Context: A Modular Approach to Sequencing

Liam Goodacre
F1 802 Success Towers, Panchavati, Pashan, Pune, MH 411008, India

Abstract

Context is a modular sequencer for Pure Data which reimagines composition in the form of a network. A single Context unit functions as a step sequencer, a selector and an embedding environment for samples. Multiple units are interconnected to form Context networks. Users build networks to create their own sequencing environments which can be highly specialized, incorporating algorithmic, stochastic, structural, reactive and interactive composition.

Introduction

The modular approach to audio synthesis has been popular since the pioneering Moog and Buchla systems of the 1960s (Pinch and Trocco 2009), but has proven to be equally successful with computers (Kreidler 2009). In both its analog and digital application, the basic idea of modularity is to separate a larger system into smaller components (oscillators, signal processors, etc.), so that the composer can connect them together and control them in a large number of ways. Sequencers play an important role in modular systems (Jenkins 2009). A sequencer is a hardware or software device that sends signals or messages in a timed, orderly fashion, essentially determining for electronic music *what happens, when* (Brandt and Dannenberg 1999). It is common to find sequencers controlling other units within modular environments, for example (Rothstein 1995), but rarely do we see modular environments devoted primarily to sequencers.

Context is a sequencer that is designed to be truly and fully modular. That is, Context is a sequencer which may be perpetually duplicated and interconnected within its software hosting environment. Context is built in Pure Data (Pd) (Puckette 1996) as an abstraction and is used to control other Pd objects and instruments by sending messages. By instantiating multiple Context units and interconnecting them, the user designs a special sort of sequencing environment called a *Context network*, which can store and perform compositions of any size. This paper presents the Context sequencer and explores the potential of modular sequencing environments. The first section describes the GUI and language of a Context unit, showing how it borrows techniques from traditional linear and non-linear composition. The second section focuses on the Context network, looking at how multiple Contexts function when they are connected together. The third section explores some possible applications for Context, and how it might prove useful for composers and performers. Emphasis is given to algorithmic and metacreative methods, where the software is used to generate its own musical expression.

Description of the Context unit

Context appears as a single object in a larger Pd patch (hence it is often referred to in the singular). A Context unit, *fig 1*, has a simple GUI layout which will seem familiar to anyone with experience in electronic music production.

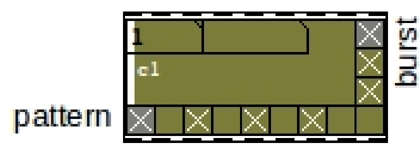

Figure 1: A single Context unit

The bottom contains a row of on/off toggles boxes (collectively called the *Pattern*) which play a selected pattern linearly, like a normal step sequencer. Each toggle corresponds to a term from a database (not depicted), allowing custom messages to be sent during playback, similar to *note*∼ for Max/MSP (Resch 2013). The Context GUI is easily re-sizable using the mouse, so the Pattern can be of any length. The Pattern length can be thought of as the number of beats in a bar, while the number box in the top-left corner of the unit sets the duration of the playback in seconds. When Context is started, a cursor moves across the screen from left to right in the given time and the selected pattern is played sequentially, a process called the *Context cycle*.

The step sequencer is a standard tool for linear playback, and in this sense Context's Pattern, while being useful, limits composition to a deterministic framework (Cioslowski 2011). However, Context has a number of generative and algorithmic capabilities which place it beyond the linear realm. Two of these, the *Burst* and the *Output Language*, are described here.

The Burst

The vertical row of toggles on the right side of Context's GUI are known as the *Burst*. As with the Pattern, each Burst toggle corresponds to a database term and can trigger other events in the network, but instead of firing in a sequence, they all fire together at the end of the cycle. More important than *when* the toggles fire is *which* toggles fire. This assignment is randomizable, and the user can control the distribution, position and quantity of toggles that fire. This is depicted by the Bell curve in *fig 2*. (The image is rotated 90 ° in order to correctly render the graph. Imagine that the width and position of the curve can be set. The graph represents the likelihood of a given toggle being selected.)

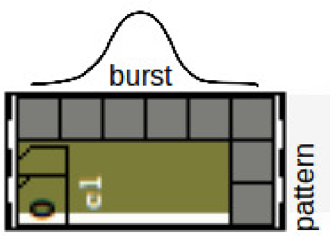

Figure 2: Distribution of Burst toggle selection

The distinction between the Pattern and the Burst is central to the Context sequencer, with the Pattern (x-axis) representing a series distributed through time and the Burst (y-axis) a series distributed through probability. The Burst is primarily used as a selector, but can also generate chords.

Output Language

Context has a database which stores and sends custom messages. These messages are subject to a Term Rewrite system (Nierhaus 2009), known as Context's *Output Language*, which allows the user to build mathematical formulas. The syntax is similar to Pd's native `[expr]` in that it performs arithmetic, respecting bracket order (Yadegari 2003), but also substitutes a set of variables with state specific values. Variables include random numbers, position and state information; expressions include arithmetic, musical scale mapping and custom functions processed through other Pd objects. For example, the message `10 - (? @)` would return a value of 10 minus a random number (?) whose upper limit is determined by Context's position on the canvas (@). Such formulas can be extended indefinitely, allowing for a great deal of complexity and control.

Input Language

Context has its own Domain Specific Language to control its parameters. Some commands alter Context's state (i.e. its dimensions, toggle allocations and message database) while others instruct it to perform certain tasks (i.e. `start` and `stop`). The syntax requires commands, delimited by colons, followed by command-specific arguments, followed optionally by further commands. For example, `:X 9 :x 1 3 4` sets the Context's x-axis to 9 units wide and opens Pattern toggles 1 3 and 4. Input Language commands are sent to Context from within the patch, or assigned as creation arguments. This allows for persistent behavior, as the Context state is automatically written to the creation argument whenever the patch is saved. The main GUI reflects only some of the state, not all of it. Thus, Context can be seen primarily as a text based system, with GUI access to its essential elements.

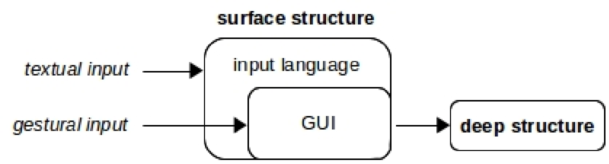

Figure 3: Relationship between Input Language and GUI

Embedding

Context's canvas (the green area in *fig 1*) is an embeddable timeline for arrays, markers and other special objects. Embedded objects can be moved and resized around the canvas and play linearly with the cursor as part of the cycle. In this way, Context is used for sample playback and can even emulate some basic DAW behavior (see for example Ableton's "Arrangement overview", (Margulies 2014)).

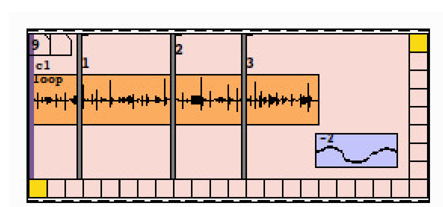

Figure 4: A Context with embedded arrays and markers

Embedded objects are stored in a part of the patch known as the *Overlay*. Any Pd object can be created in the Overlay, allowing the user to modify or hack the normal behavior of Context. The Input Language has an embedded interpreter to read and write the Overlay file to the creation argument, so that Overlay modifications may be saved.

Context Networks

A single Context has been seen to consist of a linear sequencer, a formula builder, a random selector, and an embeddable timeline, making it a useful though hardly original device. But Context is designed to exist in a network, not in isolation. In a Context network, Context agents communicate with each other by receiving, processing and sending information asynchronously, resulting in complex heterarchical behavior. There are three main ways of connecting Contexts together: *connections*, *commands* and *rules*, each being a subset of the last. Being the simplest and most important, connections are treated first. (It should be noted that one Context can be connected to itself with any of these methods just as easily as it can be connected to another.)

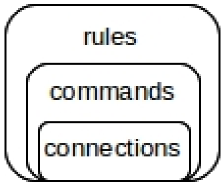

Figure 5: Ways of connecting Contexts

Connections

Connections are the familiar Pd cables used to connect objects together via native inlets and outlets. Context has one inlet and one outlet for every toggle along its x-axis. Inlets are used to start a Context and are all essentially the same; outlets are synched to the pattern toggles. When a connection is made from x to y, a basic rule is formed: *Context y will start its cycle when toggle x fires*. In *fig 6*, Context 1 cycles for one second and then starts Context 2, which cycles and then starts Context 1, resulting in a loop. Turning a toggle off suspends the rule and terminates the loop.

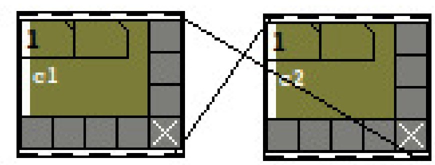

Figure 6: two Contexts connected to form a basic network

In *fig 6* there are two Contexts, but any number may be connected together in arbitrarily complex ways. The most immediate result of connecting multiple Contexts into a network is the fragmentation of a composition into its elements. The user can assign a musical phrase to a particular Context and then relate it to other phrases in a "physical" way, patching in new sequences at any time. Phrases can be any length, from a single note to an entire piece of music, and connections can be broken as easily as they are made. This makes composition very accessible. Context networks have a very low "Premature Commitment" (Bellingham, Mulholland, and Holland 2014), as users are not "forced to decide on implementation detail before they would otherwise be ready to" (Bellingham, Mulholland, and Holland 2014). [1]

Commands

Connections allow one Context to trigger another by sending virtual `start` messages. But it is also possible to send commands other than `start` to control Context in different ways. This happens when an outgoing message is directed towards another Context and contains a command from the Input Language, such as `:x 4` (open the fourth x-axis toggle) or `:d 2` (set the cycle time to 2 seconds). In this case, the events that Context sequences are not musical events, triggering some sound, but *meta-events* aimed at altering the network. Since meta-events can be incorporated into the composition itself, music generated with Context can evolve in a structured way over time. This evolution can unfold deterministically or randomly, at the user's discretion. This matches Eigenfeldt's description of an "Adaptive System" (Eigenfeldt et al. 2014), where "agents interact and influence one another and behave in different ways over time due to their own evolution" (Bellingham, Mulholland, and Holland 2014). [2]

Rules

Commands allow one Context to alter another using the Input Language. Rules perform the same type of alteration, but only given the trigger of some predefined condition. A rule is comprised of two elements: a *condition* and a *consequence*. A consequence is any command from the Input Language, while a condition is a Boolean expression which is evaluated against the current Context state. For example, a rule could demand that *if* more than two toggles are open, *then* the cycle time will be increased by 1 second, or that *if* the note C# is played, *then* toggle 1 will open. Rules are determined textually, offering a high amount of complexity and precision.

Rules allow effects (consequences) to propagate over a network according to the state of its agents. In this way, a Context network resembles the Cellular Automata (CA) approach, where behavior emerges from a grid of cells, each of which depends on the conditions of adjacent cells to determine its own state (Nierhaus 2009). Miranda has applied CA to composition in his project CAMUS and CAMUS 3D (Miranda, McAlpine, and Hoggar 1999).

> "The composition process is modeled on pattern propagation… As the composition progresses, the patterns are subjected to certain transformations… according to the formal structures that the composer has chosen for the work". (ibid).

This kind of composition is similarly feasible in Context, although Context differs from CA in two important ways:

1. Connections between Context's 'cells' are programmable in any way, rather than being confined to a grid;
2. Each 'cell' can take its own specific rule.

In this way, a Context network is seen to cover a larger domain than CA, as networks need not be homogeneous and their cells can have individual identity.

Applications

Sequencing environments are well developed Context networks which may house an entire piece of music. Rather than working within the confines of a DAW or other large software suite, the Context composer designs her own sequencing environment according to her own taste and the

[1] In fact, it might be found that Context networks satisfy Bellingham's challenge for "structure-aware composition software" (Bellingham, Mulholland, and Holland 2014).

[2] This should not be confused with the evolution of genetic algorithms and neural nets, since Context has no learning algorithm. A Context network may modify itself in a systematic way, but it cannot be *trained* (Lischka 1991). Learning algorithms are one opportunity for the future development of Context.

style of the composition. This section describes some possible sequencing environments which pertain to metacreative music. Explanations accompany 'pseudo-patch' diagrams which could easily be followed to construct functioning Context networks.

Markov Chains

If a connection is made from a Burst toggle, the network may become stochastic. The Pattern plays out as normal, but when the Burst toggles fire at the end of the cycle, the outcome is uncertain. By this property a Context network can function as a Markov chain (Miranda, McAlpine, and Hoggar 1999), as in *fig 7*, where the Burst settings determine the probabilities and the structure of the network determines the order [3].

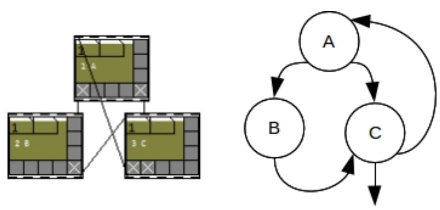

Figure 7: A Markov chain represented as a Context network, and as a Directed Graph. Here, the x-axis represents the Burst.

Markov chains are especially accessible in Context since the network closely resembles the Directed Graph diagrams that are often used to illustrate them (compare this to the Transitions Table approach taken i.e. in *Jam Factory* (Zicarelli 1987)). Loy states that "Directed graphs embody [the] sense of place and transition" within Markov chains (Loy 2006). Context networks are similarly intuitive to work with, as the path is directly depicted and controllable by simply opening and closing connections. In its representation of Markov chains, Context will be seen to resemble *Nodal* (developed by Monash University), a software which "uses spatial, directed graphs that are traversed in real-time by one or more state-based agents", giving users "the ability to structure and control processes in a compositional sense" (McCormack et al. 2007). In fact, Nodal's mapping of Markov chains is more explicit than Context's, given the intuitiveness of its GUI (*viscosity* in (Bellingham, Mulholland, and Holland 2014)). However, Context has one inherent advantage, in that the nodes need not be single notes. They are musical phrases of any length, as determined by the Pattern.

Reactive Systems

Systems which respond in a spontaneous way to user input ("Generative systems" in (Eigenfeldt et al. 2013)) are well within the grasp of Context networks. To achieve diverse behavior, input from a MIDI source is routed in Pd and sent to various different starting points in a network. Alternatively, Context rules can be written to distinguish one input from another. From the various parallel starting points, the Context network take the form of a series of terminating Markov chains which respond to the input in various ways. The user chooses whether or not the different channels converge to a singles point, or remain distinct from each other. [4]

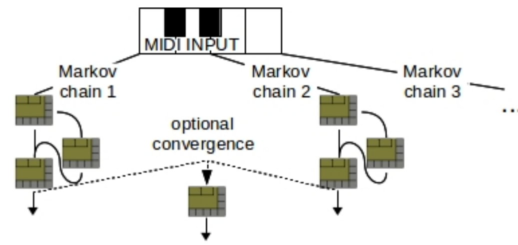

Figure 8: Sketch of a reactive Context network

Interactive Systems

With interactive music programs,

> "The computer responds to the performer and the performer reacts to the computer, and the music takes its form through that mutually influential, interactive relationship" (Chadabe 1983).

This can be achieved in Context by designing extended networks which cycle and evolve indefinitely of their own accord. Interaction is then a sort of dialog, with both composer and computer suggesting new ideas (Goodacre 2016). A Context network structured in an organized hierarchy is shown in *fig 9*. At the top is one Context which acts as a master clock for the whole system and sets the tempo. Next is a Context which defines a structure, say `A B A C` (if a cluster of Contexts is used here, the structure may be multi-layered). The structure is sent to various parallel channels, where Patterns and Markov chains are used to create musical sequences based on `A B` and `C` (as described in Reactive Systems). Before this, a gate distributes the messages, deciding which instruments are active and which are silent. In this configuration, the pattern, structure and channel assignment of the music can all be defined by altering toggles, or can evolve autonomously with Context meta-events. The composer must decide how to respond to the new themes and structures that develop. (A handy *Undo* feature can be built into the Overlay which allows the user to quickly cancel any unwanted changes).

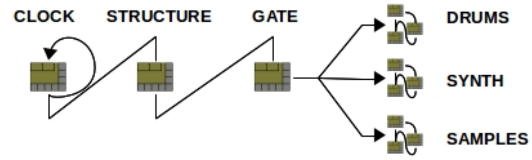

Figure 9: Outline for an interactive Context network.

[3] Unfortunately, Pd does not allow for connections on the side of an object. In order to allow for Burst connections, a special mode in Context flips the position of the Pattern and Burst along the axes. Thus, in *fig 7*, the Pattern lies on the y-axis, and the Burst on the x-axis, contrary to the order described in *Description of the Context unit* above.

[4] Another relevant feature is the *persistence variable* in the Output Language which performs arithmetic on incoming data, allowing for melodies which play relative to a given starting note.

Collaborative Systems

Context networks are highly decentralized systems, and there is no need for one Context to hold absolute authority (as in *fig 9*). More organic, "messy" systems can be developed where separate channels are not distinct and may bleed into each other so that the music is less predictable. It is also possible for multiple users to control the same network collaboratively. Networks could be specially made to designate control over certain elements to different users, and could even incorporate audience input as a form of *Distributed Performance* (Swift et al. 2009). This approach differs from the contemporary *Musebots* (Eigenfeldt et al. 2014), *NetPd* (Haefeli 2013) and *Ableton Link* (Brinkmann 2016) in that the sequencing and sonification elements of the patch would not be distributed between computers, rather there would be one network with multiple access points. The implementation of a collaborative network has yet to be worked out, but the point remains that Context is a promising tool for building custom collaborative and distributed sequencers.

Conclusions

It has been shown how Context successfully applies the modular paradigm to sequencing, and how it marries linear and non-linear elements in a single object. Individual Contexts functions as step sequencers, sample players, random selectors, and a language for generative music (in fact, any / all of these at once). Collectively, Context builds networks to house complex compositions, incorporating algorithms and stochasticity in any way. By programming multiple Contexts and connecting them together, the user always has the freedom to decide how Context should operate and what type of network to build. Eigenfeldt asks us to imagine

> "a continuum between traditional praxis or performance tools, and metacreations. At one end, the software simply acts as a tool to be manipulated by the creator... On the other extreme, pure metacreations are autonomous and proactive in their creative choices, and require no human intervention once running" (Eigenfeldt et al. 2014).

Context realizes this continuum (or at least part of it) as a traversable axis. In designing a network, the user decides the extent to which the composition is generative, autonomous and non-linear. Perhaps more importantly, choosing one point on this axis does not preclude choosing another. A user can construct a network that is linear in one place and non-linear in others, and jump between these various paradigms without any restriction. It is hoped that Context will make algorithmic techniques more accessible to composers and offer new possibilities for performance.

Acknowledgements

Matthew Barber, Julian Brooks, Tristan Chambers, Joe Deken, Sunu Engineer, Andy Farnell, Alexander Harker, Miller Puckette, Katja Vetter, the organizers of PDCon16, and countless other members of the PD community.

References

Bellingham, M.; Mulholland, P.; and Holland, S. 2014. *An analysis of algorithmic composition interaction design with reference to cognitive dimensions*. The Open University.

Brandt, E., and Dannenberg, R. B. 1999. *Time in Distributed Real-Time Systems*. International Computer Music Association.

Brinkmann, P. 2016. *Ableton Link integration for Pure Data*. Proceedings of the 5th International Pure Data Convention.

Chadabe, J. 1983. *Interactive Composing: An Overview*. Computer Music Journal, MIT Press.

Cioslowski, J. 2011. *Generative Sound Application*. Aalborg University, Copenhagen.

Eigenfeldt, A.; Bown, O.; Pasquier, P.; and Martin, A. 2013. *Towards a Taxonomy of Musical Metacreation: Reflections on the First Musical Metacreation Weekend*. Papers from the 2013 AIIDE Workshop.

Eigenfeldt, A.; Thorogood, M.; Bizzocchi, J.; and Pasquier, P. 2014. Mediascape: Towards a video, music, and sound metacreation.

Goodacre, L. 2016. *Structure, composition and the Context sequencer*. Proceedings of the 5th International Pd Convention.

Haefeli, R. 2013. *netpd - a Collaborative Realtime Networked Music Making Environment written in Pure Data*. Proceedings for the Linux Audio Conference 2013.

Jenkins, M. 2009. *Analog Synthesizers: Understanding, Performing, Buying*. CRC Press.

Kreidler, J. 2009. *Programming electronic music in Pd*. Hofheim: Wolke Verlag.

Lischka, C. 1991. *Understanding music cognition: a connectionist view*. MIT Press.

Loy, G. 2006. *Musimathics: The Mathematical Foundations of Music (Volume 1)*. MIT Press.

Margulies, J. 2014. *Ableton Live 9 Power!: The Comprehensive Guide*. Nelson Education.

McCormack, J.; McIlwain, P.; Lane, A.; and Dorin, A. 2007. *Generative Composition with Nodal*. ECAL.

Miranda, E.; McAlpine, K.; and Hoggar, S. 1999. *Making music with algorithms: A case-study system*. Computer Music Journal, MIT Press.

Nierhaus, G. 2009. *Algorithmic composition: paradigms of automated music generation*. Springer Science and Business Media.

Pinch, T. J., and Trocco, F. 2009. *Analog days: The invention and impact of the Moog synthesizer*. Harvard University Press.

Puckette, M. 1996. *Pure Data: another integrated computer music environment.* Proceedings, Second Intercollege Proceedings, Second Intercollege Computer Music Concerts, Tachikawa, Japan.

Resch, T. 2013. *Note∼ for Max, An extension for Max/MSP for Media Arts and Music*. 2013 NIME Proceedings.

Rothstein, J. 1995. *MIDI: A comprehensive introduction*. AR Editions, Inc.

Swift, B.; Gardner, H. J.; Riddell, A. 2009. *Distributed Performance in Live Coding stuff*. Proceedings of the Australasian Computer Music Conference 2009.

Yadegari, S. 2003. *Chaotic signal synthesis with real-time control: solving differential equations in PD, MAX/MSP, and JMAX*. Proceedings of the 6th International Conference on Digital Audio Effects.

Zicarelli, D. 1987. *M and jam factory*. Computer Music Journal, MIT Press.