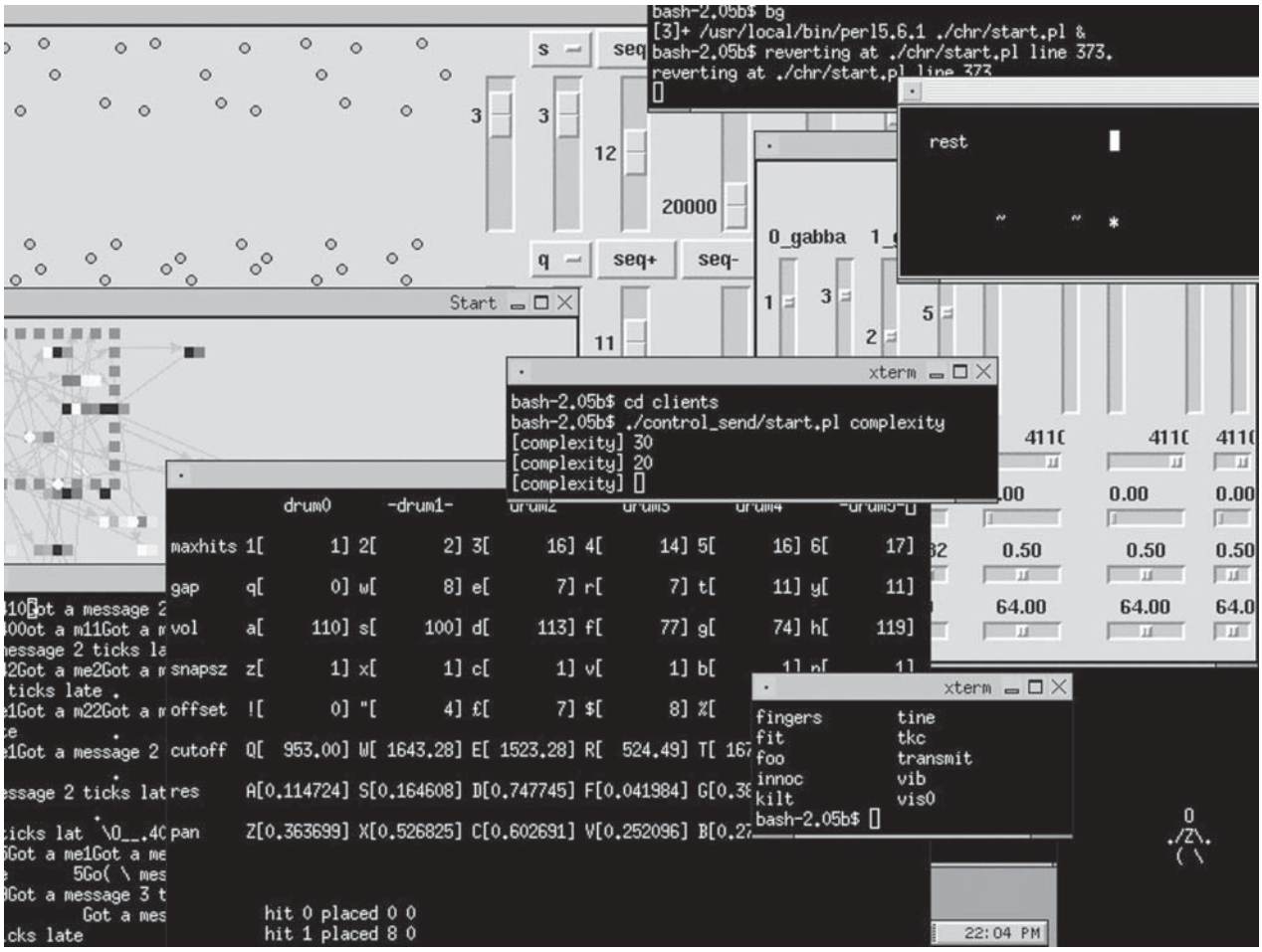

Figure 1. Alex's messy performance desktop.

watching a computer execute their own programs, the code is also executing in their minds. They have intimate knowledge of the process, and so can imagine it running. In this way, the code is alive in slub and in their computers, and hopefully also in their sound and the audience too. The music is also alive in these four places. As far as slub are concerned, code *is* music.

3.2. Interfaces

Slub often project their laptop screens to give some impression of the processes in motion that form the music. This allows the audience to see the live quality of the code as well as hear it. This quality is at times polished and professional, at other times wild and uneven, but always particular to slub.

And so here, we explore two slub interfaces. They only tell part of the story, but are particularly relevant to this article in that they involve live manipulation of code. Both of these applications were written in REALbasic by Adrian.

The first, called Map, responds to those who might refer to Max, and the entire ethos of graphical patching, as programming. Map is itself a graphical patcher, with data flowing through patch cables between blocks. But this similarity is turned into direct abuse; these blocks contain chunks of code, which can be edited in real time during a performance (figure 2).

The code used here is a home-made interpreted language sporting assembly-language- like syntax. Data moves between blocks via patch cables, which modify values in registers – each block has its own set of registers which the code can manipulate. There are language extensions which allow communication with MSG, and custom patch controls (such as the sequencer block and number inspectors shown above) which further enrich the performance capabilities. Here, the combination of a graphical interface along with the (somewhat ironic) juxtaposition of an assembly language offer two extremes of live coding possibilities; where the overall flow of data can be modified by manipulating the graphic interface, and where the minutiae of the data processing can be modified by editing the code.

The second piece of software that we describe here is called Pure Events (figure 3). Pure Events combines

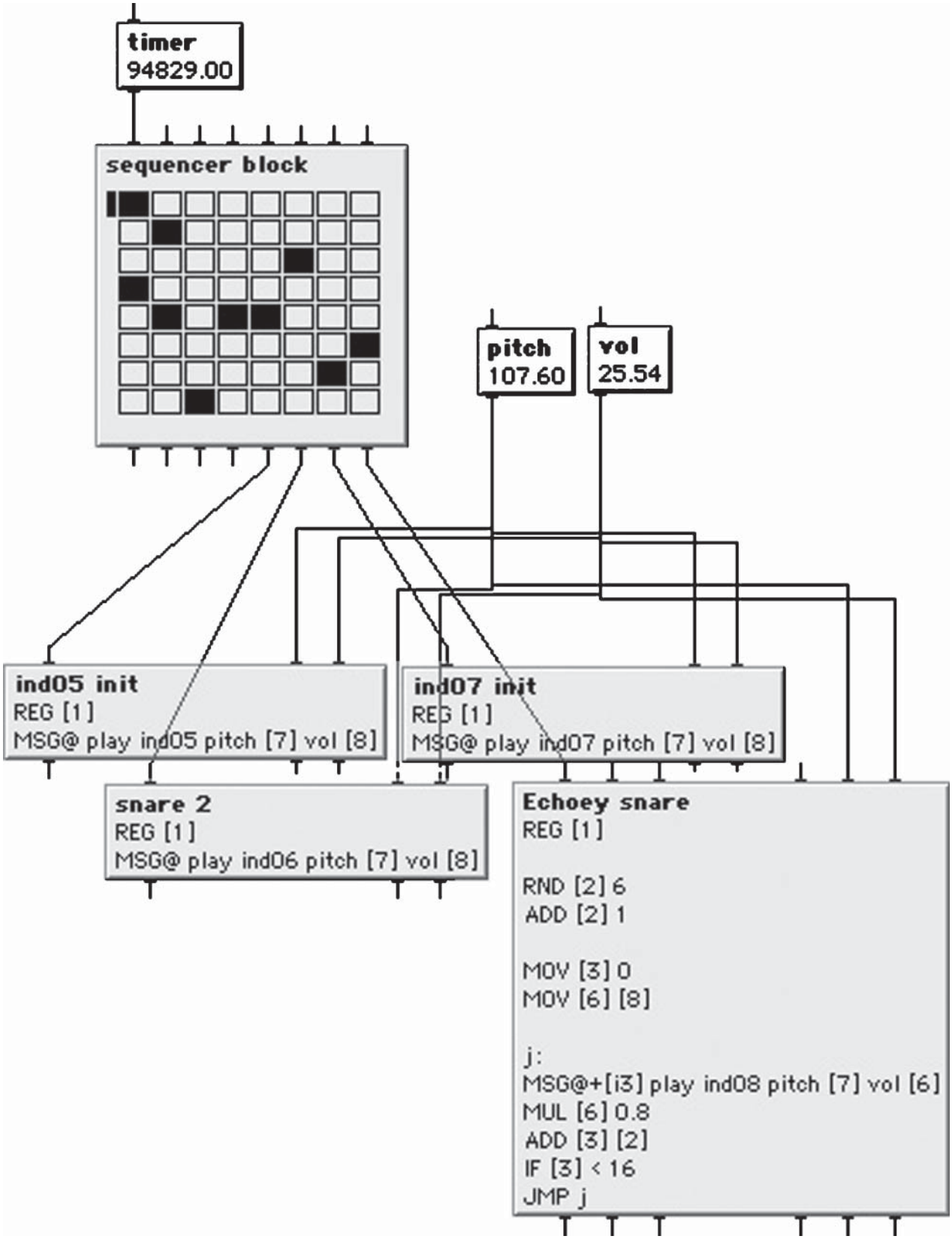

Figure 2. Map screenshot.

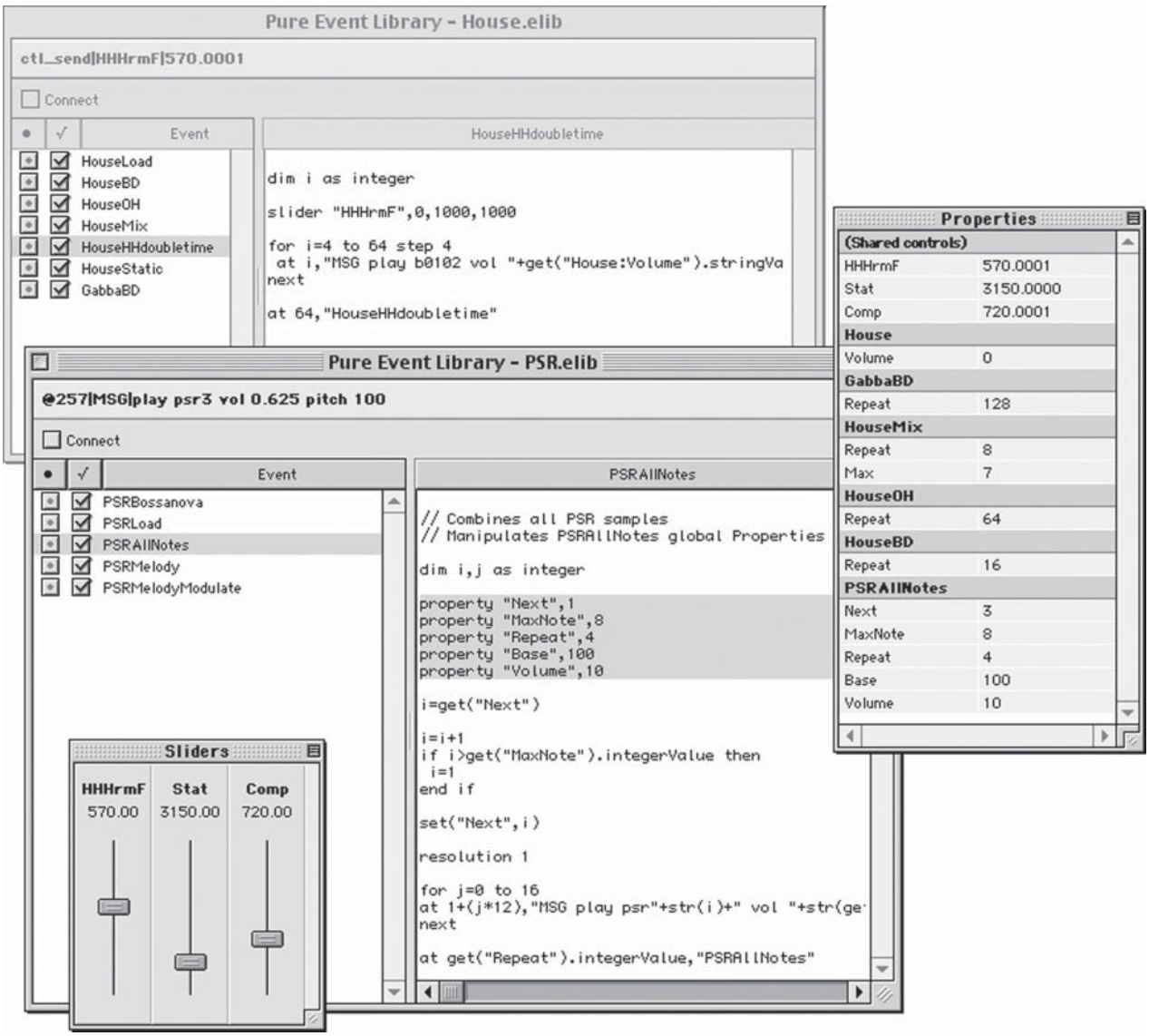

Figure 3. Pure Events screenshot.

the REALbasic runtime interpreter with the conceptual model that every event (be it macro or micro in scale) in a musical performance can be tied to a piece of code. Pure Events offers a real-time environment in which chunks of BASIC code can be scheduled to execute at any particular time, which may then schedule other chunks of code, or send instructions to the slub audio renderer.

Pure Events focuses on code that sequences musical events, and the manipulation of that code to vary the musical output. Global properties and UI controls such as sliders manipulate the data that these routines operate with, but the important factor in both of these pieces of software is that the musical code can be changed at will during a performance.

Neither application allows instant musical creation – this is not some mythical dystopian device grown from the idea that software means ultimate automation. Slub suffers somewhat from this conceptual extreme in that many think they are trying to deprecate the creative role of a performer. On the contrary, slub is opening up the determinate processes of a computer in order to generate music. A great deal of work has to be done before any performance, and any particular coding that is done during a performance is always open to problems. Part of slub's performance is about allowing those errors and glitches to occur in a context that works aesthetically. Slub software creates slub music, whilst the slub music aesthetic nurtures slub software manipulation.

4. TEST CASE 2 – JUST IN TIME LIBRARY

SuperCollider 2 (McCartney 1998) is a language for audio programming that has a dedicated user base

and is of great facility for real-time generative and interactive music. Through classes such as the MixerPlayer, SuperCollider users have been able to use the interpreter to run new code blocks as they perform, mixing in new sound functions they only just completed or edited.[1] *Extension libraries have given even wider support to this way of working (though the technicalities may have been hidden from the average user below the surface), in particular the extensive Crucial Library. What is tremendously exciting is that the new version of SuperCollider, SC Server on Max OS X and Linux (McCartney 2002), allows any programming language, not just the Smalltalk-derived SuperCollider language, to become a front end to the synthesis engine, and to pass new synthesis graphs to the renderer in real time. It is almost a call to arms for those interested in live coding: the proven power of the SuperCollider sound harnessed to the scripting language of your choice.*

Julian Rohrhuber has been a SuperCollider user since its inception, and maintains the 'swiki' site supporting the SuperCollider community. He is the author of the Just in Time Library, or jitlib, dedicated to facilitating live coding in performance.

4.1. Live coding with SuperCollider

When talking about software, be it an application or a programming environment, it is almost invisibly implied that we are dealing with some sort of tool. This tool, as with a programming language, might be itself designed for creating further software, yet is mostly seen as a device that aims towards a product. The underlying time structure is such that the toolmaker refrains from direct production, scheduling it for later, perhaps in the hope of being more effective in the future. So the product of this delay comes together with this expectation and divides the creative process in two parts, tool and its use, often even in person, toolmaker and user. In the arts this is one source of the division between the technician and the artist which is mirrored in distinctions of professions and of the concepts of the theoretical and the applied. In computer-based arts, designing the program and finally performing with it shows the same characteristic and fits well (often together with the stage and the separation of consumers and producers) into a very common artistic practice.

As long as the program has to be compiled in order to be able to run and to simulate a user interface, the time delay between creating the tool and using it seems to be very dominant – the cycle of testing, writing and compiling is slow, and the aim is separated from its description. Because there is no need for postponing activity, an essential difference comes into play when interpreted languages allow programs to be written while they run. This is not a difference in degree, it is a quantum leap that merges tool and product.

In computer music there used to be a strong distinction between the program (the instrument) and the score (the player). The mediation between them was either a written score or a graphical user interface. In the case of an interpreted language of which SuperCollider is one example, this mediation can be constructed, but essentially the distinction is blurred. An algorithm can be seen as the controller or the controlled, one process can control another process, be it the cursor position, the typing of text, a waveform or the quarter hour stroke. It is unsure who or what is the source of a sound – it could be some algorithm, some recording or some activity of that person behind that screen.

This world of synthetic cause and effect shifts meanings quickly like the weather. A programming improvisation can follow unexpected paths and interplay with its ensemble or with other circumstances. My favourite type of programming activity is to comment conversation or film by writing code, just as a bar pianist would do it, and as the program unfolds its own way between cultural and mathematic code, to discuss or do something else and wait for the next idea (Pihel 1996 is relevant here, portraying the spontaneous and risky situation of a freestyle in hip hop). This type of 'drinking while diving', of flipping the script, of walking into situations, became possible in this way in the early versions of SuperCollider 2 (SC2) when, nearly unnoticed by anyone at first, the interpreter kept running while synthesis was on.

Despite the fundamental trade-off between variability and efficiency, James McCartney has always been working towards offering real-time control of processes, the basis that made concert programming more and more realisable. After the fundamental change mentioned before, in version 2.2 the method ::triggerSynth was introduced (TrigXFade in 2.2.4), which made it possible to mix in new sound programs while others were running. Subsequently, Ron Kuivila wrote a 'special class' SC which made parallel sound processes conveniently controllable. The error message 'already playing' moved further into the past with the experimental version of SC3d and SC Server (SC3), opening a more and more neutral field of possible spontaneous interconnections between sounds and their controls. The Crucial Library player system (by Chris Sattinger) neutralised also the separation between running sound, a control, and a stored object, also introducing a more abstract scheme of reusing code. The introduction of a network layer (UDP, OSC) in version 2.2.6 that even allows us to execute code on a remote computer increased the possibilities of new rules of the game.

[1]Simple demo code for a minimal set-up based on a TSpawn and the triggersynth method would look like this:

```
{t = TSpawn.ar(nil)}.play
t.source.triggerSynth(Synth.new({SinOsc.ar(440,0,0.1)}))
```

4.2. The just in time library

My reason to introduce the just in time programming library (jitlib) was to make an interface to write code while playing that removes the distinction between preparation and action, so that I could more easily change things and not lose the connection to the written representation. Probably it has emerged from my own habit to change things up until the last moment before the performance and in many cases even during playing.

SuperCollider 3 is now based entirely on a style of command-line programming. The sound is controlled by messages that are sent to the sound server and all other constructions are built on that. We can expect these scripts to develop a new style of concert programming.

```
//define a synth definition and send it to the server
SynthDef("iiccicii", { arg r=1.0; Out.ar(LFClip
Noise.kr(r,0.5,0.5),PinkNoise.ar(SinOsc.kr(r,0,0.1).max(0)))
}).send(s);
s.sendMsg(9,"iiccicii",6788,1,0); //start a synth
s.sendMsg(15,6788,\r,128);        //change a parameter
```

One of the aims of jitlib here is to be able to exchange synth definitions and other processes within a referential environment conveniently so that there is less need to plan changes beforehand. Symbols are used to store placeholders (proxy contexts) for sound objects or processes, which constitute a referentially transparent system in which parts can be written and exchanged at runtime without a graphical interface; it lives from the visibility of code which for me was always one of the most important features of SuperCollider. In the sound programming group at Hamburg art school that I am teaching we have also made experiments with distributed referential spaces where changes influence a whole network of computers, on each of which programs run that react in their own way to a shift of the general 'data climate'.

In the SuperCollider language, lines or blocks of code can be evaluated one by one, and functions, numbers or object instances can be referenced by assigning them to variables (one-letter interpreter variables do not need initialisation) or to keys in the current environment (e.g. ~mrLloyd=78). When using a ProxySpace as an environment, these assignments are redirected to a system of placeholders that manage processes like a crossfade from one sound to another when replacing the sound function.[2] This must not be confused with simply assigning an object to a variable which does not alter the object itself. To demonstrate how such a concert program can look, I go on with a very basic commented code excerpt for SuperCollider 3:

```
prepare the set by replacing the current environment
with a proxy space on a server (could be anywhere)
which will allow me to create references to new objects
and thus substitute or change them later on


prepare a context (acting as a key to a dictionary)
to play through the output channels
s=Server.local; s.boot; ProxySpace.push(s);
~out.play;

play a classic sine sound function in it
~out = { SinOsc.ar(440,0,0.05) };

replace the current sound with another
~out = { LFPar.ar(434,0,0.02) };

start two channels with slightly different sine functions
~x1 = { SinOsc.kr(0.1) };
~x2 = { SinOsc.kr(0.14) };

controlling the rate of a filtered trigger impulses by x1 and x2
~trig = { Impulse.ar(30 *
            [ ~x1.kr.max(0), ~x2.kr.max(0) ],
0, 2)
};
~out = { RLPF.ar(~trig.ar(2), [400,410], 0.05) };

reassigning x1 and x2 with intermodulating sine waves
~x1 = { Mix(
          SinOsc.kr(2*[1,1.1,1.3,1.04] * ~x2.kr(1))
        ) + 1 * 0.2 };
~x2 = { Mix(
          SinOsc.kr(2*[1,1.1,1.31,1.2] * ~x1.kr(1))
        ) + 1 * 0.2 };
use new references to add sound processes
(
~out1 = {
       var amp, t;
      t = ~trig.ar(1) * 2;
      amp = Decay2.ar(t, 0.001, 0.01);
      BPF.ar(t, [400,431]*(amp+1), 0.05)
};
)

turn on the fridge
~out2 = { SinOsc.ar(4*50*[1,1.1], 0, 0.02) };

mix it in
~out = ~out1 + ~out2;

prepare a soft cross-fade
~out2.fadeTime = 2.0

phase modulate the fridge a little
```

[2]This kind of on-the-fly creation is not as efficient in SC3 as it would be to prepare a performance with all needed sound functions and then interconnect them, but as long as this is not done twenty times a second the flexibility is worth the expense. (There are many techniques that can be used for automated event-based processes that can be combined with this system.)

```
~out2 = { SinOsc.ar(
            4*50*[1,1.03], SinOsc.ar(300, pi), 0.02)
};

some more complex interconnections between parameters:
~out2={ SinOsc.ar(4*50*[1,1.032]*[1,~x2.kr*2+1],
SinOsc.ar(20) * pi * ~x1.kr, 0.03) };

a placeholder in this environment also constitutes a
parameter context that can be modified and is applied
to every new sound process that takes that place.

~out2={ arg freq=200, ffreq=20;
SinOsc.ar(freq*[1,1.03], SinOsc.ar(ffreq, pi) * ~x1.kr * pi,
0.02)
};

set the arguments
~out2.set(\freq, 240, \ffreq, 1200);

map argument to a proxy
~out2.map(\ffreq, ~x1);

change that proxy
~x1 = { LFNoise0.kr(3)*10 };

a new function uses the previously changed arguments
~out2 = { arg freq=200, ffreq=20;
SinOsc.ar(freq*[1,1.05], LFPulse.ar(ffreq * ~x1.ar(1), 0,
0.5) * pi, 0.02)
};

/*
. . . //up to imagination
*/

end of performance, turn off the fridge
(
~out2 = { SinOsc.ar(400*[1,1.1] * Line.kr(1, 0, 10),
SinOsc.ar(1, 0.8 * ~x1.kr + pi), 0.02) };
~x1 = { Line.kr(3, 0, 10) };
~x2 = { Line.kr(3, 0, 10) };
)
```

My own composition environment is usually a combination of such lines as well as predefined instruments, routines and patterns that similarly combine to a referential system. These sheets of code I keep modifying, and sometimes I save alternate versions. This has proved useful also in finding new algorithms, or in doing film sound, as changes can be made as part of the conversation. When a performance exceeds the lab or living room context I often project my screen to a wall but avoid sitting on a stage whenever possible – I prefer the perspective of driving a car: watch, enjoy and drive. Watching a computer music performer on stage might have a rather minimalist thrill, but I find this deficiency a good opportunity to change the default set-up. To be able to see the textual changes that cause the sound to move on is only a consequence of recognising code as an artistic form. Sometimes this situation can grow to amusing sonic offline chats when several people join in and interfere – open source then turns out to mean simply conversation.

5. CONCLUSIONS

A new area of performance is being opened up by laptop musicians attempting to work with scripting languages in live concerts. These live coding performances have a great improvisatory potential and can adapt to the night and venue, as well as great dangers. They are hardly the best way of solving the laptop-performer-stuck-behind-the-laptop dilemma, nor of minimising crashes. Yet they are of sufficient interest to warrant further attention, and whilst not all performers would ever wish to tackle a constant set of text entry, one can use the techniques in integration with spawned user interfaces, with external controllers or other more conventional audio programs.

For the interested reader, SuperCollider and related code (including jitlib and the Crucial Library work) are available via www.audiosynth.com. Audio examples for Julian's work are part of jitlib, and come complete with a download of SuperCollider Server. http://swiki.hfbk-hamburg.de:8888/MusicTechnology/500 is a page on jitlib.

For slub audio examples see fals.ch, the MEGO MP3 site, or the slub.org website (for example, http://slub.org/sounds/20020525.mp3).

ACKNOWLEDGEMENTS

Many thanks to Tim Perkis, Tom Betts, Fabrice Mogini and the sc-users mailing list for fruitful discussions on these topics. Ade wishes to thank Geoff Cox, who got him started thinking about code conceptually, and who has always supported slub in whatever way he could.

REFERENCES

Cox, G., McLean, A., and Ward, A. 2001. The aesthetics of generative code. In *Proc. of Generative Art.*

Dean, R. 2003. *Hyperimprovisation: Computer-Interactive Sound Improvisation*. Middleton, WI: A-R Editions, Inc.

Flores, F., and Winograd, T. 1995. *Understanding Computers and Cognition: A New Foundation for Design*. Addison Wesley.

Loy, D. G. 1989. Composing with computers – a survey of some compositional formalisms and programming languages for music. In M. Mathews and J. Pierce (eds.) *Current Directions in Computer Music Research*. Cambridge, MA: MIT Press.

Lyon, E. 2002. Dartmouth Symposium on the Future of Computer Music Software: a panel discussion. *Computer Music Journal* **26**(4): 13–30.

McCartney, J. 1998. Continued evolution of the SuperCollider real time synthesis environment. In *Proc. of the Int. Computer Music Conf.* Ann Arbor, Michigan.

McCartney, J. 2002. Rethinking the computer music language: SuperCollider. *Computer Music Journal* **26**(4): 61–8.

McLean, A. 2001. Hacking sound in context. On the CD-ROM *Proceedings of Music without Walls.* De Montford University, Leicester, UK, 21–3 June 2001.

McLean, A. 2003. ANGRY – usr/bin/bash as a performance tool. In S. Albert. (ed.) *Cream 12*, from the generative.net mailing list. Available online from http://twenteenthcentury.com/saul/cream12.html

Perkis, T. 2003. Personal communication.

Pihel, E. 1996. A furified freestyle: Homer and hiphop. *Oral Tradition* **11**(2).

Roads, C. 1996. *The Computer Music Tutorial*. Cambridge, MA: MIT Press.