An Approach to Live Algorithmic Composition using Conductive

Renick BELL
Tama Art University
2-1723 Yarimizu
Hachioji, Tokyo 192-0394
Japan
renick@gmail.com

Abstract

Algorithmic composition can be done as a live performance using live coding tools. An example approach to such performances is described. Using the Conductive library for the Haskell programming language in conjunction with some external tools, samples are triggered according to interonset interval patterns generated at a variety of densities. Automatic movement through those density levels is accomplished through a specialized data structure, which is also used to time-vary other parameter values. The performer manages the state of the above items, and finally audio is output through effects.

Keywords

algorithmic composition, live coding, Haskell

1 introduction

This paper describes an approach to performing extended sets of live algorithmic composition. It was the author's goal to perform generative music live with the computer was an active partner of the user. Rather than prepare data and algorithms completely in advance, it was desired that those algorithms or at least their parameters could be adjusted as the music is being performed.

To carry out such performances, a live coding interface was chosen for its flexibility, its light-weight character, its compatibility with a tiling window manager, and its ability to employ a user interface that was already very familiar: the vim text editor and the command line. Some further discussion on the reasons for choosing this approach have been described in a previous paper, which led to the development of a Haskell library called Conductive to provide some basic components for doing such performances in conjunction with some external tools (Bell 2011). Those tools alone had been determined insufficient for extended performances, however. In order to fully realize such performances, additional modules were developed.

Before explaining this system of tools, the paper first briefly explains live coding. Tools used in conjunction with this system are listed. A brief review of Conductive core concepts is followed by a description of additional modules developed for handling event density, time-varying values, a sampling synthesizer, and mutable data. The paper concludes with a discussion of the results of this approach and some proposed future research directions.

2 live coding

According to TOPLAP, an organization for the promotion of live coding, live coding practice begins in the 80s. The 90s appear dry, while the 21st century starts with the band Slub in what TOPLAP calls the "projection era" and continues to the present in which an increasing variety of live coding systems are available and used in performances (McLean and Others 2010). Now live coding conferences take place (unknown 2013) and it is scheduled to be the theme of an upcoming issue of Computer Music Journal (McLean 2012).

Live coding enables a more abstract manipulation of a representation of music than physical gestures used for playing instruments. It is also thought to be more convenient in many regards than windows-icon-mouse-pointer (WIMP) software (Bell 2011).

From another perspective, it takes the potential of algorithmic composition and turns it into a live performance rather than a write/compile/run loop from traditional software development or electronic music composition. Seen this way, it can be thought of as an extension of algorithmic composition practices that could extend back as far as Ptolemy's music theory (Maurer 1999), and certainly as far back as music dice games such as Mozart's Dice Music (Hedges 1978). More modern examples of algorithmic composition practice include the twelve-tone music of Schoenberg (Schoenberg

1999), work by Caplin and Prinz followed by Hiller and Issacson (Ariza 2011), the aleatoric music of Cage and Stockhausen (Kostelanetz 2002)(Paul 1997), Xenakis's stochastic music (Xenakis 2001), and the generative sequences made in Max on Autechre's Confield (Tingen 2004).

One of the drawbacks of live coding is the hard mental operations that it requires. For a more complete discussion of the usability issues involved in live coding, see Blackwell and Collins (Blackwell and Collins 2005). Another factor is the potentially slow text manipulation that live coding requires (Sorensen and Brown 2007).

The system described below is intended to address some of these difficulties.

3 system and related tools

This section first explains what tools developed by other authors are used when performing. It then reviews some core concepts of Conductive, and finally it describes the newly-developed aspects of Conductive.

3.1 tools developed by other authors

In order to use this system, there are some prerequisites.

The first of those is a Haskell programming environment. The Glasgow Haskell Compiler, which contains an interpreter (GHCi) that allows the interactive evaluation of source code (SL Peyton Jones et al. 1993), is used by the performer to call functions from the Conductive library. The process of writing source code and sending it to GHCi is made more usable with vim (a text editor) (Moolenaar 2008), tmux (a terminal multiplexer)(Marriott and others 2013), and a vim plugin called tslime that allows text to be sent from the editor to the interpreter through tmux (Coutinho 2010).

As Conductive does not directly handle sound synthesis, a method for synthesizing sound is necessary. This paper describes the use of the scsynth component of the SuperCollider package (McCartney 2010). At present, synthesis events are programmed in Haskell and employ Rohan Drape's hsc3 Haskell library for communicating with scsynth (Drape 2009). A sampler (described below) uses samples that have been generally recorded and edited using Ardour (Davis 2006), and they have largely originated from hardware synths. All of the samples are individual sounds, from single-shot percussion sounds to bass samples. Most are wav files under 300 K.

Finally, in order to achieve a solid sound closer to that of commercial releases or broadcasts, the output of scsynth is processed through Calf plugins hosted by the Calf stand-alone host (Foltman et al. 2007). An EQ is followed by a multiband compressor and then a limiter, whose output is directed to the soundcard. Output is also directed to JAAA for monitoring (Adriaensen 2004). Patchage is used for ease of routing (Robillard 2011). Recording of performances is done with either Ardour (in the case of audio) or gtk-recordmydesktop (in the case of video) (Varouhakis and Nordholts 2008).

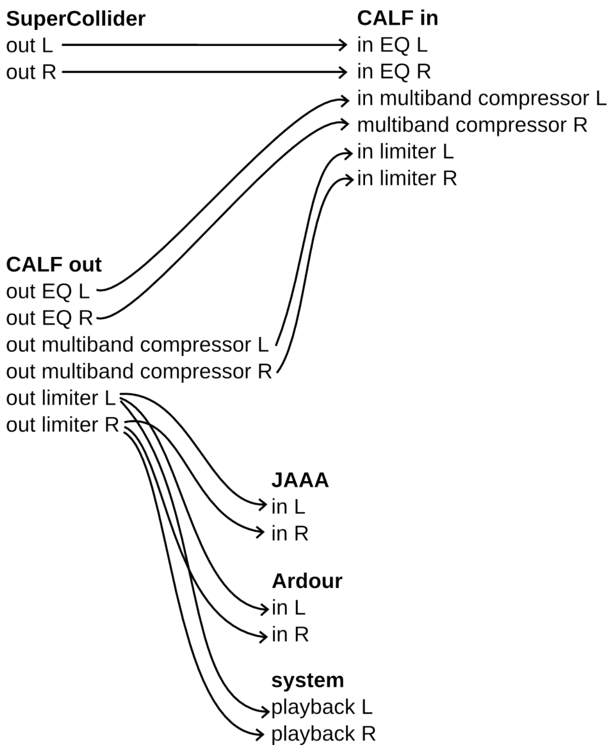

Figure 1: signal flow

3.2 summary of Conductive concepts

A system called Conductive is used, which is a set of modules for the Haskell programming language handling concurrent processes with a music-oriented interface.

Some basic concepts for using Conductive include the notion of Players, action functions, interonset interval (IOI) functions, and TempoClocks. These concepts are explained in more detail in a paper from 2011 (Bell 2011), but a short summary is included here.

Players are representations of concurrent processes that perform actions separated by periods of time called interonset intervals (IOIs). A Player runs its specified actions and then waits for an IOI determined by its specified IOI function. This loop is instantiated by employing the "play" function with a Player as an argument.

Actions functions define what is done by a Player. These actions could include triggering a synthesis event or modifying the general system state. The only limitation is their type signature, since Haskell is a statically-typed language. This means that the types of arguments to an action function are fixed, and they must return the unit type in the IO monad, or "IO ()". Currently, a sampler action is used predominantly.

IOI functions define how long to wait between actions. Any methods available to the programmer could be used to generate those times, from simply returning a value, such as one second, every time, to table lookup of values, to the calculation of values based on complex mathematical formulae.

One minor change from the 0.2 system of 2011 is that IOI functions now take additional arguments and return the beat of the next event rather than the IOI value directly. The play function uses that value in conjunction with a TempoClock to actually determine how long to wait before running the next action.

4 new modules for Conductive

This section explains the new modules for Conductive: density, TimespanMaps, and MutableMaps.

4.1 IOI values and density

Previously, IOI functions used hand-written IOI patterns or patterns which were determined mostly at random. A more sophisticated approach was sought that would require less manual intervention during a performance.

The sequence of IOI values determines the rhythm of a sequence of events. Rather than enter sequences by hand, they are generated algorithmically. The IOI patterns are looping ordered lists of IOI values in terms of beats, whole or fractional.

Pattern generation is based on a performer-selected core unit used to generate potential IOI values. Selection of a core unit, in conjunction with the length of the pattern, largely determines the metrical feel of the pattern. A list of scalars is determined by the performer, from which a function randomly selects a user-specified number of scalars.

The user specifies a number of subphrases to generate and the length of those phrases in terms of number of scalars to use, from which the final phrase will be constructed. Those subphrases are generated to the specified length by randomly choosing the specified number of scalars from the subset selected above and multiplying them by the core unit.

Finally, a user-specified number of subphrases are chosen at random from the resulting list by the algorithmic composition function. The user determines the length of the final phrase in terms of beats. If the length of the concatenated subphrases does not equal the specified length, the final IOI value is padded. If the length exceeds the specified length, the final IOI value is truncated.

An example of those steps follows. Items are determined by the user are followed with a "u":

- core unit (u): 0.25
- potential scalars (u): 1, 2, 3, 4, 5, 6, 7, 8
- number of scalars to be selected (u): 5
- selected scalars: 1, 2, 3, 4, 6
- potential IOIs: 0.25, 0.5, 0.75, 1.0, 1.5
- number of subphrases (u): 2
- subphrase length (u): 3
- selected subphrase scalars: 1, 3, 2; 4, 1, 6
- initial subphrases: 0.25, 0.75, 0.5; 2, 0.25, 2.5
- phrase length in terms of subphrases (u): 3
- initial randomly determined phrase: 0.25, 0.75, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 2.5
- total phrase length: 7.75
- phrase length in terms of beats (u): 8
- final phrase: 0.25, 0.75, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 2.75

Given a particular IOI pattern, a series of related patterns (both denser and less dense) is generated. It is built out to maximum and minimum density. This means making a list of IOI patterns ordered in terms of density. When reducing density, an item from the pattern is chosen at random and combined with a neighboring value to yield a similar pattern of reduced density. This process is repeated until the IOI pattern contains only a single item. When increasing density, an item is chosen at random and replaced with two items: an item of lesser value from the list of potential IOIs and the difference between the original IOI value and the lesser value. This is repeated until all of the items in the pattern are the smallest of the potential IOIs. By sandwiching the original IOI pattern between the less-dense and denser patterns, a table is generated.

Here is a continuation of the previous example:

- potential IOIs: 0.25, 0.5, 0.75, 1.0, 1.5
- input phrase: 0.25, 0.75, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 2.75
- one level decrease in density: 0.25, 0.75, 0.75, 0.75, 0.5, 2, 0.25, 2.75
- second decrease in density: 0.25, 0.75, 1.5, 0.5, 2, 0.25, 2.75
- minimum density phrase: 8
- one level increase in density: 0.25, 0.5, 0.25, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 2.75
- second increase in density: 0.25, 0.5, 0.25, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 0.75, 2
- maximum density phrase: 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25

Based on a user-specified density value, a particular IOI pattern is chosen from the table. The user queries the table with a value between 0 and 1, and a linear conversion to a list index is done. The value returned is the IOI pattern at that index.

Density values can vary with time. One method for doing so is employing a TimespanMap, which is described below.

4.2 TimespanMaps

TimespanMaps are maps or dictionaries with intervals as keys to any kind of value and a specified total length for the whole TimespanMap. In the case of IOI pattern tables described above, the values are either density values (for determining which IOI pattern to use) or an IOI value to be selected from a particular IOI pattern.

A time in beats is passed to the dictionary. The interval that time falls in is determined to be the key, and the corresponding value for that interval is returned.

TimespanMaps can be used for any parameter, not just the ones described above. For example, in this system it also used for determining the amplitude and pitch of a particular triggering of a sample, as well as which sample to trigger.

The rate of change in a TimespanMap is up to the user and can change within the map. The number of items in a TimespanMap is limited only by memory or performance constraints. The range of time covered by a particular key can be as large or small as the user determines to be appropriate and is limited only by the Double data type in the Haskell language (double-precision floating point number).

A sample TimespanMap with a time length of four might look like this:

- length: 4
- 0: “a”
- 2: “b”
- 2.5: “c”

When passed a time of 0, the TimespanMap returns “a”. With 0.5, “a” is also returned. With a time of 2.25, “b” would be returned, and with a time of 3.5, “c”. If passed a time of 4, the list loops and “a” is returned.

Figure 2 shows the relationship between IOI patterns and density tables. It includes one TimespanMap mapping intervals to density values. Missing from the illustration (in order to keep it less cluttered) is the fact that the IOI patterns themselves are TimespanMaps from intervals to IOI values. The figure shows the calculation of the next IOI value of a running Player from beat 16 to beat 24. The example does not show the complete contents of the density map in order to save space, but actually generating a density map provides the full range of IOI patterns.

Convenience functions for TimespanMaps with random key values and interpolated TimespanMaps (in which fixed-length steps are linearly interpolated from a set of points in time) are provided in the library.

In addition to using TimespanMaps with IOI values and in density maps, they have been used for samples. Previously, one Player was assigned to each sample. In order to use 70 samples, it was necessary to instantiate 70 players. Managing 70 Players was challenging, so the sampler was rewritten to employ a TimespanMap. Subsets of sample sets are chosen at random and one is assigned for each interval in the map. When the sampler is triggered, the sample is chosen according to the current beat. By doing so, the number of Players needed was reduced by roughly a factor of 10.

4.3 mutable state

Several stateful parameters have now been described, and that state is stored in mutable data

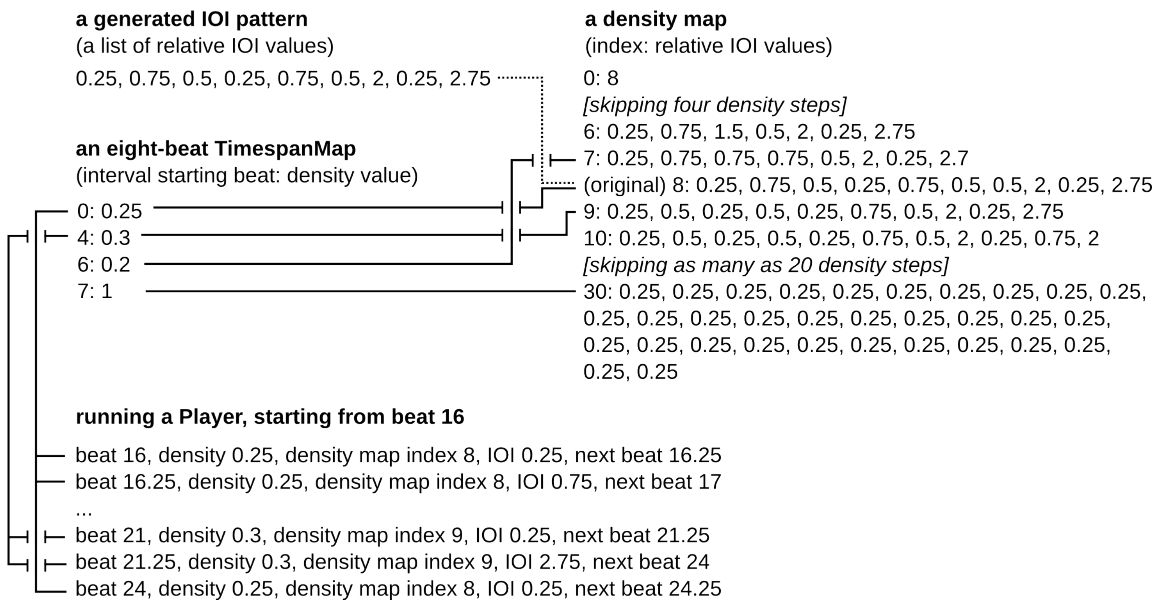
a generated IOI pattern
(a list of relative IOI values)
0.25, 0.75, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 2.75
an eight-beat TimespanMap
(interval starting beat: density value)
0: 0.25
4: 0.3
6: 0.2
7: 1
a density map
(index: relative IOI values)
0: 8
[skipping four density steps]
6: 0.25, 0.75, 1.5, 0.5, 2, 0.25, 2.75
7: 0.25, 0.75, 0.75, 0.75, 0.5, 2, 0.25, 2.7
(original) 8: 0.25, 0.75, 0.5, 0.25, 0.75, 0.5, 0.5, 2, 0.25, 2.75
9: 0.25, 0.5, 0.25, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 2.75
10: 0.25, 0.5, 0.25, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 0.75, 2
[skipping as many as 20 density steps]
30: 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25
running a Player, starting from beat 16
beat 16, density 0.25, density map index 8, IOI 0.25, next beat 16.25
beat 16.25, density 0.25, density map index 8, IOI 0.75, next beat 17
...
beat 21, density 0.3, density map index 9, IOI 0.25, next beat 21.25
beat 21.25, density 0.3, density map index 9, IOI 2.75, next beat 24
beat 24, density 0.25, density map index 8, IOI 0.25, next beat 24.25

Figure 2: an example showing the relationship between IOI patterns, density tables, and TimespanMaps, based on the example from section 4.1

structures appropriately called MutableMaps. They are maps containing keys and their corresponding values, often a string key and a TimespanMap as the returned value or a mapping of string to string. These maps are stored in a container. Previously, some stateful containers had been used (MVar), but it was suspected that the concurrent operations were not functioning properly in all cases. This was replaced with a TVar, which employs software transactional memory (STM) in order to safely carry out concurrent processes. A full discussion of STM can be found in Jones (2007).

Much of a performance consists of adjusting this state by changing values or adding new key-value pairs. The MutableMap data structure is intended to make doing so easier, with convenience functions for adding, deleting, or swapping values, as well as changing keys.

Because this is live coding, it is easy to write functions to execute multiple simultaneous state changes. Once such functions have been defined, they can be used with a single keyword: the function name.

5 performance method

This section explains how the system above is used to perform. It is divided into two parts: preparation necessary before a performance, and what occurs during a performance.

5.1 pre-performance preparation

An initiation module is prepared before performing. This module imports the various Conductive modules as well as other modules that are useful during performance, such as those in the Data.Map module or list utilities in Data.List. This module also provides functions for the initialization of the mutable state to be used, such as synthesis parameters and references to which sets of synthesis parameters each Player is to use. It also contains the definitions of actions to be used by those Players.

Immediately before the performance, necessary applications must be launched: JACK, Patchage, scsynth, calfjackhost, JAAA, ghci, and vim. The necessary routing must be set up between these applications, including manually routing audio in Patchage and setting vim to send to ghci.

The Conductive modules are loaded in GHCi as well as the previously prepared initialization routines. Preparation of scsynth is done, and samples are loaded into scsynth. For the sampler to be effective for an extended performance, it is necessary to prepare enough samples.

IOI patterns are generated along with their corresponding density tables. TimespanMaps are created to vary the density values used for selecting IOI patterns. Players are specified, along with their initial action and IOI functions. The density patterns and IOI patterns are assigned to Players, which use the current beat to determine a density and thus an IOI value to be used between instances of triggering a sample. The sample that is triggered is also determined by a TimespanMap.

5.2 during a performance

When the data has been prepared, Players are started according to user's intentions for the performance. As the Players are running, the state is manipulated by the user to vary the performance. This includes changing synthesis parameters or other system parameters. New patterns can be generated by the performer running the algorithmic composition routine described above. New density maps can be generated and assigned to Players. New samples can also be loaded. During the performance, Players can be stopped, restarted, or additional Players can be added and similarly manipulated. It is also possible to define new action functions or IOI functions during a performance. All Players are stopped when the performance has reached an end.

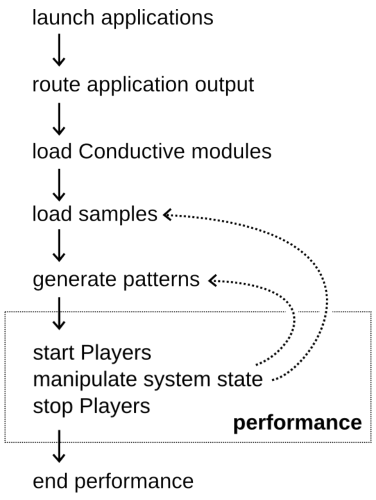

Figure 3: performance flowchart

6 conclusion

This section evaluates the results of using the system described above. It then describes some directions for future research.

6.1 evaluation of results

The approach above makes it possible to perform for extended periods of time, mostly limited by the amount of samples that have been prepared in advance. It is also necessary to use different sets of parameters when generating IOI patterns, such as differing numbers of scalars, differing core unit sizes, and so on.

It is challenging to keep a mental model of the parts described above during a performance, even though what has been described is mainly concerned with the timing of events and not timbre. This suggests that adding the complexities of generating different timbres through synthesis during performance will be burdensome. Part of this burden can be overcome through more practice with the system, but it seems that there is still a higher-level of abstraction to be achieved for optimal usage.

By using TimespanMaps with the sampler, it was possible to reduce the number of players for 70+ samples from 70 Players to between four and eight Players triggering hundreds of samples. This arrangement was found to be much more manageable and sonically-attractive than the previous one. It was very difficult previously to look at the list of Players and see which ones were playing and which ones were not. It also made changing the arrangement very hard, as Player-related functions often required long lists of Players. The current arrangement still uses lists of Players at times, but the lists are much shorter, rarely containing more than three or four Players.

The lack of continuous timbral modification through effects is a sore spot. A moving timbre can make the sound much more lively, but this is possible to a limited degree in the system above because of the design of the current sampler. Varying the sample of a particular Player does change the timbre, but sometimes a change which unfolds in a discernible direction over time can be a more effective compositional device. This has only been achieved in the current version for the sample pitch and amplitude.

In the system described above, parameters for synthesis are initiation-rate values. That means that the timbre of a particular event does not change over time other than what is contained in the original sample.

Changing from MVars to TVars with STM is thought to have solved some mysterious runtime misbehavior.

Current methods for organizing the text data or source code used during a performance are poor. As a result, the text in the text editor quickly becomes messy in the course of a performance. That makes it harder to stay in control of the performance or to run previously-defined functions at the most ideal times. Maximum effectiveness of use of the editor environment probably has not yet been achieved. Editor usage skills or tools to aid in this are probably needed.

6.2 future directions

Many things for this system can be developed. Those possibilities include the following ideas.

Further refinements of the abstractions described above can be done. That includes using value-lists rather than single values as core units in the IOI pattern generation process. More intelligent ways of generating the various densities of those patterns can be imagined.

Methods for generating IOI patterns with a greater sense of relationship is desired. While the patterns generated above are related in terms of density and rerunning a pattern generation function with the same parameters can yield similar patterns, there must be more sophisticated ways to generate sets of related patterns. More investigation into music theory and the algorithmic composition techniques of others is needed. Such research should be included in future versions of the pattern-generation functions.

Several chance operations are involved in this approach. It would be desirable to try weighted probabilities or other deterministic means as substitutes for those chance operations.

An increased use of pitched synths can be included. This will make it easier to achieve the timbral variation desired as well as expand the focus from its time-based focus at the moment to more involvement with the frequency domain. An efficient, easy-to-use method of synthesis that can also provide a wide range of timbres is being sought. Samples are musically effective but take a lot of time to prepare and remove a level of spontaneity that is desired.

Algorithmic control of effects at various stages would be nice. This means writing those effects and the corresponding action functions.

Player processes which alter other running Player processes should be experimented with, such as Players that stop and start other players. Another possibility to try in the near future is Players which change between sample sets.

Visualization methods for system state should be undertaken.

A convenience function for concatenating TimespanMaps is also desired.

Better methods for managing the code used in a performance should be sought.

7 acknowledgements

Thanks to Henning Thielemann and the reviewers for useful suggestions on the contents of this paper. Thanks also goes to Akihiro Kubota and Yoshiharu Hamada for research support.

8 bibliography

Adriaensen, Fons. 2004. "Kokkini Zita - Linux Audio." `http://kokkinizita.linuxaudio.org/linuxaudio/`.

Ariza, Christopher. 2011. "Two pioneering projects from the early history of computer-aided algorithmic composition." *Computer Music Journal* 35 (3): 40–56.

Bell, Renick. 2011. "An Interface for Realtime Music Using Interpreted Haskell." In *Proceedings of LAC 2011*.

Blackwell, Alan, and Nick Collins. 2005. "The Programming Language as a Musical Instrument." In *Proceedings of PPIG05*. University of Sussex.

Coutinho, C. 2010. "tslime." `http://www.vim.org/scripts/script.php?script_id=3023`.

Davis, Paul. 2006. "Ardour." `http://ardour.org`.

Drape, Rohan. 2009. *Haskell supercollider, a tutorial*.

Foltman, Krzysztof, Markus Schmidt, Christian Holschuh, and Thor Johansen. 2007. "Home @ Calf Studio Gear - Audio Plugins." `http://calf.sourceforge.net/`.

Hedges, Stephen A. 1978. "Dice music in the eighteenth century." *Music & Letters* 59 (2): 180–187.

Jones, SL Peyton, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. 1993. "The Glasgow Haskell compiler: a technical overview." In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*. Vol. 93.

Jones, Simon Peyton. 2007. "Beautiful concurrency." *Beautiful Code: Leading Programmers Explain How They Think*: 385–406.

Kostelanetz, Ric. 2002. *Conversing with Cage*. Routledge.

Marriott, Nicholas, and others. 2013. "tmux." `http://tmux.sourceforge.net/`.

Maurer, John. 1999. "A Brief History of Algorithmic Composition." `https://ccrma.stanford.edu/~blackrse/algorithm.html`.

McCartney, J. 2010. "SuperCollider Documentation." *http://www.audiosynth.com*.

McLean, Alex. 2012. "Computer Music Journal special issue on Live Coding \textbar TOPLAP." `http://toplap.org/cmj/`.

McLean, Alex, and Others. 2010. *TOPLAP website.* `http://www.toplap.org/index.php/Main_Page`.

Moolenaar, Bram. 2008. "The Vim Editor." `http://www.vim.org`.

Paul, David. 1997. "Karlheinz Stockhausen." *interview, Seconds Magazine* 44.

Robillard, David. 2011. "Patchage." `http://drobilla.net/software/patchage/`.

Schoenberg, Arnold. 1999. *Fundamentals of Musical Composition.* Ed. Gerald Strang and Leonard Stein. Faber & Faber.

Sorensen, A., and A. R. Brown. 2007. "aa-cell in Practice: An approach to musical live coding." In *Proceedings of the International Computer Music Conference.*

Tingen, Paul. 2004. "Autechre, recording electronica." *Sound on Sound* 19 (6): 96–102.

Varouhakis, John, and Martin Nordholts. 2008. *recordMyDesktop Version 0.3. 7.3.*

Xenakis, Iannis. 2001. *Formalized Music: Thought and Mathematics in Composition.* 2nd ed.. Pendragon Pr.

unknown. 2013. "live.code.festival 2013 – Call for Participation." `http://imwi.hfm.eu/livecode/call/`.