TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

Viktor Pavlov 200582IAIB

ALGORITHMIC IMPROVISATION FOR LIVE CODING MUSIC PERFORMANCE

Bachelor Thesis

Supervisor
Edward Morehouse
PhD
Supervisor
Chad Nester
MsC

Tallinn 2022

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Viktor Pavlov 200582IAIB

ALGORITMILISE REAALAJA-IMPROMUUSIKA IMPLEMENTATSIOON LAIVKOODIMISE KESKKONNAS

Bakalaureusetöö

Juhendaja
Edward Morehouse
PhD
Juhendaja
Chad Nester
MsC

Tallinn 2022

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, the literature and the work of others have been referenced. This thesis has not been presented for examination anywhere else.

Author: Viktor Pavlov

May 30, 2022

Annotatsioon

Laivkoodimine (inglise keeles *live coding*) on esitusviis, kus esineja loob heliteost programmiliselt reaalajas. Sonic Pi, Tidal Cycles ja SuperCollider on platvormide hulgas, mida kasutatakse algoritmilise muusika esitamiseks. Kuigi mainitud platvormid on võimekad esinemise tööriistad, ei paku ükski neist autori parimate teadmiste kohaselt liidestamist algoritmilise muusika komponeerimise tarkvaraga.

Selle lõputöö projekti põhieesmärk on uurida algoritmilist muusika improvisatsiooni, liidestades Sonic Pi platvorm tarkvarasüsteemiga, mis suudab genereerida reaalajas variatsioone olemasolevatest üksiku hääle meloodiatest. Selle uurimistöö oodatav tulemus on muuta esinemise protsess koostöövõimelisemaks, laiendades arvuti rolli pelgalt instrumendilt improvisatsioonilise assistendini.

Selle lõputöö projekti tulemus on rakendus, mis toetab liidestamist Sonic Pi platvormiga, kasutades Open Sound Control protokolli. Variatsioone saab genereerida MIDI-allikatest ning otse reaalajas Sonic Pi saadetud sõnumitest. Algoritmilised muusika komponeerimise teostused põhinevad Markovi ahelatel ja Google AI Magenta närvivõrgul.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 36 leheküljedel, 6 peatükki, 12 joonist, 4 tabelit.

Abstract

Live coding is a type of performance where the performer creates music programmatically in real-time. Sonic Pi, Tidal Cycles and SuperCollider are among the platforms used for algorithmic music performance. While the mentioned platforms are powerful performance tools, none of them offers out of the box integration with algorithmic music composition software, to the best of author's knowledge.

The main goal of this thesis project is to explore algorithmic music improvisation by integrating Sonic Pi with a software system that can generate variations on existing single voice melodies in real-time. The expected outcome of this exploration is to make the performance process more collaborative, by expanding the computer's role from a mere instrument to an improvisational assistant.

The result of this project is an application that supports integration with Sonic Pi, utilizing the Open Sound Control protocol. Melody variations can be generated from MIDI sources and from messages sent by Sonic Pi directly in real-time. Algorithmic music composition implementations are based on Markov chains and Google AI's Magenta neural network.

The thesis is in English and contains 36 pages of text, 6 chapters, 12 figures, 4 tables.

List of abbreviations and terms

MIDI	*Musical Instrument Digital Interface*
OSC	*Open Sound Control*
RNN	*Recurrent Neural Network*
LSTM	*Long Short Term Memory*
GUI	*Graphical User Interface*
DSL	*Domain-specific Language*
ML	*Machine Learning*
ASCII	*American Standard Code for Information Interchange*
IEEE	*Institute of Electrical and Electronics Engineers*
UDP	*User Datagram Protocol*

Table of Contents

List of Figures vii

List of Tables viii

1 Introduction 1

2 Background 3

2.1 Live Coding . . . 3

2.1.1 SuperCollider . . . 3

2.1.2 Sonic Pi . . . 3

2.1.3 Tidal Cycles . . . 4

2.1.4 Orca . . . 5

2.2 Algorithmic Music Composition . . . 5

2.2.1 Magenta . . . 5

2.2.2 Max . . . 5

2.3 Related Projects . . . 6

3 Methodology 7

3.1 MIDI . . . 8

3.1.1 MIDI Files . . . 8

3.1.2 Header Chunks . . . 8

3.1.3 Track Chunks . . . 9

3.1.4 Messages . . . 9

3.1.5 Example MIDI File . . . 11

3.2 Markov Chains . . . 12

3.2.1 Music Generation with Markov chains . . . 13

3.3 Magenta MusicRNN . . . 14

3.4 OSC . . . 15

4 Implementation 16

4.1 Architecture . . . 16

4.2 MIDI Parsing . . . 19

4.3 Internal Musical Sequence Representation . . . 20

4.4 Generators . . . 21

4.4.1 Markov Chains Music Generator . . . 21

4.4.2 Magenta MusicRNN Music Generator . . . 23

4.5 Integration with Sonic Pi 24

5 Running the Application **27**

6 Summary **31**

Bibliography **32**

Appendices **35**

Appendix 1 - Non-exclusive licence for reproduction and publication of a graduation thesis **35**

Appendix 2 - Sonic Pi Demo Endpoint **36**

List of Figures

1 *Sonic Pi GUI* 4
2 *Live coding with Tidal Cycles* 4
3 *Orca programming* 5
4 *Max GUI* 6

5 *MIDI notes representation* 10
6 *Twinkle, Twinkle Little Star Score* 11
7 *First-order Markov chain based on the "Twinkle, Twinkle Little Star" note sequence* 12

8 *Architecture* 16
9 *Sequential Mode Flow Chart* 17
10 *Dialogue Mode Flow Chart* 18

11 *MIDI file of Arvo Pärt's "Spiegel im Spiegel"* 29
12 *Sonic Pi's Cue Viewer* 29

List of Tables

1 *MIDI Message data* 9
2 *MIDI Message types* 10
3 *Magenta MusicRNN Checkpoints* 14
4 *OSC Message Example* 15

1. Introduction

Algorithmic music composition was pioneered by Iannis Xenakis in the 1963 work "Formalized Music: Thought and Mathematics in Composition" [1]. In his work Xenakis describes the process of stochastic music composition using Markov chains. The stochastic approach was later extended by musical grammars and combinatorics in David Cope's system Emmy [2].

As an alternative to Markov chains, recurrent neural networks (RNNs) were first used for music generation in the late 80's and were updated with "long short term memory" (LSTMs) cells in Douglas Eck's 2002 work "Finding Temporal Structure in Music: Blues Improvisation with LSTM Recurrent Networks" [3].

Live coding is a type of performance where the performer creates music by programming loops and synthesizers in real-time as the composition plays. Experiments with live coded performances began in the early 2000's. Alex McLean and Adrian Ward formed the band Slub in 2000, and are considered to be among the pioneers of live coded music performances, using their own software that later led to the development of Tidal Cycles [4]. Additionally, Alex McLean together with Nick Collins were responsible for starting the "algorave" movement, by organizing a series of events where people danced to live coded music [5]. In 2012 Sam Aaron released the live coding platform Sonic Pi that was also designed for educational purposes [6].

The main goal of this project is to combine algorithmic music composition and live coding in a collaborative way, by establishing communication between live coding platforms and music generation software in real-time. Similar to multiple band members improvising together, the generator and the live coding platform should be able to exchange messages between each other and develop a common musical theme. Additionally, the software should be accessible to anyone who wants to experiment with machine learning in a live coding context, without having to implement the algorithms from scratch.

The application should be able to generate variations from MIDI (Musical Instrument Digital Interface) sources and sequences coming directly from Sonic Pi. Multiple modes of interaction should be supported by the application, including pushing and polling sequences

to and from Sonic Pi, utilizing the OSC (Open Sound Control) protocol. The technology stack should have a rich ecosystem with libraries that support the MIDI standard and the OSC protocol. Additionally, the technology stack should provide static typing for internal musical sequence representation and efficient development process (by spotting compilation errors at time of development). Modularity and abstraction should be taken into account to in order to make the solution flexible and extensible for future developments.

2. Background

In this chapter prior work on algorithmic music composition and live coding platforms is listed, highlighting the relevant functionalities and disadvantages. Additionally, similar projects are presented in the last section.

2.1 Live Coding

In this section an overview of the relevant live coding platforms will be given.

2.1.1 SuperCollider

SuperCollider [7] is an open-source software, dynamic programming language and environment used for real-time audio synthesis and algorithmic composition, written in C++ [8].

Released in 1996 by James McCartney, SuperCollider has evolved into a framework for algorithmic music and live coding, among other things.

2.1.2 Sonic Pi

Sonic Pi [9] is a live coding environment based on Ruby [10], developed by Sam Aaron. The main goal of Sonic Pi is to facilitate the teaching of programming within schools. The platform also enables musicians to deliver virtuosic live coded music performances [6].

Sonic Pi offers the possibility to make music by writing and modifying code live, similarly to playing an instrument. The interface is intuitive and consists of a code editor, some controls, a log viewer and help system (see Figure 1).

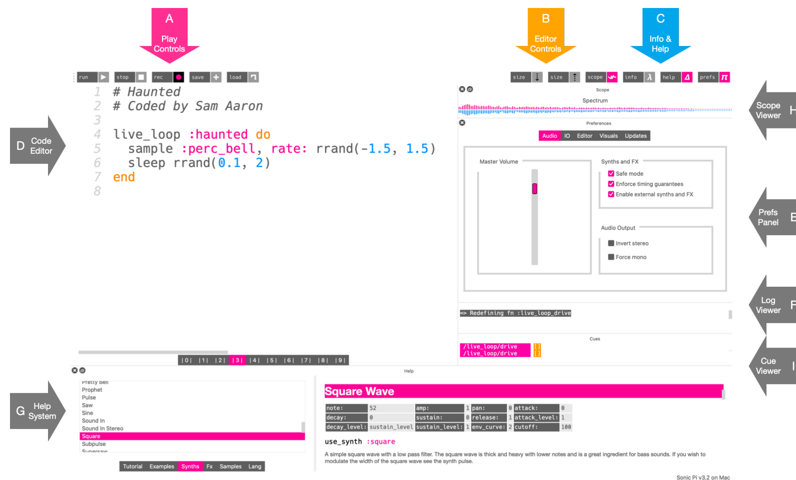

Figure 1. *Sonic Pi GUI*

The main building blocks in Sonic Pi are *live loops*. In essence, live loops are separate threads that concurrently execute code specific to an instrument in a musical piece. Sonic Pi offers the possibility of syncing loops and sharing states across loops in a thread-safe and deterministic manner. SuperCollider is used by Sonic Pi as a synthesis engine [11].

Sonic Pi does not offer out of the box machine learning functionality.

2.1.3 Tidal Cycles

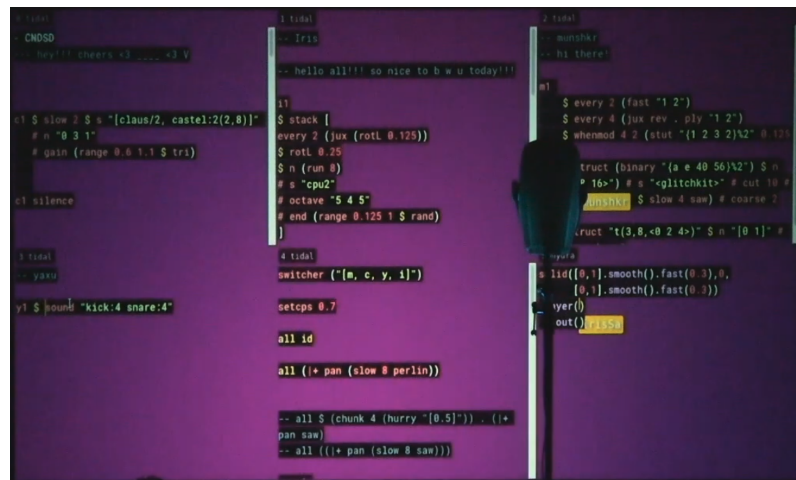

Figure 2. *Live coding with Tidal Cycles*

Tidal Cycles [12] is a live coding environment and a DSL (Domain-specific language) written in Haskell [13, 14]. Tidal Cycles does not synthesize sound by itself, but requires SuperCollider.

While there exist AI integrations with Tidal Cycles like Cibo [15] (An Autonomous Tidal Cycles performer that takes Tidal Cycles code as input and produces Tidal Cycles code as

output), Tidal Cycles itself does not have built-in AI functionality.

2.1.4 Orca

Figure 3. *Orca programming*

Orca [16] is an esoteric programming language and live coding environment. Similarly to Tidal Cycles, Orca it not producing sound, but provides an interface for live coding.

Some interesting examples of using Orca include Sonic Pi control [17], utilizing the OSC protocol, but no AI functionality is included.

2.2 Algorithmic Music Composition

In this section an overview of the relevant algorithmic music composition software will be given.

2.2.1 Magenta

Magenta [18] is a research project by Google AI, exploring the role of ML (Machine Learning) in creative processes. In particular, the library `@magenta/music` offers a number of trained machine learning models for musical sequences. The models are based on LSTM networks and are best at continuing musical sequences, provided as inputs.

2.2.2 Max

Max [19] is a visual programming language for music and multimedia [20]. Max is modular and extensible and, essentially, is a set of shared libraries.

Max includes built-in support for generating music with Markov chains, but involves a

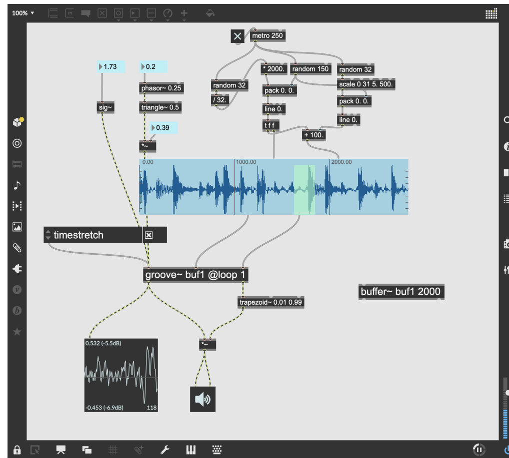

Figure 4. *Max GUI*

steep learning curve to get started and building an integration with a live coding platform.

2.3 Related Projects

Cibo [15] is an autonomous Tidal Cycles performer that is implemented as sequence-to-sequence neural net algorithm to generate Tidal Cycles code. The algorithm takes Tidal Cycles code as input and generates Tidal Cycles code as output. In comparison with the current project, this software is Tidal Cycles specific and not generic enough.

Live Coding with Machine Learning [21] is a project that integrates Magenta's Drum RNN with Sonic Pi in order to generate extensions of drum patterns. First and foremost, the project is based only on drum patterns - note sequences are not supported. Additionally, this project does not support MIDI sources and flexibility in terms of different modes of sending data to the live coding platform. Furthermore, it is neither modular nor extensible. The author would like to acknowledge this project as a kind of prototype for exchanging musical patterns between machine learning software and Sonic Pi.

3. Methodology

On a high level, the application should be able to generate variations on melodies from MIDI sources and from sequences coming directly from Sonic Pi.

The following algorithm would be applied to all MIDI inputs:

1. MIDI files would be parsed into an internal representation of note sequences, which in turn would be passed to the generator. The generator would then produce variations on these sequences based on nth-order Markov chains, Magenta MusicRNN, or other possible implementations.
2. The generated output would be transformed back to MIDI and stored in an internal structure.
3. The application would in turn transmit the selected data to Sonic Pi, utilizing the OSC protocol.

In case of data coming directly from Sonic Pi, the following algorithm would be applied:

1. Incoming sequences would be parsed into internal representation of note sequences, which in turn would be passed to the generator, that would produce variations on these sequences based on nth-order Markov chains, Magenta MusicRNN, or other possible implementations.
2. The application would in turn transmit the generated data to Sonic Pi, utilizing the OSC protocol.

In this chapter the main tooling is presented, highlighting MIDI, Markov chains, Magenta MusicRNN and OSC.

3.1 MIDI

MIDI (Musical Instrument Digital Interface) [22] is standard providing a way for interchanging time-stamped data between electronic musical instruments, computers and different musical software.

3.1.1 MIDI Files

MIDI files contain streams of MIDI events, with time information for each event. Song, sequence, track structures, tempo and time signature information are among the supported information that can be encoded in MIDI events. MIDI events consist of time information and MIDI messages that are represented as 8-bit binary data streams.

MIDI files are made up of chunks. Each chunk consists of 4-character type and 32-bit length, indicating the number of bytes in a chunk. Header chunks and tracks chunks are the two types of chunks present in a MIDI file. A MIDI file always starts with a header chunk, followed by one or more track chunks, as follows:

```
MThd  [length of header data]
[header data]
MTrk  [length of track data]
[track data]
MTrk  [length of track data]
[track data]
...
```

3.1.2 Header Chunks

The header chunk contains information about the file type, number of tracks and measurement of delta-times.

The header chunk can be represented and broken down as follows:

```
 4D 54 68 64   00 00 00 06    ff ff     nn nn      dd dd
[chunk type]    [length]    [format]  [ntrks]  [division]
```

- `[chunk type]` - represents the four ASCII characters `MThd`
- `[length]` - 32-bit representation of the number 6

- `[format]` - represents the file format, there are three such formats:
  0. single track
  1. multiple tracks, synchronous (start at the same time)
  2. multiple tracks, asynchronous (do no start at the same time)
- `[ntrks]` - represents the number of tracks in the midi file
- `[division]` - represents the meaning of delta-times. Delta-times are defined as "ticks" per quarter note. For example, if the division is 96, the eighth note between two events would be 48.

3.1.3 Track Chunks

Track chunks store the actual song data. In essence, track chunks are streams of MIDI events that consist of delta-times and MIDI messages.

The track chunk can be represented and broken down as follows:

```
 4D 54 72 6B   xx xx xx xx        ...
[chunk type]    [length]    [MIDI event]+
```

- `[chunk type]` - represents the four ASCII characters `MTrk`
- `[length]` - represents the length of the track in bytes
- `[MIDI event]+` - represents one or more MIDI events

3.1.4 Messages

A MIDI message in most cases is a set of 3 bytes that is interpreted in 4 pieces of information (see Table 1).

Table 1. *MIDI Message data*

Name	**Length and Range**	**Description**
Status byte	0-15	Type of MIDI message
Channel byte	0-16	MIDI Channel Number
Data byte 1	0-255	First MIDI message data byte
Data byte 2	0-255	Second MIDI message data byte

The message tells the MIDI gear to perform certain types of actions like play a note, change the volume, add effects, and other types of actions. MIDI messages come in two types:

channel messages and meta messages. Channel messages present information related to the control of the musical instrument like the current note playing or synthesizer program change. There are 16 possible MIDI channels (0–15). Meta messages present information like instrument name, copyright and lyrics. Here are some common MIDI channel message types:

Table 2. *MIDI Message types*

Status	**Expected Data**	**Comments**
8x	channel, note, velocity	Note Off
9x	channel, note, velocity	Note On (velocity 0 = note off)
Ax	channel, note, value	Polyphonic pressure
Bx	channel, controller, value	Controller change
Cx	channel, program	Program change (instrument)
Dx	channel, value	Channel pressure
Ex	channel, value	Pitch bend

In the context of this project only Note On and Note Off messages need to be extracted. Both Note On and Note Off messages require note and velocity values that can be represented in the 0–127 range (see Figure 5).

Octave						**Notes**						
Number	**C**	**C#**	**D**	**D#**	**E**	**F**	**F#**	**G**	**G#**	**A**	**A#**	**B**
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127	–	–	–	–

Figure 5. *MIDI notes representation*

3.1.5 Example MIDI File

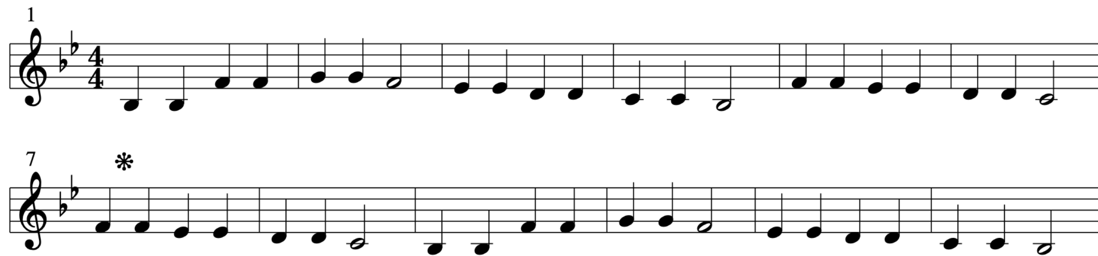

Figure 6. *Twinkle, Twinkle Little Star Score*

As an example, the MIDI representation of the first measure of "Twinkle, Twinkle Little Star" (see Figure 6) is shown below.

Assuming the MIDI file would have only one track, the representation in hexadecimal would be as follows.

Header chunk:

`4D 54 68 64`	MThd
`00 00 00 06`	chunk length
`00 00`	file format 0 (single track)
`00 01`	one track in the file
`03 C0`	960 delta ticks per quarter note

Track chunk:

`4D 54 72 6B`	MTrk
`00 00 00 27`	length of the track
`00 91 3A 4C`	delta-time 0, Note On
`87 40 81 3A 00`	delta-time 960, Note Off
`00 91 3A 55`	delta-time 0, Note On
`87 40 81 3A 00`	delta-time 960, Note Off
`00 91 41 61`	delta-time 0, Note On
`87 40 81 41 00`	delta-time 960, Note Off
`00 91 41 53`	delta-time 0, Note On
`87 40 81 41 00`	delta-time 960, Note Off
`FF 2F 00`	end of track

3.2 Markov Chains

In probability theory, an event is a set of outcomes of a trial, where each outcome is assigned a probability [23].

If an outcome of an event does not depend on some other event, these two events are considered to be independent. For example, the event of getting heads the first time a coin is flipped and the event of getting heads the second time are independent. In contrast to independent events, dependent events take into account other events that might influence the outcome of the current event. The event of getting two heads in a row after the second time a coin is flipped is dependent on the result of the first flip.

A Markov chain is a stochastic process that satisfies the Markov assumption, in which the current state depends on only a finite fixed number of previous states [24]. The order of a Markov chain specifies the number of previous states that the current state depends on. For example the probability distribution for a first-order Markov chain considers only the previous state $P(X_t|X_{t-1})$, while the second-order Markov chain takes into account previous two states $P(X_t|X_{t-1}, X_{t-2})$.

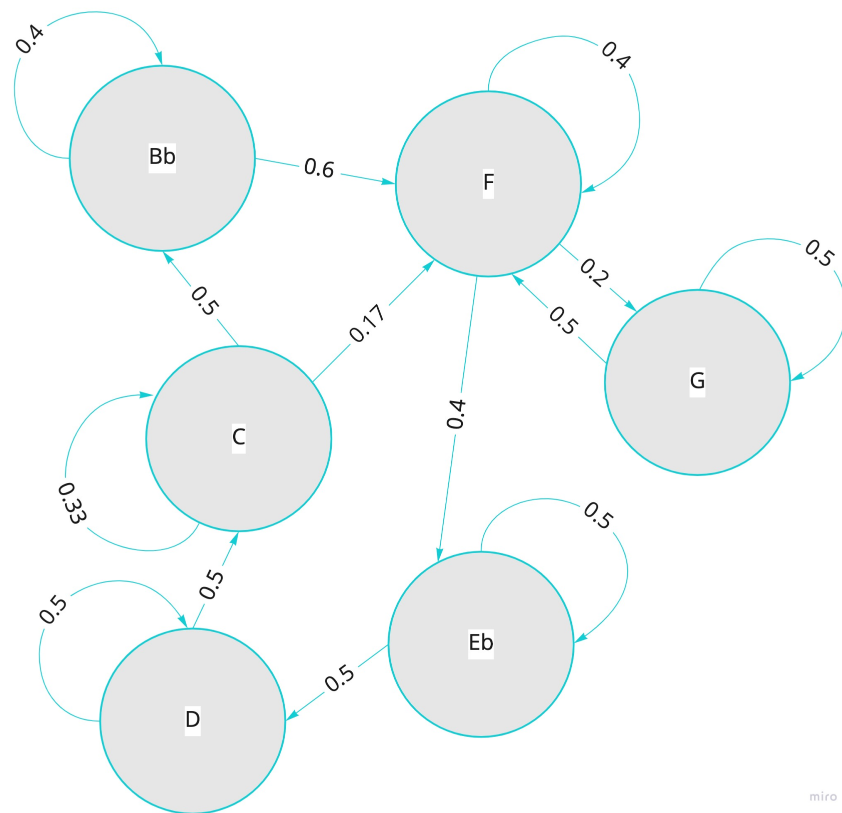

Figure 7. *First-order Markov chain based on the "Twinkle, Twinkle Little Star" note sequence*

3.2.1 Music Generation with Markov chains

Markov chains can be used to describe sequences of possible events. The Markov chains can then be used to generate new sequences of events by sampling from the probability distributions of consecutive events.

As an example, we can analyze the complete sequence of notes in the "Twinkle, Twinkle Little Star" melody (see Figure 6). The first-order Markov chain based on the melody can be constructed as seen in Figure 7. The Markov chain can also be represented using the following transition matrix:

current	Bb	F	G	Eb	D	C
Bb	0.4	0.6	0	0	0	0
F	0	0.4	0.2	0.4	0	0
G	0	0.5	0.5	0	0	0
Eb	0	0	0	0.5	0.5	0
D	0	0	0	0	0.5	0.5
C	0.5	0.17	0	0	0	0.33

Each row in the transition matrix above represents the probability distribution of one note's transition to the next one, adding up to 100%. For example, the note B♭ transitions to the note B♭ 25% of the time and transitions to F 75% of the time.

Similarly, the second-order Markov chain based on the "Twinkle, Twinkle Little Star" sequence can be represented as the following transition matrix:

current	F	G	Eb	D	C	Bb
BbBb	1	0	0	0	0	0
BbF	1	0	0	0	0	0
FF	0	0.5	0.5	0	0	0
FG	0	1	0	0	0	0
GG	1	0	0	0	0	0
GF	0	0	1	0	0	0
FEb	0	0	1	0	0	0
EbEb	0	0	0	1	0	0
EbD	0	0	0	1	0	0
DD	0	0	0	0	1	0
DC	0.25	0	0	0	0.5	0.25
CC	0	0	0	0	0	1
CBb	0.5	0	0	0	0	0.5
CF	1	0	0	0	0	0

Given a Markov chain, a new musical sequence can be constructed by choosing a random seed note at the start, and then sampling the next note from the probability distribution of notes at each stage. This process should be repeated until the specified piece duration is exceeded [25].

3.3 Magenta MusicRNN

Magenta MusicRNN is an LSTM-based language model for musical notes. It it used for continuing note sequences that are given as input [26].

The Music RNN can be configured by the number of steps and the "temperature" - the higher the temperature, the more random and less like the input the output sequence will be.

Similar to MIDI, Magenta MusicRNN uses an abstract representation of musical sequences as a series of notes with pitches, durations and velocities [27]. The model expects *quantized* sequences as inputs - meaning that the durations are defined as steps (much like delta-times in MIDI). The durations of steps can be configured per quarter note.

The Magneta MusicRNN exposes a number of checkpoints (different snapshots of the model [28]):

Table 3. *Magenta MusicRNN Checkpoints*

ID	**Description**
`basic_rnn`	36-class MelodyRNN model
`melody_rnn`	128-class MelodyRNN model
`drum_kit_rnn`	9-class DrumsRNN model
`chord_pitches_improv`	36-class melody model conditioned on chords

Compared to Markov chains that generate variations on a particular sequence (by preserving similar melodic patterns to the source sequence), the Music RNN produces extensions of the sequence.

3.4 OSC

OSC (Open Sound Control) is a messaged-based protocol used by computers and synthesizers to exchange musical and control information [29].

OSC data is constructed using following data types:

- *int32* - signed 32-bit big endian integer
- *float32* - 32-bit big endian IEEE 754 floating point number
- *OSC-timetag* - 64-bit big endian timestamp
- *OSC-string* - a sequence of ASCII characters
- *OSC-blob* - an 32-bit integer size count followed by arbitrary binary data

The OSC data is usually sent in packets, utilizing the UDP protocol. Packets contain OSC messages or OSC bundles. An OSC message contains an address that can be pattern-matched, OSC type tag and arguments corresponding to the specified type. An OSC bundle consist of OSC-string `#bundle` followed by an OSC-timetag and zero or more OSC bundle elements, that consist of size and contents. The contents of an OSC element can be an OSC message or an OSC bundle.

Due to it's simplicity and flexibility OSC is supported by many applications, including Sonic Pi, Tidal Cycles and Max.

As an example, the OSC message containing the first four note pitches of "Twinkle, Twinkle Little Star" (see Figure 6) can be represented as follows:

Table 4. *OSC Message Example*

Property	**Value**	**Comments**
Address	`/hello/world`	address path
Argument types	`",iiii"`	four *int32* arguments
Arguments	`58, 58, 65, 65`	four note pitches

4. Implementation

In this chapter the implementation details are presented, by highlighting the main parts: architecture, MIDI parsing, internal note sequence representation, music generator implementations and synchronization with Sonic Pi.

4.1 Architecture

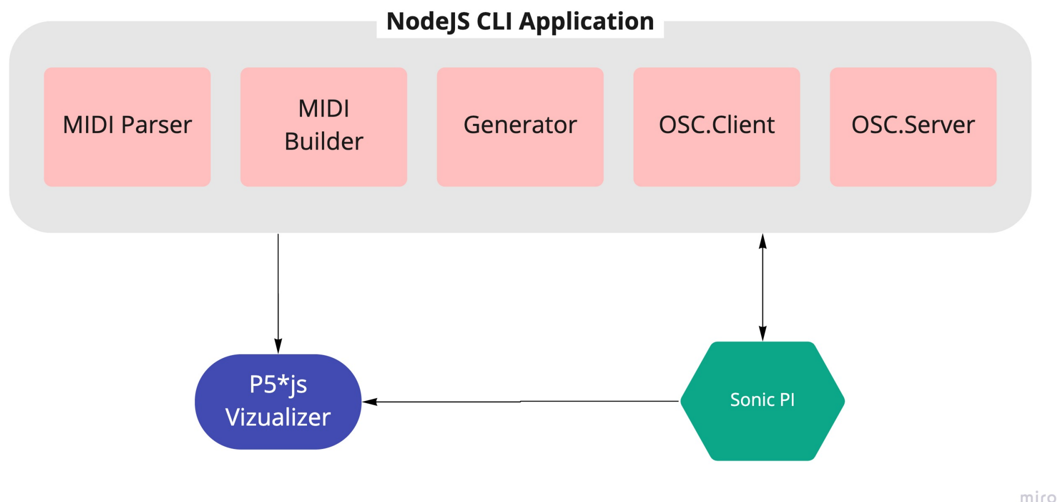

Figure 8. *Architecture*

The main building blocks of the application are defined as follows:

- *MIDI Parser* - handles parsing of MIDI files into internal MIDI representation
- *MIDI Builder* - handles building the result of the music generation process into a MIDI file
- *Generator* - handles the music generation process based on source sequences
- *OSC Client* - handles sending OSC messages to the specified endpoint
- *OSC Server* - handles incoming OSC messages
- *NodeJS [30] CLI Application* - handles the main logic of the application and connects all the components above
- *P5.js [31] Visualizer* - handles visualization of note sequences in the browser

The application has 3 modes that can be defined as follows:

- `DIALOGUE`
- `MIDI`
- `SEQUENTIAL`

In the `DIALOGUE` mode the messages are exchanged between the application's OSC Server and Sonic Pi's OSC listener in a request/response loop. Sonic Pi makes a request containing the source sequence and gets an immediate response from the application with the generated variation based on the source sequence. The sequences are sent as complete units, utilizing the application's internal musical notes representation, containing pitches and durations.

In the `MIDI` and the `SEQUENTIAL` modes the application pushes the messages to Sonic Pi, while Sonic Pi's OSC listener is waiting for incoming messages. The generated sequences are based on MIDI source and the main difference between `MIDI` and `SEQUENTIAL` modes is that in the former sequences are being sent as complete units (same as in `DIALOGUE` mode), while in the latter sequences are sent as separate notes, containing only pitch information and the application thread is being suspended for the duration of the specific note.

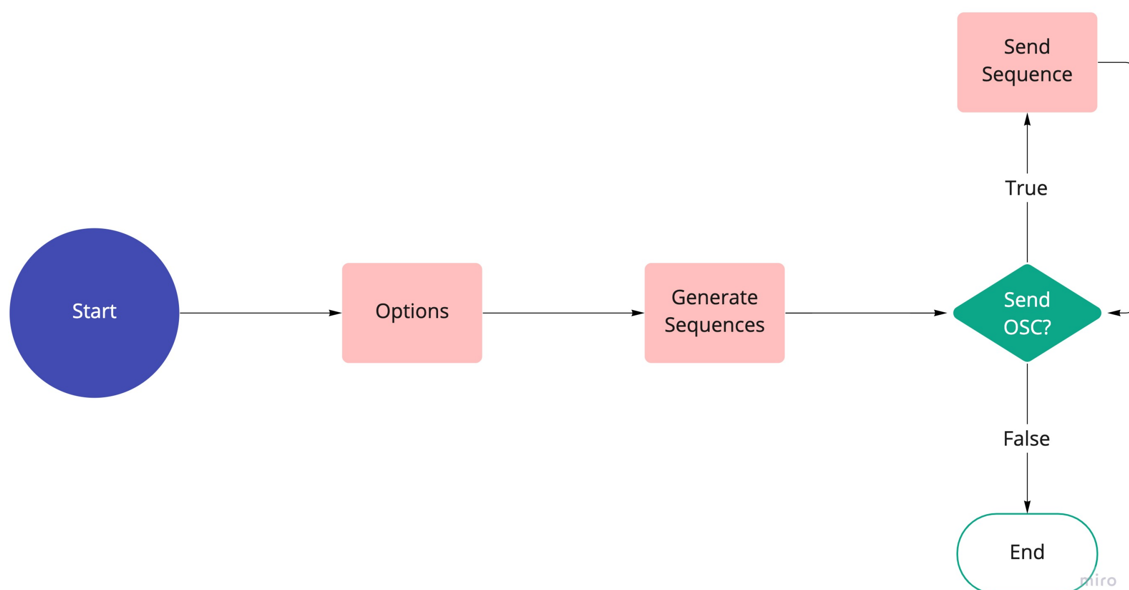

Figure 9. *Sequential Mode Flow Chart*

In case of the `MIDI` and the `SEQUENTIAL` modes, the flow (see Figure 9) can be described as follows:

1. At start of the application the user is asked to provide some options, including music generator implementation, MIDI source path, output path and number of outputs
2. Sequences are generated based on the options provided in the previous step
3. The user is presented with an option to send sequences via OSC
   - Sequences are sent to Sonic Pi and step 3 is repeated, in case of confirmation
   - Otherwise, the application exits

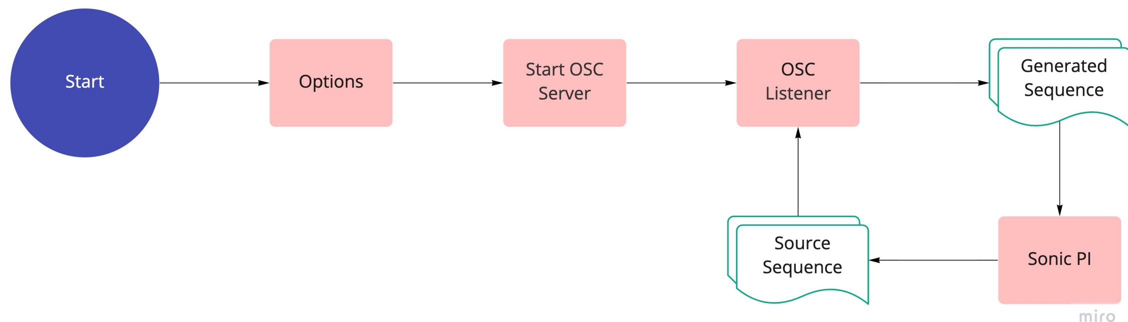

Figure 10. *Dialogue Mode Flow Chart*

In `DIALOGUE` mode the flow (see Figure 10) can can be described as follows:

1. At the start of the application the user is asked to provide music generator implementation options
2. The OSC Server is started and begins listening for incoming messages
3. Sequences are generated based on the incoming data and sent back to Sonic Pi
4. The process is repeated until the application is terminated

4.2 MIDI Parsing

The current MIDI parser implementation is a modification of the `midi-parser-js` library [32], that enables to parse the MIDI file into the internal MIDI representation.

As an example, the raw data of the `twinkle_twinkle.midi` that is the MIDI representation of the first bar of "Twinkle, Twinkle Little Star" (see Figure 6):

```
4D 54 68 64 00 00 00 06
00 00 00 01 03 C0 4D 54
72 6B 00 00 00 27 00 91
3A 4C 87 40 81 3A 00 00
91 3A 55 87 40 81 3A 00
91 41 61 87 40 81 41 00
00 91 41 53 87 40 81 41
00 FF 2F 00
```

would be parsed into the following internal JSON representation of MIDI:

```
{
  format: 0, // Single Track File Format
  ntrks: 1,
  division: { ticksPerBeat: 960 },
  tracks: [
    [
      [ 0, { channel: 1, note: 58, velocity: 76 } ],
      [ 960, { channel: 1, note: 58, velocity: 0 } ],
      [ 0, { channel: 1, note: 58, velocity: 85 } ],
      [ 960, { channel: 1, note: 58, velocity: 0 } ],
      [ 0, { channel: 1, note: 65, velocity: 97 } ],
      [ 960, { channel: 1, note: 65, velocity: 0 } ],
      [ 0, { channel: 1, note: 65, velocity: 83 } ],
      [ 960, { channel: 1, note: 65, velocity: 0 } ],
      [ 1, {} ] // End of Track
    ]
  ]
}
```

4.3 Internal Musical Sequence Representation

The application defines an abstract representation of note sequences that will be used by the algorithmic music composition implementations.

Similar to MIDI format, notes are represented as pitch in the 0–127 range and duration in terms of steps. The resolution of steps is defined as steps per quarter note. Additionally, a tempo representation is defined, indicating the number of beats (quarter notes) per minute.

For example, the first bar of "Twinkle, Twinkle Little Star" (see Figure 6) would be represented as follows:

```
{
  quantization: { stepsPerQuater: 960 },
  tempo: { bpm: 120 },
  notes: [
    [ 58, 960 ],
    [ 58, 960 ],
    [ 65, 960 ],
    [ 65, 960 ]
  ]
}
```

4.4 Generators

This section presents the *Generator* interface for algorithmic music composition and implementations based on two strategies: Markov chains and Magenta MusicRNN.

In order to achieve modularity in the application, the following Typescript [33] interface for algorithmic music composition is defined that takes the internal representation of musical sequence as input and returns the same representation as output:

```
export interface Generator {
    /**
     * Generator interface for algorithmic music composition.
     * @param  {Sequence} input Input Sequence
     * @return {Sequence}       Output Sequence
     */
    generate(input: Sequence): Promise<Sequence>;
}
```

4.4.1 Markov Chains Music Generator

The implementation has following parameters:

- *steps* - the number of notes to be generated
- *order* - the order of the Markov chain

The initial step is to build a transition graph that can be used for generating music at a later stage. Internally, an adjacency list is used to represent the graph. Keys in the adjacency list are notes and have the format `pitch:steps`.

The first-order Markov chain for the note sequence in "Twinkle, Twinkle Little Star" (see Figure 6) has the following transition graph:

```
Map(9) {
  '58:960' => Map(2) { '58:960' => 2, '65:960' => 2 },
  '65:960' => Map(3) { '65:960' => 4, '67:960' => 2, '63:960' => 2 },
  '67:960' => Map(2) { '67:960' => 2, '65:1920' => 2 },
  '65:1920' => Map(1) { '63:960' => 2 },
  '63:960' => Map(2) { '63:960' => 4, '62:960' => 4 },
  '62:960' => Map(3) { '62:960' => 4, '60:480' => 2, '60:1920' => 2 },
  '60:480' => Map(2) { '60:480' => 2, '58:1920' => 2 },
  '58:1920' => Map(1) { '65:960' => 1 },
  '60:1920' => Map(2) { '65:960' => 1, '58:960' => 1 }
}
```

Each row in the graph, represented as an adjacency list corresponds to the transitions distribution of a particular note. For example, the note `58:960` transitions to the note `58:960` two times, and two times to the note `65:960`.

The graph above can be represented as a transition matrix as follows (please note that the probabilities in this example are not normalized - not summing up to 1):

current	58:960	65:960	67:960	63:960	65:1920	62:960	60:480	60:1920	58:1920
58:960	2	2	0	0	0	0	0	0	0
65:960	0	4	2	2	0	0	0	0	0
67:960	0	0	2	0	2	0	0	0	0
65:1920	0	0	0	2	0	0	0	0	0
63:960	0	0	0	4	0	4	0	0	0
62:960	0	0	0	0	0	4	2	2	0
60:480	0	0	0	0	0	0	2	0	2
58:1920	0	1	0	0	0	0	0	0	0
60:1920	1	1	0	0	0	0	0	0	0

The next and final step is to generate the sequence, based on the transitions graph, constructed in the previous step.

- A random seed note is picked at the start
- Then at each stage the next note is randomly chosen from the transitions distribution corresponding to the current note, taking into account the weights of each transition (for example the note `65:960` from the second row would be chosen with a probability of 50%)
- Finally, the next note is returned, while making a recursive call to continue the process
- The process is repeated until steps is equal to 0

4.4.2 Magenta MusicRNN Music Generator

The Magenta MusicRNN Generator implementation utilizes the `chord_pitches_improv` checkpoint and has following parameters:

- *steps* - the number of quantized steps to be generated
- *temperature* - the amount of randomness in the result. Any value above 0 is accepted and anything above 1.5 will result in random (less like the input) results
- *chordProgression* - represents the chord progression the result should be based on

The current implementation is a wrapper around the MusicRNN model class, provided by the `@magenta/music` library. Additional mapping is implemented between the application's internal musical sequence representation and Magenta's note sequence representation.

The model requires a quantized sequence as input. The first bar of the "Twinkle, Twinkle Little Star" melody (see Figure 6) would be represented as follows:

```
{
    timeSignatures: [ e { numerator: 4, denominator: 4, time: 0 } ],
    keySignatures: [],
    tempos: [ e { qpm: 120, time: 0 } ],
    notes: [
      e {
        pitch: 58,
        startTime: 0,
        endTime: 0.5,
        quantizedStartStep: 0,
        quantizedEndStep: 4
      },
      e {
        pitch: 58,
        startTime: 0.5,
        endTime: 1,
        quantizedStartStep: 4,
        quantizedEndStep: 8
      },
      e {
        pitch: 65,
        startTime: 1,
        endTime: 1.5,
        quantizedStartStep: 8,
        quantizedEndStep: 12
      },
      e {
        pitch: 65,
        startTime: 1.5,
        endTime: 2,
        quantizedStartStep: 12,
        quantizedEndStep: 16
      }
    ],
    totalTime: 2,
    quantizationInfo: e { stepsPerQuarter: 4 },
    totalQuantizedSteps: 16
}
```

The quantized sequence would in turn be passed to the `continueSequence` method, exposed by the MusicRNN model class. For `twinkle_twinkle.midi` following parameters could be used:

```
steps: 100
temperature: 1
chordProgression: ["Bb", "Eb", "Bb", "Eb", "Bb", "F7", "Bb"]
```

An extended sequence would be returned by the model as output.

4.5 Integration with Sonic Pi

The following section describes the process of integration between the application and Sonic Pi. In a live coding context the following cases might be encountered when it comes to synchronization with the platform:

1. no current live loops are playing and there is no necessity to sync the generated sequence with the beat/metronome (for example, the generated sequence is only used as a source for sound synthesis)
2. the generated sequence needs to be synced with the current beat/metronome in context of an ongoing musical piece

In the first case, the application mode `SEQUENTIAL` can be used. Utilizing the OSC Client, the notes are sent as OSC messages in real-time, while suspending the current thread in between for the duration of the note. For example, in case of *120 bpm*, the suspension time can be calculated as follows:

$$(quantizedSteps/stepsPerQuater) * 500\ milliseconds$$

In the second case, generated sequences can be sent as two OSC messages: pitches and steps (the application modes `MIDI` and `DIALOGUE` implement sending the sequences as a two messages). Additionally, the live loop, playing the generated sequence is synced with the beat/metronome.

Sonic Pi's *Time State* structure is utilized to share information between live loops in a thread-safe and deterministic way [34]. Time State provides two methods for persisting and retrieving information from the store: `set` and `get`. Time State is also used internally by Sonic Pi for working with OSC data, allowing Sonic Pi to send and receive OSC messages

without any configuration. Example of an OSC listener is presented below:

```
seq = sync "/osc*/gen/sequence"
```

The `/gen/sequence` path is a path Sonic Pi uses for the `sync` listener of incoming OSC messages. The `osc` prefix is added by Sonic Pi for all incoming messages. Execution of the following code is blocked by the listener.

Considering the blocking behaviour of the OSC listeners, the live loop for incoming generated sequences can be defined as follows:

```
live_loop :receive_sequence do
  use_real_time
  seq = sync "/osc*/gen/sequence"
  steps = sync "/osc*/gen/steps"
  set :sequence, seq.zip(steps)
end
```

In the loop above, the program is listening for incoming pitches and steps and storing the data, utilizing Time State store.

Next, a live loop is defined that is acting as a metronome:

```
live_loop :beat do
  sample :drum_bass_soft
  sleep 1
end
```

As a final step in this synchronization example, a live loop is added that plays the generated sequence and is in sync with the metronome:

```
live_loop :play_gen_sequence, sync: :beat do
  use_synth :piano
  notes = get[:sequence] || []
  puts notes

  if notes.empty?
    sleep 1
  else
    notes.each do |note, step|
      play note, release: 1, amp: 0.4, sustain: 0.5
      sleep step
    end
  end
end
```

Additionally, the live loop above fetches and plays the next sequence only when the current sequence is exhausted, being in phase with the metronome. Furthermore, the performer can add more live loops to continue developing the musical theme.

Besides making sure the live loops are in phase with each other, the duration of the notes must be a factor of one whole note that is also a power of 2 (since musical notation is based around the powers of 2 - 1 (whole note), 1/2, 1/4, 1/8, 1/16 etc). To achieve this the application does additional quantization of the steps, by rounding the steps to the nearest value from the grid of acceptable values. The resulting values are then normalized to comply with Sonic Pi's duration of sleep (for example, `sleep 1` - refers to sleep for one beat).

5. Running the Application

In this chapter an example on how to generate variations based on MIDI input will be presented.

The first step would be to copy the code from Sonic Pi Demo Endpoint (see Appendix 2 6) to an empty buffer in Sonic Pi. The code includes the following:

1. a live loop that is listening for OSC messages containing the generated sequences
2. a live loop that is acting as a metronome
3. a live loop that is playing the piano part
4. a live loop that is playing the generated sequences

After pressing `Run` in Sonic Pi, the piano part should start playing. The next step is to start the application.

At the start of the application, the user is presented with multiple modes to choose from. The mode `MIDI` will be chosen in this example:

```
 ____                              ___
/\  _`\   __                      /\_ \
\ \ \L\ \/\_\    ___    ___    ___\//\ \     ___
 \ \ ,__/\/\ \  /'___\ /'___\ / __`\\ \ \   / __`\
  \ \ \/  \ \ \/\ \__//\ \__//\ \L\ \\_\ \_/\ \L\ \
   \ \_\   \ \_\ \____\ \____\ \____//\____\ \____/
    \/_/    \/_/\/____/\/____/\/___/ \/____/\/___/


? Choose application mode ...
> MIDI
  DIALOGUE
  SEQUENTIAL
```

Next, some options need to be provided including the relative path of MIDI input, the location of the output files, the number of outputs and name of output file. In this example the file `spiegel.midi` is chosen (this file can be found in the `midi/` folder in the root directory of the project repository).

```
? Please provide the following information
*                Source : midi/spiegel.midi
*                   Out : midi_out/
*         No. of outputs : 5
*   Name of output file : test_melody
```

Afterwards the user will be presented with the music generator choice and corresponding options:

```
? Choose generator type ...
> Markov Chain
  Magenta MusicRNN
```

```
*     The order of the Markov chain : 2
*   Number of steps to be generated : 100
```

If the MIDI file has more than one track, the user will be presented with the following question:

```
? The provided MIDI track includes multiple tracks.
Enter source track number: > 0
```

By examining the `spiegel.midi` (see Figure 11) file, it can be seen that the MIDI file consists of multiple tracks. Since variations on the viola melody will be generated in this example, the track number zero ("viola") needs to be chosen (the numeration of tracks starts from zero).

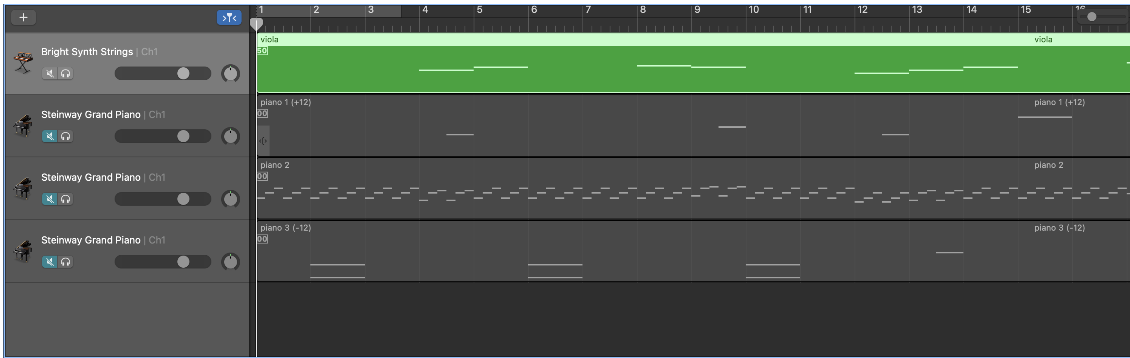

Figure 11. *MIDI file of Arvo Pärt's "Spiegel im Spiegel"*

The application will proceed with generating the sequences and the user will be presented with a choice of sequences to send via OSC after the generation finishes.

```
Send sequence via OSC? (y/N) * true
? Choose sequence ...
> midi_out/test_melody_0.midi
  midi_out/test_melody_1.midi
  midi_out/test_melody_2.midi
  midi_out/test_melody_3.midi
  midi_out/test_melody_4.midi
```

The selected sequences will then be sent to Sonic Pi and the incoming OSC messages will be seen in Sonic Pi's Cue Viewer:

```
Cues
/live_loop/rhytm  []
/live_loop/rhytm  []
/osc:127.0.0.1:52181/gen/sequence  [59.0, 57.0, 56.0, 54.0, 57.0, 56.
/osc:127.0.0.1:52181/gen/steps     [1.0, 0.25, 0.5, 2.0, 0.25, 0.25,
/set/sequence  [[59.0, 1.0], [57.0, 0.25], [56.0, 0.5], [54.0, 2.0],
/live_loop/receive_sequence  []
/live_loop/beat  []
/live_loop/rhytm  []
```

Figure 12. *Sonic Pi's Cue Viewer*

The process of choosing and sending the generated sequence to Sonic Pi can be repeated until the user types `N` in the `Send another sequence via OSC?` prompt.

Additional steps the user can do in order to experiment with the piece:

- adding more layers to the musical piece by programming additional live loops
- manipulating the generated sequence by adding effects and playing around with the ADSR (Attack, Decay, Sustain, Release) envelope
- starting another instance of the application with in a different mode or different music generator implementation

6. Summary

The main goal of this project was to combine algorithmic music composition and live coding in a collaborative way. In the process of achieving this goal prior research on existing software and related work was made, methodologies and tooling were chosen based on existing standards (like MIDI and OSC) and the application was developed with SOLID [35] principles in mind in order to be extensible and flexible for future developments.

The application was developed in NodeJS with Typescript as the backed language, P5.js library was utilized for the visualizations. MIDI standard was used for input sources. Integration between the application and Sonic Pi was built, utilizing the OSC protocol, syncing concurrent live loops and quantizing of note sequences. The Markov chains algorithmic music composition method was implemented to produce variations on source sequences. Additionally, a wrapper around the Magenta MusicRNN was implemented as the second algorithmic music composition method. The application is able to generate variations from MIDI sources and sequences sent directly from Sonic Pi in real-time, while being in sync with the live coding platform. Therefore, the main goal of this project is achieved.

Future developments could include additional implementations for algorithmic music composition (the Twelve-tone technique, for example), extension of the communication between the live coding platform and the application and making the solution more generic in order to be compatible with other live coding platforms (like Tidal Cycles).

Bibliography

[1] Iannis Xenakis. *Formalized music: thought and mathematics in composition*. 6. Pendragon Press, 1992.

[2] David Cope. “Experiments in musical intelligence (EMI): Non-linear linguistic-based composition”. In: *Journal of New Music Research* 18.1-2 (1989), pp. 117–139.

[3] Douglas Eck and Juergen Schmidhuber. “Finding temporal structure in music: Blues improvisation with LSTM recurrent networks”. In: *Proceedings of the 12th IEEE workshop on neural networks for signal processing*. IEEE. 2002, pp. 747–756.

[4] Tom Armitage. *Making music with live computer code*. 2009. URL: https://www.wired.co.uk/article/making-music-with-live-computer-code-.

[5] Rob Marvin. *Algoraves: Dancing to live coding*. 2014. URL: https://sdtimes.com/algoraves/algoraves-dancing-to-live-coding/.

[6] Sam Aaron. “Sonic Pi–performance in education, technology and art”. In: *International Journal of Performance Arts and Digital Media* 12.2 (2016), pp. 171–178.

[7] *SuperCollider*. URL: https://supercollider.github.io/.

[8] James McCartney. “SuperCollider: a new real time synthesis language”. In: *Proc. International Computer Music Conference (ICMC’96)*. 1996, pp. 257–258.

[9] *Sonic Pi*. URL: https://sonic-pi.net/.

[10] *Ruby Programming Language*. URL: https://www.ruby-lang.org/en/.

[11] Samuel Aaron, Dominic Orchard, and Alan F Blackwell. “Temporal semantics for a live coding language”. In: *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*. 2014, pp. 37–47.

[12] *Tidal Cycles*. URL: https://tidalcycles.org/.

[13] *Haskell Language*. URL: https://www.haskell.org/.

[14] Alex McLean and Geraint Wiggins. “Tidal–pattern language for the live coding of music”. In: *Proceedings of the 7th sound and music computing conference*. 2010, pp. 331–334.

[15] Jeremy Stewart and Shawn Lawson. “Cibo: An autonomous tidalCyles performer”. In: *Proceedings of the Fourth International Conference on Live Coding*. 2019, p. 353.

[16] *Orca*. URL: https://github.com/hundredrabbits/Orca.

[17] *Using Orca to control Sonic Pi with OSC*. URL: https://in-thread.sonic-pi.net/t/using-orca-to-control-sonic-pi-with-osc/2381.

[18] *Magenta*. URL: https://magenta.tensorflow.org/.

[19] *Max*. URL: https://cycling74.com/products/max.

[20] M Sheffield. “Max/MSP for Average Music Junkies”. In: *Hopes&Fears. Preuzeto* 17.9 (2015), p. 2019.

[21] *Live Coding with Machine Learning (Magenta.js)*. URL: https://in-thread.sonic-pi.net/t/live-coding-with-machine-learning-magenta-js/4462.

[22] *Standard MIDI Files 1.0*. The MIDI Manufacturers Association. Los Angeles, CA, 1996.

[23] A Leon-Garcia and A Leon-Garcia. *Probability, statistics, and random processes for electrical engineering. 2008.*

[24] Stuart Russell and Peter Norvig. “Artificial intelligence: a modern approach”. In: (2002).

[25] Darrell Conklin. “Music generation from statistical models”. In: *Proceedings of the AISB 2003 Symposium on Artificial Intelligence and Creativity in the Arts and Sciences*. Citeseer. 2003, pp. 30–35.

[26] *Making music with magenta.js*. URL: https://hello-magenta.glitch.me/#musicrnn.

[27] *A library for common manipulations of NoteSequences*. URL: https://magenta.github.io/magenta-js/music/modules/_core_sequences_.html.

[28] Lak Lakshmanan. *ML Design Pattern 2: Checkpoints*. URL: https://towardsdatascience.com/ml-design-pattern-2-checkpoints-e6ca25a4c5fe.

[29] Matt Wright. *OpenSoundControl Specification 1.0*. 2002. URL: https://opensoundcontrol.stanford.edu/spec-1_0.html.

[30] *Node.js*. URL: https://nodejs.org/en/.

[31] *p5.js*. URL: https://p5js.org/.

[32] *MidiParser.js*. URL: https://github.com/colxi/midi-parser-js.

[33] *TypeScript: JavaScript With Syntax For Types*. URL: https://www.typescriptlang.org/.

[34] *Sonic Pi Tutorial v3.2 Part 10: Time State*. URL: https://sonic-pi.net/tutorial.html#section-10.

[35] *SOLID: The First 5 Principles of Object Oriented Design | DigitalOcean*. URL: https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design.

Appendices

Appendix 1 - Non-exclusive licence for reproduction and publication of a graduation thesis [1]

I, Viktor Pavlov

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis , supervised by Edward Morehouse and Chad Nester
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - Sonic Pi Demo Endpoint

```
# Coded by Viktor Pavlov
# Piano part from Arvo Pärt's "Spiegel im Spiegel"

use_bpm 80

set :sequence, []

live_loop :receive_sequence do
  use_real_time
  seq = sync "/osc*/gen/sequence"
  s = sync "/osc*/gen/steps"
  set :sequence, seq.zip(s)
end

live_loop :metronome do
  sleep 1
end

with_fx :reverb, room: 1 do
  live_loop :piano_part, sync: :metronome do
    F = (ring :C5, :F5, :A5)
    Gm7 = (ring :As4, :F5, :G5)
    Bbmaj7 = (ring :D5, :A5, :Bb5)

    use_synth :piano
    use_synth_defaults sustain: 0.8, release: 0.2, hard: 0.1

    18.times do
      play F.tick
      sleep 1
    end
    6.times do
      play Gm7.tick
      sleep 1
    end
    24.times do
      play F.tick
      sleep 1
    end
    6.times do
      play Bbmaj7.tick
      sleep 1
    end
  end

  live_loop :play_gen_sequence, sync: :metronome do
    notes = get[:sequence] || []
    puts notes

    if notes.empty?
      sleep 1
    else
      notes.each do |note, step|
        use_synth :blade
        use_synth_defaults amp: 0.4, attack: step * 0.4, decay: step * 0.1,
          sustain: step * 0.3, release: step * 0.2, vibrato_rate: 7
        play note
        use_synth :square
        play note, amp: 0.03
        sleep step
      end
    end
  end
end
```