Improvising with Synthesised Vocables, with Analysis Towards Computational Creativity

Alex McLean
Goldsmiths College
University of London
ma503am@gold.ac.uk

September 21, 2007

Abstract

In the context of the live coding of music and computational creativity, literature examining perceptual relationships between text, speech and instrumental sounds are surveyed, including the use of vocable words in music. A system for improvising polymetric rhythms with vocable words is introduced, together with a working prototype for producing rhythmic continuations within the system. This is shown to be a promising direction for both text based music improvisation and research into creative agents.

1 Acknowledgements

Special thanks to my supervisor Geraint Wiggins for providing great inspiration, support and guidance in good measure throughout this project. Likewise Frederic Leymarie, my fellow students from MSc Arts Computing, Tim Blackwell and all at ISMS for support and ideas, and for making the last two years so interesting. Thank you also Robert Atwood, Carl Forsell, Jamie Forth, Chun Lee and Ben Moran for your valued input.

Great gratitude to Adrian Ward, my long time collaborator. Having introduced me to the soul of generative music seven years ago we have been making software to make music to drink beer to together as 'slub' ever since, more recently joined and enriched by the extraordinary Dave Griffiths. This collaboration forms the basis of everything I do. I have also learned much from collaborations with Lottie Child, Cormac Heron and Alex Garacotche. Thank you Nick Collins for many things, including lighting the path in algorithmic dance music research ahead of me. Thank you Rohan Drape for all that HSC3 is. I must also thank all of TOPLAP, state51, openlab, dorkbot, runme, (void) and hashmoo, fine organisations to which I am honoured to have belonged. Thanks of course to my family Douglas, Valerie, Andrea and Stuart McLean. Andrea you have always been showing me what dedication to the world of imagination means, I hope that some of your painting is in my programming.

All the people I do not have space to mention, all those I have met, am yet to meet and who I have otherwise forgotten, thank you for the good times.

Most of all, thanks to Jessica Elmore, our son to be, and Olga and Teff for showing me all the joys of life outside of computation.

2 Introduction

The investigation below touches on a number of fields. The starting point is in live coding, where humans use formal language as a creative medium. We then examine a formal model of improvisation in terms of a general model of computational creativity. This sets the main context for our thesis, which is set at a prospective meeting point between live coding and computational creativity, being formal musical language.

We note that it is common for humans to map from words to instrumental as well as vocal articulations in music. We examine motor theory of speech perception, which has us perceiving not speech sounds but the articulations that encoded them. This suggests that software could gain much from dealing with musical sounds not as complex auditory signal but as articulations. This reduction not only allows more efficient processing, but also models certain musical forms in a way close to human experience.

From here we see great motivation for a system for improvising with synthesised vocables, finding an immediate and expressive system of representation as a bridge between human and computational creativity. Thanks to the existence of tools and models such as Karplus-Strong physical modelling synthesis, Levenshtein edit distance and the synthesis and parsing libraries within the Haskell programming language, we have been able to follow this motivation through to working software.

Video examples of the software, and the Haskell source code of the system itself may be accessed at `http://doc.gold.ac.uk/~ma503am/vocable/`.

3 Survey: Text, Speech and Improvised Music

3.1 Live coding

Live coding is defined as the *modification of rules while they are followed.* Typically, live coding places a programmer on stage, writing software before an audience, generating music and/or video. The live coder makes incremental edits to the sourcecode of a running program, enacting each modification with a hotkey. This is made possible through the use of *dynamic programming languages*, those allowing new and modified instructions to be *hot-swapped* at run-time, without requiring restarts or the loss of state. The live coder projects their screen so that the audience may see the source code develop while experiencing the musical or visual output of the running program.

Live coding has shown a recent surge in popularity with some academic (Collins et al., 2003; Ward et al., 2004; Blackwell and Collins, 2005; Sorensen and Brown, 2007; Zmölnig and Eckel, 2007) and media attention (Andrews, 2006; Uehlecke, 2006). Several international meetings to discuss and perform livecoding have occurred, most notably Changing Grammars in Hamburg in 2004 and LOSS Livecode in Sheffield in 2007.

Live coding is often related to generative music, which Eno (1996) describes as "...planting seeds into your computer, and then using the computer to grow those seeds for you". In contrast, live coding has the human growing the seeds, using the computer as assistant. Here the metaphor with gardening really falls apart, but we can try to imagine live coding a plant by modifying its genetics while it grows. The art then would not be in the resulting plant but in its life of growth as closely controlled by the live coder.

There are several approaches to live coding, from the gradual building of a single function, to execution of individual blocks of source code in order to modify a synthesis graph, to the editing and reinterpretation of whole classes in order to change the behaviour of objects in memory. Through practice, live coders build a repertoire of algorithmic techniques to employ during a performance. By reflecting on their use of the *Impromptu* software, Sorensen and Brown (2007) provide an excellent overview of the kind of algorithms a live coder may employ, with reference to music theory. They detail their use of probabilistic, polynomial and periodic functions, set theory and recursion in both synthesis parameters and compositional structures.

```
$self->t({sample => 'gabbalouder/0',
          volume => $vol / 1000,
          speed  => 1,
          delay  => rand(0.03),
          delay2 => rand(0.03),
          loops => 1,
          shape  => 0.99,
          cutoff => $cutoff,
          resonance => $res,
         }
       );
```

Figure 1: Perl code to play a distorted kick drum sample

Live coding may not necessarily involve the use of computers. For example, a leading neo-folk musician known as 'Adem' arranges musical improvisations that he calls 'Assembly', where tens of musicians improvise music together on acoustic instruments. The rules of the assembly are written on a blackboard while the performers play, each rule enacted by a hand signal. During such a performance at the Shunt venue in London on the 15th June 2007, a few musicians took turns adding and modifying rules, which at times took the form of a pictorial score and at others included alternation — "Long notes, then short notes + repeat" and a conditional — "stamp your feet if you feel like it".

3.1.1 Live Coding as Improvisation

If we are live coding a piece of music before an audience, then we can say we are *creating a musical work, or the final form of a musical work, while it is being performed.* In fact, this is a definition of improvisation from the New Grove Dictionary (Nettl, 2001). From this we infer that live coding is improvising with code.

Nonetheless, there are barriers, both real and imagined, to fully accepting a live coder as an improviser. Leaving aside the imposing social norms and stereotypes that have developed around programming culture, one of the biggest problems faced by an improvising live coder is time. While in some sense it is true that live coders have full control over every sound they make, they may take minutes to describe one. In contrast a traditional instrumentalist may introduce a new sound in a single physical movement. Figure 1 illustrates the overhead in typing time alone. In effect, the live coder is building an instrument while trying to play it — no wonder their response times may at times be long. More detailed exploration of this problem with reference to human haptic rates is provided by Collins (2006).

What the live coder may lose in instant expression, they can attempt to gain through higher order expressivity. That is, the live coder works not only with individual sound events but with the musical structures they describe. Further, properties shared by a series of sound events may be abstracted into computer language constructs such as functions, operators or object classes. Constructing these abstractions is analogous to constructing an instrument, and a change in an abstraction affects all the sounds generated from them. These abstractions may be specified during practice and development sessions, in order to greatly improve response times during performance.

These techniques of abstraction are of course common throughout computer science. There is also much to learn from review in other fields. §3.5 examines how instrumental sounds may be represented in the field of music, exemplified by tabla and bagpipe notation.

3.2 Improvisation as a Creative System

Wiggins (2006a,b) provides a framework for describing and reasoning about creative systems, based strongly upon the work of Boden (1990) but with some important additional clarifications.

The framework terms a creative system as “a collection of processes, natural or automatic, which are capable of achieving or simulating behaviour which in humans would be deemed creative.” Here we term the framework the *creative systems framework*, abbreviated to *CSF*.

The CSF makes use of the following symbols and functions

c A concept

U A universe of all possible concepts

L A language in which rules may be expressed

R Rules defined within L defining validity of a concept

$[[R]]$ A function interpreting R, resulting in a function indicating adherence of a concept to R

C A conceptual space, defined by $[[R]](U)$

T Rules defined within L to define a traversal strategy to locate concepts within U

E Rules defined within L which evaluate the quality or desirability of a concept c

$\langle R, T, E \rangle$ A function interpreting the traversal strategy T, informed by R and E. It operates upon an ordered subset of U (of which it has random access) and results in another ordered subset of U.

It is worth distinguishing these terms further. First, between membership of a class of thing, and *valued* membership; for example we might recognise an artefact as conforming to all the syntactical rules of a limerick, but not be funny. Accordingly, R governs membership of concepts to the limerick class whereas E defines rules which assign value judgements of these concepts. In contrast, T does not describe anything about the artefact, only how a particular agent may find it.

3.2.1 Creative behaviour

We may now use this notation to describe different forms of creative behaviour. Simplest is to recursively apply $\langle R, T, E \rangle$ until a desired concept is found within C, perhaps starting with the empty concept $\top$. Given a C with valued (according to E) yet undiscovered concepts, the likelihood of success depends entirely on the ability of T to navigate the space.

However, applying $\langle R, T, E \rangle$ may locate one or more concepts within U that do not conform to R and are therefore not members of C. Such a result is termed an *aberration.*

Concepts in an aberration which do not conform to R may still conform to E and hence be valued. If all such concepts are valued then we have *perfect aberration* and R should be transformed to extend the conceptual space so the new concepts are included. If no such concepts are valued then we have *pointless aberration*, and T should be transformed so that the concepts are avoided in the future. If some such concepts are valued and others not, then we have *productive aberration* and both R and T should be transformed.

3.2.2 Improvisation

Following a broad review of research into music improvisation from viewpoints including physiology, neuropsychology, folklore, human intuition and artificial intelligence, Pressing (1987) provides a formal model of human improvisation. For the purposes of this dissertation, Pressing’s model is termed the *Improvisation Model,* abbreviated to *IM.* While the IM is based on research into *human* improvisation, and indeed is introduced with the bold heading *“How people improvise”*, Pressing puts the IM forward as useful in the design of improvising computer agents.

We cannot necessarily expect an easy, direct mapping between the symbols of the CSF and of the IM. However Pressing (1987) does discuss creative behaviour in the context of the IM, and indeed the CSF is in part designed to help us understand and discuss such models. Before we compare them, we must examine the IM itself.

3.2.3 The improvisation model

The IM is described using the notation summarised below. Note the prime (') suffixes do not appear in the IM, and are only used here to distinguish from notation used in the CSF.

E'	A cluster of sound events
K'	A sequence of E' event clusters, where event cluster onsets do not overlap with those of a following one
Interrupt	A trigger to curtail the current K' sequence and begins a new one.
I'	An improvisation, partitioned by *interrupts* into a number of K' sequences
R'	An optional referent, such as a score or stylistic norm
G'	A set of current goals
M'	Long term memory
O'	An array of *objects*
F'	An array of *features*
P'	An array of *processes*

Object, Feature and Process arrays A particular E' is constructed with reference to its own set of three arrays; O', F' and P'. Each array lists a number of factors according to its type, in particular

- O' lists the objects, cognitive or perceptual entities such as chords, notes or rests
- For each entry in O', F' lists the available feature parameters, describing shared properties of an object, such as pitch or modulation
- P' lists the processes which govern how an object or feature changes over time, such as "use trichords", "follow contour" or "randomly select notes from scale"

Each factor in each array is explicitly given a *cognitive strength*, weighting the factor according to the importance or level of attention given to it by the improviser. This strength governs the amount of influence each factor has over the result, so that a factor with no influence carries a cognitive strength of 0, and one with high influence a strength towards 1.

A particular configuration of the O', F' and P' arrays maps to a particular E' cluster of events. We can therefore reduce the problem of improvisation from generating individual events to generating strengths within array configurations. This reduction is key to the power of the IM, where decision making is based upon event clusters and not individual events.

So cognitive strengths of array elements are modified to develop an improvisation, but where do those elements come from in the first place? Pressing (1987) suggests a model based on ecology where elements of O' are inferred by invariance in sensory input over time and musical space, F' by similarity or contrast in sensory input, and P' by change in an object or along a feature dimension with time. This is long-term learning, where few changes if any are made during a given performance, but Pressing suggests that novel behaviour results. This latter claim is made in the context of references to metaphor, so we assume that novel behaviour results from combining concepts learned from different sources. In any case we should note that learning may be helpful to creative systems, but is not in itself a form of creative behaviour.

Pressing goes on to describe a form of what Boden later termed *combinational creativity*, giving an example where two parameters in different dimensions namely *soft* and *fast* combined to produce novel *soft-fast* behaviour.

Continuation and interrupts A *continuation* is the generation of new O', F' and P' arrays in order to generate the next event cluster. The IM describes two types of continuation, associative and interruptive. Associative continuation proceeds with arrays either similar to or straightforwardly contrasting that of the previous E', while interruptive continuation involves *resetting* of many factors within one or more of the arrays. Pressing (1987) refers to an interruptive continuation simply as an *interrupt.* Each interrupt creates a partition, ending one K' and beginning a new one. With either method, the result is a new set of arrays determining the construction of a new E'.

Interrupts are triggered with reference to a tolerance for repetition compared to the period of time since the previous interrupt and the nature of the current K'. The size of K' is considered along with the strong O', F' and P' components shared by its member event clusters.

Pressing (1987) provides examples for continuations both within the context of a melodic piece and a rhythmic piece. Each example is characterised by the arrays acted upon and the associative or otherwise interruptive nature of those acts. For example *"notes E, A, D; rhythmic displacement"* describes an associative continuation operating on O' and P' while *"toss lead shot in air and catch it"* describes an interruptive continuation operating on F' and P'.

3.2.4 The improvisation model as a creative system

Now we consider how we may view the IM as a creative system, and what insights may we gain in the process. We present how the notation of the CSF might be applied to the IM before visiting and explaining each notated aspect in turn.

U	The universe of possible improvisations, including incomplete improvisations. Non-specific so that a concept within a music improvisation may be related to one within another field, for example a dance
c	An improvisation, as a timed sequence of of cognitive strengths within the object, feature and process arrays of an instance of the IM
L	Language rules for describing and defining cognitive strengths of objects, features and processes arranged into arrays
C	The musical action space, being the set of possible improvisations conforming to the referent
R	A referent R' giving a theme, motive, mood and/or score to which the improvisation should relate. May also include a set of stylistic norms, either implicitly or explicitly
T	A method for generating continuations, which modify or reset cognitive strengths
E	Rules regarding the suitability or value of a given improvisation

c **- The improvisation within on-line operation** A major consideration is that the IM requires *on-line* operation, being as Pressing (1984) puts it a "problem solving activity that does not allow editing." The CSF allows for partial concepts, and so we may consider each traversal operation as a continuation; generating a new partial concept that includes what has gone before.

L **- Language rules** The language structure of the IM is that of the Object, Feature and Process arrays, containing factors governing event generation each with a number denoting its cognitive strength. Object and Feature arrays consist of straightforward labels but process arrays contains functions such as "Randomly select notes from scale", "follow contour" and "use contours". A formal method for defining such functions is not provided.

C **- Musical action space as a conceptual space** We may strongly relate the conceptual space C with Pressing's portrayal of a *musical action space* (Pressing, 1987), where a particular point E' is shown as a certain configuration of the arrays. A sequence K' inherits a point in the action space from its component E' event clusters, being the point they share or deviate from.

Two different K' clusters occupying similar positions are therefore deemed to share a theme. An improvisation with the structure ABA'C is shown to revisit the neighbourhood of the first section in the third section before proceeding to the fourth.

R **- A referent defining a musical action space** Defining the musical action space initially seems troublesome. There appear to be many influences over it including long term goals and the stylistic norms that are employed. However here we focus upon what Pressing (1984) terms the referent, characterised as structures such as a theme, motive and/or score, in other words a set of pre-conditions. We extend Pressing's definition to include stylistic norms, because such scores or motives are often cast within the context of a particular style even if that style is not explicitly included.

It is important to note that if a referent contains a time dimension, such as a score with a time axis, then so will the resulting conceptual space. That is, as well as all the other constraints defined by R, certain concepts will only be permissible during certain time periods within the improvisation. It would then be up to the traversal strategy to examine the slice of the conceptual space along its time axis relevant to the current position in the improvisation.

T **- Continuations as traversal strategies** The continuation of the IM is strongly related to the traversal strategy within the CSF. However the IM describes a number of possible strategies for continuation whereas the CSF requires a single traversal strategy. If a traversal strategy is a function, and each continuation can be likewise formalised as a function, then we need a way of uniting these continuation functions into a single traversal function.

This is the problem Pressing (1987) alludes to as residual decision-making. He characterises a number of explanations for how humans make such decisions, namely models based on intuition, free will, physicalist interaction with the environment and randomness. Indeed, this brief review of explanations for decision making is reminiscent of Boden (1990)'s review of explanations for creativity. However, we are not concerned directly with human creativity but with a model for modelling human-like creative behaviour. We can be content then with a formal model where a continuation is chosen for example in an IF-THEN-ELSE manner.

The IM places great emphasis on the influence of motor control over an improvisation. It posits that the improviser generates some ideal sequence of events which they can only realise to the limits of their training which is never perfect. Their imagined sequence is therefore tempered by what is realised, as observed by their sensory apparatus, before being fed back into the generation of the next sequence. Pressing (1999) later presents a model for such referent based human behaviour in detail and with supporting experimental evidence as *Referential Behavior Theory.*

We can view such motor control errors as analogous to the aberration classes from the CSF, as follows. Perfect error - a guitar player following a line to a 'wrong note' which does not fit the stylistic norms but somehow sounds good within the context, taking the theme somewhere else without requiring some part of it to be dropped. Productive error - for example where that note has an interesting effect, but clashes with some other aspect of the theme which should be discarded. Pointless error - the note sounds bad and leads nowhere.

We could model such behaviour by careful inclusion of a pseudo random number generator in our traversal strategy. Careful because such physical errors occur within certain constraints, for instance if we pluck a string by mistake, it would likely be one adjacent to that intended. Another approach would be to use electric motors with sensory feedback in an improvising system, where the inaccuracy of the motors and environmental interference would colour the output of the improvisation.

E **- Feedback evaluator** Pressing (1984, p353) describes an *evaluation processor* as the primary source of feedback, judging output in the context of "goals, previous training, what has gone before, and the sounds produced by other musicians." A direct analogy can be drawn with the E rules of the CSF, which operates on a concept to produce a number reflecting its suitability.

It is not clear however whether such evaluation is only operated on actioned events, or whether some auditioning is allowed. That is, whether events can be evaluated and discarded without being put into action. It would seem possible that several concepts may be considered at each point, and the most suitable chosen. This is suggested by Pressing, and is certainly possible within the CSF, which allows several concepts from an agenda to be considered at once.

3.2.5 Discussion

The above analysis highlights a number of areas for consideration while translating the IM to a working computer model. Firstly, one of language; how can we formally describe the Objects, Features and Processes?

Another key question is that of how traversal strategies are formed. We have understood what a continuation may look like, but not clearly how one may be chosen over another.

Whereas Pressing may be correct in asserting that new array entries are rarely formed during an improvisation, we nevertheless need to understand how this occurs when it does, as such modifications are key to creative behaviour, amounting to transformation of the traversal strategy T. A fuller application of the CSF would also include transformations to the rules R, perhaps to model how a score for a new improvisation may be derived from an old one. It may also consider how E should be transformed so that the system can modify what it considers good and bad with experience.

Broadly speaking, we have shown that the IM may be described within the CSF. Realising this has helped clarify the IM and highlight some areas for development. In transferring the IM to the notation of the CSF we may consider music improvisation in a clearer manner and have a common language in which to compare it with other models.

3.3 Perception and Production of Speech

Speech production can be modelled as sound *excitation* plus sound *filtration.* The excitation comes from *vocal folds* being drawn together with a force to cause oscillation between open and closed by continuous breath pressure from the lungs, the frequency at which depends on muscle controlling the tightness of the folds. The filtration comprises of several articulators including the tongue, lips, jaw, velum and larynx, producing sounds in such categories as closed *fricative* sounds and open *vowel* sounds (Cook, 1999a). Their combined positions provide an enormous range of sounds which may then be combined as *articulations* producing further compound sounds such as dipthongs. Some portion of the possible range of sounds are utilised in a particular human language, with variations as dialect.

The *motor theory of speech perception* (Liberman and Mattingly, 1985) posits that the production and perception of speech are inextricably linked, to the extent that we perceive speech not as an auditory signal but as a sequence of articulatory gestures. Indeed experiments have shown human subjects unable to learn a new set of phonemes constructed from non-vocal sounds (Cook, 1999b), suggesting that in order to decode a complex vocal message it must first be simplified as a sequence of gestures related to our own vocal apparatus.

Auditory illusions support motor theory, for example the McGurk effect, where lip-reading the word 'ga' while simultaneously hearing the word 'ba' commonly produces the perception 'da' (Cook, 1999b; Banks, 2004). This illusion is made all the more uncanny when the human test subject finds that opening and closing their eyes changes the sound that they perceive as hearing. That we perceive a sound half way between the two sounds suggests an internal physical model decoding the speech signal.

Liberman and Mattingly (1985) describe motor theory in terms of a neurological 'module' responsible for both speech perception and production, set in competition with more general auditory perception. If true this has serious repercussions to musicians interested in using speech and speech-like sounds in their work. As noted by Jones (1987), where humans attend to the verbal information carried by speech, we are to some extent deaf to the complex sounds that carry it. We can contrast then phonetic perception with nonphonetic perception, the difference between perceiving the *form* or the *content* of a message encoded as speech.

3.4 Articulatory speech synthesis

Articulatory speech synthesis involves construction of a physical model of the vocal tract, and methods for controlling the voice (excitation) and articulators (filtration) within the model. The sheer complexity of the vocal tract has hindered its understanding and modelling and so much research has instead focussed on *concatenative synthesis*, based upon audio sampling of diphones from particular human voices. We can say then that articulatory synthesis models the human vocal tract whereas concatenative synthesis samples it. However speech production is characterised not only by a physical model but also its control processes. Much can be done to control such prosodic parameters as intonation, rhythm and stress in concatenative synthesis, but to have full continuous control a physical model is required. Similarly, voice character parameters such as gender and age can be adjusted continuously as physical parameters, but in concatenative synthesis require a new set of diphones recorded for every new age and gender combination.

3.5 Non lexical vocables

A *non-lexical vocable* is a written, spoken or sung word that has musical but not wider semantic meaning. In this thesis the term is shortened simply to "vocable".

The use of vocables is a global phenomenon, for example occurring in Chassidic Jewish tunes, Japanese *Kakegoe*, Indian *Bols*, Northern American *Scat singing* and Gaelic *Diddling* (Chambers, 1980). Vocables have found broad uses such as; pedagogy, mnemonic aid, performance, dancing, musical experimentation and notation.

Chambers (1980) classifies vocables within two categories. The first distinction is between *improvisatory* and *jelled* vocables. An improvisatory vocable is one created by an individual performer, albeit from an inventory of accepted sounds. A jelled vocable is one prescribed by convention or transcription. There is of course a grey area between these absolutes, where prescribed vocables may be improvised with to a degree.

The second distinction is between *imitative* and *associative* vocables. An imitative vocable is one intrinsically similar to the instrumental sound it represents. By contrast an associative vocable is assigned to an instrumental sound or gesture without sharing significant structural or aural features. In practice this distinction is difficult to make, as Chambers (1980)[p.13] reports: "Occasionally a piper will say that a vocable is imitative (indigenous evaluation) when analysis seems to indicate that it is actually associative (analytic evaluation) because he has connected the vocable with the specific musical detail for so long that he can no longer divorce the two in his mind." This also occurs in onomatopoeia, for example an Englishman may hear a hen say "Cluck", while his German neighbour will likely perceive the same sound as "Tock tock" (de Rijke et al., 2003). This again points toward the strong relation between words and perception seen in §3.3.

3.5.1 Gaelic vocables and Canntaireachd

Canntaireachd is a system of vocables used in bagpipe music, as opposed to *Diddling* which is the more general use of "nonsense words" in Gaelic song.

As transcribed in forms such as in Figure 2, Canntaireachd represents only certain aspects of music, in particular the structure of a piece and the phrasing within it; the emphasis, stresses and rhythmic patterns. We should not then underestimate the importance of the oral tradition in Canntaireachd, as Chambers (1980, p.68), warns "As written notation Canntaireachd can convey a skeleton of the melody and ornamentation of the Pibroch, but without a tutor to flesh out the

Hinen hoen himen hioen, hinen hoen heen cheen, hinen hoen himen hioen,
hinen cheen hoen hinen, (repeat)
Hinen heen cheen cheen, hinen hioen hioen hioen, hinen heen cheen cheen,
hinen cheen hoen hinen,
Hinen heen cheen cheen, hinen hioen hioen hioen, hinen hoen himen hioen,
hinen cheen hoen hinen.

Figure 2: Excerpt of Var. 1 of "The Cave of Gold" in Nether Lorn Canntaireachd

musical bones, with a traditional interpretation, the written Canntaireachd is to a large extent useless".

Through extensive analysis, Chambers (1980, 117-121) identifies a syllabic structure for Canntaireachd vocables, found to be quite distinct from diddling. Six syllabic formulae were found in Canntaireachd; V, CV, CVV, VC, CVC and CCVC, where C represents a consonant syllable and V a vowel syllable. The latter form, featuring a pair of consonants, is rare. Indeed, clusters of consonants in vocables throughout Scottish traditional music are rare. The vowel syllables tend to relate to pitch, and consonant syllables to gesture. Chambers (1980) further identifies extensive formal grammar rules describing which pairs of sounds may follow together.

3.5.2 Bol Syllables in Indian Tabla

In Indian drum and dance music, vocables are known as *Bol* syllables, which are assigned to strokes and steps. Like Canntaireachd, Bols are an historic oral tradition, but in recent times systems of notation have been developed. In this section we focus on Bol syllables in the context of the tabla drum.

Lacking a pre-existing notation system with a level of detail suitable for his study of tabla playing, Kippen (1988) proposes a notation system using phonetically written syllables and a minimal set of symbols. The notation system achieves simplicity by not encoding every aspect of dynamics and timbre, instead focussing on groups of sounds. Nonetheless the phonetic basis of the system avoids ambiguity present in other systems such as those based on staff notation (Kippen, 1988, p.xvi). A brief summary of features of the notation system follows.

Each bol syllable is associated with both an articulation and its associated sound. For example *ṭe* relates to a non-resonating stroke with the 1st finger on the centre of the dāhinā right hand tabla drum.

Two or more bols may be grouped together into one part. For example *dhāti* where *dhā* and *ti* are played in quick succession over a single beat.

A pause is indicated by a dash:
`dhā - na`
Bols that are by convention played but not spoken are enclosed by square brackets:
`dhā[ge] - dhī nā`
Bols that are optional are enclosed in parenthesis, providing an alternative piece:
`dhā(gena) dhāgedhin`
Sections that are to be repeated three times, known as *tihā'īs*, are surrounded by vertical bars:
`| kṛa dhā tī ghin - ta dhā |`
Further mnemonic symbols are described which may be placed above or below a bol to further modify how a stroke should be played.

3.5.3 The Bol Processor

Bernard Bel, through close work with John Kippen has developed computer software for the transcription and analysis of tabla rhythm named the *Bol Processor*. Seeing the expressive power of textual representation of instrumental sounds, he developed further versions for the *generation* of music. While version two of this system retains clear influence from tabla, particularly in its hierarchical, polymetric structuring of time, it has been generalised to allow composition in practically any style of music. This software is a fine example of how expressive textual representation of music can be, and is a key inspiration for the software underlying this thesis.

3.5.4 Sound poetry

Spoken vocables are not always representative of other sounds, for example they are central in what is known variously as *abstract*, *concrete* or (as here) *sound poetry*. Here vocables are used to construct poems without lexical meaning. This is a large area, McCullough (1989) providing a bibliography running to over a thousand pages.

The Ursonate by Kurt Schwitters is a particularly fine example of sound poetry. A short excerpt is shown in Figure 3, although the full poem is arranged in four movements over a total of twenty nine pages, a reading of which takes around half an hour (Elderfield, 1985, pp. 175-176). Schwitters would perform his Ursonate from memory, an intricate piece composed entirely of vocables with German intonation, arranged in rhythmic themes permeated with hisses, roars and crowing. He would often perform to naive audiences who would at first not know how to react but eventually be reduced to howls of laughter, to which Schwitters would respond by increasing the volume of his voice still further (Crossley, 2005, p. 18). A final version of the poem was published as the final edition of Merz magazine (Schwitters, 1932).

```
dll rrrrr beeeee bö
dll rrrrr beeeee bö fümms bö,
     rrrrr beeeee bö fümms bö wö,
             beeeee bö fümms bö wö tää,
                        bö fümms bö wö tää zää,
                             fümms bö wö tää zää Uu:
```

Figure 3: Excerpt from the Ursonate by Kurt Schwitters

Some sound poets take things further by rejecting linear transcription as well as lexical meaning. For example Cobbing (1970) arranges words in a two dimensional pictorial fashion, sometimes using fragments of words, sometimes obscuring words completely through overtyping. Despite the nature of the written form of the poem, he would nonetheless use them during poetry readings.

3.5.5 Vocables in computer music

Given the strong relationship between speech and music discussed so far, it is not surprising that speech also has its bearing on computer music. Two artists are highlighted as examples; David Evan Jones from the field of electroacoustic music, and Chris Jeffs from the field of intelligent dance, or braindance music.

Jones (1987, 1990) discusses what he terms “speech like timbre”, equivalent to our use of the word *vocable*. In his own pieces “Still life dancing” and “Scritto for computer tape” he uses the CHANT software (Rodet et al., 1984) to apply vowel like quality to instrumental sounds to allow us to perceptually relate them to speech like sounds. The listener is then able to focus on the other timbral qualities of speech and appreciate them as part of the music.

Chris Jeffs, who composes and performs music as Cylob, has developed a speech synthesis system which features heavily on his “Formant Potaton” album, released in 2007. His speech synthesis features just two low pass and one high pass filter, with each phoneme assigned parameters found from spectral analysis of Jeffs’ own voice. This speech synthesis is one component of the Cylob Music System (Jeffs, 2007), developed by Jeffs in Supercollider (§3.8.1) since 2001, with which Jeffs now composes all his music. While it is possible to identify a song title where it is repeated as lyrical motif, only occasional lyrical fragments are otherwise discernable. We would therefore classify this system as vocable rather than speech synthesis. Perhaps because the speech

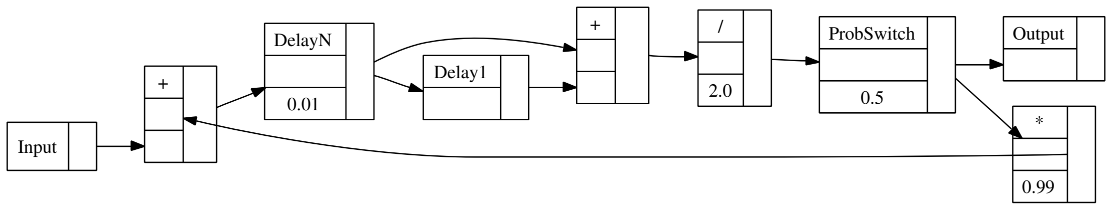


Figure 4: Percussive Karplus-Strong Synthesis

synthesis is from the same hand as the rest of the synthesis within the music, the vocable sounds blend well with the ‘instrumental’ sounds.

We feel that both Jones and Jeffs are successful in using vocal-like sounds to take advantage of human psycholinguistic classification abilities, while avoiding or obscuring semantic meaning to place the focus on timbral qualities.

3.6 Karplus-Strong Synthesis

Karplus and Strong (1983) defined a physical model for a plucked string, known as the *Karplus-Strong* algorithm or *loop filter*. The algorithm is computationally cheap, in its simplest form consisting of little more than a delay loop and an averaging filter, so that even at the time of its inception could be calculated in real time on commodity hardware.

Karplus-Strong synthesis is perhaps the simplest form of waveguide synthesis, being a model of a wave travelling along a section of a ‘perfect’ string or tube. Much work has gone into extensions to the algorithm (Jaffe and Smith, 1997) and into more complete waveguide models (Cook, 2002). However within the scope of this thesis a simple Karplus-Strong loop filter suffices; while there may only be two control parameters to the model, they may be controlled continuously to produce a wide range of expressions.

Figure 4 illustrates an implementation of the Karplus-Strong algorithm in HSC3 (§3.8.1). Each box represents a *unit generator* in the language. The *input* is a burst of white noise, long enough fill the wavetable *DelayN*. In this example the wavetable is one hundredth of a second long, specified by the *0.01* parameter.

The three unit generators, *Delay1*, *** and */* work together to average each sample with the previous one, forming a lowpass filter. This models a ‘dampening’ of the string.

ProbSwitch inverts the sign of samples with a probability of *n*, called the *blend factor*. A blend factor of around 0.5 stops the wavetable acting as a periodic resonator, removing tonal qualities so that the sound is more like that of a drum. The signal is multiplied by 0.99 so that energy is reduced before being fed back and added to the input signal.

3.7 Analysis of vocable text

3.7.1 Edit distance and minimum cost path

We aim to represent sounds with word-like strings of characters, so that the *morphology* of a string has perceptual significance. If successful we may apply techniques for measuring the similarity of strings as a measure of the sounds they represent. One such technique is *edit distance*, being the minimum number of edit operations required to modify one string to become another.

There are several edit distance measures which differ only in the edit operations that are allowed. For example, *Hamming distance* (Hamming, 1950) only has a single ‘replace’ operation and therefore may only be used to compare two strings of the same length. However as well as replace, *Levenshtein distance* (Levenshtein, 1966) allows ‘add’ and ‘delete’ operations, so strings of arbitrary lengths may be compared. Damerau-Levenshtein distance (Damerau, 1964) further adds a ‘transpose’ operation, for swapping adjacent positions in a string.

Levenshtein distance is used within this project for its simplicity, where dynamic programming techniques may be employed for calculation. In particular, we use an algorithm identified by Allison (1992), optimised for languages with lazy evaluation such as Haskell.

Before we apply the edit distance, we must choose the unit elements of the strings that we are editing. For example when measuring edit distance between strings, we may choose to apply the operations to letters, or combinations of letters such as syllables. We might also choose different weightings, for example to measure edit between two vowels or two constants of having a lesser cost than between a consonant and a vowel. However adding such weightings complicates matters, breaking the assumptions of the efficient algorithm described above so that it no longer applies.

Related to edit distance is the *minimum cost path*, a sequence of edits to go from one string to another. In the case of Levenshtein distance, it follows that halfway along a minimum cost path of edits, we find a new string of equal similarity to the two original strings.

3.7.2 Markov chains

Markov chains may straightforwardly be applied to vocable text, for example by building a statistical model of the likelihood of one vocable following another. Such a statistical model can then be used for generating new compositions. An illustrative example of a first order Markov chain of the poem excerpt in Figure 3 can be seen in Figure 5.

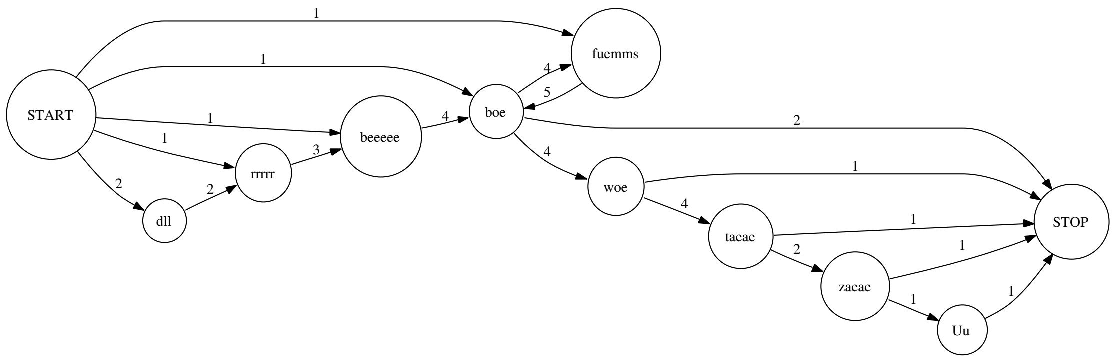


Figure 5: Markov chain of an excerpt of the Ursonate by Kurt Schwitters

3.8 Haskell

Haskell as a programming language has been most recently standardised as Haskell98 (Jones, 2002), with ongoing work on Haskell' (pronounced *Haskell Prime*) which will standardise many of the extensions existing in popular implementations. The term Haskell is used in this thesis to describe Haskell98, plus extensions implemented by GHC version 6.6.1.

According to the its homepage at `http://www.haskell.org/`, Haskell is a general purpose, purely functional programming language featuring static typing, higher order functions, polymorphism, type classes, and monadic effects. The following paragraphs examine these features in order.

General purpose Despite being an active centre of computer science research, hosting language features unavailable in more mainstream languages, Haskell is nonetheless in active use in a number of domains. Conversely, Haskell is also lauded as an excellent host to *domain specific language*, due to its clean, highly extendable syntax.

Purely functional With few exceptions for debugging purposes, Haskell only allows *pure functions* which may not produce side effects. That is they take input and return output, but may not otherwise engage in any other I/O or modify any external data structure. As a result Haskell's functions are guaranteed to produce the same output every time they're executed with the same input.

Static typing A statically typed language is one where the types of variables are set at compile time. Haskell's type system is featureful, for example the support for *type classes* allows consistent interfaces across types. For example a new type may be accepted to the *Ord* class with a definition of the *compare* function. Once that is done all of the sorting functions which operate on *Ord* may be used with our type.

Specifying the type of every variable and function can be tiring for a programmer, leading to criticism of some statically typed languages (such as Java) for requiring it. Haskell however has advanced *type inference* so that type information can often be omitted in the sourcecode. Nonetheless it is recommended to include type declarations for functions in general; types go a long way to describing what a function does, and can clear up many programmer errors at compile time.

Higher order functions Support for higher order functions is an important feature of any functional language. A higher order function is one that is parameterised by or returns one or more other functions, allowing powerful abstractions.

Polymorphism and Type classes Polymorphism allows unified interfaces across different types. Haskell supports polymorphism through type classes – a type may be declared a member of a class only if it provides a set of functions with names and types defined by that class.

Monadic effects Monads came to computer science from category theory, and have revolutionised pure functional programming by allowing sequences of actions to be described. This has led to such technologies as monadic parser combinators seen in the excellent Parsec module, and elegant pure functional solutions to such problems as error handling and I/O.

Earlier we noted that pure functions cannot perform I/O. However using monads they can describe a sequence of I/O actions to be performed by the caller.

Monads are also used to construct synthesis graphs within HSC3, as described in the following section.

3.8.1 Haskell and Supercollider Server

Supercollider is a computer language designed for music, featuring thorough support for object oriented programming, a realtime, low latency architecture, and excellent libraries to aide synthesis and composition. Supercollider version three features a client-server network architecture, where the supercollider language interpreter is a client to the synthesis engine server. This separation of concerns allows several advantages detailed in McCartney (2002), but also allows alternative language clients to interact directly with the supercollider synthesis engine.

Drape (2007) has released a library which provides Embedded Domain Specific Language (EDSL) for communicating with SuperCollider Server. This allows synthesis graphs to be specified directly as Haskell source code, for example the synthesis graph illustrated in Figure 4 may be specified as follows:

```
ks i = do let (gain, delay, blend) = (in' 1 KR (offsetU i 15),
                                      in' 1 KR (offsetU i 17),
                                      in' 1 KR (offsetU i 18)
                                     )
              laggedDelay = lag delay (in' 1 KR (offsetU i 16))
              laggedBlend = lag blend (in' 1 KR (offsetU i 16))
              vFilter i f a b = resonz i f (b / f) * dbAmp a
              n = sinOsc AR 200 0
              a0 = decay (impulse AR 200 0) 0.0025 * n
              a1 = (localIn 1 AR + (a0 * gain))
              a2 = delayN a1 0.01 laggedDelay
              a3 = delay1 a2
              a4 = (a2 + a3) / 2.0
              a5 = probSwitch a4 blend
              a6 = mix (vFilter a5
                        (in' 5 KR (offsetU i 0))
                        (in' 5 KR (offsetU i 5))
                        (in' 5 KR (offsetU i 10))
                       )
          return $ MRG [localOut (a5 * 0.99), out 0 (MCE [a6, a6])]
```

Both Supercollider and the Haskell Supercollider library are released using a GNU Public License, allowing free use and promoting sharing of sourcecode.

4 System for improvisation of rhythms with vocables

The software underlying this thesis is the result of two main research motivations.

The first motivation is to develop a system where rhythms formed of vocables may be improvised in text form. To this end we provide a system for controlling physical synthesis with vocables within a polymetric syntax.

The second motivation is to investigate machine learning of structure behind textual descriptions of music, with a working prototype of software that makes aesthetically valued edits based on a corpus of past improvisations.

In practice these two parts are separated in sourcecode but are compiled into a single software application. For discussion we treat them separately, terming the first part *the language* and the second *the rhythm generator*.

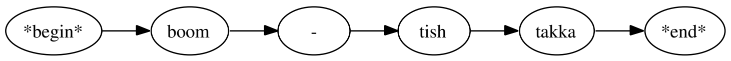
4.1 Language

The language has a polymetric syntax is adapted from that of the Bol Processor version 2 (Bel, 2001).

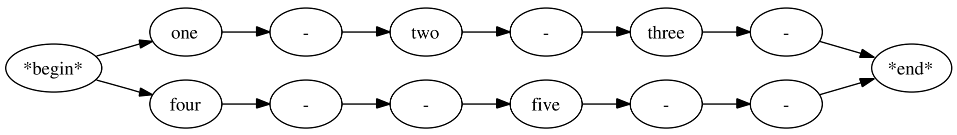
```
data Event = Sound String | Silence

data Structure = Atom Event
               | Cycle [Structure]
               | Polymetry [Structure]
```

Figure 6: Haskell data types for representing polymetric structure

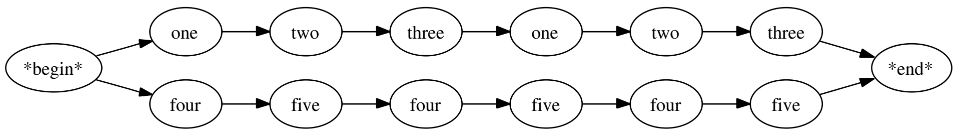
A monophonic sequence of vocables is specified simply separated by whitespace, with pauses denoted with an underscore. For example
`boom _ tish takka`



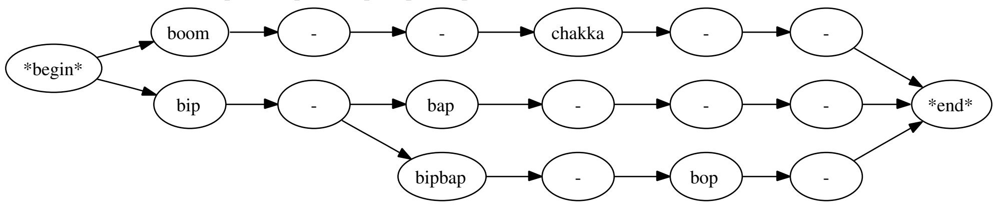
Polymetric sequences are surrounded in braces, the different parts separated by commas, for example
`{one two three, four five}`



Notice that pauses are inserted in order to arrange two sequences of different lengths. This is done using the lowest common multiple of the part lengths. If square brackets are used rather than braces, then the parts are repeated rather than padded with pauses, thus:
`[one two three, four five]`



Polymetric constructs may also be nested:
`{boom chakka, bip {bap, bipbap bop}}`



The language is parsed using the Haskell Parsec library (Leijen and Meijer, 2001). It is parsed into a data type structure shown in Figure 6, where *Structure* data describe an arrangement of *Event* data.

This shows that an *Event* data type can be instantiated either as a *Sound* with a *String* parameter, or as *Silence*. A *Structure* data type describes an n-ary tree, where branches can be instantiated as *Polymetry* or *Cycle*, and leaves with *Atom* and an *Event* parameter.

A well formed structure will have Polymetry composed only of Cycle instances, and Cycle instances composed only of Polymetry and/or Atom instances.

4.2 Onomatopoeic Synthesis - Transforming words into physical model articulations

Sound synthesis is based on the Karplus-Strong model shown in §3.6. A word is transformed into a sequence of sets of synthesis parameter changes, where consonants are mapped to the wavetable size and blend parameters, and vowels are mapped to formant filter parameters.

The consonant mappings were chosen by hand, using the mouse to select a point on the screen, where the an x/y position of the cursor is mapped to the wavetable size and blend parameters. Parameters were chosen to produce a timbre with some similarity to the spoken consonant. However we placed the greatest effort in producing a broad range of interesting sounds.

The vowel formant values were taken from the female alto voice parameters in the *FormantTable* Supercollider library by Putnam (2006).

The sequence of parameter settings is then sent to Supercollider server at fixed rate, giving a striated rhythm to the articulation. The synthesis graph includes a lag delay for each parameter, sliding linearly from a previous value to a new one, giving a sense of continuous, speech-like articulation.

4.3 User interface

The user interface is built upon the GNU Readline library. Readline is a standard interface for editing lines of text, used in many console applications including the UNIX Bash commandline. It supports many text navigation and editing features from the celebrated EMACS text editor, ideal for moving around and manipulating a rhythm expressed as a line of text at speed. It also supports history, so previous rhythms may be browsed and searched through.

With this interface then, the performer simply types in a rhythm as a line of text, using the language detailed in §4.1. When they press return, if the line parses then the resulting musical structure is placed into a shared variable for sequencing by a separate thread. If the line does not parse, for example due to mismatched parentheses, then an error message is displayed and no change is made to the sound.

The sequencing thread simply loops the last successfully parsed rhythm, where the vocables and silences between them are articulated at a fixed rate. Polymetric rhythms are voiced by assigning concurrent parts of a rhythm to different synthesis graph instances.

We extend the readline interface with two commands. The first is that by pressing the exclamation mark key (!), the rhythm currently being edited will be ‘normalised’§4.4.3. The second is that when the hash (#) key is pressed, a new rhythm is generated to follow from the previous two rhythms, using statistical analysis of past performances. We describe how this generation is done in the following sections.

4.4 Statistical model

We have so far introduced the language environment within our software that allows a performer to improvise with vocables. We now focus on how the software builds a statistical model of use in order to produce new rhythms.

The statistical model consists of a number of first order Markov chains. Each chain represents either continuation from one vocable to the next or from one rhythmic pattern to the next.

4.4.1 Vocable continuation

Within a rhythm, a vocable is considered a continuation of a previous one if no other vocables occur between them. A continuation is sensitive to polymetric structure – two vocable instances which appear within different parts of the same polymetry may not form a continuation, even if they appear to be adjacent in a ‘flattened’ form of the time structure. That said, two vocable instances occurring in different, successive polymetries may form a continuation. This means that a vocable instance may form a continuation with more than one predecessor and/or more than one successor.

A first order Markov chain is constructed to represent these continuations.

4.4.2 Rhythmic continuation

Two further Markov chains are constructed in order to represent continuations from previous rhythmic structures to the next. Taking use the following example of sequence of rhythms used during an improvisation:

```
{boom tish, _ bang tish}
{bam nok tish, _ tish tash}
{bam nok, _ tish tash beee}
```

We represent a rhythmic structure in the abstract by enumerating each vocable within it, resulting in what we term a *rhythmic schema.* For example we represent this rhythm:

```
{boom tish, _ bang tish}
```

with the following schema:

```
{1 2 , _ 3 2}
```

We also represent similar words within the rhythm. For example our second example:

```
{bam nok tish, _ tish tash}
```

results in the following:

```
{1 2 3, _ 3 4}
S(3, 4)
```

S(3, 4) represents a *constraint. S()* may be applied to two constants, in this case 3 and 4, to signify that variables (here, the vocable words *tish* and *tash*) assigned to them should be similar. Similarity here is a binary measure, where a 50% threshold is applied to Levenshtein distance (§3.7.1), expressed as a percentage of the length of the longest vocable.

We find such a schema for each rhythm, and build a first order Markov chain describing the likelihood of one following another.

However this is not fully expressive of the continuation; the reuse of the vocable *tish* in both rhythms is lost, as is the similarity between *boom* in the first rhythm and *bam* in the second. To solve this problem we take in more context, where each schema represents a pair of rhythms. Going back to our example, we would take the first and second rhythms together:

```
{boom tish, _ bang tish}
{bam nok tish, _ tish tash}
```

finding:

```
{1 2 _, _ 3 2}
{4 5 2, _ 2 6}
S(2, 3), S(3, 4)
```

and then take the second and third rhythms:

```
{bam nok tish, _ tish tash}
{bam nok, _ tish tash beee}
```

finding:

```
{1 2 3, _ 3 4}
{1 2, _ 3 4 5}
S(3, 4)
```

These two schemata together model a continuation from two rhythmic structures to a third. Another Markov chain is constructed around these continuations.

We in fact construct Markov chains for both forms of rhythmic schemata; the latter has greater context but the former is more general. In §4.5 we describe our backoff technique for using them together.

4.4.3 Normalisation

We now step back to note a glaring problem with this representation; there is more than one way of representing equivalent rhythms. In particular, the parts of a polymetry comprise an unordered set; it is of no consequence which order the parts within polymetric rhythms are in. For example, the following two rhythms are equivalent:

```
a) {boom tak, tak _ tok} b) {tak _ tok, boom tak}
```

So before applying analysis described above, we need to normalise the cycle, so that cycles with equivalent schemata are represented identically. For this we need a reliable method for finding a canonical representation of a cycle, in terms of its rhythmic schema. Here our canonical form is based only upon the enumeration of cycle, not any particular vocables used within it.

The first step in finding a canonical form is 'compressing' the cycle. This simply ensures that *Cycles* within *Cycles* are merged, and subcycles containing only *Silence* atoms are removed.

The second step is a depth-first sort of Polymetry substructures. Only the rhythmic schemata of the substructures are considered during this sort, because it is only the schemata which we will later wish to compare. We must perform a depth-first recursive sort, so that the members of each Polymetry is sorted in turn according to an enumeration of its members. The default Haskell ordering is used to determine sort order.

For example, the polymetries in the above example would be enumerated thus:

```
a) {1 2, 1 _ 2}  b) {1 _ 2, 1 2}
```

In our example, a) is the canonical form.

We apply this canonical form to out rhythms before finding the Markov chains of schemata as described above.

4.5 Generating rhythms

Now we have described the statistical model in the previous section, we can describe how we use it to generate rhythms. We first identify a rhythmic schema and then fill in the schema with vocables.

It should be noted that rhythms are intended to be generated on-line, during a musical improvisation. This means that once a rhythm is chosen we are committed to that choice; there is no opportunity for back-tracking to a previous state.

In all cases Markov models are applied using a pseudorandom Monte Carlo algorithm. Where a predicate is matched, a successor is chosen with a chance proportional to the number of previous instances of the continuation divided by the total number of instances of the predecessor.

Currently Markov models are not updated on-line - they are built from a corpus of past improvisations.

4.5.1 Choosing a rhythmic schema

A rhythmic schema is chosen as a continuation of the schemata of the previous rhythmic cycles of the current improvisation.

First a match against the Markov chain over bigrams of rhythmic schemata is attempted. If unsuccessful a reattempt is made but with similarity constraints ignored.

If still unsuccessful, a match is attempted against the Markov chain over single schemata. Again if no match is made, the match is reattempted without considering constraint rules.

If a match has *still* not been found, a schema is chosen arbitrarily from all those seen before.

4.5.2 Choosing vocables

Once a rhythmic schema is chosen, vocables are assigned to constants within it. First, a directional graph of constants is constructed, describing continuations from constant to constant in the same way as we found vocable continuations in §4.4.1.

We start with the known vocables from the previous cycle. Remaining unassigned constants are filled in turn. Because we match constants from lower to higher, we can only consider continuations of constants from lower to higher – with our current model we only apply continuations from predicates to successors and not in reverse.

We then fill each remaining constant in turn. If there is more than one predicate their successors within the Markov model are grouped together before Monte Carlo selection is performed. We first attempt to select from the successors matching any applicable similarity constraints. If no applicable successors are found, then the software attempts to generate a new vocable that does fit the constraints. This is done by finding the vocables with which the new vocable shares a similarity constraint, and applying half the edits on a minimum cost path between them (§3.7.1).

If there are no applicable similarity constraints then there is nothing to base the generation of a new vocable. In that case a match is reattempted but with the similarity rules ignored. As a last resort a vocable is picked at random from all those seen before.

5 Evaluation

5.1 The language

Qualitative evaluation of the language was performed by interview using electronic musicians, all of whom have experience with performing with their own software. A brief demonstration was given to each interviewee, letting them experiment with the system for ten minutes and then asking how they felt about using the system. The interviews are recorded in Appendix A, together with a sample of the rhythmic continuations made by each interviewee. Any deviations from the described interview format are detailed there.

Feedback during this short term exposure to the language was on the whole encouraging. While one interviewee saw limitations in the range of timbres produced, others saw opportunities. They all picked up the system quickly with little or no prompting, apart from explaining the polymetric syntax.

5.2 The rhythm generator

The rhythm generation aspect of this project deserves a full quantitative evaluation, using a framework such as Amabile's Consensual Assessment Technique (Pearce, 2005). Such a detailed assessment is outside the scope of this thesis, and is left for future work.

Instead we produced three improvisations on consecutive days, and observed the analysis of them by the rhythm generator. The improvisations has a mean average of 67.7 rhythms each, totalling 203 rhythms between them. They lasted between 5 and 7.5 minutes. They were produced in order to explore use of the system, the method of evaluation had not been decided upon prior to their production. This is of course a very small corpus, but allows us the opportunity to observe the system in operation at least.

We looked for *pivot points*, where a Markov model identified a reoccurence of a rhythmic schema. Our most complex model where the surface data were schemata of pairs of rhythms, showed one, three and three pivot points within our three improvisations respectively.

Our first backoff step, where constraint rules are ignored showed two, seven and four pivot points. This suggests both that constraints are indeed constraining the possibilities within our data, and that the backoff step is useful.

Our second backoff step, with schemata of single rhythms, again had more pivot points; eight, nine and eleven.

There were few pivots across improvisations. Pivots were almost all continuations with small changes, replacing single words with another.

The data is not suitable to base any firm conclusions on; we need a larger corpus to find whether this system is able to model the style of an improviser across improvisations. In any case, style is unlikely to emerge without a strong familiarisation the language – what we have captured here is an improviser exploring the limits and character of the system. It may take months of practicing and performances before a corpus is available for fuller analysis. We can however see the system working as intended.

6 Further work

This system leaves many avenues unexplored, in some cases due to lack of time, and in others due to a desire to keep parts of the system as simple as possible as a clear proof of concept.

Performance rules and timing Our system confirms to a strict time structure, not only triggering vocables at a fixed rate, but performing articulations within vocables at a fixed rate. In this form it is therefore unlikely to find much use beyond the limits of electronic dance music. One approach could be to make the language more expressive of timing, perhaps by allowing uneven distributions of sound events that are reconciled by subtle adjusting of timing. Another could be by applying timing rules to rhythms, for example KTH performance rules (Friberg et al., 2006).

Computational creativity We have really only scratched the surface of what is possible, and not begun to properly evaluate this side of our work. We have shown how vocable words can be a useful shared language between human and other creative agents, and look forward to developing this idea further, fulfilling a greater part of the CSF.

Fuller physical models We chose just about the simplest physical synthesis model possible with which to explore our ideas. A fuller tube resonance model using more complex waveguide synthesis would open up the range of possible timbre significantly, requiring more complex articulations to control it.

Improved merging of vocable words Our technique of using the minimum cost path to find a new vocable half way between two given vocables is useful, but could be improved. The edits are made from one end of the string to the other, if a method for more evenly distributed edits along a minimum cost path were possible, that could produce more interestingly blended words.

Word morphology Rather than treating words as a sequence of letters, we could instead consider them as a sequence of syllables. By getting closer to the morphology of for example English words, we could perhaps strengthen the human perceptual connection between written and synthesised vocable words. Where vocable control is applied to differing synthesis systems, generic means of mapping vocable morphology to synthesis parameters could well emerge, easing wider adoption further.

Zeroth order Markov chains Our 'last resort' backoff step has been to pick at random from those data seen before. If this was weighted, so that a datum with many instances was more likely to be picked than one with few, then resulting continuations would perhaps be more plausible.

Reverse matching Markov chains Our model only considers the likelihood of one datum following another. When applying vocables within a rhythmic schema where following position may already have been matched, finding vocable likely to *precede* another could contribute towards more plausible results.

Embedding in other systems Several live programmers have commented that they would like to see our language embedded in the languages they use. In its present form the system is simple enough to make this quite feasible, and could result in wider use. It could be particularly interesting to integrate our vocable synthesis into the Bol processor language environment.

Language The language could be more expressive if functional abstractions were possible, taking the language into the realm of live coding. How this could be done so that a software agent could make valued edits is an interesting question, requiring further research. There is also great scope in developing syntax for musical structure further.

7 Conclusion

We have demonstrated a method for controlling synthesis graphs with vocable words, and found this to be useful in producing music. We have further demonstrated how the resulting textual representation is open the analysis and generation of rhythms by software agents.

As we saw in the previous section, many doors have been thrown open to future work. We have a strong belief that vocable synthesis is a highly promising direction which could have applications throughout electronic music to aide the expressive specification and manipulation of complex sounds. We also feel our intuition that representing sounds as vocables could find use in the search for computational creativity has been supported by our work so far. We hope to properly investigate and test these feelings and beliefs further in future work.

For the short term though, we are pleased to have built a system suitable for using as part of group improvisation. It is our hope that this inspires greater focus on the word morphology in software based music, particularly in improvisatory live coding.

A Interviews

A summary of each interview is shown together with a sample of their input. The interviews focussed on the vocable synthesis and not the rhythm generation, so all rhythms shown were typed by the interviewee.

Interviewee A would have preferred a wider range of timbre, and found consonants and vowels did not produce sounds that you might expect. He found it confusing that words could interfere with each other in synthesis, behaviour which I corrected before the later interviews. We then entered a long conversation about “abc”, an interesting textual notation system for fiddle melody and useful reference for future work.

```
hello
hell o
ell ho
oooooooooooooooo
oooeeeooooeeeeoooo
ooookkkkooookkkooo
ooo
ooo ooo oo ooo
eee eee eee e
kkk kkk kkk k
oop
eep
uo uop pou eep pee uu
ou _ uo _ aaaaaaaaa
aaa___aaa____aa___
```

```
zzz___zzz\____
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz_____________
zoopoopeeuup
egbert__begbert
bip_bop_pop_bob
bip_bop_pop_bob_
bi_bo_po_bo_
bi_bo_po_bo_p_
bi_bo_po_bo_ppppp_
bi_bo_po_bo_ppoppp_
bi_bo_po_bo_pop_
oi_oi oi_oi
oi_oi oi_oi
oi__oi__oi__oi
oi__oi--0
zero zero zoom ___ zoom zero ___
zeeero zeero zoom ___ zoom zero ___
```

Interviewee B found the system fun to play with, and very accessible. He felt that without a graphical representation of the rhythm, he had to use his imagination more. He admitted being a shy person, and self-conscious when typing 'silly' words. He started with typing onomatopoeic words, and while he found it slightly disappointing in the results he found this a good entry point for exploring the sounds the system could make, after which he understood the system enough to invent new, interesting sounding vocables.

```
pok pik pakpot pit
pkkk wiikk
pkpkpk _ gtgtgt eeek
schweeeeiiiuuuuu plooooooeeeeaaaalll
da ji pa xiao ji
{da _ ji _, pa _ xiao _, o cha _ p}
{zhuuu schweeeeeuuuuiii, { p t j { y o } } }
{zhuuu schweeeeeuuuuiii, { p t j { y o } } bo }
{zhuuu schweeeeeuuuuiii, { p t j { y o } }, bo }
zhuuu schweeeeeuuuuiii, { p t j { y o } }, bo
zhuuu schweeeeeuuuuiii, { p t j { y o } }, bo ba pa
zhuuu schweeeeeuuuuiii, { p t j { y o } }, bobdbd ba pa
zhuuu schweeeeeuuuuiii, { p { y o } }, bobdbd ba pa
zhuuu schweeeeeuuuuiii, { p _ _ _ { y o } }, bobdbd ba pa
pik zhuuu schweeeeeuuuuiii, { p _ _ _ { y o } }, bobdbd ba pa
schweeeeeeeiiiiiii schwiiiieeeeuuuuuuuuoo
schweeeeeeeiiiiiii schwiiiieeeeuuuuuuuuu
schweeeeeeeiiiiiii schwiiiiuuuuuuuuu
schweeeeeeeiiiiiiiiiiiiiiiiiiiiiii schwiiiiuuuuuuuuu
wiiiiiiiiiiiiii   wooooooooooooooo
```

Interviewee C was more interested in representing particular rhythms than the previous interviewees. He managed to transcribe a rhythm quite quickly, although has to make conscious effort to get the various offsets right. He suggested perhaps extra key combinations or visual help could be included to help with aligning rhythmic parts. He also suggested the notation could be applied to visual as well as musical form.

```
{bar foo wii mii, moo}
{tik tik tik tik, moo}
```

```
{tik tik tik tik, moo _ moo _}
{_ tik _ tik, moo _ moo _}
{_ tik _ tik, moo _ moo _, wo}
{_ tik _ tik _ tik _ tik, moo _ moo _ moo _ moo _, wo}
{_ tik _ tik _ tik _ tik, moo _ _ _ moo _ moo _, wo}
{_ tik _ tik _ tik _ tik, moo _ _ _ moo _ moo _}
{_ tik _ tik _ _ _ tik, moo _ _ _ moo _ moo _}
{_ tik _ tik _ _ _ tik, moo _ _ _ moo _ moo _, b b b b b b b b}
{_ tik _ tik _ _ _ tik, moo _ _ _ moo _ moo _, pkpkppkpkpkpkpkppkkp}
{_ tik _ tik _ _ _ tik, moo _ _ _ moo _ moo _, a b c d e f g h}
{_ tik _ tik _ _ _ tik, moo _ _ _ moo _ moo _, a b c d e f g h, wooooo
{_ tik _ tik _ _ _ tik, moo _ _ _ moo _ moo _, a b c d e f g h, wooooo oo, wii mii}
{_ tik _ tik _ _ _ tik, moo _ _ _ moo _ moo moo, a b c d e f g h, wooo oooo, wii mii}
```

Interviewee D was able to install and run the software on his own computer, and returned the following thoughtful response by email.

It makes some really nuanced sounds that I would find it very hard to program a synth to do directly.

The sounds it makes often aren't what I imagined when typing a new word, but by playing with it interactively, adding pauses, etc. it's quite easy to map what you're hearing back to the words you typed, and then you can quickly learn to direct what it produces - interaction is very fundamental for it.

I really liked the line at a time interface that is ideal for experimenting, and understanding that mapping between input and sound. The simplicity means you don't end up with the aural equivalent of mixed-together brown plasticine, and clarifies the responsibilities of each word. I wonder how you could make the system more powerful for more intricate compositions while keeping the clarity?

I wanted to use capitals for accents, and punctuation for prosody

It is exhilarating to type

```
[klntrx _ pu po pi _ _ y, {_ ke}, { t tt _}]
```

and get something sensible back! Going from a programming background where a single typo ruins the program to being able to freely make words up is quite liberating.

Interviewee E found trying to predict what sounds would be made was engaging. He was particularly interested in playing with the polymetric syntax, it was initially difficult to understand but learning how to control the time expansion was fun. He found the range of timbre fine to play with during this session but thought that it would become tiresome if used for a whole solo performance. However if it was part of a performance with other instruments it may work better – often in computer music broad ranges of sounds are used, so it could be nice to identify such a particular sound as this from a particular improvisor. He was was wondering if the synthesis could be more like 'human beat-boxing'. He was disappointed to not find a nice 'kick' sound, but did find other distinctive sounds that gave a sense of orientation within a looped rhythm. He enjoyed the immediacy of the commandline interface, sliders for letters within a GUI would have become tiresome.

References

Allison, L. (1992). Lazy dynamic-programming can be eager. *Information Processing Letters*, 43(4):207–212.

Andrews, R. (2006). Real djs code live. `http://wired.com/`.

Banks, J. (2004). Rorschach audio. *Strange Attractor*, 1(1):124–159.

Bel, B. (2001). Rationalizing musical time: syntactic and symbolic-numeric approaches. In Barlow, C., editor, *The Ratio Book*, pages 86–101. Feedback Studio Cologne.

Blackwell, A. and Collins, N. (2005). The programming language as a musical instrument. In *Proceedings of PPIG05*. University of Sussex.

Boden, M. (1990). *The Creative Mind*. Abacus.

Chambers, C. K. (1980). *Non-lexical vocables in Scottish traditional music*. PhD thesis, University of Edinburgh.

Cobbing, B. (1970). *Sonic Poems*. Writers Forum.

Collins, N. (2006). *Towards Autonomous Agents for Live Computer Music: Realtime Machine Listening and Interactive Music Systems*. PhD thesis, Centre for Science and Music, Faculty of Music, University of Cambridge.

Collins, N., McLean, A., Rohrhuber, J., and Ward, A. (2003). Live coding techniques for laptop performance. *Organised Sound*, 8(3):321–330.

Cook, P. (1999a). *Music, Cognition and Computerized Sound*, chapter Voice Physics and Neurology, pages 105–116. MIT Press.

Cook, P. (1999b). *Music, Cognition and Computerized Sound*, chapter Articulation in Speech and Sound, pages 139–147. MIT Press.

Cook, P. (2002). *Real sound synthesis for interactive applications*, chapter Strings and Bars, pages 97–107. A K Peters.

Crossley, B. (2005). *The Triumph of Kurt Schwitters*. Armitt Trust.

Damerau, F. J. (1964). A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176.

de Rijke, V., Ward, A., Stockhausen, K., Drever, J. L., and Abdullah, H. (2003). Quack project cd cover notes, `http://quack-project.com/`.

Drape, R. (2007). Haskell supercollider, a tutorial. `http://www.slavepianos.org/rd/sw/hsc3/`.

Elderfield, J. (1985). *Kurt Schwitters*. Thames and Hudson.

Eno, B. (1996). Generative music. *In Motion Magazine*.

Friberg, A., Bresin, R., and Sundberg, J. (2006). Overview of the kth rule system for musical performance. *Advances in Cognitive Psychology, Special Issue on Music Performance*, 2(2-3):145–161.

Hamming, R. (1950). Error Detecting and Error Correcting Codes. *Bell System Techincal Journal*, 29:147–160.

Jaffe, D. A. and Smith, J. O. (1997). Extensions of the karplus-strong plucked string algorithm. *Computer Music Journal*, 7(2):1983.

Jeffs, C. (2007). Cylob music system. `http://durftal.com/cms/cylobmusicsystem.html`.

Jones, D. E. (1987). Compositional control of phonetic/nonphonetic perception. *Perspectives of New Music*, 25(1):138–155.

Jones, D. E. (1990). Speech extrapolated. *Perspectives of New Music*, 28(1):112–142.

Jones, S. P., editor (2002). *Haskell 98 Language and Libraries: The Revised Report.* `http://haskell.org/`.

Karplus, K. and Strong, A. (1983). Digital synthesis of plucked string and drum timbres. *Computer Music Journal*, 7(2):43–55.

Kippen, J. (1988). *The Tabla of Lucknow - A cultural analysis of a musical tradiation.* Cambridge University Press.

Leijen, D. and Meijer, H. (2001). Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Institute of Information and Computing Sciences, Utrecht University.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady.*, 10(8):707–710.

Liberman, A. M. and Mattingly, I. G. (1985). The motor theory of speech perception revised. *Cognition*, 21(1):1–36.

McCartney, J. (2002). Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68.

McCullough (1989). *Concrete Poetry - An Annotated International Bibliography, with an index of poets and poems.* Whitston Publishing.

Nettl, B. (2001). Improvisation (i). In Sadie, S., editor, *Grove music online (accessed March 2007).* Macmillan.

Pearce, M. (2005). *The Construction and Evaluation of Statistical Models of Melodic Structure in Music Perception and Composition.* PhD thesis, Department of Computing, City University, London, UK.

Pressing, J. (1984). Cognitive processes in improvisation. In Crozier and Chapman, editors, *Cognitive Processes in the Perception of Art*, pages 345–363. Elsevier Science Publishers.

Pressing, J. (1987). Improvisation: methods and models. In Sloboda, J. A., editor, *Generative Processes in Music*, pages 129–178. Oxford University Press.

Pressing, J. (1999). The referential dynamics of cognition and action. *Psychological Review*, 106(4):714–747.

Putnam, L. (2006). Supercollider extensions – formanttable. `http://www.uweb.ucsb.edu/~ljputnam/sc3.html`.

Rodet, X., Potard, Y., and Barriere, J.-B. (1984). The chant project: From the synthesis of the singing voice to synthesis in general. *Computer Music Journal*, 8(3):pp. 15–31.

Schwitters, K. (1932). Ursonate. *Merz*, 24.

Sorensen, A. and Brown, A. (2007). Aa-cell in practice: an approach to musical live coding. *Proceedings of the International Computer Music Conference.*

Uehlecke, J. (2006). Tanz den maschinencode. *Zeitwissen*, page 96.

Ward, A., Rohrhuber, J., Olofsson, F., Mclean, A., Griffiths, D., Collins, N., and Alexander, A. (2004). Live algorithm programming and a temporary organisation for its promotion. In *README Reader, Proceedings of the README software art conference, Aarhus.* Nordic Institute for Contemporary Art.

Wiggins, G. A. (2006a). A preliminary framework for description, analysis and comparison of creative systems. *Journal of Knowledge Based Systems.*

Wiggins, G. A. (2006b). Searching for computational creativity. *New Generation Computing*, 24(3):209–222.

Zmölnig, I. and Eckel, G. (2007). Live coding: An overview. *Proceedings of the International Computer Music Conference*.