Skini on Node.JS

Instructions for use and implementation

B. Petit-Hédelin, 05/2024

1 INTRODUCTION

Skini is a collaborative and generative music composition platform. This document covers the technical aspects of implementing Skini on Node.JS and creating orchestrations. It corresponds to the operation of Skini in its May 2024 version. To understand how Skini works, please refer to the document "*Time and Duration: from synchronous programming to musical composition*" or to the various articles published on this platform (Programming journal 2020, ICMC 2021, NIME 2019).

This document also includes an appendix with procedures for using Finale and Ableton Live with Skini.

1.1 COMPOSITION PROCESS

Skini has been designed to compose music that will be performed in interaction with an audience or produced automatically by random processes. The solution includes a Web server that integrates orchestration modules written using a graphical programming tool that is an abstraction layer on top of the HipHop.js language. It is also possible to program in HipHop.js and JavaScript without using the graphical programming interface.

The composition method is based on two basic concepts: patterns and orchestration. These elements are described in the thesis "Time and duration: from synchronous reactive programming to musical composition". Here, we only deal with the practical dimension of the tool, i.e. its implementation.

1.2 PATTERNS

The composer can create patterns without any particular constraints from Skini. They will be seen as elements activated by a MIDI command and having defined durations in terms of number of pulses. Patterns are made available to the audience in the form of groups[1] . There are no constraints on pattern group sizes.

1.3 ORCHESTRATION

The composer defines the way in which groups are made available to the audience by means of orchestration. Orchestration enables pattern groups to be *activated* and *deactivated.* Among the information that enables it to evolve, we have listening to group selections by the audience, measuring duration, and listening to MIDI or OSC information.

Orchestration can be seen as a "super sequencer", bringing *queue-based* interaction functions and complex automation to commercial DAWs. Orchestration can also control the queuing of patterns without an audience. MIDI, note or control change commands can be issued directly from the orchestration.

[1] If the composer wishes to make a particular pattern available to the audience or simulator, he can create a group with a single element, for example.

The orchestration is written graphically in HipHop.js. A basic knowledge of synchronous programming is required to design a first orchestration. Once you've mastered HipHops.js, you'll be able to produce rich, inconceivable pieces.

1.4 DAW PIECES

In this document, we will only deal with Ableton Live and Bitwig Studio. The use of any other DAW will be based on the same principles as those proposed for either solution.

The composer creates patterns (clips). To each of these patterns, the composer associates a Skini note. This Skini note will be converted and sent to the DAW from commands issued by the audience or from a random process.

The composer can create as many patterns as he likes, bearing in mind that these patterns will be organized into groups, and it is these groups that will be made available to the audience or the random process. For a collaborative piece, the dimensions of the groups must therefore be compatible with the possible display on an interface for the audience. The composer also needs to strike the right balance between short patterns that will energize the interaction and long patterns that will stabilize the musical discourse.

In the case of interaction with the audience, for each pattern the composer must create a *sound file*, mp3 or wav, whose name is associated with the pattern in the pattern configuration file. By default, these are mp3 files that will be downloaded by the audience for listening before selection.

Once the patterns and associated MIDI commands have been created, all that remains is to create the configuration files as described in the Configuration chapter, then move on to orchestration.

1.5 PIECES WITH MUSICIANS (TO BE PORTED TO NODE.JS)

Skini can be used to communicate with musicians, using Raspberry Pi devices with the appropriate client, with or without synthesizers. The approach is identical to that of a DAW. The patterns activated no longer consist in issuing a MIDI command, but in displaying scores on dedicated musician clients, previously deposited in a subdirectory of the "./image" directory. This subdirectory is configured in the piece's configuration file. These files are in jpg format. The names of the jpg files are the same as those of the sounds associated with the patterns.

2 CONFIGURATION

Directories are referenced to the main directory where Skini is installed.

2.1 Installation

Simply install Node.js, a widely used solution, and copy the Skini files into a directory. If any packages are missing when Skini is launched, Node.js will report this with error messages. Simply install the packages with npm.

2.2 Installing Processing for the OSC/MIDI gateway

The OSC/MIDI gateway is used when the DAW only understands MIDI commands and the Skini server is not on the same machine as the DAW. If the Node.js server and the DAW are on the same computer, there's no need to install the OSC/MIDI gateway, so this step is not necessary. You can use Skini by accessing your machine's MIDI port directly.

So if communication with the DAW is via MIDI, and you're using one computer to host the Node.js server and another to host the DAW, you'll need to install Processing (www.processing.org). The two computers will be able to communicate via OSC, and you'll need the gateway. In Processing, you'll need to install the oscP5, TheMidiBus and WebSockets libraries (menu: add a tool -> libraries).

2.3 Configuration

There are two configuration files to be updated before launching Skini. One concerns network configuration, the other MIDI configuration.

2.3.1 IP configuration and directory

It is done with the file "./serveur/ipConfig.json".

Example of ipConfig.json :

```
{
        "remoteIPAddressImage": "192.168.82.96",
        "remoteIPAddressSound": "localhost",
        "remoteIPAddressLumiere": "192.168.82.96",
        "remoteIPAddressGame": "192.168.82.96",
        "serverIPAddress": "localhost",
        "webserveurPort": 8080,
        "websocketServeurPort": 8383,
        "InPortOSCMIDIfromDAW": 13000,
        "OutPortOSCMIDItoDAW": 12000,
        "distribSequencerPort": 8888,
        "outportProcessing": 10000,
        "outportLumiere": 7700,
        "inportLumiere": 9000,
        "sessionPath": "./pieces/",
        "piecePath": "./pieces/"
}
```

`serverIPAddress`: server address.

`remoteIPAddressLumiere`: for use with QLC for OSC dialogue

`remoteIPAddressDAW`: IP of Processing server for MIDI commands to Ableton or another DAW.

`remoteIPAddressSound`: Processing server IP for MIDI commands to REAPER, for the GOLEM show.

`remoteIPAddressImage`: IP of Processing server for widescreen display.

For standard use, simply update `remoteIPAddressAbleton` and `serverIPAddress`.

To change the port for websockets `"websocketServeurPort"`, use the powershell script ***changePortSkini.ps1***. (Otherwise, you'll need to run browserfity again on clients).

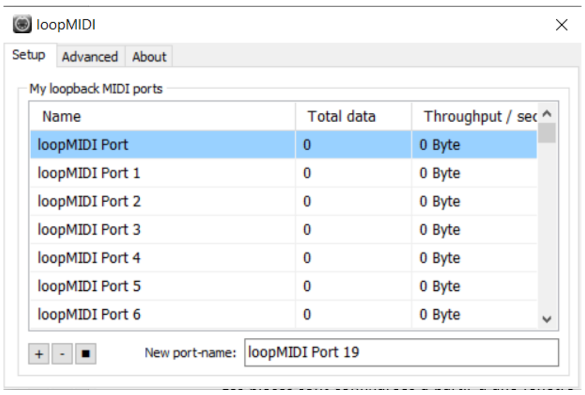
2.3.2 Access to pieces and pattern descriptors

The *sessionPath* and *piecePath* parameters are part of the system settings and are therefore on the same level as the network. *sessionPath* defines the directory for piece parameters, and *piecePath* those for Blockly orchestration. It is not possible to access the full path from a browser, which is why these parameters are necessary. These paths must be in a directory immediately below the *nodeskini* root (a constraint linked to access to midiConfig.json).

2.4 MIDI CONFIGURATION

MIDI ports are configured using the ./serveur/midiConfig.json file. This file defines MIDI buses according to the computer's configuration. This file is used by Skini and Processing.

Here is an example using LoopMIDI on Windows:



Name	Total data	Throughput / sec
loopMIDI Port	0	0 Byte
loopMIDI Port 1	0	0 Byte
loopMIDI Port 2	0	0 Byte
loopMIDI Port 3	0	0 Byte
loopMIDI Port 4	0	0 Byte
loopMIDI Port 5	0	0 Byte
loopMIDI Port 6	0	0 Byte

New port-name: loopMIDI Port 19

Figure 1 The LoopMIDI virtual cable

```
[
  {
    "type": "OUT",
    "spec": "clipToDAW",
    "name": "loopMIDI Port 6",
    "comment": "Bus for launching the clips in the DAW"
  },
  {
    "type": "IN",
    "spec": "syncFromDAW",
    "name": "loopMIDI Port 9",
    "comment": "for sync message from DAW"
  },
  {
    "type": "IN",
    "spec": "clipFromDAW",
    "name": "loopMIDI Port 12",
    "comment": "for clip activation message from DAW"
  },
  {
    "type": "IN",
    "spec": "controler",
    "name": "nanoKEY2",
    "comment": "to test a MIDI controller"
  }
]
```
The "type" field defines whether this is an IN or OUT port.

The "spec" field defines the use with:

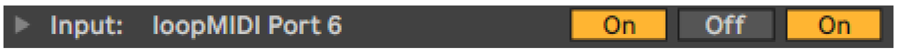
- "clipToDaw" defines the port that allows the DAW to receive MIDI commands from Skini.
- "syncFromDAW" defines the port that will receive MIDI synchronization from the DAW.
- "clipFromDAW" defines the port via which DAW sends clip start messages.
- "controle" corresponds to a port to which a MIDI controller (keyboard, PAD...) is assigned. is not necessary for Skini's basic operation.

The "name" field contains the name of the MIDI port on the computer. Here we're talking about ports on the LoopMIDI interface.

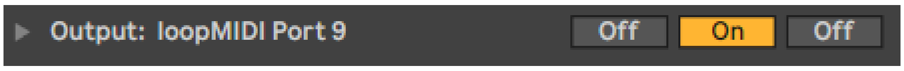
The "comment" field allows you to comment on the use of the port.

Here's an example of how to configure MIDI ports in Ableton to match the above example.

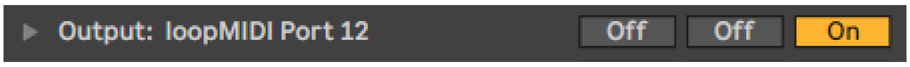
Port 6, IN for Ableton and OUT for Skini, enables control from Skini.



Port 9 sends MIDI synchronization to Skini. This is only used when using MIDI or MIDI/OSC synchronization mode with a gateway. With Link this is not necessary.



Port 12 is used for "feedback" information from Ableton, when a clip is played.



For more details, especially with Bitwig Studio, see the appendices of this document.

Note: The *busMidiDAW* parameter in a piece's configuration file is only useful when using the *Processing* gateway. It's an index that allows *Processing to* find the MIDI port to send *noteOns* to the DAW. If the Processing gateway is not used, *busMidiDAW* is not used. In the case of

Node.js, the gateway is only used when the DAW and server are on two different machines, as MIDI can be spoken from node.js.

2.5 Piece configuration

Pieces are configured from a window opened by the "Parameters" button in the main Blockly window (patterns are configured by clicking on the "Patterns" button).

For the curious: The piece is configured in a JavaScript file, which is loaded at the same time as the orchestration. It is selected from the orchestration programming window. The name of this file must correspond to the orchestration. For example, the orchestration opus1.xml should have the configuration file opus1.js.

Let's take a look at the piece's configuration parameters.

2.5.1 How to receive MIDI commands

If `"Direct Midi"` is ticked, this means that communication between Skini and the DAW is via MIDI. Otherwise, communication between Skini and DAW is via OSC. OSC is used with the Processing gateway, essentially for Ableton Live, while Bitwig Studio can communicate directly via OSC if the Skini controller is installed.

2.5.2 Reaction mode

The `"React on Play"` parameter defines how the orchestration automaton reacts. The default is on selection. With `"React on Play"` checked, it reacts when the pattern is played. This has a major impact on the way the automaton thinks. *Stingers* (for transitions) are only possible with `"React on Play"` active.

2.5.3 Sound files for customers and clients

The `Sound Files Path` parameter defines the path of sound files, associated with patterns, that are downloaded by clients from the .\sounds directory.

The `"Number of client groups"` parameter `sets` the number of groups of people in the audience that the orchestration can manage.

With the `"Simulator in a seperate Group"` parameter, we introduce the possibility of dedicating a group to the simulator. This means that the patterns available to the simulator will not be seen by the audience. This requires a number of groups greater than 2, as the last group will be that of the simulator. If the `"Simulator in a seperate Group"` parameter is unchecked, there is no group dedicated to the simulator.

The `"Algo Fifo management"` parameter can be used to activate an algorithm for processing queues (see below).

2.5.4 Synchronization

There are 4 possible synchronization modes. By Midi, by Midi via OSC, by Ableton Link and locally with a node.js worker. You can only have one mode at a time.

Midi mode is the simplest, and works with all DAWs that can send Midi sync. It is possible to communicate using OSC with Bitwig studio, as there is a Skini controller that allows this. For

Ableton, you'll need to use the Processing gateway. Ableton Link is very easy to use and allows you to synchronize over a network.

The Node.js worker is useful for projects without a DAW, i.e. with Musicien or Raspberry that don't provide sync, and without external Ableton Link sync from any software. It's mainly a test tool when you don't have a DAW.

To choose the synchronization mode we have the following parameters:

- *Syncho On Midi Clock* is used for MIDI synchronization from the DAW.
- *Synchro Link* is used for synchronization using the Ableton Link protocol.
- *Synchro Skini* is used for synchronization since Skini manages its own synchronization according to a *timer* parameter that gives the tick in ms. In this case, you cannot change the tempo in the orchestration.

If all these synchronizations are inactive, Bitwig can send the sync to OSC via Bitwig's Skini controller.

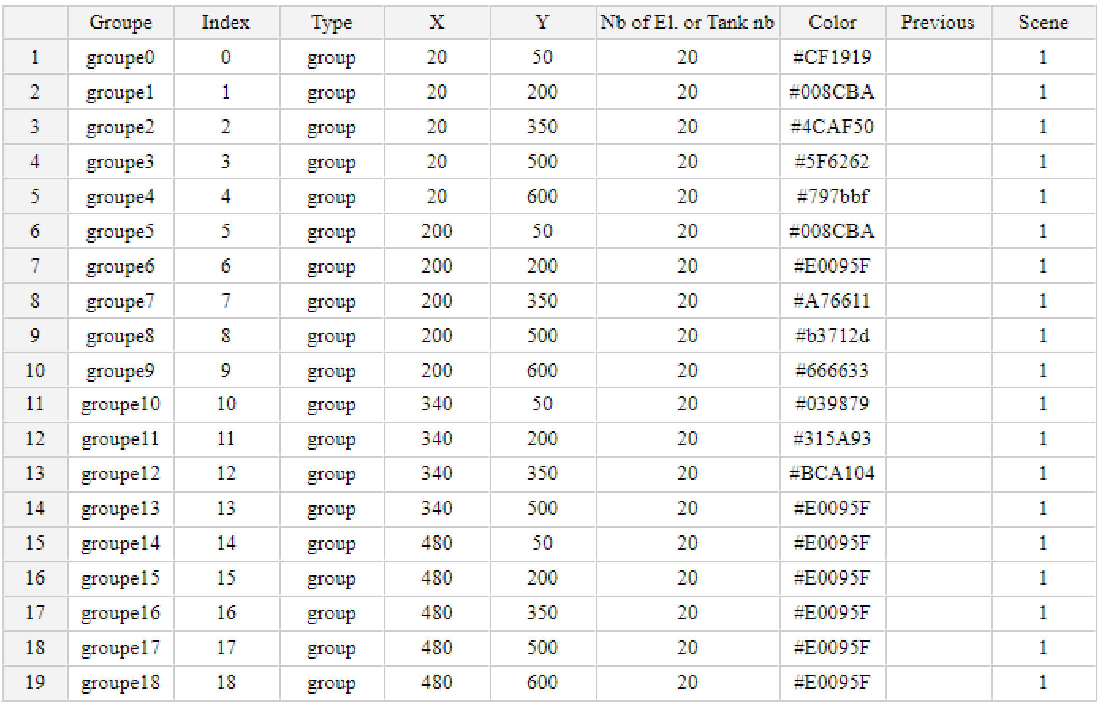
Please note:

1) Skini must be restarted when the synchronization mode is changed.

2) You must not have Bitwig's OSC sync via the Skini0 controller at the same time as another sync. You'd receive too many potentially offset and duplicate synchronization messages.

2.5.5 Pattern groups

Pattern group naming is done via a table accessible from the main window and the "Parameters" button. Pattern group names are used to create HipHop orchestration signals. Here's an example:



	Groupe	Index	Type	X	Y	Nb of El. or Tank nb	Color	Previous	Scene
1	groupe0	0	group	20	50	20	#CF1919		1
2	groupe1	1	group	20	200	20	#008CBA		1
3	groupe2	2	group	20	350	20	#4CAF50		1
4	groupe3	3	group	20	500	20	#5F6262		1
5	groupe4	4	group	20	600	20	#797bbf		1
6	groupe5	5	group	200	50	20	#008CBA		1
7	groupe6	6	group	200	200	20	#E0095F		1
8	groupe7	7	group	200	350	20	#A76611		1
9	groupe8	8	group	200	500	20	#b3712d		1
10	groupe9	9	group	200	600	20	#666633		1
11	groupe10	10	group	340	50	20	#039879		1
12	groupe11	11	group	340	200	20	#315A93		1
13	groupe12	12	group	340	350	20	#BCA104		1
14	groupe13	13	group	340	500	20	#E0095F		1
15	groupe14	14	group	480	50	20	#E0095F		1
16	groupe15	15	group	480	200	20	#E0095F		1
17	groupe16	16	group	480	350	20	#E0095F		1
18	groupe17	17	group	480	500	20	#E0095F		1
19	groupe18	18	group	480	600	20	#E0095F		1

2.5.6 Musicians (Not in place)

The presence of musicians must be specified with the lines:

```
exports.withMusician = true;
exports.decalageFIFOwithMusician = 4;
exports.patternScorePath1 = "";
```

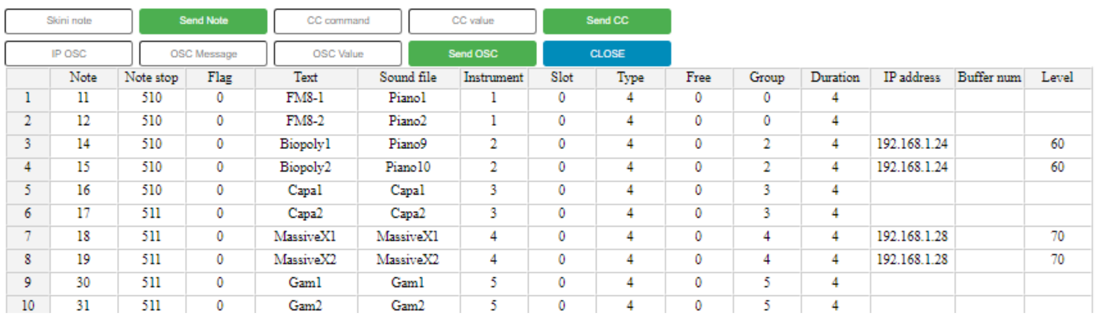
`decalageFIFOwithMusician` gives the pulse count before the first pattern is played. This is because, unlike a DAW, a musician needs to prepare before playing a pattern. This parameter introduces a systematic delay if there are no patterns queued for the instrument concerned. If there are patterns queued, the musician client displays the pattern following the current one. This ensures that the musician is not surprised.

`PatternScorePath1 is` the subdirectory of the "./images" directory where the orchestration partitions are located.

In the musician client, you need to log in with the instrument number. This allows the server.js to manage messages as images.

2.6 Pattern configuration

This is done from the window opened from the orchestration page by clicking on "Patterns". (This generates a csv file in the directory specified in the piece configuration).



	Note	Note stop	Flag	Text	Sound file	Instrument	Slot	Type	Free	Group	Duration	IP address	Buffer num	Level
1	11	510	0	FM8-1	Piano1	1	0	4	0	0	4			
2	12	510	0	FM8-2	Piano2	1	0	4	0	0	4			
3	14	510	0	Biopoly1	Piano9	2	0	4	0	2	4	192.168.1.24		60
4	15	510	0	Biopoly2	Piano10	2	0	4	0	2	4	192.168.1.24		60
5	16	510	0	Capa1	Capa1	3	0	4	0	3	4			
6	17	511	0	Capa2	Capa2	3	0	4	0	3	4			
7	18	511	0	MassiveX1	MassiveX1	4	0	4	0	4	4	192.168.1.28		70
8	19	511	0	MassiveX2	MassiveX2	4	0	4	0	4	4	192.168.1.28		70
9	30	511	0	Gam1	Gam1	5	0	4	0	5	4			
10	31	511	0	Gam2	Gam2	5	0	4	0	5	4			

The first column shows the MIDI note corresponding to the pattern in the DAW. These notes do not correspond exactly to MIDI notes. In fact, to simplify coding, we haven't imposed any limits on these numbers, unlike the MIDI standard, which only allows 128 notes on a channel. You can go beyond the limit of 127. Transcription into MIDI notes involves applying the following calculation:

```
var channel = Math.floor(note / 127) + 1;
note = note % 127;
```

This shows that the notion of MIDI channel is included in the Skini *note.* The equivalence between Skini notes and MIDI notes is only immediate for notes < 127. This makes it possible to associate MIDI commands with patterns without any channel constraints. This method frees us from tedious channel management.

The **Note Stop** column is the MIDI note used to interrupt a pattern in progress. This is specific to Ableton Live and is not generally useful.

Flag Usage is not a parameter, it's a tool for the Skini engine.

The **name** column gives the texts that will be associated with each pattern for the different clients. The *Sound file* column gives the names of the sound files in the directories defined in the JavaScript configuration file, which are mp3 files by default. To use *wave* files, add a ".*wav"* extension to the file names.

The **instrument** column associates patterns with a specific instrument, corresponding to a MIDI instrument or musician. There is no correspondence between these numbers and a MIDI configuration.

The **slot** columns relate to ongoing developments in live pattern recording. They can be ignored for Skini sessions without Live recording. The same applies to the last two columns, "waiting" and "pseudo".

The **type** column is used to qualify a pattern so that queues can be reorganized. 39, " Queue priority ".

The **group** column corresponds to the group indexes described in the group configuration (parameters). *It* is this parameter that associates the pattern with a group and enables patterns to be made available via orchestration.

The **duration** column defines the length of the pattern in terms of the number of **pulses** emitted by the synchronization, usually MIDI.

The **IP address** column is used to associate a pattern with an OSC command when Skini is working with distributed devices such as Raspberries Pi with an appropriate application. Always with Raspberries Pi **Buffer Num** is an OSC command parameter that defines the buffer associated with the pattern. **Level** is used to define the sound level specific to this pattern. If the **buffer num** field is empty, the pattern is activated on the DAW. OSC commands can be sent from this window to test distributed patterns.

2.7 LARGE-SCREEN ORCHESTRATION DISPLAY CONFIGURATION

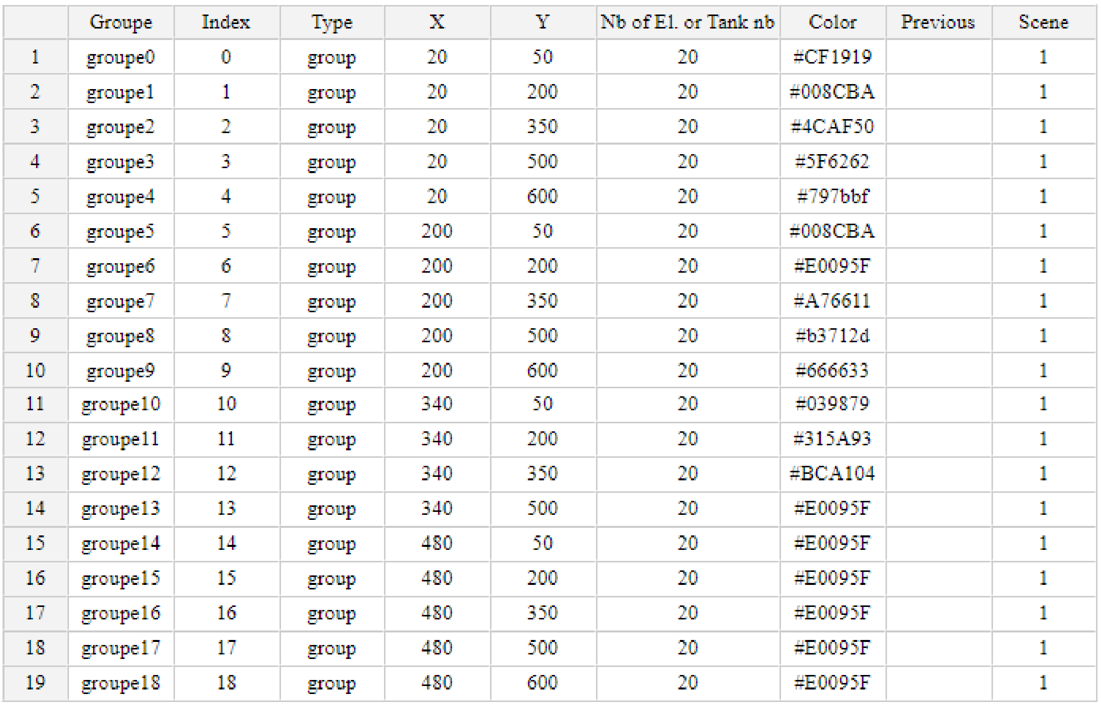
An orchestration sequence can be displayed in a browser. Skini offers a display based on connected boxes. Groups are represented by rectangles, and tanks by rectangles with rounded edges. Tanks reduce in size as they empty. You can access a visual of the orchestration from the programming window or with the url :

http://'*server*'/score

Parameters for this display are set in the piece's "Parameters" window. Below, we'll look at an example of how to set up a graph for opus1. For each group of patterns, we have fields giving the position of the rectangles, a color, a list of predecessors (i.e. groups pointing to the group in the line) and a scene number. The format is slightly different for groups and tanks. Between brackets is the index of the line's table.

For groups: group name (0), index (1), type (2), x(3), y(4), no. of elements(5), color(6), predecessors(7), graphic scene no.(8)

For tanks: tank name (0), index (1), type (2), x(3), y(4), tank number(5), color(6), Previous(7), graphic scene number(8) .



	Groupe	Index	Type	X	Y	Nb of El. or Tank nb	Color	Previous	Scene
1	groupe0	0	group	20	50	20	#CF1919		1
2	groupe1	1	group	20	200	20	#008CBA		1
3	groupe2	2	group	20	350	20	#4CAF50		1
4	groupe3	3	group	20	500	20	#5F6262		1
5	groupe4	4	group	20	600	20	#797bbf		1
6	groupe5	5	group	200	50	20	#008CBA		1
7	groupe6	6	group	200	200	20	#E0095F		1
8	groupe7	7	group	200	350	20	#A76611		1
9	groupe8	8	group	200	500	20	#b3712d		1
10	groupe9	9	group	200	600	20	#666633		1
11	groupe10	10	group	340	50	20	#039879		1
12	groupe11	11	group	340	200	20	#315A93		1
13	groupe12	12	group	340	350	20	#BCA104		1
14	groupe13	13	group	340	500	20	#E0095F		1
15	groupe14	14	group	480	50	20	#E0095F		1
16	groupe15	15	group	480	200	20	#E0095F		1
17	groupe16	16	group	480	350	20	#E0095F		1
18	groupe17	17	group	480	500	20	#E0095F		1
19	groupe18	18	group	480	600	20	#E0095F		1

Setting up reservoirs is more complex than setting up groups. A reservoir is a set of groups. A tank is defined by the field in fifth position. All groups/patterns in a tank have the same tank number.

Please note the numbering of predecessors. A group or tank can be used as a predecessor.

In the configuration below, generated in text format and not from Skini's parameter spreadsheet, we have added the group number as predecessor at the end of the line. Up to line 11, everything is simple. On line 12 we start a tank that ends on line 15. These 4 lines are in the same "predecessor" of value 11. So line 16 begins the "predecessor" of value 12. (Note that colors are expressed in hexadecimal format).

```
1.  [ "violinsScale",     0, "group",   323, 176, 12, ochre, [3,8], 1], //0
2.  [ "violinsChrom",     1, "group",   800, 180, 16, ochre, [9], 2], //1
3.  [ "violinsTonal",     2, "group",   450, 25, 24,  ochre, [30], 3], //2
4.  [ "altosScale",       3, "group",   200, 114, 12, violet,[5], 1], //3
5.  [ "altosChrom",              4, "group",   800, 270, 16, violet,[9], 2], //4
6.  [ "cellosScale",      5, "group",   52, 173, 10, green, [], 1], //5
7.  [ "cellosChrom",      6, "group",   800, 360, 16, green, [9],2], //6
8.  [ "cellosTonal",      7, "group",   650, 100, 8, green, [2], 3], //7
9.  [ "ctrebassesScale", 8, "group",   200, 244, 12, blue, [5], 1], //8
10. [ "ctrebassesChrom", 9, "group",   590, 430, 16, blue, [25, 29], 2],//9
11. [ "ctrebassesTonal", 10, "group",  650, 160, 6, blue, [2], 3], //10
12. ["trumpetsScale1",11, "tank",      273, 374, 1, orange, [5,0], 1], //11
13. ["trumpetsScale2",12, "tank",      200, 10, 1, orange, [],],
14. ["trumpetsScale3",13, "tank",      200, 10, 1, orange, [],],
15. ["trumpetsScale4",14, "tank",      200, 10, 1, orange, [],],
16. ["trumpetsTonal1",    15, "tank",   650, 280, 2, orange, [2], 3], //12
17. ["trumpetsTonal2",    16, "tank",   200, 100, 2, orange, []],
18. ["trumpetsTonal3",    17, "tank",   200, 100, 2, orange, []],
```

The orchestration display is triggered by the orchestration.

Note: The score client uses the JavaScript version of Processing, P5js. The source code can be found in *./Processing/P5js/score/score.js*

3 LAUNCH SKINI

For Skini to work with musicians without electronics, only Node.js is required. With electronics, you need:

- A Digital Audio Workstation (e.g. Ableton Live or Bitwig Studio).
- Eventually, MIT Processing will bridge the gap between OSC and MIDI.

To launch Skini you need:

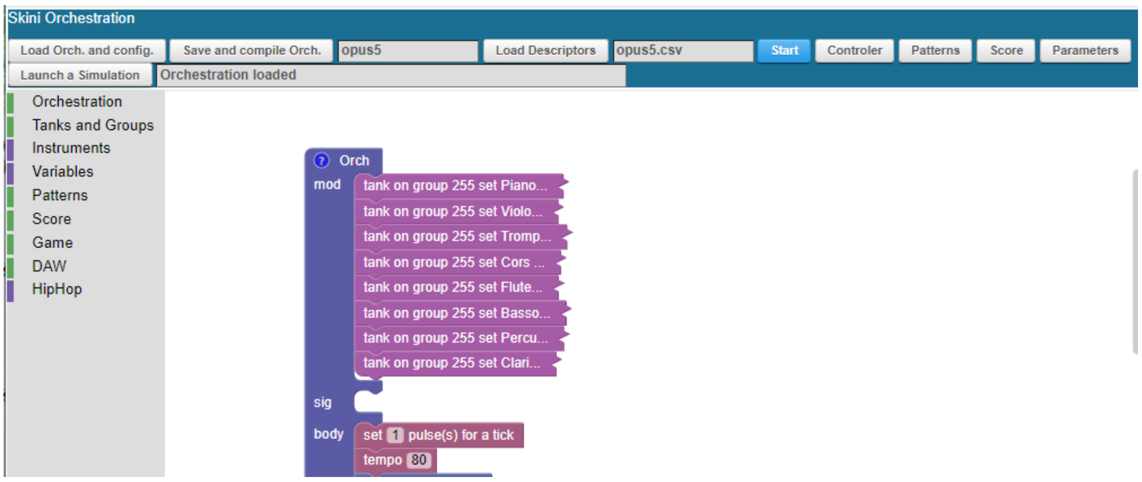
- Run: **node skini.***mjs*

With electronics, you must then:

1 If using the OSC/MIDI gateway, start *Processing* with the program *sequenceurSkini.pde*. The Processing console reports that the gateway has connected to the server. Remember that Bitwig studio and the Skini_0 controller do not use this gateway.

2 Launch and load patterns into the DAW. Start playback on the DAW to activate MIDI synchronization, if used. The Processing display should scroll, indicating that Skini is on its way.

It is now possible to load the orchestration programming in a Web browser with *http://IP server* :8080/block. Skini is ready for performance.



The orchestration window provides access to the controller, configuration and display of a graphical score.

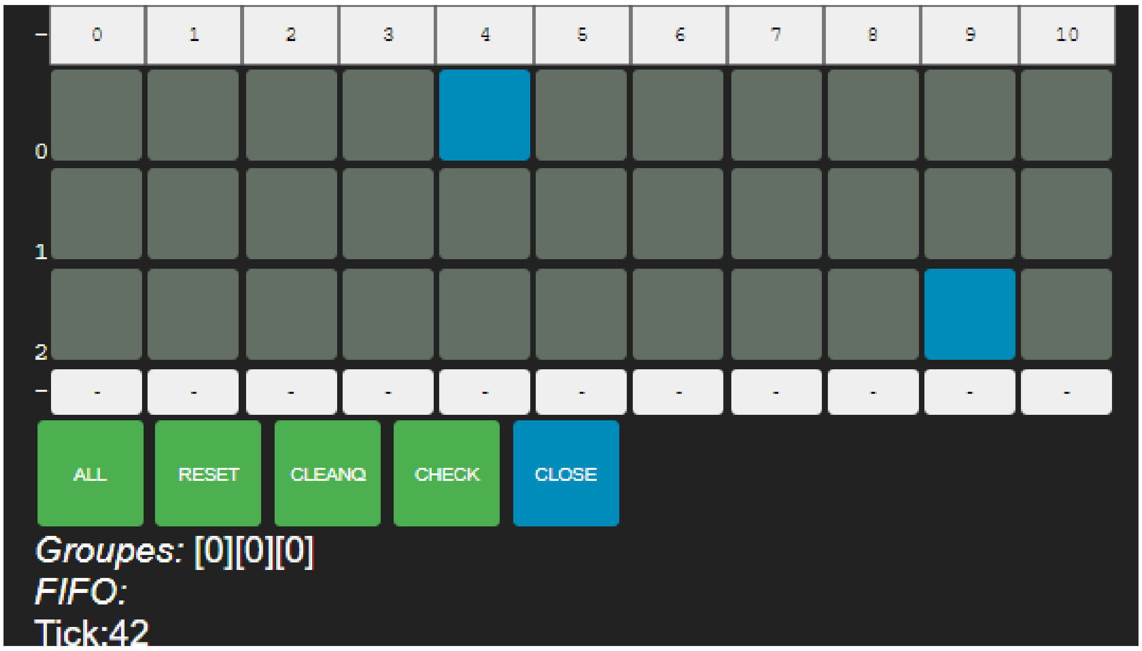
4 CONTROLLER OPERATION

The controller is opened from the orchestration window. It provides an overview of audience groups and pattern groups. It also enables real-time control of the *matrix of possibilities*, displayed as a table with the audience groups defined in the room configuration file as rows, and the pattern groups as columns. The controller can activate or deactivate a pattern group by clicking in the matrix. Clicking on the pattern group number activates or deactivates all groups in the audience for that group.

Please note: The correspondence between controller indexes and pattern group numbers is only valid if the group numbers follow each other in the piece configuration file. Controller indexes correspond to the line in the pattern group table, not to the group index. (To be reviewed, if possible).

The "ALL" button activates all groups. The "RESET" *button* deactivates all groups. The "CLEANQ" button clears all queues.

The "CHECK" button displays the pattern configuration file in the Skini console.



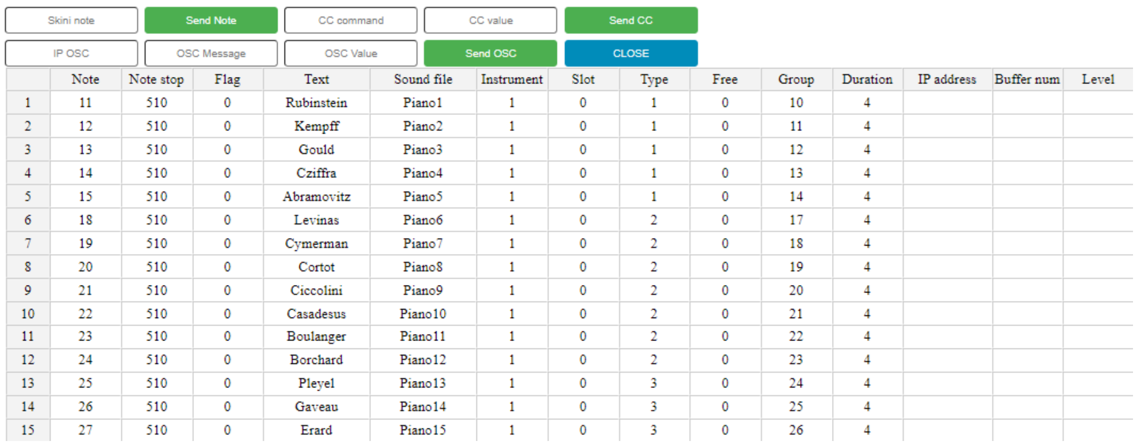
Groupes: [0][0][0]
FIFO:
Tick:42

Group, Scrutineer and FIFO displays provide system status.

5 Using the MIDI configurator

This tool is used to set up the DAW and the pattern configuration file, also known as the descriptor.

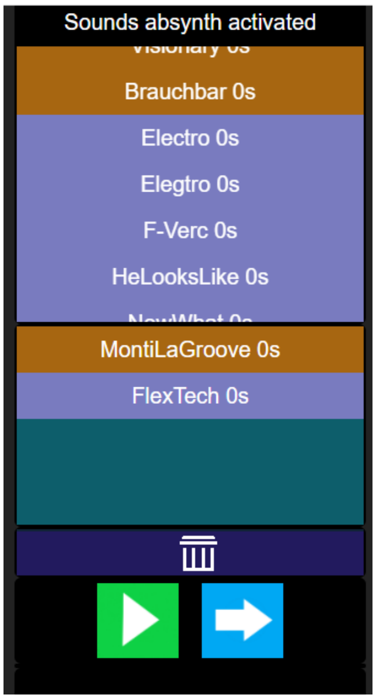
DAWs generally have a function for listening to MIDI commands from controllers. To facilitate the implementation of pieces, Skini can send commands like a controller. The MIDI Configurator thus behaves like a MIDI controller. Skini converts "Clip" commands into MIDI channels and notes. The configurator lets you send notes with "Send Clip" and MIDI "Control Changes (CC)" with "Send CC". The first field above "Send CC" is for the CC, while the field to the right of it lets you set a value for the CC. The configurator is accessed from the orchestrator (or via *http://adresse on the* server/conf). Pattern configuration is described in chapter " Pattern configuration ".



	Note	Note stop	Flag	Text	Sound file	Instrument	Slot	Type	Free	Group	Duration	IP address	Buffer num	Level
1	11	510	0	Rubinstein	Piano1	1	0	1	0	10	4			
2	12	510	0	Kempff	Piano2	1	0	1	0	11	4			
3	13	510	0	Gould	Piano3	1	0	1	0	12	4			
4	14	510	0	Cziffra	Piano4	1	0	1	0	13	4			
5	15	510	0	Abramovitz	Piano5	1	0	1	0	14	4			
6	18	510	0	Levinas	Piano6	1	0	2	0	17	4			
7	19	510	0	Cymerman	Piano7	1	0	2	0	18	4			
8	20	510	0	Cortot	Piano8	1	0	2	0	19	4			
9	21	510	0	Ciccolini	Piano9	1	0	2	0	20	4			
10	22	510	0	Casadesus	Piano10	1	0	2	0	21	4			
11	23	510	0	Boulanger	Piano11	1	0	2	0	22	4			
12	24	510	0	Borchard	Piano12	1	0	2	0	23	4			
13	25	510	0	Pleyel	Piano13	1	0	3	0	24	4			
14	26	510	0	Gaveau	Piano14	1	0	3	0	25	4			
15	27	510	0	Erard	Piano15	1	0	3	0	26	4			

6 USING THE SKINI CLIENT

The client, called by the URL http://<serveraddress:port>/skini, allows audience members to create a list of patterns and send it to the server. The client can feed a list (lower part of the screen) from the available pattern lists (drag and drop). The green button plays the list locally, while the blue button sends the list to the server. The trash can is used to delete items from the list.



Note: The patterns in the list of choices will fill the server's queues. If we have patterns for different instruments in a list, the queues will be fed. So we won't have the sequence heard locally between instruments. Sequences are respected per instrument.

This client comes with orchestration functions for dynamically defining list length and emptying the list.

Once a list has been sent, the customer will not be able to send another until the requested list has been played through.

This client, coupled with appropriate orchestration, can be used to create musical games. For example, you could let groups of clients take it in turns to design lists of patterns over a period of time, and then rate the quality of these lists, giving winners and losers. Queue "pause/resume" mechanisms can be used to manage periods of pattern design and play. The orchestration display tool, for example, can be coupled to a large-screen display. It is also possible to communicate with a video game platform such as Unreal Engine via signals for more graphically rich musical games.

Important note: Do not use queue reorganization algorithms with this client if these algorithms can remove patterns from FIFOs. This is because the client is waiting for confirmation of the set of patterns before authorizing the sending of a new (or the same) list. If a requested pattern disappears, the client will be blocked.

The cleanQueue block should be handled with care. Clients are blocked until all patterns in a list have been played. If the Fifo's are empty, some clients may be blocked because the patterns have disappeared from the Fifo's and will never be played. It's better to do "cleanChoiceList" (255) with a "cleanQueues".

If the patterns are typed, the server will evaluate a score according to the relevance of the succession of types in the list defined by a client. The scoring algorithm can be found in *websocketserver*.js, in the *computeScore*() function. The scoring rules must be modified in the source code. The scoring mechanism is based on pseudonyms.

The orchestration accesses the current winner using display score blocks.

7 USING THE SIMULATOR

The simulator was originally a tool for testing the behavior of a piece before it was used with an audience. It also enables patterns to be activated randomly during a performance with an audience. It therefore has two basic behaviors that are defined in the piece parameters with the fields:

```
Number of client groups
Simulator in a seperate Group
```

The purpose of defining the number of client groups is to define the number of groups of people in the audience that the orchestration will be able to manage independently. The allocation of a group to an audience member is cyclical. Each member is assigned a group according to its predecessor at the time of connection.

7.1 SIMULATOR SPECIALIZATION

The `Simulator in a seperate Group` parameter, when checked, means that the last client group is reserved for the simulator. The audience will not have access to it. Otherwise, it means that the simulator can behave like any other audience group.

7.2 ACTIVATION

The audience simulator is launched either from the blockly programming window or with the command:

```
./nodeskini/client/simulatorListe/node simulatorListe.js
```

Simulator outside the audience on the last "group of people" when `Simulator in a seperate Group` is `checked,` it is launched from the main window or with the command:

```
./nodeskini/client/simulatorListe/node simulatorListe.js -sim
```

The simulator features a mechanism that prevents two successive repetitions of the same pattern over three selections.

7.3 SETTINGS

The simulator is set in the piece configuration file with the lines:

```
Tempo Max In ms
Tempo Min In ms
Limit Waiting Time (in pulse)
```

Each call to the server is made at a time defined by:

tempoInstant = Math.floor((Math.random() * (tempoMax - tempoMin)) + tempoMin);

This is a random duration between two limits, `tempoMax` and `tempoMin.`

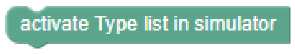
`Limit Waiting Time` is the parameter that defines the waiting time for a pattern, beyond which the simulator will not call the server.

NB: The simulator includes a mechanism to avoid repetition of the same pattern in a history of 3 previous patterns. This is the `selectRandomInList` function.

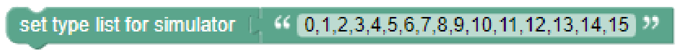
7.4 MANAGING PATTERN TYPES IN THE SIMULATOR

You can ask the simulator to process lists of patterns to be solicited. In other words, to apply a defined order to them according to a list. For example, if we have 5 types of patterns (0,1,2,3,4), we can ask the simulator to 'send Skini lists of patterns classified according to the series (2,3,0,4). When the simulator receives a list of patterns from Skini, it will choose randomly from this list, classifying its choices according to the series (2,3,0,4). If one of the types is missing, the classification will be made without the missing type. If the simulator receives a set of patterns containing no type 0, it will classify according to (2,3,4).

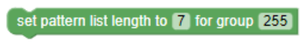
The function is activated from the orchestration with:


activate Type list in simulator

The type series is defined by the:


set type list for simulator " 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 "

When using this type of management, the "pattern list length" definition in the

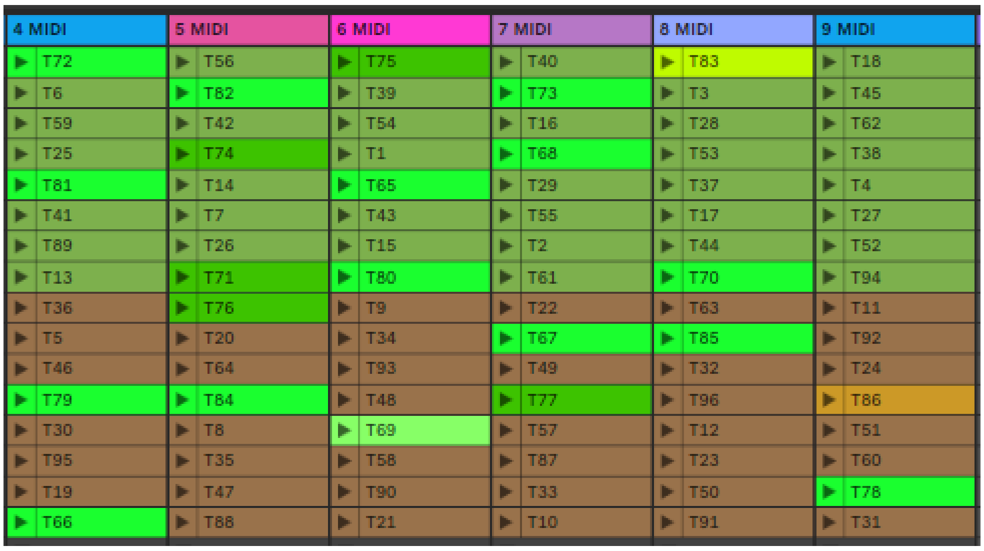

set pattern list length to 7 for group 255

block is not considered. The pattern list sent will be at most the length of the defined type of list.

Caution: Type management in the simulator must not be used in conjunction with Skini's FIFO management algorithm (the "Algo Fifo managements" parameter must be set to 0). Both techniques use *pattern types*, but for different purposes that may conflict with each other.

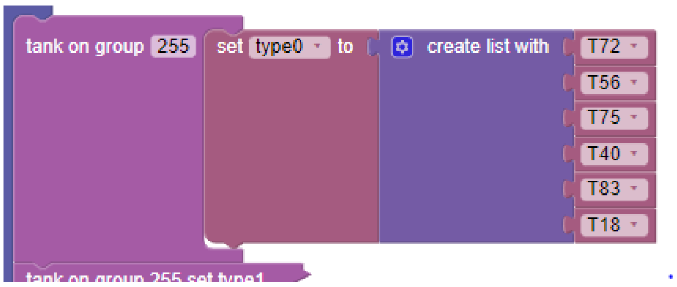
7.5 EXAMPLE OF THE USE OF TYPE LISTS: THE WÜRFELSPIEL

We used type management to implement a Stadler trio, created in the early 19th century for a würfelspiel. We created Stadler's patterns in Ableton Live according to the model proposed by Stadler:



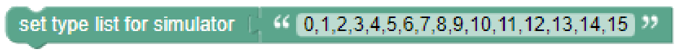
4 MIDI	5 MIDI	6 MIDI	7 MIDI	8 MIDI	9 MIDI
▶ T72	▶ T56	▶ T75	▶ T40	▶ T83	▶ T18
▶ T6	▶ T82	▶ T39	▶ T73	▶ T3	▶ T45
▶ T59	▶ T42	▶ T54	▶ T16	▶ T28	▶ T62
▶ T25	▶ T74	▶ T1	▶ T68	▶ T53	▶ T38
▶ T81	▶ T14	▶ T65	▶ T29	▶ T37	▶ T4
▶ T41	▶ T7	▶ T43	▶ T55	▶ T17	▶ T27
▶ T89	▶ T26	▶ T15	▶ T2	▶ T44	▶ T52
▶ T13	▶ T71	▶ T80	▶ T61	▶ T70	▶ T94
▶ T36	▶ T76	▶ T9	▶ T22	▶ T63	▶ T11
▶ T5	▶ T20	▶ T34	▶ T67	▶ T85	▶ T92
▶ T46	▶ T64	▶ T93	▶ T49	▶ T32	▶ T24
▶ T79	▶ T84	▶ T48	▶ T77	▶ T96	▶ T86
▶ T30	▶ T8	▶ T69	▶ T57	▶ T12	▶ T51
▶ T95	▶ T35	▶ T58	▶ T87	▶ T23	▶ T60
▶ T19	▶ T47	▶ T90	▶ T33	▶ T50	▶ T78
▶ T66	▶ T88	▶ T21	▶ T10	▶ T91	▶ T31

We have defined 16 types numbered from 0 to 15, each corresponding to a line in this table. For each type we have 6 patterns. For each type we define a tank with the 6 patterns of a line, e.g. for type 0 :



```
tank on group 255   set type0 to   create list with   T72
                                                     T56
                                                     T75
                                                     T40
                                                     T83
                                                     T18
tank on group 255 set type1
```

The trio consists of successively playing patterns according to a list of types corresponding to the rows of the table. We define it as follows:



```
set type list for simulator  " 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 "
```

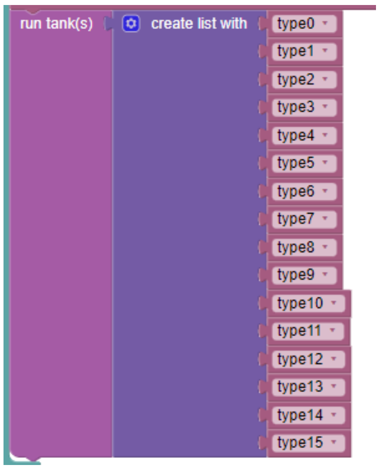
As we use tanks for each line, no patterns are replayed. The Skini programming of the würfelspiel consists in matching:

- the **Group** fields of the pattern descriptor with the **Index** fields of the parameters,

- the **Types** fields of the pattern descriptor with the **Tanks nb** fields of the parameters,

- Text fields of the pattern descriptor with **Group** fields of the parameters,

Then simply activate all the tanks and hold an instruction.



Unlike Stadler's würflespiel, there is no repetition between the trio's different sequences.

Note: The implementation of this würfelspiel is an example of how Skini can be used to generate tonal music. The list of types follows a harmonic structure, which is simple here, but could be enriched or modified during the piece.

8 GRAPHIC ORCHESTRATION PROGRAMMING

For orchestration programming, the composer accesses an interface using the *Blockly* solution provided as open source by Google. Blockly is easy to use and user-friendly. This is the interface used by Scratch, the programming learning tool for children.

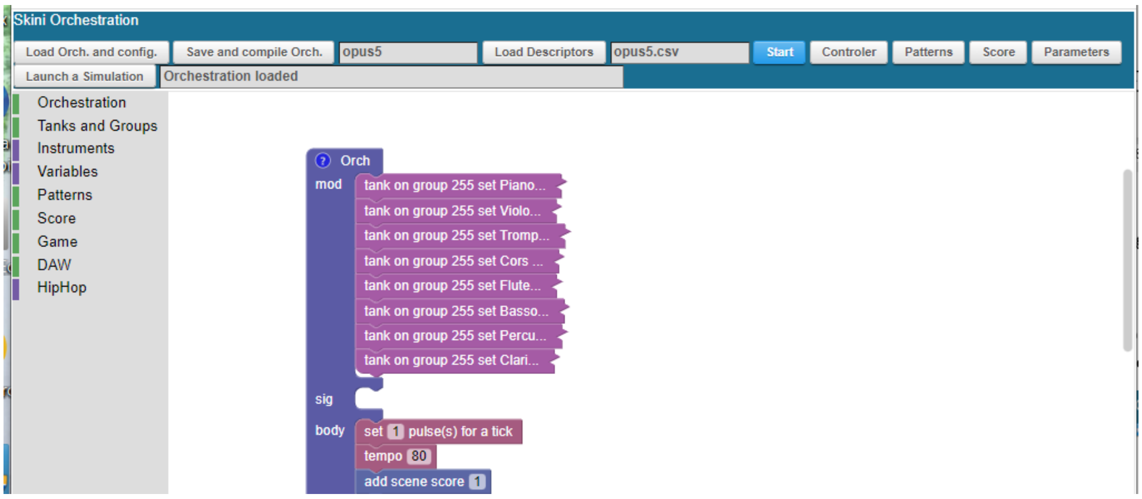
We won't go into how Blockly works here. A visit to the

https://developers.google.com/blockly

will be more complete and effective than a general Skini presentation.

8.1 SKINI'S GRAPHICAL INTERFACE

The Blockly interface generates HipHop.js programs without the need to know how to program in this language. The composer can load an orchestration with "Load Orch. and config".



The "Save and Compile" button does two things. It saves the current file as an *xml* file with the name entered in the text field. It creates a HipHop.js file. "Start" launches the orchestration.

"Launch a Simulation" lets you launch a simulator without using a console.

"Patterns" opens the pattern descriptor configuration window. "Score" opens the pattern block display window and "Controler" the control window. "Parameters" opens the piece parameters window.

Note: The controller provides more information on active pattern groups and ticks. Group information is visible with the orchestration display (Skini "score" client), if this is provided for in the piece configuration.

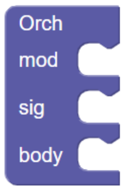
Let's look at the main blocks of the Blockly interface.

Note: "Save and compile" compiles Blockly into a ./myReact/orchestrationHH.js file. This file is not useful for the composer. The path is set in websocketServer.js with the variable generatedDir.

8.2 Orchestration blocks

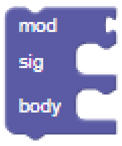
These are the blocks that define the structure of the orchestration.

The first module is indispensable. "Orch".



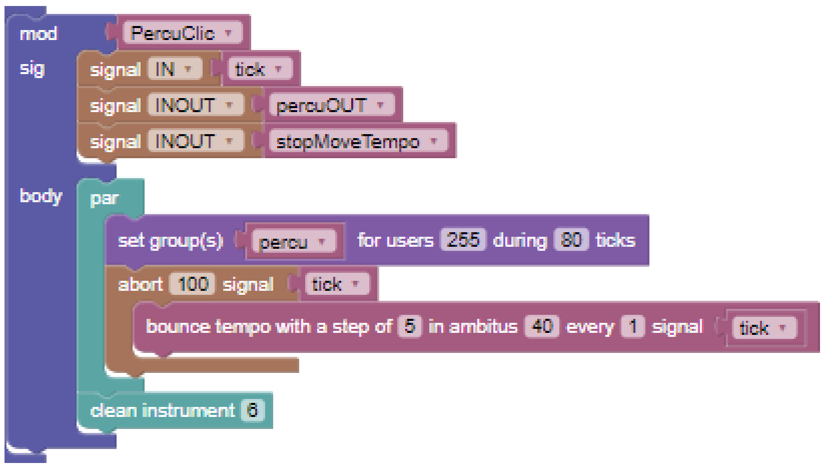
Blockly code is organized in blocks that can be paralleled with the:

The "mod" field is used to create "Modules", such as the reservoirs we'll see below, which will be called up in the orchestration body. The "sig" field is used to declare signals used in the orchestration.



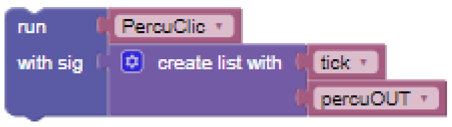
The use of modules with groups or tanks calls for "implicit" signals, i.e. signals that do not appear in the Blockly orchestration. A group activation corresponds to a xxxOUT signal where xxx is the group name, and a signal from a xxxIN group. There are also default signals in the main program which are not default signals in the modules, and which must be declared as input or output in the modules if required: *start, halt, DAWON, tick, patternSignal, controlFromVideo, pulsation, midiSignal, emptyQueueSignal, stopReservoir, stopMoveTempo.*

Here's an example of a module that uses implicit signals, here *stopMoveTempo* linked to the "bounce tempo" block and *percuOUT* linked to the "percu" group.



```
mod  PercuClic
sig  signal IN     tick
     signal INOUT  percuOUT
     signal INOUT  stopMoveTempo
body par
       set group(s) percu for users 255 during 80 ticks
       abort 100 signal tick
         bounce tempo with a step of 5 in ambitus 40 every 1 signal tick
     clean instrument 6
```

The module is called with :



```
run       PercuClic
with sig  create list with  tick
                            percuOUT
```

8.3 HipHop blocks

They correspond to the commands of the HipHop synchronous reactive language. See the Skini tutorial for a complete overview. Here's an example:

By default, blocks are executed one after the other, but for ease of reading or when you want to parallel sequences of instructions, you can group them together using the "seq" block:

The advantage of this block is that you can easily activate Blockly's "collapse block" function to make orchestration more synthetic.

8.4 Time programming

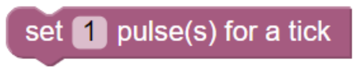
Time management in Skini is designed to ensure consistency in pattern sequencing and programming flexibility. It is based on two main parameters:

- *Pulsation*, which is supplied either by midi synchronization, Ableton Link or directly by Skini (see Setting up a piece).
- The *tick*, which is a multiple of the *pulse* and is used by the queue read-out mechanism.

8.4.1 Using the tick signal

By defining a *tick* as a multiple of a pulse, we can define the cycle on which pattern starts in queues will be synchronized. If we set a *tick to* 4 pulses, each pattern in a queue will be launched at the earliest at the start of a 4-pulse cycle. At the earliest, because a pattern can last longer than 4 pulses, the next pattern will have to wait until the end of the previous one. This mechanism avoids overlapping patterns for the same instrument.

A pattern is therefore placed in a queue each time it is selected, and the queues are unstacked at regular intervals that are multiples of the pulse, called a *tick*. This interval is set with:

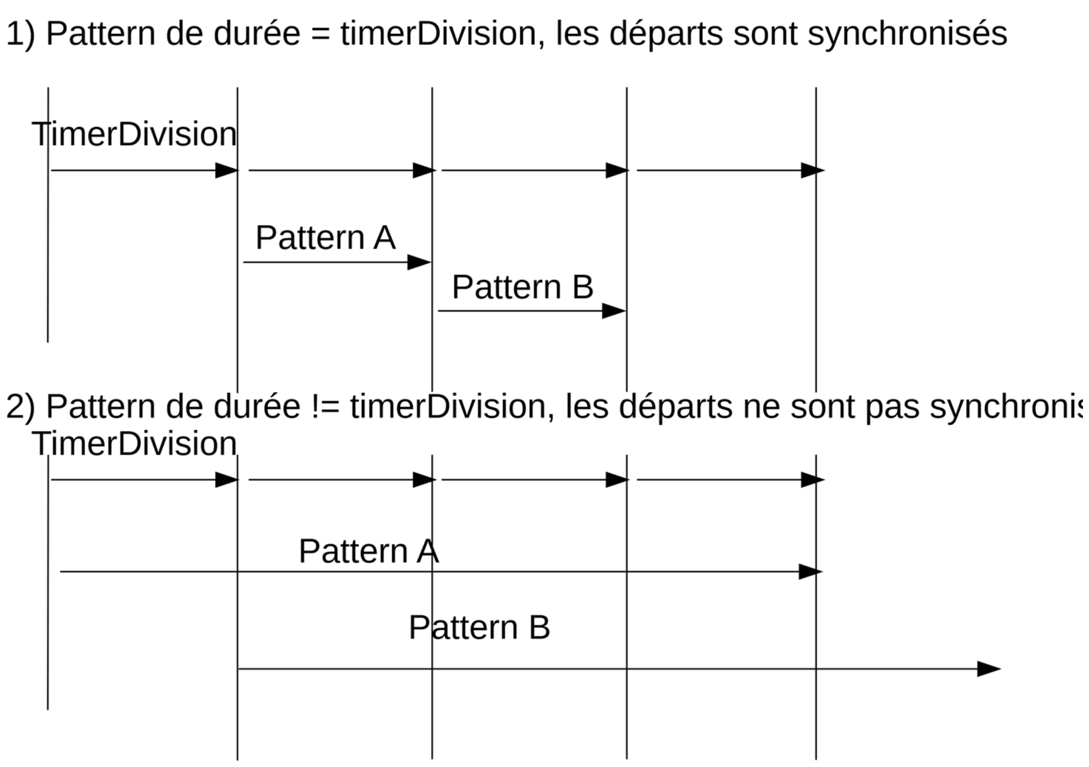


For pieces with patterns of the same duration, the *tick* can be given this duration (a certain number of pulses), which ensures that the pattern starts at the same instant.

As mentioned above, when you want to introduce patterns of different durations, you can give the *tick* a value corresponding to the duration of the shortest pattern. Care must be taken to match *tick* duration to pattern duration. If a pattern has a duration that is not a multiple of a *tick,* Skini will not play it and will give an error message. If you set the *tick to* 4 pulses and you have a pattern with 3 pulses or 5 pulses, it will not be played.

The combination of *tick* and pattern durations can lead to complex behaviors. Let's take the example where patterns have multiple *tick* durations. Let's take patterns with the same duration, for example 16 pulses and a *tick* equal to 4 pulses. If pattern A is requested (queued) at time T and pattern B at time T + 5 pulses, for example (between T + 5 and T + 8). Pattern B will start a *tick* cycle after pattern A. So, the 16-pulse patterns, although all of the same duration, can be staggered according to multiples of the *tick*.

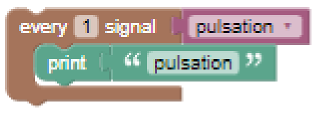
The following diagram illustrates the mechanism. Patterns A and B are on two different instruments. *TimerDivision* is equivalent to *tick*.


1) Pattern de durée = timerDivision, les départs sont synchronisés
TimerDivision
Pattern A
Pattern B
2) Pattern de durée != timerDivision, les départs ne sont pas synchronis
TimerDivision
Pattern A
Pattern B

When using different pattern durations, it's important to ensure musical consistency throughout the *tick* cycle.

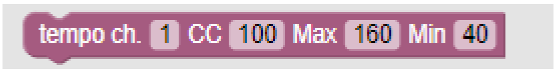
8.4.2 Using the pulse signal

In a room, it is possible to manage time using pulses. To do this, activate "Pulsation ON" in the room parameters. It is then possible to manage a pulsation signal (declared by default). Here's an example of how to display a message for each pulsation.

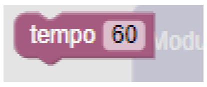


8.5 Tempo

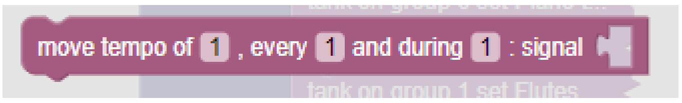
When synchro Link is not used, tempo changes from the orchestration require the DAW to be set up to receive tempo-based control changes. We have seen that a MIDI control bus is used to send MIDI information to the DAW. This is the bus used for tempo controls. In the case of Ableton, the control function requires you to declare the parameters used by Live for this control, i.e. a max and min value for the tempi. The following block sets these parameters.


tempo ch. 1 CC 100 Max 160 Min 40

Once these parameters have been set, or directly when using Link. The block

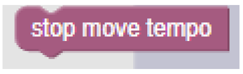

tempo 60

Allows you to intervene on tempo at any point in the orchestration. The


move tempo of 1 , every 1 and during 1 : signal

automates tempo changes. It allows you to vary the tempo by a fixed value at "every" occurrence of the signal given as a parameter. The tempo change is reversed after "during" occurrence of the same signal. This block makes it easy to integrate tempo movement without programming.

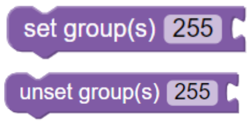
To interrupt tempo movements, apply the:


stop move tempo

8.6 Reservoirs and pattern groups

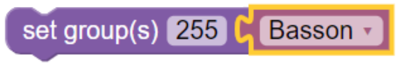
Although both are pattern set processing, they are two quite different classes of process. The groups are controlled by signals, and the activation and deactivation actions are not blocking.

8.6.1 Pattern groups

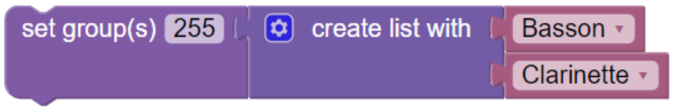

set group(s) 255

unset group(s) 255

User groups are defined in the piece configuration file. They are numbered from 0 to 254. Group 255 is in fact all groups. The following block activates the Basson group for all users:

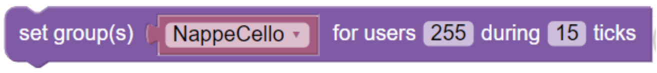

set group(s) 255 Basson

You can activate several groups with lists, e.g.:

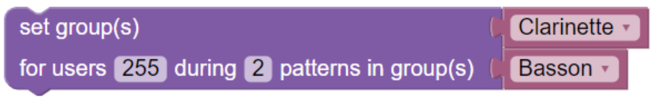

set group(s) 255 create list with Basson Clarinette

8.6.2 Group-related actions

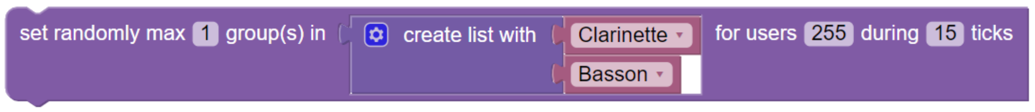
Skini provides high-level blocks to enable complex actions with groups.


set group(s) NappeCello for users 255 during 15 ticks

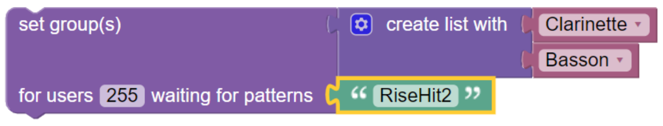
activates one or more groups during a period. Lists can be used instead of a single group.


set group(s) Clarinette
for users 255 during 2 patterns in group(s) Basson

activates one or more groups waiting for patterns to be played in one or more groups.



randomly activates "max" groups from a list over a period.



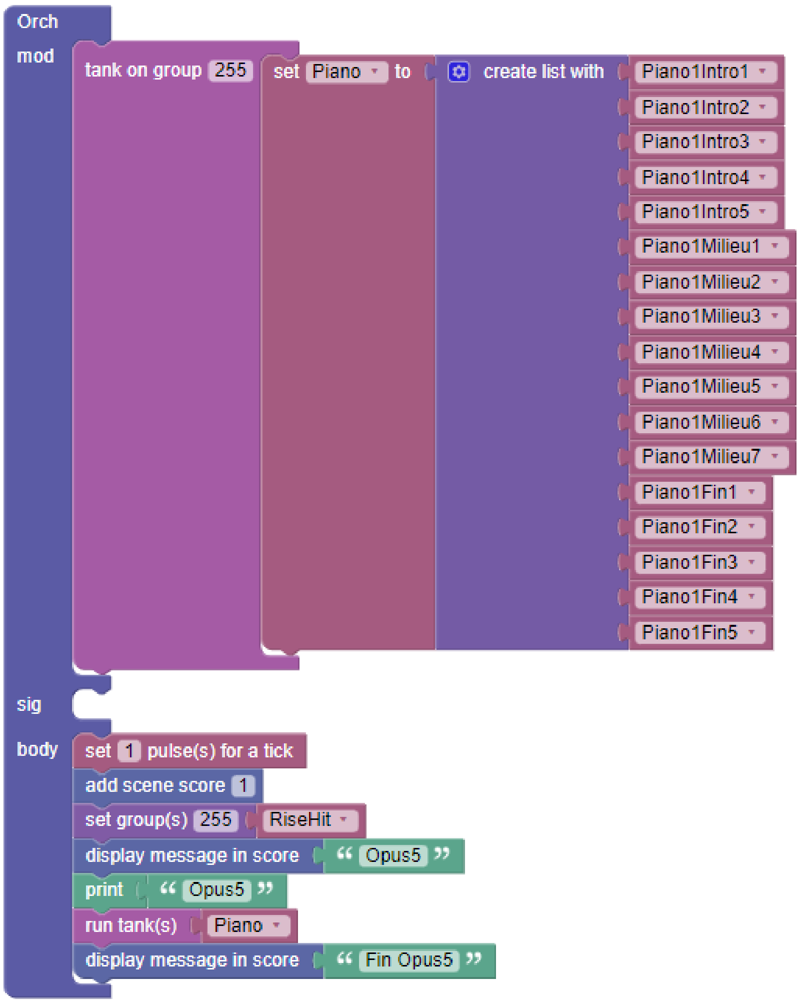
activates one or more groups waiting for one or more specific patterns.

Note: Groups are Blockly variables, patterns are Blockly strings.

8.6.3 Reservoir creation

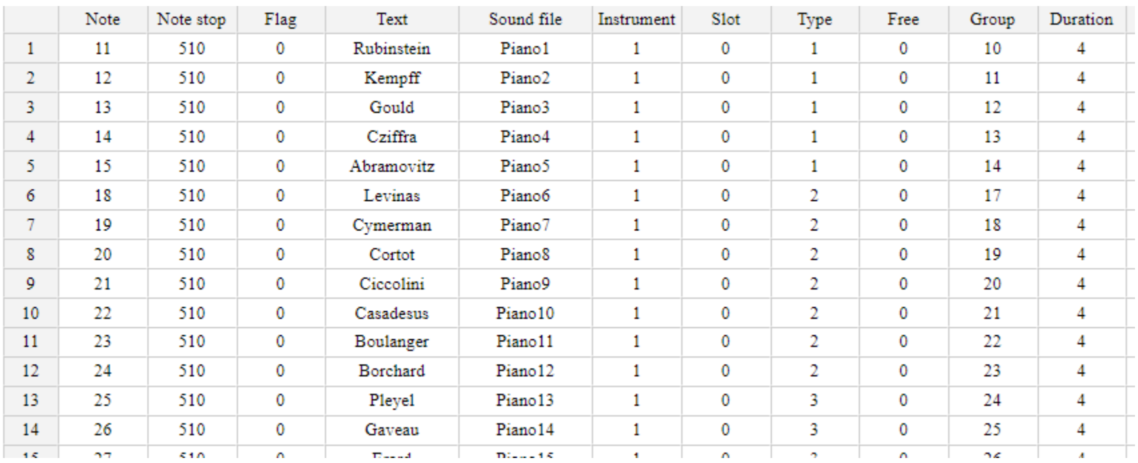
A reservoir is a three-level module and is therefore placed in the "Mod. Orchestration field. First, a list of patterns, then a variable associated with this list. The variable is placed in a "tank". A tank is associated with a user group or all users. Here's an example of a "percussion" tank of percussion patterns assigned to all user groups (255). Patterns are variables or character strings containing pattern names as described in the pattern csv file.

Here's an example of reservoir use:



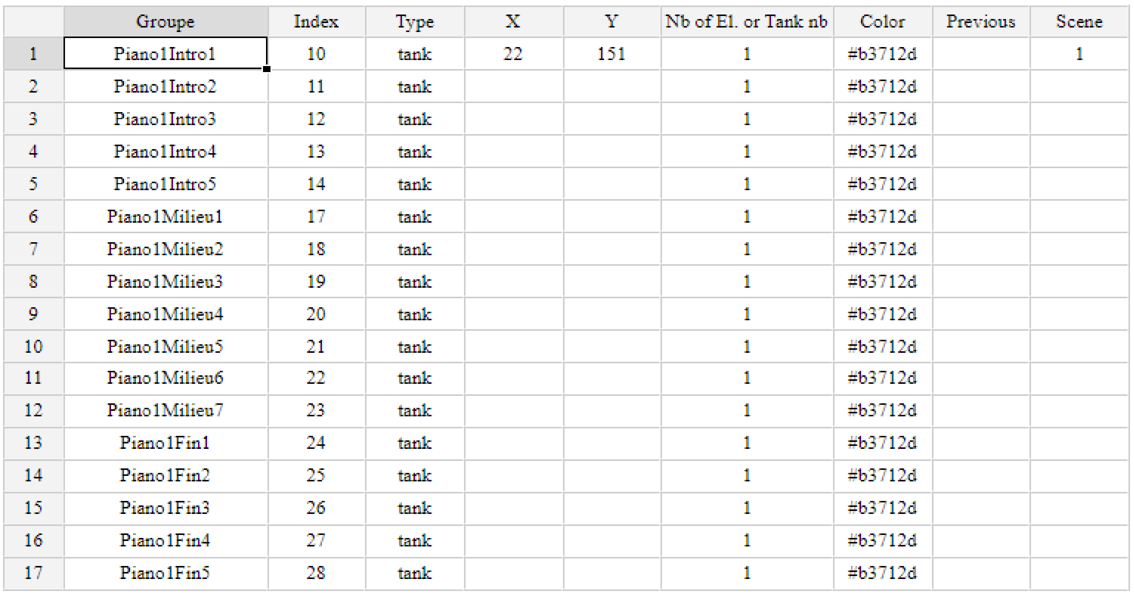
```
Orch
mod   tank on group 255
        set Piano to  create list with  Piano1Intro1
                                        Piano1Intro2
                                        Piano1Intro3
                                        Piano1Intro4
                                        Piano1Intro5
                                        Piano1Milieu1
                                        Piano1Milieu2
                                        Piano1Milieu3
                                        Piano1Milieu4
                                        Piano1Milieu5
                                        Piano1Milieu6
                                        Piano1Milieu7
                                        Piano1Fin1
                                        Piano1Fin2
                                        Piano1Fin3
                                        Piano1Fin4
                                        Piano1Fin5
sig
body  set 1 pulse(s) for a tick
      add scene score 1
      set group(s) 255  RiseHit
      display message in score  " Opus5 "
      print  " Opus5 "
      run tank(s)  Piano
      display message in score  " Fin Opus5 "
```

With patterns:



	Note	Note stop	Flag	Text	Sound file	Instrument	Slot	Type	Free	Group	Duration
1	11	510	0	Rubinstein	Piano1	1	0	1	0	10	4
2	12	510	0	Kempff	Piano2	1	0	1	0	11	4
3	13	510	0	Gould	Piano3	1	0	1	0	12	4
4	14	510	0	Cziffra	Piano4	1	0	1	0	13	4
5	15	510	0	Abramovitz	Piano5	1	0	1	0	14	4
6	18	510	0	Levinas	Piano6	1	0	2	0	17	4
7	19	510	0	Cymerman	Piano7	1	0	2	0	18	4
8	20	510	0	Cortot	Piano8	1	0	2	0	19	4
9	21	510	0	Ciccolini	Piano9	1	0	2	0	20	4
10	22	510	0	Casadesus	Piano10	1	0	2	0	21	4
11	23	510	0	Boulanger	Piano11	1	0	2	0	22	4
12	24	510	0	Borchard	Piano12	1	0	2	0	23	4
13	25	510	0	Pleyel	Piano13	1	0	3	0	24	4
14	26	510	0	Gaveau	Piano14	1	0	3	0	25	4
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

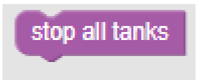
And the groups:



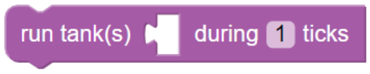
	Groupe	Index	Type	X	Y	Nb of El. or Tank nb	Color	Previous	Scene
1	Piano1Intro1	10	tank	22	151	1	#b3712d		1
2	Piano1Intro2	11	tank			1	#b3712d		
3	Piano1Intro3	12	tank			1	#b3712d		
4	Piano1Intro4	13	tank			1	#b3712d		
5	Piano1Intro5	14	tank			1	#b3712d		
6	Piano1Milieu1	17	tank			1	#b3712d		
7	Piano1Milieu2	18	tank			1	#b3712d		
8	Piano1Milieu3	19	tank			1	#b3712d		
9	Piano1Milieu4	20	tank			1	#b3712d		
10	Piano1Milieu5	21	tank			1	#b3712d		
11	Piano1Milieu6	22	tank			1	#b3712d		
12	Piano1Milieu7	23	tank			1	#b3712d		
13	Piano1Fin1	24	tank			1	#b3712d		
14	Piano1Fin2	25	tank			1	#b3712d		
15	Piano1Fin3	26	tank			1	#b3712d		
16	Piano1Fin4	27	tank			1	#b3712d		
17	Piano1Fin5	28	tank			1	#b3712d		

8.6.4 Actions on reservoirs

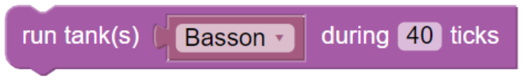
Tanks are Skini submodules whose parameters are patterns. They are blocking elements in a sequence. A tank will stop once it is empty, or once it has been killed with



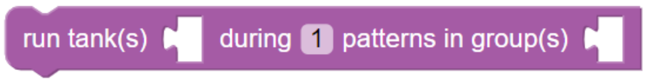
Functions are available to simplify reservoir management:

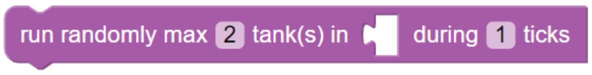


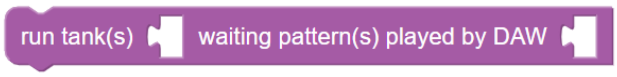
This block will launch one or more reservoirs given in parameter as a Blockly variable and "kill" this reservoir after "during" occurrences of a signal. The next block will leave the Basson reservoir patterns available for up to 40 ticks:



The operation of the following blocks is intyuitively understood:



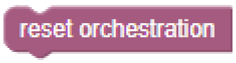




This block allows the DAW to consider the set of specific patterns.

Note: Group and tank names correspond directly to the music room configuration file.

Notes: All **group** and **reservoir** processing uses **Blockly variables**. Everything directly related to **patterns** uses strings.



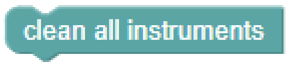
Resets the orchestration matrix, all active groups are deactivated.

8.7 INSTRUMENT QUEUES

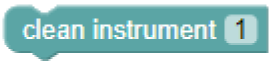
Queues are associated with instruments. They are created according to the pattern description in the pattern csv file.

8.7.1 Emptying queues

These blocks are useful when the composer no longer wants to hear any pending patterns for an instrument. He can clear all queues with:



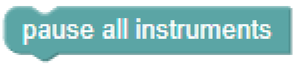
The skini client clears its choice list as soon as a clean all queue is performed. It can empty the queue of a particular instrument by giving its index with:



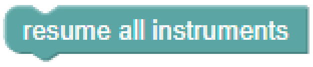
The skini client empties its choice list as soon as a clean queue is performed. It does not take into account the FIFO number, as does controleAbleton.js. (The block does not perform a cleanChoiceList).

Please note: queues, like groups in the audience, are set with **indexes** starting from **0.** Score displays are set with parameters starting at 1. (The best score is 1).

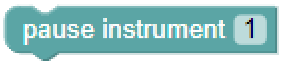
8.7.2 Pause and test queues



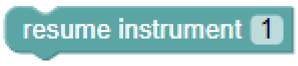
Stops pattern playback for all instruments.



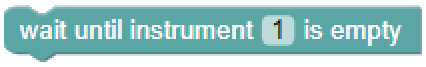
Resumes playing patterns for all instruments.



Stops playing patterns for an instrument.



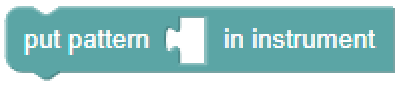
Resumes playing patterns for one instrument.



Blocks orchestration until an instrument's patterns have been played.

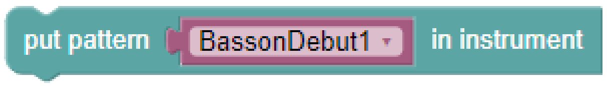
8.7.3 Queuing a specific pattern

The composer can make a specific pattern play imperatively with the block.



The block parameter is a string with the pattern name.

Ex :

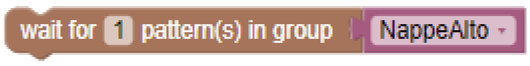


This block lets you introduce sequencer-type elements into a collaborative or generative music orchestration.

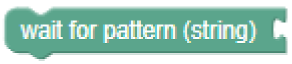
Warning: "put pattern" is sensitive to synchronization. Although Skini considers that system actions are performed in zero time, this is not actually the case. If, at a given moment, you need to receive synchronization, put a pattern in the FIFO, read the FIFO and send a command to the DAW, skini may find itself in a situation where the DAW command doesn't go through at the right synchronization tick. The solution is to introduce a delay before the DAW launches the pattern (launch synchronization). In this case, the DAW launches the command after a short delay, allowing the command to be received safely. The DAW then sends a synchronization tick and waits a while before triggering the patterns.

8.8 Patterns

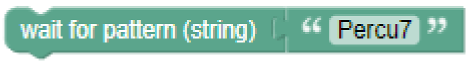
As Skini is mainly designed for processing groups of patterns and reservoirs, there are few functions dedicated to patterns. Here are two of them:



This block will wait for the execution (or request, depending on how the piece is set up) of a certain number of patterns belonging to a group.

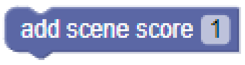


Is a block that waits for the execution (or request, depending on the piece's parameters) of a certain pattern passed as a parameter in the form of a character string. For example:

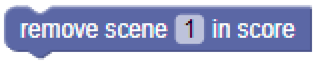


8.9 Orchestration display

The large screen display of the orchestration progress is controlled by the orchestration with specific blocks:

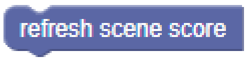

add scene score 1

A display can be made on several levels. This block displays one of the levels. Levels are associated with pattern groups in the piece configuration file.

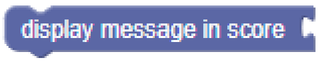

remove scene 1 in score

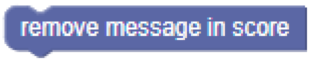
Disappears a level.

The next block is used to refresh the display following operations on queues, for example:


refresh scene score

The following blocks allow you to display and hide a message in a popup window on the big screen:


display message in score

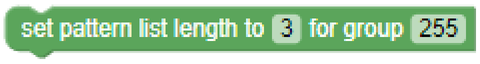

remove message in score

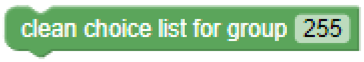
8.10 Game Skini

Skini offers a standard audience interface that allows participants to create lists of patterns and send them to the server to be played. The composer can use this interface to invent games, such as "finding the right sequence of patterns within a group". Game scenarios are not programmed by the orchestrator. They are JavaScript functions on the server.

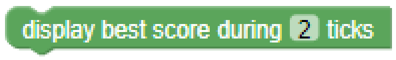
A basic model is proposed, which consists in defining pattern types and associating notes to the way the participant will organize his lists. The winner will be the one who accumulates the best lists during the play.

What is possible from the orchestration is to act on the participants' lists by defining list lengths and emptying the lists of a group.

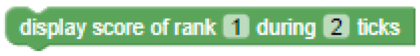

set pattern list length to 3 for group 255


clean choice list for group 255

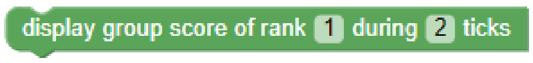
The next block displays the current best score:

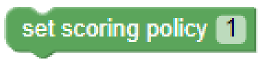

display best score during 2 ticks

To display the other scores (and the best one), use the

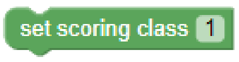

display score of rank 1 during 2 ticks

Ranking starts from 1.


display group score of rank 1 during 2 ticks



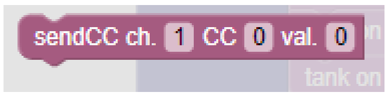
Defines the type of algorithm used to calculate scores. These algorithms can be found in the ./serveur/computeScore.js file.

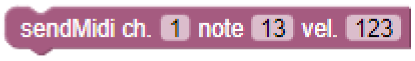


Defines the class used to calculate the score. The *class* is one of the parameters (type in index 7) of the patterns in the pattern description csv file. See the ./serveur/computeScore.js file on class processing.

8.11 MIDI Control and Control Change

MIDI Control Changes (CCs) can be issued from the orchestration. It's up to the composer to define and parameterize CC functions in the DAW. The channel on which CCs will be sent directly to the control must be defined.





This block is used to send a MIDI command.

The choice of MIDI channel depends on how the MIDI assignments to clips have been programmed in the DAW. **There's a difference between the channel seen on Skini, which starts at 1, and on DAWs, where it can start at 0.** This difference requires no attention apart from these controls. Don't hesitate to do a test and compare the "sendMIDI" activation with the configuration tool, for example.

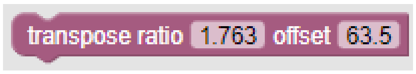
Notes on MIDI control: The definition of the MIDI bus (MIDI port) associated with the Block is set in the configuration file. Blocks therefore never use this parameter.

Notes on MIDI commands: Generally speaking, the operations performed by Skini blocks don't affect MIDI notes (in the Skini sense), but pattern groups, reservoirs and even patterns, albeit using their names. This is true except for the sendMIDI and sendCC blocks, which are addressed directly to a clip in the DAW.

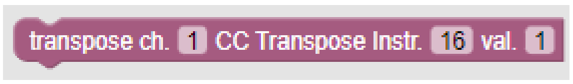
The composer must therefore pay close attention to MIDI channels in the case of blocks acting on Control Changes (CC) and MIDI commands.

8.12 Ableton Live specific

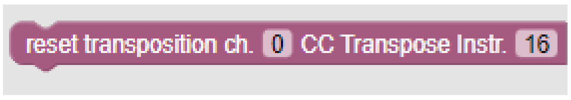
These blocks are used to control Ableton Live's transposition tool. Ratios and offsets are calculated according to this tool.



Defines ratio and offset for transposition calculation.



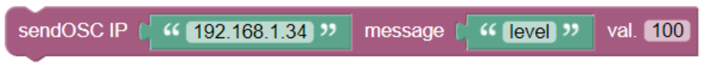
Transposition block, values are in semitones. The MIDI channel number must be given. The CC command controls Ableton's transposition tool.



Block to reset transposition. The previous command performs an increment.

8.13 OSC COMMANDS FOR RASPBERRY

To send an OSC command from the orchestrator, use the sendOSC block in the DAW menu. Here's an example:



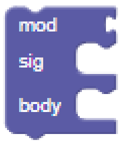
Which sends the message OSC */level* with the value 100 to the address 192.168.1.34 on the "Raspberry OSC Port" of the parameters (associating OSC and Raspberry is clumsy!!). In the current version, only one value is associated with the command.

8.14 ADVANCED BLOCKS

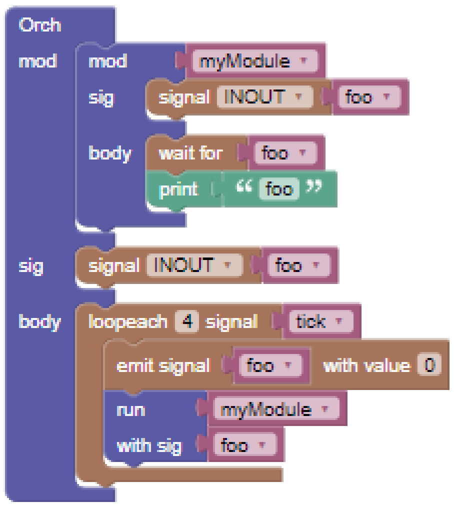
It's possible to create complex orchestrations without knowing the details of HipHop, but for functions based on non-standard blocks there is a series of menus for programming in HipHop with Blockly. This requires a basic knowledge of synchronous reactive programming.

These menus are: Signals, Signals Advanced and Module Advanced.

In addition to tanks, Skini sub-modules can also be created.



These sub-modules contain input/output signals that must be created in the signal field. The body of the sub-module is in body. Here's an example of a sub-module:



A submodule is executed with the run command:

8.15 Control Interface Z

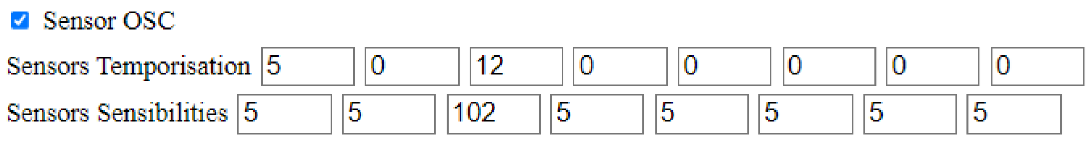
Skini integrates a **16-input (8ana - 8num) OSC network** card and a **MiniWi** Interface Z card. These cards emit OSC messages according to the sensors connected. The 8ana-8num operates on a wired Internet network, while the MiniWi uses WiFi. The Skini interface takes into account the 8 analog inputs of the 8ana-8num and the 4 inputs of the MiniWi, i.e. 12 possible sensors.

Card configuration is done in *ipConfig.json* with:

- *interfaceZIPaddress*, IP address of InterfaceZ 8ana-8num board.
- *portOSCToInterfaceZData*, for the port that sends commands from the 8ana-8num's analog and digital sensors
- *portOSCToInterfaceZMidi*, for the port that sends commands from the 8ana-8num board's MIDI interface (not currently used).
- *portOSCFromInterfaceZ*, for the port on the board that receives MIDI commands from the 8ana-8num
- *portOSCFromInterfaceZMiniWi*, for receiving OSC messages from MiniWI.

Configuration example: The interface is programmed as 192.168.1.250 with data port 3005 and MIDI port 3006. The system is wired to a switch with IP address 192.168.1.251 for the PC.

Sensor settings are made in the parameter window:



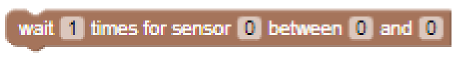
☑ Sensor OSC

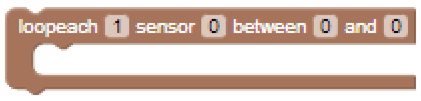
Sensors Temporisation	5	0	12	0	0	0	0	0
Sensors Sensibilities	5	5	102	5	5	5	5	5

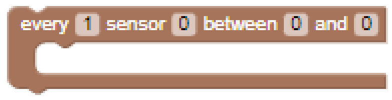
The time delay defines the frequency at which the information sent by the sensors is to be taken into account. In fact, the Interface Z 8ana-8num board systematically sends the status of the sensors at a frequency set by a potentiometer, which can be varied from 100 to 3000 messages per second. To avoid slowing down Skini's response times, a *thread* is dedicated to processing these messages. The *timers* set the frequency at which messages are taken into account for each sensor. 15 means that only one message out of 15 is considered. For example, for 100 messages per second from the card, a signal will be sent to Skini every 150 ms, instead of every 10 ms.

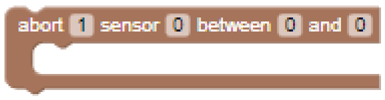
Sensitivity is used to take into account the permanent variations of analog sensors. If the variation (plus or minus) with the value taken into account at the previous time delay is less than the sensitivity, no signal is emitted. Here again, the aim is to avoid overloading Skini with irrelevant signal information.

Skini offers a series of blocks to specifically manage information from Interface Z sensors. These blocks are dedicated to :



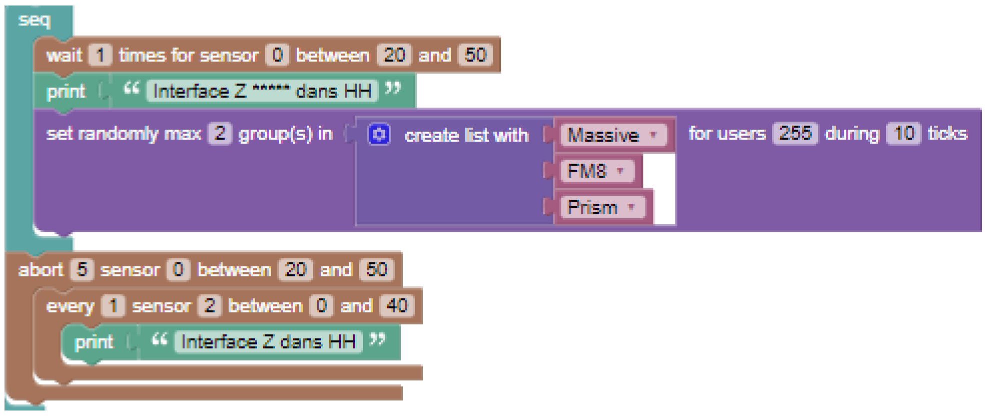






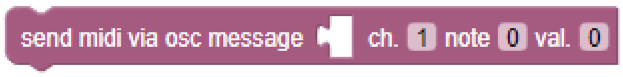
They have the same structure. *sensor* is the sensor number on Interface Z boards. Sensors from 0 to 7 are those on the 8ana-8num board, and from 8 to 11 those on the MiniWi board. As the values supplied by analog sensors fluctuate, the signals are considered according to a range between a minimum and a maximum.

Programming example:


seq
wait 1 times for sensor 0 between 20 and 50
print " Interface Z ***** dans HH "
set randomly max 2 group(s) in
create list with
Massive
FM8
Prism
for users 255 during 10 ticks
abort 5 sensor 0 between 20 and 50
every 1 sensor 2 between 0 and 40
print " Interface Z dans HH "

Order to be reviewed, more operational (1/9/2022)

This command is used to send a MIDI command via the Z interface OSC. This is a very specific use case.



8.16 Programming stinger transitions

The principle is to associate a transition pattern with a group of patterns (or a pattern that is a singleton). In the world of video game audio, this is known as a *stinger*. Ideally, a *stinger* should be associated with a specific transition from a group of patterns A to a group of patterns B. The difficulty with Skini is that the automaton has no vision of how the FIFOs are filled. The automaton can't identify the "temporal" sequence of two patterns in two different FIFOs.

In the *selection reaction* scenario, the PLC can see how the FIFOs are filling up, but there's no provision for managing their history (if you need to create a sort of duplicate FIFO in HH, or imagine signals coming from the FIFOs to HH). This management could even be quite complicated if patterns have variable durations.

In the *runtime reaction* scenario, we would only need a view of the patterns waiting in the FIFOs to locate the right successor to A. That is, a B in a waiting position at least as long as A in its FIFO.

In fact, Skini is designed to manage the processes with which FIFOs fill up, but not their organization, except in the case of priority management in queues, as explained in chapter Queue priority, p.39.

First, we'll simplify the problem by dealing with the case where we want to associate one or more *stingers with* each pattern in a group of patterns A. In this case, all we must do is wait for the execution signal of a pattern from A and then launch a pattern S as an offset or launch a *stinger* that integrates the offset. So, it's easy to make *stingers* "on exit".

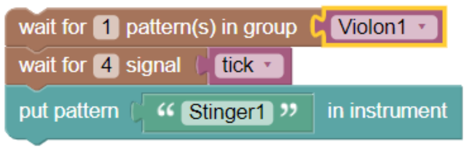
8.16.1 Reaction to execution

This scenario is straightforward in the case of *runtime feedback, i*.e. when the pattern is launched in the DAW. To manage S's delay, we need to set a *tick* that allows us to take into account a delay when the stinger is launched. Note that *tick* must be at least the duration of this delay (cf. Time programming, p.24).

A *stinger* can be programmed as follows:

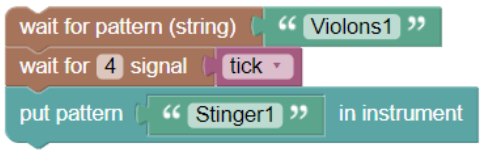
```
1. Wait for patternAIN
2. Wait for X tick
3. Put pattern Stinger in instrument!
```

An example for violin patterns of 8 ticks with a stinger launching at the 4th *tick*. Here the *tick* would be worth a pulse.


wait for 1 pattern(s) in group Violon1
wait for 4 signal tick
put pattern " Stinger1 " in instrument

8.16.2 Pattern feedback from DAW

The logic is the same, but the stinger principle is dependent on a particular pattern. This case works even when the reaction is *to the selection* because the pattern signal is emitted by the DAW when the pattern is launched.


wait for pattern (string) " Violons1 "
wait for 4 signal tick
put pattern " Stinger1 " in instrument

Note: Rather than counting *ticks, the stinger* can be designed to include an initial silence to offset the sound. There are no constraints on pattern duration.

8.17 Queue priority

Caution: Do not use a FIFO modification algorithm in interactive music if the algorithm can remove patterns from the FIFO queues, as this creates blocking situations on clients waiting for a return on the set of requested patterns.

This function can be used for generative music. FIFO processing can be activated. The room configuration must be updated:

Algo Fifo management at 1

(the value 1 assigned means that we can develop several types of algorithms and give them different identifiers).

For each pattern in the csv file, a pattern type must be defined in index 7 of each line. Five pattern types are defined. D: Start, M: Middle, F: End. N: Neutral (no processing) and P: Bread (one bread = bad pattern). The type is declared by a numerical value in the pattern csv configuration file: 1 for D, 2 for M, 3 for end and 4 for neutral, 5 for "bread". Type P is used in interactive game contexts, where a player selects a pattern that doesn't sound in the room or in the skini client's pattern list.)

To improve the structure of musical phrases, we look at the state of a queue before adding a pattern. As we write in a FIFO by adding a last element and bind by removing the first, we scan a FIFO starting from the end (last addition) to intervene on the last patterns put in the FIFO, i.e. the most recent ones. Here's how the algorithm works:

A) To add an F pattern to a queue:

1. If there are 2 Ds in a row, we insert F between them.
2. If there are 2 M in a row, we insert F between them.
3. If the last element in the queue is already an F, we look for a D followed by an M. If we find one, we put the F to be stacked just after this D.
4. Otherwise, we stack F (which gives two Fs in a row).

We can therefore have sequences of F

B) To add a D pattern to a queue:

1. If there is an F in the queue without a preceding D, we insert D before this F.
2. If there is an M in the queue without a D before it, we insert D before this M.
3. If not, stack D (giving two Ds in a row).

We can therefore have sequences of D

C) To add an M pattern to a queue:

1. If there is a D in the line immediately followed by an F, we put M between the two.
2. If there are 2 Ds in a row, we put M between them.
3. If there are 2 Fs in a row, we put M between them.
4. Otherwise, we stack M (which results in two M's in a row, but is not a problem).

If you don't want a sequence of D's or F's, do nothing in cases 1.4 and B.3. This is possible in generative music, but not in interactive music.

D) For a pattern N, no processing is performed.

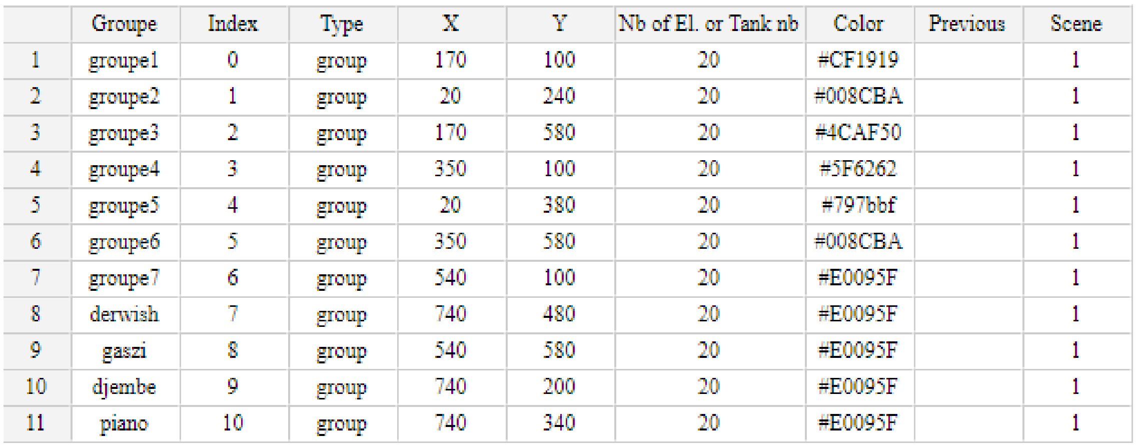
See the function `ordonneFifo()` in ./serveur/controleDAW.js for details of the algorithm actually used.

8.17.1 Example of game programming: find the percu

The aim of the game is for players to alternately choose patterns corresponding to a sound ambiance. Good choices increment a score, bad ones decrement it.

Patterns are only used in the form of groups. The simulator is used on a dedicated group that will play the musical ambience for which the player must find the corresponding patterns. Here we have player groups 0 and 1. User group 2 is the one that will play the soundscapes. To do this, "Simulator in a separate group" is activated in the parameters.

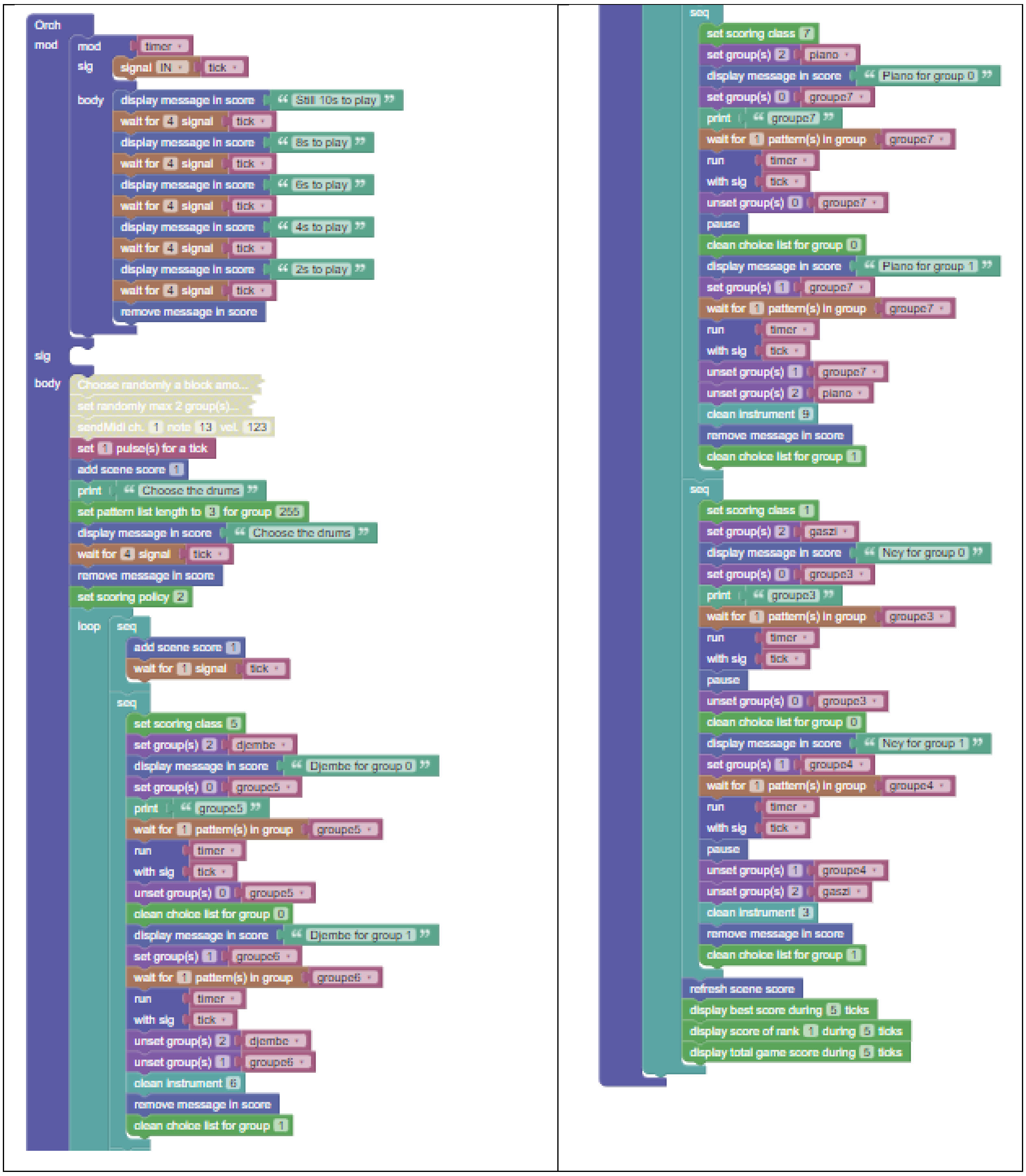
For patterns, we have sound ambiences in pattern groups 7 to 10. The other groups are for the players. These groups combine patterns of different types, from which the player must choose the right ones. There are as many patterns corresponding to the ambience as there are patterns not corresponding to the ambience. A random player has a 50% chance of success.



	Groupe	Index	Type	X	Y	Nb of El. or Tank nb	Color	Previous	Scene
1	groupe1	0	group	170	100	20	#CF1919		1
2	groupe2	1	group	20	240	20	#008CBA		1
3	groupe3	2	group	170	580	20	#4CAF50		1
4	groupe4	3	group	350	100	20	#5F6262		1
5	groupe5	4	group	20	380	20	#797bbf		1
6	groupe6	5	group	350	580	20	#008CBA		1
7	groupe7	6	group	540	100	20	#E0095F		1
8	derwish	7	group	740	480	20	#E0095F		1
9	gaszi	8	group	540	580	20	#E0095F		1
10	djembe	9	group	740	200	20	#E0095F		1
11	piano	10	group	740	340	20	#E0095F		1

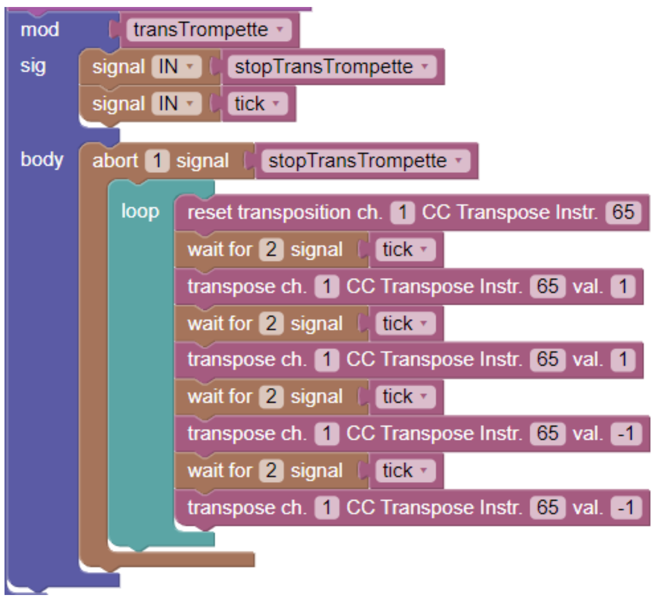
The maximum list length for players is 3 patterns.

When a player sends in a first list of up to three patterns, he'll only be able to play for a further 10 seconds. The timer in the module displays the remaining time in the "score" window. At the end of the game, the winner and scores are displayed.



```
Orch
mod   mod   timer
      sig   signal IN   tick
      body  display message in score  "Still 10s to play"
            wait for 4 signal  tick
            display message in score  "8s to play"
            wait for 4 signal  tick
            display message in score  "6s to play"
            wait for 4 signal  tick
            display message in score  "4s to play"
            wait for 4 signal  tick
            display message in score  "2s to play"
            wait for 4 signal  tick
            remove message in score
sig
body  Choose randomly a block amo...
      set randomly max 2 group(s)...
      sendMidi ch. 1 note 13 vel. 123
      set 1 pulse(s) for a tick
      add scene score 1
      print  "Choose the drums"
      set pattern list length to 3 for group 255
      display message in score  "Choose the drums"
      wait for 4 signal  tick
      remove message in score
      set scoring policy 2
      loop  seq
              add scene score 1
              wait for 1 signal  tick
            seq
              set scoring class 5
              set group(s) 2  djembe
              display message in score  "Djembe for group 0"
              set group(s) 0  groupe5
              print  "groupe5"
              wait for 1 pattern(s) in group  groupe5
              run  timer
              with sig  tick
              unset group(s) 0  groupe5
              clean choice list for group 0
              display message in score  "Djembe for group 1"
              set group(s) 1  groupe6
              wait for 1 pattern(s) in group  groupe6
              run  timer
              with sig  tick
              unset group(s) 2  djembe
              unset group(s) 1  groupe6
              clean instrument 6
              remove message in score
              clean choice list for group 1
            seq
              set scoring class 7
              set group(s) 2  piano
              display message in score  "Piano for group 0"
              set group(s) 0  groupe7
              print  "groupe7"
              wait for 1 pattern(s) in group  groupe7
              run  timer
              with sig  tick
              unset group(s) 0  groupe7
              pause
              clean choice list for group 0
              display message in score  "Piano for group 1"
              set group(s) 1  groupe7
              wait for 1 pattern(s) in group  groupe7
              run  timer
              with sig  tick
              unset group(s) 1  groupe7
              unset group(s) 2  piano
              clean instrument 9
              remove message in score
              clean choice list for group 1
            seq
              set scoring class 1
              set group(s) 2  gaszi
              display message in score  "Ney for group 0"
              set group(s) 0  groupe3
              print  "groupe3"
              wait for 1 pattern(s) in group  groupe3
              run  timer
              with sig  tick
              pause
              unset group(s) 0  groupe3
              clean choice list for group 0
              display message in score  "Ney for group 1"
              set group(s) 1  groupe4
              wait for 1 pattern(s) in group  groupe4
              run  timer
              with sig  tick
              pause
              unset group(s) 1  groupe4
              unset group(s) 2  gaszi
              clean instrument 3
              remove message in score
              clean choice list for group 1
            refresh scene score
            display best score during 5 ticks
            display score of rank 1 during 5 ticks
            display total game score during 5 ticks
```

8.17.2 Example of loop transposition



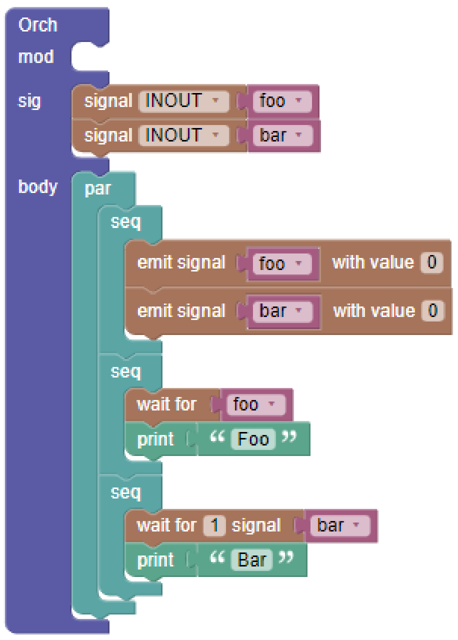
```
mod   transTrompette
sig   signal IN  stopTransTrompette
      signal IN  tick
body  abort 1 signal  stopTransTrompette
        loop  reset transposition ch. 1 CC Transpose Instr. 65
              wait for 2 signal  tick
              transpose ch. 1 CC Transpose Instr. 65 val. 1
              wait for 2 signal  tick
              transpose ch. 1 CC Transpose Instr. 65 val. 1
              wait for 2 signal  tick
              transpose ch. 1 CC Transpose Instr. 65 val. -1
              wait for 2 signal  tick
              transpose ch. 1 CC Transpose Instr. 65 val. -1
```

8.18 Some subtleties of synchronous reactive programming

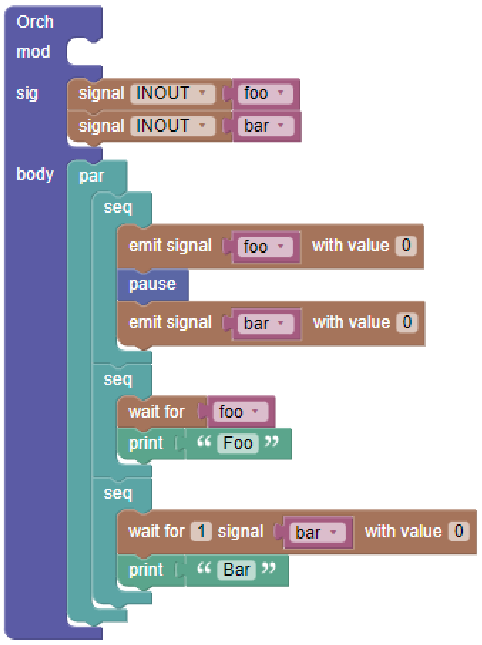
8.18.1 Immediate reaction

Synchronous reactive programming is intuitive, because it works in much the same way as we express ourselves. "I'm doing this, waiting for that. If something happens, then I'll stop doing something...". Nevertheless, it's very precise semantics can reveal inconsistencies that are hardly visible in our expressive habits. This semantics at program initialization makes the difference between a signal taken into account immediately or after an initial reaction.

If you run the program on the left, you'll see that only "Foo" appears on the console. There's a subtle difference between "wait for" and "wait for 1 signal". Wait for" reacts immediately, "wait for 1 signal" needs an initial transmission and reaction to start. In orchestration, this subtlety is rarely a problem.



```
Orch
mod
sig   signal INOUT foo
      signal INOUT bar
body  par
        seq
          emit signal foo with value 0
          emit signal bar with value 0
        seq
          wait for foo
          print “Foo”
        seq
          wait for 1 signal bar
          print “Bar”
```

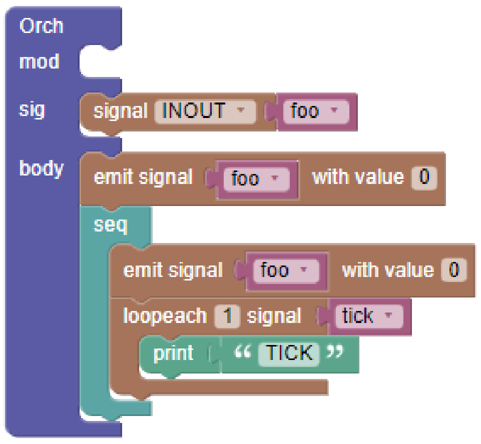


```
Orch
mod
sig   signal INOUT foo
      signal INOUT bar
body  par
        seq
          emit signal foo with value 0
          pause
          emit signal bar with value 0
        seq
          wait for foo
          print “Foo”
        seq
          wait for 1 signal bar with value 0
          print “Bar”
```

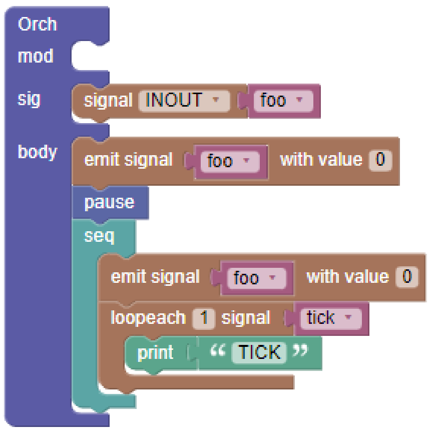
In Skini, only "wait for" and "wait for pattern" react to the first broadcast.

8.18.2 Dual transmission of the same signal

The program on the left will cause a problem at runtime. The "foo" signal will be sent twice on the same reaction. This message will be displayed in the console: TypeError: Can't set single signal `foo' value more than once.



```
Orch
mod
sig   signal INOUT foo
body  emit signal foo with value 0
      seq
        emit signal foo with value 0
        loopeach 1 signal tick
          print “TICK”
```



```
Orch
mod
sig   signal INOUT foo
body  emit signal foo with value 0
      pause
      seq
        emit signal foo with value 0
        loopeach 1 signal tick
          print “TICK”
```

The "pause" block in the right-hand program allows you to delay the second emission by one reaction.

8.19 For generative music

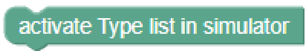
Most of Skini's basic functions can be used in conjunction with the simulator to produce music automatically.

We can reorganize the queues on the server. This is set by "Algo Fifo management" in the "Parameters" window. 0 means no processing. 1 activates a start (type 1), middle (type 2), end (type 3) and neutral (type 4) sorting algorithm.

There's another way to control the organization of pattern sequences from within the simulator. This function is richer, as it allows you to reorganize the sequence processing policy on the fly.

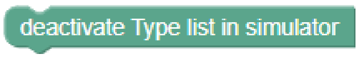
Several functions are dedicated to simulator control, allowing you to manage the construction of lists of patterns sent by the simulator. This makes it possible to structure musical phrases, for example, according to the types assigned to the patterns.

To activate the organization of lists sent by the simulator, insert the:



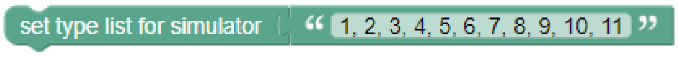
activate Type list in simulator

To disable organization during a piece:



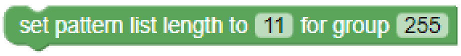
deactivate Type list in simulator

The simulator will build musical phrases based on the list of types defined by the:



set type list for simulator “ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ”

Here, among the patterns available, the simulator will arrange them in the order defined by the list of types, randomly selecting one pattern per type. If there is none for a type, it is ignored. For the process to work on all declared types, the list of patterns for clients (and therefore the simulator) must be the same size as the list of types. For example, according to the types defined above:



set pattern list length to 11 for group 255

Note 1: It is possible to order several instruments at the same time by managing different types for each instrument. See opus4 as an example.

Note 2: Music generation with type manipulation must also consider how Skini reacts to pattern activations. Activating "React on play" leaves the field open until the pattern is played, since there may be a delay between requesting a pattern and hearing it.

Note 3: The rate at which the simulator sends sequences has an impact on pattern repetition when using tanks if "React on play" is active.

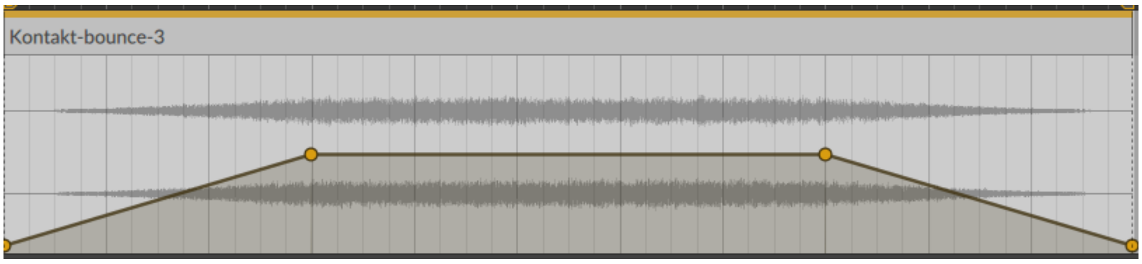
So, there's an alchemy to be achieved between reservoir, simulator rhythm and reaction model. This makes it possible to create sequences of patterns conforming to type models, and in polyphony, since several instruments can be "typed" at the same time. Overall, these functions make it possible to dynamically control and modify the probability of patterns appearing, and thus to organize the music accordingly.

8.20 Spatialization by tiling with the Pré device

Here's one way to set up spatialization effects by tiling, i.e. patterns that partially overlap over time.

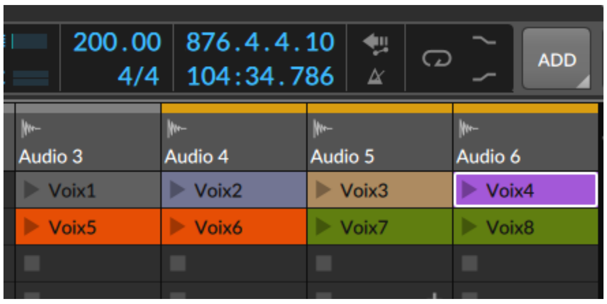
(Example space1.xml, space1.csv, space1.js).

The patterns here all have the same structure, with a Fade IN and a Fade OUT to allow patterns to overlap. We'll use a fast tempo to allow for quick overlaps, which we won't do here, but which will be possible.



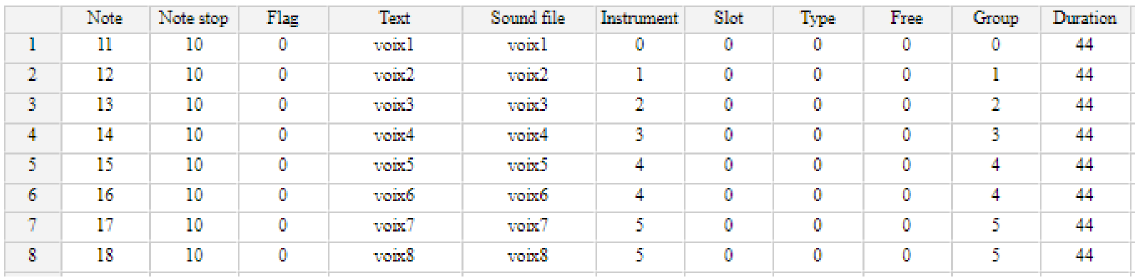
The patterns have a duration of 44 pulses, with Fade IN and OR of 12 pulses each.

Here are the patterns in bitwig Studio. The only thing to bear in mind when allocating patterns



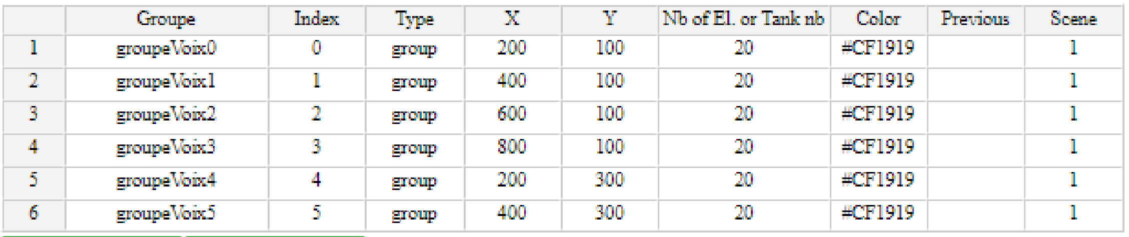
is that a pattern in progress must not be interrupted if a random process is used. This must be taken into account when assigning instruments and groups.

Here's an example of a descriptor. The example is given with a DAW, but it would be enough to assign Raspberry Pi to the Patterns.



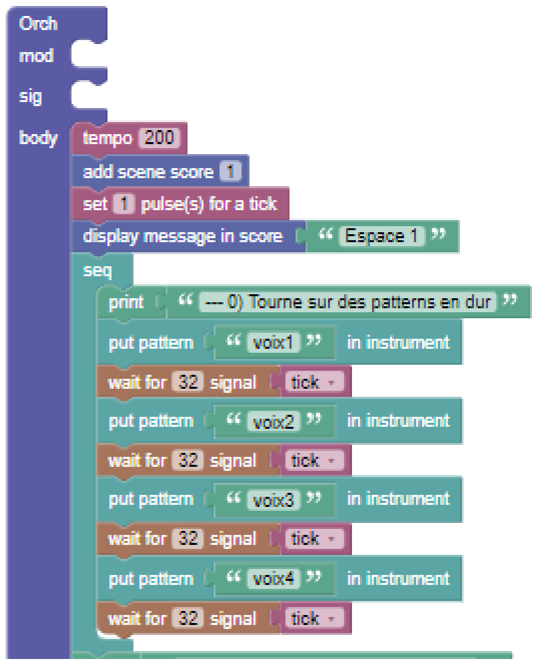
	Note	Note stop	Flag	Text	Sound file	Instrument	Slot	Type	Free	Group	Duration
1	11	10	0	voix1	voix1	0	0	0	0	0	44
2	12	10	0	voix2	voix2	1	0	0	0	1	44
3	13	10	0	voix3	voix3	2	0	0	0	2	44
4	14	10	0	voix4	voix4	3	0	0	0	3	44
5	15	10	0	voix5	voix5	4	0	0	0	4	44
6	16	10	0	voix6	voix6	4	0	0	0	4	44
7	17	10	0	voix7	voix7	5	0	0	0	5	44
8	18	10	0	voix8	voix8	5	0	0	0	5	44

Here is an example of groups:



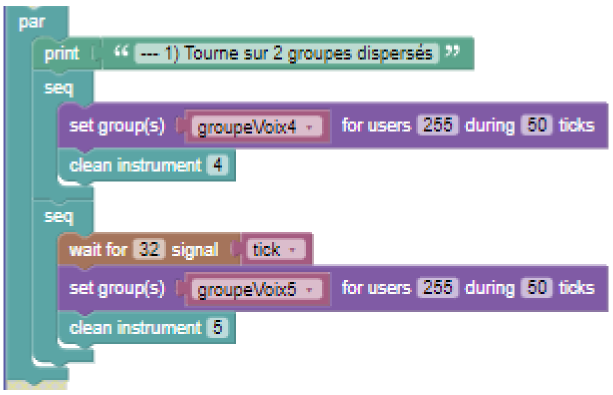
	Groupe	Index	Type	X	Y	Nb of El. or Tank nb	Color	Previous	Scene
1	groupeVoix0	0	group	200	100	20	#CF1919		1
2	groupeVoix1	1	group	400	100	20	#CF1919		1
3	groupeVoix2	2	group	600	100	20	#CF1919		1
4	groupeVoix3	3	group	800	100	20	#CF1919		1
5	groupeVoix4	4	group	200	300	20	#CF1919		1
6	groupeVoix5	5	group	400	300	20	#CF1919		1

For "hard" control, i.e. fixed, without random phenomena. Simply run the patterns one after the other, waiting for the Fade OUT to start at the 32nd beat.



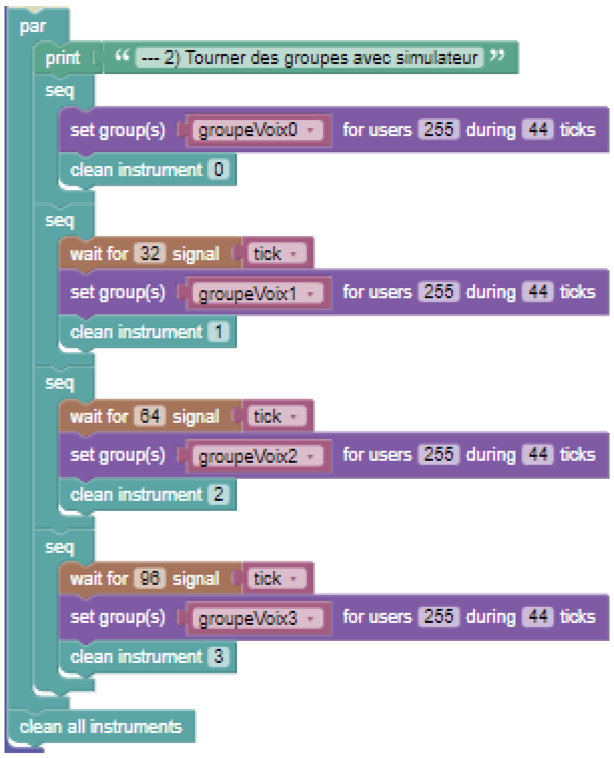
```
Orch
mod
sig
body  tempo 200
      add scene score 1
      set 1 pulse(s) for a tick
      display message in score  " Espace 1 "
      seq
        print  " --- 0) Tourne sur des patterns en dur "
        put pattern  " voix1 "  in instrument
        wait for 32 signal  tick
        put pattern  " voix2 "  in instrument
        wait for 32 signal  tick
        put pattern  " voix3 "  in instrument
        wait for 32 signal  tick
        put pattern  " voix4 "  in instrument
        wait for 32 signal  tick
```

To "run" two groups with a random selection in each group, simply shift the activation of the groups in parallel.



```
par
  print  " --- 1) Tourne sur 2 groupes dispersés "
  seq
    set group(s)  groupeVoix4  for users 255 during 50 ticks
    clean instrument 4
  seq
    wait for 32 signal  tick
    set group(s)  groupeVoix5  for users 255 during 50 ticks
    clean instrument 5
```

To run multiple groups, simply shift them in parallel. The example below gives the same result as the first "frozen" example, as we only have one pattern in each group.



```
par
  print  " --- 2) Tourner des groupes avec simulateur "
  seq
    set group(s)  groupeVoix0  for users 255 during 44 ticks
    clean instrument 0
  seq
    wait for 32 signal  tick
    set group(s)  groupeVoix1  for users 255 during 44 ticks
    clean instrument 1
  seq
    wait for 64 signal  tick
    set group(s)  groupeVoix2  for users 255 during 44 ticks
    clean instrument 2
  seq
    wait for 96 signal  tick
    set group(s)  groupeVoix3  for users 255 during 44 ticks
    clean instrument 3
clean all instruments
```

Note: It's the notion of the Skini instrument that makes tiling spatialization so easy. In an instrument, patterns are queued up, so all you have to do is play on instrument offsets to chain patterns that are in different instruments.

9 TEXT-BASED ORCHESTRATION PROGRAMMING

It is possible to program orchestrations using the HipHop.js language without using the graphical programming interface. The functions are similar, but the integration with JavaScript is richer since JavaScript can be used in the same way as HipHop.js. This way of working requires a good knowledge of JavaScript and HipHop.js programming.

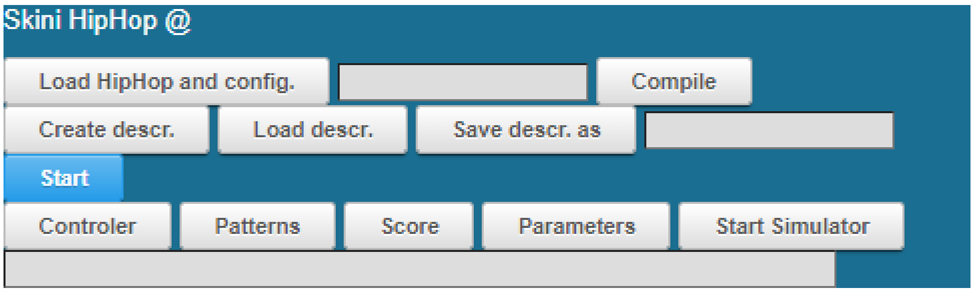
For HipHop.js programming, documentation is available at url :

http://hop.inria.fr/home/hiphop/index.html

HipHop.js orchestration files must be named as <*orchestration file*>.hh.js, the parameter file is the same as the GUI <*orchestration file*>.js. Parameters and patterns follow the same processes as those of the GUI.

9.1 CONTROL INTERFACE FOR TEXT PROGRAMMING

Orchestrations are loaded, compiled and executed via browser access to Skini: http:*//IP server*:8080/contrHH.



Most functions are like those in the Blockly window. The HipHop.js file is entered using the *Load HipHop and Config* button. You can recompile the HipHop.js file at any time without reloading it, using the *Compile* button.

Orchestration is a HipHop.js program that is loaded and executed by the Skini platform. It has a particular format in that certain functions are defined and indispensable for Skini to be able to execute the program.

A few examples are available:

- helloSkini.hh.js
- opus4.hh.js

There are two essential functions:

```
export function setServ(ser, daw, groupeCS, oscMidi, mix) {
  if (debug) console.log("hh_ORCHESTRATION: setServ");
  DAW = daw;
  server = ser;
  gcs =CS group;
  oscMidiLocal = oscMidi;
  midimix = mix;
}
```

Called in by Skini to set up all access to control functions.

```
export function setSignals(param){...}
```

which includes orchestration in the

```
  const Program = hiphop module() {...}
```

itself generated by the :

```
  const prg = new ReactiveMachine(Program, "orchestration");
```

9.2 SKINI FUNCTIONS VIA HIPHOP.JS (TO DO)

10 With musicians (not available with Node.js)

With musicians, Skini sets up a countdown before the first pattern is played in a queue.

In the room configuration file, to set up specific features for playing with musicians :

```
exports.withMusician = true;
```

This introduces an empty pattern before the first pattern in a FIFO.

ATTENTION: This offset is based on the *tick* (not the pulse) and must be a multiple of the *tick*, otherwise the player will freeze.

exports.decalageFIFOwithMusician = 8;

To specify the position of the pattern partition files in jpg format, in the piece configuration file we will have, for example :

```
exports.patternScorePath1 = "";
```

These paths are relative to the "./images" directory.

11 OSC interface

11.1 Principle

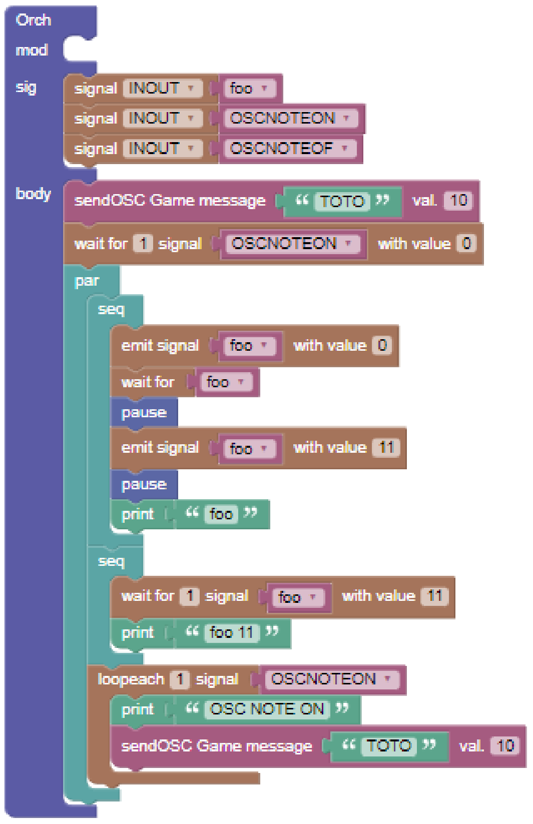
If a game development platform or controller can send OSC commands, it is possible to generate signals in the orchestration from external events, and conversely to generate OSC messages from the orchestration.

11.2 From orchestration

We need to create signals corresponding to the OSC messages received. The signals correspond to the messages without the "\" slashes. The message /FOO/BAR will give the signal FOOBAR in Skini. The transmitted message has no header slash and is added by Skini.

Note on OSC signals: Signals created for OSC should not have the same names as default signals. It's a good idea to define a common format for these signals, by adding an *OSC* or *Game* suffix or prefix, or something similar.

11.3 Implementation with orchestration



```
Orch
mod
sig   signal INOUT foo
      signal INOUT OSCNOTEON
      signal INOUT OSCNOTEOF
body  sendOSC Game message "TOTO" val. 10
      wait for 1 signal OSCNOTEON with value 0
      par
        seq
          emit signal foo with value 0
          wait for foo
          pause
          emit signal foo with value 11
          pause
          print "foo"
        seq
          wait for 1 signal foo with value 11
          print "foo 11"
        loopeach 1 signal OSCNOTEON
          print "OSC NOTE ON"
          sendOSC Game message "TOTO" val. 10
```

Example of a program for receiving and transmitting OSC commands:

- from an OSC transmitter to *serverIPAddress* on the *portOSCFromGame.*

- to an OSC receiver in *remoteIPAddressGame* on the *portOSCToGame.*

12 OSC with Bitwig Studio

It is possible to communicate Skini in OSC with Bitwig Studio, i.e. without using Processing as an OSC/Midi gateway. The Bitwig controller, Skini_0.control.js, behaves like the Processing gateway for Ableton.

On Bitwig, the Skini_0 controller has the same parameters as Processing, for example:

Remote OSC Ip address: 192.168.1.6

Remote OSC port out: 13000

Listening OSC: 12000

Midi information is sent to Bitwig as it is to Processing. There is therefore no change in the way Skini generates Midi commands for patterns or other commands. To maintain compatibility with the Processing gateway, Bitwig sends OSC commands for Midi to itself on a Midi channel in LoopMidi. (I can't send a Midi command directly to clip triggers). It is therefore necessary to declare a Generic Keyboard controller that receives midi commands from the Skini_0 controller.

There is therefore a Midi OUT port to be given in the Skini_0 controller to LoopMidi, which will be the Midi input port in the generic keyboard.

The Bitwig controller can send Midi commands from one of its controllers to Skini. This is the Midi OUT port of the Skini_0 controller. The interpretation of OSC commands from Midi to Skini can be found in server/midimix.js.

The correspondence between Bitwig's OSC commands and Skini is defined in the Skini_0.control.js and midimix.js files.

In addition, Bitwig emits an OSC tick "/BitwigTick". The calculation of this tick in the controller is not precise. It's based on Bitwig's transport bar.

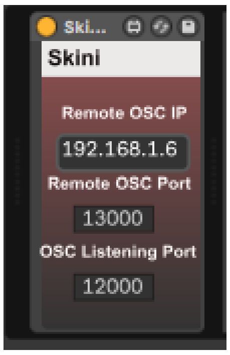
Example: technoBitwig piece on ABL3.

Conclusion: The only modifications made to Skini for a first Bitwig implementation are in midimixi.js and websocketServerSkini.js.

Note: Bitwig studio has no notion of bus (midi port). Bitwig receives messages in OSC and reroutes them for control via loopMidi. Bus parameters present in OSC commands sent to bitwig are therefore not taken into account.

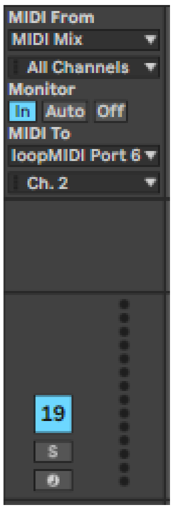
13 OSC WITH MAX4LIVE

It is possible to use Skini and Ableton without using the Processing gateway. To do this, we use Max for Live (M4L). To do this, create a Midi track into which you load the Max patch "SkiniOSC2.amxd". The track can receive a MIDI device as input, whose commands will be sent to Skini in OSC.

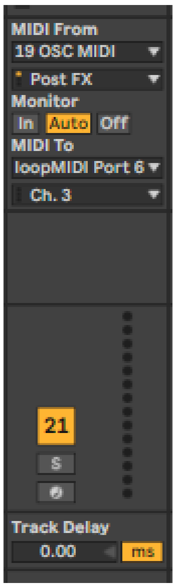

Ski...
Skini
Remote OSC IP
192.168.1.6
Remote OSC Port
13000
OSC Listening Port
12000

13.1 CONVERSION TRACK

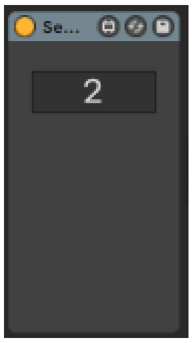
The Ableton Midi track must be routed as a Midi output to a control channel that will receive Skini OSC commands converted to Midi. In fact, it's not possible to speak directly to an Ableton Live control via M4L, so you must use a virtual cable (LoopMIDI, for example). Patch parameters concern OSC. Midi settings are made in the Ableton Live track.


MIDI From
MIDI Mix
All Channels
Monitor
In Auto Off
MIDI To
loopMIDI Port 6
Ch. 2

13.2 MIDI ROUTING TRACK

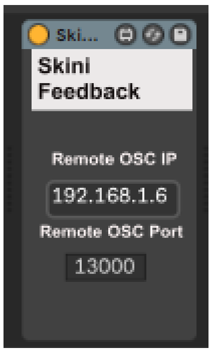


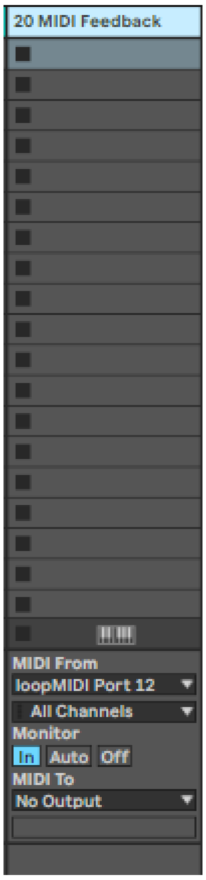
An Ableton Live bug complicates operations as soon as you have more than 127 notes associated with patterns in Skini. **M4L in an Ableton track cannot address anything other than Midi channel number 1**. To get around this problem, you need to create additional tracks, as many as there are multiples of 127 among the notes associated with the patterns in the pattern configuration file (.csv).



You need to load the "SendToMIDIChannel.amxd" M4L patch into these additional Midi tracks and set the correct channel on the "MIDI to" track. Skini channel 2 corresponds to Ableton channel 3. There is an offset of one unit between Skini and Ableton.

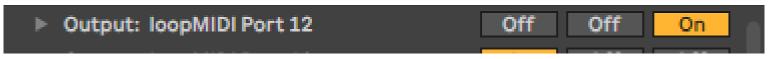
13.3 FEEDBACK TRACK





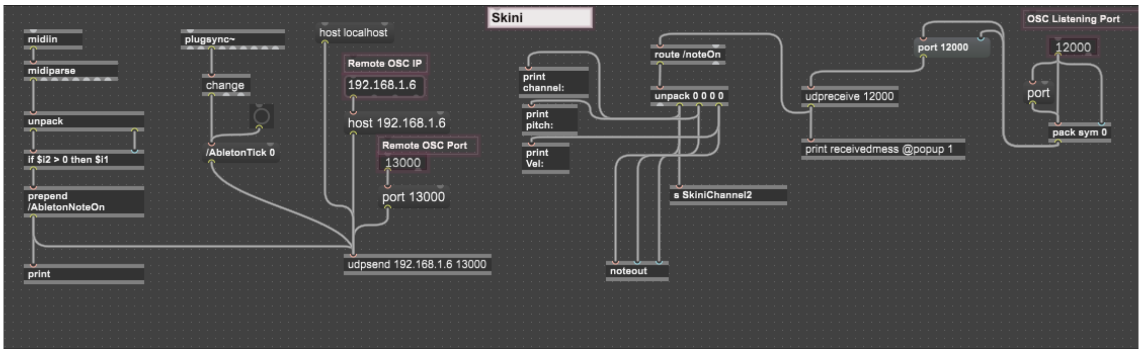
To process the Midi information sent by Ableton when a pattern is played (this function is only useful if the *patternSIgnal* signal is used in the orchestration), you need to add a Midi track with an M4L *SkiniAbletonFeedback.amxd* patch. This track has as its input port (MIDI From) a Midi port that receives "Remote Controls". For example:

Remote controls" are sent to Skini in OSC. Here again, we're limited to 127 Skini notes. To overcome this problem, it's simpler to use Processing, given the level of parameter complexity reached in Ableton.

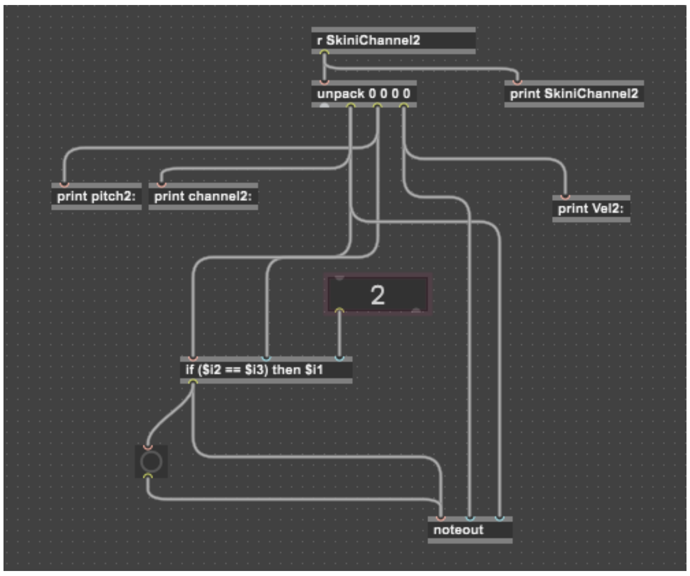


13.4 M4L PATCHES

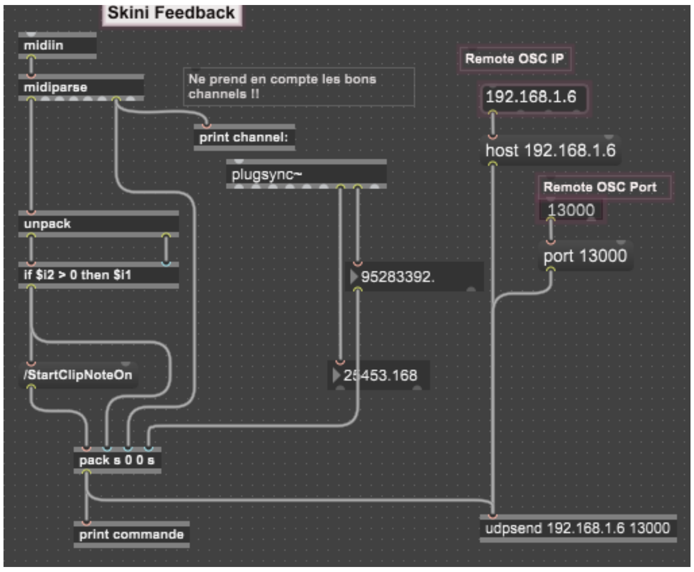
This is the *SkiniOSC2.amxd* patch:



The *SendToMIDIChannel.amxd* patch:



The *SkiniAbletonFeedback.amxd* ptach:



13.5 Conclusion OSC Skini with Ableton

Using Ableton Live with Skini is possible without Processing, but becomes complex as soon as you have more than 127 pattern notes in Skini, due to a design problem in M4L. This is a further demonstration of M4L's lack of maturity.

As an example, take Ableton's *technoDemoOSC.als.*

Remarks :

1) Assigning MIDI commands to clips in Ableton processes channels starting from 0. Skini, on the other hand, converts Midi channels starting from 1. This means that Skini commands actually start on channel 2 of Ableton's Midi control port. If you are processing fewer than 127 Skini notes, you can use only one *Midi routing* track, assigning *channel 2 on* the control Midi port as output.

2) *SkiniOSC2.amxd*'s *noteout* doesn't take into account the Midi channel given to it, but only the Midi channel at the track's output. The *SkiniOSC2.amxd* patch sends the Midi command to the *SendToMIDIchannel.amxd patch*, which compares the received channel with its parameter. This patch loses Midi notes for no reason, send/receive doesn't work from time to time or noteout doesn't do anything and there's no *try* and *catch* mechanism to find out why.

14 Pattern control on Raspberry

To use Skini for Raspberry control of the "Pré" project (by Jean-Luc Hervé) with OSC, there are 3 parameters to modify in the piece configuration:

"Use Raspberries", checked;
"Play Buffer Message" with 'test';
"Raspberry OSC Port" set to 4000;

The *Use Raspberries* parameter is used to globally disable playback on the Raspberry and play patterns on the DAW. *playBufferMessage* is the OSC message (without /) that the Raspberry understands to play a sound file. *raspOSCPort* is the UDP port used by OSC for Raspberries.

We extend the pattern descriptor with three pieces of information:

- The IP address of the Raspberry that is to play the pattern
- The sound number in the Rapsberry corresponding to the pattern ("Buffer num" field).
- Pattern sound level (0 to 128)

If the "buffer num" field is not filled in, the pattern is considered to be played by the DAW.

This makes it possible to play patterns with the DAW and the Raspberries in the same room. You can also test a pattern on the DAW before playing it on a Raspberry.

NB for developers: The *queue* stores the Raspberry IP address in index 10, the buffer number in index 11 and the level in index 12. In the pattern *configuration*, the Raspberry IP address is at index 11, the buffer at index 12 and the level at index 13.

15 Appendices

15.1 Skini with Ableton Live

We have often used Ableton Live as a DAW for our developments. Any DAW that can associate a MIDI command with a clip without synchronization constraints will do. Ableton Live offers the possibility of associating MIDI commands with many parameters, including tempo, MAX/MSP and recording controls. Its use with Skini is therefore very rich and easy to implement with the Skini configurator.

For our development work, the patterns were designed in Ableton, mostly in MIDI format. Ableton lets you convert MIDI clips into sounds if required, but you lose MIDI processing such as transpositions, mode conversions and so on.

For our pieces, we don't use global quantization in Ableton. Clip synchronization is handled by Skini.

As described in this documentation, MIDI port configuration is straightforward, although attention must be paid to the IN port used for Skini commands to Ableton, which must authorize Remote Controls ("Téléc."), and the OUT port used by Ableton to send MIDI messages to Skini (Processing gateway), which must also authorize Remote Controls. Ports are configured between lines 214 to 256 of Processing's "Skini sequencer" and Ableton's Preferences/MIDI.

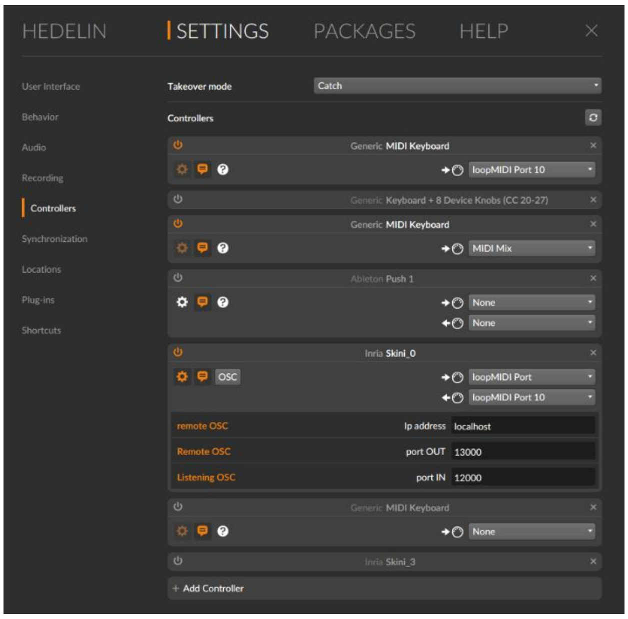
Ableton's MIDI synchronization must be activated on the OUT port corresponding to the Processing gateway's IN port for Ableton's tempo to control Skini.

In the Windows version with LoopMIDI, port 12 is used by Ableton to inform Skini when a clip has been launched (port 13 is used to pass MIDI controls from the video in Reaper). These ports are "hard-wired" in the *OSCMidi* tab of the Processing gateway.

15.2 Skini with Bitwig Studio

With Bitwig studio, it's also possible to communicate in OSC mode, thus bypassing the Processing gateway. A Bitwig controller has been developed for this purpose (./bitwig/Skini_0/Skini_0.control.js). It receives Skini commands in OSC form and, in order not to change the pattern declaration mode, Bitwig Studio reroutes these OSC commands to a virtual MIDI interface (on Windows LoopMidi, for example), which sends the received commands back in MIDI.

The figure on the right shows an example of Bitwig Studio configuration.



Takeover mode: Catch

Controllers

Generic MIDI Keyboard — loopMIDI Port 10

Generic Keyboard + 8 Device Knobs (CC 20-27)

Generic MIDI Keyboard — MIDI Mix

Ableton Push 1 — None / None

Inria Skini_0 — OSC — loopMIDI Port / loopMIDI Port 10

remote OSC — Ip address: localhost

Remote OSC — port OUT: 13000

Listening OSC — port IN: 12000

Generic MIDI Keyboard — None

Inria Skini_3

15.3 Orchestration examples

trouveLaPercuNode.xml : is an example of a game where the audience must find the correspondence between patterns and a sound ambiance. The corresponding Bitwig studio session is worldMusicGame. The Bitwig studio example uses Native Instruments VSTs.

opus5.xml: is an example that essentially uses reservoirs with MIDI transposition commands. The Ableton example uses Native Instruments VSTs. The Live set is opus5.

15.4 File system organization

From the directory where Skini is installed we have the directories:

client : The various skini clients are located in subdirectories.

docs: thesis and Skini project follow-up

images: in subdirectories for each piece, we have here the pattern partitions in the form of jpg files.

pieces: In this directory we have the piece configuration files and the pattern definition csv files.

Processing: contains Processing programs. *SequencerSkini* is the most commonly used.

sequencesSkini: contains patterns saved by the distributed sequencer.

server: contains Skini's Hop.js and HipHop.js files and general configuration.

sounds: patterns' mp3 sound files are organized in sub-directories.

blockly_hop: contains blockly programs and orchestrations in xml format

bitwig: contains extensions for Bitwig Studio

15.5 Score recording in Finale

You can't extract MIDI directly from Ableton if you fear losing the transposition MIDI commands sent directly by Processing. If there are no pattern modifications in Live, you can directly complete the MIDI tracks in Live and export them to Finale. If you wish to use pattern modifications, please follow the procedures below.

The prerequisite in this process is that no tempo modifications have been made in the Skini session. These changes must be made in the final score. Tempo changes are not important, as this is not a sound recording but a written transcription.

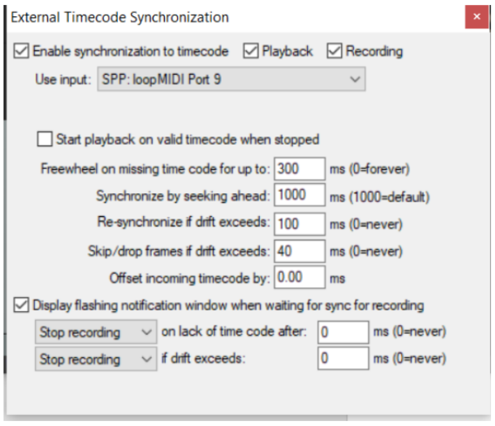
Remember to include an option in the PLCs to deactivate tempo changes.

Beware also of pattern duration. Here, you need to activate Ableton Live's global quantization to avoid MIDI micro-shifts that are impossible to reprocess. Setting global quantization is the same as setting pattern length in Skini.

With these precautions, transcription from Skini to Finale takes just a few minutes.

15.5.1 From an Ableton recording

1) Save the original Ableton Live file
2) Copy it to another file to prepare for MIDI conversion
3) Replace each track's VST with a MIDI OUT channel.
   Simply delete the VST to authorize MIDI output on a Track.
   Pay attention to the output port used for MIDI sync.
4) Remove tempo variations (if any)
5) Set up MIDI channels in Reaper (port 9 is generally used for MIDI sync).
6) Slave Reaper with MIDI sync



External Timecode Synchronization

☑ Enable synchronization to timecode ☑ Playback ☑ Recording

Use input: SPP: loopMIDI Port 9

☐ Start playback on valid timecode when stopped

Freewheel on missing time code for up to: 300 ms (0=forever)

Synchronize by seeking ahead: 1000 ms (1000=default)

Re-synchronize if drift exceeds: 100 ms (0=never)

Skip/drop frames if drift exceeds: 40 ms (0=never)

Offset incoming timecode by: 0.00 ms

☑ Display flashing notification window when waiting for sync for recording

Stop recording on lack of time code after: 0 ms (0=never)

Stop recording if drift exceeds: 0 ms (0=never)

7) Set Ableton to MIDI Sync transmission (check Ableton and Reaper sync ports)
8) Record in Reaper, setting the same tempo as Ableton Live. Make sure you have Ableton Live with global quantization.
9) In Reaper, format MIDI tracks, remove the recording start gap, quantize MIDI...
10) Export each Reaper MIDI track independently
11) Load MIDI tracks recorded by Reaper into Finale
12) Formatting in Finale.

15.5.2 Directly from Skini

Parameters and constraints on tempo variations are the same as above. You can directly record the progress of a Skini session in Reaper without recording it in Live.

Care must be taken to ensure global quantization in Ableton Live to avoid small MIDI shifts or drifts from Skini. It should be possible to adjust the MIDI sync delay in Live to be perfectly in phase between Live and Reaper.

15.6 Switch from Finale to Microsoft Word or PowerPoint

- Export score in PDF format to Finale
- Load the PDF into INKSCAPE via import, using the options that integrate fonts (not the default option).
- In INKSCPAPE, save in simple SVG format.
- Drag SVG to Microsoft

15.7 Receive MIDI clip launch info from Ableton

- You need IP addresses in IPconfig, not localhost.

- Set an unused MIDI out channel to "Telec" (remote). This is equivalent to considering that there is a control surface on the port (in opus1 and 2 it's loopMIDI Port out 12).
- The correspondence must be made in Processing (sequencerSkini.pde) in the "myBusIn" table and in the MIDI command source tests (OSCMidi). Processing translates commands coming from the MIDI port into OSC "StartClip" commands.
- Ableton sends MIDI commands to launch clips on this channel.

Remark: In "track" mode on a MIDI out channel, Ableton transmits the MIDI notes it processes, but not the CCs. If you want to retrieve CCs, they'd have to come from clips.

MIDI commands sent by Ableton follow an odd protocol that is processed by midimix.js, the program that processes OSC data from Processing (the name isn't very good, it goes back to the Golem).

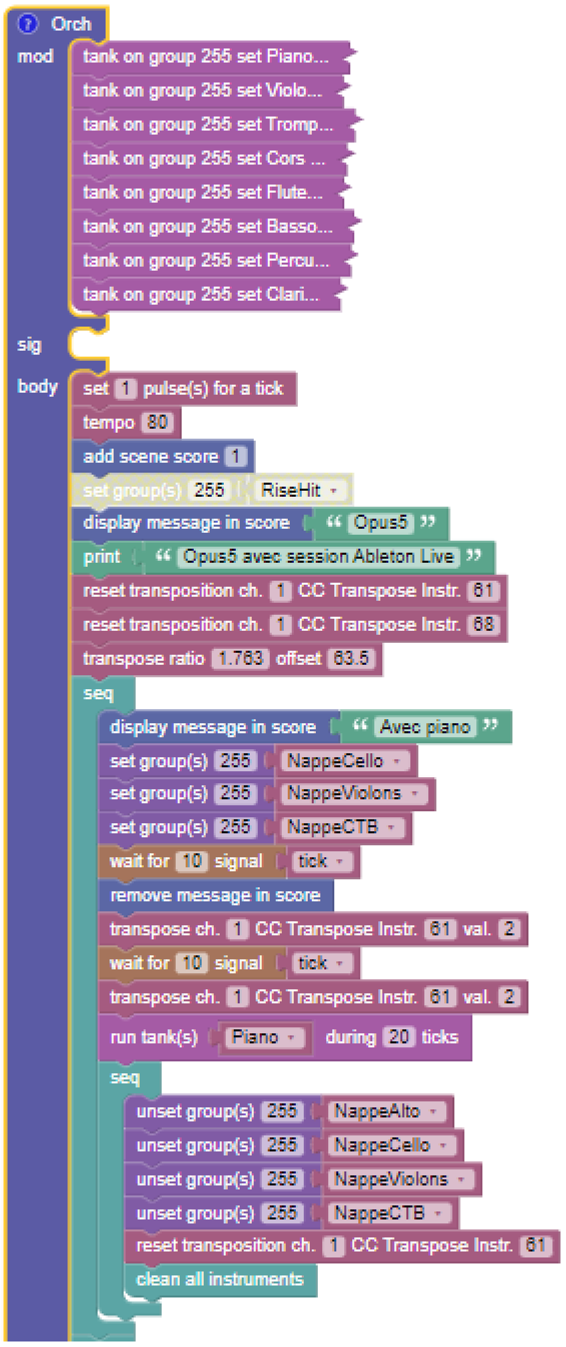
Midimix is created in golem.js. It passes information to websocketServer, and it is websocketServer that informs the automaton. The only way to access the automaton is via websocketServer.

15.8 LIVE PATTERN RECORDING IN THE DISTRIBUTED SEQUENCER

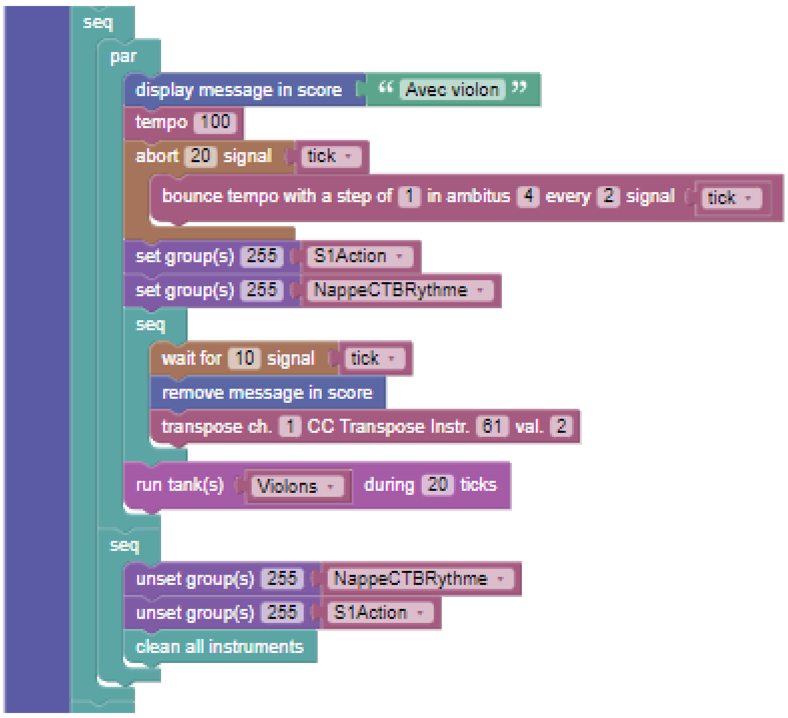
(to do, give the structure of the programs.).

16 A SAMPLE PIECE: OPUS5

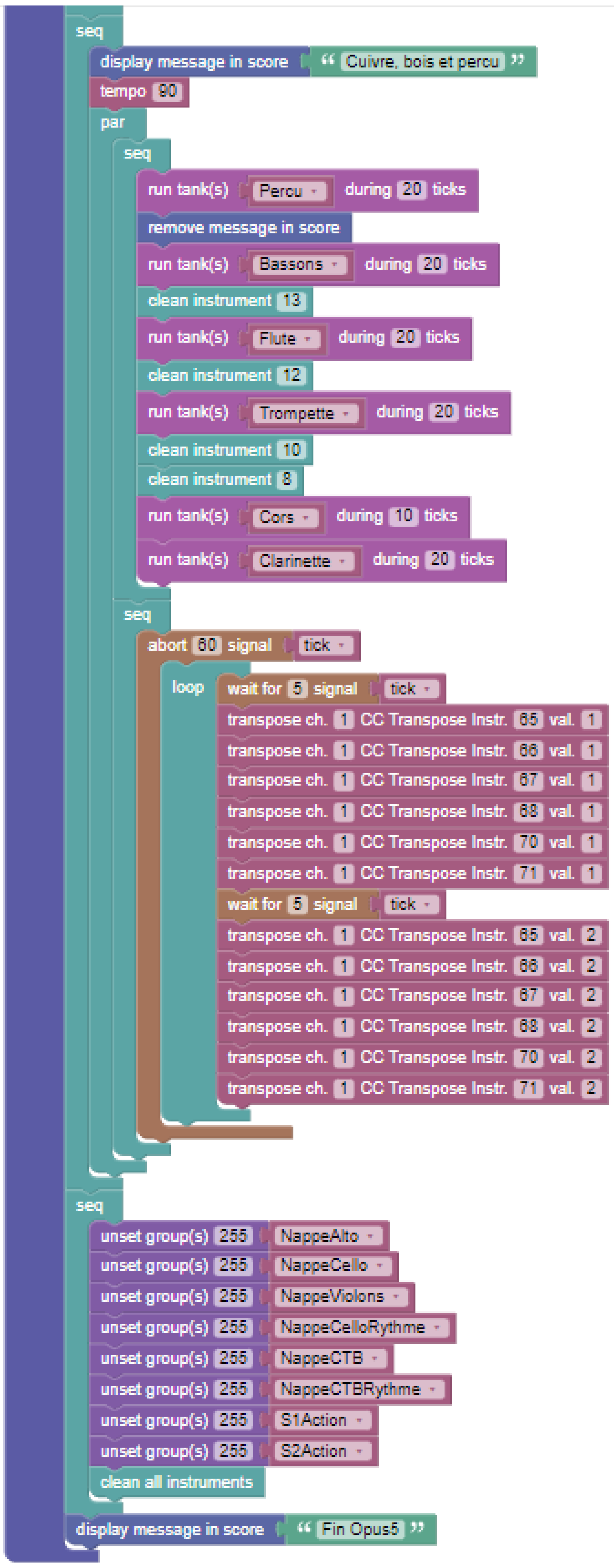
This piece uses most of Skini's basic principles.



```
Orch
mod   tank on group 255 set Piano...
      tank on group 255 set Violo...
      tank on group 255 set Tromp...
      tank on group 255 set Cors ...
      tank on group 255 set Flute...
      tank on group 255 set Basso...
      tank on group 255 set Percu...
      tank on group 255 set Clari...
sig
body  set 1 pulse(s) for a tick
      tempo 80
      add scene score 1
      set group(s) 255 RiseHit
      display message in score " Opus5 "
      print " Opus5 avec session Ableton Live "
      reset transposition ch. 1 CC Transpose Instr. 61
      reset transposition ch. 1 CC Transpose Instr. 68
      transpose ratio 1.763 offset 63.5
      seq
        display message in score " Avec piano "
        set group(s) 255 NappeCello
        set group(s) 255 NappeViolons
        set group(s) 255 NappeCTB
        wait for 10 signal tick
        remove message in score
        transpose ch. 1 CC Transpose Instr. 61 val. 2
        wait for 10 signal tick
        transpose ch. 1 CC Transpose Instr. 61 val. 2
        run tank(s) Piano during 20 ticks
        seq
          unset group(s) 255 NappeAlto
          unset group(s) 255 NappeCello
          unset group(s) 255 NappeViolons
          unset group(s) 255 NappeCTB
          reset transposition ch. 1 CC Transpose Instr. 61
          clean all instruments
```

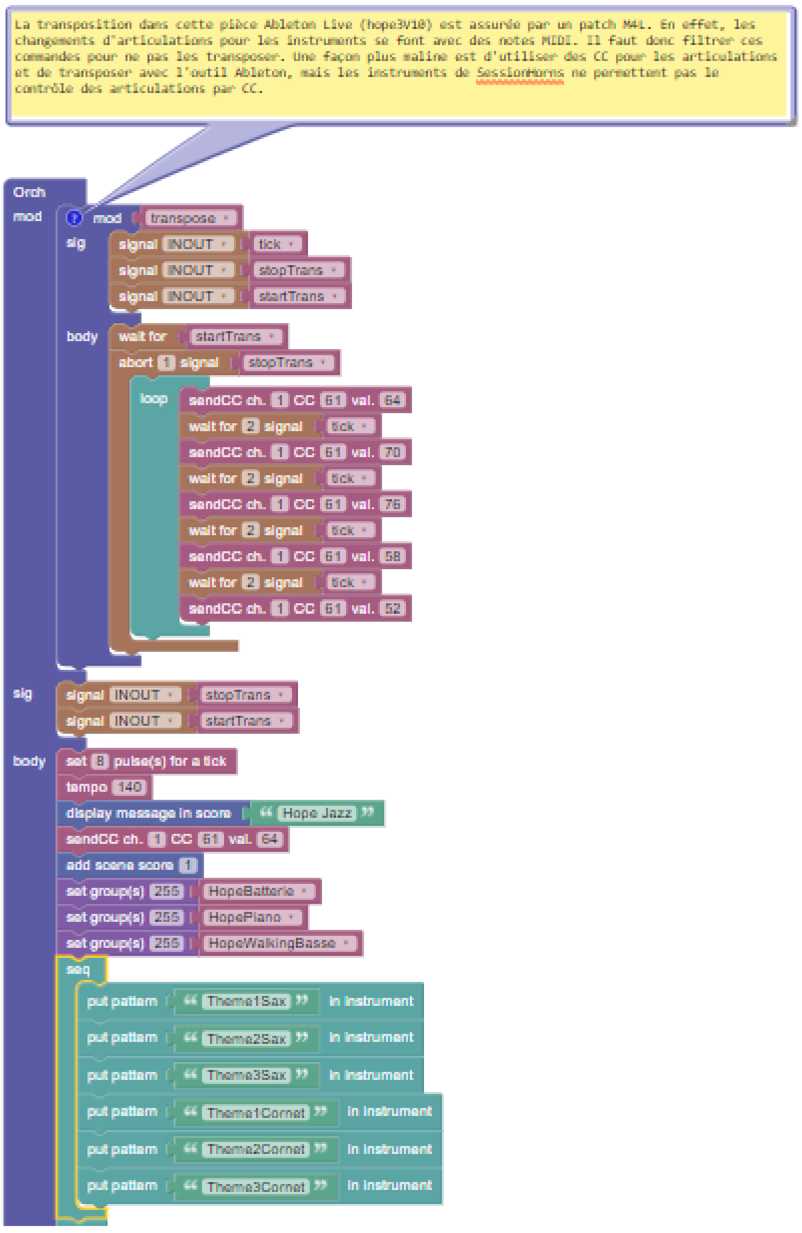


```
seq
  par
    display message in score “ Avec violon ”
    tempo 100
    abort 20 signal tick
      bounce tempo with a step of 1 in ambitus 4 every 2 signal tick
    set group(s) 255 S1Action
    set group(s) 255 NappeCTBRythme
    seq
      wait for 10 signal tick
      remove message in score
      transpose ch. 1 CC Transpose Instr. 81 val. 2
    run tank(s) Violons during 20 ticks
  seq
    unset group(s) 255 NappeCTBRythme
    unset group(s) 255 S1Action
    clean all instruments
```



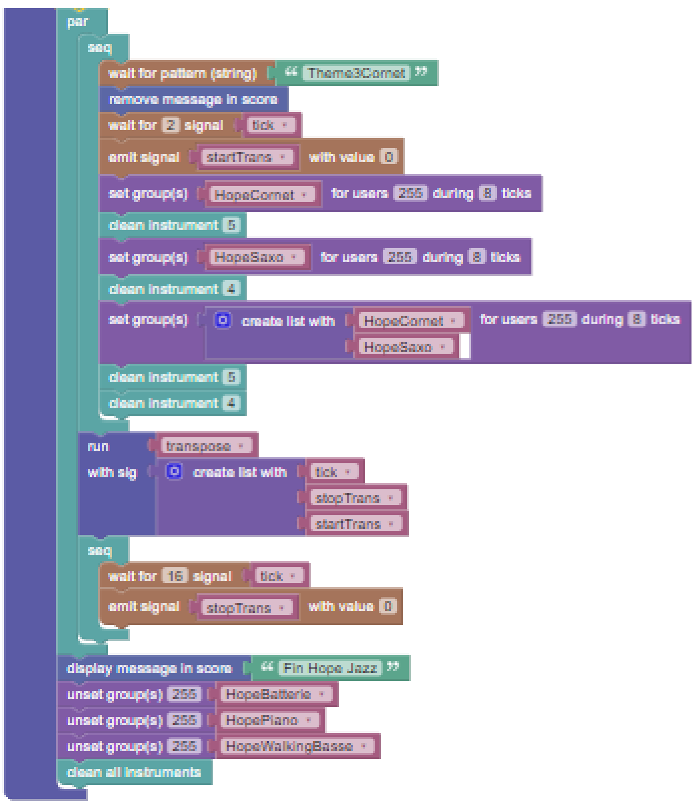
```
seq
  display message in score “ Cuivre, bois et percu ”
  tempo 90
  par
    seq
      run tank(s) Percu during 20 ticks
      remove message in score
      run tank(s) Bassons during 20 ticks
      clean instrument 13
      run tank(s) Flute during 20 ticks
      clean instrument 12
      run tank(s) Trompette during 20 ticks
      clean instrument 10
      clean instrument 8
      run tank(s) Cors during 10 ticks
      run tank(s) Clarinette during 20 ticks
    seq
      abort 60 signal tick
        loop
          wait for 5 signal tick
          transpose ch. 1 CC Transpose Instr. 65 val. 1
          transpose ch. 1 CC Transpose Instr. 66 val. 1
          transpose ch. 1 CC Transpose Instr. 67 val. 1
          transpose ch. 1 CC Transpose Instr. 68 val. 1
          transpose ch. 1 CC Transpose Instr. 70 val. 1
          transpose ch. 1 CC Transpose Instr. 71 val. 1
          wait for 5 signal tick
          transpose ch. 1 CC Transpose Instr. 65 val. 2
          transpose ch. 1 CC Transpose Instr. 66 val. 2
          transpose ch. 1 CC Transpose Instr. 67 val. 2
          transpose ch. 1 CC Transpose Instr. 68 val. 2
          transpose ch. 1 CC Transpose Instr. 70 val. 2
          transpose ch. 1 CC Transpose Instr. 71 val. 2
seq
  unset group(s) 255 NappeAlto
  unset group(s) 255 NappeCello
  unset group(s) 255 NappeViolons
  unset group(s) 255 NappeCelloRythme
  unset group(s) 255 NappeCTB
  unset group(s) 255 NappeCTBRythme
  unset group(s) 255 S1Action
  unset group(s) 255 S2Action
  clean all instruments
display message in score “ Fin Opus5 ”
```

17 A PIECE WITH TRANSPOSITION CONTROL DIRECTLY WITH CC MIDI



```
La transposition dans cette pièce Ableton Live (hope3V10) est assurée par un patch M4L. En effet, les
changements d'articulations pour les instruments se font avec des notes MIDI. Il faut donc filtrer ces
commandes pour ne pas les transposer. Une façon plus maline est d'utiliser des CC pour les articulations
et de transposer avec l'outil Ableton, mais les instruments de SessionHorns ne permettent pas le
contrôle des articulations par CC.
```

```
Orch
mod   ? mod  transpose
        sig   signal INOUT  tick
              signal INOUT  stopTrans
              signal INOUT  startTrans
        body  wait for  startTrans
              abort 1 signal  stopTrans
                 loop  sendCC ch. 1 CC 61 val. 64
                       wait for 2 signal  tick
                       sendCC ch. 1 CC 61 val. 70
                       wait for 2 signal  tick
                       sendCC ch. 1 CC 61 val. 76
                       wait for 2 signal  tick
                       sendCC ch. 1 CC 61 val. 58
                       wait for 2 signal  tick
                       sendCC ch. 1 CC 61 val. 52
sig   signal INOUT  stopTrans
      signal INOUT  startTrans
body  set 8 pulse(s) for a tick
      tempo 140
      display message in score  " Hope Jazz "
      sendCC ch. 1 CC 61 val. 64
      add scene score 1
      set group(s) 255  HopeBatterie
      set group(s) 255  HopePiano
      set group(s) 255  HopeWalkingBasse
      seq
         put pattern  " Theme1Sax "  in instrument
         put pattern  " Theme2Sax "  in instrument
         put pattern  " Theme3Sax "  in instrument
         put pattern  " Theme1Cornet "  in instrument
         put pattern  " Theme2Cornet "  in instrument
         put pattern  " Theme3Cornet "  in instrument
```



```
par
  seq
    wait for pattern (string) " Theme3Cornet "
    remove message in score
    wait for 2 signal tick
    emit signal startTrans with value 0
    set group(s) HopeCornet for users 255 during 8 ticks
    clean instrument 5
    set group(s) HopeSaxo for users 255 during 8 ticks
    clean instrument 4
    set group(s) create list with HopeCornet for users 255 during 8 ticks
                                  HopeSaxo
    clean instrument 5
    clean instrument 4
  run transpose
  with sig create list with tick
                            stopTrans
                            startTrans
  seq
    wait for 16 signal tick
    emit signal stopTrans with value 0
display message in score " Fin Hope Jazz "
unset group(s) 255 HopeBatterie
unset group(s) 255 HopePiano
unset group(s) 255 HopeWalkingBasse
clean all instruments
```