PETROL: REACTIVE PATTERN LANGUAGE FOR IMPROVISED MUSIC

Alex McLean, Geraint Wiggins

Intelligent Sound and Music Systems
Department of Computing
Goldsmiths, University of London

ABSTRACT

Humans infer patterns from musical sequences, perceiving them as repeated themes or processes of transformation. Computational means of representing and transforming patterns are reviewed, motivating the introduction of Petrol, a new live coding environment including pattern language embedded in the Haskell programming language. Petrol represents patterns as functions over time, and provides a combinator library for constructing and transforming those patterns, designed for use during live coded music performance. The

1. INTRODUCTION

A pattern is a theme of events repeating in a predictable manner. Humans predict forthcoming events in a patterned stream by inferring a process by which previous events may have been generated. It follows that the essence of a pattern is not a particular sequence, but the underlying process we infer from the sequence. This process then grounds our perception of further development of the sequence. Many approaches to the composition and improvisation of music are concerned with the processes of pattern, which motivates representation of pattern in computer music.

The desire to capture musical patterns with machines goes back to well before digital 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 [10]. Patterns may however be applied to any musical dimension; not just of pitch but also of time, of one of the dimensions of timbre, or indeed across several such dimensions, interacting or running in parallel. In this paper we are concerned with the description of such musical patterns with computer languages, primarily in the context of live coding [2, 12] where computer language serves as a human environment for the live patterning of improvised music. Many of the issues covered are potentially of general interest to the field of computer music; computational approaches to music analysis, indexing and composition all have focus on discrete musical events and the pattern rules which they conform to [8, §4.2]. To take best advantage of the present medium, examples are shown in the visual form of colour patterning.

2. PATTERN LANGUAGE

The term *pattern language* was coined in the field of urban design by Christopher Alexander [1]. Inspiration has been gained from this work across disciplines, but while insightful analogies between architecture and music have been drawn by artists, Alexander's architectural focus on problems and solutions does not lend itself easily to music. We could perhaps align Alexander's problem *forces* with those of musical expectation, but that would lead us too far into issues of human perception, outside of the scope of the present paper. The music languages discussed here provide building blocks for the construction and transformation of pattern, arrived at through analysis and introspection, but do not come with prescription for *where*, or even less *why* these might be used, at least not here.

A need for music pattern language was identified by Laurie Spiegel in her 1981 paper "Manipulations of Musical Patterns" [9]. Twelve pattern transformations, 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 modern commercial sequencer software such as Roland Cubase and Apple Logic Studio. However Spiegel is a strong advocate for the role of the musician programmer, and hoped these pattern transformations would be formalised into programming libraries. Such libraries have indeed emerged follow-

ing Spiegel's early vision. For example, the scheme based *Common Music* environment, developed from 1989, includes a well developed object oriented pattern library [11]. Classes are provided for pattern transformations such as permutation, rotation and random selection, and pattern generation such as Markov models, state transition and rewrite rules. The SuperCollider language [6] also comes with a well developed pattern library, benefiting from an active free software development community.

The pattern language within both Common Music and SuperCollider represent processes well explored in algorithmic composition, and programmers may integrate their own pattern operations. It is generally helpful to have these algorithms together in a coherent library, but the primary motivation for a pattern library is provision for *composition* of patterns. In both cases, patterns may be composed of numerous sub-patterns in a variety of ways and to arbitrary depth, to produce complex wholes from simple parts.

The representation of pattern streams used in both Common Music and SuperCollider are equivalent to *lazy lists* [5], with sequential access and delayed evaluation allowing efficient representation of long, perhaps infinite lists. This allows the representation of cyclic, fractal or pseudo-random patterns. In Haskell, lists are lazily evaluated by default, and so we could represent a pattern as follows:

```
type Pattern a = [[a]]
```

This declares a pattern to be a two dimensional list of any type `a`. The list is two dimensional to allow multiple values for each time element – polyphony. However a drawback in this representation, is that patterns must be accessed sequentially – you cannot directly request the 1000th value without first evaluating the first 999. This presents a problem for live coders, who may wish to replace a pattern with a new one, but continue at the position they left off. The same problem exists in Common Music and SuperCollider, although in the latter live coding is made possible using PatternProxys [7]. PatternProxys act as placeholders within a pattern, allowing a programmer to define sub-patterns which may be modified later.

3. PETROL

The Petrol pattern language is a library for the dialect of the Haskell programming language implemented by the Glasgow Haskell Compiler (GHC). Haskell is a purely functional programming language based on the lambda calculus, but with monadic modelling of 'real world' computation such as I/O side effects. The Petrol library is a domain specfic language embedded in Haskell, defining the Pattern type as a Applicative Functor instance and providing a suite of functions for constructing and transforming patterns. The Petrol environment is a complete live coding environment including a scheduler for combining patterns into OSC messages, time synchronisation between processes via *netclock* (`http://netclock.slab.org/`) and integration with the *emacs* editor. This paper however focusses on Petrol's provision for representation and combination of patterns.

Rather than representing patterns as lists of lists, as shown in the previous section, Petrol uses the following:

```
data Pattern a =
  Pattern {at :: Int → [a], period :: Int}
```

This is a datatype with two parameters. The first, given the fieldname `at`, is `Int →[a]`, a function from integers to lists. This acts as a function from a discrete time point to values co-occurring at that point; a temporal lookup function. The second parameter is the integer `period` of the pattern; the point at which the pattern repeats. The assumption of a finite period makes Petrol only suitable for cyclic patterns.

Patterns may be constructed and accessed as follows:

```
p = Pattern {at n = (n % 4) * 2, period = 4}
l = map (at p) [0 .. 16]
```

Where *l* would evaluate to the list:

```
[0, 2, 4, 6, 0, 2, 4, 6]
```

This representation of patterns as functions over time is strongly related to Functional Reactive Programming (FRP) [3]. FRP generally treats time as continuous, allowing behaviour to be described with a declarative function with arbitrarily high time resolution. Continuous time, represented as a floating point number, would make a great deal of sense if we were representing music on the physical signal level. However, patterns exist on a higher perceptual level of scored events, and so in Petrol time is instead represented using integers. This does not however rule out expressive manipulation of time, as we will see in §4.

3.1. Constructing patterns

A pattern may be specified as a string, made possible through a string overloading extension to GHC. The pattern type is inferred from the context, and the string parsed accordingly. For example the `draw` function requires a colour pattern, parsed into type `Pattern ColourD`, and renders a diagram visualising the pattern:

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

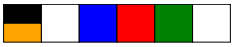
The `draw` and similar convenience functions are used to visualise the output of patterns through the remainder of this section. Syntax for specifying polymetric patterns is provided by Petrol, where co-occurring events are visualised here as vertically stacked colours. Sub-patterns with different periods may be combined in two straightforward ways, by repetition and by padding. In both cases the result is a combined pattern with period of the lowest common multiple of that of the constituent patterns.

Combining patterns by repetition is straightforward, and denoted by square brackets, where constituent parts are separated by commas:

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

Combining by padding each part with rests is denoted curly brackets, and inspired by the Bol Processor Bel01. In this example the first part is padded with one rest ever step, and the second with two rests:

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



Polymetries may be embedded to any depth:

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

There are other ways of constructing patterns, for example the `sine1` function produces a sine cycle of floating point numbers with a given period, here rendered as grey values with the `drawGray` function:

```
drawGray $ sine1 16
```

3.2. Pattern transformation

When the underlying pattern representation is a list, a pattern transformer must 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 Petrol, because a pattern is a function from time to events, a transformer may manipulate time values as well as events. Accordingly the Petrol function `rotL` for rotating a pattern to the left is 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 `every` function allows transformations to only be applied every *n* cycles. For example, to rotate a pattern by a single step every third repetition:

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

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 Petrol `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. Note the use of tildes to denote rests.

```
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 offset and/or velocity pattern from it.

4. MUSICAL APPLICATION

The colour pattern examples in the previous section should have given an impression of Petrol usage, however we must now turn the discussion back to music. In improvised music performance, Petrol patterns are used to control a variety of parameters.

A central time server controls tempo across multiple instances of Petrol, controllable via a `bpm` pattern. For example the following decreases the tempo from 120 to 60 bpm over each 16 beat cycle:

```
bpm $ tween 120 60 16
```

Besides controlling `bpm`, patterns are used to send Open Sound Control (OSC) messages [4] to synthesisers. A pattern may be defined and redefined for each synthesiser parameter while the music plays. A pattern `offset` specifies a time offset to be added to OSC timestamps. This allows OSC events to be scheduled in the future, or notionally in

the past, at least within the system-wide latency (defaulted to 0.2 seconds). For example, the following adds a subtle shuffling forwards and backwards by up to 0.04 seconds over a 24 beat cycle, combined with a steady beat (the `"0"`).

```
bpm $ combine ["0", (* 0.04) <$> sine 24]
```

The synthesisers currently controlled in Petrol performances are the datadirt wavetable synthesiser by the first author (`http://yaxu.org/datadirt/`), and nekobee (`http://www.nekosynth.co.uk/`), an emulator for the Roland TB-303 Bassline synthesiser. Many parameters may be controlled, including sample name, vowel formant filter, playback acceleration, waveshape, pan, combfilter, cutoff and resonance parameters for datadirt, along with the parameters for the TB-303 emulator. Simple patterns giving effects obvious to a listener are quick to specify, before being developed further, leading the listener into complexity. As a trivial example, to apply an oscillation to bassline note duration (thereby interfering with the TB-303 glide) we may quickly apply the following:

```
duration303 $ sine1 16
```

And then start building in some complexity:

```
duration303 $ (*) <$> sine1 16 <*> sine1 12)
```

Screencasts demonstrating Petrol are available online together with sourcecode for the whole system under the GNU Public License, at `http://yaxu.org/petrol/`.

5. CONCLUSION

Through terse, expressive syntax, Petrol allows fast application of patterns across many parameters, allowing rich musical texture to be built and manipulated during live performance. Petrol has already been successfully used before large audiences including club venues by the first author, and our particular approach of treating pattern as manipulation of behaviour has proved rewarding. However some desires have come to light through its use; these include OSC message structure easily configurable, more easily transforming OSC parameters together, allowing representation of non-cyclic patterns and finding alternative approaches to combining patterns. An extensive rewrite addressing these points, with the working title of *Tidal* is nearing completion, and will be detailed in future work.

6. REFERENCES

[1] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*, 1st ed. Oxford University Press, August 1977.

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

[3] C. Elliott, "Push-pull functional reactive programming," in *Haskell Symposium*, 2009.

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

[5] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 359–411, September 1989.

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

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

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

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

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

[12] 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.