A principled approach to developing new languages for live coding

Samuel Aaron
University of Cambridge
Computer Laboratory
Cambridge
sam.aaron@acm.org

Alan F. Blackwell
University of Cambridge
Computer Laboratory
Cambridge
alan.blackwell@cl.cam.ac.uk

Richard Hoadley
Anglia Ruskin University
Digital Performance Labs
Cambridge
richard.hoadley@anglia.ac.uk

Tim Regan
Microsoft Research
Cambridge
timregan@microsoft.com

ABSTRACT

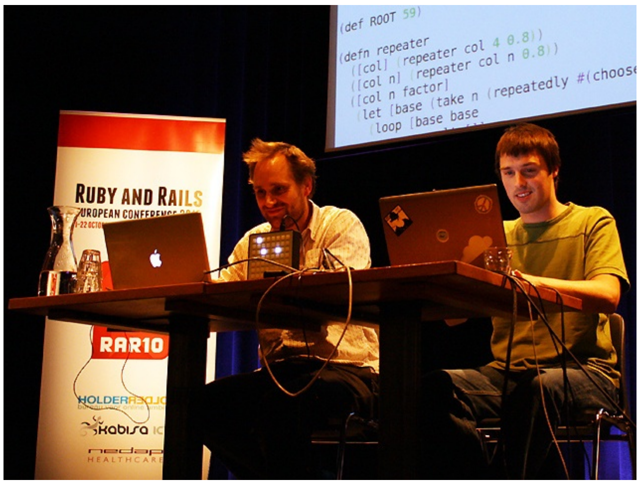
This paper introduces Improcess, a novel cross-disciplinary collaborative project focussed on the design and development of tools to structure the communication between performer and musical process. We describe a 3-tiered architecture centering around the notion of a Common Music Runtime, a shared platform on top of which inter-operating client interfaces may be combined to form new musical instruments. This approach allows hardware devices such as the monome to act as an extended hardware interface with the same power to initiate and control musical processes as a bespoke programming language. Finally, we reflect on the structure of the collaborative project itself, which offers an opportunity to discuss general research strategy for conducting highly sophisticated technical research within a performing arts environment such as the development of a personal regime of preparation for performance.

Keywords

Improvisation, live coding, controllers, monome, collaboration, concurrency, abstractions

1. INTRODUCTION

The Improcess project aims to create new tools for performing improvised electronic music in a live setting. The key goal of these tools is to structure the communication between the performer and a suite of concurrently executing musical processes. This fits within the broad genre known as live coding, where the performer writes and manipulates a computer program to generate sound. The Improcess project has started with a specific focus on a particular technical architecture and research strategy. The technical starting point has been to explore the potential for live coding performance combining domain specific music programming languages together with general purpose musical interface devices, such as the monome. Figure 1 shows a recent performance by the first author (shown on the left), in which a monome is used on stage together with a laptop running live coded software which is made visible to the audience via a large projection of the laptop's screen.



```
(def ROOT 59)

(defn repeater
  ([col] (repeater col 4 0.8))
  ([col n] (repeater col n 0.8))
  ([col n factor]
   (let [base (take n (repeatedly #(choose
     (loop [base base
```

Figure 1: The ($\lambda - tones$) performing live at the European Ruby on Rails conference in Amsterdam, 2010

The research strategy of Improcess has emphasised the creation of a cross-disciplinary team of collaborators rather than the more typical historical development of live coding systems, in which an individual developer works within an interdisciplinary context. We have also attempted to structure the project with a specific emphasis on the development of a reflective performance practice. The remainder of the paper discusses each of these aspects in turn: first our approach to experiments with the monome and with domain-specific languages (DSLs). Next, we address the architecture, the implementation, and the integration of these components into a personal regime of preparation for performance. We also reflect on the structure of the collaborative project itself, which offers an opportunity to discuss general research strategy for conducting highly sophisticated technical research within a performing arts environment.

2. THE END-USER APPROACH TO DOMAIN SPECIFIC LANGUAGES

In contrast to previous research in live coding, we start from a technical motivation related to the design and implementation of DSLs, languages specialised for creating particular kinds of application [15], and the design of languages for 'end-user programmers', people having no formal training in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NIME'11, 30 May–1 June 2011, Oslo, Norway.
Copyright remains with the author(s).

programming [18]. The languages that have been developed for live coding in the past could all be classed as DSLs, although this terminology is not necessarily used. Some have also been designed according to principles that are well-known in end-user programming (EUP) - for example, the visual syntax of Max/MSP. However, many live-coders to date have been experienced programmers, rather than musicians who acquired coding skills as a way to advance their musical practice.

In the Improcess project, we believe that there is an advantage to be obtained by drawing on broader research addressing DSLs and EUP from contexts outside of music. We have noted in the past that live coding can be an interesting object of study for researchers in EUP [10]. However in this project we draw lessons in the other direction. In particular, we consider the 'cognitive ergonomics' of language design, and of programming environments. This leads us to consider the tools in the programming environment, alternative types of syntax (including options for 'visual' diagrammatic syntax as well as textual syntax), and also the underlying computational model. As an example of the diversity of tools, syntax and computational models that are considered in EUP research, the spreadsheet is often analysed as a DSL for the accounting domain. The tools are those for inspecting, navigating and modifying the visual grid. The syntax is the combination of the diagrammatic grid with formulae that refer to that grid, and the computation model is a form of declarative constraint propagation.

The Cognitive Dimensions of Notations (CDs) is a useful framework in which to consider the interaction of these different factors [12]. When designing new DSLs and programming environments, it is possible to optimise for different criteria - for example making languages that are very easy to change (low viscosity) or occupy small amounts of screen space (low diffuseness). All of these decisions involve tradeoffs, however, that can reduce the usability of the system in other respects. Duignan has in the past used CDs productively to analyse studio technology such as Digital Audio Workstations [14]. However, in addition to his concerns, we are also interested in abstraction gradient - is it possible for newcomers to start producing music without a large prior investment of attention? Some elegant computational models preferred by computer scientists are highly abstract, to an extent that discourages casual use. This is another trade-off that we consider explicitly.

Finally, our group has in the past explored the complex relationship between direct manipulation and programming abstractions [8]. In a performance context, direct manipulation allows an audience to perceive a direct relationship between action and effect. Programming languages, in our analysis, are essentially indirect - indeed, this is a fundamental tension of live coding. We hope that the monome can be used, not only as a device to trigger events (direct manipulation), but to modify the structure of musical processes in a way that maintains, enhances or interrogates this fundamental tension.

3. INTEGRATING CONCRETE INTERACTION

Many live-coders use peripheral devices, for data capture and control. However the software architectures tend to be influenced by conventional synthesis concerns, rather than being driven by the interaction devices, since live coding interaction is largely carried out via the laptop keyboard. We were interested in the opportunities that would arise by taking a live coding approach to a specific music interaction device, and using this as a fundamental element of the performance system being developed. We therefore made the decision to incorporate a concrete performance interface into the technical architecture that we are developing.

We are currently focussing on the monome, an interaction device consisting of 64 buttons arranged in an 8x8 grid on top of a box (a monome is visible between the two laptops in figure 1). Each button contains a concealed LED which allows them to be individually illuminated. Button presses are communicated via a serial link to the host computer, where they may trigger events within executing processes. The computer can illuminate the LEDs either individually, by row or column, or all 64 buttons. The monome only produces and responds to these simple data; there is no hard-wired or audio functionality. The monome's design is open-source, making it an ideal platform for free modification, rapid prototyping and experimentation [24]. Although the monome is described by its makers simply as a general purpose 'adaptable, minimalist interface', the list of published applications demonstrates that it is mainly popular as a music controller. The videos provided by the makers (for instance, [5]), most often adopt a control layout in which horizontal button rows control ordering in time and vertical columns either control pitch or trigger some selection from a variety of samples.

As a potential tool for the live coding performance context, we believe that the monome is complementary to the qwerty keyboard, offering a number of specific advantages. Its visible physical presence makes the actions and effects of interaction evident to the observer. Unlike conventional text programming languages it offers an extremely simple interface onto which musical patterns may be mapped as shapes. The set of button triggers enable responsive and direct communication with musical environment. Finally, the embedded LEDs provide feedback allowing the software environment to communicate aspects of its current state back to the performer and audience. This allows the monome to support a range interactive styles from a simple set of button triggers to a sophisticated low resolution GUI.

The majority of monome applications to date have been constructed using the visual dataflow language Max/MSP (56 out of the 68 applications listed on the monome community site). Max/MSP provides an excellent entry point to programming for musical end-users, but it lacks a number of important software engineering capabilities, which limit the ability of users to develop and maintain complex and sophisticated applications. For example, the visual format is not compatible with the version control tools that are essential for collaborative development. There is no framework to support the specification of unit tests needed for maintenance of robust systems. Finally, the only abstraction mechanism is a form of modularity through the creation of sub-patches. It is not possible to create abstractions capable of code-generation (i.e. 'higher-order-patches') which we believe to be a key capability for exploring new live coding practices. Our proposed architecture and implementation strategy provides an alternative framework that directly addresses these issues.

4. ARCHITECTURE

The Improcess architecture extends the traditional separation of audio synthesis technology from programming language implementation to explicitly consider a suite of bespoke interface components. This approach can support multiple alternative DSLs and external interfaces by providing a common run-time on top of which these clients may be designed and built. By sharing a common run-time environment these clients can therefore inter-operate allowing a performer to create and combine a set of specifically designed interfaces for a given musical task. This separation of the traditional language implementation into system and client aspects allows for explicit design decisions regarding those system concerns that don't change from a given work or performance to the next (such as abstractions representing process and sound), from user concerns that must be changeable at any time, (such as the definition of new virtual 'instruments' and their particular control interfaces).

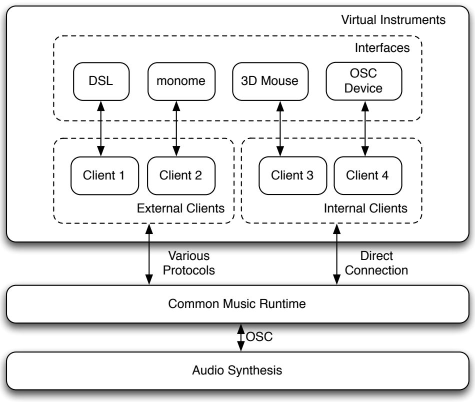


Figure 2: The Improcess architecture

The Improcess architecture consists of three tiers as illustrated in figure 2. At the bottom tier we have chosen to use the SuperCollider server as the audio synthesis engine. This provides an efficient, flexible and real-time-capable foundation, and allows us to focus on interaction and language design issues rather than audio processing. Controlling and manipulating the audio synthesis engine is the Common Music Runtime environment (CMR) which provides a suite of abstractions specifically designed for the creation and manipulation of musical processes and sound synthesis. Direct communication between the CMR and the SuperCollider server is defined in terms of Overtone [3], an open source project initiated by Jeff Rose with significant contributions by the first author.

Overtone, and the CMR, are implemented in Clojure, which was chosen because it provides a firm foundation for language experimentation and design. Clojure is an efficient functional language with an emphasis on immutability and concurrency as well as full wrapper-free access to the comprehensive Java ecosystem, highly flexible lisp syntax and meta-programming facilities that are currently best in class.

The top layer of the architecture is primarily meant for end-user interaction. This is via a suite of composable modular interfaces, which we call virtual instruments, at a level of complexity that can include complete DSLs. Each instrument has two aspects: an interface and a client implementing the required logic. Clients may either be implemented as a separate external process or as a concurrent extension to the CMR. The interaction with these instruments may either be via a conventional programming environment such as a GNU/Emacs buffer, or via a physical device such as the monome. These instruments then communicate with the CMR via the appropriate protocol in order to create and manipulate musical processes.

A frequent risk in the design of abstract architectures is the potential to disconnect from practical concerns within the application domain [9]. This can result in a design that may appear computationally powerful, yet not offer significant practical benefit for its intended purpos.e Another potential risk is the efficacy of the chosen technologies. For example, a given programming language might be able to express the desired operations yet not offer the performance semantics necessary to execute the operations within the required constraints.

We therefore took a strategic decision at the outset of the project to explore the interactive potential of our proposed software architecture by taking a number of established music applications for the monome, and re-implementing them within the proposed architecture using Clojure. Initially this has consisted of the re-implementation of monome prior-art including applications such as a sample looper, boiingg, Blinkin Park and Press Cafe. This also allowed us to explore some key technical parameters in the context of a mature interactive music application, rather than encountering them only while interacting with exploratory prototypes.

5. ABSTRACTION DESIGN

A useful and sufficient set of musical abstractions is a challenge for musical systems [7] and key to the success of the CMR. The programming abstractions found in SuperCollider are also widely available in other programming languages [19] and so it is clearly possible to reproduce equivalent semantics via a number of alternative methods. Hence we were left with the question of which musical abstractions to present to the user in the coding layer. Typical SuperCollider programs contain abstractions such as synthesisers, milliseconds, random numbers and routines. Would it be useful to also offer notions such as tunings, scales, melodies, counterpoint, rhythm and groove?

Such abstractions allow expressions that would have occurred in terms of the original complex sub-parts to be articulated more succinctly and accurately. Through redeveloping prior-art monome applications we were able to develop a vocabulary of abstractions pertaining to the monome shared across the group. We feel that this has provided a marked increase of the musical relevance of our discussions allowing much more subtle and precise notions of novel monome applications to be considered.

The main goal of the CMR is to present a suite of abstractions useful to the general task of building music performance processes. These can then be used and shared across a given set of clients to create new forms of instrument. The CMR currently supports many of the standard SuperCollider abstractions in addition to others found in alternative environments such as Impromptu's notion of 'recursion through time' [23] whereby recursive function calls are guarded by timed offsets.

An important aspect to consider with regard to abstractions is their symbolic representation. Within a procedural environment it is possible to automatically create new abstractions as part of the normal execution of the system. The ease with which this is possible is very much dependent on the syntactic structure of the representation. One of the main advantages of using a lisp as the syntactic framework is that it is very amenable to code generation. This is made possible because the syntax is represented by the basic data structures of the language (lists, maps and vectors in Clojure's case) and so generating and manipulating new code is as trivial as generating and manipulating these basic data structures.

As an example of this consider the CMR's synthesiser abstraction which comes directly from the notion of a synth in SuperCollider's server implementation. This is essentially a directed graph of unit generators such as oscillators and filters. Consider the 'bubbles' synth described SuperCollider's documentation. In listing 1 we have the SuperCollider syntax for this synth which should be compared with Overtone's version in listing 2. Notice that conceptually they are very similar in addition to being represented with a similar number of characters. The Overtone syntax does not focus specifically on typing efficiency [22] but it is in a form that is relatively straightforward to generate automatically. This opens up a number of exciting possibilities where synthesiser definitions may be automatically generated by bespoke procedures driven by client interfaces. For example, we expect to create monome applications that allow the performer to both design and instantiate new synthesisers live as part of the performance.

Listing 1: SuperCollider bubbles representation

```
SynthDef("bubbles", {
  var f, zout;
  f = LFSaw.kr(0.4, 0, 24, LFSaw.kr([8,7.23],
                0, 3, 80)).midicps;
  zout = CombN.ar(SinOsc.ar(f, 0, 0.04), 0.2,
                   0.2, 4);
  Out.ar(0, zout);
});
```

Listing 2: Overtone bubbles representation

```
(definst bubbles []
  (let [root (+ 80  (* 3  (lf-saw:kr 8 0)))
        glis (+ root (* 24 (lf-saw:kr 0.4 0)))
        freq (midicps glis)
        src  (* 0.04 (sin-osc freq))]
    (comb-n src :decaytime 4)))
```

6. IMPLEMENTATION ISSUES

Music processing architectures often introduce subtle and demanding engineering issues relating to the management of processor load, timing mechanics, and latency. A detailed discussion of these engineering issues is not appropriate to this overview paper, but in this section, we record some of the general implementation challenges that we have encountered while developing the CMR, both as a warning to newcomers, and to confirm findings reported by other developers in the past.

6.1 Event stream architecture

One of the main roles of the CMR is to convert performer actions such as button presses into generated audio that is specified in terms of musical processes. From an implementation perspective a key consideration is the internal representation of these actions and processes, particularly with respect to time [16]. Existing music synthesis architectures are often event based, but as already noted, they do not support higher-order descriptions to the extent offered by functional programming languages i.e. function composition. But in a functional language, it might also be considered more natural to represent action in terms of a function. We therefore had to decide whether to use functions or events as the "first class" internal representation of musical actions.

A key concern within our process-based model was support for thread concurrency. Function calls are typically a synchronous mechanism, with function calls executing within the current thread. Events are typically asynchronous, resulting in concurrent execution by a separate thread. For this reason, we chose an event model of musical actions, to better support our overall concern with concurrent processes in music. This also plays to Clojure's strengths given its novel concurrency semantics which are currently state of the art relative to other functional languages. In particular, events can then be represented by immutable data structures. This means that they may not be modified once created, allowing them to be freely passed around from one thread to another without risk of being modified by one thread whilst being simultaneously used a second - a common cause of error in concurrent systems, and one that would be undesirable in a live coding context.

Events also provide an intuitive computational approach to combining musical actions. For example, we may combine a series of events into a stream which is ordered in time. We may then consider flowing the stream of events through a series of functions in a similar manner that we may wish to flow an audio stream from an electronic guitar through a series of effects pedals. As in other audio processing systems such as Max/MSP, this patching together of streams of events through functions also opens us up to the possibility of forking and merging given streams. We can therefore uses a source stream to create a number of new streams which contain an identical series of events but continue on their own path. One application of this would be to generate a chord with each note played by a different synthesiser yet triggered by one root event.

For example, consider Figure 3 which represents a simple event stream setup for sending basic monome keypress events to two separate synthesisers A and B. In this example we are taking the basic events from the monome, and adding them to a buffer which then forked to two separate streams - one which is connected directly to synthesiser B and another which is mapped through a function to modify the parameters of each event and then sent to synthesiser A. This approach is extremely flexible allowing the performer to define arbitrary modifications to a given input stream. Provided the performance semantics of the mapping functions is within acceptable boundaries the end to end latency of the event stream should be acceptable for live performance.

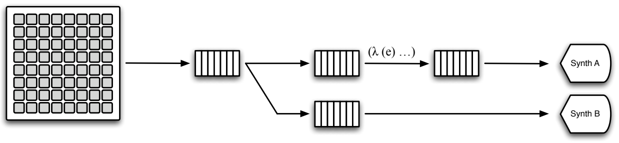


Figure 3: A typical monome event stream configuration

6.2 Performance and timing

Prior to the Improcess project the first author had implemented a monome application framework, 'monomer', using a generic programming language [4]. Monomer was intended as a general purpose toolkit for building monome applications which interfaced with external audio synthesis software via a basic MIDI interface. The result was sufficiently capable to build tools such as 8track (a Roland X0X-style MIDI step sequencer with 8 instruments / tracks [2]), but monomer suffered from two technical problems that posed considerable obstacles to building more sophisticated applications. First was the issue of performance. Even applications with minimal logic used an excessive amount of CPU. Whilst this might be acceptable for basic applications, it meant that adding any more complexity soon saturated the machine. Secondly, monomer's timing mechanics were built on top of Ruby's kernel method sleep. However variable delay in the process scheduler meant that the actual sleep time was greater than specified, and resulted in poor synchronisation with external rhythmic sources.

These experiences informed the Improcess architecture and implementation. Precise timing of event triggering is handled by SuperCollider, which uses a prioritised real-time thread rather than Ruby's non-realtime sleep operation. Clojure's execution performance far exceeds that of Ruby, and is able to further optimise particular code paths by adding type hints. Within our new architecture the main performance issue currently faced is the variable latency of message streams within the CMR. This is not always acceptable within a performing context (it may result in stuttering or jitter of audio output, or delayed response to button presses). This is a general issue of modern computing architectures due to the fact that time is often abstracted away [17]. In our specific case it may be due to low level mechanisms such as the JVM garbage collector (GC) halting all executing threads during the compaction phase or intermediate IO buffers interrupting the flow of events. In most cases these issues are resolved by fine tuning the GC, scheduling events ahead of time where possible and the removal of blocking event calls. However, this is not always an option - particularly in a situation whereby the performer wishes to trigger an immediate musical event.

7. DEVELOPING A PRACTICE REGIME

A key concern of this project has been to maintain a research focus on live coding as a performance practice, rather than simply a technical practice. An explicit goal of the research was for the first author, who was already a highly competent software developer and computer science researcher, to acquire further expertise as a performer. We build on research previously reported at NIME, including Nick Collins' (alias Click Nilson) discussion of live coding practice [21] and Jennifer Butler's discussion of pedagogical etudes for interactive instruments [13]. Butler's advice could certainly be applied to monome performers, as a 'method' to develop virtuosity on that particular controller. However, the notion of virtuosity for a programmer (or composer, as in Nash's work [20]) is more complex.

As Click Nilson reports of experiments in reflective practice by his collaborator Fredrik Olson:

> *"'I [Olson] feel I'd have to rehearse a lot more to be able to do abrupt form changes or to have multiple elements to build up bigger structures over time with. I sort of got stuck in the A of the ABA form.' Yet we also both felt we got better at it, by introducing various shortcuts, by having certain synthesis and algorithmic composition tricks in the fingers ready for episodes, and just by sheer repetition on a daily basis."* [21], p. 114.

Nilson suggests a series of live coding practice exercises, which he warns must be approached in a reflective manner. As he notes,

> *"it would be a Cagean gesture to compose an intently serious series of etudes providing frameworks for improvisation founded on certain technical abilities"* (ibid, p. 116).

Despite Click Nilson's love of Cagean gestures, we agree that there is potential for a reflective approach to live coding practice to inform the development of etudes for the live coder. Others have noted this, for example Sorenson and Brown [22] describe the problem of being able to physically type fast enough, and suggest that technical enhancements such as increased abstraction and editor support are the most important preparations for effective performance.

Our collaborative project between senior computer scientists and music academics has encouraged a perspective that extends beyond virtuosity as a purely technical accomplishment, to the artistic engagement of a practised performer with a live audience. This resulting change with respect to the intellectual scope of the first author's prior experience of DSL research can be compared to recent attention in computer science, and especially within HCI, to the value of a "critical technical practice" [6] that integrates critical and philosophical perspectives into computing. In the case of our own work, we consider performance as a category of human experience that does not arise naturally from technical work, and hence must be an explicit focus of preparation and reflection. We might call this a 'Performative Technical Practice', by analogy to Agre's work.

As noted by Nilson, Butler and Sorenson, while all musicians prepare themselves through regular practice, the analogies between conventional musical instruments and programming languages are far from straightforward. Live coding is, in many ways, more similar to musical composition than to instrumental performance. Yet it seems Quixotic to distinguish between 'practicing composition' and 'doing composition'. A composer practices composition by doing more composition. If regarded in that light, any coding might be regarded as practice for live coding. However, we did not believe that this offers adequate insight into the special status of live coding. We preferred to distinguish between coding that is presented as a performance in front of an audience - live coding - and preparation for that performance, with no audience present - practice.

From this perspective, not all coding is necessarily effective practice. At one extreme, the first author's day-to-day programming requires detailed engineering work that is essential to successful performance, but might not be effective if presented as coding to an audience (e.g. the optimisation of device drivers). At the other extreme, as noted in previous research on this topic, the live coding context expects a degree of improvisation, so preparation by simply writing the same program over and over (as when playing scales on a musical instrument) seems pointless - if the program is always the same, could it not simply be retrieved from a code library? Preparation for performance should therefore involve activities that are neither original engineering, nor simple repetition. This suggests an analogy to jazz improvisation, rather than composition or classical instrumental competence.

We explored this analogy with our advisory board member Nick Cook, a professor of 'mainstream' music research, but with specialist knowledge in both performance research and digital media. Although aware of live coding as a performance genre, he had not engaged with it in the context of preparation for performance, so provided us with a fresh perspective from which to review the previous literature. This helped us to distinguish between those aspects of instrumental practice that are predetermined and intentionally over-learned (e.g. scales), those concerned with 'difficult corners' in a specific genre that might be practiced (e.g. a tricky bridge when a verse has a key change), those that develop a vocabulary of reference to other works (perhaps through riffs or 'formulae'), and those that prepare a specific 'piece' for performance (although the notes played within any given improvised performance will naturally vary).

Our current strategy has therefore been to integrate these elements into a series of daily exercises that develop fluency of low-level actions relevant to live coding. We assume that in actual performance, the program being created will be novel. But a fluent repertoire of low-level coding activities will allow the performer to approach performance at a higher level of structural abstraction - based on questions such as 'where am I going in this performance' and 'what alternative ways are there for getting there'. In accordance with proposals by Collins and Butler, we are planning to publish a set of etudes that can develop fluency in aspects of a program related to rhythm, to harmonic/melodic structure, to texture, and to software structure. We also recognise that live coding performance is a multimedia genre, in which the contents of the screen display, and the stage presence of the performer are also essential components. A daily practice regime of preparation for performance should also incorporate etudes that develop fluency of action in these respects. These aspects of our work might be considered as constructing an 'architecture' of performance skill that complements the technical architecture of the software infrastructure.

8. CONCLUSIONS & FUTURE WORK

We have described the initial results of the Improcess project, which has developed an architecture for live coding performance motivated by the considerations of domain-specific programming language design, and by research into end-user programming. We also have an explicit concern with music performance practice that has led us to treat the integration of interface devices such as the monome as a primary architectural consideration. We have created an effective architecture with a functioning implementation, and have demonstrated that it can be used to reimplement some popular monome applications. We are now using this platform to further develop a regime of preparation for performance within a reflective research context. The Improcess environment provides a technical foundation that mirrors this musical and collaborative goal. We expect that it will enable rapid exploration of a diverse range of language options, including the potential implementation of new language features in live contexts, and even the construction of tangible performance 'instruments' that can themselves generate code processes.

We have also found that explicit reflection on interdisciplinary collaboration has been a valuable element of our research. Improcess is hosted by the Crucible network for research in interdisciplinary design, which aims to make informed contributions to public policy on the basis of projects like this [11]. In Improcess we found the intellectual contrasts between computational and musical perspectives sufficiently extreme (for example, in different team-members various interpretations of the monome as either ideally flexible or undesirably grid-like) that we represented not only multiple organisations, but multiple research cultures. We have used the Cross-Cultural Partnership template [1] to help us bring together people from these different 'cultural norms and legal frameworks for sharing culture'.

As a project demonstrating a 'performative technical practice', we believe that this juxtaposition and improvisation of cultural, technical and musical processes represents the essence of the live coding enterprise.

9. ACKNOWLEDGEMENTS

Thanks to the other members of the Improcess partnership - Tom Hall, Stuart Taylor, Chris Nash and Ian Cross - for their continued support and encouragement, and to the members of our advisory board: Simon Peyton Jones, Nick Cook, Simon Godsill, Nick Collins and Julio d'Escrivan. Thanks also to Jeff Rose and Fabian Aussems for their work on the Overtone project.

10. REFERENCES

[1] http://connected-knowledge.net/.

[2] http://docs.monome.org/doku.php?id=app:8track.

[3] http://github.com/overtone/overtone.

[4] http://github.com/samaaron/monomer.

[5] http://www.vimeo.com/290729.

[6] P. E. Agre. *Toward a Critical Technical Practice : Lessons Learned in Trying to Reform AI.* Lawrence Erlbaum Associates, 1997.

[7] P. Berg. Abstracting the Future: The Search for Musical Constructs. *Computer Music Journal*, 20(3):24–27, 1996.

[8] A. F. Blackwell. First steps in programming: a rationale for attention investment models. *Proceedings of IEEE Symposia on Human Centric Computing Languages and Environments*, pages 2–10, 2002.

[9] A. F. Blackwell, L. Church, and T. Green. The Abstract is 'an Enemy': Alternative Perspectives to Computational Thinking. *Proceedings of the 20th annual workshop of the Psychology of Programming Interest Group*, pages 34–43, 2008.

[10] A. F. Blackwell and N. Collins. The Programming Language as a Musical Instrument. *Psychology of Programming Interest Group*, pages 120–130, 2005.

[11] A. F. Blackwell and D. A. Good. *Languages of Innovation*, pages 127–138. University Press of America, 2008.

[12] A. F. Blackwell and T. Green. *Notational Systems - the Cognitive Dimensions of Notations framework*, pages 103–134. Morgan Kaufmann, 2003.

[13] J. Butler. Creating Pedagogical Etudes for Interactive Instruments. *Proceedings of the International Conferences on New Interfaces for Musical Expression*, pages 77–80, 2008.

[14] M. Duignan, J. Noble, and R. Biddle. Abstraction and Activity in Computer Mediated Music Production. *Computer Music Journal*, 34(Barr 2003):22–33, 2010.

[15] M. Fowler. *Domain-Specific Languages.* Addison-Wesley, 2011.

[16] H. Honing. Issues on the representation of time and structure in music. *Contemporary Music Review*, 9(1):221–238, 1993.

[17] E. A. Lee. Computing needs time. *Communications of the ACM*, 52(5):70–79, May 2009.

[18] H. Lieberman, F. Paterno, and V. Wulf. *End User Development.* Springer, 2006.

[19] J. McCartney. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4):61–68, Dec. 2002.

[20] C. Nash and A. Blackwell. Beyond Realtime Performance : Designing and Modelling the Creative User Experience. *Submission to NIME*, 2011.

[21] C. Nilson. Live coding practice. *Proceedings of the 7th international conference on New interfaces for musical expression*, page 112, 2007.

[22] A. Sorensen and A. R. Brown. aa-cell In Practice : An Approach to Musical Live Coding. *Proceedings of the International Computer Music Conference*, 2007.

[23] A. Sorensen and H. Gardner. Programming With Time Cyber-physical programming with Impromptu. *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications*, pages 822–834, 2010.

[24] O. Vallis, J. Hochenbaum, and A. Kapur. A Shift Towards Iterative and Open-Source Design for Musical Interfaces. *Proceedings of the 2010 Conference on New Interfaces for Musical Expression (NIME 2010)*, (Nime):1–6, 2010.