STUDIA UBB MUSICA, LXIII, 2, 2018 (p. 9 – 20)
(RECOMMENDED CITATION)
DOI:10.24193/subbmusica.2018.2.01

INTERACTIVE SOUND TOOLS AND ENGINES

ADRIAN BORZA[1]

SUMMARY. The compositional algorithms are recognized as solutions to music problems which appear in computer programming. They are small to large-sized patches designed for analyzing, for example, the direction, size, and the magnitude of change of a melodic interval; analyzing the rest between two MIDI events; analyzing the register, duration, delta time and dynamics of MIDI events; analyzing the number of notes of a chord; storing, limiting and sorting pitch or intensity values; processing MIDI events; making chord progressions; compositing a canon; tracking a musical MIDI score.

Keywords: Computer Music, compositional algoritm

Introduction

The practical goal of this paper is to help increase creative thinking since a musician will focus on compositional algorithms and programming practices applied in Computer Music. Therefore, we introduce several Max patches as solutions to problems which often appear in programming. These patches, named sound tools and engines, are classified according to keywords and functional categories. For example, the class *Register* refers to two programs called *range* and *range.sel*. The patch *range* identifies the pitch numbers of a MIDI event within a range of notes and the patch *range.sel* signals the pitch value within a defined register. They have in common particular keywords: pitch, range, and register.

The second objective is to assist musician to understand the concepts, terms, and practices associated with Max, and to acquaint him with a software development methodology in general, with the intent of better planning and managing his effort. However, it is assumed that the reader has accumulated knowledge of basic concepts of music theory and a rudimentary understanding of computer and software.

[1] *He is an Associate Professor at the Gheorghe Dima Music Academy, Str. Ion I.C. Brătianu, nr. 25, Cluj-Napoca 400079, Romania. Email: aborza@gmail.com*

Conventions

The names of the Max objects are displayed in colored style, like this. Messages and lines of code are displayed in Courier New bold style, **`like this`**. Additional names, filenames, and relevant concepts are displayed in italic style, *like this*. A Max patch refers to a single small- (tiny-) mid- or a large-sized program made in a *Patcher window*. The patch is saved under a filename and extension, *like.this.maxpat*. An abstraction is a patch file saved under a filename and extension, *like.this.abs.maxpat*. The abstraction is used as a Max object (i.e. object box) inside the main patch.

Musician's abstractions

We are going to present several categories of author's small-sized encapsulated programs, called musician's abstractions. They are designed in a modular practice so that the formally initiated programmer-musician and the debutant student make interactive programs in the shortest possible time.

These abstractions offer solutions to the following musical issues: analyzing the direction, size, and magnitude of the melodic interval change; analyzing the size, and magnitude of the harmonic interval change; analyzing the rest between two sound events (MIDI); analyzing the number of notes of a chord; analyzing the register, duration, delta time and dynamics of sound events performed on a MIDI keyboard; storing, limiting, transposing and sorting of pitch values; storing, limiting, and sorting of intensity values; recording and processing of sound events (MIDI) by augmentation and rhythmic diminution, by decreasing and increasing the tempo; making chord progressions; humanizing or diversifying the chord structure, with regard to the attack and the intensity of the notes; compositing of a two-voice canon, synchronized and unsynchronized; tracking and automated accompanying of musical score (MIDI).

The musician's abstractions are structured on the functional criterion, illustrating common properties and actions within structural networks. Abstractions are ordered alphabetically and are accompanied by keywords. Functional categories are Keyboard, Global Transport, Register, Score Following, Canon, Pitch, Sequencer, Transposition, Interval, Velocity, Duration and Time, Humanize, Chords and Progressions, and MIDI. The diagram contains a descriptive name and abstraction definitions. A brief description of the main abstractions is also given.

Table 1

Keyboard: interaction

kslider.interactive	Interactive Keyboard

Table 2

Global Transport

gt.control	Control the Global Transport

Table 3a

Register: pitch, range

range	Identify the pitch numbers of an event within a range
range.sel	Signal the pitch value within a register

Table 3b

The patch *range.sel* signals once, with a message of type **bang**, the pitch value of a MIDI event according to the low and high limits typed-in the number objects. The patch is restored to its original state once the pitch surpasses the limits.

A MIDI event is any note, harmonic interval or chord.

Input: raw MIDI data, **int**.

Output: **bang**.

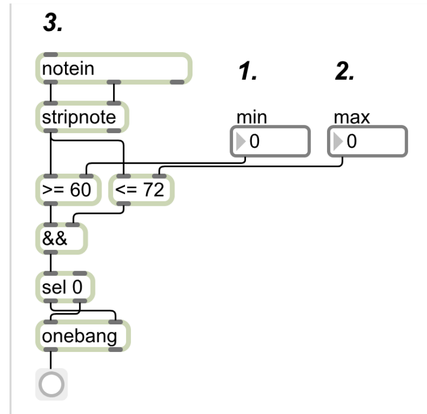


Table 4

Score Following: live performance, synchronization

follow	Listen to a live music performance
follow.sync	Listen to a live music performance and synchronize events

Table 5

Canon: quantization

canon	Make a canon of 2 voices
canon.sync	Make a canon of 2 voices quantized to a time-boundary

Table 6a

Pitch: range, threshold, velocity

pitch.past	Signal when a threshold is exceeded
pitch.sel	Signal the pitch value of an event
pitch.vel.limit	Limit the pitch or velocity numbers to the **0–127** range

Table 6b

The program *pitch.vel.limit* prevents incoming pitch and velocity values from surpassing the **0–127** range.

It also identifies and signals, with a message of type **bang**, the numbers within that range.

Input: **int**.

Output: **int**, **bang**.

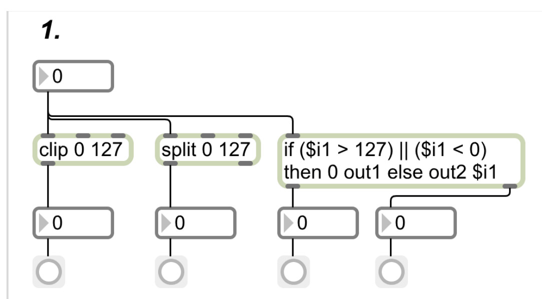


Table 7a

Sequencer: controller, keyboard

seq.control	Control the seq object
seq.ctrl	Control a sequencer using a keyboard controller and the notein
seq.ctrl.midiin	Control a sequencer using a keyboard controller and the midiin

The patch *seq.ctrl.midiin* deals with basic commands for the seq object such as recording (**record**), stop (**stop**) and playback (**start**). Pressing C4 (**72**) key on a keyboard controller, the patch records incoming MIDI events, pressing C#4 (**73**) it stops the sequencer, and pressing D4 (**74**) it plays back previously recorded data. The patch uses the midiin object for input MIDI data. Input/Output: raw MIDI data.

Table 7b

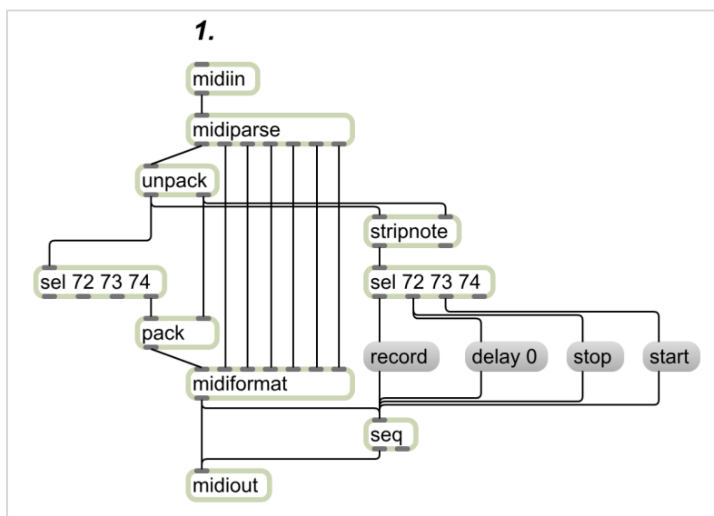


Table 8a

Transposition: pitch

transpo	Move up or down the pitch numbers of an event
transpo.abs	The abstraction of the program *transpo*
transpo.abs.init	The interface of the abstraction *transpo.abs*
transpo.gui	Move up or down the pitch numbers of an event with a constant value stored in the preset object

The patch *transpo.gui* moves up or down the incoming pitch values of a MIDI event by a constant number typed-in the number object. The result is constrained to the minimum and maximum limits of **0** and **127**, respectively. If the result is outside this range, is replaced by **0** or **127**. The user interface contains the preset object for storing and retrieving transposition values. Input: raw MIDI data, **int**, mouse movement. Output: raw MIDI data.

Table 8b

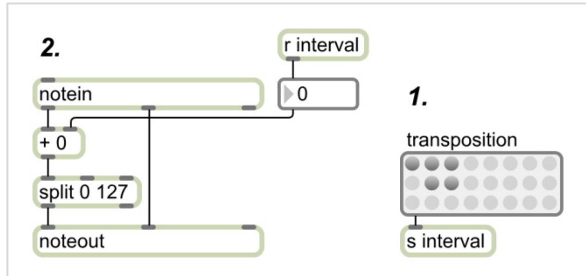


Table 9a

Interval: direction, harmonic, melodic, pitch, size

int.chord.sort	Split pitch values into **`int`** or **`list`** numbers
int.dir	Identify the direction of a melodic interval
int.dir.change	Signal when the direction of a melodic interval is changed
int.dir.size	Identify the direction and size of a melodic interval
int.size.2n	Identify the size of a harmonic interval
int.table	Store the pitch values of a MIDI event into a table

Table 9b

The patch *int.table* stores in the itable object and displays the pitch values of a MIDI event. It also filters out incoming values less than C1 (**36**) and greater than B4 (**83**).

Input: raw MIDI data, **`int`**, **`symbol`**.

Output: **`int`** (**`36`**–**`83`**).

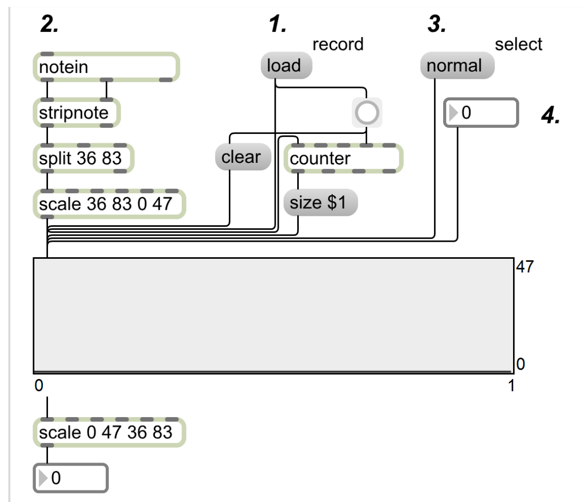


Table 10a

Velocity: decrease, increase, range, voice

vel.gui	Identify and split the velocity values of an event in ranges
vel.inc.dec	Increase or decrease the velocity of an event
vel.rec	Store the velocity values of a single voice
vel.sel	Signal the velocity range of an event
vel.table	Store the velocity values of an event into a table
velocity	Identify the velocity values of an event according to a range

Table 10b

The patch *vel.rec* filters harmonic intervals and chords, and then stores the velocity numbers of MIDI notes which are identified within a single voice. A velocity value stored inside the funbuff object is recalled by typing its address number in the number object.

Input: raw MIDI data, **int**.

Output: **int**.

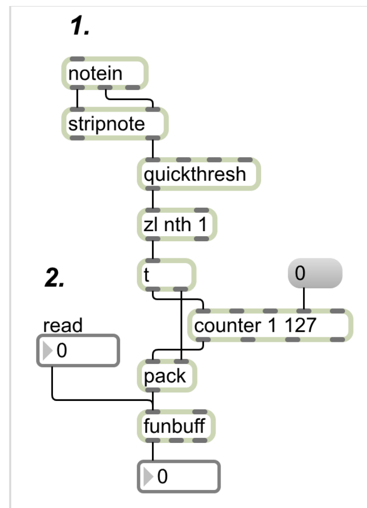


Table 11a

Duration and Time: augmentation, diminution, speed, tempo

aug.dim	Augment or diminish the duration by a **float** number
aug.dim.gui	Augment or diminish the duration by a fixed amount
tempo.sec	Decrease or increase the speed of a sequence in seconds
tempo.ticks	Decrease or increase the tempo of a sequence in ticks
delta time, duration, silence, voice	
dur.sil	Identify duration and silence in milliseconds
dur.sil.dt	Identify duration, silence, and delta time in milliseconds
dur.sil.dt.poly	Identify duration, silence, delta time, and voice
silence	Signal the silence between 2 events

Table 11b

This program called *dur.sil.dt.poly* recognizes the duration of an incoming MIDI note, the silence and delta time between two consecutive notes of each allocated MIDI voice. The voice is represented by a single number assigned to the *note on* and *note off* pair. It reports the results in milliseconds counted for duration, silence and delta time.

Input: raw MIDI data.

Output: `int`.

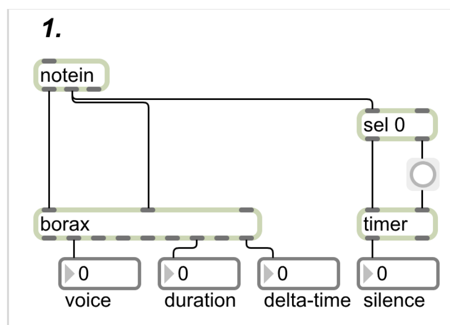


Table 12a

Humanize: chord, delay, velocity

hum.pitch.abs	The abstraction of the program *human.pitch*
hum.pitch.abs.init	The interface of the abstraction *hum.pitch.abs*
hum.vel.abs	The abstraction of the patch *human.vel*
hum.vel.abs.init	The interface of the abstraction *human.vel.abs*
human.audio	An interface for abstractions called *hum.pitch.abs* and *human.vel.abs*
human.pitch	Simulate a human performance of a chord using discrete delay time
human.vel	Simulate a human performance of a chord using discrete velocity
humanize	Simulate a human performance of a chord using discrete delay time and discrete velocity

The patch *humanize* combines features of the *human.pitch* and *human.vel* programs to simulate a human performance. The patch named *human.pitch* filters out incoming MIDI events, removing harmonic intervals and chords, thus it allows passing a single voice. It also computes pitch numbers in order to generate 3-notes chords by adding values typed-in the number objects to the incoming value. Both first and second note of the chord is delayed by a very small and different number of milliseconds in order to simulate a human performance. Duration numbers remain unchanged. The patch called *human.vel* randomly generates three velocity values of the chord within a small range in order to simulate a human performance. Input: raw MIDI data, `int`. Output: raw MIDI data.

Table 12b

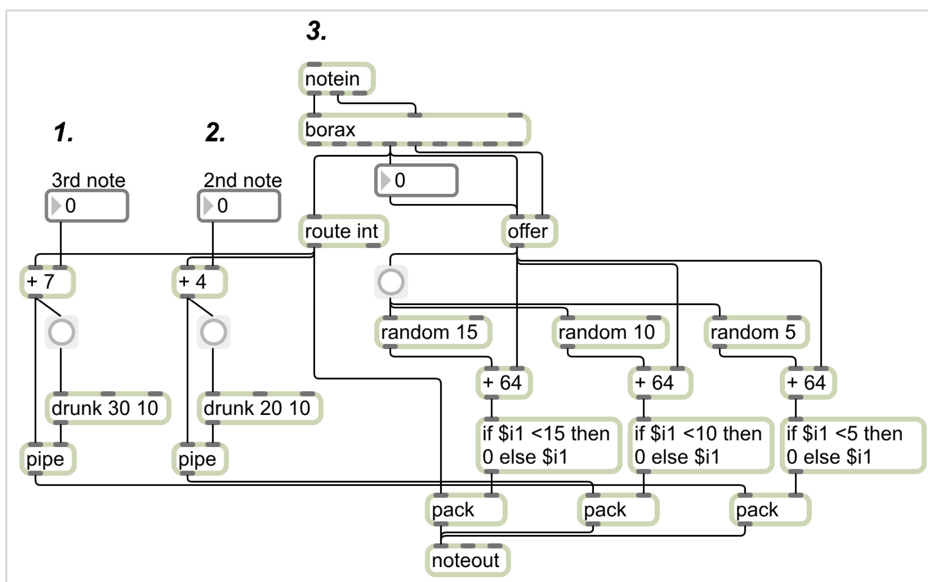


Table 13a

Chords and Progressions: interval, pitch, velocity

chord.3n	Generate 3-notes chord
chord.rec	Store and sort lists of pitch values
chord.size	Identify the number of notes of an event
chord.size.8n	Identify the size of intervals of an 8-notes chord
chord.sort.route	Sort the pitch values for the route object
chord.sort.spray	Sort the pitch values for the spray object
chord.vel	Identify the velocity of notes of an event
chord progression	
progress.abs	The abstraction of the program *progressions*
progress.abs.int	The interface of the abstraction *progress.abs*
progressions	Make chord progressions

The program *progressions* generates 3-notes chords as a response to the MIDI pitch values `60`, `62`, `64`, `65`, `67`, `69` and `71`, which represents C3, D3, E3, F3, G3, A3 and B3. Their velocity and duration values are identical with the input numbers of these parameters. Each received note generates a different chord specified inside the file *progressions.txt*. The

line of code holds an address followed by three pitch values: `1, 64 67 72;` `2, 65 69 74;` `3, 64 67 71;` `4, 65 69 72;` `5, 62 67 71;` `6, 64 69 72;` `7, 62 65 71;`. Input: raw MIDI data, `symbol`. Output: raw MIDI data.

Table 13b

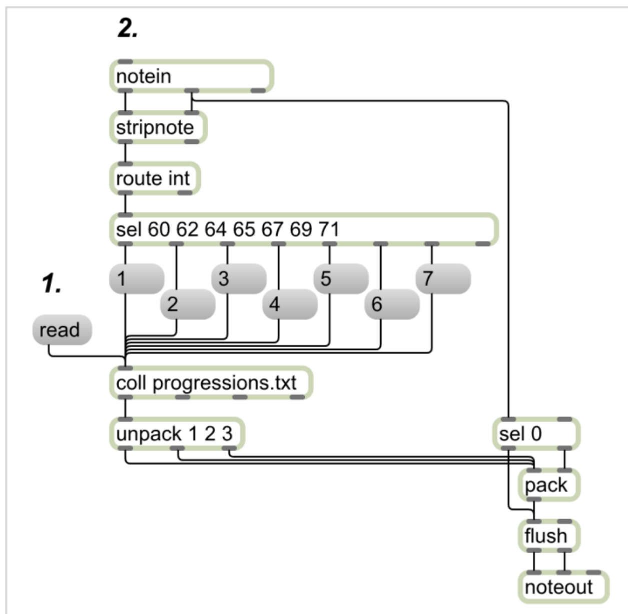


Table 14a

MIDI: chord, driver, duration, note, text, VSTi

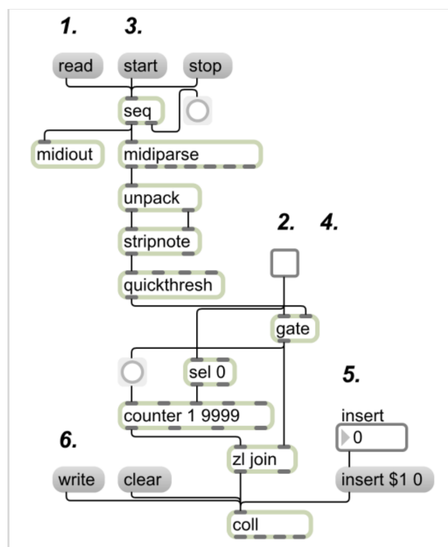
midi.bag	Make notes with the bag object
midi.chord.3n	Make 3-notes chord with the offer object
midi.driver	Select installed MIDI drivers
midi.flush	Make events with the flush object
midi.make	Make events of fixed duration with the makenote
midi.makenote	Make events of variable length with the makenote
midi.offer	Make events with the offer object
midi.text	Convert MIDI messages into text format
midi.vsti	Send MIDI messages to a VSTi plug-in
midi.vsti.abs	The abstraction of the patch *midi.vsti*
midi.vsti.abs.init	The interface of the abstraction *midi.vsti.abs*

Table 14b

The program *midi.text* filters and converts MIDI messages into `symbol` messages for pitch numbers identified in incoming flow data. Pitch numbers are stored as `list` messages in the coll object in order to be used with the programs called *follow* and *follow.sync*.

Input: raw MIDI data, `int`, `symbol`.

Output: raw MIDI data, `list`.



Conclusions

This paper highlights elements of formalization of musical language. The musician-programmer deals, in most cases, with the abstraction and transfer of rules and composition procedures. Working with high-level musical concepts, the cognitive transfer is completed in mathematical and logical operations.

The features of the computer object refer to what the Max object means to a musician and how the object behaves in a program made by him. This vision, from the outside, on the computer object belongs to the musician concerned with what he represents and what is the object.

The attributes of the Max object also refer to what the object's resource file contains and how its source code works. This vision, from the inside, is suitable for the musician interested in what makes up and how the object code is running.

The Max object is the indispensable foundation for the construction of compositional algorithms.

REFERENCES

Cycling ’74, *Max Fundamentals*, 4.6, 2006.
Cycling ’74, *Max Tutorials*, 2016.
The MIDI Manufacturers Association, *MIDI 1.0. Detailed Specification*, 4.2, Los Angeles, 1995.