A Cognitive Dimensions analysis of interaction design for algorithmic composition software

Matt Bellingham
Department of Music and Music Technology, Faculty of Arts, University of Wolverhampton
matt.bellingham@wlv.ac.uk

Simon Holland
Music Computing Lab, Centre for Research In Computing, The Open University
simon.holland@open.ac.uk

Paul Mulholland
Knowledge Media Institute, Centre for Research in Computing, The Open University
p.mulholland@open.ac.uk

Keywords: POP-I.C. end-user applications, POP-II.B. design, POP-III.C. Cognitive Dimensions, POP-IV.B. user interfaces, POP-V.A. theories of design, POP-VI.F. exploratory

Abstract

This paper presents an analysis of the user interfaces of a range of algorithmic music composition software using the Cognitive Dimensions of Notations as the main analysis tool. Findings include the following: much of the reviewed software exhibits a low viscosity and requires significant user knowledge. The use of metaphor (staff notation, music production hardware) introduces multiple levels of abstraction which the user has to understand in order to use effectively: some instances of close mapping reduce abstraction but require the user to do more work. Significant premature commitment is not conducive to music composition, and there are clear opportunities for the greater provisionality that a piece of structurally-aware music software could provide. Visibility and juxtaposability are frequently compromised by complex design. Patching software reduces the hard mental operations required of the user by making the signal flow clear, although graphical complexity can have a negative impact on role-expressiveness. Complexity leads to error-proneness in several instances, although there are some tools (such as error-checking and auto-completion) which seek to ameliorate the main problems.

1. Introduction

This paper presents an analysis, using the Cognitive Dimensions of Notations (Green & Blackwell, 1998), of a representative selection of user interfaces for algorithmic music composition software. Cognitive Dimensions of Notations (CDN) are design principles for notations, user interfaces and programming language design, or from another viewpoint 'discussion tools' for designers (Green & Blackwell, 1998). For the purposes of this report, algorithmic composition software is software which generates music using computer algorithms, where the algorithms may be controlled by end users (who may variously be considered as composers or performers). For example, the algorithms may be created by the end user, or the user may provide data or parameter settings to pre-existing algorithms. Other kinds of end-user manipulation are also possible. The software accepts either precise or imprecise, indicative input from the user which is used to output music via emergent behaviour. A wide variety of algorithmic composition software is considered, including visual programming languages, text-oriented programming languages, and software which requires or allows data entry by the user. The paper considers a representative, rather than comprehensive, selection of software. The analysis also draws, where appropriate, on related discussion tools drawn from Crampton Smith (Moggridge, 2006), Cooper et al. (2007) and Rogers et al. (2011). Finally, the paper reflects on the compositional representation of time as a critical dimension of composition software that is implicit in several of the CDNs.

For detailed images from a wide variety of algorithmic composition software illustrating all issues touched on in this paper, see Bellingham et al. (2014), of which this paper is an abbreviated version.

2. Structure and connections

Sections, and the connections between them, are an important feature of much music. Several genres of music require a sectional structure (ABA, for example). An interface which matches the user's conceptual structure will reduce both repetition viscosity (caused when the software requires several actions to achieve a single goal) and knock-on viscosity (created when a change is made and the software requires further remedial action to restore the desired operation). Cooper et al. (2007) suggest three separate models for the perception of software; the *implementation model* of the software (how it works), the user's *mental model* (how the user imagines the software to work) and the *representational model* of the software (what the software shows the user). Repetition viscosity could be significantly reduced by better matching the mental model to the representational or implementation models. For example, if the implementation model made use of repeating sections the user could apply a change to the section and it would play back correctly both times. If the representational model showed repeating sections (as visual blocks, for example) and then relayed the changed material to all relevant repeats in the implementation layer, the user would input the desired changes once and they would be propagated out to the playback system. This model links to provisionality, which refers to the degree of commitment the user must make to their actions (Green & Blackwell, 1998). It allows users to make imprecise, indicative selections before making definite choices. Provisionality reduces premature commitment as it allows a composer to create sketches before allowing for specific details.

Some of the software under review allows the software to make selections within a given range. *SuperCollider* makes use of `.coin` and `.choose` messages for this reason; `.coin`, for example, represents a virtual toss of a coin. Other software, such as *Max*, can make use of pseudo-random numbers in parameters; this allows the composer to issue a command such as 'use a value between x and y'. Such selections can increase provisionality in the system (the degree of commitment the user must make to their actions), although more complex variations require significant planning which negates the benefits of being able to quickly create a functional piece of code. DAW software such as *Logic Pro* (Apple Inc., 2013) makes use of audio and MIDI loops to facilitate provisionality in composition and arrangement. Users are able to create sketches using loops, replacing them later in the process. Some music composition software allows the user to create music following basic harmonic or rhythmic parameters. *Noatikl* has preset objects which can be used to create sequences, and *Impro-Visor*'s preset algorithms allow for quick musical sketches based on a chord progression (Keller, 2012). One possible design would allow the user to specify the desired structure and then populate the sections. Repeated sections would require two 'pools' of information; those attributes common to both, and those for that specific iteration.

Most software in the domain allows links to be made only between pre-existing entities. In these cases the user is unable to say 'I don't know what is going here', which can be a useful option when composing. One possible solution to this problem would be to decouple the design of the patch/composition from the actualisation. This could take the form of a graphical sketching tool which would allow the user to test the structure and basic design of the patch.

The role-expressiveness of an element relates to how easily the user can infer its purpose (Green & Blackwell, 1998). The use of metaphor (such as mixing desks, synthesisers, piano rolls and staff notation) allows users to quickly understand the potential uses of each editor. Abstractions can be used to make software more effectively match the user's mental model (Cooper et al., 2007). Multiple steps can be combined to make the software conform to the user's expectations. Such abstractions can make use of a metaphor such as the hardware controls of a tape machine. Other hardware metaphors are used in current algorithmic composition software. The *Cylob Music System* by Chris Jeffs (2010) makes use of step-sequencer and drum machine designs, among others.

There are two types of links made in the software under consideration; one-way and symmetric. One-way links send data, whereas symmetric links can both send and receive information. One-way links, such as a send object in *Pure Data* or a variable in *SuperCollider*, do not reflect changes made elsewhere in the system. Visual audio programming systems typically use a patch cable metaphor and, as the majority of physical patching utilises a unidirectional (i.e. audio send or return) rather than bidirectional (i.e. USB) connection, software such as *Max* and *Pure Data* retains a one-way connectivity metaphor. Visual patching systems allow users to see links at the potential expense of increased premature commitment. The patch-cable metaphor used in visual programming languages makes dependencies explicit and reduces the potential for hidden dependencies. In addition, the use of patch cables is an example of closeness of mapping (Blackwell & Green, 2003).

Both graphical and text-oriented languages can make use of variables and hidden sends and returns. If users are required to check dependencies before they make changes to the software the search cost is increased. This in turn can lead to higher error rates (via knock-on viscosity). Abstractions can impose additional hidden dependencies; users may not be able to see how changes will affect other elements in the patch. Both *Max* and *Pure Data* allow graphical elements (such as colour, fonts and canvas objects) to be added to patches, enhancing the information available via secondary notation (extra information conveyed to the user in means other than the formal syntax). Graphical languages are substantially more diffuse (verbose) than text-oriented languages, and can make hidden dependencies explicit. Text-oriented languages typically have a lower role-expressiveness (how easily the user can infer the software's purpose). *ChucK* (Wang & Cook, 2013) is an interesting hybrid in this respect. Data can be 'chucked' from one object to another using the => symbol, the use of which imitates a patch cable. The other text-oriented languages reviewed do not make direct use of graphical or spatial interconnectivity. In this way *ChucK* makes limited use of Crampton Smith's second dimension of IxD; visual representation (Moggridge, 2006).

3. Time

The passage of time is highly significant when considering the representation of music. Time is not viewed as a separate entity in Cognitive Dimensions, although it is implicit in some dimensions. Payne (1993) reviewed the representations of time in calendars, which primarily focussed on the use of horizontal and vertical spatial information to imply the passage of time: in many cases a similar approach can be taken by music software. Sequencers, such as *Cubase* (Steinberg GmbH, 2013), frequently use horizontal motion from left to right to denote the passage of time. Trackers, such as *Renoise* (Impressum, 2013), frequently show the passage of time as a vertical scroll from top to bottom. Live coding software can present alternative representations of time, as seen in software such as *ixi lang* (Magnusson, 2014), *Overtone* (Aaron et al., 2011) and *Tidal* (McLean, 2014).

Much musical software makes use of cyclic time (loops), as well as linear time. Both of these kinds of time can be sequenced, or mixed, or arranged hierarchically at different scales, or arranged in parallel streams, or all of these at once. *Tidal*, for example, has been developed to allow the representation of linear and cyclic time simultaneously (Blackwell et al., 2014). Software written to perform loop-based music frequently uses a different interface to denote the passage of time. *Live* (Ableton, 2014) makes use of horizontal time in some interface components; other interface elements allow the user to switch between sample and synthetic content in real-time with no time representation. *Mixtikl* (Intermorphic, 2013) is a loop-based system and, in several edit screens, does not show the passage of time at all as the user interacts with the interface.

In a classic paper, Desain and Honing (1993) discuss different implicit time structures in tonal music. They point out that, in order to competently speed up piano performances in certain genres, it is no good simply to increase the tempo. While this may be appropriate for structural notes, decorations such as trills tend to need other manipulations such as truncations without speed-ups or substitutions to work effectively at different tempi. Similarly, elements of rhythm at different levels of periodicity, for example periodicities below 200 ms vs. above 2 seconds, may require very different kinds of compositional manipulation since the human rhythm perception (and composers and performers) deal very differently with periodicities in these different time domains (Angelis et al., 2013; London, 2012). In a related sense, Lerdahl & Jackendoff (1983) uncover four very different sets of time relationships in harmonic structures in tonal music.

Honing (1993) differentiates between tacit (i.e. focussed on 'now'), implicit (a list of notes in order) and explicit time structures. Some of the software under review can be considered in this way; for example, some modes of operation in *Mixtikl* and *Live* utilise tacit time structures, the note lists in

Maestro Genesis and *MusiNum* are implicit time structures, and software such as *Max* or *Csound* can generate material which uses explicit time structures. The flexibility of many of the programming environments under consideration means that the user can determine the timing structures to be used. Honing (1993) also applies the same process to structural relations: he suggests that there are tacit, implicit and explicit structural relations. A system which uses explicit structural relations would allow the musical structure to be both declarative and explicitly represented.

4. Complexity and stability

There are several compositional software interfaces which range from the highly complex and flexible to simple, limited designs. Viscosity - a measurement of the software's resistance to change - is not necessarily a negative attribute. Highly viscous software can present a user with a single, stable, well defined use-case. An example of this is *Wolfram Tones* (Wolfram Research Labs, 2011) which presents the user with a limited control set as a 'black box' (Rosenberg, 1982). Another example is *Improviser for Audiocubes* (Percussa, 2012), in which the complexity of the performance is generated by the physical layout of the Audiocubes (Percussa, 2013). As a result, keeping the sequencing interface simple avoids over-complicating the composition and performance processes. This simplicity, however, increases the viscosity and arguably limits compositional opportunity.

The text-oriented systems under review exhibit poor discriminability due to easily confused syntax, which invites error (Blackwell & Green, 2003). For example, Thomas Schürger's *SoundHelix* (2012) produces code with a large number of XML tags, potentially reducing human readability and increasing the time taken to write the commands. Such a system increases the error-proneness of the system (whether the notation used invites mistakes). Issues of this type can be ameliorated by the syntax checking seen in the Post windows of *SuperCollider* and *Pure Data*, in which errors are outlined in a limited way. A more thorough error-checking system would be a significant improvement in the software's usability. *SuperCollider 3.6* introduced an IDE (Integrated Development Environment) based design, including autocompletion of class and method names. Such a system significantly reduces errors introduced by mistyping.

There are several music metaphors used in the software in this domain which require the user to be conversant in music theory. *Harmony Improvisator* (Synleor, 2013) requires input in the form of scales, chords and inversions. *Noatikl* (Intermorphic, 2012) uses abstractions to create what it refers to as 'Rule Objects' ('Scale Rule', 'Harmony Rule', 'Next Note Rule' and 'Rhythm Rule') to control how the software generates patterns. The *Algorithmic Composition Toolbox* (Berg, 2012) makes reference to note patterns and structures. Roger Dannenberg has explained how staff notation is rich in abstractions (1993); software which uses elements of staff notation is building abstractions on top of abstractions. An example of a consistent design is *Mixikl* (Intermorphic, 2013). The design language refers metaphorically to both hardware synthesisers (the use of photorealistic rotary potentiometers and faders) and patching (patch cables which 'droop' as physical cables do). *Fractal Tune Smithy* (Walker, 2011) makes use of a less consistent design language. The design makes use of notation, piano roll, hardware-style controls, text-based data entry and window and card metaphors. The software is, as a result, highly capable of a wide variety of tasks but potentially at the expense of usability. There can also be consistency issues when software does not use standard operating system dialogue boxes. An example is *SuperCollider*'s save dialogue (McCartney, 2014), in which the 'Save' button is moved from the far right (the OS standard) to the far left. This is a clear example of poor consistency which could lead to unintended user error.

5. Summary

There are opportunities for future work to consider the design of structurally-aware algorithmic composition software. It would be interesting to further employ Cognitive Dimensions in suggesting concrete improvements to the design of the software under review. A full review of time with reference to the CDN using the format suggested by Blackwell et al. (2001) would be a useful process. The CDN is an evolving body of work and there are several new dimensions which could be utilised in future work in the area.

The issue of time raises particular questions. Algorithmic composition tools use varied interaction designs, and may promiscuously mix diverse elements from different musical, algorithmic and interaction approaches. Consequently, such tools can raise challenging design issues in the compositional representations of time. To some degree, these issues parallel similar issues in general programming, for example concerning sequence, looping, hierarchy and parallel streams. However, growing knowledge about how people perceive and process different kinds of musical structure at different time scales suggests that the design of algorithmic composition tools may pose a range of interesting new design issues. We hope that this paper has made a start in identifying opportunities to create or extend design tools to deal better with these challenging issues.

References

Aaron, S., Blackwell, A. F., Hoadley, R. and Regan, T. (2011) 'A principled approach to developing new languages for live coding', *Proceedings of New Interfaces for Musical Expression*, pp. 381–386.

Ableton (2014) 'Ableton live 9', [online] Available from: https://www.ableton.com/en/live/new-in-9/ (Accessed 14 February 2014).

Angelis, Vassilis, Holland, Simon, Upton, Paul J. and Clayton, Martin (2013) 'Testing a computational model of rhythm perception using polyrhythmic stimuli', *Journal of New Music Research*, 42(1), pp. 47–60.

Apple Inc. (2013) 'Logic pro x', [online] Available from: https://www.apple.com/uk/logic-pro/ (Accessed 24 October 2013).

Bellingham, Matt, Holland, Simon and Mulholland, Paul (2014) *An analysis of algorithmic composition interaction design with reference to cognitive dimensions*, Milton Keynes, UK, The Open University, [online] Available from: http://computing-reports.open.ac.uk/2014/TR2014-04.pdf.

Berg, Paul (2012) 'Algorithmic composition toolbox', [online] Available from: http://www.koncon.nl/downloads/ACToolbox/ (Accessed 26 January 2014).

Blackwell, A.F., Britton, C., Cox, A., Green, T.R.G., et al. (2001) 'Cognitive dimensions of notations: cognitive dimensions of notations: design tools for cognitive technology', In Beynon, M., Nehaniv, C., and Dautenhahn, K. (eds.), *Cognitive technology 2001 (lNAI 2117)*, Springer-Verlag, pp. 325–341.

Blackwell, Alan and Green, Thomas (2003) 'HCI models, theories, and frameworks: toward a multidisciplinary science', In Carroll, J. M. (ed.), San Francisco, Morgan Kaufmann, pp. 103–134.

Blackwell, Alan, McLean, Alex, Noble, James and Rohrhuber, Julian (2014) 'Collaboration and learning through live coding (dagstuhl seminar 13382)', *Dagstuhl Reports*, Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 3(9), pp. 130–168.

Cooper, Alan, Reimann, Robert and Cronin, David (2007) *About face 3: the essentials of interaction design*, 3rd ed. John Wiley & Sons.

Dannenberg, Roger B. (1993) 'Music representation issues, techniques, and systems', *Computer Music Journal*, 17(3), pp. 20–30.

Desain, Peter and Honing, Henkjan (1993) 'Tempo curves considered harmful', In Kramer, J. D. (ed.), *Time in contemporary musical thought*, Contemporary Music Review, pp. 123–138.

Green, Thomas and Blackwell, Alan (1998) 'Cognitive dimensions of information artefacts: a tutorial', [online] Available from: http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf (Accessed 18 September 2013).

Honing, Henkjan (1993) ‘Issues on the representation of time and structure in music’, *Contemporary Music Review*, 9(1-2), pp. 221–238.

Impressum (2013) ‘Renoise’, [online] Available from: http://www.renoise.com (Accessed 24 October 2013).

Intermorphic (2013) ‘Mixtikl’, [online] Available from: http://www.intermorphic.com/tools/mixtikl/index.html (Accessed 27 March 2013).

Intermorphic (2012) ‘Noatikl’, [online] Available from: http://www.intermorphic.com/tools/noatikl/index.html (Accessed 27 March 2013).

Jeffs, Chris (2010) ‘Cylob music system’, [online] Available from: http://durftal.com/cms/cylobmusicsystem.html (Accessed 22 March 2013).

Keller, Robert (2012) ‘Impro-visor’, Harvey Mudd Computer Science Department, [online] Available from: http://www.cs.hmc.edu/~keller/jazz/improvisor/ (Accessed 27 March 2013).

Lerdahl, Fred and Jackendoff, Ray (1983) *A generative theory of tonal music*, MIT Press.

London, Justin (2012) *Hearing in time*, Oxford University Press.

Magnusson, Thor (2014) ‘Herding cats: observing live coding in the wild’, *Computer Music Journal*, 38(1), pp. 8–16.

McCartney, James (2014) ‘SuperCollider’, [online] Available from: http://supercollider.sourceforge.net (Accessed 26 January 2014).

McLean, Alex (2014) ‘Tidal - mini language for live coding pattern’, [online] Available from: http://toplap.org/tidal/ (Accessed 16 May 2014).

Moggridge, Bill (2006) *Designing interactions*, MIT Press.

Payne, Stephen J. (1993) ‘Understanding calendar use’, *Human-Computer Interaction*, 8(2), pp. 83–100.

Percussa (2013) ‘Audiocubes’, http://percussa.us/, [online] Available from: http://percussa.us/ (Accessed 27 March 2013).

Percussa (2012) ‘Improvisor’, [online] Available from: http://land.percussa.com/audiocubes-improvisor/ (Accessed 22 March 2013).

Rogers, Yvonne, Sharp, Helen and Preece, Jenny (2011) *Interaction design: beyond human-computer interaction*, 3rd ed. John Wiley & Sons.

Rosenberg, Nathan (1982) *Inside the black box: technology and economics*, Cambridge University Press.

Schürger, Thomas (2012) ‘SoundHelix’, [online] Available from: http://www.soundhelix.com/ (Accessed 26 March 2013).

Steinberg GmbH (2013) ‘Cubase’, [online] Available from: http://www.steinberg.net/en/products/cubase/start.html (Accessed 24 October 2013).

Synleor (2013) ‘Harmony improvisator’, [online] Available from: http://www.synleor.com/improvisator.html (Accessed 27 March 2013).

Walker, Robert (2011) ‘Tune smithy’, [online] Available from: http://www.robertinventor.com/software/tunesmithy/music.htm (Accessed 27 March 2013).

Wang, Ge and Cook, Perry R. (2013) ‘ChucK’, CCRMA, [online] Available from: http://chuck.cs.princeton.edu/ (Accessed 26 March 2013).

Wolfram Research Labs (2011) ‘WolframTones’, Wolfram Alpha, [online] Available from: http://tones.wolfram.com/ (Accessed 27 March 2013).