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, using cognitive dimensions (Green & Blackwell, 1998), of a representative selection of user interfaces for algorithmic composition software. Cognitive dimensions 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. 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 not satisfactorily addressed by cognitive dimensions, or any of the other discussion tools.

1 Introduction

The paper considers all fourteen cognitive dimensions in turn, using each dimension to illuminate design issues in a wide variety of user interfaces for algorithmic composition. Finally, issues in interaction design for algorithmic composition that are arguably not well addressed by cognitive dimensions, such as the compositional representation of time, are considered.

2 Viscosity

Viscosity is a measurement of the software's resistance to change, or how easily the user can change a patch once it has been written. *Repetition viscosity* is caused when the software requires several actions to achieve a single goal. *Knock-on viscosity* is created when a change is made and the software requires further remedial action to restore the desired operation (Green & Blackwell, 1998).
Visual patching languages such as *Max* (Cycling '74, 2014[1]) and *Pure Data* (Puckette, 2014) can become highly viscous. Connections are made with virtual patch cables, and when there are multiple

1 *Cycling '74* is the name of the company that produces *Max*.

connections the patch can become difficult to read and work with (sometimes referred to as 'spaghetti').

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

The use of subpatches can ameliorate some viscosity issues. Subpatches allow the programmer to create objects with inlets and outlets which can abstract and simplify a patch at a given level. Subpatches can also be used to easily copy material, further reducing repetition viscosity. Abstracted connections ('send' and 'return' objects, for example) allow for non-visual connections with an increase in search cost (see *hidden dependancies*).

The structure of a piece of music, depending on the genre, can be a crucial element of the composition. If the genre requires a strongly structured piece there will be a conceptual structure (ABA, for example). An interface which matches the user's conceptual structure will reduce both repetition and knock-on viscosity. 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 (see Provisionality).

Some text-oriented programming languages such as *Csound* (Vercoe, 2014) and *SuperCollider* (McCartney, 2014) allow the user to create elements that can be easily reused and redefined. Such polymorphism (either ad hoc or parametric depending on context) can reduce repetition viscosity as changes in the variable's parameters will cascade to all instances. The 'rules' implemented in *Noatikl* (Intermorphic, 2012) allow for a reduced viscosity via cascading changes in a similar way. Knock-on viscosity can be created by introducing wide-ranging changes in this way.

There are links between viscosity and another of the dimensions, *premature commitment*. Premature commitment refers to situations in which the user has to make a decision before they have access to all relevant information. A piece of software with high viscosity makes it hard to amend work once it has been started. It is reasonable to assume that many compositions start with sketches that develop; high viscosity, such as is found in visual patching software, makes such development difficult.

Viscosity in software 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.

Live coding (editing code while it generates sound) requires software with a low viscosity. Elements need to be individually controllable with an efficient syntax, low repetition viscosity and minimal knock-on viscosity. *Impromptu* (Sorensen, 2010) is a Lisp-based environment which allows for the real-time creation and control of objects. *Usine* (Sens, 2013) has a GUI which allows for mouse and keyboard control, with a graphical signal flow based on a modular synthesis concept.

3 Abstraction

An abstraction presents the user with a representation which is closely aligned with the semantic meaning of the entity. The implementation of the entity is hidden, or *abstracted*. The *abstraction barrier* (Green & Blackwell, 1998) describes the number of abstractions the user must master before

using the software. Graphical languages typically have a lower abstraction barrier than text-based languages. Green and Blackwell describe three classes of software; *abstraction-hungry* systems which require user-defined abstractions, *abstraction-tolerant* systems which permit them, and *abstraction-hating* systems which do not allow them (1998).

Many of the text-oriented languages used in the space (such as *Impromptu* (Sorensen, 2010) and *SuperCollider* (McCartney, 2014)) are abstraction-hungry. They frequently also have a high abstraction barrier as they require the user to learn the syntax and abstractions used. An example of abstraction-tolerant software is the *Algorithmic Composition Toolbox* by Paul Berg (2012), which presents objects to the user and allows the creation of new abstractions. The software makes use of musical metaphors (such as a rudimentary piano roll) for some elements. The user defines objects such as sections, shapes, masks and note structures which the program's 'generators' use to create new material. A selection of software in the field do not require the user to interact with the method of generation; these are abstraction-hating systems. Robert Walker's *Fractal Tune Smithy* (2011) and Jonathan Middleton's *Musical Algorithms* (2004) both require musical input which is then acted on using algorithms which are both hidden from, and inaccessible to, the user.

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. The processes involved in saving the play state, halting and saving sound generation and effects processing, and then reloading the state to allow the track to continue need to be abstracted to reinforce the metaphor. The metaphor imposes some of the characteristics of a tape machine: the tape is in motion, stopped, and then resumes playback from the same place when set into motion.

There are several music metaphors used in the software in the field 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 manuscript is rich in abstractions (1993); software which uses elements of manuscript is building abstractions on top of abstractions.

Abstraction has a link to *visibility*, another of the cognitive dimensions. A high level of abstraction can result in low visibility. An effective design would be for the software to have a low abstraction barrier but be abstraction-tolerant. Such a design would allow new users to work with the language without writing new abstractions, while more advanced users could write abstractions when appropriate.

4 Hidden Dependencies

Hidden dependancies occur when important links between entities are not fully visible. There is a *search cost* which reflects the effort required to locate the dependancy (Green & Blackwell, 1998).

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. The patch-cable metaphor used in visual programming languages makes one-way dependancies explicit and reduces the potential for hidden dependancies. 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. MIDI, 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.

Both graphical and text-oriented languages can make use of variables and hidden sends and returns. If users are required to check dependancies 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 dependancies; users may not be able to see how changes will affect other elements in the patch.

5 Premature Commitment

Premature commitment refers to constraints on the order of operation, which leads to the user making a decision before all relevant information is available. *Enforced lookahead* describes how the user is forced to decide on implementation detail before they would otherwise be ready to (Green & Blackwell, 1998).

While experienced users can leverage their understanding of a piece of software to minimise premature commitment, less experienced users may have to rewrite a patch as it develops. Alan Cooper refers to 'survivors' (Cooper, 2004); those who manage to use software but find the process of composition is made more difficult by the limitations of the tool. He describes the cognitive dissonance caused by tension between the user's mental model and the implementation and/or representational models (Cooper et al., 2007). The software can dictate the way the composer writes.

Most software in the algorithmic space allow 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.

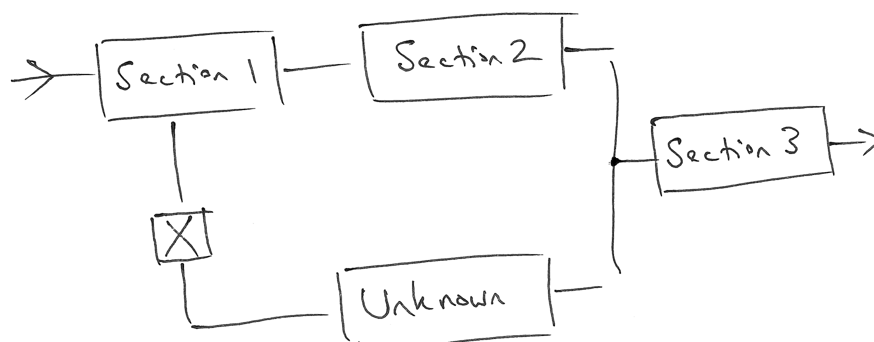

Image 1 - Potential layout for structure-aware composition software

There is a link between premature commitment and the viscosity of the programme; if viscosity is low then the user can redesign their patch with relative ease. Graphical systems such as *Noatikl* show the order of actions using a signal flow analogy which offers an easy way to reorder, thereby reducing viscosity.

Green & Petre (1996) introduce two subsets of premature commitment; *commitment to layout* and *commitment to connections*. Graphical systems such as *Max* encourage users to start with one element and then expand the patch. This forces the user to commit to the layout, and as the patch grows in complexity the viscosity of the software can limit subsequent changes. Graphical systems suggest a commitment to connections as significant planning is needed to design a flexible patch. A simple patch using a patch-cable analogy can be easily readable, but increased complexity can lead to a visually congested patch which is hard to maintain. This leads to higher viscosity.

6 Secondary notation

Secondary notation refers to extra information conveyed to the user in means other than the formal syntax (Green & Blackwell, 1998). Examples of secondary notation are in the collection of controls in *Mixtikl* (Intermorphic, 2013) and in the design of the *Cylob Music System*. Information conveyed by placement is known as *escape from formalism* (Green & Blackwell, 1998). The spacial placement of the controls adds to the information available, making the device easier to learn. Both *Max* and *Pure Data* allow for graphical elements (such as colour, fonts and canvas objects) to be added to patches.

Another example of secondary notation is code indentation; Green refers to this as *redundant recoding* (1998). Indentation helps to improve legibility and comprehension when reading and writing code. Indentation is used in all of the text-oriented languages under review. Object-oriented languages (such as *SuperCollider*) and XML-based syntax (such as that used in *SoundHelix* by Thomas Schürger (2012)) make significant use of the placement and context of commands for both the readability and functionality of the code. Adding comments to code is another example of secondary notation, and all of the musical programming languages under consideration allow for commenting.

There is a link between secondary notation and viscosity. If a patch's structure is changed the secondary notation (such as the placement of controls) can also be affected. The design of the patch therefore needs to include the required secondary notation which will lead to a higher viscosity due to the lack of flexibility in future changes.

7 Visibility

Visibility is the ability to view components easily. *Juxtaposability* is the ability to view two components simultaneously; this can be useful when comparing two elements (Green & Blackwell, 1998). There is a balance to be struck in composition software between having too much information and not having enough to complete a given task. Key parameters must be made visibility without introducing clutter to the design.

The user needs to be informed of their current position in the control tree. Software such as *Mixtikl* shows the current position by having a main view and using windows to access specific elements. Animation and placement denote the layer of the interface that is currently open. *Pure Data* allows users to create subpatches that open in successive layers on the screen; the user can use the 'Window' menu to view the open windows and to move to a specific place but the interface lacks a clear 'breadcrumb'-like structure (Adkisson, 2005).

Form-based data entry, as used in software such as *Tune Smithy* and the *Algorithmic Composition Toolbox*, allows the user to review multiple parameters simultaneously. Such designs do not allow the user to see older entries as they are replaced, which has a negative impact on the juxtaposability of the software. The user is asked to remember the old settings, increasing the work required of the user (Cooper et al., 2007).

Data flow visibility is variable in the software under consideration. Graphical languages such as *Max* and *Pure Data* can exhibit excellent data visibility within single patches, although the use of send and receive objects can impact on this. *Noatikl* and *Mixtikl* show data flow very clearly. Visibility in text-oriented systems is lower and the user may have to wireframe the patch separately before creating the code.

8 Closeness of mapping

Mapping refers to the correlation between the interface and the actual tasks being performed by the software. A close mapping (Green & Blackwell, 1998) is modelled on the implementation model (Cooper et al., 2007). In this case the software directly represents the way the software works, rather than abstracting this information. An example of this is *Pure Data* (Puckette, 2014). Some operations require the user to understand processes which are normally abstracted; for example, if a user is to play an audio file they need to create a line~ object to play each sample of the audio file in the required time. *Pure Data* also contains a large number of abstractions which represent a more distant mapping.

A distant mapping (Green & Blackwell, 1998) presents a significantly different representation model (Cooper et al., 2007), potentially requiring the use of new concepts. Distant mapping allows the interface to better match the user's mental model (Cooper et al., 2007). An interesting example of this is *Maestro Genesis* (Szerlip & Hoover, 2012), which uses the metaphor of animal breeding to allow the user to control the characteristics of 'generations' of music. The software abstracts the generation algorithms behind a 'DNA' button with an icon of a DNA double helix; the user can select from the resulting 'generations'.

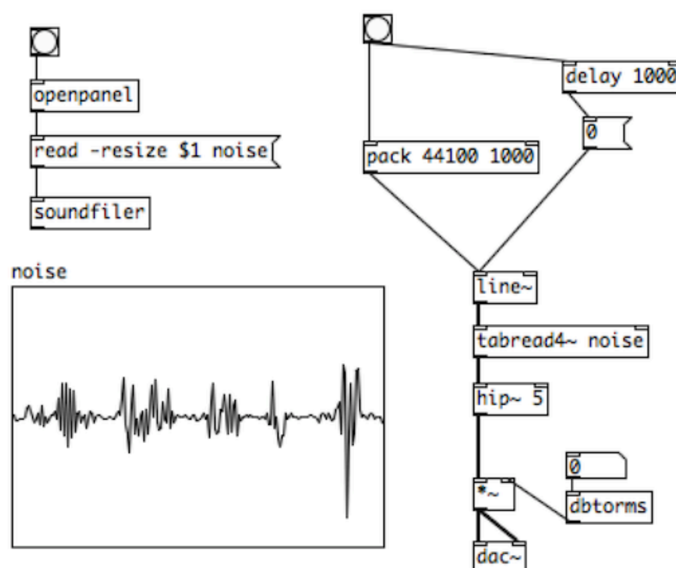

Image 2 - Sample playback in Pure Data, displaying a close mapping between the interface and the task being performed by the software

9 Consistency

Consistency refers to the way in which similar semantics are used in the user interface design (Green & Blackwell, 1998). If an interface is consistent it can positively affect the learnability of the software. The consistency of the user interface design affects usability more than learnability; once a user has learned the interface consistency is less important.

An example of a consistent design is *Mixikl*. The design language leverages both hardware synthesisers (the use of photorealistic rotary potentiometers and faders) and patching (patch cables which 'droop' as physical cables do). *Fractal Tune Smithy* 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, 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.

10 Diffuseness

Diffuseness measures the verbosity of language used in the software (Green & Blackwell, 1998). Shorter names and descriptions can reduce the memory work required of the user (Cooper et al., 2007). Examples of the use of short names can be seen in graphical programming languages such as *Strasheela* by Torsten Anders (2012) and Andrew Sorensen's *Impromptu* (2010). Inappropriate terseness can conversely lead to user error as there might be too little information for the user. As an example, an *Impromptu* patch can exhibit a high degree of diffuseness and can therefore be difficult to read. Diffuseness can be increased by making both variable names and comments more verbose.

11 Error-Proneness

The error-proneness of the system relates to whether the notation used invites mistakes (Green & Blackwell, 1998). The text-oriented systems under review exhibit poor discriminability due to easily confused syntax, which invites error (Blackwell & Green, 2003). Such issues 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.

12 Hard mental operations

Hard mental operations are those that place a high demand on the user's cognitive resources (Green & Blackwell, 1998); Cooper et al. (2007) also refer to the negative impact of requiring the user to undertake significant cognitive work.

Working in a code-based environment requires the internalisation of signal flow and the logical development of patterns. *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).

```
// actual FM using sinosc (sync is 0)
// (note: this is not quite the classic "FM synthesis"; also see fm2.ck)

// modulator to carrier
SinOsc m => SinOsc c => dac;

// carrier frequency
220 => c.freq;
// modulator frequency
20 => m.freq;
// index of modulation
200 => m.gain;

// time-loop
while( true ) 1::second => now;
```

Image 3 - An example of the => patching syntax in ChucK (Wang & Cook, 2013)

Software using the patching metaphor (graphic systems such as *Noatikl* (Intermorphic, 2012) or visual programming languages like *Max* (Cycling '74, 2014)) allow the visualisation of elements such as signal flow and boolean logic. Externalising the connections between elements reduces the cognitive work required of the user.

13 Progressive evaluation

Progressive evaluation allows the user to access the current state of their work at any time (Green & Blackwell, 1998). Such evaluation is important in music software as it allows for iterative development, an important compositional technique (Collins, 2005).

Two examples of software with minimal progressive evaluation options are both web-based. *Musical Algorithms* (Middleton, 2004) and *Wolfram Tones* (Wolfram Research Labs, 2011) require the user to upload and download data to a 'black box' with little control of the process. They are effectively offline, non-realtime processes which do not offer feedback before the final product is created.

Code-based systems such as *Csound* require a patch to be completed before the code can be evaluated and run. The use of variables allows for iterative progression without significant refactoring of the code; this leads to an approximation of progressive evaluation (a change - test - evaluate cycle).

14 Provisionality

Provisionality 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, although more complex variations require significant planning which negates the benefits of being able to create a basic wireframe. 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 of the software under review 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.

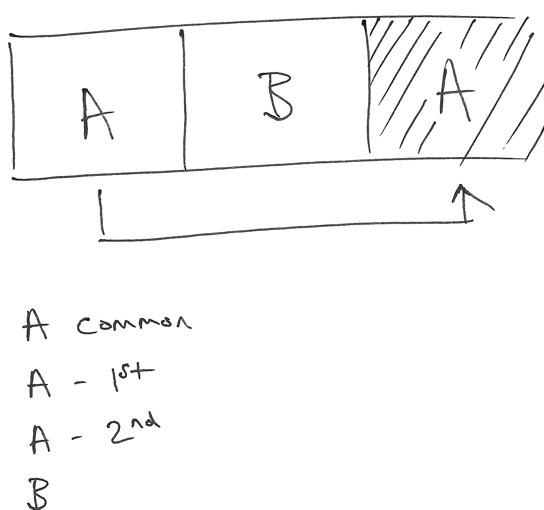

Image 4 - Possible layout allowing common material to be shared between sections

15 Role-expressiveness

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 manuscript) allows users to quickly understand the potential uses of each editor.

Text-oriented programming languages (such as *Csound*) are not as role-expressive as graphical systems unless the user is experienced in reading and understanding the code. *AthenaCL* by Christopher Ariza (2011) is a composition framework written in Python. The role-expressiveness of the code is low; the system allows for graphical output in the form of piano-roll-style mapping, but there is minimal use of metaphor.

Graphical languages (such as *Pure Data*) can be expressive as the connections between elements are clear. It is possible for a user to see an image of a *Pure Data* patch and understand the interrelationships of the objects and the purpose of the patch.

Impro-Visor (Keller, 2012) uses a lead sheet metaphor. The software is targeted at jazz musicians with an assumption that users will be familiar with the use of manuscript and chord progressions. The interface includes a transport control metaphor (play, pause, stop, record) and presents algorithmic choices as musician's names to denote the intended style. The role-expressiveness will therefore be high among the target users.

16 Towards design tools for the compositional representation of time

Some important issues in interaction design for algorithmic composition do not appear to be fully addressed by cognitive dimensions, or by other design discussion tools noted in the present paper. One such issue is the variety of demands made on compositional representations of time. Of course, notions of time are considered in design discussion tools. For example, Crampton Smith identified time as one of the four dimensions of interaction design (Moggridge, 2006). Time is not viewed as a separate entity in cognitive dimensions, although it is implicit in some dimensions. Kutar et al. (2000) reviewed the representation of various time granularities in TRIO, a real-time logic language (Morzenti et al., 1989), with reference to cognitive dimensions. There are various representations of time in software generally; research suggests that there is no one preferred type (Kessell & Tversky, 2008).

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.

The use of a static horizontal plane to denote time is common (sometimes with a moving pointer or index); *Tune Smithy*, *Maestro Genesis* and *Noatikl* are examples of this. Some of the reviewed software does not directly show the passage of time; offline, non-realtime software such as *Musical Algorithms* output files for use elsewhere. The text-oriented systems reviewed here are generally capable of generating graphical elements but this is not vital to their operation; the software can operate without any visual feedback for the user.

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. This can also be true of general programming languages, but is often a detailed focus of algorithmic composition software. 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* 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.

Audiocubes (Percussa, 2013), used with *Improvisor* (Percussa, 2012), use a static view of a perpetually looping step sequencer and so do not need to show time elapsing. *Audiocubes* are wireless hardware devices that use their orientation with relation to other cubes (via 4 IR ports) to trigger rhythmic and melodic patterns. Patterns are created in the *Improvisor* software according to the different orientations of each combination of cube surfaces. Performance is therefore achieved by the spatial placement of the cubes.

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 (see *Repetition Viscosity* above): 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.

17 Summary

Cognitive dimensions have proven to be a useful, if not quite comprehensive, tool in the analysis of algorithmic music software and in the articulation of issues affecting how usable these tools are and how well they work. Much of the reviewed software exhibits a low viscosity and requires significant user knowledge. The use of metaphor (manuscript, 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) to ameliorate the main problems.

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.

The issue of time raises particular questions and problems. 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.

18 References

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

Adkisson, Heidi P. (2005) ‘Breadcrumb navigation’, *Web Design Practices*, [online] Available from: http://www.webdesignpractices.com/navigation/breadcrumb.html (Accessed 04 September 2011).

Anders, Torsten (2012) ‘Strasheela’, [online] Available from: http://strasheela.sourceforge.net/strasheela/doc/index.html (Accessed 26 March 2013).

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).

Ariza, Christopher (2011) ‘athenaCL’, Flexatone HFP, [online] Available from: http://www.flexatone.org/article/athenaCLMain (Accessed 26 March 2013).

Bellingham, Matt, Holland, Simon and Mulholland, Paul (2014) *An analysis of algorithmic composition interaction design with reference to cognitive dimensions*, The Open University, [online] Available from: http://computing-reports.open.ac.uk/2014/TR2014-01.pdf.

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

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.

Collins, David (2005) 'A synthesis process model of creative thinking in music composition', *Psychology of Music*, 33(2), pp. 193–216.

Cooper, Alan (2004) *The inmates are running the asylum: why high-tech products drive us crazy and how to restore the sanity*, 2nd ed. Sams Publishing.

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

Cycling '74 (2014) 'Max', [online] Available from: http://cycling74.com/products/max/ (Accessed 10 February 2014).

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, T. R. and Petre, M. (1996) 'Usability analysis of visual programming environments: a 'cognitive dimensions' framework', *Journal of Visual Languages and Computing*, 7, pp. pp.131–174.

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.

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).

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).

Kessell, Angela M. and Tversky, Barbara (2008) 'Cognitive methods for visualizing space, time, and agents', Berlin. In Stapleton, G., Howse, J., and Lee, J. (eds.), *Diagrams 2008*, Springer-Verlag, pp. 382–384.

Kutar, Maria, Britton, Carol and Wilson, Jonathan (2000) 'Cognitive dimensions – an experience report', In A.F.Blackwell and E.Bilotta (eds.), *PPIG*, Psychology of Programming Interest Group, pp. 81–98.

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

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

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

Middleton, Jonathan N. (2004) ‘Musical algorithms’, *Northwest Academic Computing Consortium (NWACC)*, [online] Available from: http://musicalgorithms.ewu.edu/index.html (Accessed 27 March 2013).

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

Morzenti, A., Ratto, E., Roncato, M. and Zoccolante, L. (1989) ‘TRIO, a logic formalism for the specification of real-time systems’, In *Real time 1989*, pp. 26–30.

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).

Puckette, Miller (2014) ‘Pure data’, [online] Available from: http://puredata.info/ (Accessed 10 February 2014).

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).

Sens, Olivier (2013) ‘Usine’, Sensomusic, [online] Available from: http://www.sensomusic.com/usine/ (Accessed 14 February 2014).

Sorensen, Andrew (2010) ‘Impromptu’, [online] Available from: http://impromptu.moso.com.au/ (Accessed 14 February 2014).

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

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

Szerlip, Paul and Hoover, Amy (2012) ‘Maestro genesis’, Evolutionary Complexity Research Group, University of Central Florida, [online] Available from: http://maestrogenesis.org/ (Accessed 27 March 2013).

Vercoe, Barry (2014) ‘Csound’, MIT.

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).