Genetic Hierarchical Music Structures

Charles Fox
Robotics Research Group
Department of Engineering Science
University of Oxford
Oxford OX1 3PJ, UK
charles@robots.ox.ac.uk

Abstract

Music has structure at many levels, from grand arrangements of verses and choruses down to patterns in small riffs and themes. A bracketed L-system — the SARAH language — is used to represent compositions in terms of these hierarchical structures, allowing genetic programming to meaningfully mutate and crossbred compositions at all levels of structure. Automated composition becomes possible, using evolution with human aesthetic judgment as a fitness function. SARAH is also human-readable and can be used as a human tool for rapid structural composition development; or as a semi-automated composition system, mixing human and evolved contributions. The system has bred pleasing compositions starting from basic musical materials and has also been used to crossbreed Bach with the Spice Girls: these examples are presented as audio files.

Introduction

The majority of pop songs share an identical top-level structure (or simple variants of it):

```
POP-SONG => {INTRO, VERSE, CHORUS,
VERSE, CHORUS, MIDDLE 8, CHORUS, OUTRO}
```

They also contain similar structure at lower levels. A `VERSE` may be decomposable as follows:

```
VERSE=>{RIFF-A,RIFF-B,RIFF-A,RIFF-B}
```

And a `RIFF` may be broken down into parts for different instruments (played simultaneously, not sequentially; we notate this by using square brackets instead of braces):

```
RIFF=>[VOCALS,BASS,GUITAR]
```

Structure is present even at the level of individual notes:

```
VOCAL-TUNE=>{G,G,A,A,C#,A,F#,G}
```


We could imagine describing a complete composition using such rules. When considered as a generative system (i.e. when used in conjunction with a compiler that creates actual musical output from the rules) they form an L-system (Prusinkiewicz & Lindenmayer 1990). Further, a *bracketed* L-system allows us to base all the rules on a single musical terminal. We use the terminal `NOTE` to represent a single whole-note tonic in C Major (i.e. middle C), played at medium volume on MIDI channel 1. A bracketed L-system is one which allows transforms in the rule definitions, for example:

```
SONG=>{VERSE,(^3)VERSE,(^5)(~2)(R)VERSE}
```

This describes a song consisting of a verse; then a copy of the verse raised my a major third; then another copy raised by a perfect fifth, timestreched by a factor of 2, and retrograded (played backwards). A simple melody fragment can be represented by transformed `NOTE`s:

```
TWINKLE =>{NOTE,NOTE,(^5)NOTE,(^5)NOTE,
(^6)NOTE,(^6)NOTE,(~2)(^5)NOTE}
```

Holtzman's GCDL language (Holtzman 1980) provided a notation for human composers to rapidly develop structured compositions by specifying production rules similar to those shown above. GCDL is a powerful language, allowing complex function definitions and computations. The present work provides a simpler bracketed L-system notation called SARAH, consisting only of term rewriting with transforms, and using a single terminal. Typically, a SARAH representation of a composition is longer than a GCDL program, requiring more rules and more transforms to compensate for the simpler language. The advantage of this representation (aside from its faster learning curve for human composers) is that simple rules are amenable to be operated on by Genetic Programming (Koza 1992), enabling SARAH to be used for automated and semi-automated composition as well as human composition.

The next section describes SARAH in detail. We then discuss its use as an aid to human composition; then as a tool for semi-automated evolutionary composition. Finally we touch on some cultural and philosophical issues raised by this mode of composition.

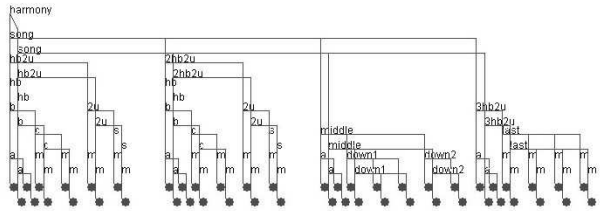

Figure 1: The compiled tree for Happy Birthday

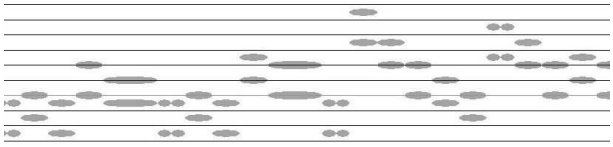

Figure 2: The compiled score for Happy Birthday

The SARAH Language

A composition is represented in SARAH by a list of context-free term-rewrite rules. The right-hand sides of the rules are enclosed in braces or square brackets to represent whether the structure is to be played melodically (in series) or harmonically (in parallel) respectively. Within these brackets is a list of structure names. The names must refer to structures that are previously defined in the list, or may be the terminal `NOTE`. Each name may be prefixed by a list of transforms. For example, here is Happy Birthday written in SARAH:

```
a => {NOTE, NOTE}
m => {(~2)NOTE}
down1 => {(^5)NOTE, (^3)NOTE, NOTE}
down2 => {(^-2)NOTE, (^-3)NOTE}
s => {(~2)m}
c => {(^2)m, m}
b => {a, c}
2u => {m, (^-2)s}
hb => {(^-4)b}
last => {(^3)m, m, (^2)m, m}
hb2u => {hb, 2u}
2hb2u => {hb, (^2)2u}
middle => {(^-4)a, (~2)down1, (~2)down2}
3hb2u => {(^4)a, last}
song => {hb2u, 2hb2u, middle, 3hb2u}
harmony => [song, (^3)song]
```

This generates the tree shown in fig. 1. Applying the transforms to the leaves gives the melody shown in fig. 2. Note that there is no nesting of brackets or transforms *within* each rule, as is generally allowed in bracketed L-systems.

The possible transforms are:

- `(^n)` shift pitch by *n* scale degrees
- `(&n)` shift key by *n* scale degrees
- `(~n/m)` timestretch by rational *n/m*
- `(n/m)` multiply amplitude by rational *n/m*
- `(R)` retrograde (play backwards)
- `(I)` inversion (reflect pitches about the tonic)

We use a *scale-degree* representation of pitch. Each note is represented by a 6-tuple: (key, degree, time, duration, volume, channel). Key is an integer MIDI note number which represents the root note of the (relative) Ionian major scale that the note is in. For example, for a piece in C major, D Dorian or A Aeolian it could be 60 (C3/middle C). For a piece in D Major, E Dorian or B Aeolian it could be 62 (D3). The pitch is then determined by the scale degree within that key. So in key 60 (C3 Ionian), a note of degree 3 would be the major third of the scale rooted on C3, i.e. E3. A note of degree 10 would be major tenth, i.e. E4. At present the scale pattern is fixed to the Ionian scale, allowing melodies in standard western modes to be represented without changing key. To represent accidentals, one must make a key change as well as a degree shift. For example, the following shows how to notate the melody A3,F3,B3 in G Major:

```
TUNE=>{(^2)NOTE,(&-5)(^4)NOTE,(^3)NOTE}
SONG=>{(&5)TUNE}
```

Recall that each `NOTE` starts out as the tonic in C3 Major. So `SONG` shifts the key of `TUNE` up a fifth to G3 Major. `TUNE` itself defines the melody, the second note of which is the fourth degree of a new key - changed down a fifth back to C Major. The accidental is thus conceived of as occurring in a temporary mode change.

We use the scale-degree representation for two related reasons. First, we find it more cognitively useful for human composers to think in terms of scale degrees rather than absolute pitch. It provides a higher level of abstraction for the composer to make use of: once she has changed to F minor she can then think and write in terms of that scale rather than have to worry about how to translate from that scale into absolute pitches. Secondly, we find it gives rise to much more aesthetic compositions from the genetic composition system discussed later. This is because it allows the system to make musical changes at this cognitively meaningful layer. So for example, it may change the key or mode of a whole section of music with a single mutation; or shift the degrees of random notes *within* their home scale.

Human Composition

We have created a GUI environment for human composers to quickly edit, compile and view SARAH compositions (fig. 3). The user can examine the tree and score representations of the complete score or sub-scores, and may also force particular types of genetic operations to occur. This environment is called *Music Genie*. Audio example 1 presents a short minimalist-style composition written by a human using Music Genie. The composer commented that he was able to develop this composition much faster than by hand due to his enhanced abilities to replicate and transform existing structures.

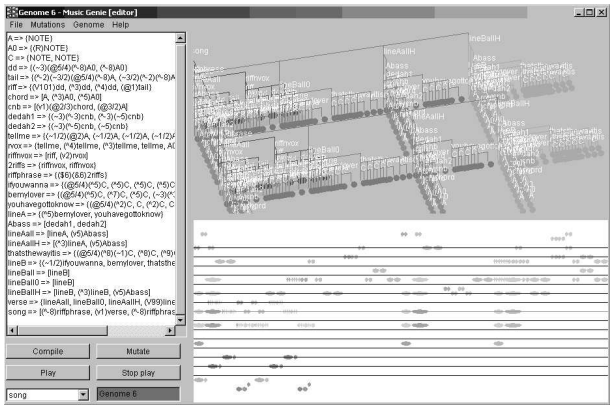

Figure 3: Music Genie's composition environment

Evolution

Following Koza's conception of Genetic Programming (Koza 1992), we consider consider each rule in a SARAH composition to be a gene, and the complete composition as a genome. As with Genetic Algorithms, Genetic Programming requires four operations to be specified: mutation methods, crossover methods, a fitness function and an initial population.

Mutation

We use the following mutation types: Random transforms may be inserted into genes. Existing transforms may have their numeric arguments altered. Transforms may be deleted. Random structure names may be inserted into genes (subject to their structures being defined higher in the genome). Structure names may be removed from genes. Melodic and harmonic structures may be flipped (i.e. the square brackets replaced by braces and vice versa). New random genes may be added to the genome. Genes may be deleted from the genome (subject to all references to them being removed also). Each of these mutation types has a probability, set by hand.

Crossover

Crossover is more complex. We would like to combine two different compositions in a musically meaningful way. Because our genes represent hierarchical structures, and we are using the scale-degree representation, it is possible to do this in an elegant way. To create a child from some parents, we choose random genes from each parent and put them together to form a new genome. When a selected gene from parent *p* refers to other structures from *p* whose genes were also selected, then their names are preserved in the child. When it refers to structures whose genes were not selected, we must change the names to refer to an 'analogous' structure in the child. Choosing the 'most analogous' structure is a difficult problem: analogy-making is an active area of research (Hofstadter 1996), (Gentner, Holyoak, & Kokinov 2002) and we do not yet know the best way to go about it. At present we use a simple heuristic. Recall that the genomes are *ordered* lists. For example, genes may only refer to names of genes defined earlier than themselves in the genome (to prevent circular definitions). This ordering gives an admittedly crude heuristic for analogy-making. Call each genes position in the genome its *level*. To perform crossover, use the following algorithm, whose action is illustrated in fig. 4:

for each child c **do**
  for each level i **do**
    $p \leftarrow$ randomly chosen parent
    $g \leftarrow i^{th}$ gene of p
    copy g into c's genome
    for each name n on the RHS of g **do**
      $gr \leftarrow$ the gene in p with name n
      if gr was previously copied to c **then**
        do nothing
      else
        $j \leftarrow$ the level of gr in p
        $p_{old} \leftarrow$ the parent whose level j gene was copied to c
        $n_{old} \leftarrow$ name of the level j gene in p_{old} in c's copy of g, substitute n_{old} for n
      end if
    end for
  end for
end for

Fitness Function

Whilst it may be possible to automate some human aesthetic preferences and encode a fitness function based on them, we take a human-driven approach to evolution. Music Genie includes a GUI (fig. 5) that presents users with a current population of compositions and asks them to select their favorites. These selections are used as parents to breed the next generation. We do however use a single, very basic heuristic that screens out compositions containing less than 5 notes. This was added because the heuristic cross-over system described above can sometimes lead to badly-drawn analogies which produce very short compositions. An interesting open question is to what extent automated fitness functions are possible: could and should we for example screen out compositions that change key too often, or whose rhythmic structures are irregular? There is of course a trade-off here: restricting the space of possible compositions in exchange for requiring the user to spend less time selecting out 'obviously' bad ones.

We have also implemented a web-based client-server version of this GUI that allows many users to collaborate on evolving compositions.

Initial Population

The evolution system may be initialized by loading a set of human compositions as the initial parents; alternatively, single `NOTE` terminals could be used, so that the compositions grow entirely from evolved material. In practice we prefer the first option as it would take many generations of uninteresting compositions before the second method produced interesting results. Cross-over and mutation may also be

(1) Align parents:

Parent1:		Parent2:	
RIFF	=> NOTE	TUNE1	=> NOTE
CHORUS	=> {(^5)RIFF}	A	=> {TUNE1, (&5)TUNE1}
VERSE	=> {RIFF, (^3)RIFF}	B	=> {B, (^5)B}
SONG	=> {VERSE, (&2)VERSE,CHORUS}	RONDO	=> {A,B,A,B,A}

(2) Convert to standard form and select genes:

Parent1:		Parent2:	
1	=> NOTE	1	=> NOTE
2	=> {(^5)1}	2	=> {1, (&5)1}
3	=> {1, (^3)1}	3	=> {3, (^5)3}
4	=> {3, (&2)3,2}	4	=> {2,3,2,3,2}

(3) Create child from selected genes:

1	=> NOTE
2	=> {1, (&5)1}
3	=> {3, (^5)3}
4	=> {3, (&2)3,2}

(4) Rename with original string names:

RIFF	=> NOTE
A	=> {TUNE1, (&5)TUNE1}
B	=> {B, (^5)B}
SONG	=> {VERSE, (&2)VERSE,CHORUS}

Figure 4: Example of the crossover process.

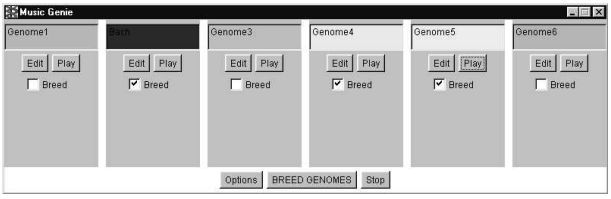

Figure 5: Music Genie's evolution manager

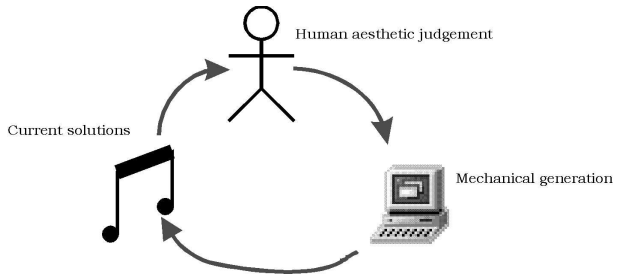

Figure 6: The creative process for semi-automated composition

run manually from the human composition environment in Music Genie, allowing a human composer to generate new works by controlled evolution of his existing works.

Results

Audio example 2 presents a typical example of a user working with the web-based selection system to evolve a new composition. Audio example 3 presents the result of such an evolution after about 100 generations. Audio example 4 illustrates the effects of various random mutations on an extract from Bach's *Well Tempered Clavier*, and the effect of crossbreeding it with an extract from the Spice Girls song *Wannabe*. Most listeners find these compositions to be reasonably aesthetically pleasing.

Discussion

Note that at present, the input compositions must be manually parsed into SARAH, by musically-trained humans. This is time consuming and appears to require considerable intelligence and pattern-recognition skills. Future work could examine the use of grammar induction methods to automate some or all of this process. See for example (Reis 1999) for an agent-based discussion of this area.

We present SARAH and Music Genie not only as a useful tool for human composers and a fun way of evolving short compositions, but as raising questions about the nature of creativity: to what extent should creativity be credited to the machine and to the human(s) who participate in evolutions?

A criticism of machine composition (e.g. (Cope 2001)) is that it cannot currently be called 'artistic' because it does not deliberately *express* anything: it does not start with a conceptual model of emotions or stories to be conveyed by the language of music. If a machine was to produce Beethoven's Ninth by chance, it would still not be artistic unless it had started with a model of what it is like to be hurtling through a thunderstorm in the rain. Rather, this creative credit should be granted to the human *curator* who selects that chance composition — probably after considerable effort spent listening to millions of others — and perceives it as expressing those concepts. The chance composition acts like a 'found object'. Hence the composer becomes synonymous with the curator.

Music Genie suggests a further shift in this direction, from the curator-as-composer to the listener-as-composer (fig. 6). Reminiscient of Turing's notion of mathematical creativity (Turing 1938), listener-composers inject emotion and meaning to the creative process, but the laborious work of creating the actual structures and notes is performed by the machine. The composition could exist as a permanently-evolving population, and every listening acts as a further step in the evolution. The credit for composition should then be assigned to the community of listeners themselves rather than an *èlite* creator or curator.

References

Cope, D., ed. 2001. *Virtual Music: Computer Synthesis of Musical Style*. Cambridge, MA, USA: MIT Press.

Gentner, D.; Holyoak, K. J.; and Kokinov, B. K., eds. 2002. *The Analogical Mind: Perspectives from Cognitive Science*. Cambridge, MA, USA: MIT Press.

Hofstadter, D. R. 1996. *Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought*. New York, NY, USA: Basic Books, Inc.

Holtzman, S. 1980. *Generative Grammars and the Computer Aided Composition of Music*. Ph.D. Dissertation, Edinburgh University.

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press.

Prusinkiewicz, P., and Lindenmayer, A. 1990. *The algorithmic beauty of plants*. New York, NY, USA: Springer-Verlag New York, Inc.

Reis, B. Y. 1999. A multi-agent system for on-line modeling, parsing and prediction of discrete time series data. In *Intelligent Image Processing, Data Analysis and Information Retrieval*. IOS Press Holland. pp. 164–169.

Turing, A. M. 1938. *Systems of Logic based on Ordinals*. Ph.D. Dissertation, Princeton University.

Appendix

The following is the code for the Bach example:

```
a => {($6)(&5)NOTE}
Bold => {a, (^3)a, (^-4)a, (~2)(^-5b)a,
         (~3)(^-4)a}
Cold => {(~1/2)(^-3)a, (~1/2)(^-2)a, a,
         (~1/2)(^-2)a, (~1/2)(^-3)a, (^-2)a}
Dold => {(^3)Cold, (^6)Cold}
B => {(^5)a, (^5#)(@5/4)a, a,
      (~2)(^-2b)(@5/4)a, (~3)(@5/4)a}
C => {(~1/2)(^2)a, (~1/2)(^3)a, (^4)a,
      (~1/2)(^3)a, (~1/2)(^2)a}
D => {(^3)(~2)a, (~3)a, (~1/2)(^2)a,
      (~1/2)a, (^-2)a, (~1/2)a,
      (~1/2)(^2)a, (^3)a, a,
      (^-3)a, (^4#)a}
E => {(^5)(~3)a, (^6#)a, (^7)a, (^6#)a,
      (^7)a, (^8)a, (^9)a}
t => {B, C, D, E}
tt => {t, (&5)t}
pause => {(@0)(^5)B, (@0)(^5)C, (@0)a}
delayed => {pause, Bold, Cold, Dold}
z => [tt, (v1)delayed]
```

The following is the code for the Spice Girls example:

```
A => {NOTE}
C => {NOTE}
dd => {(~3)(@5/4)(^-8)A, (^-8)A}
tail => {(^-2)(~3/2)(@5/4)(^-8)A,
    (~3/2)(^-2)(^-8)A, (^-2b)(^-8)A}
riff => {dd, (^3)dd, (^4)dd, tail}
chord => [A, (^3)A, (^5)A]
cnb => [(v1)(@2/3)chord, (@3/2)A]
dedah1 => {(~3)(^-4)cnb, (^-3)(~5)cnb}
dedah2 => {(~3)(^-5)cnb, (~5)cnb}
tellme => {(~1/2)(@2)A, (~1/2)A, (~1/2)A,
  (~1/2)A, A, (~1/2)A, (~1/2)A}
rvox => {tellme, tellme, tellme, tellme}
riffnvox => [riff, (v2)rvox]
2riffs => {riffnvox, riffnvox}
riffphrase => {($6)(&6)2riffs}
ifyouwanna => {(@5/4)(^5)C, (^5)C,
               (^5)C, (^5)C}
bemylover => {(@5/4)(^5)C, (^6)C, (^5)C,
              (~3)(^3)C}
youhavegottoknow => {(@5/4)(^2)C, C,
                (^2)C, C, (^3)(~3)C}
lineA => {(~1/2)ifyouwanna, bemylover,
          youhavegottoknow}
Abass => {dedah1, dedah2}
lineAall => [lineA, (v5)Abass]
lineAallH => [lineA, (^3)lineA, (v5)Abass]
thatsthewayitis => {(@5/4)(^8)(~1)C,
 (^8)C, (^9)C, (^5)C, (^6)(~2)C, (^5)(~2)C}
lineB => {(~1/2)ifyouwanna, bemylover,
   thatsthewayitis}
lineBall => [lineB, (v5)Abass]
lineBallH => [lineB, (^3)lineB, (v5)Abass]
verse => {lineAall, lineBall,
          lineAallH, lineBallH}
song => {riffphrase, (v1)(^8)verse,
         riffphrase, (v1)(^8)verse}
```

The MP3 audio examples and the Music Genie software are freely available from `www.charlesfox.org.uk`.