Program Report: Generative Looping Sampler

Arthur Carabott

Abstract

A "live looping" tool is implemented with additional generative composition and mixing techniques. The goal is to provide musical and sonic variation to looping audio material recorded on-the-fly.

1 Introduction

A popular tool among performing musician's is the 'loop pedal' a piece of hardware that can record an input signal, play it back as a continuous loop with the option to overdub further material to build layers. This enables solo musicians to compose pieces of music on the fly, with more concurrent layers of music than they would normally be able to perform. An early example of a loop system is *King Crimson* guitarist Robert Fripp's 'Frippertronics' which used two analog tape reels (Gaskin and Fripp, 1979). In more recent times loop pedals have adopted digital sampling technology (AKAI, 2008; Roland, 2008), this switch to from analog to digital has enabled further features such as loop quantisation so that recorded material always loops perfectly (see appendix 2 for an example of a good and bad loop).

While the method of loop production has been improved with technological advances, the underlying principle of the system, and its flaws have remained the same. The main weakness of such systems is that every iteration of a loop is identical, resulting in a very static performance from the virtual band or as world-renowned guitarist and loop pedal exponent Kaki King puts it "the vibe just goes away" (King, 2008). In comparison to a real ensemble of musicians, "loopers" performances also suffer sonically as all loops are heard from the same source location (typically a guitar amplifier, or possibly a P.A) with no panning or mixing facilities to spread the sounds across the stereo spectrum.

Despite plenty of research into automated generation of material (Collins, 2006; Cope, 2004; Weinberg et al., 2008; Eigenfeldt, 2007; Bel, 1992; Biles, 1999), as well as automated mixing (Gonzales and Reiss, 2007; Gonzalez and Reiss, 2008a,b), to the author's knowledge they have yet to be applied to a dedicated looping system.

It would be fair to argue that generation of new material might not appeal to all performers (King, 2008 was undecided), it certainly requires relinquishing a certain level of control, although little more than when playing with a fellow musician (the fact that "loopers" tend to be solo musicians suggests that even this may be too much!). Generative mixing of levels is likely to have a wider circle of appreciation, as it could be set to optimise the mix (for the conservative) or alter it heavily (for the adventurous). It was King's (2008) dissatisfaction with the sonic output of current loop pedals that spurred the idea of automated mixes.

To overcome the previously mentioned shortcomings a system has been built that uses generative computational techniques to produce loops with greater musical and sonic variety.

2 Illustration

An example output has been prepared and supplied as an appendix 1.

- 0:00-0:18 A beatboxed drum loop is recorded and analysed.
- 0:19-0:36 A 3 beat percussive loop is recorded, creating emergent patterns with the 4 beat drum loop. Automated mixing is turned on, the changes in pan position are most noticeable.
- 0:37-0:53 The phrase "Ge-ne-ra-tive Cre-a-tiv-i-ty" is recorded, analysed and set to automated mixing.
- 0:54-1:13 Automated generation of new material starts using the "Generative Creativity" phrase. Pitch shifts and rhythms are obvious.
- 1:14-1:37 A guitar chord is recorded, analysed and set to automated mixing and material generation.
- 1:38-1:47 A second guitar chord is recorded and treated in the same way.

- 1:48-1:59 Probability of triggering is increased, events occur far more frequently.
- 2:00-2:50 A guitar improvisation begins and can be heard faintly in the left channel.
- 2:51-3:25 Drum track among others are manually turned down, piece ends abruptly.

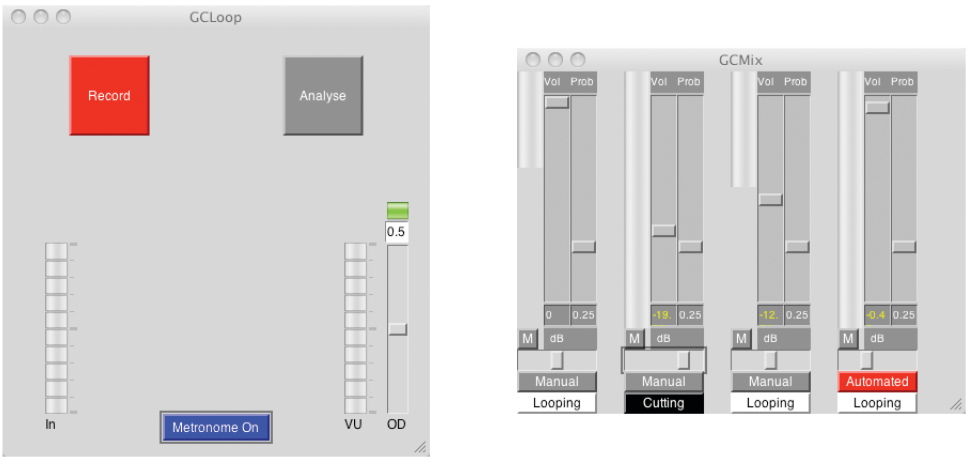

Figure 1: Recording/Analysis GUI (L) and Mixer GUI (R)

3 System Overview

System Abstract A system has been built to overcome some of the limitations and constraints of current loop pedal technology. Facilities for recording an audio signal and continuously playing recorded material quantised to a given tempo are provided, in line with current technology. Out of tradition with most loop pedals, quantisation of loops is performed to each beat (crotchet) rather than to the bar, using a variety of loop lengths, polymetric and polyrhythmic structures can emerge. To further advance the system beyond the status quo, stochastic generation of new material is introduced in a manner similar to probabilistic drum machines.

Note The system has been implemented in SuperCollider 3, an environment for real-time audio synthesis and algorithmic composition. While SuperCollider's object-orientated programming style may be familiar to the reader, SuperCollider objects and the term **SynthDef** (Synthesizer Definition) may not. A **SynthDef** is analogous to a Class in an OOP language, in that it is a "recipe" for a Synthesizer on SuperCollider's server from which many instances (Synths) can be made. SuperCollider is actually composed of numerous elements, a healthy level of abstraction is adopted here with only a distinction between the language (an audio programming language) and the server (an audio server) (SC-Developers, 2009). All **SynthDefs** in this system have been written by the author. Any SuperCollider specific objects will be highlighted and be disambiguated.

3.1 System Class

The system has been designed as a single class *GCLoop* which initialises all necessary variables and structures (such as **Busses** and **Buffers**), sends **SynthDefs** to the server, creates **Synth**instances, establishes a connection between server and language, creates a GUI and provides methods necessary for operation.

Instantiating the *GCLoop* class requires the following arguments

- *aTempo* Tempo for the performance.
- *aTimeSig* Number of beats per cycle/bar.
- *aBufferTime* Size of the recording buffer in number of beats. As loops lengths are typically a factor or multiple of the time signature, the buffer size will be adjusted to be a multiple of the time signature.

An example instantiation:

```
var looper = GCLoop.new(60, 4, 120);
```

When instantiated *GCLoop* will call the following setup methods:

- *setSynthDefs:* Sends the required **SynthDefs** to the server.
- *setOSCResponder:* Creates an **OSCResponder** object for communication between server and language. Set language responses to server messages.

- *setSynths:* Creates instances of the immediately required **Synths**.
- *setMainGUI:* Sets up the recording/analysis GUI (see figure 1).
- *setMixerGUI:* Sets up the mixer GUI (see figure 1)

3.2 Time in the system

Time and synchronically are achieved via the suitably named *clickTrack* **SynthDef**. *clickTrack* takes as input:

- *buf* A **Buffer** (piece of memory allocated by the server) from which the correct rate of increase (for the **Phasor**) is determined using a **BufRateScale** object.
- *tempo* The desired tempo, in beats per minute.
- *oneBeat* The number of samples per beat (at this tempo). While this could be calculated inside the **SynthDef**, it is a value used so frequently throughout the system that it became easier to pass it as an argument.
- *timeSig* The number of beats per cycle or loop, making possible any time-signature, with the assertion that quaver denominator time-signatures such as 7/8 are synonymous with a crotchet denominator time signatures at double the tempo.
- *outClock* A **Bus** to output clock information on, for use by other Synths.
- *recTrigger* An (integer) flag where a change from negative to positive indicates that recording will begin at the next beat.
- *outStartRecBus* A **Bus** used to trigger the recording Synth.
- *onsetTrigger* A flag used to begin analysis, akin to *recTrigger*.
- *outStartOnsetBus* A **Bus** used to trigger the analysis Synth.
- *beepVol* Volume control of the metronome beeps.

An example instantiation:

```
var metroSynth = Synth(\clickTrack, [\buf, 6, \outClock, clockBus,
\tempo, 120, \timeSig, 4, \oneBeat, 24000, \recTrigger, 0,
\outStartRecBus, startRecBus, \onsetTrigger, 0,
\outStartOnsetBus, startOnsetBus, \beepVol, 1]);
```

Within a *clickTrack* **Synth** a **Phasor** object is created to output a continuously increasing value at the host machine's sample-rate. The modulo operator is used on this value with **Trig** objects to trigger (change value from 0 to 1) at the first beat of every bar, on every beat and on every half beat. These triggers are used to create the metronome beeps (simple **SinOsc** sine wave oscillators), and are sent to the language via **SendTrig** objects.

A *clickTrack* **Synth** will output the following:

- The output of the **Phasor** object via a **Bus** to be used as a clock by other Synths.
- Triggers to start the recording and analysis Synths.
- Bar, beat and half beat messages to the language.
- Audible metronome beeps at the given tempo, with an accent on the first beat.

Operation A *clickTrack* **Synth** is created when the *GCLoop* class is instantiated, stored in the variable *metroSynth*. The data from the **Phasor** object is makes timing information available to other **Synths** on the server. The **SendTrig** objects in the *metroSynth* provide the SuperCollider language with timing information via an **OSCresponder** object (variable *oscr*). Each message sent from the server to the language has a unique id number, so that the correct method can be called for each message.

3.3 Recording

Recording is carried out using an instance of the *GCRec* **SynthDef**, which takes as input:

- *sigIn* The input index of the hosts audio interface to use.
- *clockIn* The **Bus** carrying the **Phasor** clock information.

- *bufnum* The unique ID of the **Buffer** to record to.
- *trigger* The **Bus** carrying the trigger to start recording.
- *stopRec* A flag that tells the **Synth** it is *going* to stop recording at the next beat, so that the envelope can closed.
- *oneBeat* The number of samples per beat at this tempo.

An example instantiation:

```
recSynth = Synth(\GCRec, [\bufnum, 3, \clockIn, clockBus,
\trigger, recStartBus, \oneBeat, 48000]);
```

The *GCRec* **Synth** focuses around a **SoundIn** object using the index from *sigIn*, this feeds the incoming audio signal into a **BufWr** object which writes it to a **Buffer**. The clock for the **BufWr** object is driven by the clock from the *clickTrack* Synth, which has been passed in via the **Bus** specified by *clockIn*.

An **EnvGen** object is used to envelope the input signal at the start and end of recording. Initially the envelope is held open when triggered (via the *trigger* argument), but once the *stopRec* argument is changed to 1, a trigger based on a second clock that runs in sync with, but 20ms ahead of the recording clock is used. As a result the envelope closes 20ms before the recording stops, avoiding the artefacts that result from signal discontinuities.

Technically a *GCRec* Synth is recording continuously, however when set "not to record" the envelope (which the input signal is multiplied by) is set to 0, resulting in a silent signal.

The *GCRec* **Synth** also features a **SendReply** object used to send the amplitude of the input signal to the language every 100ms, to control a level indicator in the GUI.

A *GCRec* **Synth** will output the following:

- An audio input signal, enveloped, outputted onto a **Buffer**.
- Amplitude information of the input signal, sent to the language.

A *GCRec* **Synth** is created when the *GCLoop* class is instantiated, stored in the variable *recSynth*.

Operation Recording is started/stopped via the GUI's record button, which calls the methods *startRecordingAction* and *stopRecordingAction* in alternation.

- *startRecordingAction* tells the *metroSynth* to trigger the *recSynth* at the next beat, create an instance of *GCPlay* for playback as well as keeping track of the beat in the recording buffer (*recBuf*) at which recording started (*startRecBeat*), and the total number of beats used for this recording (*totalRecBeats*). This is made possible by the **SendTrig** object in the *metroSynth* sending a message at every beat.
- *stopRecordingAction* tells the *metroSynth* to stop the *recSynth* recording at the next beat, and tells the *recSynth* to close the envelope 20ms before the next beat.

The *cropAndErase* method is then called, taking the following arguments:

- *startBeat* The beat in *recBuf* at which recording started.
- *totalBeats* The total number of recording beats.

Using *totalBeats* a new **Buffer** is allocated with the required number of samples (*totalBeats*oneBeat*), onto which the signal recorded onto *recBuf* is copied. In the case that recording started near the end of *recBuf* and looped back to the start of *recBuf* the signal is copied in two chunks, as shown in figure 2.

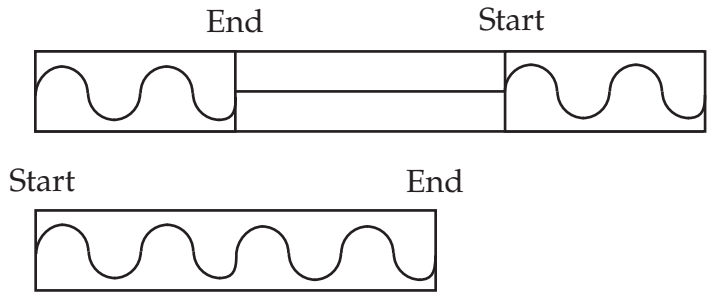

Figure 2: *cropAndErase* with a buffer loop back

A problem arose when copying changing buffer as it was necessary for playback to begin at the next beat, but also necessary to crop and copy the

signal onto a new buffer up until the last sample before the loop would start playing! Attempts were made to read from *recBuf* for the first loop and then change buffer, but the transition could cause audible problems. If the problem is ignored, and cropping occurs when the recording button is pressed (which will always be some amount of time before the next beat) the result is a portion of unnatural silence at the end of the loop. The current solution is to crop and copy to a new buffer as soon as the stop recording button is pressed, begin playing back from the new buffer, and then re-copy the signal once recording has actually stopped. Once looping buffers are finalised they are stored in the **List** *loopBufs*.

```
copyFunc.(newBuf);      //Copy signal so far on the new Buffer
(60/tempo).wait;        //Wait until recording has finished
copyFunc.(newBuf);      //Copy all signal onto the new buffer
recBuf.zero;            //Erase the recording buffer
```

Figure 3: *cropAndErase* signal copying pseudo-code

3.4 Onset Analysis

Once a loop has been recorded it is necessary to detect the onsets so that meaningful sections (those which contain musical events) of the loop may used to generate new material. An instance of the **SynthDef** *onTrig* serves well here.
onTrig takes as input:

- *clockIn:* The index of the **Bus** providing clock information.
- *bufnum:* The index of the **Buffer** to analyse.
- *val:* The threshold level (0-1) for the onset detector

An example instantiation:

```
onsetSynth = Synth(\onTrig, [\clockIn, analysisClockBus,
\bufnum, 2]);
```

Inside *onTrig* exists a **BufRd** object for reading the audio buffer, driven by the clock information from *clockIn*. This signal is converted to the frequency domain via an **FFT** object, which the **Onsets** object is capable of analysing. The threshold of the onset detector is set via a slider on the GUI, where a pseudo LED flashes, triggered via a **SendTrig** in *onTrig*, making it easy to find a correct setting. Whenever an onset is detected, the current frame (provided by the clock) is sent to the language via a **SendTrig** and is stored in a **List** (see figure 4). A **SendReply** object is also used to send the amplitude of the signal to a level indicator on the GUI.

List[2496, 49600, 99776, 112064, 134080, 146368]

Figure 4: Example of a list of onset frames

Operation An instance of *onTrig* is created and stored as *onsetSynth* when the *GCLoop* class is instantiated. After recording has finished, the user needs to set the threshold level via the GUI, and start analysis using the Analyse button. This sets the *nowAnalysing* variable to **true**, telling *oscr* (the **OSC Responder**) to start storing onset frames from the next beat until the loop has played in its entirety. Once the loop has been analysed, the **List** of onset frames is stored in another **List** *onsetsList* the indexes of which correspond to those of *loopBufs*.

3.5 Playback and Generation

Once material is recorded and analysed, the user is able to choose between recorded or generative loops.

3.5.1 Recorded Loops

Recorded loop playback is the continuous playing of the recorded material, in sync with the system's clock. This is the modus operandi of traditional loop pedals. In this system comes courtesy of the *GCPlay* **SynthDef**.

GCPlay takes the following arguments:

- *bufnum:* The index of the **Buffer** to play from.

- *trig:* The index of the **Bus** used to trigger playback.
- *clockOut:* The index of the **Bus** to send clock information from.
- *amp:* The amplitude of the output signal.
- *pan:* The pan position of the signal (-1 is 100% left, 0 is centre and 1 is 100% right).
- *index:* The id number to send via a **SendTrig**.
- *ampLag:* Amount of time to reach the new value when the *amp* argument is changed.
- *panLag:* As with *ampLag* but for the *pan* argument.

An example instantiation:

```
syn = Synth(\GCPlay, [\bufnum, 8, \trigBus, onsetStartBus,
\clockOut, analysisClockBus, \amp, 1, \pan, 0.5, \index, 6,
\ampLag, 3, \panLag, 3]));
```

Inside *GCPlay* is a **BufRd** object for playing the **Buffer** specified by *bufnum*, driven by a **Phasor** object. Uniquely, instead of using the *metroSynth* **Phasor** to drive the **BufRd** directly, a new **Phasor** is used, triggered (via *trigBus*) by the *metroSynth*. This runs in sync with the *metroSynth* **Phasor**, but between the values of 0 and the number of samples in the **Buffer** being read.

Initially the *metroSynth* clock was used directly in the form *metroSynthClock%numberOfSamplesInBuffer*, but this lead to problems with odd length loops (e.g. 3 beats) with even length cycles. For example, given the sample-rate 48000 samples/sec, and a tempo of 60bpm we get 48000 samples per beat. If a loop is started on the first beat of the second bar (sample 191999) and is stopped after three beats (at sample 335999, 144000 samples long), it should begin at looping from sample 0 at the next beat (sample 336000), unfortunately $336000\%144000 = 48000$ (the second beat of the loop). Rather than send further information (time signature, tempo, beat at which recording started) to the **SynthDef** to calculate the necessary offset, it was easier to simply trigger a new **Phasor** from the *metroSynth*.

Operation Instances of *GCPlay* are created as part of the *startRecordingAction* method, to ensure they are ready before they are due to start playing. Each *GCPlay* instance has a **Select** object which selects between two signals to use as amplitude control for the output. The two signals are stored in an array ([*trigBus*'s value, *amp*'s value]), with the index used determined by *trigBus*. The value of *trigBus* is 0 until triggered by *metroSynth* at which point it changes to 1, selecting *amp*'s value.

As multiple instances of *GCPlay* will be used, it is necessary to change the **Bus** passed into *trigBus* to prevent future triggers affecting established **Synths**. For this reason the **SynthDef** *onBus* and **Bus** *alwaysOnBus* are passed to established *GCPlay* instances to provide a continuous "on" signal.

A **SendReply** object is used to send amplitude information to the server to be displayed using a level indicator on the GUI.

Once *GCPlay* instances are established they are stored in a **List** *onsetsList*, the indexes of which correspond to those of the **List** of buffers *loopBufs*, and the **List** of analysis data *onsetFrames*.

3.5.2 Generative Loops

Generative loops are the result of probabilistic creation of *GCHit* **SynthDef** instances using the timing information sent from *metroSynth* to the language. The *GCHit* **SynthDef** takes the following arguments:

- *bufnum:* The index of the **Buffer** to read from.
- *startFrame:* The sample at which to start playback.
- *endFrame:* The sample at which to end playback.
- *rate:* The rate of playback where 1 = normal and 2 = double speed (-1 plays backwards).

An example instantiation:

```
Synth(\GCHit, [\bufnum, 4, \startFrame, 48000, \endFrame, 96000,
\rate, 1.5]);
```

Inside *GCHit* is a **Line** object outputting a signal from *startFrame* to *endFrame* in the required duration time *(endFrame-startFrame)/(sampleRate*rate)*. This controls **BufRd** object that reads from the **Buffer** specified by *bufnum*.

The output signal is multiplied by an **EnvGen** object which envelopes the start and end of the signal with a time of 20ms to avoid unwanted artefacts. Once playback has completed the **Synth** is freed from memory with the *doneAction:2* command of the **EnvGen**.

Operation When the user sets a channel to 'cut', a boolean **true** value is stored in the **List** *cuttingList* at the index of the channel. Upon receiving a beat message from *metroSynth*, the **OSC Responder** calls the *allHitFunc* method.

The *allHitFunc* takes the parameter *index* which distinguishes which beat called the method. It then loops through the **List** *cuttingList*, calling the *randomHitFunc* method using the relevant probabilities from the **List** *channelProbs*.

The *randomHitFunc* method takes the following arguments:

- *chance:* The chance that a 'hit' will be triggered.
- *bufnum:* The index of the **Buffer** to read from.
- *hitList:* The **List** of onsets corresponding to given **Buffer**.

Inside *randomHitFunc* the *chance* is tested, and if **true** an onset point is selected from *hitsList* to be used as the *startFrame*, with the following item in *hitsList* being the *endFrame* (in the case that the last item in the **List** is selected, the *endFrame* will be the end of the **Buffer**). A rate of playback is selected from the **Array** [-1, -0.5, 0.5, 1, 2] using the following weights for selection [0.15, 0.1, 0.6, 0.15]. Finally the method *hitFunc* is called which takes the same arguments as *randomHitFunc* and simply creates an instance of the *GCHit* **SynthDef** using them.

hitFunc and *randomHitFunc* are kept separate for modularity and future revisions which may use different parameter generation methods before calling *hitFunc*.

3.6 Mix Automation

Mix automation is triggered using the "Manual/Automated" button on each channel of the mixer GUI. When pressed the following routine is started or stopped for that channel.

```
Routine {
    var waitTime, ampVal, panVal;
    inf.do {
        waitTime = 5.rand;
        ampVal = 1.0.rand.ampdb;
        panVal = rrand(-1.0, 1.0);
        Synth.set(\ampLag, waitTime, \panLag, waitTime);
        guiMixerAmps[index].valueAction_(ampVal);
        guiMixerPans[index].valueAction_(panVal);
        waitTime.wait;
    };
};
```

This routine randomly selects a time between 0 and 5 seconds until the next value is set, this value is passed to the **Synth's** *ampLag* and *panLag* arguments (in turn passed into **Lag** objects) to smooth the transition between values. Random values are then chosen for the new amplitude and pan settings, which are used to with the mixer GUI object's *valueAction* method to update the GUI which in turn update the **Synth** settings.

4 Limitations and Extensions

The implementation of a looping system with on-the-fly recording and sample-accurate synchronisation of the many recorded layers was a complex and time consuming task, resulting in the use of primitive (albeit effective) generative techniques.

4.1 Material Generation Limitations

The current method assigns a probability of triggering a note/event to the first beat of the cycle, every crotchet (quarter note), quaver (eight note) and semiquaver (sixteenth note). The triggers for these probability layers are kept separate so a stacking of probabilities occurs (e.g. a trigger for the first beat of the bar, a crotchet, quaver and semiquaver will occur simultaneously) this increases the likelihood that at least one event will be triggered, but also makes possible the generation of multiple voices from a single track.

While the results can be pleasing, and fulfil the goal of generating musical variety, they can tend to lack a sense of form. A future extension to

overcome this would the implementation of more sophisticated algorithmic composition techniques. Currently a simple probability table is used, with hierarchy between the note durations (a crotchet is more likely to trigger than a semiquaver), but position in the bar is not taken into account. As a result beats two, three and four of a 4/4 cycle have equal weighting, which is not often the case in human music generation. The exception here is the first beat of the bar, which is weighted heavily to provide stability. Approaches such as probability templates and beat partitioning as implemented by Collins (2003) would provide more structured material, while using Charles Ames' (Ames, 1990; Collins, 2003) statistical balance method could be used to control repetition and guide future choices (Collins, 2003). For a comparison see figures 5 and 6.

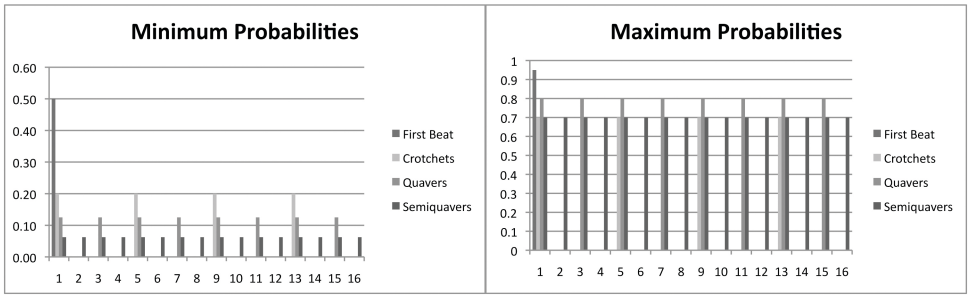

Figure 5: *GCLoop* minimum and maximum probabilities for one 4/4 bar of an individual track

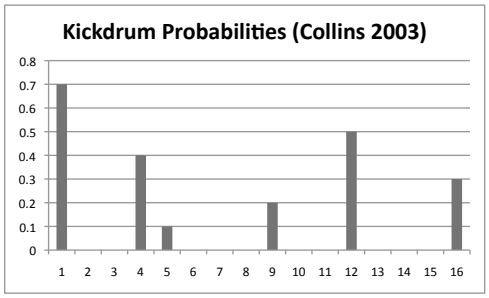

Figure 6: Probabilities for one 4/4 bar of a kick drum track in (Collins, 2003)

4.2 Generative Mixing Limitations

Currently the generative mixing capabilities of the system are limited to adjustments of pan and volume, while providing some sonic variation this could be improved greatly. Machine listening techniques such as those explored by Gonzales and Reiss (2007); Gonzalez and Reiss (2008a,b) could be used to find optimum gain levels and pan positions, while more explicit FX such as reverb could be included and automated. The accuracy of the computer could be exploited to automate FX beyond human capabilities, applying effects to individual events as with BBCut2 (Collins, 2006).

4.3 Onset Detection Problem

The current method of onset detection requires the entire recorded loop to be played before 'cutting' can begin. While this is acceptable for short loops, it is unsuitable for more experimental recordings (e.g. recording audience reactions for use as part of the composition) which are likely to be longer and not well suited to an identical second listening. A simple workaround would be to mute the track while onset detection is taking place, and alter the system so that events are available as soon as they are analysed.

A better but more difficult solution would be to use offline onset detection as with software such as Propellerheads' Recycle (Propellerheads, 2009). The difficult of correct threshold levels arises, but useable solutions to this problem have been discussed (Collins, 2006).

5 Conclusion

In this report two flaws of current loop systems were identified, with possible generative solutions posited. The goal of applying automated generative techniques to address these shortcomings was achieved with pleasing results. The implementation of these techniques shone light on their limitations, resulting in ideas for future revisions. As the benefits of applying generative techniques becomes more apparent to the public, it is hoped that these sorts of processes may be implemented in hardware devices that are more suited to live performance.

References

AKAI. E2 head rush, 2008.

C. Ames. Statistics and compositional balance, 1990.

B. Bel. Modelling improvisatory and compositional processes. *Languages of Design, Formalisms for Word, Image and Sound*, 1(1):11–26, 1992.

J. Biles. Life with genjam: Interacting with a musical ga. In *Proceedings of the IEEE Systems, Man and Cybernetics Conference*, Tokyo, 1999.

N. Collins. Algorithmic composition methods for breakbeat science. *ARiADA*, 3, 2003.

N. Collins. *Towards Autonomous Agents for Live Computer Music: Realtime Machine Listening and Interactive Music Systems*. Phd, 2006.

D. Cope. *Virtual Music*. MIT Press, 2004.

A. Eigenfeldt. The creation of evolutionary rhythms within a multi-agent networked drum ensemble. In *Proceedings of the International Computer Music Conference*, Copenhagen, 2007.

R. Gaskin and R. Fripp. August 1979 interview with robert fripp, 12/08/79 1979.

E. P. Gonzales and J. Reiss. Automatic mixing: Live downmixing stereo panner. In *10th International Conference on Digital Audio Effects (DAFx-07)*, Bordeaux, France, 2007.

E. P. Gonzalez and J. Reiss. An automatic maximum gain normalization technique with applications to audio mixing. In *124th Audio Enineering Society Convention*, Amsterdam, The Netherlands, 2008a.

E. P. Gonzalez and J. D. Reiss. Improved control for selective minimization of masking using inter-channel dependancy effects. In *11th International Conference on Digial Audio Effects (DAFx-08)*, Espoo, Finland, 2008b.

K. King. Private interview conducted with kaki king, 08/07/2008 2008.

Propellerheads. Recycle, March 2009. URL `http://www.propellerheads.se/`.

Roland. Gk-3 divided pickup, 2008.

SC-Developers. *SuperCollider Help Files: Client versus Server Architecture and Operations*. SuperCollider, 2009.

G. Weinberg, M. Godfrey, A. Rae, and J. Rhoads. A real-time genetic algorithm in human-robot musical improvisation. In *Computer Music Modeling and Retrieval. Sense of Sounds: 4th International Symposium, CMMR 2007, Copenhagen, Denmark, August 27-31, 2007. Revised Papers*, pages 351–359. Springer-Verlag, 2008. 1422974.

7 Appendices

7.1 Audio Examples

Track Number:

1. System Example
2. Good Loop / Bad Loop

7.2 Source Code

Begins on following page.