Live Coding Patterns and a Toolkit for Pure Data

ANDREW R. BROWN

Griffith University, South Brisbane, Australia
Email: andrew.r.brown@griffith.edu.au

Creative activities often involve specific processes and techniques that reflect the unique nature of the activity. For live coders, these processes and techniques can be expressed as algorithms and functions in live coding languages. In many fields, these idiomatic processes are referred to as design patterns. Design patterns are important to understand because they can structure thought and direct users towards particular outcomes. This article examines the design patterns in live coding practices and languages, specifically focusing on the Live Coding Toolkit for Pure Data. Pure Data is a visual programming language, but few live coders have traditionally used it. This article explains how the Live Coding Toolkit allows Pure Data to effectively express the patterns of practice required for successful music live coding performance.

1. INTRODUCTION

Design patterns are abstractions that capture idiomatic tendencies of practices and processes. These abstractions are embedded in live coding languages as functions and syntax, allowing for the reuse of these practices in new projects. The use of idiomatic patterns is well established in musical practices; for example, in playing acoustic instruments and compositional techniques (Sudnow 1978; de Souza 2017). Music live coding adopts some of these and introduces new patterns particular to the demands of coding as performance. The variety of syntax and user interfaces across live coding languages can obscure the similarities between them. Identifying design patterns common across live coding languages can demystify any superficial diversity and provide guidance in the development of new languages. Understanding these patterns can aid in the design and use of live coding systems and provide a framework for analysing live coding independent of the programming language. This article discusses the development of such a framework and the creation of a new toolkit for live coding in the Pure Data (Pd) programming language, which is a visual language used for creating real-time interactive media (Puckette 1996).

McPherson and Tahıroğlu (2020) point out that all music technology systems afford idiomatic practices. Following Heidegger (1977), such affordances are widely acknowledged as a principle of technological thinking and practice. Heidegger considered technologies to be a form of 'enframing', a way of fixing a perspective, and that such framing is 'revealing' of an underlying understanding. Design patterns similarly make concrete our perceptions of what is afforded to us and/or the way we organise our world. The use of design patterns in live coding languages allows for (even promotes) certain musical outcomes, making the choice of design patterns crucial for the effectiveness of the language. This article examines the patterns commonly found in live coding performances and how they are implemented in code, using the Live Coding Toolkit (LCT) for Pd as an example.[1]

The design of live coding mini-languages plays a crucial role in shaping live coding practices, similar to how Norman (1988) describes how the design of products and interfaces can influence their usage. He builds on the insights of J. J. Gibson from behavioural psychology that individuals perceive opportunities for action in the environment and that actions are determined by the possibilities that can be seen. During live coding performances, the decisions made about the progression of music are influenced by the performer's understanding of the capabilities and opportunities presented by their tools, as well as their skill in utilising them.

Music live coding languages are designed to facilitate common musical patterns through the use of algorithms. This concept is not new, as it is rooted in the cultural structures of musical organisation, such as scales, tonalities, meters, and timbres, which have existed and evolved over a long period of time. Collins (2018) traces this history of algorithmic music going back centuries across many cultures. The longevity of this history can be attributed to the fact that, as Whitehead observed, 'Art is the imposing of a pattern on experience, and our aesthetic enjoyment is recognition of the pattern' (Whitehead and Price 1954: 225). These pattern-making activities are fundamental to human culture and continue to evolve in new forms of musical expression, including live

[1] https://github.com/algomusic/Live-Coding-Toolkit-for-Pure-Data/.

Organised Sound **28**(2): 264–275 © The Author(s), 2023. Published by Cambridge University Press.
doi:10.1017/S1355771823000365

coding performance, where performers manipulate musical structure at both micro and macro levels.

This article primarily focuses on the event-based organisation of sound and how patterns of notes over time generate music, but it also takes into account patterns in sound design and signal processing. The LCT, which is used as a case study in this article, implements several event-based functions on top of the Pd language, which is heavily oriented towards sound design (Puckette 1996). The role of the LCT in this article is twofold, its development precipitated reflection on necessary and appropriate pattern abstractions, and it also serves as a common exemplar of implementing of pattern-based considerations under discussion.

2. MUSICAL PATTERNS AND STRUCTURES

The study of Western musical patterns and structures is well established in music theory and musicology (Berry 1976; Temperely 2001), traditionally based on the analysis of written scores. Live coding mini-languages, on the other hand, are a form of musical notation articulated in a programming language (textual or visual) and interpreted by a computer in real-time. These notations often describe patterns and processes that generate individual events. For example, Tidal Cycles succinctly represents grouping, repetition and concurrency using the symbols [, *, and . respectively. As in this code snippet: `p "demo" $ s "[bd*3 . hh:2*2] . hh*4 cp"`. Live coding languages describe what Taube calls the 'musical metalevel', where the notation represents not just the composition but also the 'composition of the composition' (Taube 2004: 3).

The design of computer languages for algorithmic composition has a long history, dating back to the 1960s and 1970s with the development of languages such as the MUSIC series by Max Mathews and others, which used a component-based architectural metaphor. Other languages such as CMIX by Paul Lansky (1990) and CSound by Barry Vercoe and others, separated sound and events into 'score' and 'instrument' or 'orchestra', and Common Lisp Music by Bill Schottstaedt and others followed the MUSIC V approach of using lists of 'notes' to control sound generators. Puckette notes that the design of music languages involves balancing performative and compositional goals, with the relationship between processes and data being a key factor. He argues that the original goal of Pd was to 'remove the barrier between event-driven real-time computation and data' (Puckette 2004: 4). The LCT, introduced in section 6 and built on Pd, also aims to remove this barrier by encapsulating both process (algorithms) and data into a single object (abstraction).

In the 1980s, composers recognised the need for languages to support the creation and manipulation of musical patterns and structures beyond sound synthesis (Loy and Abbott 1985). Spiegel, for example, proposed the development of 'a basic "library" consisting of the most elemental transformations which have consistently been successfully used on musical patterns' (Spiegel 1981: 19). She outlined suggestions for processes and patterns that such a library might contain. Wishart, in his book *On Sonic Art* (1985), provided an analysis and tutorial on various techniques of computer music composition. Patterns such as these were incorporated into computer music languages, including those designed for live coding, such as Impromptu (Sorensen and Brown 2007). More recently, techniques from electroacoustic music have been joined by patterns commonly found in popular music and electronica, as evident in Algorave performances within the live coding community.

McPherson and Tahıroğlu (2020) investigated the impact of computer music language design and the patterns they emphasize, on composition and performance practices. They posit that there is 'a complex interplay between language, instrument, piece, and performance' (ibid.: 53) and that the creators of music programming languages should be considered as active participants in this creative conversation. This conversation involves three parties: the human, the computer and the language.

3. SHARED AGENCY

Live coding languages are designed to provide the performer with agency over the musical outcome. However, there is always a degree of agency in the language itself, and many algorithmic processes are specifically intended to provide the computer with influence (agency) over the outcome; for example, with the use of random functions. The emphasis on agency in language design often prioritises performer expressivity and flexibility, but it can be at odds with simplicity and readability. As previously argued by Brown (2016), when code is abstracted into black boxes, as is done in the LCT and other domain-specific languages, it limits access to direct manipulation of the algorithm. This creates a tension between a language's ability to empower a sense of agency in the performer and the algorithmic power of automation within the language. It may be that agency is a limited resource and that a balance needs to be found between the influences of the language, the machine, and the musician.

Handing over agency to the machine is a fundamental aspect of algorithmic and generative arts. For many live coders, the excitement of performance stems from the risks and rewards of unexpected

https://doi.org/10.1017/S1355771823000365 Published online by Cambridge University Press

outcomes. Live coding shares this embrace of the unexpected with other improvisational performance practices, while recognising that performances are a partnership between human and machine. The main difference being that a machine actor may not recognise an inappropriate outcome in the way a human actor might. From one perspective, the goal of a live coding language is to allow the machine to take control of the surface-level details of the performance, allowing the human performer to focus on issues such as development, structure, momentum and aesthetics.

Agency also exists in the pattern languages used in live coding music. These patterns (functions), inherent in mini-language elements, are partially replications of often-used techniques and habits, but they also have the potential to be used in new ways for new outcomes. The architect Christopher Alexander first noted this in his observations of how farmers build barns, writing that 'patterns in our minds are, more or less, mental images of the patterns in the world: they are abstract representations ... The patterns in the world merely exist. But the same patterns in our minds are dynamic. They have force. They are generative' (Alexander 1979: 181–2). This view of the suggestive power of patterns has an element of technological determinism, similar to Heidegger's idea of 'thrownness' inherent in all encounters with technologies (Heidegger 1977). However, even though languages and patterns have a directionality or affordance, each person uses the language differently and draws on their own skills and aesthetic preferences, enabling them to create a work that is unique to them.

4. DESIGN PATTERNS

Design patterns are 'the core abstractions from successful recurring problem solutions' (Riehle 1997: 218). Abstractions articulated as algorithmic processes are frequently discussed in the context of live coding. The inclusion of specific pattern-generating structures in mini-languages is a significant aspect of their differentiation, along with syntax and interface choices. In relation to design patterns for live coding, Roberts and Wakefield argue that 'abstractions are not merely structural convenience: through their constraints and affordances, abstractions effectively present a model of a world with which a live coder maintains discourse' (Roberts and Wakefield 2018: 303).

In Alexander's original conception, pattern languages were developed to capture well-tried solutions to common design problems in urban architecture. For Alexander, design patterns in architecture were a ubiquitous, if at times unconscious, tool for ensuring design quality:

> [E]very building, every town, is made up of certain entities which I call patterns: and once we understand buildings in terms of patterns, we have a way of looking at them ... Second, we have a way of understanding the generative processes which gave rise to these patterns ... these patterns always come from certain combinatory processes, which are different in the specific patterns they generate, but always similar in their overall structure, and in the way they work. (Alexander 1979: 11)

According to Alexander, the real work of any design process is the creation of a language, due to its ability to influence the generation of designs that use it. This may partly explain the numerous mini-languages for live coding, including the LCT.[2] Beyond personal preference for certain programming languages, it is likely that certain patterns of musical structure appeal to different musicians' concepts of music and stylistic preferences. For example, it appears that within the live coding community, those producing dance-like metrical music tend to gravitate towards languages that prioritise clear cyclic (repetitive) structures and sample-based sounds, such as Sonic Pi (Aaron and Blackwell 2013), ixi lang (Magnusson 2011), or Tidal Cycles (McLean 2014), while those producing work focused on arhythmic timbral exploration tend to gravitate towards languages that prioritise synthesis and signal processing, such as SuperCollider (McCartney 2002) or Pd.

In the 1990s, design patterns gained popularity in software development circles, particularly in Silicon Valley, California, near where Alexander resided, as a way to describe reusable coding structures that could be applied to various situations (Gamma, Helm, Johnson, Johnson and Vlissides 1994). Following this trend, Dannenberg and Bencina (2005) explored design patterns for computer music systems, with a focus on optimal scheduling algorithms. More recently, Magnusson and McLean (2018) examined patterns in computer music, using examples from their languages ixi lang and Tidal Cycles. They concluded that such domain-specific languages are a double-edged sword, as they can scaffold the building of musical structures while also directing compositions and performances down predefined pathways.

5. PERFORMANCE CONSIDERATIONS

Ultimately, live coders must navigate the practical challenges of making music in the high-pressure setting of live performance, using code that was not originally intended for such scenarios. In my experience, audiences' reactions to live coding are most extreme for professional computer programmers, who understand the importance of trial and error in coding

[2]https://github.com/toplap/awesome-livecoding.

https://doi.org/10.1017/S1355771823000365 Published online by Cambridge University Press

and debugging and are amazed by the risks taken to get the syntax right, first time, during live coding.

There are some important considerations driven by the practicalities of live performance that play into the design of live coding mini-languages. These practicalities have as much sway as the more philosophical concerns of abstraction and agency. Many of these considerations are covered in some detail by Roberts and Wakefield (2018) and those that relate specially to design pattern choices are highlighted in this section with some discussion of how the LCT treats these issues.

Automation is a fundamental aspect of computer programming, and for live coding it is essential for creating music so that the performer can focus on writing new code or modifying existing code. Live coding environments need to prioritise time as a key factor, and many general-purpose programming languages neglect time in favour of order of execution, assuming that as-fast-as-possible is all that is necessary in terms of temporality (Sorensen and Gardner 2010). The LCT is fortunate in that Pd, the language it is built on, already pays close attention to time and scheduling of events. However, as we will see, the LCT's functions provide a concise way of sequencing note events temporally. Hot swapping of code is closely related to automation. It refers to the ability to replace one operating function on the fly with another without interruption. This is in contrast to conventional computer programming where applications are halted to allow recompiling and then run again. Each live coding environment manages hot swapping in its own way. In the case of the LCT, and in Pd more generally, direct hot swapping of node graphs is not possible, although hot swapping of single objects is and PD is designed as a real-time system. Live parameter changes are well supported, and workarounds for relatively seamless substitution of new graph organisation are possible. However, these may not be idiomatic to standard Pd development processes, so using it in a live coding context requires special attention to ensure interruptions are minimised when editing or adding to a patch.

Efficiency in quickly creating a well-formed musical process is crucial in live coding as the timely initiation and progression of the work is essential for keeping the music engaging and the audience interested. In a live coding performance, a musician typically has a limited amount of time to write a piece, usually working on it while it is being performed. A significant factor in selecting design patterns for a mini-language is to provide ready-made functions for processes that might otherwise be time-consuming to repeatedly hand-code in each performance. This allows for more time to focus on the creative aspects of the performance.

Designing live coding mini-languages is a delicate balance between simplicity and flexibility. Live coding practice has been shown to require high cognitive load from performers, making it more challenging compared with instrumental performance (Sayer 2016). Fluency in touch typing can help alleviate some of this cognitive load, but it still requires a high level of conscious decision-making. For this reason, designers of live coding mini-languages must consider the balance between ease of use and expressiveness. Providing fewer choices can simplify the process but limit the musical outcomes. In the design of the LCT, this balance was evident at both the micro and macro levels. Each abstraction aimed to keep the number of inputs and outputs small to improve learnability and reduce errors. At the macro level, the desire to keep the library compact was balanced with the need to provide patterns for different stylistic contexts. However, this led to an increase in the number of objects over the development period to meet the demands of different musical situations.

Another language design factor effecting the cognitive load during live coding performance is the readability of code. In this regard, an advantage of text-based languages is that variables and functions can be given arbitrary names allowing easier identification of their purpose; for example, a sequence could be called 'bassline1'. In Pd and other visual languages, objects have fixed names with the downside that each instance can look similar. While comment text blocks can be added in Pd to annotate the patch, another way around this situation is to add an additional argument, which will be ignored by the computer, as a name to an object; for example [cycle 60 100 4] can be [cycle 60 100 4 baseline1].

In all languages there is a challenge surrounding the number and order of arguments. Typically, there are little or no cues for what these arguments should be so, again, succinctness and consistency aids memory. One interesting solution to this is Extempore's (Sorensen 2018) use of code snippets that insert boiler-plate code for particular patterns in the editor with placeholder defaults for arguments. The partial solution in the LCT is to have reasonable defaults for all parameters and make arguments optional so that it is straightforward to get the music started.

The live coding manifesto implores performers to 'show us your screens'.[3] The visual clarity of node-based visual programming languages, such as Pd, may be considered an advantage to audience appreciation of the algorithmic processes, however, these languages can become confusing as patches become more complex, leading to the argument that they do not scale well. The time pressures of live performance can exacerbate this issue, making it difficult to maintain neatness and organisation. Pd editing commands such

[3]https://toplap.org/wiki/ManifestoDraft.

https://doi.org/10.1017/S1355771823000365 Published online by Cambridge University Press

as Duplicate the Triggerize (to insert a node on a connector) can assist efficient live coding and the high-level abstractions in the LCT aim to address this complexity by reducing the number of objects required and improve the live coding experience.

Many live coding environments have explored adding GUI elements to enhance the text-based coding interface (Roberts, Wakefield, Wright and Kuchera-Morin 2015). Visual programming environments, such as Pd, may provide an advantage for live coding by offering GUI elements, such as sliders and buttons, to interact with the program. This allows for a more direct way of varying parameters and adjusting audio levels compared with typing numbers.

6. INTRODUCING THE LIVE CODING TOOLKIT FOR PURE DATA

The LCT is a mini-language built within Pd that provides a library of abstractions to support musical live coding. Pd uses a dataflow style of programming, where a visual graph of objects with specific functions, such as mathematical operators or MIDI message parsers, make up a program known as a 'patch' in Pd. The LCT includes Pd patches that appear as objects that can be added as an extra library of functions for Pd programmers to use. These functions include operations such as cycling through a data list, pitch quantization, generating probabilistic sequences and simple predefined synthesizers. Some of the LCT abstractions will be discussed in later sections of this article.

Because Pd is already a music-specific language, it may be unclear why a live coding toolkit is necessary. When building a live coding tool on top of general-purpose computer programming languages, sound and time are often low priorities that need to be addressed. Pd, however, already prioritises both. While Pd is oriented towards sound design and music composition, the LCT is a library designed to complement Pd's strengths in these areas. Pd supports event-based organisation, but lacks many event-based pattern primitives found in other live coding languages. The LCT provides some of these missing primitives.

The live coding community has many mini-languages,[4] which, like the LCT, are built on top of an underlying programming language. Designing a domain-specific language requires the developers to be explicit about decisions regarding possibilities for music organisation and structure and to offer specific functions for music-making. All mini-languages provide such a vocabulary for musical thinking and expression.

The LCT for Pd is a high-level language that aims for simplicity and efficient expression, not unlike other live coding languages, including ixi lang, Tidal, ChucK (Wang and Cook 2003), Gibber (Roberts and Kuchera-Morin 2011), Fluxus (Griffith 2007), FoxDot (Kirkbride 2016) and Sonic-Pi.

Most music live coding languages are text based, which is advantageous as text can be flexible and expressive. Pd, as a visual programming language, has its own advantages, such as a more graphical data flow representation and being less prone to minor syntactic errors. However, patching, which is usually done with a single mouse pointer, can be slower than typing in a text-based language. The LCT offers higher-level building blocks for common live coding patterns that can help compensate for the slower pace of patching while retaining many of the benefits of visualising data flow. Additionally, text-based live coding allows for significant edits to a function prior to re-evaluation, a technique that was established early in the development of live coding practices (Collins 2003) and refined over time (Sorensen, Swift and Gardner 2014). In patching environments such as Pd, updates to node connections take effect immediately, so the process of editing a patch involves techniques specific to those environments. These techniques include concurrent connection–disconnection of nodes, preparation of default values for Number boxes prior to connection to avoid discontinuities in parameter changes, and copying and modifying patch network sections for subsequent replacement.

Since Pd is already a powerful sound synthesis library, the LCT includes some basic synthesizers (pitched and percussion) written in Pd, similar to how other live coding mini-languages include sound generators written in SuperCollider. Users can also create their own sound patches in Pd. It also includes MIDI input and output pathways for controlling parameters with external MIDI controllers and outputting musical event triggers to external audio plugins and synthesisers. Given the improvisational and compositional focus of the LCT and many live coding practices, the next section of this article will explore design patterns and processes commonly found in live coding practices and included in the LCT.

7. LIVE CODING DESIGN PATTERNS

Like any practice, live coding relies on certain conventions and tropes, referred to here as design patterns. This is not to say that all live coding is the same, as the outcomes of live coding are certainly diverse, but rather that the task of creating algorithmic music as a real-time performance requires specific solutions. In the following sections, a variety of these common patterns will be discussed and how they have been implemented in the LCT will be outlined.

[4] https://github.com/toplap/awesome-livecoding#languages.

https://doi.org/10.1017/S1355771823000365 Published online by Cambridge University Press

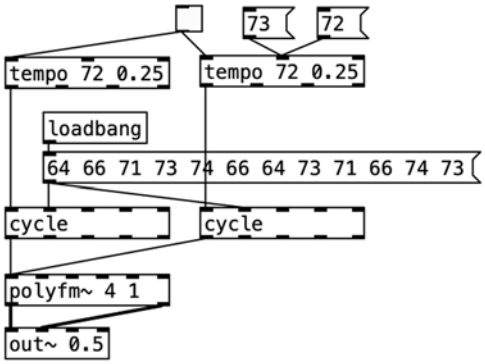

Figure 1. An implementation of *Piano Phase* in the Live Coding Toolkit for Pd.

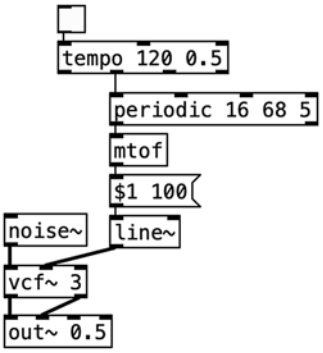

Figure 2. A 'beach wind' LCT patch using slow periodic oscillation of a resonant low pass filter cutoff.

7.1. Timing

Keeping time is fundamental to all musical practices. Every live coding language has a method for keeping a steady pulse and scheduling events. The technical details of these processes are often quite low level in the programming environment. For example, the computer processor clock is typically very fast and very accurate, yet operating system timers built on this often need to compromise on time accuracy when managing a multi-threaded environment, audio processes may not be the highest priority in these environments, and programming languages add yet another layer of execution prioritisation (Dannenberg and Bencina 2005; Sorensen and Gardner 2010). Many musical genres rely on a single clock to keep parts in time while multiple time bases or variations in timing are used for more experimental forms of music.

The LCT Tempo abstraction uses Pd's Metronome object as the foundation for its timing, but with the added feature of a Tempo abstraction. This Tempo abstraction changes the default time measurement to beats per minute, and also allows for beat subdivision/multiplier value for the output pulse. The Tempo abstraction outputs a standard Bang message, an incrementing beat counter, an inter-onset timer and a reset trigger on start, which is useful for synchronising with other objects. While one Tempo object is sufficient when parts are synchronised, multiple Tempo objects can be used. The patch in Figure 1 demonstrates an implementation of Steve Reich's *Piano Phase* duet, which uses two Tempo objects set to ¼ beat pulses (semiquavers/sixteenth notes) and can be set to different tempi to produce Reich's desired effect of one instrument speeding up while the other stays at a fixed tempo, which shifts the parts through different temporal phase relationships.

7.2. Periodicity

Periodic patterns, like those described by trigonometry and more straightforward cyclic repetitions, are prominent in many natural systems and found in many aspects of music and acoustics (Milne 2018). In music, these are found in the undulating pulsation of a low frequency oscillator for vibrato or tremolo effects, they can outline the rise and fall of melodic contours and can represent the dynamic pulsation of metric rhythms.

In live coding contexts, Periodic motion is a common pattern, especially in the Extempore environment where the cosr function is frequently used for various musical purposes. The LCT has a Periodic abstraction that generates output following a stepped cosine shape and is controlled by the Tempo object. Arguments and inputs such as the number of steps in a cycle, the centre of the range, and amplitude deviation can be specified. Figure 2 shows an example patch where the Periodic object controls the rise and fall of the cutoff frequency on a filtered noise tone. Arguments to the Periodic object in this patch are expressed as MIDI pitch values, which are then converted to frequencies and sent through a smoothing process to the filter cutoff.

7.3. Cycles

Looping is a common pattern in musical live coding and electronic and popular music genres. There are a wide range of ways to implement cyclic patterns and variations in complexity, from simple looping phrases in ixi lang, to nested for-loops in Sonic Pi, Extempore's use of temporal recursions, and Tidal Cycle's extensive cycle manipulation options.

The Cycle abstraction in the LCT is a simple approach to sequence looping. It takes a list of values and outputs them in order, and loops back to the beginning when it reaches the end of the list. Lists of values are passed in with a Pd Message object or they can be returned from a Pd Array.

A common feature of cycling operations in live coding is the ability to separate note parameters,

https://doi.org/10.1017/S1355771823000365 Published online by Cambridge University Press

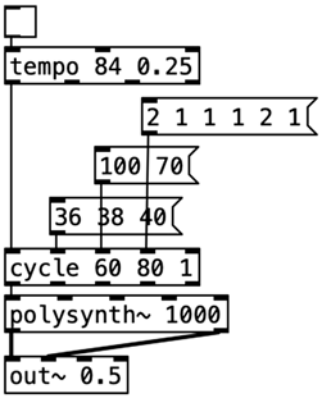

Figure 3. The LCT Cycle abstraction accepts lists of values for pitch, dynamic and duration.

which are typically combined in Western notation. In the LCT, different lists can be used for pitch, dynamic and duration, allowing them to cycle independently. This approach is demonstrated in Figure 3, where there are three pitches, two dynamic values and six duration values, which can cycle out of phase within the same Cycle abstraction.

The separation of note attribute parameters also allows data from one dimension to be easily varied while maintaining consistency elsewhere. For example, changing the pitch sequence while maintaining the rhythm. These techniques support an evolutionary development of the music without the need to re-specify every event each time.

7.4. Indeterminacy

The use of pseudo random numbers and probabilistic distributions is common in algorithmic and generative music and other media, as it provides variety and unpredictability that is often highly valued in this style of music. This idea of indeterminacy has a long history in music composition, from musical dice games to the use of stochastic procedures by Iannis Xenakis (1971). Most programming languages have a built-in random function, which is also inherited by live coding mini-languages. The LCT is no exception, it has the Rand abstraction that builds on Pd's Random object and includes three different random outputs: a linear distribution, a gaussian distribution and a random walk. These were selected as they appear most frequently in literature on algorithmic composition (Dodge and Jerse 1997; Taube 2004; Gifford, Brown and Davidson 2013). Additionally, the LCT has the RandSeq and RandChord abstractions, which generate a linear random sequence or set of a specified size and value range. The RandSeq output can be passed to a Cycle abstraction for looping playback, and the RandChord output can be passed to the PolySynth~ or Midi abstraction for playback.

The patch in Figure 4 shows an extensive use of the LCT's Rand abstraction applied to many parameters and the use of RandSeq for the pitch sequence. The sound source is a noise modulated sawtooth waveform made of Pd objects.

7.5. Branching

Building on the use of probabilistic parameter choices, we now focus on probabilistic transitions between processes. Branching, in which decisions are made (even at random) to move in one direction or another, is a powerful organising pattern in algorithmic systems. This probabilistic branching process is the foundation for Markov models, state transition networks and timed Petri nets. These structures are often used to determine the statistical character of a work or section of it, and the probabilities used can be derived from the analysis of existing works, providing a compact way to embed those tendencies.

The LCT includes the Direct abstraction to support the creation of these probabilistic branching models. The Direct abstraction is a simple, four-way branching node. It takes a signal (value) to be redirected in one inlet and the branch number (outlet) to send it to in the other inlet. Typically, the output is processed through some decision system to choose the next branch value, which is passed back in for the next event. This recursive process continues. An example of this can be seen in Figure 5, where an incrementing counter is fed into the left inlet of the Direct abstraction and the outlet number is provided in the right inlet. From each output, a Bang message is sent to trigger a particular percussive sound, and as input for an algorithm that decides the next branch number.

8. EVALUATION

A series of studies were conducted to evaluate the LCT, and the functions written for it that articulate the musical patterns outlined in previous sections. These studies were designed to ensure that the LCT and its functions were suitable for their intended purpose. The studies covered a range of musical genres commonly used in live coding and each focused on different library abstractions (functions). Each study involved a period of testing and composing, followed by a from-scratch narrated live coding 'performance'. Videos of each study are available online.[5] A summary of the studies and the LCT features they were designed to test is outlined in Table 1.

[5]https://youtu.be/070_kvkYp6o.

https://doi.org/10.1017/S1355771823000365 Published online by Cambridge University Press

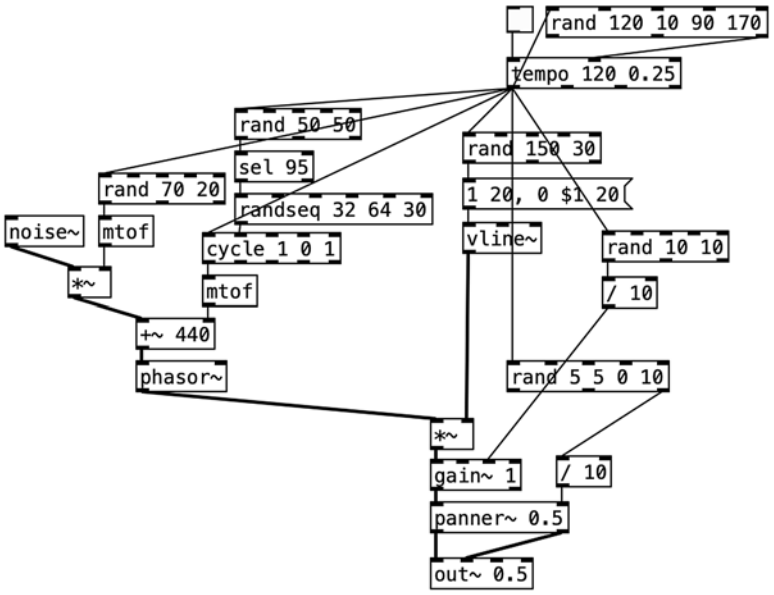

Figure 4. An LCT patch that makes extensive use of the random-based abstractions to create a chaotic soundscape.

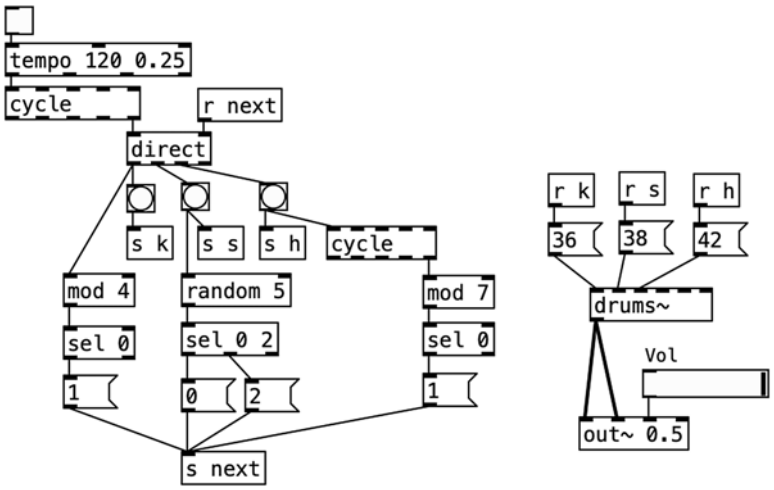

Figure 5. The Branch abstraction is at the centre of a transition network that determines a rhythm pattern playing three synthesised drum kit sounds.

The method for this evaluation involved a combination of autoethnographic reflection and user testing (Ellis, Adams, and Bochner 2011). Specific evaluation criteria were used, including patch robustness, succinctness, consistency and flexibility. My reflections on these trials are also informed by my experience live coding since 2005. Although the process is inherently subjective, the technical corrections and feature modifications made as a result were as much about resolving systemic issues and ensuring interoperability as they were about personal musical or coding preferences. The availability of the videos for each study walk-through increases the integrity of the investigations because it allows the reader to review the data and check the reported results against these. During the four-month evaluation period, over 160 changes were made to the library, as documented in the Github repository.

8.1. Supporting design patterns within Pd

For consistency with Pd naming conventions, a tilde (~) was added to the name of all audio objects. This applied to objects used for sound synthesis/playback, signal effects and audio output, including synth~, fm~, sampler~, echo~, panner~ and out~.

https://doi.org/10.1017/S1355771823000365 Published online by Cambridge University Press

Table 1. A table outline of the live coding studies used for evaluation and refinement of the LCT

Study	Musical genre	Abstractions featured	Description
1	Minimalism	tempo, cycle, midi	Steve Reich's *Piano Phase*
2	Soft rock	periodic, synth~, fm~, drums~, quant, rand, defer, gate and out~	Mimicking elements of a performance by Andrew Sorensen
3	Noise	panner~ and echo~	Controlling 'raw' Pd audio objects
4	Breakbeat	sample~, polysample~ and randchord	Exploring chord generation and sample playback options
5	Ambient	ramp and rand	Modulating arhythmic parts with extensive use of randomness
6	Algo dance	randseq	Randomised sample sequences often used in Algoraves
7	Solo piano	direct and midi	Bartók inspired using a Disklavier piano
8	Algo EDM	periodic and cycle	Sample-based audio and Hydra graphics
9	Synth groove	euclid and periodic	MIDI synth audio and Hydra graphics
10	Pop chords	randchord, cycle and euclid	LCT instruments and Hydra graphics
11	Noise feedback	rand and ramp	Modulating Pd delay lines and Hydra graphics

Like all programming languages, objects in Pd, and thus the LCT, can use a variety of data types for arguments in both input and output. During the creation of the studies, it was determined that abstractions would be more flexible if they could also accept additional types to those initially assumed, a technique known as method overloading. For example, the Cycle abstraction can handle input from bangs, floats, integers and lists of numbers, and the Tempo abstraction can output both bangs and a pulse counter. To support greater interoperability and robustness, several LCT abstractions were modified to overload inputs and provide additional outputs, such as the Polysynth~ abstraction, which can accept either a single pitch or a list of pitches (a chord), and the Trans abstraction, which can transpose either a single pitch or a list of pitches (such as a chord or sequence) and output the corresponding data type.

The studies utilised different sound sources such as the LCT's built-in synths and sampler, 'vanilla' Pd audio processes, and MIDI output to synthesisers and a MIDI-controlled piano. This showcases how the LCT can utilise Pd's existing features for different input–output modalities, and how it can offer a wide range of timbre and genre flexibility. However, there are still more capabilities of Pd that were not explored, including the use of MIDI controller input for adjusting parameters, audio input for signal processing and collaboration via network communication using Open Sound Control.

In the final four studies the Hydra web-based graphics environment[6] was used to provide visual interest to the performance screen. Hydra enables compositing of a Pd window within the generative graphics, ensuring the audience can see both the live patch and the graphics. The Hydra-MIDI script[7] was used to enable MIDI communications from Pd to Hydra, allowing synchronisation between changes in the sound and image.

8.2. Supporting musical patterns

The objects in the LCT that implemented common design patterns, as described in section 4, were found to be highly adaptable across a wide range of genres in the studies. The Cycle, Periodic and Rand functions were heavily utilised and their functionality was refined during the evaluation stage, but not fundamentally changed. The complementarity of deterministic and repetitive processes such as Tempo, Cycle and Periodic with stochastic functions such as Rand, RandSeq, RandChord and Gate once again demonstrates the well-established maxim that aesthetic interest relies on a

[6]https://hydra.ojack.xyz/.

[7]https://github.com/arnoson/hydra-midi.

https://doi.org/10.1017/S1355771823000365 Published online by Cambridge University Press

balance of order and chaos. This finding reinforces both the ubiquity of these processes in the literature on algorithmic music, and their prevalence in most (perhaps all) live coding mini-languages.

Several objects were added as it became clear through composing the studies that certain functions were repeatedly required. These included:

- randchord – generates and stores a randomised set of pitches within a range and scale;
- gate – probabilistically let data through so it acts as a stochastic filter, limiting the data stream;
- gate~ – probabilistically let an audio signal through;
- polysample~ – an eight-voice version of the sample playback object;
- modrand – triggers one of four outlets at random at each modulo cycle.

The inclusion of Randchord in the LCT is a logical addition as it complements the functionality of Randseq. The use of a probabilistic Gate abstraction, which is more versatile than Pd's Spigot object and which can only be open or closed, is a nod to the use of more advanced data filters, sometimes referred to as sieves, by legendary algorithmic composers such as Gottfried Michael Koenig and Iannis Xenakis. The Polysample~ abstraction is consistent with the other poly versions of 'synths' in the LCT. The Modrand abstraction is used to trigger a random choice at regular metric intervals, such as on the downbeat of a measure. It serves as a convenience function that encapsulates what would typically be a Modulo → Random → Select network in Pd. During this evaluation process, examples and help files were updated for new and modified objects.

8.3. Limitations and further development

During the evaluation process, certain limitations of the LCT were identified that could not be addressed without significant changes. These limitations fell into two categories: procedural and interface restrictions.

Currently, the LCT does not make extensive use of data storage and data processing. Some areas where it does include the Randseq, Trans and Sample abstractions. Randseq stores a sequence of values, Trans does some advanced list processing to enable diatonic transposition, and Sample stores audio data and supports segmentation. However, these are only limited extensions beyond what is already available in Pd for list processing or audio data capture and editing. There are other live coding languages that better facilitate sequence list manipulations and a few that support live audio processing. The studies described here did not require these features, but there are musical practices that could benefit from them, so they remain options for future expansion of the LCT.

During the evaluation, some interface limitations became apparent, but these issues were not unexpected. When creating the series of studies, the number of inlets and outlets in LCT abstractions meant that it took time to become familiar with them. Pd does not include inlet/outlet names and the inlet or outlet does not indicate the required data type(s), other than the distinction between data and audio connections. At present, the goal for the LCT is to keep the number of inlets and outlets limited, but the experience of developing the studies showed that even with this, things can still get confusing. A feature such as pop-up contextual cues would be useful but is currently not feasible.

The mouse-based connecting of nodes in the Pd patch graphs is an inherent limitation in the interface, though in the studies, the extensive use of auto-connecting new nodes to selected ones was found to be very helpful in speeding up patch building. The speed of patch creation can be improved by using fewer, higher-level LCT functions; however, the experience of composing the studies reinforced the importance of prioritising elegant algorithms. The search for elegant algorithms is an ongoing challenge for all live coding practices and was only partially successful in the evaluation studies. Additionally, when creating the evaluation documentation videos, the live narration process was found to be a hindrance to rapid patch development as it required time to explain what was happening and increased cognitive load.

9. CONCLUSION

Design patterns, which can be described using algorithms, are widely used in various fields. Given that algorithmic music lends itself to pattern-based algorithmic description, it is not surprising that music performance blossomed as a live coding practice. As a result, multiple mini-languages have been developed to support live coding in music performance. These languages often share similar design patterns that are useful for music-making in general or live coding performance specifically.

Design patterns that represent abstractions of commonly used processes and techniques useful for live coding have been discussed. It was argued that these are not only a structural convenience in composition or performance, but they also shape the way the live coder interacts with the system by providing constraints and opportunities. Abstractions, through their presence in mini-languages, provide a representation of the world for the live coder to interact and communicate musically. Indeed, the efficiency of the constraining tendency of

https://doi.org/10.1017/S1355771823000365 Published online by Cambridge University Press

technologies, such as design patterns, is matched by a danger that it obscures other potentials. In Heideggerian terms, 'the extreme danger [of technologies] lies in the holding sway of Enframing' (Heidegger 1977: 29). Being aware of these tendencies can hopefully allow us to use them productively, yet avoid unintended consequences.

This article explored the history and application of design patterns in various fields, including algorithmic music. Based on reviews of literature and practice, a typology of significant live coding design patterns was presented. These included timing control, periodicity, cycling or repetition, indeterminacy and branching or decision networks.

The LCT for Pd was presented as a mini-language that enhances Pd's audio and music capabilities for live coding. The design patterns identified for live coding and their inclusion in the LCT were outlined. The effectiveness of the LCT was demonstrated through testing on various live coding studies spanning different musical genres. These showcased the versatility of the identified patterns and the LCT's implementation of them in a live coding context. Some limitations of the LCT and/or Pd for live coding have been highlighted. The LCT is an open-source project, and so it can continue to be improved through user feedback and developer contributions. Visual programming languages such as Pd have not been widely adopted in the live coding community, possibly due to a lack of support for relevant design patterns. The LCT aims to make it easier for Pd users to get into live coding, and for live coders to use Pd as a platform for their practice.

The design of live coding mini-languages plays a critical role in shaping live coding practices. Including algorithmic descriptions of common musical conventions and techniques in live coding mini-languages, such as the LCT, should enable more efficient and successful live coding performances.

REFERENCES

Aaron, S. and Blackwell, A. F. 2013. From Sonic Pi to Overtone: Creative Musical Experiences with Domain-Specific and Functional Languages. *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modelling & Design*. Boston, MA: ACM, 35–46.

Alexander, C. 1979. *A Timeless Way of Building*. New York: Oxford University Press.

Berry, W. 1976. *Structural Functions in Music*. Englewood Cliffs, NJ: Prentice Hall.

Brown, A. R. 2016. Performing with the Other: The Relationship of Musician and Machine in Live Coding. *International Journal of Performance Arts and Digital Media* **12**(2): 179–86.

Collins, N. 2003. Generative Music and Laptop Performance. *Contemporary Music Review* **22**(4): 67–79.

Collins, N. 2018. Origins of Algorithmic Thinking in Music. In R. T. Dean and A. McLean (eds.) *The Oxford Handbook of Algorithmic Music*. New York: Oxford University Press, 67–78.

Dannenberg, R. B. and Bencina, R. 2005. Design Patterns for Real-Time Computer Music Systems. *ICMC 2005 Workshop on Real Time Systems Concepts for Computer Music,* Barcelona, Spain.

De Souza, J. 2017. *Music at Hand: Instruments, Bodies, and Cognition*. New York: Oxford University Press.

Dodge, C. and Jerse, T. A. 1997. *Computer Music*. New York: Schirmer Books.

Ellis, C., Adams, T. E. and Bochner, A. P. 2011. Autoethnography: An Overview. *Historical Social Research/Historische Sozialforschung* **12**(1): 273–90.

Gamma, E., Helm, R., Johnson, R., Johnson, R. E. and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Indianapolis, IN: Addison-Wesley.

Gifford, T., Brown, A. R. and Davidson, R. 2013. Amplifying Compositional Intelligence: Creating with a Psychologically-Inspired Generative Music System. *Proceedings of the International Computer Music Conference*. Perth: ICMA, 357–60.

Griffiths, D. 2007. *Fluxus*. Collaboration and learning through live coding, report from Dagstuhl Seminar No. 13382. Schloss Dagstuhl.

Heidegger, M. 1977. *The Question Concerning Technology and Other Essays*. New York: Harper & Row.

Kirkbride, R. 2016. Foxdot: Live Coding with Python and Supercollider. *Proceedings of the International Conference on Live Interfaces*, Brighton, UK, 194–8.

Lansky, P. 1990. The Architecture and Musical Logic of Cmix. *Proceedings of the International Computer Music Conference*, Glasgow, Scotland.

Loy, G. and Abbott, C. 1985. Programming Languages for Computer Music Synthesis, Performance, and Composition. *ACM Computing Surveys* **17**(2): 235–65.

Magnusson, T. 2011. The ixi lang: A Supercollider Parasite for Live Coding. *Proceedings of the International Conference on Music and Computers*, Huddersfield, UK.

Magnusson, T. and McLean, A. 2018. Performing with Patterns of Time. In A. McLean and R. T. Dean (eds.), *The Oxford Handbook of Algorithmic Music*. New York: Oxford University Press, 245–66.

McCartney, J. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal* **26**(4): 61–8.

McLean, A. 2014. Making Programming Languages to Dance to: Live Coding with Tidal. *Proceedings of the 2nd ACM SIGPLAN International Workshop on FUNCTIONAL ART, Music, Modeling & Design*, Gothenburg, Sweden, 63–70.

McPherson, A. and Tahıroğlu, K. 2020. Idiomatic Patterns and Aesthetic Influence in Computer Music Languages. *Organised Sound* **25**(1): 53–63.

Milne, A. 2018. Linking Sonic Aesthetics with Mathematical Theories. In A. McLean and R. T. Dean (eds.), *The Oxford Handbook of Algorithmic Music*. New York: Oxford University Press, 155–80.

Norman, D. A. 1988. *The Psychology of Everyday Things*. New York: Basic Books.

Puckette, M. 1996. Pure Data: Another Integrated Computer Music Environment. *Proceedings*, *Second Intercollege Computer Music Concerts*. Tachikawa: Kunitachi College of Music, 37–41.

Puckette, M. 2004. A Divide between 'Compositional' and 'Performative' Aspects of Pd. In *Proceedings of the First International Pd Convention*. Graz, Austria: PdCon.

Riehle, D. 1997. Composite Design Patterns. *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York: ACM, 218–28.

Roberts, C. and Kuchera-Morin, J. 2011. Gibber: Live Coding Audio in the Browser. *Proceedings of the International Computer Music Conference*. Huddersfield, UK: International Computer Music Association.

Roberts, C. and Wakefield, G. 2018. Tensions and Techniques in Live Coding Performance. In R. T. Dean and A. McLean (eds.), *The Oxford Handbook of Algorithmic Music*. New York: Oxford University Press, 293–319.

Roberts, C., Wakefield, G., Wright, M. and Kuchera-Morin, J. 2015. Designing Musical Instruments for the Browser. *Computer Music Journal* **39**(1): 27–40.

Sayer, T. 2016. Cognitive Load and Live Coding: A Comparison with Improvisation Using Traditional Instruments. *International Journal of Performance Arts and Digital Media* **12**(2): 129–38.

Sorensen, A. and Brown, A. R. 2007. aa-cell in Practice: An Approach to Musical Live Coding. *Proceedings of the International Computer Music Conference*. Copenhagen: ICMA, 292–9.

Sorensen, A. and Gardner, H. 2010. Programming with Time: Cyber-Physical Programming with Impromptu. *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. New York: ACM, 822–34.

Sorensen, A., Swift, B. and Gardner, H. 2014. The Many Meanings of Live Coding. *Computer Music Journal* **38**(1): 65–76.

Sorensen, A. C. 2018. *Extempore: The Design, Implementation, and Application of a Cyber-Physical Programming Language*. PhD thesis, Australian National University, Canberra, Australia.

Spiegel, L. 1981. Manipulations of Musical Patterns. *Proceedings of the Symposium on Small Computers and the Arts*. Los Angeles, CA: IEEE, 19–22.

Sudnow, D. 1978. *Ways of the Hand: The Organization of Improvised Conduct*. Cambridge, MA: MIT Press.

Taube, H. 2004. *Notes from the Metalevel: Introduction to Algorithmic Music Composition*. London: Taylor & Francis.

Temperley, D. 2001. *The Cognition of Basic Musical Structures*. Cambridge, MA: MIT Press.

Wang, G. and Cook, P. R. 2003. ChucK: A Concurrent, On-the-fly, Audio Programming Language. *Proceedings of the 2003 International Computer Music Conference*. Singapore: ICMA, 219–26.

Whitehead, A. N. and Price, L. 1954. *Dialogues of AN Whitehead, as Recorded by Lucien Price*. Biddeford, ME: David R. Godine.

Wishart, T. 1985. *On Sonic Art*. New York: Imagineering Press.

Xenakis, I. 1971. *Formalized Music: Thought and Mathematics in Composition*. Bloomington, IN: Indiana University Press.