White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of York.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/113747/

Version: Accepted Version

Book Section:
Hickinbotham, Simon John and Stepney, Susan (2016) Augmenting Live Coding with Evolved Patterns. In: International Conference on Evolutionary and Biologically Inspired Music and Art; EvoMusArt 2016. Lecture Notes in Computer Science (LNCS). Springer, 31–46.

https://doi.org/10.1007/978-3-319-31008-4_3

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

UNIVERSITY OF LEEDS
University of Sheffield
UNIVERSITY of York

Augmenting Live Coding with Evolved Patterns

Simon Hickinbotham and Susan Stepney

YCCSA, University of York, UK
sjh518@york.ac.uk

Abstract. We present a new system for integrating evolvutionary processes with live coding. The system is built upon an existing platform called Extramuros, which facilitates network-based collaboration on live coding performances. Our evolutionary approach uses the Tidal live coding language within this platform. The system uses a grammar to parse code patterns and create random mutations that conform to the grammar, thus guaranteeing that the resulting pattern has the correct syntax. With these mutations available, we provide a facility to integrate them during a live performance. To achieve this, we added controls to the Extramuros web client that allows coders to select patterns for submission to the Tidal interpreter. The fitness of the pattern is updated implicitly by the way the coder uses the patterns. In this way, appropriate patterns are continuously generated and selected for throughout a performance. We present examples of performances, and discuss the utility of this approach in live coding music.

1 Introduction

We explore the use of evolutionary techniques to generate new musical constructs during live coding performances. Live coding is the use of domain-specific languages (DSLs) to improvise new musical pieces in a live concert setting. We present a new evolutionary system which augments this process by generating and maintaining a population of coded musical patterns that are interactively expressed as audio during the performance. The system allows the actions of the human coders to be fed back to the population as adjustments to the fitness of patterns, resulting in a novel musical interactive genetic algorithm (MIGA, [8]). The system is sufficiently flexible to allow the generation of pieces using evolved patterns alone, or any mix of evolved patterns and manually configured patterns.

Using artificial evolution to generate music has a long history. The core approach is usually to generate populations of complete musical pieces and to assign fitness values to each individual piece via human perception [8]. This process requires that each generated piece is heard at least once, and so causes a significant bottleneck in the evolutionary process. In addition, there is the issue of *listener fatigue*, in which the human assignment of fitness values varies as the listener wearies of listening to large numbers of generated pieces. There are ways around this problem, usually by either only presenting a subset of the generated pieces to the listener [9], or making the pieces very short [2]. Both of these

solutions have obvious drawbacks both in terms of the evaluation of each piece and the eventual result. In addition, the *positive* side of changes in the human evaluation of a musical pattern throughout a performance has received relatively little attention in previous work. Recognising when a repeating pattern should be changed is an important part of the composition process.

Our approach differs from the above methodologies as follows. Firstly, we are using the paradigm of *live coding* [3] as our musical framework. Here, rather than evolve entire pieces of music *ab initio*, the evolutionary process is interactive, and takes place while the piece is being performed. This allows for a more natural interaction between the evolutionary mechanism and the composer/performer. The recent development of live coding systems affords a new opportunity for researchers in evolving systems because new musical patterns must be generated many times during a performance. We give the artist (here referred to as the *live coder* or just *coder*) a new facility to augment their hand-designed patterns with ones that are generated by the evolutionary algorithm, and to blend the two together as they see fit. The evolutionary system *augments* the process of improvisation by generating novel patterns which the live coder can integrate and respond to. The population of evolved patterns changes gradually with the piece, and provides a reference point and storage for a changing bank of aesthetically pleasing patterns.

The recent development of the Extramuros [4] system has made the implementation of this idea more generic, and easier to achieve. Extramuros is a system that allows browser-based collaboration of a group of performers on live-coded pieces of music and graphics across a network. The availability of this system makes it easier to use automation to suggest patterns for use and further manipulation by the coder. An evolutionary process is an ideal candidate for the automatic generation of new coding patterns within this context.

Tidal [3] is the live coding language that we use in this work. Other live coding languages exist, such as Sonic Pi [1] and Chuck [10]. We selected Tidal as our live coding language because it is relatively well documented and straightforward to install in an Extramuros framework. The live coding approach allows more immediate interaction between human and automaton. Live coding languages are more like conventional computer code than other musical notation systems but with more emphasis on brevity and ease of editing than is common in programming languages, sometimes at the expense of clarity. Live coding requires a text-based grammar to encode the musical sounds, designed to be constantly manipulated rather than simply written once and interpreted many times. This grammar is amenable to a genetic programming (GP) approach to evolving systems.

The experimental work we present here tests this idea as a proof of concept. We have augmented the Extramuros system with an interactive evolutionary algorithm, which maintains a population of patterns on the client, and uses a separate server to carry out the process of parsing and mutating individuals in the population.

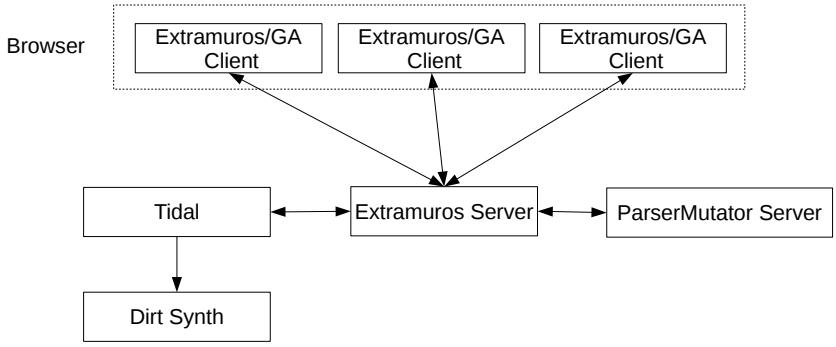

Fig. 1. Overview of the system

Evolutionary systems require a mechanism for mutation. A naive mutation model would substitute a single ASCII character in the pattern for another, but this would rarely yield a pattern that could be parsed by the interpreter. Clearly, a more sophisticated mutation scheme is required which preserves the syntax of the coded pattern. One approach would be Genetic Programming (GP), in which a tree representation of a program is used to organise the options of mutation. However, GP still has limitations, because it must be possible to interchange any node on the tree representation.

2 Methodology

We first describe the workflow of a performance, as this will give a clear understanding of the complete system and how it is used in a live performance setting. Following from this, we present a more detailed description of the evolutionary aspects of the system that we have implemented to test the concept.

Our evolutionary system is built around the Extramuros platform, figure 1. In a standard Extramuros system, each performer controls their contribution to the performance by entering Tidal code into text-boxes in a web-browser-based client (see https://www.youtube.com/watch?v=zLR02FQDqOM). The client can be configured to have any number of input boxes, and each user writes their code in one or more of them. The Extramuros server uses ShareJS, a library for concurrently editing content over a network. This means that each performer can see the code that every other performer in the ensemble is creating in near-real time in every browser. Indeed, it is only by agreed convention between performers that they do not attempt to edit each other's code, although there are no barriers to doing so.

When a performer has completed the latest edit to their code, they submit it to the Tidal interpreter by pressing the 'eval' button next to the text box, or by using a keyboard shortcut. (There were versions of Tidal which did not need this 'submission' step, but it was found that this led to jarring audio as the new

pattern was being typed in.) A piece of edited code is called a *pattern*. The Tidal interpreter parses the pattern and sends the resulting code to a synthesiser, usually the Dirt synthesiser that Tidal was developed to control. Other compatible synthesisers can also be used, and recent developments in Tidal also allow it to send MIDI data.

In order to allow evolution of patterns, we developed a client-side genetic algorithm and an additional server, shown in figure 1 as the ParserMutator server, which checks the syntax of each pattern as it is added the population, and rejects patterns which do not conform to the grammar. When requested, the ParserMutator server also generates mutated patterns which are syntactically correct. A population of patterns is held in the Document Object Model (DOM) of each client page. Each user can maintain their own population of patterns in this way without interfering with other coders in the performance, although a coder can push code that appears in another coder's text box to their own population if they want to. The core idea is that aesthetically pleasing patterns are 'pushed' to the GA population, and mutants of these patterns are 'pulled' from it later on. The fitness of a pattern is increased or decreased according to the way it is used in the performance. Since the system is designed to operate in a live performance, it is important to let the performer(s) decide when to do this. To facilitate this flexibility, we added five new buttons to each text-entry box, giving a total of six operations:

1. **eval**: Send the pattern to the Tidal interpreter
2. **push**: Send the pattern to the population (via the ParserMutator)
3. **pull**: Pull a pattern from the population
4. **pullmut**: Request a mutation of the current pattern from the ParserMutator
5. **up**: Increase the fitness of the current pattern (if it exists in the population)
6. **dn**: Decrease the fitness of the current pattern; remove it from the population if fitness < 0

We describe how these operations are used to change the fitness of a pattern below (section 2.3) In addition to the extra controls, we added a small notifications area to each box so that the client can communicate to the user the effect of the last action in each box. For the layout of a each text entry box, see the lower panel of figure 2.

The actions that these buttons encapsulate are connected into a performance workflow as shown in figure 3. The commands for these actions were written in JQuery and JavaScript. The workflow proceeds as follows. One (or more) Tidal patterns are typed into the text entry box(es) by the coder, and edited until a pleasing pattern is produced. When this happens, the patterns are 'pushed' to the GA population. The population is initially zero, and has a maximum of 20 individuals. This limit on population size is large enough to allow sufficient varibaility in the population, but small enough to provide some selection pressure within the timescale of a performance. When a pattern is initially pushed to the population, its fitness is set to 1. The system checks whether the pattern already exists, and if it does then its fitness is increased by 0.5. The 'up' and 'dn' buttons

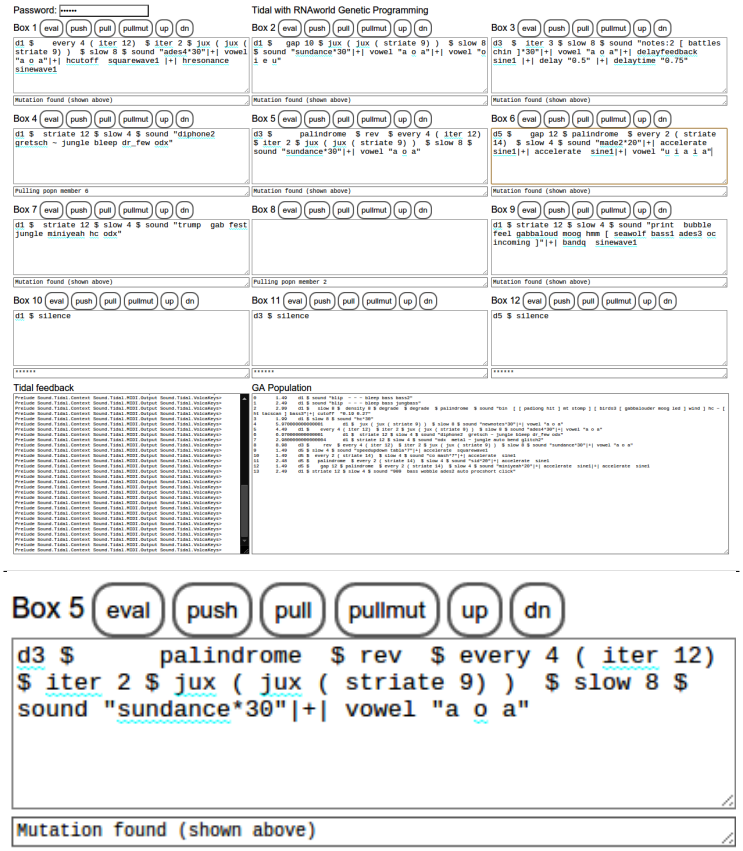

Fig. 2. The evolving Extramuros platform (top), and closeup view of pattern entry box 5(bottom) (colours inverted for clarity).

allow fitness to be respectively increased or decreased by 0.25 at any time. This allows fine-tuning of the fitness of the whole population.

Once there are some patterns in the population, it becomes possible to copy a pattern from the population into a text box using the 'pull' button. An indiviudal pattern is selected randomly from the population, with the chance of being selected weighted by the fitness of each individual. One copied to a text box, the pattern can be submitted to the Tidal interpreter, edited manually or mutated by pressing the 'pullmut' button.

When the maximum permitted number of patterns are in a population, new patterns that are submitted to the population take the place of the least fit existing pattern. If there is more than one pattern with the minimum fitness value, then one of these low-scoring patterns is selected for deletion at random.

Having described the overall function of the system, we now turn to the mechanism by which we maintain a population of Tidal patterns, and how we

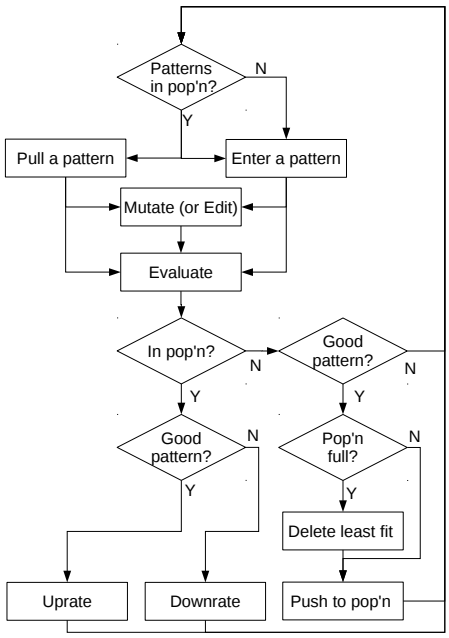

Fig. 3. Flowchart of the operation of the genetic algorithm. "Population" is abbreviated to "pop'n"

are able to mutate them. This is achieved by reference to a grammar, from which we construct a pattern parser and a mutation operator to build our genetic algorithm (GA). These are described below, after a brief description of the Tidal pattern language.

2.1 Tidal

In order to evolve music, an abstract representation of the sounds is needed upon which the evolutionary operators can act. Here, we use Tidal, the Haskell-based live coding pattern language. For a complete description of Tidal, see [3] and `tidal.lurk.org`. The language is designed to facilitate ease of composition in live performances, and allows the user to quickly generate complex manipulations of synthesised sound. To get a flavour of how patterns are coded in Tidal, we describe the components of the pattern from the bottom of figure 2:

```
d3 $ palindrome $ rev $ every 4 (iter 12) $ iter 2
$ jux (jux (striate 9)) $ slow 8
$ sound "sundance*30" |+| vowel "a o a"
```

The `d3` at the beginning specifies that sound will be sent to the 3rd of the 9 channels in the Dirt synthesiser. A series of *pattern transformer* operations

are applied. Each `$` sign applies the operator to everything that follows, as if the whole of the remainder of the pattern were in parentheses. These operators change the speed, order and sub-sampling of the pattern of samples that is specified by the `sound` operator, which is the only mandatory command. The `sound` command has a single argument: a pattern of samples enclosed within quote marks. In this example, the sample `sundance` is repeated 30 times. There are many different ways to specify when the sample will be played within a prespecified time period (default 1 cycle per second). Following from the `sound` command, a series of synthesiser parameters can be used, separated by the `|+|` operator. In this example, the `vowel` command uses a formant filter to mask the sample with a vowel-like sound. The vowels used are *also* specified by a pattern, in this case `"a o a"`. The pattern of samples is controlled by a pattern of vowel shapes, and each pattern is organised in time independently of the other. This combination of transformations can radically change the way these samples can sound to the human ear.

2.2 Grammar and Parser

There are strong arguments for running the evolutionary algorithm externally to the Tidal process. Firstly, it is good design practice to keep the code generation/manipulation functionality independent from the code execution – this is safer because it the former will not interfere with the scheduling of the latter. Secondly, it means we can run the grammar tools as if they were acting as another composer. Finally, it means that we have a separate representation of the grammar that we can manipulate in order to yield attractive patterns.

Constructing the grammar There are many ways to represent a language in a grammar. Note that our aim here was not to create a grammar that could parse all *possible* Tidal patterns, but rather to create a functional grammar that could parse the majority of patterns and allow us to implement some mutation functions to act upon that reduced grammar, mainly because we wanted to test the whole system before expending too much time refining the grammar. Accordingly, we created a grammar that was sufficiently representative of patterns that live coding use. This is not an exhaustive grammar, but the majority of the structures in the test corpus patterns could be parsed. It was more important that syntactically illegal patterns were excluded from the evolutionary system.

We used two sources of pattern data to construct the grammar: the Tidal reference pages at `tidal.lurk.org`; and a set of example patterns available from `yaxu.org/tmp/patternlib/`. The former was used to ensure that all the documented commands in Tidal were represented in the grammar, and that the syntax from the examples was correctly implemented. The latter was used to test the ability of the grammar to parse the more sophisticated constructs that can be created in Tidal. We call this data set the *test corpus*.

We used Antlr [6] to generate a java library from the grammar we developed. A subset of the parsing rules are shown in figure 4 in the Antlr grammar

```
trans_spec
: trans_0arg
| LBRK trans_spec  RBRK (POINT LBRK trans_spec RBRK)*
| slow_pattern
| STUT INTEGER zero2one (zero2one | LBRK MINUS zero2one RBRK)
| SUPERIMP LBRK trans_spec RBRK
| cont_frag
| (DEG_BY|TRUNC) zero2one
| (number)? (BEATR|BEATL)
| int_arg_trans intint
| EVERY INTEGER ((LBRK trans_spec RBRK)| trans_spec)
| FOLDEVERY LSQB INTEGER (COMMA INTEGER)+ RSQB LBRK trans_spec RBRK
| (SOMETIMESBY_ALIASES | SOMETIMESBY zero2one) trans_spec
| WHENMOD INTEGER INTEGER LBRK trans_spec RBRK
| WITHIN LBRK zero2one COMMA zero2one RBRK LBRK (trans_spec|(KNIT cont_frag )) RBRK
| JUX LBRK trans_spec (POINT trans_spec)* RBRK
| ZOOM LBRK zero2one COMMA zero2one RBRK
| STRIATE1 INTEGER LBRK (ONE|INTEGER) DIVID (ONE|INTEGER) RBRK
| SMASH intint LSQB number (COMMA number)* RSQB
| slowspread_pattern ;
```

Fig. 4. Example of rules used to construct the Tidal parsing grammar

format. This library was then used to build the parse functions needed to generate patterns automatically, and also to build the mutation operators for the evolutionary algorithm.

One of the advantages of working with live coding systems is that they are designed to be 'crash-proof': badly formed patterns do not break the system, they merely generate an error message as feedback to the composer. This is analogous to biological systems, where expressed enzymes cannot change the laws of chemistry (c.f. crash the system), but can be sufficiently harmful (c.f. generate an error message) to alert the organism that evolved them to respond appropriately.

2.3 Evolutionary algorithm

Our goal is to prove the concept of evolving DSL-based pattern music in a live setting. Accordingly, we developed a minimal genetic algorithm, capable of generating mutated patterns from a population of patterns on demand.

Our grammar-based genetic algorithm uses the parser described above to evolve the population. Since the parser can tokenise any legal Tidal pattern, it is possible to use the patterns themselves as the genotype of the evolutionary process, since mutation is also handled by the parser. In this respect, our approach differs from Grammatical Evolution [5], which uses a sequence of integers to generate a pattern from the parse tree. Our approach is more akin to Genetic Programming, but is able to handle different values of terminal nodes due to the use of specific mutation operations for particular node classes.

Default values for the parameters of the evolutionary algorithm are shown in figure 5. Genetic algorithms have five stages. The first is an *initialisation* stage, followed by an iteration through stages of *selection*, *crossover*, *mutation* and *evaluation*. Finally, a *termination* step is introduced to decide when to stop.

Variable	Value	Notes
Mrate	0.4	Mutation rate
maxpop	20	Maximum number of individuals in the population

Fig. 5. Default values for the parameters in the system

Each of these stages requires special consideration in a live coding setting. For ease of reference, we detail each of these stages below.

Initialisation Although there are many potential strategies for automatically initialising a population, we felt that it was important for the coder to select patterns to submit to the algorithm from the outset. There are two reasons for this. Firstly, it means that mutations will always have basis in the tastes of the coder. Secondly, it means that we did not have to implement a full pattern generator via the grammar (this will be the subject of future work).

Selection Our selection strategies work in two ways. Positive selection is based on the fitness of the individual. When the coder uses the 'pull' command to select a pattern from the population, the chance of a pattern being selected is proportional to fitness. When a new pattern is added to the population, one of the patterns with the minimum current fitness is automatically deleted.

Crossover Crossover is a controversial topic in genetic algorithms, and it was not necessary to implement it in our current algorithm. We don't do crossover automatically yet, but we propose that if we only swap over identical node types, then crossover can never be fatal. That is the advantage of a grammar-based mutation strategy. Coders have the opportunity to cut and paste fragments of mutated patterns should they see fit, but there is no pressure to do so.

Mutation To test our approach, we implemented three mutation mechanisms which operated on three different regions of a Tidal pattern: insertion of pattern and sample transformers (e.g. the phrase `$ every 4 (iter 12)` for the pattern in figure 2); substitution of sample patterns (e.g.`sundance` cold be changed for `jvbass` or `[bd cp jvbass]`; and insertion of synthesiser parameters (e.g. `|+| vowel "a e o i"`). (For more details of this syntax, see `tidal.lurk.org`). Mutations were positioned in the pattern by identifying the appropriate nodes in the parse tree that the mutations could be applied to. This guarantees that the mutations are always syntactically correct, and allows complicated nesting of sample patterns and transformations to emerge through evolution. Given the lack of crossover, it was important that the mutation opeartor could insert new branches into the parse tree. This is achieved in two ways: insertion of transformations and synthesiser patterns; and insertion of new groups of pattern events in a single event (for example, mutating the pattern `"sn sn bd sn"` to `"sn [hc ho hc] bd sn"`).

As in standard GAs, 'harmful' mutations are bound to emerge. The 'damage' that they will do in this application is evaluated by the coder - they are sounds which are in some way incompatible with the performance at the current time. Also, some mutations are neutral. For example, if a 'delayfeedback' command is created, it will have no effect unless the 'delay' parameter is set. Since this will do no harm (other than to make the pattern text more obscure), there is no real problem if these things emerge. The coder, observing such mutations, might consider to turn these features 'on' by adding the necessary text, feeding this back in to the evolving pattern population.

It is possible to produce many modifications of the same class, for example `|+| vowel "a e i" |+| vowel "a e o u i"`. The Tidal interpreter will superimpose these transformations on each other, placing them at appropriate positions through the time period. In this example, the first pattern will be spread out at a period of 1/3 of the cycle time and the second pattern will be spread out over 1/5 of the cycle time.

Mutation is a stochastic event. Since our genetic algorithm is interactive, and mutations are requested by the user, it is important that a change *usually* happens. However, it is also important that the mutation is not so great that there is no relationship between parent and offspring. When a coder presses 'pullmut', they are *requesting* a new mutation, so they don't want to get something back that they've seen already – there is much more emphasis on novelty. We experimented with a range of mutation rates, and further work here is needed, but we found that setting the probability of 0.4 per mutable node gave a good balance of stability and innovation. The number of mutable nodes varies depending upon the sample pattern, so the mutation rate will change between different patterns. Every pattern contains at least one `message` node, and at least one `sample` terminal node, so the probability $p_p$of mutation per pattern is $p_p \geq \sqrt{0.4}$.

Although we have shown how mutation happens when the 'pullmut' button is pressed, it is important to emphasise that the coder continues to edit the evolving patterns throughout the performance, and could be considered to be another sort of mutation or crossover operator. This is one of the strengths of the approach we have developed, as it allows a more flexible interaction between the coder and the algorithm.

Evaluation Evaluation is the assignment of fitness to patterns. The perceived fitness of a pattern is dependent on the current patterns being interpreted into audio at any one time. In this application, fitness is determined by the buttons that the coder presses to evaluate a pattern and interact with the current population of patterns. For any pattern in an input box that also exists in the genome, when a button is pressed the fitness will be changed by the following values:

- **eval**: +0.5
- **push**: +0.5 (if pattern already exists in the population)
- **pull**: 0
- **pullmut**: +0.49

– **up**: +0.25
– **dn**: -0.25

Termination The choice of when to end a performance is up to the ensemble of coders. It is possible that the ensemble will stop using the GA towards the end of a performance in order to craft a satisfying ending to the performance by hand.

3 Evalutation

Source code for our evolvable version of Extramuros is available at `github.com/anon/Extramuros`. Source code for the ParserMutator server, which runs on Tomcat 7, is avalable at `github.com/anon/ParserMutator`. For clarity, the concept we are testing is the use of evolved patterns during a live coding performance, which proceeds as follows:

1. the Extramuros system is initialised as normal
2. the ParserMutator web server is initialised
3. the coder(s) specifies the web client url in the browser
4. the coder(s) enters the password.
5. Begin live coding with evolution as in figure 3

The following sections detail the evaluations we have carried out on the system

3.1 Is the system generating legal Tidal patterns?

Yes. The mutations we designed were a subset of all possible instances of the grammar, and could always be parsed by the Tidal interpreter. Note though that not all hand-coded patterns can be parsed as legal by the ParserMutator, as it only recognises a subset of all possible Tidal patterns. For example, patterns using Haskell's applicative functor notation are not currently recognised by our parser, but they can still be submitted to the Tidal interpreter during a performance. We plan to extend our parser in the future so that (almost) all Tidal patterns can be recognised.

3.2 What is the relative level of complexity of the generated patterns?

Figure 6 (top) shows the parse tree for the following evolved pattern:

```
d3 $ palindrome $ every 5 (degrade) $ palindrome $ striate 8 $
sound "feelfx speechless seawolf [tech tink [birds3 gabbalouder tabla2
incoming]] f" |+| delaytime "0.39 0.64 0.33" |+| speed "10.7 0.62"
|+| end sinewave1
```

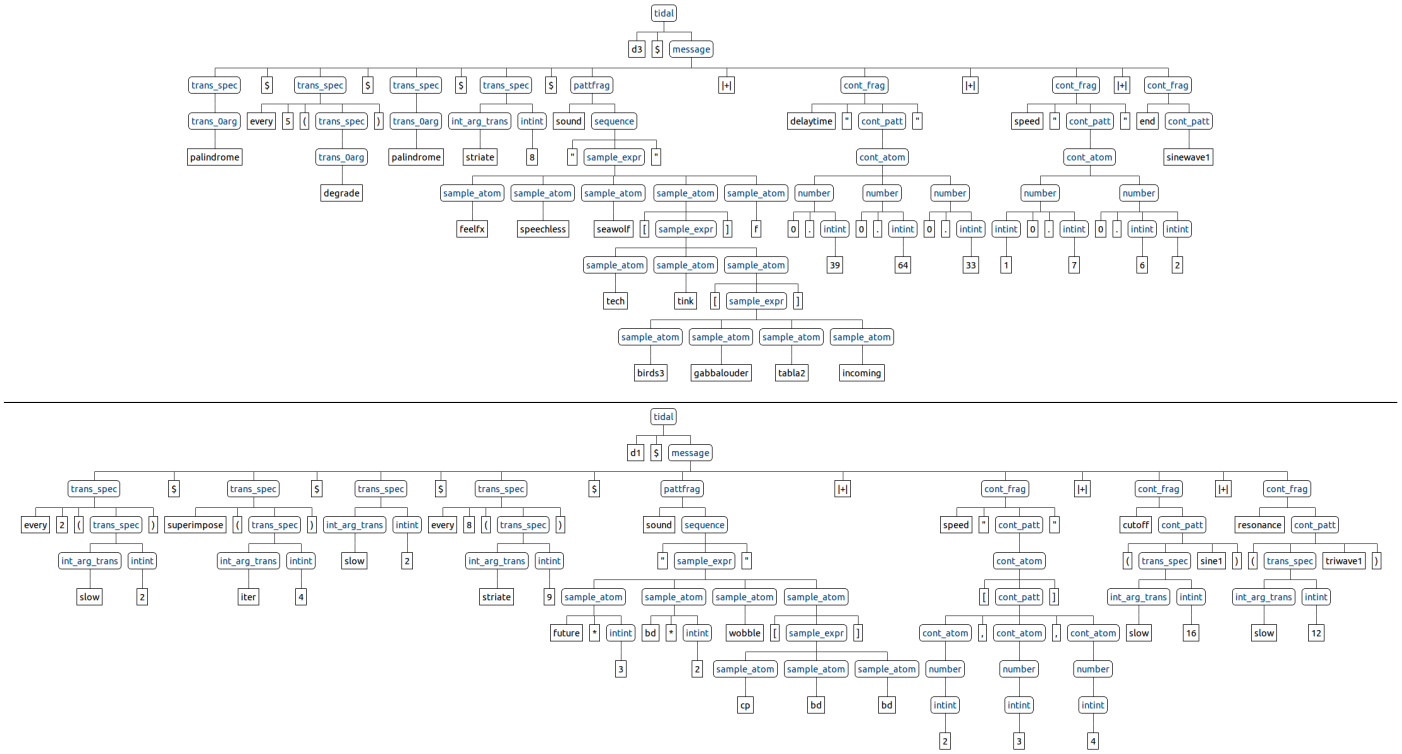

Fig. 6. Parse tree of an evolved pattern (top) and a hand-coded pattern from the Yaxu corpus (bottom).

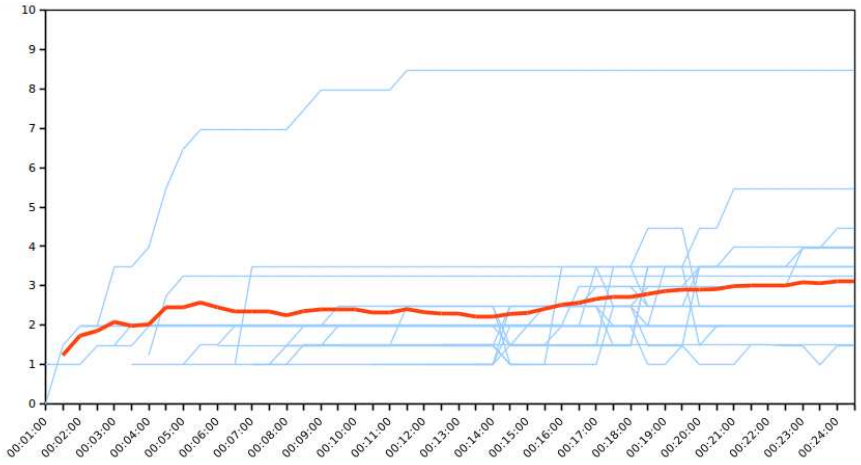

Fig. 7. Change in fitness for evolved patterns. x-axis is the performance time (hh:mm:ss), y-axis is fitness. The fitness of the individual patterns is shown in blue and mean fitness is in red.

Figure 6 (bottom) shows the parse treee for the following hand-coded pattern:

```
d1 $ every 2 (slow 2) $ superimpose (iter 4) $ slow 2 $
every 8 (striate 9) $ sound "future*3 bd*2 wobble [cp bd bd]"
|+| speed "[2 , 3, 4]" |+| cutoff (slow 16 sine1)
|+| resonance (slow 12 triwave1)
```

These diagrams of the parsing process show that the structures are intutively similar. Both are representative of the level of complexity that live-coded patterns tend to reach, particularly if they are the only patterns that the synthesiser is processing at the time (i.e. no other channels are being processed simultaneously). Patterns are usually in three sections: *transition specifications*, *pattern specifications*, and *continuous modifiers*. Our mutation mechanisms create parse nodes in each of these areas, and so yield similar parse trees to hand-coded patterns. (A more rigorous comparison will be the subject of future work).

3.3 How does the generated audio compare with Tidal?

Several videos of the system are available on the (anonymous) YouTube channel www.youtube.com/channel/UCu0_2dIhFvfKV-c975snmXQ. In each of these, the 'pullmut' function was the main source of innovation in the patterns, with the occasional hand-coding or editing of evolved patterns by the user.

These performances compare favourably with hand-coded Extramuros-based performances such as www.youtube.com/watch?v=zLR02FQDqOM, especially if it is considered that the users of the evolved system have little or no previous experience of live coding. We currently have no means of quantitatively comparing live coding performances, which will be the subject of future work.

3.4 How does the fitness of the population change?

A plot of the change fitness values of a population of evolved patterns is shown in figure 7. Early on in the run, there are few patterns in the population because the coder has not added many at this point. Average fitness increases until the 5^{th} minute and then drops off as more new patterns with low initial fitness values are added to the population. An individual pattern rapidly increases in fitness as it is used to create new patterns via mutation. The population reaches the size limit of 20 at around the 15^{th} minute. From this point when a new individual is added, unfit genomes are removed. Fitness rises from this point onward.

4 Conclusion

We have demonstrated a new system for evolving live coded music in collaboration with human coders. The novelty of the system is that it exploits the networked sharing of live code to allow interaction with an evolutionary music generation module. Having linked a live coding framework to an evolutionary algorithm, we have a convenient system for experimentation with a range of evolutionary algorithms. Several avenues for further research naturally present themselves.

At the moment, a population of patterns exists on each browser client. Whist this is convenient, and allows different coders to maintain their own population of patterns, it might be better to maintain a larger population on the server side, or to allow "horizontal gene transfer" between populations of patterns.

The method of assigning fitness values based on the interaction of the coder with the mutated patterns deserves further attention. We plan to evaluate a range of different fitness scoring strategies within this approach. A key problem that needs to be addressed is the relatively low fitness of newly-generated patterns compared with patterns that exist in the population. One could envisage 'inheriting' a proportion of fitness from the parent pattern to counteract the relatively high fitness of patterns that have been established in the population for some time. This is a challenging problem.

The current ParserMutator server carries out a small set of mutation types on the parsed patterns, which needs to be extended further to investigate the utility of the system in a more general setting. Firstly, we need to have mutation operators for a wider range of node types in the parse tree. We could also experiment with weighting the distribution of mutations at different nodes in the grammar. This would allow us to make certain changes occur more frequently and push the evolution in particular directions for particular applications.

4.1 On the experience of live coding with evolved patterns

Our experience as users of the system is worth some comment. There are usually nine channels into the Dirt synthesiser, and during a live Extramuros performance, each coder usually sends patterns to a pre-agreed channel. Patterns that

are pushed to the GA population have the channel encoded in them, and there is no mutation operator that can change the channel. It is left to the coder to change the channel specification by manual editing. It might be better to formalise the way that the channels are organised. At the moment they are stored on the genome, but this may not be the best way to arrange things.

Working with the GA as a single user, one tends to spread out patterns across multiple windows - it is as if we are using the GA as one or more extra performers, and allowing it to populate several text boxes with evolved patterns (see video, linked in section 3.3). Although this is different to the convention used by coders in Extramuros, the two approaches can be made to be compatible by prior consent.

Audiences at live coding events are very open-minded, and used to 'glitches' appearing as the performance continues. As long as unsympathetic patterns are removed, than that is accepted as part of the experience. These are the ideal conditions for experiments with music evolution.

The only technical problem we encountered with the evolved patterns was due to differences in sample length in the library of samples available to the Dirt synthesiser. Some samples were short such as the bass drum `"bd"`, whereas other samples were longer, for example `"bev"`, which samples a few seconds of singing. The problem arises when the pattern repeats the long sample in such a way that overloads the memory available to Dirt, resulting in buffer overruns. This reveals one issue where the live coder has an advantage over the genetic process - familiarity with the sample library prevents the coder making these errors. With experience, it is possible for the coder to spot where an evolved pattern is beginning to flood the Dirt buffer, and edit the pattern before the buffer is full. Related to this, our system is useful for novice Tidal coders unused to the nuances and range of functions available in the Tidal language, as the system generates examples of how the various features of Tidal can be used.

4.2 On the need for a quantitative evaluation

Our goal with this contribution was to evolve Tidal patterns for use in a live coding performance. We have offered a qualitiative evaluation of this work, but recognise that a quantitative evalution is desirable, particularly as we implement the improvements discussed above. There are two routes to obtaining a qualitative evaluation, which we sketch out here. The first approach is to make a comparison of the statistics of the patterns. For example we could compare the graphs of the evolved patterns with hand-coded patterns quantiatively using graph statistics, as we did subjectively in section 3.2. The second approach is more challenging, which is to somehow evaulate the artistic qualities of the generated piece. One could apply audio statistics to this problem, but this does not truly capture artistic merit unless it is done by reference to high-quality pieces of music in a similar style. As Fernandez and Vico [8] point out, this is a challenging problem for all automated music generation. It is worth remembering that a quantitative evaluation also offers a route to unsupervised generation of live coding patterns, since it can be used as a fitness function.

4.3 Closing remarks

By making the connection between the musical process and evolution, we have necessarily made a more direct connection between the parser, the patterns themselves, and the audio that is produced. The audio is in effect part of the phenotype of the pattern. This is very Pattee-like [7] – the actual specification is spread across all representations - function, pattern and parser. It is also worth stating the analogy with biological systems. Biology exists in time. It is not static, and must respond to a dynamic, changing environment. One of the attractions of working in musical systems is that time must be considered implicitly, and the dynamics of the performance must be accommodated.

5 Acknowledgements

This work was funded by the EU FP7 project EvoEvo, grant number 610427.

References

1. Aaron, S., Blackwell, A.F.: From Sonic Pi to overtone: creative musical experiences with domain-specific and functional languages. In: Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design. pp. 35–46. ACM (2013)
2. MacCallum, R.M., Mauch, M., Burt, A., Leroi, A.M.: Evolution of music by public choice. Proceedings of the National Academy of Sciences 109(30), 12081–12086 (2012)
3. McLean, A.: Making programming languages to dance to: live coding with Tidal. In: Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design. pp. 63–70. ACM (2014)
4. Ogborn, D., Tsabary, E., Jarvis, I., Cardenas, A., McLean, A.: extramuros: making music in a browser-based, language-neutral collaborative live coding environment. In: McLean, A., Magnusson, T., Ng, K., Knotts, S., Armitage, J. (eds.) Proceedings of the First International Conference on Live Coding. p. 300. ICSRiM, University of Leeds (2015)
5. O'Neill, M., Ryan, C.: Grammatical evolution: evolutionary automatic programming in an arbitrary language, vol. 4. Springer Science & Business Media (2012)
6. Parr, T.: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, Dallas Texas (2013)
7. Pattee, H.H.: Cell psychology: An evolutionary approach to the symbol-matter problem. In: LAWS, LANGUAGE and LIFE, pp. 165–179. Springer (2012)
8. Rodriguez, J.D.F., Vico, F.J.: AI methods in algorithmic composition: A comprehensive survey. Journal of Artificial Intelligence Research 48, 513–582 (2013)
9. Unehara, M., Onisawa, T.: Composition of music using human evaluation. In: Fuzzy Systems, 2001. The 10th IEEE International Conference on. vol. 3, pp. 1203–1206. IEEE (2001)
10. Wang, G., Fiebrink, R., Cook, P.R.: Combining analysis and synthesis in the chuck programming language. In: Proceedings of the International Computer Music Conference. pp. 35–42 (2007)