MUSIC NOTATION USING REACTIVE SYNCHRONOUS PROGRAMMING

Bertrand Petit
INRIA
Sophia-Antipolis, France
Bertrand.petit@inria.fr

ABSTRACT

This article presents a notation system for music based on *patterns* (or *clips*) as they have been popularized for more than twenty years with the Digital Audio Workstations on the market like *Ableton Live*, *Bitwig Studio* or *FL Studio*. This system named *Skini* uses the *HipHop.js* programming language to describe music pieces. This language, belonging to the family of synchronous reactive languages, was initially designed for the orchestration of Web services. Skini, by combining HipHop.js and queuing mechanisms, was developed for interactive and generative music performances. It has also proven to be an efficient tool for notating musical pieces outside of these interactive and generative contexts because of its ability to describe the structure of a piece of music in a form close to its expression in everyday language. Moreover, Skini, while using certain concepts specific to electronic music, can be used for the creation and performance of instrumental and orchestral music.

1. INTRODUCTION

According to a common definition, music notation consists of transcribing a musical work onto a medium to interpret, preserve, protect, and disseminate it. The systems of notation are dependent on the media available to support this notation. For example, Hurrian songs have been found on clay tablets from around 1400 BC. The paper and the staff system from the neumes of the Middle Ages was for a long time the only support in the occidental societies until the information sciences appeared and proposed other ways to notate music. With the help of computers, it is not only a matter of facilitating the manipulation of the staff system, but also of introducing other ways of representing music. Of the many ways to score music, we will focus here on a system that uses a computer language from the family of *synchronous reactive languages* [1]. These languages, imagined in the 80's, are not initially intended for music but for critical systems (airplane, train, nuclear power plant...). However, we will see that one of these languages, HipHop.js [2], is well adapted for representing music in the form of *patterns* or *musical elements* similar to what is called *clips* in current Digital Audio Workstations (DAW). The HipHop.js language is implemented in a platform named Skini which was initially designed for collaborative music in interaction with an audience, but which can also be used to produce generative music or the musical notation of clip-based pieces.

2. RELATED WORK

There are several families of tools allowing the notation of musical pieces using programming languages or computer systems.

In the family of programming tools for sound creation, *Open Music* from IRCAM [3] is an example of a solution allowing to describe musical processes using graphical programming that generates LISP code. *Csound* [4] is another popular tool for producing music from a computer language, this time from the C language.

Another family of tools is constituted by the *Live Coding* solutions. These solutions deal with algorithmic music improvisation, which *Open Music* or *Csound* were not designed for. Popular languages in this category are for example *ChucK* [5], *SuperCollider* [6] or *Fluxus*.

Graphical languages like *MAX/MSP* and *PureData* [7] are not intended for music notation per se but for the combination of musical processes and signal processing tools. However, many pieces are designed from these tools without any other notation mode than the *Patches* written in these languages.

Some Digital Audio Workstation (DAW) like *Ableton Live* [8] or *Bitwig Studio* [9] are also de facto notation tools. A piece of music is expressed as a matrix of clips with a whole set of properties. Each of the clips can be expressed in the form of MIDI *piano-roll*, which is a notation system commonly used by DAWs.

We will not discuss here the case of *score editors* such as *Finale*, *Sibelius*, *MuseScore* or *Dorico* which are tools for entering and formatting scores as they have been written for several centuries. These tools, although computerized, do not fundamentally call for programming skills on the part of the composer.

Each of these families of tools constitutes a solution to particular music production problems. *Open Music* is intended for composers with a process approach to music

creation. This tool is aimed at musicians with good algorithmic skills. The Live Coding tools were originally designed for performance and improvisation. Tools such as *MAX/MSP*, *PureData* or *Csound* are intended for general sound production, not specifically for an improvisation context. DAWs like *Ableton Live* or *Bitwig studio* are initially designed for live performance while offering a relatively simple way of structuring music compared *to MAX-MSP* for example, whose learning curve is long.

Skini is somewhere between clip-based DAWs and computer-based programming in the sense of *Open Music* or *Csound*. Skini is not a *Live Coding* tool because it is not designed for improvisation, even though it is a tool initially designed for live and interactive performances. We only discuss here the use of Skini in the well-defined context of music notation and not in the context of generative music described in the article [10] or for collaborative music described in the article [11]. We will see that Skini is mainly interested in a method of conception of complex musical structures. “Structure design” is not the basic problem of Computer Music solutions which will deal with other topics such as signal processing, live production, score editing or improvisation. We will see that, in this sense, Skini is a complementary notation tool to most of those already available.

3. SKINI NOTATION BASIC CONCEPTS

We will review the main concepts behind the music notation used by Skini. These are patterns, instruments, interaction, and scenarios.

3.1 Patterns

Musical patterns, or musical elements, are the raw material of Skini. They are short musical phrases designed by the composer and arranged in such a way as to constitute the musical piece. They are close to clips in the vocabulary of DAWs, except that clips cannot be graphic extracts from scores, which is possible with Skini when the music is played by musicians and not by a DAW.

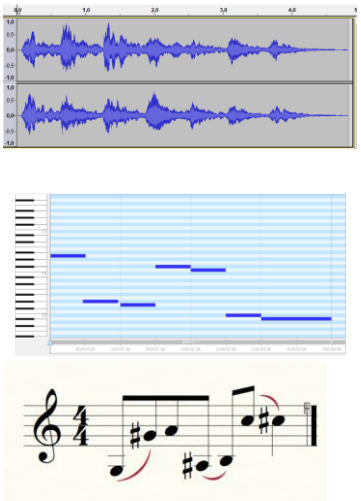

Figure 1: Three ways to describe a pattern.

As we can see in **Figure *1***, a pattern can take the form of a sound file, a MIDI sequence, or a few bars of a score. Skini does not impose any constraints on the design of the patterns, nor on their duration. The forms that the composer will give to the patterns will depend on the style of music and the type of performance imagined. A composer who wants to control synthesizers will naturally use MIDI patterns. A composer writing for instrumentalists will more naturally use score elements.

3.2 Instruments

Patterns are played by instruments. We will see that in his notation system, Skini can request the execution of several patterns by one instrument at a time. As we consider that an instrument can only play one pattern at a time (as an instrumentalist does), if several patterns are requested for the same instrument at the same time, they will be placed one after the other in a *queue* to be played one after the other. The principle of queues for patterns was initially implemented to guarantee a good coherence of the musical pieces in interaction with the audience, and to allow the use of Skini with instrumentalists. In the simpler context of notation of a pattern-based musical piece without calling for interaction, queues allow to simplify the implementation of musical sequences in parallel. For example, the composer can decide to load a complete musical sequence at an instant of the score without having to worry about this sequence while the rest of the score is processing other sequences on other instruments. Schematically, queues allow to implement musical sequences without worrying about their duration.

3.3 Interaction

Skini natively considers the possibility of interacting with music in the form of events produced by sensors, web interfaces proposed to the audience, messages received by a video game or random processes. We will not discuss in detail the complex question of interaction here. But this is an important aspect of music notation with Skini. Indeed, few tools have a way to describe music and interactions with events outside the music. *Antescofo* [12] and *In-Score*[13] are examples of these tools. For more information, we refer you to another article on interaction [11].

3.4 Scenario or score

As we have seen, Skini proposes a way to describe a musical piece based on the concepts of pattern, instrument, and interaction. Skini is interested in the way the composer will organize these concepts. It is a tool to define structures, *orchestrations* but in a computer science sense. The term *score* would be well adapted for this description, but it is most often associated with the notation of pitches in time, which is not the purpose of Skini. The term *orchestration* in a musical context would be closer, but it does not include the notion of pattern. We retain the rather vague term of *musical scenario* associating patterns with

interaction, and queues by instruments. Skini scenarios are written using the computer language: HipHop.js[1].

4. PRINCIPLE OF SYNCHRONOUS REACTIVE PROGRAMMING WITH HIPHOP.JS

Synchronous reactive programming languages are languages meant for programming *reactive systems*. Computer systems are often classified into three categories. *Transformational systems* that take inputs, process them, provide their outputs, and terminate their execution. *Interactive systems* that continuously interact with their environment, at their own speed. A typical example is a web application. *Reactive systems* that continuously interact with their environment, at a speed imposed by the environment. A typical example is the control system of a vehicle. Reactive systems react to environmental stimuli. HipHop.js belongs to this family of *synchronous reactive languages*, and is very close to the *Esterel* language [14] initially conceived in the 80s. As with *Esterel*, programming with HipHop.js consists in approaching an algorithm by thinking in terms of *events* and *reactions*. Events are materialized by means of *signals*. A HipHop.js program, once compiled, is like a black box to which one submits signals linked to events, and which produces other signals when it is solicited. This solicitation provokes what is called an *immediate* ,and thus *synchronous*, reaction since it does not introduce any delay (at least in a theoretical way). This model is represented by the **Figure *2***. The signals A, X, W, Z are purely indicative. There is no limit or constraint on the number of signals and their types.

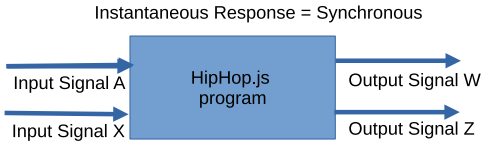

Figure 2: Reactive Synchronous principle.

The syntax of HipHop.js uses a set of about twenty statements whose operation is quite intuitive. For example, it is possible to wait for a signal (`await` statement) to move to the next statement. It is possible to emit a signal (`emit` statement). One of the important characteristics of HipHop.js is to integrate natively the parallelism. To allow two blocks of statements to run at the same time. Few languages natively integrate this important feature, which allows to act on musical sequences by easily modifying their juxtaposition for example. Skini offers the composer two different ways of programming. For composers who are familiar with computer tools and textual programming, it is possible to program in the HipHop.js language which also supports JavaScript. For composers less familiar with textual programming, it is possible to use a graphic programming tool. This tool offers the same primitives as HipHop.js in textual form but is less well adapted to the integration of JavaScript code to realize, for example, complex logical operations within the notation of the piece.

5. SKINI ARCHITECTURE

Skini provides a notation system within a music production environment. **Figure *3*** is a logical view of the Skini platform used to implement HipHop.js scenario programming in its music production environment. The architecture is composed of 3 layers.

The central layer *Control and decision* layer is the one that contains the HipHop.js program and the instrument queues. These queues will receive pattern commands from the HipHop.js program. It can also receive pattern commands from a stochastic engine or an audience. This layer is implemented by means of the multi-tier Web platform *Hop.js* [15], developed by INRIA, or *Node.js* [16]. These platforms use JavaScript. HipHop.js is thus a Domain Specific Language (DSL) supported by JavaScript.

The *event* layer deals with events external to the course of the music piece. It can be a clock or various events produced by sensors, video games, etc. In a simple implementation dealing with notation only, this level can be reduced to the use of a clock.

The *Music Production* layer is necessary to experiment or simulate a musical result and especially when Skini

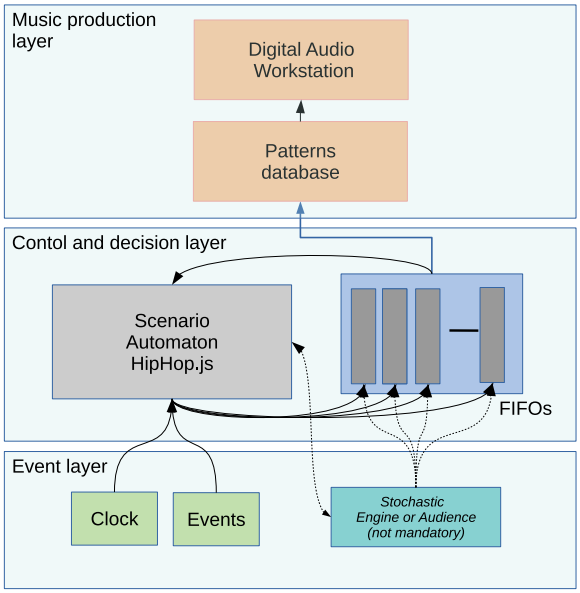

Figure 3: Skini architecture.

communicates with a DAW that implements a set of patterns. For live performances, instead of using a DAW, Skini can call upon musicians equipped with an interface

[1] Without reference to a particular musical genre. The authors of this language have chosen this name as a pun towards a platform developed by INRIA called Hop.js.

that can display patterns in the form of scores (PC or Tablets).

For a composition and simulation work, in the case of a use of MIDI patterns with a DAW, the DAW will be able to record the piece in MIDI format (cf. Figure 4) to verify that Skini's notation corresponds to the expected musical result.

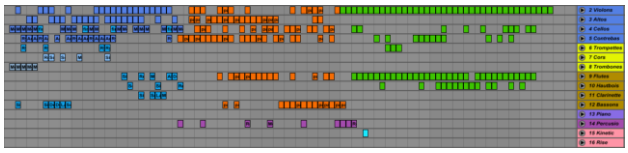

Figure 4: A Skini score view in Ableton Live.

It is then possible to export the results of the composition to an editing software (*Finale* or *Sibelius* for example) to produce a "classical" score (cf. Figure 5). Skini is therefore not limited to the production of electronic music, or live performances but can also be a tool for composing orchestral pieces.

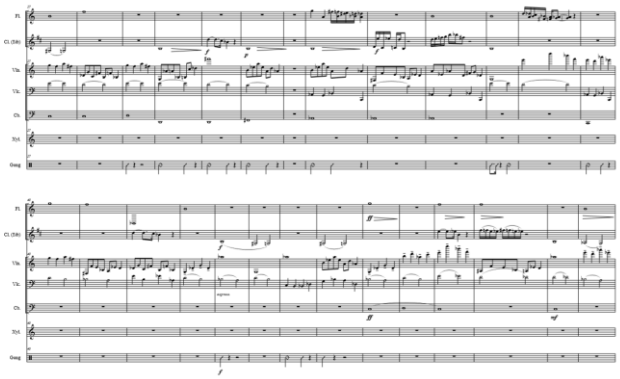

Figure 5: A Skini score view in Finale.

6. EXAMPLES OF SCORE IMPLEMENTATIONS

Without worrying about the details of the implementation of a complete score by means of HipHop.js programming or the graphical programming tool allowing to abstract from a part of the HipHop.js syntax, we will see some simple examples. For more details on HipHop.js programming applied to music, we recommend the article *"Interactive Music and Synchronous Programming"*[17].

6.1 Programming loops

Here is an example of programming a loop on a pattern in HipHop.js.

```
every count (4, tick.now) {
   hop{ putPatternInQueue("Morricone");}
}
```

Using the graphical programming tool, the equivalent will be:

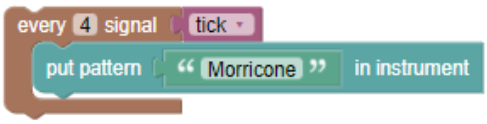

Figure 6: A loop with every.

The statement `every` will execute the body between braces every time 4 ticks. The *tick* signal emitted by a clock has been set according to the duration of the patterns. Here the pattern *Morricone* has been described in a configuration file with a duration of 4 ticks. The `hop` statement is a facility of HipHop.js to pass JavaScript commands. The graphical programming is perfectly equivalent.

Here is another example of loop programming involving two patterns on the same instrument. Each pattern lasts 4 ticks.

```
loop{
   hop{putPatternInQueue("Morricone");}
   hop{putPatternInQueue("Rosenman");}
   await count (8, tick.now);
}
```

Using the graphical programming tool the equivalent will be:

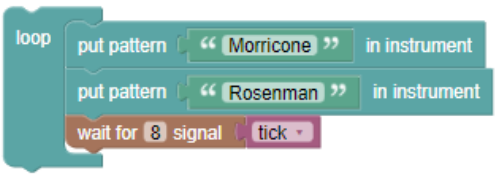

Figure 7: A loop using the loop statement.

Instead of an `every` statement we use here a `loop` statement, which loops indefinitely over the body between braces. The two `putPatternIn-Queue` commands or "put pattern" blocks are executed "at the same time". This means that at each loop, 2 patterns are sent to the instrument's queue to be played one after the other. The `await` statement will stop the loop until 8 ticks have been received. This is a way to avoid overloading the queue immediately.

6.2 A more complex scenario

These two simple examples give an idea of how to program with Skini, but they are not enough to demonstrate the relevance of this programming compared to other solutions such as those offered by a clip-based DAW. Let's look at a slightly more complex case.

We want to loop two patterns *Mancini* and *Silvestri*, let's name this loop *loop1*. At each occurrence of two *loops1* we want to execute another sequence of patterns which consists of playing the pattern *Rota* twice on one instrument and a pattern *Geoffroy* on another instrument. At the same time as these two loops are running, we want to play another sequence a trumpet solo which consists of a rather long sequence of trumpet patterns. We could express this scenario graphically in a sequencer. Space does not allow

us to do so in this article, but it is easy enough to imagine that this is a simple but tedious job. In a clip-based DAW, one way to implement this scenario would be to create a control track that sends MIDI commands to tracks containing the patterns (clips) of the instruments. Using a MIDI virtual cable, the commands from the control track could be sent back to the DAW to trigger the patterns. We could create the loops using a *follow action* of each clip and create a MIDI command to stop the track that would be driven by the control track. To implement the scenario, we just need to create control clips in the control track that will issue the start and stop commands for the loops. Technically, this method works. However, it is not very easy to read, and it is difficult to scale up as soon as the scenario becomes more complex.

Here is how this scenario is expressed with Skini:

```
fork{
   every count (8, tick.now){
      hop{putPatternInQueue("Mancini");}
      hop{putPatternInQueue("Silvestri");}
      emit boucle1();
   }
}par{
   every count (2, boucle1.now){
      hop{putPatternInQueue("Rota");}
      hop{putPatternInQueue("Rota");}
      hop{putPatternInQueue("Geoffroy");}
   }
}par{
   run ${soloTrompette}(...);
}
```

Or graphically:

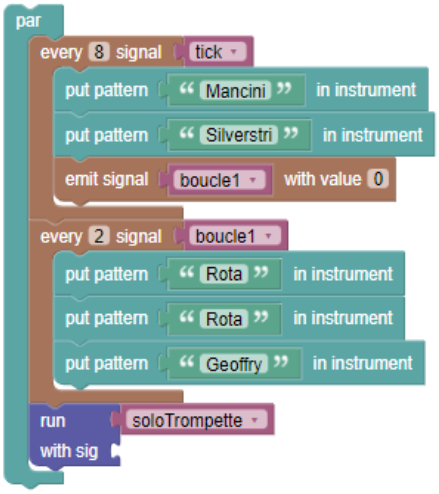

Figure 8: A more complex scenario.

In HipHop.js programming, the `fork` and `par` statements are used to parallel blocks of statements between braces. In graphical programming the `par` block parallels the `every` and `run` blocks. The `run` block calls a module that contains the trumpet solo. The `soloTrompette` module is written in HipHop.js as follows:

```
var soloTrompette = hiphop module () {
   fork{
      hop{ableton.putPatternInQueue("Altenburg");}
      hop{ableton.putPatternInQueue("André");}
      hop{ableton.putPatternInQueue("Hardenberger");}
      hop{ableton.putPatternInQueue("Thibaud");}
      hop{ableton.putPatternInQueue("Gambati");}
      hop{ableton.putPatternInQueue("Foveau");}
      hop{ableton.putPatternInQueue("Friedrich");}
      hop{ableton.putPatternInQueue("Sauter");}
      hop{ableton.putPatternInQueue("Arban");}
   }par{
      await count(44, tick.now);
      hop{ console.log("END OF TROMPET SOLO"); }
   }
}
```

with as graphic equivalent in the module Figure 9. In this module, there are two branches in parallel. One loads the

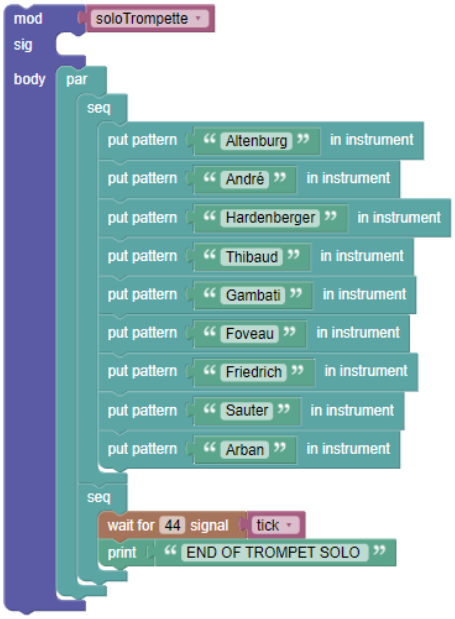

Figure 9: The trumpet solo.

queue of the trumpet instrument with a sequence of patterns. The other branch counts down the total duration of all the queued patterns. Here it gives 44 ticks. After 44 ticks a message is displayed on the console using a JavaScript statement to inform us that the solo is finished.

The first `every` of the main scenario of Figure 8, in which `run` is called, has the same structure as the previous scenario example **Figure *6***. The difference is that a `boucle1` signal is sent every 8 ticks. The second `every` uses the `boucle1` signal from the first `every` to count down. The trumpet solo is executed independently of the two `every`. Although simple, this dependency between the `every` statements is difficult or even impossible to implement with a clip-based DAW. This would require that a clip of an instrument can issue a control command, which is not standard with current DAWs. But beyond feasibility, the major difference between Skini and a DAW is readability and maintainability. With Skini it is very easy to add more loops. It is also easy to define complex logical combinations between signals on the conditions of the `every`

or other signal processing constructs such as `await`, `abort`, `if`, `suspend`, and to structure the piece of music in modules that can be combined and recombined at will.

Regarding the comparison between HipHop.js and general languages (Java, JavaScript, C++...), see chapter 3 of the article [17]. The conclusion of this comparison is close to the one made with a DAW. Writing a script with a general-purpose language quickly becomes very difficult to implement and maintain when the project gets complicated.

6.3 Other writing features

Beyond the description of complex scenarios, Skini includes primitives allowing the introduction of random phenomena in the writing of a piece of music, and to use different OSC or MIDI controls for electronic music. The use of HipHop.js in combination with JavaScript makes it easy to extend the Skini primitives to complex control processes. This is not useful for most composers, but it does mean that it is simple to extend the vocabulary of the notation system.

7. CONCLUSION AND FUTURE WORK

Skini was initially designed for the creation of musical performances in interaction with an audience. The goal of our research was to provide a set of tools for composing, executing, and verifying the aesthetic coherence of interactive music pieces. Examples of music designed using Skini are available at: https://soundcloud.com/user-651713160.

The definition of scenarios that can handle complex combinations of events led us to use a language designed for automata programming. We chose HipHop.js because this language based on the Synchronous Reactive Language *Esterel* [18] seemed to us well adapted to our problem which deals on the one hand with complex automata but also with interaction. Indeed, HipHop.js works on development platforms for the Web using JavaScript. Nowadays, Web technologies are the most common for large-scale interactions. Beyond this dimension of interaction with an Audience, Skini with its ability to manage events, has proved to be a solution to produce music from random processes controlled by events from sensors or video games for example. This association between random process and scenario brought Skini into the world of generative music solutions and more particularly combinatorial generative music. After these first use cases, the use of Skini for score notation without random phenomena or interaction has proved to be interesting. It is indeed complementary to clip-based approaches developed for more than twenty years by manufacturers such as *Ableton*, *Bitwig* or *FL studio* for example. Skini is not a toolbox competing with *Open Music* or *MAX/MSP* because its vocation is to express the structure of a musical work rather than its detail. For Skini, the detail comes from the patterns for which there is no recommendation or constraint imposed by our system. The composer can create each pattern manually, or use a tool like *Open Music*, or use an Artificial Intelligence solution for example.

We think that the creation of music based on patterns in the form of clips has the merit of being widely diffused and that it offers a strong potential for development in combination with tools allowing the management of complex structures as proposed by Skini. DAW editors such as *Ableton* or *Bitwig* are integrating more and more functionalities that go in the direction of scenario management in the sense that we have detailed in this document. However, they have not yet addressed the problem in the form of a notation system as powerful as Skini. This is not a coincidence because the problems raised by the notation of complex scenarios are difficult to solve. *Synchronous reactive languages* are the only ones to have provided a viable solution through substantial research work, but although they have been very successful in the industry, they have not yet been widely used in the music world.

Skini can therefore address a population of DAW users wishing to create musical pieces with complex structures that cannot be done with the generalist tools.

8. REFERENCES

[1] P. A. Laplante and S. J. Ovaska, “Programming Languages for Real-Time Systems,” in Real-Time Systems Design and Analysis, 2011.

[2] G. Berry and M. Serrano, “HipHop.js: (A)Synchronous reactive web programming,” in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2020, pp. 533–545.

[3] J. Bresson, “Reactive Visual Programs in OpenMusic,” [Research Report] IRCAM / ANR-13-JS02-0004-01 Effic., pp. 0–12, 2015.

[4] V. Lazzarini, S. Yi, J. Ffitch, J. Heintz, Ø. Brandtsegg, and I. McCurdy, Csound: A sound and music computing system. 2016.

[5] G. Wang, “The ChucK Audio Programming Language “ A Strongly-timed and On-the-fly Environ / mentality,” 2008.

[6] J. McCartney, “Rethinking the computer music language: SuperCollider,” CMJ, 2002.

[7] M. Puckette, “Max at seventeen,” CMJ, 2002.

[8] Ableton, “Ableton Live 10 - Ableton,” Ableton, 2018. [Online]. Available: https://www.ableton.com/fr/live/. [Accessed: 10-May-2020].

[9] S. Truss, “Bitwig Studio 3,” Electron. Music., vol. 35, no. 11, 2019.

[10] B. Petit, “Generative Music Using Reactive Programming,” Proc. ICMC 2021, pp. 320–324, 2021.

[11] B. Petit and M. Serrano, “Composing and Performing Interactive Music using the HipHop . js language,” in New Interfaces for Musical Expression 2019, 2019, pp. 71–76.

[12] A. Cont, “Antescofo: Anticipatory synchronization and control of interactive parameters in computer music,” in International Computer Music Conference, ICMC 2008, 2008.

[13] D. Fober, S. Letz, Y. Orlarey, and L. France, “Programming Interactive Music Scores with INScore,” SMC, vol. October 20, pp. 2–7, 2013.

[14] G. Berry, “The Foundations of Esterel,” in Proof, Language, and Interaction, MIT Press., 2000.

[15] M. Serrano and V. Prunet, “A glimpse of hopjs,” in ICFP 2016 - Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, 2016.

[16] R. Prediger, R. Winzinger, R. Prediger, and R. Winzinger, “Node.js,” in Node.js, Carl Hanser Verlag GmbH & Co, 2015, pp. I–XV.

[17] B. Petit and M. Serrano, “Skini: Reactive programming for interactive structured music,” Art, Science, and Engineering of Programming, vol. 5, no. 1, 2020.

[18] G. Berry, The {Esterel} v5 Language Primer. Sophia-Antipolis, France, 2000.