This paper proposes an initial framework for using PonyGE, a python implementation of Grammatical Evolution in generating ChucK files for music creation. We develop a number of grammars for creating individual instruments as shreds that can be added or sporked to the ChucK VM via the command line. A quicktime video example of the system running is provided. We propose that this strongly-timed system can be developed and generalised in the future to fully explore the capabilities of ChucK and to consider implications for Evolutionary Music.

Introduction

Grammatical Evolution (GE) is a grammar based Evolutionary Computation (EC) algorithm (Brabazon, O’Neill, and McGarraghy, 2015). Evolutionary systems, such as GE, develop solutions to a given problem by considering a population of individual solutions over a series of successive generations, rather than trying to deterministically improve one single solution. In recent years, such systems have been applied to creative tasks such as art and music. We present the initial framework for developing an evolutionary system that generates music using the live-coding language ChucK (Wang, Cook, and others, 2003). By the using the on-the-fly command programming capabilities of ChucK, shreds (each written as an individual file) can be added or sporked to a ChucK VM by the user at the command line. We develop a series of grammars that generate individual ChucK files for new shreds, that currently correspond to specific instruments. Each of these shreds can then be added or removed to the running virtual machine (VM) by the programmer.

‘My Little ChucKy’ is so named as it is an early stage system that uses PonyGE — a python implementation of GE in evolving the ChucK files. The following section introduces GE and live coding and describes some previous work in evolutionary composition. The remainder of the paper describes the system framework and proposes how it will be developed in future work.

Abstract

This paper proposes an initial framework for using PonyGE, a python implementation of Grammatical Evolution in generating ChucK files for music creation. We develop a number of grammars for creating individual instruments as shreds that can be added or sporked to the ChucK VM via the command line. A quicktime video example of the system running is provided. We propose that this strongly-timed system can be developed and generalised in the future to fully explore the capabilities of ChucK and to consider implications for Evolutionary Music.

Background

This section introduces Grammatical Evolution, Evolutionary Composition and Live Coding for musical applications.


Grammar

The creative capabilities of GE in part result from the choices offered within the mapping of the grammar. Typically, the genome is represented by a combination of 8 bit integers known as *codons*. These codons select the particular rule for a given expression according to the mod value from the number of choices for that rule.

\[
\text{Rule} = \text{(Codon Integer Value)} \mod \text{(\# of choices)} \quad (1)
\]

Using this we can introduce biases to our grammar by including multiple instances for preferred choices. For example, the operand depicted below offers three choices, two of which are choice1. Thus there is a 2:1 bias towards the selection of choice1 over choice2. Such biases can be exploited in the design of grammars.

<operand>::= <choice1>|<choice1>|<choice2>

The grammar specified for the proposed system is strongly typed to produce legitimate ChucK code as detailed in the next section.

Evolving Music

A number of previous studies have employed EC techniques for melodic composition. One of the most successful and well-known applications is GenJam (Biles, 1994) which
uses a Genetic Algorithm (GA) to evolve jazz solos. This system has been modified and developed into a real-time, MIDI-based, interactive improvisation system that is regularly used in live performances in mainstream venues (Biles, 2013). A modified GA was used in GeNotator (Thywissen, 1999) to manipulate a musical composition using a hierarchical grammar. Göksu et al. evolved and evaluated both melody and rhythm separately using multi-layered perceptrons (MLPs) (Göksu, Pigg, and Dixit, 2005). These evolved melodies were then mixed to produce verses and whole songs. Dahlstedt developed a system that implements recursively described binary trees as genetic representation for the evolution of musical scores. The recursive mechanism of this representation allowed the expression of expressive performances and gestures along with musical notation (Dahlstedt, 2007). Adapted GAs were used with local search methods to investigate human virtuosity in composing with unfigured bass (Munoz et al., 2016), with a grammar to augment live coding in creating music with Tidal (Hickinbotham and Stepney, 2016), and with non-dominated sorting in a multi-component generative music system that could generate chords, melodies and an accompaniment with two feasible-infeasible populations (Scirea et al., 2016).

GE and Music Composition While a number of systems used grammar based systems for music composition McCormack (1996), GE was first specifically used for this purpose in de la Puente, Alfonso, and Moreno (2002). In this paper, GE generated melodies for a specific processor but the melodies produced were not presented or discussed. GE has been implemented for composing short melodies in Reddin, McDermott, and O’Neill (2009). From four experimental setups of varying fitness functions and grammars they determined that users preferred melodies created with a structured grammar. GE was again employed for musical composition using the Wii remote for a generative, virtual system entitled Jive, Shao et al. (2010). This system interactively modifies a combination of piece-wise linear sequences to create melodic pieces of musical interest.

GE was used with a number of different fitness measures in numerous versions of a system that created MIDI melodies by developing a grammar that expanded the genome into a series of notes, turns, chords or arpeggios. Early versions of the system used pre-defined fitness measures based on statistical tonal measures (Loughran, McDermott, and O’Neill, 2015b) or Zipf’s Laws (Loughran, McDermott, and O’Neill, 2015a). Later version of the system were developed to emulate a self-adaptive system, whose fitness measure was based on the concept of conforming to the popular opinion of the population (Loughran and O’Neill, 2016). This resulted in a complex adaptive system that was self-referential and autonomous once it had been initialised. This system was generalised from a ranking-based system to a clustered based system in (Loughran and O’Neill, 2017).

Live Coding

Live coding is a practice where software that creates music (and sometimes visuals) is written and manipulated in real-time as part of a live performance (Brown and Sorensen, 2009). Typically in a performance, the code is made visible on large screens, thus providing a more transparent experience to the audience. This leads to the possibility of glitches (or all-out crashes) within the software during the performance, the re-working or patching of which is part of the skill and nuance displayed by the composer (Collins et al., 2003). While a number of specific coding environments have been developed specifically for live coding music, live coding can be implemented in any computing language. The proposed system is developed using ChucK, a popular, strongly typed live coding programming language (Wang, Cook, and others, 2003).

Method

This section offers an overview of the system, detailing the representation, grammar and fitness function used.

ChucK

The ChucK usage in this version of our system is based on an adaptation of the on-the-fly programming synchronisation examples provided by the creators of ChucK, Perry Cook and Ge Wang, available at http://chuck.cs.princeton.edu/doc/examples/. These initial experiments involved the creation of individual grammars that could re-create a valid variation of each of the provided .ck files. GE was run independently multiple times, the run number specifying the grammar to be used and the name of the .ck file to be created. Each time the GE program is called, six individual .ck files are created. These can then be added or removed as ChucK shreds from VM by the programmer at the command line.

Grammar

We have written six distinct grammars, each of which is run with GE to create a distinct instrument that can be sporked as a ChucK shred. In following the on-the-fly examples, each file begins with a specified code block followed by a while loop. Each grammar specifies the initial conditions for the given instrument and then introduces a flag before continuing to the code loop. Hence the first line in each grammar consists of:

<return> ::= <pre> L <code>

This states that the result will be comprised of some pre-coded specified in <pre> followed by the body of code <code> and separated by the flag ‘L’. The <pre> specifies the details for starting the code. For example in the first grammar, which creates the kick drum beat:

<pre> ::= .5::second => dur T;
T - (now % T) => now;
SndBuf buf => Gain g => dac;
me.dir() + "data/kick.wav" => buf.read;
.5 => g.gain;
The code for this project is available at https://github.com/roisis/PonyGE2/releases/tag/MuMe17. Please note that this code is still in development. A quicktime video example of the system running can be found at https://loughranroisin.wordpress.com/home/projects/. This shows numerous runs of the GE program creating multiple versions of each ChucK Instrument that are added and deleted from the VM at the programmers whim. At the moment, we acknowledge that this system does not, on the surface, achieve substantially more than the code provided by Cook and Wang. However, we consider that the strength and potential in this system lies in the facility for expansion. The grammars specified above are currently limited to conform to the examples. The next step in our development is to create more general grammars that can create valid and more expressive ChucK code. EC has a long history of use for program synthesis and is becoming an increasingly popular approach in Software Engineering (Harman, Manouri, and Zhang, 2012). We plan to contribute to program synthesis in the live coding domain specifically using the ChucK environment.

As the grammars create code that is more generalised, the next most important step we envisage is in the creation of a more meaningful fitness measure. The current system only creates valid instruments, known in each run, so any individual is a valid selection. In EC, however, selection is what runs the evolution, and in EC on creative applications, such
as music production, this selection is based on a subjective measure — what makes one instrument or shred better than another? This point is far from trivial, and in effect causes the crux of the problem in applying EC to creative or subjective tasks. Once we have a more generalised system, we plan to address this issue in developing a more sophisticated fitness function. Ideally we would like to develop a fitness function that measures the worth of an individual, not through some pre-defined numerical measure, but rather from a response of the system to its current state such as that presented in Loughran and Ó’Neill (2016). Ideally this would lead to a self-adaptive system that could generate valid code for creating music with minimal human input.

Expansion in this way moves towards more autonomous music generation: We do not wish to write code that makes music, but to write code that writes code that makes music.

**Conclusion**

We have presented an implemented and working framework for a system that augments live coding with GE, presently entitled ‘My Little ChucKy’’. While the system is still in development, at the time of writing we are currently and rapidly producing a more powerful and generalised system that will take advantage of the capabilities of both ChucK and GE. We plan to have a more sophisticated version of this system ready for presentation at MuMe 2017.

**Acknowledgments**

This work is part of the App’Ed project funded by Science Foundation Ireland under grant 13/IA/1850.

**References**


