TIDAL – PATTERN LANGUAGE FOR LIVE CODING OF MUSIC

Alex McLean and Geraint Wiggins
Centre for Cognition, Computation and Culture
Department of Computing
Goldsmiths, University of London

ABSTRACT

Computer language for the description of pattern has been employed for both analysis and composition of music. In this paper we investigate the latter, with particular interest in pattern language for use in live coding performance [1]. Towards this end we introduce Tidal, a pattern language designed for music improvisation, and embedded in the Haskell programming language.

Tidal represents polyphonic patterns as a time varying function, providing an extensible range of pattern generators and combinators for composing patterns out of hierarchies of sub-patterns. Open Sound Control (OSC) messages are used to trigger sound events, where each OSC parameter may be expressed as a pattern. Tidal is designed to allow patterns to be created and modified during a live coded performance, aided by terse, expressive syntax and integration with an emerging time synchronisation standard.

1. INTRODUCTION

When we view the composed sequence "abcabcabc. . . " we quickly infer the pattern "repeat abc". This is inference of hierarchy aiding memory of long sequences, prediction of future values and recognition of objects. Pattern pervades the arts; as Alfred Whitehead [2] eloquently puts it, "Art is the imposing of a pattern on experience, and our aesthetic enjoyment is recognition of the pattern." To our shame these words were background to Whitehead lambasting those taking quotes out of context, but nonetheless communicate a role of pattern supported here; one individual encodes a pattern and another decodes it, both actively engaged with the work while creating their own experience. In this paper we examine the encoding of pattern in particular, introducing Tidal, a computer language for encoding musical patterns during improvised live coding performances [1].

Pattern is everywhere, and the subject of musical pattern is a broad subject alone. The desire to capture musical patterns with machines goes back to well before electronic computers. For example, Leonardo da Vinci invented a hurdy gurdy with movable pegs to encode a pattern, and multiple adjustable reeds which transformed the pattern into a canon [3]. Hierarchies and heterarchies of repeating structure run throughout much of music theory, and computational approaches to music analysis, indexing and composition all have focus on discrete musical events and the rules to which they conform [4, §4.2]. From this we assert that the encoding of pattern is fundamental to music making. In the following we review support given to musical pattern making by computer language, and then introduce Tidal, a language for live improvisation of musical pattern. Before that we motivate the discussion through brief review of the practice of *live coding*, for which Tidal has been created.

Copyright: ©2010 Alex McLean et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License 3.0 Unported, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

2. LIVE CODING

Since 2003 an active group of practitioners and researchers [5] have been developing new (and rejuvinating old) approaches to improvising computer music and video animation; activity collectively known as *live coding* [6, 1, 7]. The archetypal live coding performance involves programmers writing code on stage, with their screens projected for an audience. The code is dynamically interpreted, taking on edits on-the fly without losing process state, so that no unwanted discontinuities in the output occur. Here a software development process is the performance, with the musical or visual work generated not by a finished program, but its journey of development from an empty text editor to complex algorithm, generating continuously changing musical or visual form along the way.

A key challenge set to live coders is to react quickly in musical response to other performers, or else on their own whim. This can be difficult due to a straight trade off in the level of abstraction they have chosen; while a traditional instrumentalist makes one movement to make one sound, a live coder makes many movements (key presses) in order to describe many sounds. It is in a live coder's interest to find highly expressive computer language that allows their ideas to be described succinctly. The subject of We believe the prese focus on the composition of pattern language provides a positive step in the right direction.

3. PATTERN LANGUAGE

Literature on pattern language is mainly concerned with *analysis* of composed works relative to a particular theory of music. For example Simon and Sumner [8] propose a formal language for music analysis, consisting of a minimal grammar for describing phrase structure within periodic patterns. Their language allows for multidimen-

sional patterns, where different aspects such as note value, onset and duration may be expressed together. The grammar is based on a language used for description of aptitude tests which treat pattern induction as a correlate for intelligence. Somewhat relatedly, research has since suggested that there is a causal link between music listening and intelligence [9], known as the "Mozart effect". However this result has proved highly controversial [10], and we would certainly not claim that pattern language makes you clever. Deutsch and Feroe [11] introduced a similar pattern language to that of Simon and Sumners, for the analysis of hierarchical relationships in tonal music with reference to gestalt theory of perception.

The analytical perspective shown in the above languages puts focus on simple patterns with unambiguous interpretation. Music composition however demands complex patterns with many possible interpretations, leading to divergent perception across listeners. Therefore pattern language for synthesis of music requires a different approach from analysis. Indeed, a need for the design of pattern language for music composition is identified by Laurie Spiegel in her 1981 paper "Manipulations of Musical Patterns" [12]. Twelve classes of pattern transformation, taken from Spiegel's own introspection as a composer are detailed: transposition (translation by value), reversal (value inversion or time reversal), rotation (cycle time phase), phase offset (relative rotation, e.g. a canon), rescaling (of time or value), interpolation (adding midpoints and ornamentation), extrapolation (continuation), fragmentation (breaking up of an established pattern), substitution (against expectation), combination (by value – mixing/-counterpoint/harmony), sequencing (by time – editing) and repetition. Spiegel felt these to be 'tried and true' basic operations, which should be included in computer music editors alongside insert, delete and search-and-replace. Further, Spiegel proposed that studying these transformations could aid our understanding of the temporal forms shared by music and experimental film, including human perception of them.

Pattern transformations are evident in Spiegel's own Music Mouse software, and can also be seen in music software based on the traditional studio recording paradigm such as Steinberg Cubase and Apple Logic Studio. However Spiegel is a strong advocate for the role of the musician programmer, and expresses hope that these pattern transformations would be formalised into programming libraries. Such libraries have indeed since emerged. Hierarchical Music Specification Language (HMSL) developed in the 1980s includes an extensible framework for algorithmic composition, with some inbuilt pattern transformations. The Scheme based *Common Music* environment, developed from 1989, contains a well developed object oriented pattern library [13]; classes are provided for pattern transformations such as permutation, rotation and random selection, and for pattern generation such as Markov models, state transition and rewrite rules. The SuperCollider language [14] also comes with a extensive pattern library, benefiting from an active free software development community, and with advanced support for live coding. These systems are all inspiration for our own pattern language, introduced below.

4. TIDAL

Tidal is a pattern language embedded in the Haskell programming language, consisting of pattern representation, a library of pattern generators and combinators, an event scheduler and programmer's live coding interface. This is an extensive re-write of earlier work introduced under the working title of *Petrol* [15]. Extensions include improved pattern representation and fully configurable integration with the Open Sound Control (OSC) protocol [16].

4.1 Features

Before examining Tidal in detail we first characterise it in terms of features expected of a pattern language.

4.1.1 Host language

Tidal is a domain specific language embedded in the Haskell programming language. The choice of Haskell allows us to use its powerful type system, but also forces us to work within strict constraints brought by its static types and pure functions. We can however turn this strictness to our advantage, through use of Haskell's pioneering type-level constructs such as functors and monads. Once the notion of a pattern is defined in terms of these constructs a whole range of cutting edge computer research becomes available, which can then be explored for application in describing musical pattern.

Tidal inherits Haskell's syntax which is both terse (thanks to its declarative approach) and flexible, for example it is trivial to define new infix operators. Terse syntax allows for faster expression of ideas, and therefore a tighter programmer feedback loop more suitable for creative tasks [17].

4.1.2 Pattern composition

In Tidal, patterns may be composed of numerous sub-patterns in a variety of ways and to arbitrary depth, to produce complex wholes from simple parts. This could include concatenating patterns time-wise, merging them so that they co-occur, or performing pairwise operations across patterns, for example combining two numerical patterns by multiplying their values together. Composition may be heterarchical, where sub-pattern transformations are applied at more than one level of depth within a larger pattern.

4.1.3 Random access

Both Common Music and SuperCollider represent patterns using lazy evaluated lists, where values are calculated one at a time as needed, rather than all together when the list is defined. This allows long, perhaps infinitely long lists to be represented efficiently in memory as generator functions, useful for representing fractal patterns for example. In some languages, including Haskell, lists are lazily evaluated by default, without need for special syntax. This is not how patterns are represented in Tidal however. Lazy lists

are practical for linear operations, but you cannot evaluate the 100th value without first evaluating the first 99. This is a particular problem for live coding; if you were to change the definition of a lazy list, in order to continue where you left off you must regenerate the entire performance up to the current time position. [1] Further, it is much more computationally expensive to perform operations over a whole pattern without random access, even in the case of straightforward reversal.

Tidal allows for random access by representing a pattern not as a list of events but as a function from time values to events. A full description is given in §4.2.

4.1.4 Time representation

Time can be conceptualised either as *linear change* with *forward order* of succession, or as a repeating cycle where the end is also the beginning of the next repetition [18]. We can relate the former to the latter by noting that the phase plane of a sine wave is a circle; a sine wave progresses over linear time, but its oscillation is a repeating cycle. As a temporal artform, the same division is present in music, in repeating rhythmic structures that nonetheless progress linearly. For this reason Tidal allows both periodic and infinite patterns to be represented.

Another important distinction is between discrete and continuous time. In music tradition, time may be notated within discrete symbols, such as Western staff notation or Indian bol syllables, but performed with subtle phrasing over continuous time. Tidal maintains this distinction, where patterns are events over discrete time steps, but may include patterns of floating point onset time deltas. More details on this in §5.1.

4.1.5 Ready-made generators and transforms

A pattern library should contain a range of basic pattern generators and transforms, which can be straightforwardly composed into complex structures. It may also contain more complex transforms, or else have a community repository where such patterns may be shared. Tidal contains a range of these, some of which are inspired by other pattern languages, and others that come for free from Haskell's standard library of functions, including its general support for manipulating collections.

4.1.6 Community

> "Computers're bringing about a situation that's like the invention of harmony. Sub-routines are like chords. No one would think of keeping a chord to himself. You'd give it to anyone who wanted it. You'd welcome alterations of it. Sub-routines are altered by a single punch. We're getting music made by man himself: not just one man." *John Cage, 1969* [19]

John Cage's vision has not universally met with reality, much music software is proprietary, and in the United States sound synthesis algorithms are impeded by software patents. However computer music languages are judged by their communities, sharing code and ideas freely, particularly around languages released as free software themselves. A pattern language then should make sharing abstract musical ideas straightforward, so short snippets of code may be easily used, modified and passed on. This is certainly possible with Tidal, although this is a young language which has not yet had a community grow around it. Towards this end however, the first author is developing a website for sharing snippets of musical code, for Tidal and other languages.

4.2 Representation

We now turn to the detail of how Tidal represents patterns. The period of a pattern – the duration at which it repeats – is represented in Haskell's type system as an integer:

```
type Period = Maybe Int
```

The integer type `Int` is encapulated within the `Maybe` type, so that we can represent both periodic and non-periodic (i.e. infinite) patterns. For example the pattern "*a* followed by repeating *b*s" has a `Period` of `Nothing`, and "*abcdefgh*, repeated" would have a `Period` of `Just 8` [2].

The structure of a pattern is defined as a function from integer time to a list of events:

```
type Behaviour a = Int → [Maybe a]
```

The name of the `Behaviour` type is borrowed from reactive programming nomenclature [20], where a behaviour is the term for a time-varying value. Note that `Behaviour` is an abstract type, where `a` is a wild card standing for any other type. For example a pattern of musical notes could be of type `Behaviour String`, where pitch labels are represented as character strings, or alternatively of type `Behaviour Int` for a pattern of MIDI numbered note events. Another thing to note is that the `Maybe` type is again employed so that non-values may be included in a list of events. The reader may ask, why would you want to store non-values in a list at all? We might simply answer that a rest has a particular musical identity and so needs to be represented. More practical motivation is shown in §5, where `Nothing` is shown to have different meaning in different situations.

A pattern then is composed of a `Behaviour` and `Period`, given the field names `at` and `period` respectively:

```
data Pattern a =
  Pattern {at :: Behaviour, period :: Period}
```

A pattern may be constructed as in the following example representing the repeating sequence "0, 2, 4, 6":

```
p = Pattern {at = λn → [Just ((n `mod` 4) * 2)],
             period = Just 4}
```

We access values by evaluating a behaviour with a time value, for example with the above pattern, `at p 1` evaluates to `[Just 2]`. As this is a cyclic pattern of period 4, `at p 5` would give the same result, as would `at p (-3)`.

[1] SuperCollider supports live coding patterns using PatternProxiess [7]. These act as place-holders within a pattern, allowing a programmer to define sub-patterns which may be modified later.

[2] `Just` and `Nothing` are the two constructors of Haskell's `Maybe` type

The above pattern is expressed as a function over time. An approach more idiomatic to Haskell would be to define it recursively, in this case defining `at p 0` to return `Just [0]` and subsequent `at p n` to return the value at `n - 1` plus two. However great care must be taken when introducing such dependencies between time steps; it is easy to produce uncomputable patterns, or as in this case patterns which may require whole cycles to be computed to find values at a single time point.

4.3 Pattern generators

A pattern would not normally be described by directly invoking the constructor in the rather long-winded manner shown in the previous section, but by calling one of the pattern generating functions provided by Tidal. These consist of generators of basic repeating forms analogous to sine, square and triangle waves, and a parser of complex sequences. The `sine1` function produces a sine cycle of floating point numbers in the range 0 to 1 with a given period, here rendered as grey values with the `drawGray` function:

```
drawGray $ sine1 16
```

Tidal is designed for use in live music improvisation, but is also applicable for off-line composition, or for non musical domains. We take this opportunity to illustrate the examples in the following sections with visual patterns of colour as above, in sympathy with the present medium. For space efficiency the above cyclic pattern is rendered as a row of blocks, but ideally would be rendered as a circle, as the end of one cycle is also the beginning of the next.

Linear interpolation between values, somewhat related to musical glissandi, is provided by the `tween` function:

```
drawGray $ tween 0.0 1.0 16
```

If a pattern is given as a string, it is parsed according to the context, made possible through Haskell's type inference, and a string overloading extension.

```
draw "black blue lightgrey"
```

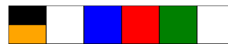
In the above example the `draw` function requires a colour pattern, and so a parser of colour names is automatically employed. Tidal can parse the basic types `String`, `Bool`, `Int` and `Float` and it is straightforward to add more as needed. All these parsers are expressed in terms of a common parser, which provides syntax for combining sub-patterns together into polymetric patterns. Sub-patterns with different periods may be combined either by repetition or by padding. In both cases the result is a combined pattern with period of the lowest common multiple of those of the constituent patterns. Combining patterns by repetition is denoted by square brackets, where constituent parts are separated by commas. In the following example the first part is repeated twice and the second thrice:

```
draw "[black blue green, orange red]"
```

Note that co-occurring events are visualised by the draw function as vertically stacked colour blocks.

Combining by padding each part with rests is denoted with curly brackets, and inspired by the Bol Processor [21]. In this example the first part is padded with one rest every step, and the second with two rests:

```
draw "{black blue green, orange red}"
```



In the above example there are steps where two events co-occur, and the block is split in two, where one event occurs, taking up the whole block, and where no events occur and the block is blank.

Polymetries may be embedded to any depth (note the use of a tilde to denote a rest):

```
draw "[{black ~ grey, orange}, red green]"
```

4.4 Pattern combinators

If an underlying pattern representation were to be a list, a pattern transformer would have to operate directly on sequences of events. For example, we might *rotate* a pattern one step forward by *pop*ping from the end of the list, and *unshift*ing/*cons*ing the result to the head of the list. In Tidal, because a pattern is a function from time to events, a transformer may manipulate time as well as events. Accordingly the Tidal function `rotL` for rotating a pattern to the left is straightforwardly defined as:

```
rotL p n =
  Pattern (λt → at p (t + n)) (period p)
```

Rotating to the right is simply defined as the inverse:

```
rotR p n = rotL p (0 - n)
```

We won't go into the implementation details of all the pattern transformers here, suffice to say that they are all implemented as composable behaviours. The reader may refer to the source code for further details.

The `cat` function concatenates patterns together timewise:

```
drawGray $ cat [tween 0 1 8, tween 1 0 8]
```

As you might expect, the period of the resulting pattern will be the sum of the constituent pattern periods, unless one of the constituents is infinite, in which case the result will also be infinite.

A periodic pattern may be reversed with `rev`:

```
drawGray $ rev (sine1 8)
```

Or alternatively expressed forwards and then in reverse with `palindrome`:

```
drawGray $ palindrome (sine1 8)
```

The `every` function allows transformations to be applied to periodic patterns every n cycles. For example, to rotate a pattern by a single step every third repetition:

```
draw $ every 3 (1 `rotR`) "black grey red"
```

The Pattern type is defined as an applicative functor, allowing a function to be applied to every element of a pattern using the <$> functor map operator. For example, we may add some blue to a whole pattern by mapping the `blend` function (from the Haskell Colour library) over its elements:

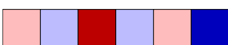
```
draw $ blend 0.5 blue <$> p
  where p = every 3 (1 `rotR`) "black grey red"
```

If we were doing something similar to a sound rather than colour event, we might understand it as a musical transposition. We can also apply the functor map conditionally, for example to transpose every third cycle:

```
drawGray $ every 3 ((+ 0.6) <$>) "0.2 0.3 0 0.4"
```

The Haskell applicative functor syntax also allows a new pattern to be composed by applying a function to combinations of values from other patterns. For example, the following gives a polyrhythmic lightening and darkening effect, by blending values from two patterns:

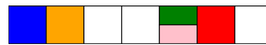
```
draw $
  (blend 0.5) <$> "red blue" <*>
"white white black"
```



The use of <∗> here deserves some explanation. It allows us to map a function over more than one pattern at a time. In the above example, for each call to blend, a value is taken from each pattern. The <∗> operator is defined for Patterns so that all events are used at least once, and no more than necessary to fulfil this constraint. Operationally, the shorter list of events is repeated until it is the same length as the longer; this is behaviour halfway between that of a Haskell `List` and a `ZipList`. For implementation details please refer to the code, but the end result are minimal combinations of polyphonic events without discarding any values.

The Tidal `onsets` function filters out elements that do not begin a phrase. Here we manipulate the onsets of a pattern (blending them with red), before combining them back with the original pattern.

```
draw $ combine [blend 0.5 red <$> onsets p, p]
  where p = "blue orange ˜ ˜ [green, pink] red ˜"
```



The `onsets` function is particularly useful in cross-domain patterning, for example taking a pattern of notes and accentuating phrase onsets by making a time onset and/or velocity pattern from it.

5. OPEN SOUND CONTROL PATTERNS

Tidal has no capability for sound synthesis itself, but instead represents and schedules patterns of OSC messages to be sent to a synthesiser. Below we see how the ‘shape’ of an OSC message is described in Tidal:

```
synth = OscShape {path = "/trigger",
                  params =
                    [ F "note" Nothing,
                      F "velocity" (Just 1),
                      S "wave" (Just "triangle")
                    ],
                  timestamp = True
                 }
```

This is a trivial `"/trigger"` message consisting of two floating point parameters and one string parameter. Each parameter may be given a default value in the `OscShape`; in this case velocity has a default of `1`, wave has a default of `"triangle"` and note has no default. This means if a OSC pattern contains a message without a note value set, there will be no value to default it to, and so the message is discarded. Pattern accessors for each parameter are defined using names given in the `OscShape`:

```
note     = makeF synth "note"
velocity = makeF synth "velocity"
wave     = makeS synth "wave"
```

5.1 Scheduling

As `timestamp` is set to `True` in our OscShape example, one extra pattern accessor is available to us, for onset deltas:

```
onset = makeT synth
```

This allows us to make time patterns, applying subtle (or if you prefer, unsubtle) expression. This is implemented by wrapping each message in a timestamped OSC bundle. A simple example is to vary onset times by up to 0.02 seconds using a sine function:

```
onset $ (* 0.02) <$> sine 16
```

Instances of Tidal can synchronise with each other (and indeed other systems) via the NetClock protocol (`http://netclock.slab.org/`). NetClock is based upon time synchronisation in SuperCollider [14]. This means that time patterns can notionally schedule events to occur

in the past, up to the SuperCollider control latency, which has a default of 0.2 seconds.

It is also possible to create tempo patterns to globally affect all NetClock clients, for example to double the tempo over 32 time steps:

```
tempo $ tween 120 240 32
```

5.2 Sending messages

We connect our OSC pattern to a synthesiser using a `stream`, passing the network address and port of the synthesiser, along with the `OscShape` we defined earlier:

```
s ← stream "127.0.0.1" 7770 synth
```

This starts a scheduling thread for sending the messages, and returns a function for replacing the current pattern in shared memory. Patterns are composed into an OSC message Pattern and streamed to the synthesiser as follows:

```
s $ note ("50 ~ 62 60 ~ ~")
    ~~ velocity foo
    ~~ wave "square"
    ~~ onset ((* 0.01) <$> foo)
  where foo = sine1 16
```

The `~~` operator merges the three parameter patterns and the onset pattern together, into a single OSC message pattern. This is then passed to the stream `s`, replacing the currently scheduled pattern. Note that both `velocity` and `onset` are defined in terms of the separately defined pattern `foo`.

5.3 Use in music improvisation

Music improvisation is made possible in Tidal using the dynamic Glasgow Haskell Compiler Interpreter (`http://www.haskell.org/ghc/`). This allows the musician to develop a pattern over successive calls, perhaps modifying the preceding listing to transpose the note values every third period, make a polyrhythmic pattern of wave shapes, or combine multiple onset patterns into a chorus effect. Tidal provides a mode for the iconic emacs programmer's editor (`http://www.gnu.org/software/emacs/`) as a GHCI interface, allowing patterns to be live coded within an advanced developers environment. [3]

6. CONCLUSION

We have introduced Tidal, a language designed for live coding of musical pattern. Tidal has already been field tested through several performances by the first author, including to large audiences at international music festivals, informing ongoing development of the system. The system will be tested further through a series of planned workshops with potential users, and full documented release of the code, which is already available in its present form at `http://yaxu.org/tidal/`. A research programme is planned towards the development of a Graphical User Interface for live musical pattern making, with Tidal providing the pattern language. Work is ongoing towards applying Tidal to the domain of live video animation, with current focus on colour transitions over time inspired by the inventions of Mary Hallock-Greenwalt [22]. As mentioned in §4.1.6, we hope that a website for sharing ideas and code for musical patterning will encourage connections between communities of musicians and programmers interested in pattern.

[3] Projecting the emacs interface as part of a live coding performance has its own aesthetic, having a particularly strong effect on many developers in the audience, either of elation or revulsion.

7. REFERENCES

[1] N. Collins, A. McLean, J. Rohrhuber, and A. Ward, "Live coding in laptop performance," *Organised Sound*, vol. 8, no. 03, pp. 321–330, 2003.

[2] A. N. Whitehead, *Dialogues of Alfred North Whitehead (A Nonpareil Book)*. David R Godine, August 2001.

[3] L. Spiegel, "A short history of intelligent instruments," *Computer Music Journal*, vol. 11, no. 3, 1987.

[4] R. Rowe, *Machine Musicianship*. The MIT Press, March 2001.

[5] A. Ward, J. Rohrhuber, F. Olofsson, A. McLean, D. Griffiths, N. Collins, and A. Alexander, "Live algorithm programming and a temporary organisation for its promotion," in *read_me — Software Art and Cultures* (O. Goriunova and A. Shulgin, eds.), 2004.

[6] A. Blackwell and N. Collins, "The programming language as a musical instrument," in *Proceedings of PPIG05*, University of Sussex, 2005.

[7] J. Rohrhuber, A. de Campo, and R. Wieser, "Algorithms today: Notes on language design for just in time programming," in *Proceedings of the 2005 International Computer Music Conference*, 2005.

[8] H. A. Simon and R. K. Sumner, "Pattern in music," pp. 83–110, 1992.

[9] F. H. Rauscher, G. L. Shaw, and C. N. Ky, "Music and spatial task performance," *Nature*, vol. 365, p. 611, October 1993.

[10] K. M. Steele, K. E. Bass, and M. D. Crook, "The mystery of the mozart effect: Failure to replicate," *Psychological Science*, pp. 366–369, July 1999.

[11] D. Deutsch and J. Feroe, "The internal representation of pitch sequences in tonal music.," *Psychological Review*, vol. 88, pp. 503–22, November 1981.

[12] L. Spiegel, "Manipulations of musical patterns," in *Proceedings of the Symposium on Small Computers and the Arts*, pp. 19–22, 1981.

[13] H. K. Taube, *Notes from the Metalevel: Introduction to Algorithmic Music Composition*. Lisse, The Netherlands: Swets & Zeitlinger, 2004.

[14] J. McCartney, “Rethinking the computer music language: Supercollider,” *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[15] A. McLean and G. Wiggins, “Petrol: Reactive pattern language for improvised music,” in *Proceedings of the International Computer Music Conference*, June 2010.

[16] A. Freed and A. Schmeder, “Features and future of open sound control version 1.1 for nime,” in *NIME*, 2009.

[17] A. McLean and G. Wiggins, “Bricolage programming in the creative arts,” in *22nd Psychology of Programming Interest Group*, 2010.

[18] G. Buzsaki, *Rhythms of the Brain*. Oxford University Press, USA, 1 ed., August 2006.

[19] J. Cage, *Art and Technology*. Cooper Square Press, 1969.

[20] C. Elliott, “Push-pull functional reactive programming,” in *Haskell Symposium*, 2009.

[21] B. Bel, “Rationalizing musical time: syntactic and symbolic-numeric approaches,” in *The Ratio Book* (C. Barlow, ed.), pp. 86–101, Feedback Studio, 2001.

[22] M. H. Greenewalt, *Nourathar, the Fine Art of Light Color Playing*. Philadelphia. Pa. Westbrook, 1946.