Exploring the sound-space of synthesis algorithms using interactive genetic algorithms.

Colin G. Johnson.
Department of Computer Science,
University of Exeter, The Old Library,
Prince of Wales Road, Exeter, EX4 4PT.
Email : C.G.Johnson@ex.ac.uk

Abstract

Exploring the sounds available from a synthesis algorithm is a complicated process, requiring the user either to spend much time gaining heuristic experience with the algorithm or requiring them to have a deep knowledge of the underlying synthesis algorithms. In this paper we describe a computer system which facilitates a more exploratory approach to sound design, allowing the user to work at the level of the sounds themselves. In this system the synthesis parameters are managed by a genetic algorithm, which is directed by the users' judgements about the sounds in the system. We describe the development of a prototype version of this system, concentrating on an interface to the FOF granular synthesis algorithm from *CSound*.

1 Introduction

Most sound synthesis algorithms are complex, and typically have correspondingly complex interfaces, requiring the specification of many parameters to achieve a desired sound. To produce a sound using such an algorithm typically requires one of two processes to be carried out. Either the user must understand the technical details of sound production (as detailed in e.g. (Wishart, 1994, 1996)) or they must gain sufficient heuristic experience with the algorithm by studying existing applications and experimenting with modifications to the parameters. In this paper we would like to present a new type of interface for sound synthesis algorithms, which facilitates an exploratory approach to the sounds that are able to be produced by the algorithm. This approach is based on *interactive genetic algorithms*. In sections 2 and 3 we review the idea of interactive genetic algorithms and some applications thereof, then in section 4 we describe a simple implementation of the idea. Sections 5 and 6 contain a discussion of the system and a detailed description of ongoing work.

One of the advantages of such a system is that it facilitates the exploration of the timbral characteristics of a synthesis algorithm. It is easy to explore the space of all melodies by picking out patterns on a guitar or piano, and rhythmic exploration is literally at our fingertips. The idea of exploring timbral and spectral qualities of sound is, however, typically restricted to choosing from a discrete set if instrumental colours, a restriction bemoaned by Trevor Wishart in his recent book *Audible Design* (Wishart, 1994) :

> "The spectral characteristics of sound have, for so long, been inaccessible to the composer that we have become accustomed to lumping together all aspect of the spectral structure under the catch all term "timbre" and regarding it as an elementary, if unquantifiable, property of sounds. Most musicians with a traditional background almost equate "timbre" with instrument type ..."

It is hoped that the development of systems such as those described in this paper will provide powerful tools for the exploration of such characteristics of sound. This leads the user towards the production of genuinely *new* sounds which are not the product either of analogies with conventional instruments, nor with the facile exploitation of obvious properties of the algorithms used.

2 Interactive genetic algorithms

Genetic algorithms (see (Goldberg, 1989; Mitchell, 1996) for comprehensive surveys) are a computing methodology inspired by evolutionary biology. Their origins lie in finding good solutions to complex engineering design problems, where it is possible to decide how good a particular potential solution is, but where there is little explicit information available from that solution as to how to improve the solution. Instead of trying to make improvements on a single design, a population of solutions is maintained, and an iterative process of choosing the best solutions, then creating a new population of (better) solutions by combining aspects of the good solutions and mutating existing good solutions. The underlying representation of these solutions is as bit-strings, which encode a list of parameters defining a solutions. Here is some pseudocode.

```
BEGIN
Create a random set of
 initial solutions
   LOOP :
      Choose a subset of good solutions
       according to some
       ``fitness measure''.
      Perform recombination on randomly
        chosen pairs of solutions.
      Perform mutation on randomly
        chosen solutions.
   UNTIL (population is stable)
END
```

In traditional applications of genetic algorithms choice of solutions is carried out by a fixed *fitness function*, which measures how good a solution is at solving a particular problem. However in the work below we will use the idea of *interactive* genetic algorithms, where the fitness measure is given by a human user interacting with the algorithm. In traditional genetic algorithms the fitness measure is a given by a fixed function, for example in an engineering design problem the fitness of a solution is a measure of how successful that particular design is. It is assumed that a good design will be made up of good components, and thus the probability of a particular design being used as the "parent" of a design in the next generation depends upon its fitness. In an interactive genetic algorithm a number of "solutions" are presented to the user of the system, who then assigns fitness values or rankings to the various solutions based either on aesthetic judgements or on the subjective sense of how close the solution is to a desired ideal solution.

3 Review

The idea of "evolutionary interfaces" is not new. The first implementation of these ideas is due to Dawkins (Dawkins, 1989, 1990), who used created the "biomorphs" system to explore some ideas in evolutionary theory. This system consists of simple two dimensional bitmap pictures, which are initially generated by uniform random selection from the set of all such pictures. The user then chooses which pictures they find most aesthetically pleasing, or which are closest to some ideal picture that they are looking for. The system then takes each picture and assigns them a weight according to the score/ranking given to them by the user. The next generation of pictures to be presented to the user consists of new pictures which have been created from the originals by choosing pairs of pictures and combining them together by choosing regions from each of the two "parent" pictures at random.

Initially this work was designed as a tool for illustrating and exploring theories in evolutionary biology, but Dawkins began to regard it as an artistic tool. A similar idea was being explored at around the same time by Todd and Latham (Todd and Latham, 1992), who used sophisticated computer-aided design systems combined with an evolutionary interface to create abstract pictures, including both still pictures and animations. Other similar work is described in Sims (1991). A more down-to-earth application of similar techniques is described in Caldwell and Johnston (1991), who have created an elegant system which allows witnesses to a crime to explore the space of possible faces as an aid to the identification of suspects.

A number of researchers have made use of the idea of combining genetic algorithms as a creative tool in music. A number of projects (Horner and Goldberg, 1991; Nelson, 1993, 1995; Putnam, 1994) have explored the use of evolutionary interfaces as a way of exploring the space of melodies. A more sophisticated implmentation of this is the work in Biles (1994, 1995); Biles et al. (1996); Biles (1998), who has created a system called *GenJam* which uses similar techniques to generate jazz solos, with the fitness function being supplied by a trainer or (more interestingly) by audience input. Two other projects have used genetic algorithms for musical purposes. Horowitz (1994) describes the creation of a tool to facilitate the evolution of rhythmic patterns and Takala et al. (1993) have looked into matching musical ideas to animation. However none of these studies have applied these techniques to providing a better interface to the synthesis of sounds themselves, the sounds being generated using standard synthesis or sampling techniques.

4 Some experiments

In order to experiment with the idea of an evolutionary interface for the generation of individual sounds we have implemented a simple system based on the well-known (and freely available) *CSound* sound synthesis system (Vercoe, 1992). A *CSound* program consists of two files. The first is the *orchestra* file which specifies a number of virtual instruments which are created using various synthesis and sound processing algorithms either drawn from the large number included in the system or written by the user using the *C* programming language. The second file is the score file which initiates these virtual instruments by specifying when they should play and what parameters they should use to create that particular sound-event.

We have been investigating a number of the algorithms available in the *CSound* language, concentrating on those where many parameters are required to specify a particular sound. Algorithms which seem particularly appropriate for this method are the `FOF` granular synthesis algorithm and the Karplus–Strong plucked sound algorithm, implemented in *CSound* as `pluck`.

4.1 Implementation

For the purposes of this paper we shall describe the implmentation of an interface to the FOF granular/formant synthesis algorithm (Clarke, 1992). Granular synthesis (Roads, 1978) is a technique for synthesizing sounds which is based on the idea of creating a complex sound by creating a random cloud of tiny sounds within given parameter ranges (see figure 1). In this case the sounds consist of thousands of short sinewave bursts, and we can adjust many parameters (e.g. fundamental frequency, pitch range, length of grains) to change the characteristics of the overall emergent sound.

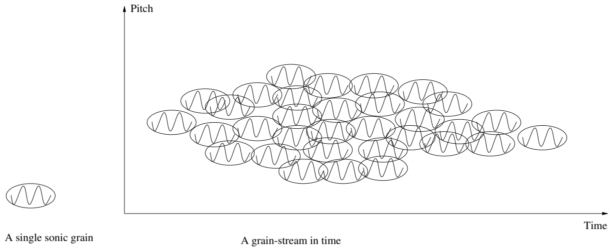

Figure 1: Granular synthesis.

To specify a sound-generation process using FOF we use the following command in the *CSound* programming language,

> ar **fof** *xamp*, *xfund*, *xform*, *koct*, *kband*, *kris*, *kdur*, *kdec*, *iolaps*, *ifna*, *ifnb*, *itotdur* *[, iphs] [, ifmode]*

where each of the italicised expressions is a parameter, for example *xfund* specifies the fundamental frequency around which the grains are generated (details of all the parameters can be found in (Vercoe, 1992; Clarke, 1992)). To add to the complexity some of these parameters can take values which are non-constant during a sound-event, so for example the fundamental frequency may descend, ascend, leap about or whatever to achieve a particular effect. Such events are achieved by setting the parameter value equal to a time-varying curve or stochastic process instead of simply specifying a fixed value. In our first implementation of the algorithm, however, all of the parameters are encoded as constants.

Each sound is encoded as a $112 (= 16 \times 7)$ bit binary string, which is turned into a sound via three stages. In the first stage the binary string is split into 7 16-bit substrings, and these are then converted into 7 integers $n_0, \ldots, n_6$. These numbers are then scaled in the following way, to provide parameter ranges which produce sensible results from the FOF algorithm. These scalings were created through empirical experience with the algorithm.

$$
\begin{aligned}
xamp &= \frac{n_0}{2.5} + 5000 \\
xfund &= n_1/70 + 200 \\
xform &= n_2/35 + 200 \\
koct &= \text{int}(n_3/9000) \\
kband &= \text{int}(n_4/2000) \\
kdev &= 100/n_5 \\
kdec &= 100/n_6
\end{aligned}
$$

Once the string has been converted into a set of parameters these parameters are strung together to create an FOF command.

> ar **fof** *xamp*, *xfund*, *xform*, *koct*, *kband*, *kris*, *kdur*, *kdec*, 5, 1, 2, p3

This command is then embedded into a standard orchestra file, a modified version of the example file given in (Clarke, 1992).

The system begins by generating an initial population of 9 sounds, chosen by randomly generating 112-bit binary strings with a uniform probability distribution. These sounds form the initial population. We then commence a loop. In each round of the loop orchestra files corresponding to those sounds currently in the population are created, and the sounds are then played using a standard score file which plays each sound for one second, The user then assigns a numerical rating r_s, where $s = 1, \ldots, 9$ to each sound, which measures how close the sound is to the desired sound, or how aesthetically pleasing the sound is. The ratings are then added to give a total t, and each sound-string is assigned a probability $p_s = \frac{r_s}{t}$ for each $s = 1, \ldots, 9$.

These probabilities are then used to choose pairs of "parents" for the next generation. For each member of the new population two "parent" strings $\mathcal{P}_1, \mathcal{P}_2$ are chosen, with probability p_s of choosing sound s (note that $\sum_{s=1}^{9} p_s = 1$). This is (Goldberg, 1989) *roulette-wheel selection* (see figure 2). Then a random number in the range $0, \ldots, 112$ is chosen, and the parent strings are sliced at that distance along the string, the beginning of $\mathcal{P}_1$ being attached to the end of $\mathcal{P}_2$ and vice versa. These new strings then become member of the new population. The process is repeated until a new population is created (an alternative (Syswerda, 1989, 1991) would be to replace one member of the population at a time, i.e. in this application replacing the least desired sound with a new sound). The idea behind this recombination procedures is that it facilitates desired parameter values from two different sounds being brought together.

In the final stage of the loop each bit of the string representing each sound in the population is mutated (flipped from 1 to 0 or vice versa) with a probability of 0.1 per bit. This facilitates "local" exploration of the sound-space, as a small number of bit-flips will change the parameters only slightly. The mutation probability was chosen by empirical experience with the system, and is much higher than the typical value found in non-interactive genetic algorithm systems. A variant on this mutation scheme is discussed below.

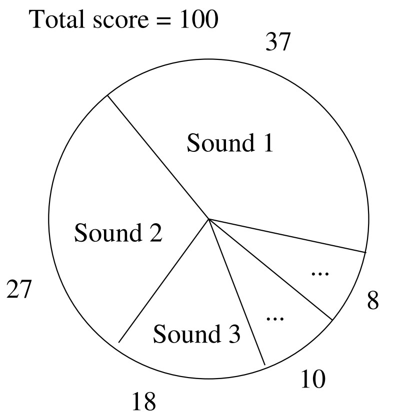

Figure 2: Roulette wheel selection based on scores allocated to the various sounds heard by the user. A sector of the wheel is chosen at random, the probability of selection being weighted by the score allocated to that sound. The sound corresponding to that sector is then used as one of the parents for the new sound.

As a summary, here is a pseudocode description of the algorithm.

```
BEGIN
Create a random set of initial sounds
 LOOP :
   Decode the solutions, and write
     a CSound orchestra file defining
     the instruments.
   Play each of the 9 sounds using
     a system call to CSound and a
     standard score file.
   Ask the user to assign scores to
     the current sounds in the
     population.
   Choose pairs of sounds for
     recombination using the
     roulette-wheel selection.
   Perform mutation on randomly chosen
     solutions.
  UNTIL (a desired sound is found)
Store the orchestra file which
 generated the sound.
END
```

This system has been implemented in *Java*, using system calls and file dumps to interact with *CSound*. When a solution-string is decoded the program writes an appropriate orchestra file. Here is a code-fragment :

```
String tempString = new String();
String fileName = new String();

fileName = "sound"+whichSound+".orc";
DataOutputStream orcFile = new
 DataOutputStream(new
 FileOutputStream(fileName));

tempString =
 "sr = 22050\nkr = 441\nksmps = 50\n\n";
tempString += "instr 1\n";
tempString += "a1 fof ";
tempString +=
  currentSolutions[whichSound]
  .parameterList();
tempString += "\n out a1 \n endin\n";

orcFile.writeBytes(tempString);
```

The *CSound* program itself is then called :

```
String tempString = new String();
tempString = "csound -v -o
              devaudio sound";
tempString += whichSound;
tempString += ".orc mainScore.sco";
Process p;
p = Runtime.getRuntime()
     .exec(tempString);
p.waitFor();
```

Regrettably the use of the `exec` command (which dumps a string onto the command line) interferes with the platform-independence of the program, as different kinds of computer require the *CSound* program to be called in different ways. This command is used to call *CSound* and play a one-second "note" using the orchestra file that has been written.

4.2 Interface

The interface for the system is illustrated in figure 3. At any one time the user is able to play any one of 9 sounds by pressing the buttons along the top of the window, and to rate these sounds by sliders situated below the button.

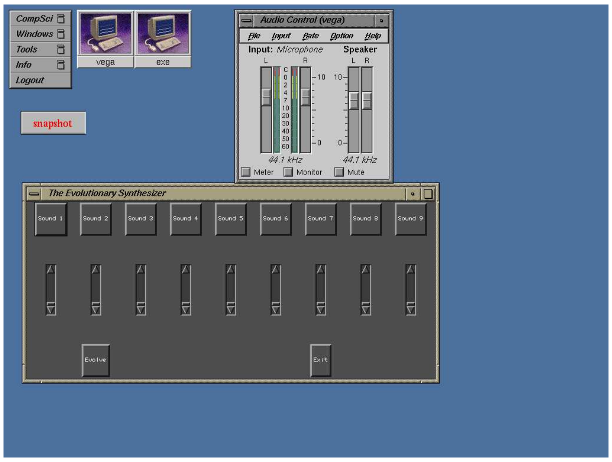

Figure 3: Interface for the evolutionary synthesizer.

An *evolve* button is at the bottom of the window, which causes the genetic algorithm to act on the sounds using the current fitness values as set by the sliders. After this has happened the sliders are reset to zero. There is an implicit *fitness scaling* process going on here. In traditional genetic algorithms fitness scaling Goldberg (1989) is a process where the mapping between fitness values and probability of being chosen is nonlinear, in a way that means that small difference in fitness values close to "the optimum" are weighted more that similar small differences far away from this optimum. The idea is that as the solutions become more fit then the overall fitness spectrum of the population decreases, making these small differences more important. This effect happens automatically here. A user searching for a particular sound will at first rate any sound that sounds vaguely like the desired one highly. However after a few rounds, when all the sounds are within a neighbourhood of the desired sound, the user make much finer judgements as to which sounds are really close to the desired one.

Figure 4 illustrates some extensions which we are currently working on to improve the flexibility of the interface. One aspect of this is allowing more careful operations on the individual sounds. The four buttons in the lower left-hand corner of the window facilitate this. If a sound is particularly interesting, then we can apply a mutation operator to just that sound in order to explore sounds that are close to it in parameter space. If a sound doesn't fit in with the currently explored sound at all, or we would like to introduce some completely new material, then we can choose to randomize a particular sound. Another facility is to store sounds that are of interest, and then reinclude these sounds at a later time. The idea is that particularly interesting sounds can be dragged into the "sound storage" box in the middle left of the interface, and then reintroduced into the population at a later time. Applications of this idea are discussed in section 5 below.

An important aspect of human computer interaction is the idea that a computer system should allow the user to make use of their increased knowledge about the system as they use it more frequently (Preece et al., 1994). In this case, as the user becomes more experienced with and knowledgeable about a particular synthesis algorithm they may gain some feel for how the various parameters affect the sound. Such knowledge can be exploited by adjusting the controls to the bottom right of the window, which affect not the absolute values of those parameters but *the amount the algorithm explores* those regions of parameter space, i.e. the extent to which corresponding regions of the genome are mutated. There is also a slider to adjust the overall exploration rate, which adjusts the rate of mutation and recombination within the population.

4.3 Evaluation

It had been hoped before commencing the experiments with this technique that two features would be observed. Firstly that the initial random process would generate a population that was sufficiently heterogeneous to give the user a basis for making the ratings. Even with the naive algorithm used above, which chooses sounds at random with only crude restrictions on the parameter space, a wide variety of sounds is produced, ranging from continuous sounds of various degrees of "smoothness" through to short percussive sounds.

The second feature that was hoped for was that navigation through the sound-space would be practical and easy. Again this has turned out well. If the user chooses a particular type of sound from the initial population then it is simple to converge to sounds having generally similar characteristics within three or four iterations of the algorithm, and a few more iterations allow a more refined exploration of that particular region of sound space. It is also easy to explore the general capabilities of the synthesis algorithm—after a few sessions with the system the user will have built up a good general feel for the types of sounds that FOF can generate, which is otherwise a long process.

At present we are working on a more formal evaluation of this system, which will compare these interfaces with other ways of interacting with *CSound*. It is important that this evaluation includes feedback from users who are both experienced and not experienced with the *CSound* system, as the system is designed to be both a tool capable of being used by experienced electroacoustic musicians, as well as a tool for beginners who want to learn about the scope of the synthesis algorithms available in *CSound*.

5 Discussion and ongoing work

Some of our ongoing work is concerned with extending the range of synthesis algorithms that can be explored using this kind of methodology, and in extending and evalu-

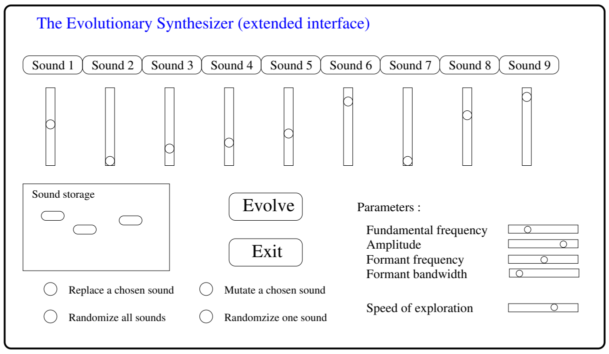

Figure 4: Extending the interface.

ating the different ways of interacting with the population of sounds as discussed above. In this section we will discuss some further extensions to this work.

One idea discussed above was the addition of a facility to allow the user to store certain sounds and put them into the population at appropriate times to guide the evolution of a population in a particular direction. The aim here is to be able to design sounds by analogy. A similar aim has been described by Eduardo Reck Miranda (Miranda, 1995a) in the context of creating an inductive learning system for sound synthesis:

> "Producing a desired sound on a musical instrument, a clarinet for example, fundamentally depends on sub-cognitive physical skills ... If we ask out clarinettist to play a melancholy sound she would have no difficulty imagining a suitable sound and producing one that satisfies our request. If instead our clarinettist turns to computer sound synthesis, this lack of an explicit understanding of how imagined sounds are produced and described presents significant problems."

The idea is that we may come across sounds that are "melancholy" in character as part of our explorations. We can save the algorithms that create these sounds. Then if we would like to add some "melancholy" aspects to a completely different sound we can reintroduce the original sounds into the population, and hope that the melancholy nature of the original melancholy sounds is generated by a similar process to the process that we would like to use to create melancholy variants on the new sounds.

It is interesting to speculate on whether this would work. How are such qualities contained within a sound? Perhaps different regions of the parameter space contain the same "quality" but the underlying process creating this quality is different—it is only at the perceptual level that we fuse these concepts. Indeed it may be possible to use such a tool as this to explore the connections between perception of qualities of sounds and their realization in terms of a particular algorithm, by letting a user explore a system for sounds with a particular subjective quality such as melancholy and seeing where these sounds lie in the parameter space of the algorithm.

Biologically we can think of this as being similar to "convergent evolution", where the same phenotypic quality emerges through different evolutionary paths (Cronin, 1993; Sigmund, 1993). There is also some common ground with the *subsymbolic* approach to artificial intelligence (Brooks, 1991a,b; Partridge, 1991).

One alternative pathway to creating such effects might be to learn (e.g. statistically, by carrying out some kind of mathematical analysis or by using neural networks) the qualitative features of the route taken through evolutionary space to get to a particular sound. This information could then be reused to guide explorations in new regions of sound-space, if the initial exploratory strategy had proven particularly effective. This fits into a broad program of using the information gained as a genetic algorithm progresses to inform future runs of the algorithm.

A more immediately practical use for this type of algorithm would be finding parameters for a particular synthesis algorithm which match a particular real-world sound sample. It is impractical to store all of the possible real-instrument sounds as samples within a sound generator, so there is a need to compress the information down by synthesizing some (as in "sample and synthesis" algorithms) or all of the sound. In programming a synthesizer to play sounds which sound like an acoustic instrument we take a synthesis algorithm (such as the Karplus-Strong algorithm for generating plucked and drum-like sounds (Karplus and Strong, 1983; Jaffe and Smith, 1983)) and find parameter settings which match the desired acoustic effect as closely as possible. This is clearly time-consuming to carry out manually. It should be possible to simplify this process by using a (non-interactive) genetic algorithm which rates sounds according to their similarity to a given sound-sample, using measures such as the amplitude profile of the sound and the spectral characteristics as fitness measures.

Other possibilities involve using this kind of system as a compositional tool. The idea of sounds transforming into one another is a well-used compositional technique in electroacoustic music. Some ideas about how to achieve these transformations have been described in Wishart (1994), and the technique has been used by composers including Wishart, Kaija Saariaho and Jonathan Harvey. A particularly elegant example of this occurs in Harvey's *Mortuos Plango, Vivos Voco* where sounds such as voice and oboe, or bell and voice, are merged into one another by spectrally decomposing the sounds and then interpolating between the sounds by changing the amount of the spectral components of each sound used in creating the new sound.

A similar idea may be achieved using the ideas above. We can seed an initial population with two very different sounds produceable by a given algorithm, and then find intermediate sounds for use in a composition by the evolutionary process. Alternatively the composer could start with a very focused sound world, and work outwards using the sounds generated via evolutionary exploration towards sonic diversity as a sonic event develops, or exploit the reverse process of beginning with a complex sound space and working inwards towards a single kind of sound.

We have also begun to experiment with the different synthesis algorithms found in the *CSound* system. *CSound* offers a wide variety of synthesis algorithms, and this evolutionary exploration methodology offers a universal interface to allow the composer to explore the capabilities of any of the algorithms found in the system.

In order to achieve richer harmonic structures we can layer the results of several synthesized sounds. This is used, for example, to create vocal-type effects using granular-style synthesis such as FOF (Vercoe, 1992). In order to achieve this effect we have been experimenting with differing-length encodings (akin to "messy genetic algorithms" (Goldberg et al., 1989, 1990)), where an additional genetic operator of duplication is included in the algorithm. A population is created consisting of single sounds, and from time to time the genome adds another length onto the end, representing an additional sound which is layered on the first sound. We are also experimenting with sound *processing* where we take a sound (either a sound co-evolved with the the processing by the above interface or a sampled sound brought into the system) and apply an amplitude envelope or spectral transformation. Again these sound processing algorithms are frequently complex multi-parameter systems and thus suited to the evolutionary interface described above.

These ideas lead onto a unifying idea of using a *genetic programming* methodology for sound synthesis (Koza, 1992). In genetic programming a whole sequence of procedures are encoded into a string by a more complex version of the procedure described above, and manipulated in a similar way. This allows the system to simultaneously evolve the parameters of the individual program components and the number and type of components used in the algorithm. We are looking at how we can use this genetic programming approach to evolve complex instruments, which could incorporate modules which carry out envelope generation, randomness generators, sound synthesis algorithms and sound processing algorithms.

6 Coda

In this paper we have described a new style of interface for complex sound synthesis algorithms. Another computational technique, genetic algorithms, has been shown to be applicable to sound synthesis, much as cellular automata (Miranda, 1995b) and fractals (Waschka II and Kurepa, 1989) have been applied in the past. It is exciting to witness this fecund crossover between developments in computing and mathematics, and their application to the provision of practical tools for creative artists.

In particular it is interesting to use these computational techniques to provide interfaces to creative systems in a way that provides access at the level of a usable artistic tool. This has been emphasized in the recent article Miranda (1995a) :

> "Modern computer technology enables the production of a virtually limitless variety of sounds by providing substantial access tot he parameter settings of synthesis algorithms. However, the production of sounds by means of a synthesis algorithm is accomplished in a very old-fashioned way: by inputting streams of numerical values for each single desired sound. Furthermore, these numerical values are usually worked out manually. The imagination of the composer in this case easily becomes vulnerable to time consuming, non-musical tasks.
>
> We believe that the power of the computer could also (a) provide the composer with better ways of expressing his or her requests to a synthesis algorithm and (b) provide appropriate aid for the exploration of sonic ideas."

In this paper we have described the beginnings of one such system, liberating the composer from technical constraints, and instead allowing structured yet exploratory access to the complexity of modern synthesis algorithms.

Notes

Copies of the programs described in this paper are available from the author on request.

References

Richard K. Belew and Lashon B. Booker, editors. *Proceedings of the Fourth International Conference on Genetic Algorithms*. Morgan Kaufmann, 1991.

John A. Biles. GenJam: A genetic algorithm for generating jazz solos. In *Proceedings of the 1994 International Computer Music Conference*, 1994.

John A. Biles. GenJam Populi: Training an IGA via audience-mediated performance. In *Proccedings of the 1995 International Computer Music Conference*, 1995.

John A. Biles. Interactive GenJam: Integrating real-time performance with a genetic algorithm. In *Proceedings of the 1998 International Computer Music Conference*, 1998.

John A. Biles, Peter G. Anderson, and Laura W. Loggi. Neural network fitness functions for an IGA. In *Proceedings of the International ICSC Symposium on Intelligent Industrial Automation (ISA'96) and Soft Computing (SOCO'96), March 26–28, Reading, UK*, pages B39–B44. ICSC Academic Press, 1996.

Rodney A. Brooks. Intelligence without reason. Technical Report (A.I. Memo No. 1293), Massachusetts Institute of Technology Artifical Intelligence Laboratory, April 1991a.

Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991b.

Craig Caldwell and Victor S. Johnston. Tracking a criminal suspect through "face-space" with a genetic algorithm. In Belew and Booker (1991).

J.M. Clarke. An FOF synthesis tutorial. Appendix 4 of the CSound manual, 1992.

Helena Cronin. *The Ant and the Peacock*. Cambridge University Press, 1993.

Richard Dawkins. The evolution of evolvability. In Langton (1989), pages 201–220.

Richard Dawkins. *The Blind Watchmaker*. Penguin, 1990.

David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

D.E. Goldberg, K. Deb, and B. Korb. Messy genetic algorithms revisited : Studies in mixed size and scale. *Complex Systems*, 4:414–444, 1990.

D.E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms : Motivation, analysis, and first results. *Complex Systems*, 3:493–530, 1989.

Andrew Horner and David E. Goldberg. Genetic algorithms and computer-aided music composition. In Belew and Booker (1991).

Damon Horowitz. Generating rhythms with genetic algorithms. In *Proceedings of the 1994 International Computer Music Conference*, 1994.

David Jaffe and Julius Smith. Extensions of the Karplus-Strong plucked-string algorithm. *Computer Music Journal*, 7(2), 1983.

Kevin Karplus and Alex Strong. Digital synthesis of plucked-string and drum timbres. *Computer Music Journal*, 7(2), 1983.

John R. Koza. *Genetic Programming : one the programming of computers by means of natural selection*. Series in Complex Adaptive Systems. MIT Press, 1992.

Christopher G. Langton, editor. *Artificial Life*. Addison-Wesley, 1989.

Eduardo Reck Miranda. An artificial intelligence approach to sound design. *Computer Music Journal*, 19 (2):59–75, 1995a.

Eduardo Reck Miranda. Granular synthesis of sounds by means of a cellular automaton. *Leonardo*, 28(4):297–300, 1995b.

Melanie Mitchell. *An Introduction to Genetic Algorithms*. Series in Complex Adaptive Systems. Bradford Books/MIT Press, 1996.

Gary Lee Nelson. Sonomorphs: An application of genetic algorithms to the growth and development of musical organisms. In *Proceedings of the Fourth Biennial Art and Technology Symposium*, pages 155–169. Connecticut College, March 1993.

Gary Lee Nelson. Further adventures of the Sonomorphs. Conservatory of Music Report, Oberlin College, 1995.

Derek Partridge. *A New Guide to Artificial Intelligence*. Ablex, 1991.

Jenny Preece, Yvonne Rogers, Helen Sharp, David Benyon, Simon Holland, and Tom Carey. *Human–Computer Interaction*. Addison-Wesley, 1994.

Jeffery B. Putnam. Genetic programming of music. Report, New Mexico Institute of Mining and Technology, 1994.

Gregory J.E. Rawlins, editor. *Foundations of Genetic Algorithms*. Morgan Kauffmann, 1991.

Curtis Roads. Granular synthesis of sounds. *Computer Music Journal*, 2(2):61–68, 1978.

J.D. Schaffer, editor. *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kauffmann, 1989.

Karl Sigmund. *Games of Life*. Oxford University Press, 1993.

Karl Sims. Artifical evolution for computer graphics. *Computer Graphics*, 25(4):319–328, 1991.

Gilbert Syswerda. Uniform crossover in genetic algorithms. In Schaffer (1989).

Gilbert Syswerda. A study of reproduction in generational and steady-state genetic algorithms. In Rawlins (1991).

T. Takala, J. Hahn, L. Gritz, J. Geigel, and J.W. Lee. Using physically-based models and genetic algorithms for functional composition of sound signals, synchronized to animated motion. In *Proceedings of the 1993 International Computer Music Conference*, 1993.

Stephen Todd and William Latham. *Evolutionary Art and Computers*. Springer, 1992.

Barry Vercoe. CSound manual. Manual distributed freely with CSound software, 1992.

Rodney Waschka II and Alexandra Kurepa. Using fractals in timbre construction: an exploratory study. In *Proceedings of the 1989 International Computer Music Conference*, 1989.

Trevor Wishart. *Audible Design*. Orpheus the Pantomime, 1994.

Trevor Wishart. *On Sonic Art*. Harwood Academic Publishers, 1996. second edition, revised by Simon Emmerson.