A Hierarchical, Modular Music Sequencer

Sean Luke
sean@cs.gmu.edu
George Mason University
Washington, DC, USA

Filippo Carnovalini
Vrije Universiteit Brussel
Brussels, Belgium
Università degli Studi di Padova
Padova, Italy
filippo.carnovalini@vub.be

ABSTRACT

Musical structure is replete with hierarchy, modularity, reuse, and variation. We are interested in how these aspects of music may be employed in a tool for composition and performance. We describe an experimental music sequencer we have developed which is meant to help musicians build complex, highly modular, hierarchical musical structures. The musician begins with basic musical elements, then groups them into larger structures to combine them in a wide variety of ways. The same elements may be reused in many contexts, and may be parameterized and customized each time, creating variation. This sequencer is meant as both a composition and performance tool. We discuss past research in hierarchy and modularity in music, detail our sequencer approach, outline ways that it provides hierarchy and modularity, and discuss how we intend to use its basic model as a jumping-off point for introducing machine learning, optimization, and experimentation into the sequencing process.

CCS CONCEPTS

• **Applied computing** → **Sound and music computing**; *Performing arts.*

KEYWORDS

Sequencer, Music Production, Music Generation, Hierarchical Representation of Music

ACM Reference Format:
Sean Luke and Filippo Carnovalini. 2024. A Hierarchical, Modular Music Sequencer. In *Audio Mostly 2024 - Explorations in Sonic Cultures (AM '24), September 18–20, 2024, Milan, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3678299.3678343

1 INTRODUCTION

Seq is a music sequencer of an unusual design, and is the result of an experiment undertaken to assist the musician by taking advantage of the hierarchical structure, modular reuse, and variation that is often found in music. This approach would allow for a more abstract representation of, development of, and direct interaction with the musical content in a way that is not dependent on the *surface level* of music, but rather on deeper relationships between elements of a piece [37].

Seq is open source, available at https://github.com/eclab/seq/ along with documentation and examples.

Music is commonly both modular and hierarchical. By *hierarchical* we mean that elements of a song often naturally group together in a tree-structured hierarchy of abstraction. By *modular* we mean that these elements — themes, motifs, and even phrases — appear repeatedly, with transformation or variation, and at different levels of abstraction in the piece. This hierarchy and modularity may be exploited in many ways by musicians. Composers will often reuse, mutate, and recombine the same elements many times in a hierarchy of modular abstraction; and as discussed later, this often forms a major element in theoretical analysis. Modularity and variation is also exploited by performers in improvisation (such as in the form of licks), or in performing music which develops in real-time (as can be found in electronic music). Similarly, generative or automatic music systems may use modularity in structure to reuse elements. Finally, modularity could be applied by artificial intelligence techniques to assist the musician in the composition or performance of music.

We built Seq in order to explore how modularity, hierarchy, and variation in musical structure might be exploited to help in composition and performance in all of the above ways. Seq is *hierarchical* in the sense that it contains sequences which themselves contain sequences, and so on. Seq is *modular* in that many parent sequences may *reuse* the same child sequences multiple times, and may apply variation or transformation to them. Furthermore, Seq can provide for real-time variation during live performance in the form of online parameterization at every level of the sequence, and even permit bulk restructuring of the song while playing.

We have also structured Seq to ultimately integrate with co-creative machine learning and optimization tools meant to collaborate with the musician in the development or performance of new music. As discussed in Section 10, we have significant previous experience in developing tools of this kind in other music contexts.

Hierarchical sequencers of this kind are very uncommon: in fact we know of only a few significant previous examples. We begin with an introduction to hierarchy and modularity in music as a motivation for our work. We then provide some previous work in sequencers in general, including previous work in hierarchical sequencers. We then introduce Seq, the details of its internal model, and some of its functionality, including macros and parameterization, two features special to Seq which increase its ability to exploit hierarchy, modularity, and variation. We then discuss where we are going with Seq and why we had developed it.

AM '24, September 18–20, 2024, Milan, Italy

ACM ISBN 979-8-4007-0968-5/24/09
https://doi.org/10.1145/3678299.3678343

2 HIERARCHY AND MODULARITY IN MUSIC

While music is usually notated and experienced in a linear fashion, many musicologists have suggested that it is best understood when described in a hierarchical, multi-layered manner. One of the most notable theorists that have opened the door to such descriptions of music is Heinrich Schenker [33]. He suggested that well-formed music is designed by starting with a basic abstract structure, which he called the *Ursatz* or *background*, and expanding and embellishing on it to form the *middle ground* and ultimately the *foreground* (the full composition as written and performed). His analyses did the opposite operation, reducing a composition (i.e. abstracting out the "less important" notes and elements) from the full song to eventually reach the Ursatz, thus creating a tree-structured hierarchy in the composition.

While Schenker's analyses were at times arbitrary in the way the reductions were performed, his work has remained very influential. A well-known later book, *A Generative Theory of Tonal Music* (or *GTTM*) [18] draws inspiration both from Noam Chomsky's work in linguistics and from Schenker's work in music, once again using the idea of reducing a melody and describing hierarchies of notes in the musical composition. GTTM differs from Schenker in two important ways: first, it is much more rigorously defined, and second, it is proposed as a theory of musical cognition, rather than simply as a way to analyze musical composition.

GTTM's cognitive perspective is significant. The claim that music is not only analyzable but also more generally conceived and perceived in a hierarchical manner can seem counterintuitive, given that we hear music notes linearly as they are performed. However, those notes are always perceived in relation to the notes surrounding them, to the general setting of the piece (primarily key and meter), and indeed in relation to the content and more abstract elements of the whole piece [32, 35].

This becomes evident when listening, for example, to pieces generated via early algorithmic approaches that did not consider long-term dependencies. In those cases, the music seems to "wander off" after a couple of measures despite sounding reasonable when only considering the local context [7, 11]. Additionally, GTTM's higher degree of formalization compared to Schenkerian analysis has practical consequences. While GTTM's authors did not really provide an algorithm for the application of the reduction rules they described, several researchers have since implemented those rules (or rules inspired from them) into automatic systems for musical analysis [20, 21, 34].

2.1 Modularity and Reuse

The term "modular", as used here, is a computer science term and not one normally found in music theory. But it is well understood that music may be divided into smaller segments that are repeated, reused, and combined to give a music piece a coherent structure.

This is usually associated with the concept of musical form, which analyzes macro subdivisions of a piece and their roles with respect with one another and to the entire song. One example is the *Sonata* form, where two main contrasting sections (also called themes in this case) are first presented, then repeated, and then varied, and finally repeated again with slight variations that reconcile the two themes, following well established practices.

It is also possible to consider musical structures at a smaller scale, dividing a piece into phrases, periods, and motifs, similar to how natural language can be subdivided into smaller chunks. These short bits of music can then be compared with each other, looking for repetition in melodic content but also by finding phrases that are variations of previous ones, or that directly contrast with other parts of the piece. The baroque era is a particularly good example of such musical reuse, as composers of that era employed retrogrades, inversions, augmentations and diminutions, and other variations of melodic motifs within a piece. The Fugue form is a testament to this practice.

Beyond these obvious examples, the reuse of melodic material is an essential building block in composition, and serves the purpose of making a piece feel cohesive and structured, as opposed to "wandering off" as mentioned in the previous Section [37].

As with hierarchy, there has been much computational musicology research based on modular approaches that concern both the segmentation of a music piece at different time scales [6, 36] and the analysis of how different segments relate to one another [1, 7, 27].

2.2 Application to Music Analysis

This history of formal treatment of hierarchy and modularity in music theory has in turn inspired software primarily meant for musical analysis and automated composition or music generation. For example, building on their computational implementation of GTTM, Hamanaka et al [22] developed a tool for the morphing of melodies. Others have used ideas inspired by either GTTM or Schenkerian Analysis for the creation of melodies [8, 15]. David Cope's *Experiments in Musical Intelligence* [10] are strongly based on the idea of dividing music into functional segments, thus making modularity a core mechanism. More recently, researchers trying to improve structural aspects of generated music have designed systems to guide generation based on similar features, reusing melodic content [7, 26].

These tools have exploited modularity and hierarchy for musical analysis and generation; but it is far less common to find ones meant for human *music making*, or which empower the (human) composer. Instead, music production tools, notably sequencers, are still largely linear. They often have simple ways to define sections and repeat them, but these are usually only as flexible as repetition marks on a score.

In the next Section we review historic and current sequencer software, including the few that have historically made use of hierarchy and modularity, and then in the following Section we will introduce Seq with our desiderata for a hierarchical and modular sequencer design for composition and performance.

3 RELATED WORK IN SEQUENCERS

For our purposes, sequencers may be divided into two rough categories: *step sequencers* and *arrangers*. Step sequencers play fixed and repeating arrays of notes or drum triggers, commonly known as *patterns*. They get their name because they are iteratively *stepped* to get the next set of notes to play. Step sequencers were the earliest examples of sequencers, originating in the synthesizer community, and have since found a niche in drum machines and grooveboxes.

Despite being a pervasive tool, step sequencers have seen little evolution since their inception. Even now many step sequencers still only allow for a single array of notes or events, with the only higher structure sometimes available being *song mode*, that is, playing a series of patterns rather than looping a single one.

In the 1990s special step sequencers called *trackers* added so-called *effects* to individual steps, such as delaying a step by a certain amount of time, or arpeggiating a chord, etc. Trackers have recently seen nostalgia-driven popularity both in hardware and software.

In contrast, *arrangers* store and edit one or more arbitrary streams of time-based note data in a manner similar to a player piano roll. Arrangers are often used to lay out entire compositions in linear order. Arranger-style sequencers originated in the early 1980s with the advent of the personal computer, and became particularly popular with the arrival of early GUIs [16]. Arrangers have thus long been implemented in software, though there are some important standalone hardware examples such as Yamaha's QY series, and certain keyboard workstations. Arrangers have since largely morphed into *Digital Audio Workstation* or *DAW* software, which can record and play back both sequencer event data and also linear audio data.

Some previous work considered new ways to interact with sequences, for example, by creating tangible step sequencers in which events or notes are represented by physical objects, like pucks in *Radear* [2] or marbles in the sequencers of Fischer and Lau [12]. While the change of interface is surely impactful, the final result here is still a basic fixed-time sequence.

One example that tries to escape from this fixedness is *pDM* [13], which employs the KTH music performance rules to obtain an expressive execution of the notes inserted in the sequence, with added options for real-time control. This approach impacts the execution, but it does not alter the way notes and musical structures are built via the sequencer.

Manhattan [28] offers such evolution, as it expands on the idea of a tracker by hybridizing it with the concept of spreadsheet coding. By allowing the events (cells) of the tracker to be formulas, the system becomes a powerful coding tool for music.

3.1 Past Graph-Based Sequencers

Some more radical approaches use graph-based representations in which the nodes are musical events and the edges or paths that connect nodes represent the time spent to reach the next event. Sequencers of this type can be structured to provide limited modularity but are not designed for modular reuse or hierarchy per se. Notable examples include *Nodal* [24] and *Midinous* [31]. Here the structure emerges from the way the multiple possible paths (which may contain probabilistic conditionals) are explored, making it an enticing alternative to more classical approaches for generative music, but not a direct evolution of the way we describe musical structures.

Context [14] defines a graph where each node is a step sequencer. In addition to producing MIDI data, nodes can send messages to other nodes based on conditional triggering rules. These messages may change parameters or state features in other nodes, or trigger events in them. Context can also trigger events with a limited degree of stochasticity. Context provides a restricted form of parallelism, as its nodes essentially form a nondeterministic finite-state automaton (NFA). It can form a simple series or tree of triggered step sequences, but is not designed for modular or multilayered hierarchy.

3.2 Past Hierarchical Sequencers

An early example of hierarchical sequencing lies in drum machines with *song mode*, of which Roger Linn's LM-1 is the earliest example we are aware. Song mode is a simple two-level hierarchy, with drum patterns as leaf nodes, and a single root non-leaf node stating how to string the patterns together to form a full "song".

Arrangers also provided a kind of fixed two-level hierarchy, with some N tracks grouped to run in parallel under a master node. Many modern DAWs also allow the notes in each track to be broken into some M clips for editing convenience, played in sequence: thus we might view the structure as one with three levels: a root node, with N parallel children (the tracks), each with some number of clips.

These are fixed hierarchies. Arbitrary hierarchy in sequencers is less common. Among the earliest (1986) was the *Hierarchical Music Specification Language* or *HMSL* [5]. HMSL organized songs as a strict tree structure, with leaf nodes being basic music sequences, and non-leaf nodes being various ways to compose or reorganize child subtrees. HMSL was not an application so much as an API written in FORTH (later Java) meant for explorative music composition. As a thin layer over Forth, HMSL nodes were themselves programs with considerable flexibility. One notable node would use a Markov Model to select its children to play at random.

Two years later saw the *Hierarchical Event Sequencer* [30], a similar tree-based music sequencing system. A more recent example is *Ossia Score* [9], a large hierarchical event sequencer for MIDI, audio, and other media. Finally, the recent Ableton plugin *Flow* [29] employs essentially a degenerate form of HMSL's model, solely using a tree of Markov Models with notes or chords as leaf nodes.

These sequencer examples do not provide both hierarchy and modularity: that combination is rare. Indeed we are aware of one major example: Goffredo Haus's *ScoreSynth* [17], an attempt to represent music structures as a hierarchy of Petri Nets for music composition. A Petri Net is a graph structure, originally from chemistry, meant to model coordination or interaction among multiple parallel entities. In a Petri Net *tokens* wander from *place* to *place*, passing through *transition* objects which control their flow and which produce parallelism through the production and consumption of additional tokens. In ScoreSynth, a token is the current position of a thread of execution where ScoreSynth is playing music, and a place is either a chunk of music or it is another Petri Net. The transitions in ScoreSynth were augmented with probabilistic transitions and counters.

ScoreSynth does not have a hierarchy so much as an arbitrary call graph: Petri Nets may contain other sub-Petri Nets, including themselves. This permits recursion, but Haus has informed us that it was considered an error to do so (as there was no stopping condition in the recursion). Without recursion, ScoreSynth's model degenerates to a *directed acyclic graph* or DAG similar to Seq's model as discussed next, but even this did not appear to be used very often. In most of the ScoreSynth literature, hierarchies were simply trees.

ScoreSynth also had a limited kind of parameterization whereby some immediate variables in a Petri Net could be modified by the

musician in real time. ScoreSynth also had "macros" (not to be confused with the macros of Section 7), which were structural templates to simplify the construction of Petri Nets.

ScoreSynth could be used as a sequencer or composition tool, but it was more often meant for music analysis and playback. To this end, it supported not just note event data but audio and multimedia facilities.

We note that hierarchy *and* modularity are supported in one form or another in music programming languages, since the combination of hierarchy and modularity is equivalent to a function call or macro expansion. There is a long history of music languages with this quality. Two current popular ones (particularly in the "live coding" community) are *SuperCollider* [23] and *Tidal Cycles* [25]. The latter in particular offers ways to design patterns which can be seen as similar to sequencers and arrangers. Both are however also synthesizers (SuperCollider is used within Tidal Cycles for sound synthesis) and deal with music and music sequencing at the audio level, unlike a classic sequencer.

4 SEQ

Seq is meant as an experimental tool, and so in designing it we had a number of desiderata unusual for a sequencer.

First, we wanted to explore how to allow the composer or performer to structure a sequence *hierarchically*, that is, to enable *parents* to modify, compose, or transform their children and ultimately descendants. Additionally, we wanted to make a wide variety of modifications, compositions, and transformations available to the musician: and so nodes in the hierarchy might consist of many kinds of operators.

Second, we wanted to exploit modularity to allow the musician to *reuse child elements*. Any sequence, or composition or transformation of sequences, should be reusable by a variety of parents as they saw fit, and independent of one another. We also sought to allow entire hierarchical musical structures to be *encapsulated* into a single element for reuse as a group or to be used as a template or pattern.

Third, we intended this modular reuse to be subject to *variation*. We wanted parents to be able to usefully modify reused children in simple and parameterizable ways. This parameterization and variation should be changeable in real-time for performance and automatic generation.

Fourth, to the best of our ability we wanted these modular elements to be recombined, played, or transformed in real time by the musician, to enable new kinds of performance. We wanted a structure that was easily "hacked" and modified on the fly. Similarly, we sought to ultimately enable the possibility of live coding.

Fifth, we ultimately wanted to explore how AI and optimization algorithms might be used to automate both the development of the hierarchical structure and the parameterization of the modules and their compositions and transformations. This could be used to enable co-creative assistants in the composition process, or generative or collaborative performance.

4.1 Static Model

Seq's hierarchical structure is a *directed acyclic graph* or *DAG*. Unlike in a tree, in a DAG any node can have zero or more parents and zero or more children, but cycles are not permitted. Nodes with no children are known as *leaf nodes*, and the others are *non-leaf nodes*. A DAG is a common data structure: it provides hierarchy by allowing parents to have any number of children or descendants; and it provides modular reuse by allowing multiple parents to share the same child.

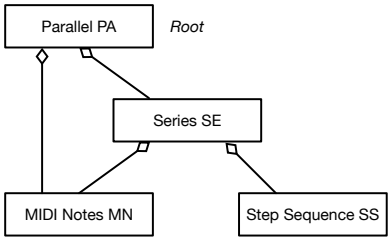

Figure 1: Trivial DAG example. PA is the Root.

In Seq, a leaf node is typically something capable of playing musical notes in time, such as a step sequence pattern or an arranger with a timeline of musical notes (a piano roll). A non-leaf node is normally a composition or transformation of one or more leaf or non-leaf nodes into a larger musical structure.

One node is designated by the musician as the *root*, and the *active sequence* consists of the root and all of its children and descendants. The active sequence is played starting at the root. This is not a "root" of the DAG the classic sense (that is, a node with no parents): indeed Seq's root may have parents, and there may be other nodes with no parents. However the *active* musical sequence is that portion of the DAG which descends from Seq's root. Other nodes in the DAG, such as the root's parents or nodes not descended from the root, are ignored.

As a simple example, consider the DAG in Figure 1. Here we have a *Step Sequence* SS and a timeline of *MIDI Notes* MN, each of which plays a single time if started. We set them both as children of a *Series* SE which, when started, restarts and plays SS three times, then MN once, and then terminates. SE and MN are themselves added to a *Parallel* node PA which, when started, plays MN, and then three measures later starts playing SE at the same time. PA finishes when both SE and MN have finished. We set PA to be the root.

In Seq, we use the term *motif* to describe the object which holds the static and structural data of a node. A motif can have more than one instance of the same motif as a child. For example, PA could be set to play SE and MN in parallel, and then later on introduce playing SE *in parallel with itself*. To keep track of this, each motif maintains a collection of unique *child* objects, each of which points to a motif and assigns it a nickname.

One special kind of motif, discussed later, is the *Macro*. This motif contains can encapsulate another entire DAG of motifs within it. This is recursive: a Macro's DAG may itself contain other Macros.

4.2 Dynamic Model

The static model is sufficient for describing sequences but not for playing them. When playing a sequence we must retain runtime information regarding each motif, such as where it is in its sequence, what current children are being played, and so on. Furthermore,

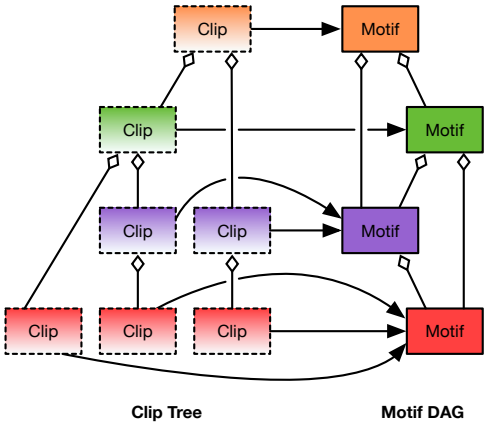

Figure 2: Motif DAG and its corresponding Clip Tree. Diamond arrowheads indicate parent-child relationships.

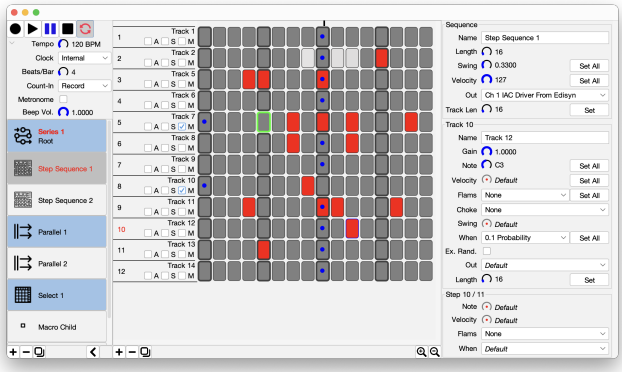

Figure 3: Seq interface, showing a Step Sequencer node. Blue clips indicate parents of the Step Sequencer.

there may be more than one instance of the same motif playing at the same time. This happens in many contexts, but parallelism is the easiest illustration. Using our previous example, we can see that PA may be playing MN, while simultaneously playing SE which is *also* playing MN, perhaps offset in time.

To deal with this, we must unravel the motif DAG into a tree of equivalent objects which hold dynamic information. Internally, Seq refers to the dynamic model as a tree of *clips*. Figure 2 shows a motif DAG and its corresponding clip tree. The clips store individual runtime information as they are distinct from one another. In particular, each clip has a *position* in time where it is currently playing. Time is relative: 0 is the start of the clip. Clips know if they are playing and/or armed to record. The dynamic model is only used at runtime: when a sequence is saved, only the motifs are saved.

Building the clip tree is straightforward. First we create a clip R' to serve as the clip tree root corresponding to the motif DAG root R. Then starting with R' and R, we recursively do the following. Given a clip P' whose corresponding motif is P, for each child C of P, we create a *unique* clip called C' and add it as a child of P'. We repeat this procedure recursively on C'. Recall that a motif may hold the same child C multiple times: these are all considered separate C and each will have its own unique C' in the clip tree.

This unravelled clip tree may be much larger than its compact DAG, particularly if the DAG employs a high degree of reuse. But we can dynamically unravel only portions of the clip tree as necessary, and then garbage collect them when they are no longer needed.

4.3 Playing a Clip

Clips are played by resetting them to their start position 0, then iteratively *stepping* them. When stepped, a clip advances its timestep, then either generates MIDI data or resets and advances child clips recursively as appropriate. Clips do not emit MIDI data directly, but rather they offer it to their parents, who hand it to their parents and so on. This gives parents a chance to transform the MIDI data, such as transposing it or changing its volume or channel. Ultimately the root clip will hand the MIDI messages to the system to output.

Messages are output to one or more *output ports* set by the user: a port is a named combination of a MIDI device and a channel. Clips of certain motifs can also be controlled, or armed and recorded, from one or more *input ports* set by the user.

When a clip finishes playing, it will inform its parent that it is done. The parent then will normally choose a different clip to play, or tell its parent that it is done as well. It could however ignore this information and continue to play the clip (with undefined consequences).

Motifs and clips can be fully modified even while playing. Music, parameters, and routings can be changed in real time of course, but the Motif DAG can also be entirely restructured, which may require that the clip tree be rebuilt starting at the highest modified node. The only restriction is that the root node may not be changed to another node while the sequencer is playing.

5 THE INTERFACE

As Seq is written in Java, Seq's graphical interface is in Swing, and consists of four major parts, as shown in Figure 3.

At top left is the *console*, holding the transport controls (play, record, and so on), plus a collapsible set of clock options. Immediately below the console is the *motif list*, showing all the motifs in the sequence. The root motif is marked *Root*, and the currently playing motifs are highlighted in red text. Exactly one motif is selected: it is in gray. Its parents are shown light blue and its children are in pink; further ancestors and descendants are shown in lighter blues and pinks.

The middle region displays the structure of the selected motif. Figure 3 shows a step sequence, and Figure 4 shows a Series object (which is also set as Root).

At the far right are the *inspectors*. The top inspector displays the properties of the selected motif. Below it, additional inspectors display properties of selected elements within the motif. For example, in Figure 3 there is an inspector for the step sequence as a whole, a second inspector for the selected step sequence track, and a third inspector for the selected step in that track.

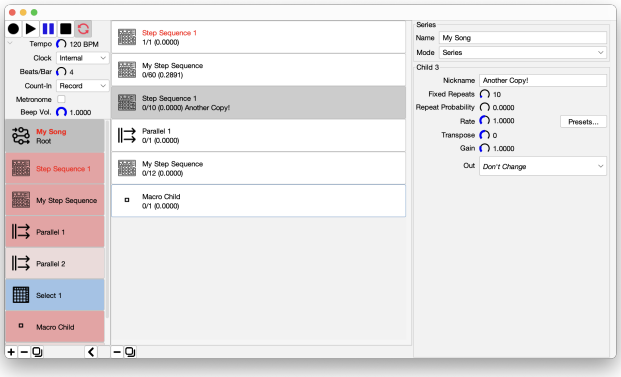

Figure 4: Seq interface, showing a Series node. Blue clips indicate parents of the Series, and pink clips are its children. The center region contains all the children, in order, to be played by the Series.

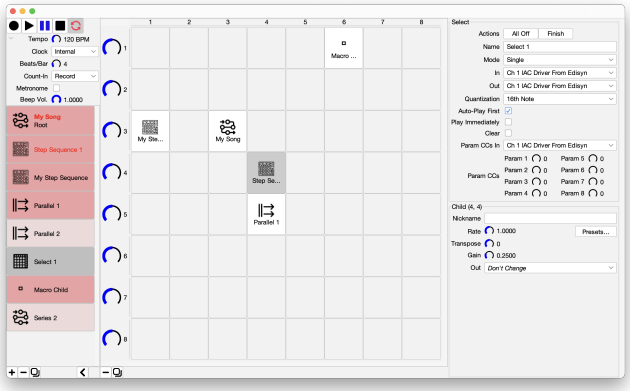

Figure 5: Seq interface, showing a Select node. The 8×8 grid maps to the buttons on a Novation LaunchPad or other grid controller: when a button is pressed, the corresponding child clip is started.

6 THE STANDARD MODULES

At present Seq contains three leaf node motif types (*Step Sequence, Notes,* and *Silence*), four non-leaf node motif types (*Series, Parallel, Automaton,* and *Select*), and two special motif types for building macros (*Macro* and *Macro Child*). Macros will be discussed later in Section 7. Their names notwithstanding, several of these motif types can serve in a variety of roles. Seq is extensible and other motif types may be added in the future: for example, we might add a tracker, an arpeggiator, or a modulation generator.

6.1 Step Sequence

The *Step Sequence* motif is a full-featured multitrack step sequencer. It can have any number of tracks, and each track may consist of a different number of steps. Each step may have its own note, velocity, flams, and rules regarding when it fires (such as probability). Tracks can have chokes, gain, swing, exclusive random groups (where only one track in a group plays at a time), and a MIDI output port; and can muted, soloed, and armed for recording notes into them. Tracks and the Sequence have default settings for many per-track or per-step features.

6.2 Notes

The *Notes* motif is a piano roll, effectively a single track of a linear arranger. It stores a collection of MIDI data, including note events and real-time parameter changes. The roll can be armed and recorded, loaded from and saved to MIDI files, and edited. Notes can specify both the Input Port for recording and the Output Port for playing.

6.3 Silence

This is simply an empty interval of time of some length.

6.4 Series

A *Series* holds a collection of motifs as children in a list, and normally plays each of them in order. It is essentially like a drum sequencer's "song mode". Children may appear more than once in a Series, and may be looped any number of times both deterministically and probabilistically. Children may be fixed in length so that they are terminated early if they continue too long (or forever). Series can also play children in random order in several different ways, including using a Markov Model (as in HMSL [5]).

Series can also transform its children in certain ways. It can change the rate at which the child is played, transpose or change the volume of its notes, or change its MIDI output port.

6.5 Parallel

Parallel, like Series, maintains a collection of motifs as children in a list; but as befits its name, Parallel plays these motifs simultaneously. Each motif can be delayed so that it starts at a different time, and so a Parallel to some degree resembles a trivial linear arranger. Parallel can also be set up to play (or mute) children with some probability. Parallel lets one set the probability independently, or stipulate that no more than N children will play, chosen at random using their probability values. Just as in Serial, child motifs of a Parallel can be fixed in length and their output notes can be transformed.

6.6 Select

Like Series and Parallel, *Select* also maintains a collection of motifs as children in a list. However Select is special: it allows the musician to manually launch and stop clips in real-time, using its graphical interface or a Novation Launchpad. Clips can be launched independently or exclusively, be one-shot or repeating; and can launch immediately or wait until the previous clip has finished.

Select is unusual because, as it is primarily a user interface mechanism, it essentially runs forever and doesn't have a time when it declares that it is done. The musician can manually declare this to its parent, however.

6.7 Automaton

Seq also sports an elaborate combination of Series and Parallel features in the form of a parallel *finite-state automaton* or *FSA*, as illustrated in Figure 6. An FSA is a graph structure of nodes, some

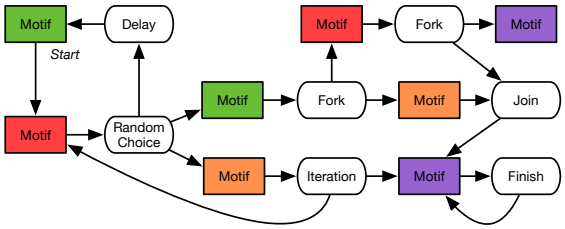

Figure 6: An Overly Complex Finite-State Automaton.

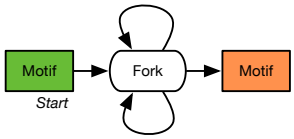

Figure 7: A simple fork bomb.

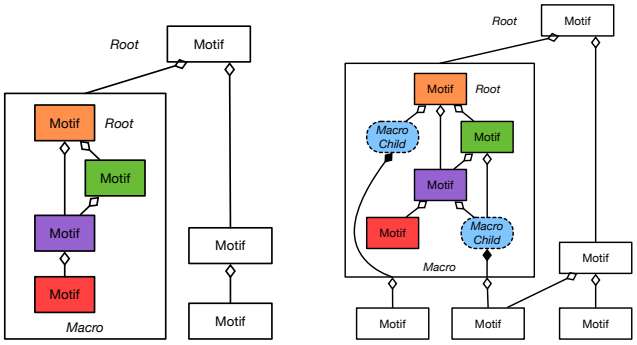

Figure 8: A Macro as a leaf node (Left) and as a non-leaf node with two children (Right) in a DAG. Diamond arrowheads indicate parent-child relationships.

of which are *active* at any particular time. One node is designated the *start node*, and when the FSA is started, this node is the only active one. When nodes cease activity, activity *transitions* to other nodes along edges connecting one node to another.

Automaton nodes are one of two types: *motif nodes* (children) and *actions*. When a motif node is active, its underlying motif is played, and when it is no longer active, it transitions to up to one follow-on node. When an action is active, it (usually) immediately transitions in zero time to one or more follow-on nodes depending on rules it follows.

Automaton offers several action nodes: *Random Choice* nodes randomly select a follow-on node using a distribution. *Iteration* nodes select a different follow-on node each time using a counter. *Finish* nodes declare to the FSA's parent that the FSA believes it is done. A *Fork* node immediately transitions to *all* of its follow-on nodes (introducing parallelism). A *Join* node transitions to its sole follow-on node only after receiving two incoming transitions (thus merging them). *Delay* nodes do not transition immediately, but rather do so after a certain period of time. *Chord* nodes play a single note or chord, then transition after the chord has completed.

An Automaton may have cycles in its graph structure, including nested cycles, and this presents a difficulty in combination with parallelism. Consider the *fork bomb* in Figure 7, where the Fork is feeding back into itself, producing more and more threads. To prevent this, we must place restrictions on the combination of cycles and parallelism. We could disallow cycles if parallelism is being used (or vice versa); or we could have threads somehow detect that they are in both a cycle and a fork, and terminate at that point. We have chosen instead to simply limit the total number of threads permitted at one time: if a Fork tries to generate more threads, it fails.

Automaton's power is at least that of the augmented Petri Nets used in [17]. Augmented Petri nets are more consistent in design, as there is exactly one kind of action node, the *transition node*, which roughly combines a random choice, iteration, fork, and join. But we think this all-in-one combo node proved rather awkward to use and difficult for composers to conceptualize or design with. We believe that by breaking out individual basic actions, the musician can more easily select from them the exact action he wants to apply.

7 MACROS

Macros add a completely new mode of hierarchy and reuse to Seq. Our goal in adding Macros was to experiment with not one but several different and powerful ways that modularity could be exploited. In addition to sharing and reusing children, a Macro can also share and reuse *entire internal subsequences;* and the leaf nodes of these subsequences can be customized with pluggable sub-subsequences. This effectively converts the subsequences into templates or customizable functions.

A Macro is a special motif which not only can have children, but can also encapsulate its own *rooted motif DAG*. Macros may themselves contain other Macros in their internal DAGs. Just as the primary motif DAG has a corresponding clip tree, a Macro's internal motif DAG has a corresponding internal clip tree. When the Macro is reset, it resets its tree; and as the Macro is stepped, it will in turn step the clips in its internal tree in order.

Figure 8 (Left) gives an example of a Macro serving as a leaf node in an outer DAG. Macros as leaf nodes may be dismissed as little more than a way to reuse whole sequences saved as files. But Macros can also be *non-leaf nodes*, as shown in Figure 8 (Right). Here, a Macro's internal DAG may contain zero or more *Macro Children* motif nodes.

A Macro Child is a special leaf node Motif which acts as a stub for an actual child of the Macro in the outer DAG. Each Macro Child is associated with a unique child of the Macro motif itself. When a Macro Child is stepped, it in turn steps the actual corresponding child of the Macro clip. This allows Macros to not only serve as sub-DAGs, but as templates or functions which process or compose children in the main DAG.

8 PARAMETERIZATION AND VARIATION

Modular reuse of sub-elements in music is not very helpful unless we can also provide variation. For example an AABA pattern is common, but perhaps more common is AABA′, where A′ is a twist on the original A. Additionally, particularly in live performance or in automated music generation, it's important to provide real-time

nuance and change to pieces of a song as they are played. We have explored approaches to doing both of these tasks.

All motifs in the sequencer can be *parameterized*. To do this, each motif has some *N parameters* which can each be set to a value from 0.0 to 1.0 inclusive. A parameter is used to make either preset or real-time changes to the motif. During play, a parent clip will set the parameters of a child clip to values of its choosing (that is, to certain *arguments*). Different parents may set the parameters to different arguments. This is not unlike setting function parameters to arguments in a programming language.

Any node, leaf or non-leaf, can assign its parameters to various features of the node. For example, a step sequence may set one parameter to the accent of notes on a certain track, or the degree of swing in the sequence. Or a non-leaf node might assign one of its parameters to the transposition of notes from its children.

Non-leaf nodes can then *bind* the arguments of their children to *ground values*: for example, a Series might set parameter 5 of a certain child to 0.25; or to the current value of MIDI Continuous Controller (CC) 42 sent from the musician's keyboard controller; or to a dial that the musician changes in the UI while the sequence is playing. A parameter may be controlled by a special motif in the form of a *Low Frequency Oscillator* (LFO) or an *Envelope Generator*, both of which change a child argument according to a time-varying automatic function (which itself may be parameterized). Or the parameter may be changed each iteration, or in real time, by a random number generator.

Furthermore, non-leaf nodes can bind arguments to their *own parameters* to pass the same parameter information to their children. Multiple child arguments could be bound to the same parent parameter.

Parameterization works for Macros as well. The parameters of the root node of a Macro's DAG are each bound to the parameters of the Macro itself; and parameters of children to the Macro are bound to the parameters of its internal Macro Child nodes.

Each motif also has a special parameter called *random*, which provides a random value. This value is changed once each time the motif is replayed. Parent motifs may not change this value, but they can provide arguments to set its minimum and maximum bounds.

8.1 Variation by Parameter and by Selection

We might use parameters to specify variation. For example, the Series motif can be set up to use Parameter 1, or the Random Parameter, to determine *which* of several children should be played. It then plays that child once, then finishes itself.

Another possible scheme would be to dedicate one parameter to indicate the *degree* of variation. Setting the degree parameter to > 0 would direct a node, if it was set to participate, to behave as a random variant of the standard node. The degree pattern would indicate how strong the variation should deviate from the standard node. Degree information could be passed from parents to children, and so the entire sub-DAG would be affected.

It it theoretically possible that variation could involve dynamic changes to the structure of the DAG itself: though it seems likely that this would be computationally too costly.

9 OBSERVATIONS

Seq's design is very different from other sequencers, and this design brings with it certain challenges and issues. We mention a few here that we are still mulling over, largely as examples of observations made during the design of this tool and the first internal tests on the working alpha version.

9.1 Bottom-Up and Top-Down Design

A musical composition might be primarily constructed *top-down*, or *bottom-up*, or a combination of the two. By top-down we mean that the composer starts with the high-level structure of the piece and little by little fills in the details. Whereas bottom-up means that the composer starts with interesting small chunks, then figures out how to arrange them together into larger elements, and so on until he has combined elements into a finished piece. In both cases, the construction involves working one's way through the hierarchy. However it seems to us that Seq strongly promotes bottom-up construction: it encourages the development of sequences, then the composition of those sequences by new parent motifs.

This is perhaps not unusual: we imagine that a linear clip-based arranger often encourages similar bottom-up construction. And to our minds, the bottom-up method has much to offer in terms of creative design. However there are many uses for top-down construction, and we would like to see tools introduced into Seq to assist in that approach. At present the musician can construct *stubs* — Silence motifs, say — to attach to a top-level object and to be replaced in the future with more detailed children. But we are contemplating what facilities a more "sophisticated", purpose-built stub motif might have, and how it could be better integrated into the framework.

9.2 Dynamic Interface Display

A single motif may have many clips which play at different times in the song, or even simultaneously. Seq can play them all without any issue: but it cannot necessarily *display them all*. The problem is that Seq's primary motif display shows two things: the static motif data (which all those clips share), and dynamic information such as playing position (which is different per-clip). As one clip stops playing and another stops, the display may lurch from one to the other: and if two clips are playing in parallel, we must decide which clip gets to display.

This is not a problem which exists in a DAW-style linear arranger, as the note data is normally stored statically. At present we are simply letting clips try to assign themselves the role of *display clip* if no one else has that role at present, and we display the data of the current display clip.

9.3 Seeking

Seq by design permits nondeterministic songs of dynamically changing or infinite length, and this makes it hard to jump to a specific location in a song and play from there. One option would be to play Seq and internally fast-forward to the point without emitting anything; but Seq will have to make arbitrary choices along the way when confronted with nondeterministic decision points (such as random choices or user selection). Another option would be to always start at the beginning of the root: but recall that the user

can set the root to any motif he likes. In some sense, changing the root is like changing Seq to work on a certain section of the project.

10 ULTIMATE GOALS AND FUTURE WORK

Seq is at the proof of concept stage: we have built the entire system, it works, and we can confidently say that it opens up many new and unusual avenues for exploiting modularity, hierarchy, reuse, and variation in a compositional and performance. But it will take additional testing to assess how, and to what degree, Seq is effective as a tool in this regard.

Our first goal will be to assess Seq directly as a tool for music composition and production. Our plan is be to work with selected composers interested in testing its unusual workflow to see how well, or if, they can embed it in their compositional practice. Secondly, we are working with colleagues in the modular synthesis and electronic music performance community to see how they might test its use in live-building of sets on-the-fly.

10.1 Automation and Computational Creativity

One of our primary ultimate goals is to produce not only a model usable for composition and performance, but also for the automated or assisted exploration of the space of songs. Past research in this area has argued for the value of abstracted hierarchical descriptions of form and musical structure [3, 7, 37].

There are many possible ways to explore the space of compositions. For example, we could apply *interactive evolutionary computation* to have the system iteratively propose, and then refine, songs optimized to a musician's criteria. We have previously developed *Edisyn*, a synthesizer patch editor library which offers a very effective interactive evolutionary computation tool to explore the space of synthesizer programs [19]. Edisyn iteratively produces and auditions a set of patches, the musician then selects those he prefers, and Edisyn then mixes and mutates them to produce new variations, and repeats. Edisyn in effect works with the musician to wander through the patch parameter space towards those synthesizer programs he prefers without him needing to program them directly. We think a similar approach could be taken here, mutating the DAG structure and its parameters, or building a DAG from basic building blocks provided by the musician.

One could also take many previous songs and morph between them both structurally and in terms of parameters, to produce an interpolation between them. Alternatively, if we had a large enough corpus of examples, we could train a neural network or similar model to build a manifold through the space of songs to generate new ones "similar" to those previously developed.

10.2 Extensions to Parameterization and Macros

Seq is aimed not only at composition and analysis but at real-time performance. Seq's Select module and Seq's intrinsic parameterization allows a musician to vary the song in many ways in real-time. In the future we aim to expand on this further. We are been considering developing a Seq motif for so-called *live coding* [4], whereby a performer builds a song from computer code in real time. We are also interested in developing motifs to do (parameterizable) modulation of parameters: envelope generators, low-frequency oscillators, random number generators, and so on.

Seq's modularity allows a musician to essentially build up a library of predefined musical structures and reuse them in a wide variety of ways. A musician could even publish this library on a central server for use by other musicians as part of their songs; the Macro facility would be particularly useful in this context.

11 CONCLUSION

Seq is a very unusual sequencer designed specifically to examine hierarchy and modular reuse in musical structure and to exploit it in composition and performance. Seq treats elements in songs, or even entire bodies of work, as building blocks to mix and combine, and to create variations. Seq has many capabilities, including systemic parameterization at all levels of the hierarchy, multiple approaches to reuse, transformation of MIDI data at different levels of abstraction, and a variety of components.

We intend for Seq to serve many roles: it can enable rapid and more creative composition; or offer sharing of compositional material; or offer new approaches for real-time generative performance. Paired with machine learning and optimization, we hope its model will make possible co-creative exploration of the music space. But more generally, we hope that Seq will offer new perspectives to musicians as to the fundamental nature of music and how one may interact with it.

ACKNOWLEDGMENTS

Our thanks to Phil Burk, Goffredo Haus, Antonio Rodà, Geraint Wiggins, and Vi Hoyle for their help, feedback, and criticism, and to LIM at U. Milan for its support.

This work is partly funded by the European Union, HORIZON-MSCA-2022-PF-01 Project ID 101108690 (CALIOPE). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Executive Agency (REA). Neither the European Union nor the REA can be held responsible for them.

REFERENCES

[1] Samer Abdallah, Nicolas Gold, and Alan Marsden. 2016. *Analysing Symbolic Music with Probabilistic Grammars.* Springer, 157–189.
[2] Daniel Gábana Arellano and Andrew P McPherson. 2014. Radear: A Tangible Spinning Music Sequencer. In *Conference on New Interfaces for Musical Expression (NIME).* 84–85.
[3] Matt Bellingham, Simon Holland, and Paul Mulholland. 2014. *An Analysis of Algorithmic Composition Interaction Design with Reference to Cognitive Dimensions.* Technical Report. Department of Computing, The Open University, https://oro.open.ac.uk/90308/.
[4] Alan F. Blackwell, Emma Cocker, Geoff Cox, Alex McLean, and Thor Magnusson. 2022. *Live Coding: A User's Manual.* MIT Press.
[5] Phil Burk, Larry Polansky, and David Rosenbloom. 1994. HMSL, the Hierarchical Music Specification Language. https://softsynth.com/hmsl/. As of 1/1/2024.
[6] Emilios Cambouropoulos. 2001. The Local Boundary Detection Model (LBDM) and its application in the study of expressive timing. In *Proceedings of the International Computer Music Conference.* 290–293.
[7] Filippo Carnovalini, Nicholas Harley, Steven T Homer, Antonio Rodà, and Geraint A Wiggins. 2021. Musical Structure Analysis and Generation Through Abstraction Trees. In *International Symposium on Computer Music Multidisciplinary Research.* Springer, 282–300.
[8] Filippo Carnovalini and Antonio Rodà. 2019. A Multilayered Approach to Automatic Music Generation and Expressive Performance. In *Workshop on Multilayer Music Representation and Processing (MMRP).* IEEE, 41–48.
[9] Jean-Michaël Celerier. 2018. *Authoring Interactied Media: a Logical & Temporal Approach.* Ph. D. Dissertation. Université de Bordeaux.
[10] David Cope. 2005. *Computer Models of Musical Creativity.* The MIT Press.
[11] Shuqi Dai, Zheng Zhang, and Gus G. Xia. 2018. Music Style Transfer: A Position Paper. arXiv:1803.06841 [cs.SD]

[12] Thomas Fischer and Wing Lau. 2006. Marble track music sequencers for children. In *Conference on Interaction Design and Children*. 141–144.
[13] Anders Friberg. 2006. pDM: An Expressive Sequencer with Real-Time Control of the KTH Music-Performance Rules. *Computer Music Journal* 30, 1 (2006), 37–48.
[14] Liam Goodacre. 2017. Context: A Modular Approach to Sequencing. In *Workshop on Musical Metacreation (MUME 2017)*.
[15] Ryan Groves. 2016. Towards the generation of melodic structure. In *Proceedings of the Fourth International Workshop on Musical Metacreation*. 1–8.
[16] Chris Halaby. 2010. The Early Days of Software Sequencers. KVR Audio. https://www.kvraudio.com/focus/the_early_days_of_software_sequencers_15670. As of 1/1/2024.
[17] Goffredo Haus and Alberto Sametti. 1991. Scoresynth: A System for the Synthesis of Music Scores Based on Petri Nets and a Music Algebra. *Computer* 24, 7 (1991), 56–60.
[18] Fred Lerdahl and Ray S. Jackendoff. 1985. *A Generative Theory of Tonal Music*. MIT press.
[19] Sean Luke. 2019. Stochastic Synthesizer Patch Exploration in Edisyn. In *Conference on Computational Intelligence in Music, Sound, Art and Design (EvoMUSART)*. Springer, 12.
[20] Alan Marsden, Satoshi Tojo, and Keiji Hirata. 2018. No Longer 'Somewhat Arbitrary': Calculating Salience in GTTM-style Reduction. In *Proceedings of the 5th international conference on digital libraries for musicology*. 26–33.
[21] Keiji Hirata Masatoshi Hamanaka and Satoshi Tojo. 2006. Implementing "A Generative Theory of Tonal Music". *Journal of New Music Research* 35, 4 (2006), 249–277.
[22] Keiji Hirata Masatoshi Hamanaka and Satoshi Tojo. 2022. Implementation of melodic morphing based on generative theory of tonal music. *Journal of New Music Research* 51, 1 (2022), 86–102.
[23] James McCartney. [n. d.]. SuperCollider. https://supercollider.github.io. As of 1/1/2024.
[24] Jon McCormack, Peter McIlwain, Aidan Lane, and Alan Dorin. 2007. Generative composition with Nodal. In *Workshop on Music and Artificial Life (at ECAL)*. 13.
[25] Alex McLean. [n. d.]. Tidal Cycles. https://tidalcycles.org. As of 1/1/2024.
[26] Gabriele Medeot, Srikanth Cherla, Katerina Kosta, Matt McVicar, Samer Abdallah, Marco Selvi, Ed Newton-Rex, and Kevin Webster. 2018. StructureNet: Inducing Structure in Generated Melodies. 725–731.
[27] David Meredith, Kjell Lemström, and Geraint Wiggins. 2002. Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research* 31, 4 (2002), 321–345.
[28] Chris Nash. 2014. Manhattan: End-User Programming for Music. In *Conference on New Interfaces for Musical Expression (NIME)*. 221–226.
[29] Orthogonal Inc. 2024. Flow. https://orthogonal.one/products/flow/. As of 1/1/2024.
[30] Christopher Young Penney. 1988. *The design of an hierarchical event sequencer as an environment for computer-aided composition*. Master's thesis. University of California, San Diego.
[31] James Ratliff ("Nornec"). 2019. Midinous. https://midinous.com. As of 1/1/2024.
[32] Martin Rohrmeier. 2013. Musical expectancy: Bridging music theory, cognitive and computational approaches. *Zeitschrift Der Gesellschaft Für Musiktheorie [Journal of the German-Speaking Society of Music Theory]* 10, 2 (2013), 343–371.
[33] Heinrich Schenker. 1935. *Free Composition (Der freie Satz)*. Longman. Longman Music Series, Ernst Oster, translator.
[34] Federico Simonetta, Filippo Carnovalini, Nicola Orio, and Antonio Rodà. 2018. Symbolic music similarity through a graph-based representation. In *Proceedings of the Audio Mostly 2018 on Sound in Immersion and Emotion*. 1–7.
[35] David Temperley. 2011. Composition, perception, and Schenkerian theory. *Music Theory Spectrum* 33, 2 (2011), 146–168.
[36] Gissel Velarde, Tillman Weyde, and David Meredith. 2013. An Approach to Melodic Segmentation and Classification Based on Filtering with the Haar-wavelet. *Journal of New Music Research* 42, 4 (2013), 325–345.
[37] Geraint A. Wiggins. 2021. Structure, Abstraction and Reference in Artificial Musical Intelligence. In *Handbook of Artificial Intelligence for Music*. Springer Nature.