# Temps et durée : de la programmation réactive synchrone à la composition musicale

Bertrand Petit

**To cite this version:**

Bertrand Petit. Temps et durée : de la programmation réactive synchrone à la composition musicale. Langage de programmation [cs.PL]. Université Côte d'Azur, 2020. Français. NNT : 2020COAZ4024 . tel-03135288

**HAL Id: tel-03135288**

**https://theses.hal.science/tel-03135288v1**

Submitted on 8 Feb 2021

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

# Temps et Durée

## De la programmation réactive synchrone à la composition musicale


## Bertrand Petit

Inria


**Présentée en vue de l'obtention du grade de docteur en Informatique de l'Université Côte d'Azur.**

**Dirigée par** : Manuel Serrano, Directeur de recherche, Inria

**Co-encadrée par** : François Paris, Compositeur, Directeur du Cirm

**Soutenue le** : 2 juillet 2020

**Devant le jury, composé de** :

Gérard Berry, Professeur émérite, Collège de France

Jean-Pierre Briot, Directeur de Recherche CNRS, Sorbonne Université

Edmund Campion, Directeur du CNMAT, Université de Californie, Berkeley USA

Jean-Louis Giavitto, Directeur de recherche CNRS, IRCAM

Jean-François Trubert, Professeur, Université Côte d'Azur

# Temps et Durée

## De la programmation réactive synchrone à la composition musicale

**Composition du jury :**

**Rapporteurs :**

Jean-Pierre Briot, Directeur de Recherche CNRS, Sorbonne Université
Edmund Campion, Directeur du CNMAT, Université de Californie, Berkeley USA
Jean-Louis Giavitto, Directeur de recherche, CNRS, IRCAM

**Examinateurs :**

Gérard Berry, Professeur émérite, Collège de France
Jean-François Trubert, Professeur, Université Côte d'Azur

**Co-encadrant de thèse**

François Paris, Compositeur, Directeur du CIRM

**Directeur de thèse :**

Manuel Serrano, Directeur de recherche, Inria

# Remerciements

Cette thèse a une longue et étrange histoire puisqu'il est possible de faire remonter son origine à mes premiers développements comme utilisateur *Esterel*. Ce qui nous amène en 1989 lors de mes travaux à l'IMRA en collaboration avec le Centre de Mathématique Appliquée de l'Ecole des Mines de Paris.

Suivons donc l'ordre chronologique en remerciant mes premiers maitres en programmation synchrone qui étaient Frédéric Boussinot et Gérard Berry.

De longues années se sont passées au cours desquelles, bien qu'en étant loin d'*Esterel*, j'ai eu la chance de pouvoir garder des liens avec l'INRIA et le monde de la recherche de Sophia-Antipolis. Parmi ceux qui m'ont permis de maintenir ces contacts :

- Merci à Pierre Bernhard, Directeur de l'INRIA, qui m'a accompagné pendant plus de trois ans (de 1995 à 1998) lors de mon retour sur Sophia-Antipolis.
- Merci à Martin Peronnet, Directeur Général de Monaco Télécom, et Frédéric Fautrier alors Directeur Technique de Monaco Telecom, qui ont soutenu mon initiative en 2007 visant à faire participer Monaco Telecom au GIE Eurécom face à un actionnaire difficile à convaincre.
- Un grand merci aussi à Ulrich Finger, directeur d'Eurécom, ainsi que tous les membres d'Eurécom pour nos projets et échanges durant ma présence au conseil scientifique d'Eurécom (de 2007 à 2016).

Merci aux différents acteurs qui ont permis qu'un concours de circonstances assez inattendues aboutissent à ce projet de thèse :

- François Paris qui m'a accueilli avec enthousiasme en 2016, qui a permis la réalisation de projets musicaux importants avec notre plateforme expérimentale au sein des activités du CIRM et avec lequel les échanges autour des techniques de composition ont toujours été riches et passionnants.
- Gérard Giraudon, qui m'a reçu en 2016 avec bienveillance comme un ami de longue date de l'INRIA et qui m'a mis en contact avec Manuel Serrano.
- Manuel Serrano, qui a accueilli et accompagné pendant trois ans avec beaucoup d'ouverture d'esprit et de patience un doctorant au profil un peu bizarre. Manuel qui a, entre autres, bouclé la boucle en me remettant en contact avec Gérard Berry, 27 ans après mes premiers pas sur *Esterel*.

Merci à ceux qui ont cru dans mes travaux et y ont participé d'une façon ou d'une autre : Camille Giuglaris du CIRM, Anne Schneider et Valérie François de l'INRIA, Gaël Navarre du conservatoire de Nice, Thibaud Ardouin stagiaire ENS, Arianna Corvi stagiaire IPAG, Frédérik Lacroix professeur de musique au collège Nucéra.

Merci aussi à Frank Lalou pour nos nombreuses collaborations musicales et pour son exemple qui est la meilleure démonstration possible de la proposition « qu'un rêve peut toujours devenir réalité ».

Pour finir, merci à mes « challengers » les plus proches sans lesquels je n'aurais pas eu l'idée de soutenir une thèse, les docteurs Isabelle Petit, Tristan Petit, Isabelle Petit-Galvin, Flore de Malglaive et Héloïse Petit.

# Résumé

L'écriture d'une partition est un processus proche de ce que nous appelons une « programmation ». Une partition décrit la succession dans le temps d'événements sonores (hauteur, timbre, durée, articulation) avec des mécanismes de renvoi, de répétition qui sont bien le déroulement d'un « programme » au sens du déroulement dans le temps d'un ensemble d'actions prédéfinies. La composition assistée par ordinateur est une discipline aussi ancienne que l'informatique. Elle s'est essentiellement développée dans la confrontation directe entre des concepts de base propres aux deux disciplines. Cette thèse s'inscrit dans ce mouvement. Elle pose la question de la relation entre l'expressivité des outils de programmation informatique et la création musicale sous l'angle de la question fondamentale de l'expression du temps, qui est un élément commun à la musique et à la science des langages informatiques.

Nous avons choisi de baser notre réflexion sur le temps au moyen du concept bergsonien de *durée.* La durée n'existe que dans la conscience de celui qui perçoit en opposition au *temps objectif* qui est celui des horloges, indépendant des consciences. D'autre part, comme notre problématique est liée à la programmation informatique et donc à l'écriture, nous avons choisi de nous inscrire dans la tradition de la musique écrite, c'est-à-dire préconçue en plus ou moins grande partie sous une forme figée qui permet au compositeur d'évaluer le contenu esthétique de son travail. Fort de ces deux présupposés, nous nous sommes confrontés à la question de faire cohabiter deux termes : écriture et durée. Parmi différentes possibilités de mise en œuvre la durée, nous nous sommes orientés sur les musiques interactives, c'est-à-dire les réalisations musicales contrôlées en partie par l'audience et non seulement par les musiciens ou par le compositeur. Ceci nous a conduit à développer un environnement informatique particulier et définir une méthode de composition musicale dans le domaine des musiques interactives et donc, en complément des questions liées aux langages informatiques, de nous intéresser aux problèmes d'interaction homme/machine.

Nos travaux nous ont conduit à élaborer une méthode de composition qui repose sur trois concepts : des éléments musicaux de base, des groupes d'éléments de base et des orchestrations. Les *éléments musicaux de base* sont des éléments sonores relativement courts, phrases musicales ou sons. Ils sont organisés en *groupes*. Ces groupes sont mis à la disposition de l'audience qui, en sélectionnant tel ou tel élément de base ou en répondant à des choix qui lui seront proposés, participe à la concrétisation de la musique. L'organisation des groupes et la façon dont ils sont mis à la disposition de l'audience constituent *l'orchestration.* Le concept d'orchestration est difficile à mettre en œuvre. C'est sur ce point particulier et sur la mise en œuvre de l'interaction que la question de l'expressivité des langages informatiques a été déterminante. Nous avons choisi de construire notre méthode de composition en la confrontant au langage réactif synchrone *HipHop.js.* En effet, ce langage, conçu à partir de questionnements sur le temps, combine la programmation d'automates complexes adaptée à notre concept d'orchestration avec la programmation Web particulièrement adaptée à la mise en œuvre d'interactions à grande échelle. Nous avons alimenté cette recherche par différentes expériences et productions musicales au moyen d'une plateforme logicielle que nous avons développée et baptisée Skini.

**Mots-clés :** programmation réactive synchrone, composition musicale, musique interactive.

# Abstract

Writing a score is a process close to what we call "programming". A score describes the succession in time of sound events (pitch, timbre, duration, articulation) with mechanisms of return, of repetition which are indeed the behavior of a "program" in the sense of the succession in time of a set of predefined actions. Computer Assisted Composition is a discipline as old as computer science. It has essentially developed in the direct confrontation between basic concepts specific to the two disciplines. This thesis is part of this movement by posing the question of the relationship between the expressiveness of computer programming tools and musical creation, from the perspective of the fundamental question of the expression of time, which is a common element in music and in the science of computer languages.

We have chosen to base our thinking on time using the Bergson's concept of duration. Duration exists only in the consciousness of the perceiver as opposed to "objective time", which is that of clocks, independent of consciousness. On the other hand, as our problem is linked to computer programming and therefore to writing, we have chosen to be part of the tradition of written music, i.e. predefined more or less in a fixed form that allows the composer to evaluate the aesthetic content of his work. With these two assumptions in mind, we were confronted with the question of the cohabitation of two terms: writing and duration. Among the various possibilities for implementing duration, we have focused on interactive music, i.e. musical productions controlled in part by the audience and not only by the musicians or the composer. This led us to develop a dedicated computer environment and to define a method of musical composition in the field of interactive music and thus, in addition to the questions related to computer languages, to address some problems of human/machine interaction.

Our work has led us to develop a method of composition based on three concepts: basic musical elements, groups of basic elements and orchestrations. Basic musical elements are relatively short sound elements, musical sentences, or sounds. They are organized in groups. These groups are made available to the audience, who, by selecting one or another basic element or by responding to choices that will be proposed to them, participates in the production of the music. The organization of the groups and the way they are made available to the audience constitute the orchestration. The concept of orchestration is difficult to implement. It is on this specific point, and on the implementation of the interaction, that the question of the expressiveness of computer languages has been decisive. We chose to build our composition method by confronting it with the *synchronous reactive language* HipHop.js. Indeed, this language, designed from questions about time, combines the programming of complex automata adapted to our concept of orchestration with Web programming, particularly adapted to the implementation of large-scale interactions. We fed this research with different musical experiments and productions using a software platform we developed and named Skini.

**Keywords:** reactive synchronous programming, musical composition, interactive music

TABLE DES MATIERES


1 INTRODUCTION 5

2 ETAT DE L'ART 11

2.1 PREAMBULE TECHNOLOGIQUE A L'EXPRESSION DE LA DUREE 13
2.1.1 MUSIQUE ET EXPRESSIVITE DES OUTILS INFORMATIQUES 13
2.1.2 LES MUSIQUES INTERACTIVES 16
2.1.3 LES TECHNOLOGIES DU WEB APPLIQUEES A LA MUSIQUE 20
2.2 L'EXPRESSION DU TEMPS EN MUSIQUE 22
2.2.1 LA PARTITION ET SON EVOLUTION 22
2.2.2 DES EXEMPLES D'EXPRESSION DU TEMPS EN MAO 24
2.2.3 LES MUSIQUES IMPROVISEES 25
2.2.4 L'ALEATOIRE DANS LA MUSIQUE 26
2.2.5 EN-TEMPS, HORS-TEMPS 26
2.2.6 LES MUSIQUES INTERACTIVES 27
2.3 LES METHODES DE COMPOSITION UTILISANT DES PATTERNS 28
2.3.1 THEME, MOTIF, CELLULE 28
2.3.2 COMPOSITION PAR CLIPS 29
2.3.3 PATTERNS D'IMPROVISATION 29
2.4 CONCLUSION DE L'ETAT DE L'ART 30

3 EVALUATION OPERATIONNELLE DE LA COMPOSITION AVEC PATTERNS 31

3.1 EMULER UNE IMPROVISATION JAZZ 33
3.2 EMULER AVEC DES TONALITES 35
3.3 EMULER LE STYLE CLASSIQUE 36
3.4 EMULER LE STYLE MODAL 39
3.5 ECRIRE SANS TONALITE 39
3.5.1 A LA SCHOENBERG 40
3.5.2 A LA MESSIAEN 40
3.5.3 COMPOSER AVEC DES SONS 40
3.6 CONCLUSION : LIMITES ET OPPORTUNITES 41

4 PENSER L'ORCHESTRATION PAR PATTERN ET AUTOMATE 43

5 LA PROGRAMMATION AVEC HIPHOP.JS 53

5.1 PRINCIPE DE LA PROGRAMMATION REACTIVE SYNCHRONE 56
5.2 PROGRAMMER AVEC HIPHOP.JS 56
5.2.1 PENSER A L'ARCHITECTURE 56
5.2.2 HIPHOP.JS ET LE TEMPS : DELAI ET INSTANT 57
5.2.3 LES INSTRUCTIONS HIPHOP.JS 58
5.2.4 ACTIVER UN MODULE DEPUIS JAVASCRIPT 60

5.2.5 PREEMPTION 61
5.2.6 CONSTRUCTION DYNAMIQUE DE PROGRAMME 61
5.2.7 HYPOTHESE INSTANTANEE 62
**5.3 HIPHOP.JS VS. JAVASCRIPT 63**

**6 REALISATION D'ORCHESTRATIONS PAR AUTOMATE 67**

**6.1 STRUCTURE DE LA PLATEFORME D'EXPERIMENTATION 69**
6.1.1 LA MATRICE DES POSSIBLES 70
6.1.2 LES FILES D'ATTENTES 70
6.1.3 LA MISE EN ŒUVRE DU TEMPS DES HORLOGES 71
**6.2 PROGRAMMATION DE PERFORMANCES 72**
6.2.1 GESTION DES SELECTIONS DE PATTERNS 72
6.2.2 GROUPES REPETITIFS ET RESERVOIRS 73
6.2.3 CONFIGURATION DES PATTERNS ET GROUPES 76
6.2.4 L'USINE 78
6.2.5 GRAND LOUP 80
6.2.6 OPUS 1 82
6.2.7 OPUS 2 86
6.2.8 UNE AUTRE ORCHESTRATION DE L'OPUS1 88
**6.3 CONCLUSIONS SUR LA PROGRAMMATION DES ORCHESTRATIONS 91**

**7 MISE EN ŒUVRE DE LA DUREE 95**

**7.1 L'INTERACTION 98**
7.1.1 LES NIVEAUX D'INTERACTION 99
7.1.2 LES INTERFACES 101
7.1.3 LE SEQUENCEUR DISTRIBUE 110
**7.2 LE SIMULATEUR 114**
**7.3 CONCLUSIONS SUR L'INTERACTION 118**

**8 ARCHITECTURE TECHNIQUE 121**

**8.1 LES SERVICE WEB 124**
**8.2 LES COMMANDES DU SON 124**
**8.3 LA DIGITAL AUDIO WORKSTATION 125**
**8.4 L'AFFICHAGE 125**
**8.5 L'ARCHITECTURE DU SEQUENCEUR DISTRIBUE 125**
**8.6 LE RESEAU 125**
**8.7 CONCLUSION SUR L'ARCHITECTURE TECHNIQUE 126**

**9 LES RETOURS D'EXPERIENCES 127**

**9.1 LE GOLEM 129**
9.1.1 LE 8 DECEMBRE 2017 130
9.1.2 LE 9 DECEMBRE 2017 132
9.1.3 CONCLUSION DU GOLEM 135

**9.2 LA FABRIQUE A MUSIQUE** **136**
9.2.1 L'USINE 136
9.2.2 CONCLUSION DE LA FABRIQUE A MUSIQUE 137
**9.3 D'AUTRES PERFORMANCES** **138**
**9.4 CONCLUSION DES RETOURS D'EXPERIENCE** **138**

**10 CONCLUSION ET PERSPECTIVES** **141**

**11 GLOSSAIRE** **147**

**REFERENCES BIBLIOGRAPHIQUES** **151**

# 1 INTRODUCTION

> *Couramment, quand nous parlons du temps, nous pensons à la mesure de la durée, et non pas à la durée même.* (La pensée et le mouvant, Henri Bergson)

L'expression du temps est un problème commun à la musique écrite et à l'informatique. Les deux disciplines relevant de compétences différentes, apportent des solutions différentes. Cependant certaines analogies sont naturelles. Ainsi l'écriture d'une partition ressemble à un processus de programmation. Une partition décrit la succession dans le temps d'événements sonores (hauteur, timbre, durée, articulation) avec des mécanismes de renvoi, de répétition qui sont bien le déroulement d'un « programme » au sens du déroulement dans le temps d'un ensemble d'actions prédéfinies. Les deux disciplines dialoguent depuis des décennies. La Composition Assistée par Ordinateur (CAO) et la Musique Assistée par Ordinateur (MAO) sont des disciplines aussi anciennes que l'informatique. Elles se sont développées dans la confrontation directe entre des concepts de base propres aux deux univers, musical et technique. Cette thèse s'inscrit dans ce mouvement en posant la question de la relation entre programmation informatique et création musicale sous l'angle de la question fondamentale de l'expression du temps, qui est un élément commun à la musique et à l'informatique. Il s'agit donc en premier lieu de réfléchir à ce qu'est le *temps*, ce qui nous amènera à préciser comment nous allons décliner notre recherche.

Dans l'histoire de la philosophie, la question du temps a été abordée par de nombreux philosophes, Héraclite, Saint Augustin, Kant et d'autres. Nous avons choisi de nous inspirer de la pensée d'Henri Bergson d'une part car il a largement traité le sujet, d'autre part encore parce qu'il s'agit d'un philosophe proche de la science qui a dialogué avec les grands scientifiques de son époque comme Albert Einstein notamment, dans son ouvrage *Durée et Simultanéité* [38].

La position de Bergson consiste à distinguer deux façons d'appréhender le temps qui se confondent dans le langage commun. Tout d'abord le temps qui se mesure, que l'on nomme aussi *temps objectif,* qui est une extrapolation de la notion d'espace et d'autre part *la durée réelle*. Le temps objectif et l'espace étant mis sur le même plan, « *il a suffi de changer un mot : on a remplacé « juxtaposition » par « succession » ».* (La pensée et le mouvant, Henri Bergson [39]), pour se détourner de la durée réelle. Bergson explique la définition commune du temps conformément au temps objectif, comme une habitude du langage qui ne peut traiter la question qu'en surface. « *La durée s'exprime toujours en étendue. Les termes qui désignent le temps sont empruntés à la langue de l'espace. Quand nous évoquons le temps, c'est l'espace qui répond à l'appel. La métaphysique a dû se conformer aux habitudes du langage, lesquelles se règlent elles-mêmes sur celles du sens commun* » (La pensée et le mouvant, Henri Bergson, [39]). Bergson ne rejette pas en bloc cette approche du temps, elle répond à un besoin de l'entendement. C'est elle qui a permis le développement de la science. La science raisonne sur des points fixes. Elle exprime la mobilité sous la forme d'intervalles, allant jusqu'à repousser la dimension des intervalles à l'infiniment petit pour exprimer la continuité. Pour Bergson cette démarche est naturelle car nos actions s'exercent plus facilement sur des points fixes et l'intelligence a pour objectif de guider nos actions. Néanmoins à ce temps mesuré Bergson oppose la notion de *durée*. Il s'oppose ainsi au mouvement positiviste développé au XIXème siècle, notamment par Auguste Comte, qui donne la primauté à la science. Il soutient que la vision du temps proposée par la science, certes efficace, ne peut pas le saisir dans sa complexité. Il illustre son approche selon son habitude à travers la célèbre métaphore du verre d'eau sucrée : « *Quand*

*on veut préparer un verre d'eau sucrée, avons-nous dit, force est bien d'attendre que le sucre fonde. Cette nécessité d'attendre est le fait significatif. Elle exprime que, si l'on peut découper dans l'univers des systèmes pour lesquels le temps n'est qu'une abstraction, une relation, un nombre, l'univers lui-même est autre chose. Si nous pouvions l'embrasser dans son ensemble, inorganique mais entretissu d'êtres organisés, nous le verrions prendre sans cesse des formes aussi neuves, aussi originales, aussi imprévisibles que nos états de conscience »* (*Ibid*.)

Sans entrer dans la justification de cet extrait - elle fait suite à ce texte dans l'ouvrage de Bergson - notons qu'il met en avant que ce qui est pertinent pour notre conscience, ce n'est pas la mesure du temps mais l'attente nécessaire pour que l'eau soit sucrée. Pour la conscience, le temps mesuré n'est pas significatif, un spectacle ennuyeux nous semblera long, un spectacle attrayant nous semblera court, même si leurs durées mesurées sont les mêmes. Il y a donc bien une distinction nette entre le *temps mesuré,* et le *temps de la conscience* que Bergson appelle la *durée*. Il s'agit d'une différence majeure entre la durée telle que nous la vivons et le temps de la science qui est une construction intellectuelle et mesurable. Le philosophe considère donc que l'écoulement du temps se saisit par *l'intuition* et non l'entendement. Par conséquent, une vision purement calculée, raisonnée ne peut traiter que la dimension purement quantifiable du temps en passant à côté de sa dimension subjective. Pour produire de la musique, il ne suffit pas de penser la durée, il faut aussi la mettre en œuvre. C'est pour cela que dans son ouvrage « la pensée et le mouvant », Bergson aborde le thème de la *continuité de transition*, du *flux*, du changement. Ce qui est réel, ce ne sont pas des états positionnés sur une ligne représentant le temps, c'est plutôt un flux, la continuité de ce flux. Ce qui fait la *réalité* d'une mélodie par exemple, n'est pas une succession de notes, mais bien un mouvement auquel nous associons une émotion. Notre recherche s'est fortement inspirée de la notion de flux même si, comme nous le verrons, nous ne lui donnons pas la dimension métaphysique que lui donne Bergson. Elle s'est aussi inspirée de la distinction entre *temps mesuré* et *durée*.

En parallèle aux évolutions des conceptions scientifiques du monde, la musique écrite occidentale s'est orientée vers une écriture essentiellement basée sur le temps objectif. La majorité des partitions s'expriment depuis le moyen-âge sur deux axes : hauteur et temps. Le temps est donc représenté dans l'espace selon la logique d'un temps mesuré, même si la mesure n'est ni précise, ni stable. Encore aujourd'hui, le développement récent et révolutionnaire dans son genre qu'est Antescofo [19] assouplit la notion de temps mesuré pour le contrôle des musiques électroniques, mais il reste fondamentalement conforme à une représentation spatiale du temps. Il permet certes de modifier les tempi des systèmes électroniques selon le comportement des musiciens, mais l'œuvre reste fondamentalement pensée pour se dérouler sur un axe du temps en suivant une partition.

Il existe des musiques qui ne sont pas basées sur des représentations en fonction du temps mesuré. Les musiques improvisées, les œuvres dites ouvertes, les musiques collaboratives ou les musiques aléatoires peuvent être basées sur d'autres modes de représentations que des partitions. Cependant, ces musiques quand elles font appel à l'écrit, ne remettent en général pas en cause la représentation spatiale du temps. Nous pensons que si la musique écrite ne sait pas s'extraire d'une représentation spatiale, ce n'est pas parce que les compositeurs ne le souhaitent pas. Il y a bon nombre d'œuvres du XXème siècle dans lesquelles des compositeurs proposent des partitions expérimentant de nouvelles représentations. Le besoin existe donc bel et bien de formaliser la musique autrement qu'avec des partitions avec mesures et portées. Mais il faut reconnaitre que les compositeurs, encore récemment, n'avaient pas beaucoup d'outils efficaces pour formaliser leurs idées de façon rigoureuse et simple dans un contexte où le temps n'est pas celui des horloges.

Le phénomène est assez similaire en informatique. La plupart des langages s'expriment de façon séquentielle et ne prennent pas en compte l'expression du temps sous une forme qui ne soit pas uniquement linéaire. Il y a peu de langages qui soient capable d'exprimer simplement le parallélisme par exemple. Comme en musique, l'expression du temps en informatique est un problème. Parmi les langages informatiques, il en existe cependant qui apportent des solutions à l'expression du temps sous forme d'événements et qui se sont posés la question du temps sous une forme proche de la nôtre. Le cœur de notre problématique sera de voir dans quelle mesure les réponses apportées par ces langages peuvent s'étendre à la musique.

Pour rendre notre approche plus concrète nous devons suivre Bergson un peu plus. Pour lui la réalité ne s'exprime pas par des états juxtaposés, mais par un flux de changement, par un acte de création permanent que l'on retrouvera généralisé sous forme du concept d'*élan vital*. La durée est constitutive de l'être vivant et au vivant Bergson associe la *conscience*. Or les langages informatiques, qui sont à la base de notre recherche, même s'ils peuvent faciliter l'expression de la durée, ne se préoccupent pas de conscience. Les machines ont assez peu à voir avec la métaphysique. Même si les développements de l'Intelligence Artificielle et des Réseaux de Neurones réalisent des tâches spectaculaires, rien ne prouve qu'il soit possible de considérer qu'une machine puisse un jour posséder une conscience de type humaine. Pour rendre opérationnel un système informatique construit autour d'une notion de durée qui se rapprocherait de celle de Bergson, il nous fallait introduire un *niveau de conscience*. Parmi les moyens possibles d'introduire ce niveau nous avons choisi de nous adresser non seulement au compositeur, aux interprètes mais aussi à *l'audience*. Ceci est un choix parfaitement subjectif, justifié en partie par l'intuition qu'une intelligence collective est plus riche qu'une intelligence isolée. C'est le choix que font la plupart des systèmes de *musiques collaboratives ou interactives*, qui constituent une des branches techniques utilisant des programmes informatiques pour produire de la musique.

Notre problématique peut donc se résumer ainsi : quelle pourrait être une méthode de composition musicale interactive mettant en œuvre un langage informatique adapté à l'expression de la *durée* dans un sens proche de celui que Bergson donne à ce terme ?

Nous verrons qu'un langage informatique et le choix de nous orienter vers des musiques interactives ne sont pas suffisants pour mettre en œuvre un environnement informatique et une méthode de composition. Nous aurons besoin d'introduire un autre concept important, celui d'élément musical élémentaire que nous nommerons *pattern,* ainsi que celui d'*orchestration* que nous utilisons dans un sens à la fois musical et informatique. Nous abordons ces concepts de différentes façons dans cette thèse, selon les étapes suivantes : La première étape est un état de l'art des différents thèmes traités par notre recherche en matière d'expressivité des outils informatiques, de technologie, d'expression du temps en musique et de compositions par pattern. L'étape suivante consiste à introduire notre méthodologie et sa mise en œuvre. Il s'agira d'abord d'évaluer quelques principes de la composition par patterns et dans un deuxième temps d'étendre ces principes pour obtenir les bases d'une méthode de composition en développant le concept d'orchestration. Nous passerons ensuite aux aspects informatiques du choix du langage en fonction de la méthode retenue, puis à la mise en œuvre du langage au moyen d'une plateforme que nous avons développée pour aboutir à des réalisations concrètes et que nous avons baptisée Skini[1]. Nous aborderons le fonctionnement de cette plateforme d'expérimentation. Elle sera suivie de descriptions des

[1] Skini signifie scène en grec. C'est l'acronyme que nous avions choisi au début de notre projet dans un contexte plus large que celui présenté ici. Mais nous avons décidé de garder cet acronyme car nous verrons que notre approche musicale peut effectivement s'étendre à la scène.

premières réalisations et évaluations de notre projet de mise en œuvre de la durée selon l'expres-sivité du langage informatique que nous avons retenu. Nous finirons en envisageant de nouvelles perspectives à nos travaux.

# 2 ETAT DE L'ART

PREAMBULE TECHNOLOGIQUE A L'EXPRESSION DE LA DUREE 13

MUSIQUE ET EXPRESSIVITE DES OUTILS INFORMATIQUES 13
LES MUSIQUES INTERACTIVES 16
LES TECHNOLOGIES DU WEB APPLIQUEES A LA MUSIQUE 20

L'EXPRESSION DU TEMPS EN MUSIQUE 22

LA PARTITION ET SON EVOLUTION 22
DES EXEMPLES D'EXPRESSION DU TEMPS EN MAO 24
LES MUSIQUES IMPROVISEES 25
L'ALEATOIRE DANS LA MUSIQUE 26
EN-TEMPS, HORS-TEMPS 26
LES MUSIQUES INTERACTIVES 27

LES METHODES DE COMPOSITION UTILISANT DES PATTERNS 28

THEME, MOTIF, CELLULE 28
COMPOSITION PAR CLIPS 29
PATTERNS D'IMPROVISATION 29

CONCLUSION DE L'ETAT DE L'ART 30

Nous allons à présent aborder un état de l'art des domaines qui nous concernent. Il s'agit tout d'abord de questions technologiques. Nous passerons ensuite à l'expression du temps en musique sous différentes formes, écrites, « live coding », improvisées, aléatoires, « en-temps et hors temps », et bien sûr interactives. Nous aborderons pour finir ce chapitre le sujet de la composition par pattern, car il s'agit d'un des principes élémentaires que nous avons retenus pour mettre en œuvre l'environnement qui a servi de base à nos travaux. Ce principe nous semble parfaitement adapté au modèle de représentation de la durée que nous avons choisi de mettre en œuvre.

## 2.1 Préambule technologique à l'expression de la durée

Nous avons abordé en introduction un certain nombre de questions qui constituent la trame de fond de notre recherche, c'est à dire de l'expression du temps et de la durée. Nous avions conclu de la difficulté pour un compositeur *d'exprimer* la durée, notamment à cause d'un manque d'outil. Avant de parler d'expression de la durée, nous commencerons donc par définir ce qu'est *l'expressivité* d'un outil informatique. Nous avions aussi conclu que l'interaction avec l'audience était une façon d'introduire une *durée* proche du sens que Bergson donne à ce terme, en faisant entrer des « consciences » dans le processus de production de la musique. Rappelons que cette façon d'introduire de la durée dans la musique n'est pas la seule et qu'il s'agit là d'un choix personnel qui s'inscrit dans la mouvance des « œuvres ouvertes » au sens d'U. Eco [23] et des « musiques interactives ». Nous verrons comment la technologie a permis un nombre important de développements dans ce domaine. Partant de ce choix de nous intéresser à l'interaction, nous évoquerons les technologies du Web. En effet, le Web est la mise en œuvre, la plus spectaculaire dans l'histoire des technologies, de mécanismes d'interaction à grande échelle.

### 2.1.1 Musique et expressivité des outils informatiques

Nous verrons qu'au cours de nos travaux nous avons eu besoin de définir et trouver des outils qui nous permettent d'atteindre des objectifs. Or il nous a semblé, dans notre cas, que poser la question des outils sous l'angle d'une relation de cause à effet, d'un outil venant résoudre un problème, n'était pas suffisante. La relation entre un outil et la réponse à une question artistique est plutôt circulaire que linéaire. L'outil apporte une réponse, mais la réponse remet en cause la question, et ainsi de suite dans un processus dialectique auquel l'artiste décidera de mettre temporairement ou définitivement fin de façon subjective. C'est cette subjectivité qui définit en partie le geste de l'artiste qui utilise des outils informatiques. Nous avons donc décidé d'aborder en premier lieu pour cet état de l'art, la question de la relation entre *outil informatique* et *création musicale*.

Posons ce que nous entendons par les *outils informatiques.* Ils regroupent un ou plusieurs *langages informatiques* ainsi qu'un *environnement* qui permet leur usage dans un contexte particulier. Un environnement informatique sera par exemple une plateforme de génie logiciel, une base de données avec son langage d'interrogation. Nous n'avons pas voulu faire une distinction forte entre ces deux termes pour parler d'expressivité des concepts musicaux et plus précisément du temps. En effet, un langage n'est expressif que dans un contexte et donc dans les environnements qui le mettent en œuvre. Par ailleurs, la distinction entre langage et environnement n'est pas toujours aisée. Dans le cas de MAX/MSP par exemple, qui est un outil majeur de la Musique Assistée par Ordinateur (MAO), il sera possible de parler de *langage graphique*, c'est ainsi que la société qui commercialise cette solution la présente. Mais il s'agit aussi d'un environnement de développement informatique complet dédié à la musique avec un éditeur graphique, un débugger, des interfaces vers des équipements électroniques et beaucoup d'autres choses.

En partant du principe qu'une partition est un processus de programmation, nous devons considérer les différents moyens qu'une solution de MAO va utiliser pour mettre en forme de la musique à l'aide de langages de programmation et d'environnements informatiques. Ayant défini ce que nous entendons par *outil informatique*, notre réflexion autour de *l'expressivité* part du constat que même s'il est possible de traiter le même problème avec des langages ou des environnements différents, les outils informatiques ne sont pas équivalents pour aboutir à un code machine ou des comportements quasiment similaires. Les concepts de base et les architectures logicielles qu'ils permettent de mettre en œuvre diffèrent et cela a un impact sur les processus de création, ce qui génère la circularité entre outil et processus de création. Avant d'aller plus loin, précisons ce que nous entendons par *expressivité*.

L'*expressivité* est caractérisée par la capacité d'un langage ou d'un environnement à exprimer des concepts et à proposer des structures de traitement de l'information en conséquence. En nous permettant un rapprochement linguistique imagé, les symboles, la syntaxe et la grammaire d'une langue permettront d'exprimer mieux certaines idées et certains concepts que d'autres. Les langues à idéogrammes, comme le japonais ou le chinois, demandent un long apprentissage des caractères par exemple. Il faut environ quinze ans pour qu'un jeune japonais soit à l'aise pour lire un journal. On imagine aisément que la charge émotive attachée à un idéogramme en japonais n'est pas la même que celle que nous attachons à un caractère de l'alphabet. D'autant plus qu'un idéogramme peut véhiculer des sens généraux, qui prennent un sens précis quand ils sont associés à d'autres caractères. Ceci crée des possibilités d'enchevêtrements sémantiques fort riches et donne aux poésies japonaises et chinoises des dimensions inaccessibles à la traduction. Si le japonais et le chinois sont particulièrement performants dans la capacité de suggestion par les symboles mêmes, ils le seront moins dans l'expression de concepts précis. A ce titre on considère souvent que l'allemand avec sa grande capacité de combinaison lexicale est mieux adapté à l'élaboration de concepts abstraits que les langues latines par exemple. Dans ce sens *l'expressivité* des langues est variable. Le phénomène est le même en informatique. Par exemple, le concept de *signal* des langages synchrones n'existe pas dans les langages généralistes comme le langage C ou Java. Penser un programme à l'aide de *signaux* constitue une démarche différente de celle qui consiste à utiliser des *variables entières.* Elle sera mieux adaptée à la programmation d'automates. Un programme construit selon des principes de *programmation orienté-objets* sera plus facile à faire évoluer qu'un programme conçu simplement de façon *impérative*. La *programmation fonctionnelle* proposée par *JavaScript* permettra de mettre en œuvre très rapidement des prototypes, mais son absence de typage conduira à des résultats moins fiables que *Java*.

Les différences de langage conduisent à des façons de « programmer », et donc de penser qui peuvent être radicalement différentes[2]. De la même façon, en MAO chaque outil et le langage dans lequel il exprime la musique, répond à des problématiques de composition musicale qui lui sont propres. Par exemple, le concept de *clip* d'Ableton n'existe pas dans les éditeurs de partition. Il permet une description non linéaire de la musique qui est difficile à mettre en œuvre avec nos seules mesures et portées. Chaque solution de MAO en adressant des problématiques différentes et en proposant un langage adéquat, définit ou oriente telle ou telle méthode de composition. De la même façon que le média oriente le message selon la théorie de Mc Luhan, nous pensons que l'expressivité d'un outil de MAO qui met en œuvre un langage informatique textuel ou graphique,

[2] L'expressivité d'un langage ou d'un environnement peut avoir un rôle déterminant non seulement sur la capacité d'un programme à traiter un problème, mais aussi sur sa capacité à évoluer et à être maintenu. Ce qui en conditionne fortement son usage.

aura un lien plus ou moins fort avec une méthode de composition et proposera notamment des façons différentes d'exprimer le temps comme nous le verrons dans un autre chapitre. Prenons quelques exemples de solutions de MAO qui impactent la pensée musicale : le *Live Coding*, qui consiste à produire de la musique sur scène à partir de programmes informatiques écrits durant la performance, propose une relation entre création musicale et outil, où l'acte de programmation fait partie intégrante de la production musicale. La méthode de composition est ici directement impactée par l'expressivité du langage informatique utilisé puisqu'elle repose intégralement sur lui. Un autre exemple d'impact évident d'un outil et d'un langage graphique sur la méthode de composition est la solution *Ableton Live* [2], développée par la société berlinoise *Ableton* créée par des musiciens issus de la mouvance des musiques *Electro* berlinoise. Cet outil propose très clairement une méthode de composition graphique à base de « clips » en rupture avec la vision linéaire de nos portées et mesures et de leurs évolutions sous forme de séquenceurs. Les outils de programmation *MAX/MSP* [71], *PureData* [81] ou *Reaktor* [77] de *Native Instruments* sont d'autres exemples de la façon dont des langages et leurs environnements orientent le travail du compositeur. Les primitives de ces langages graphiques généralistes sont très nombreuses (plus de 200 pour PureData) selon le créateur de ces solutions Miller Puckette [72] . L'influence de ces langages sur l'acte de composition est liée à leur complexité. Même s'ils ont été conçus pour permettre un accès initial rapide, la maitrise de ces outils demande un travail informatique long et fastidieux. Cet effort important d'acquisition de compétence, masqué par une interface graphique, incite le compositeur à développer des compétences dans ce domaine parfois au détriment d'autres compétences musicales.

Nous voyons avec ces quelques exemples, qu'à des niveaux différents et avec des impacts plus ou moins forts un outil de MAO orientera la façon dont pense et crée un compositeur. Bien qu'elle soit repérable sur des solutions existantes, la circularité ou dialectique évoquée entre outil et création musicale est difficile à analyser, probablement encore plus pour nous puisqu'elle est au cœur de notre démarche, qui est une démarche de recherche sous forme de création qui est donc emprunte d'une bonne part de subjectivité. Même si nous présentons dans ce document notre démarche de façon linéaire, c'est bien la mise en œuvre de cette circularité qui est la base de notre recherche et qui fait que notre contribution s'inscrit totalement dans la lignée des développements autour de la MAO. Notre spécificité est de nous intéresser plus particulièrement à *l'expression du temps et de la durée*. Comme nous le verrons, si la question de l'expression du temps est nécessairement toujours traitée plus ou moins clairement et complétement par les systèmes de MAO, la question de l'expression de la durée est peu abordée en tant que telle. Comme nous l'avions vu en introduction, nous avons choisi de mettre en œuvre la durée en nous orientant vers les musiques interactives. L'état de l'art de ce type de musiques occupe le prochain chapitre.

### 2.1.2 Les musiques interactives

> *La poétique de l'œuvre en mouvement (et en partie aussi, celle de l'œuvre ouverte) instaure un nouveau type de rapport entre l'artiste et son public, un nouveau fonctionnement de la perception esthétique ; elle assure au produit artistique une place nouvelle dans la société. Elle établit enfin un rapport inédit entre la contemplation et l'utilisation de l'œuvre d'art.* (*L'œuvre ouverte*, U. Eco, p.37).

Les musiques interactives ou collaboratives se caractérisent par une distinction floue entre interprète et audience. Tous les participants sont appelés à contribuer activement au spectacle, à la performance. Cette forme de spectacle a une longue histoire dans de nombreuses cultures et sous des formes variées. La musique reste encore associée à une pratique collective, la danse, dans toutes les cultures et demeure une façon de renforcer les liens sociaux. On peut même imaginer que la musique ait été en grande partie une pratique collective, avant d'être l'apanage de quelques individus. Le développement des musiques interactives ne serait donc qu'un juste retour des choses en apportant une dimension oubliée, celle de créer ensemble. Par ailleurs les musiques collaboratives étendent au public un mouvement intellectuel des années 60, qui regroupe des intellectuels comme le philosophe U. Eco, des musiciens comme K. Stockhausen, K. Penderecki ou L. Berio. Ces musiciens ont créé des pièces artistiques dynamiques où les interprètes choisissent la façon dont l'œuvre va se dérouler indépendamment du compositeur. Eco a inventé le terme *d'œuvre ouverte* [32] pour désigner ces œuvres où les interprètes participent activement à la création du compositeur, afin de réaliser le projet esthétique final. U. Eco a soulevé la question de la qualité de l'œuvre et la difficulté pour les compositeurs de garantir la qualité de leurs productions, puisqu'une partie de la production musicale est déléguée. Le contrôle de la qualité de la production musicale est une question centrale de notre plateforme. Il s'est agi pour nous de trouver une balance entre un haut degré d'interaction avec le public et la mise en œuvre du projet esthétique du compositeur. En considérant les deux extrêmes, si l'interaction est trop incontrôlée, le projet musical peut perdre la cohérence qu'aura voulu lui donner le compositeur, et inversement, si l'œuvre est figée, l'interaction sera sans effet. L'art de la composition de musique interactive sera de trouver un équilibre entre ces deux extrêmes et de permettre aux initiatives de l'audience de faire partie du projet esthétique du compositeur.

De façon à éclaircir le positionnement de nos travaux dans le contexte des musiques interactives, ce chapitre propose un tour d'horizon de quelques réalisations significatives dans ce domaine. Nous pouvons commencer dans les Philippines. En 1974, le professeur Jose Maceda mit en place une performance (Ugnayan) faisant appel à des postes de radio [65]. L'idée était d'associer des participants en masse en diffusant des sons sur 37 canaux radio[3] dans la ville de Manille. Canaux que tout un chacun pouvait sélectionner et diffuser dans la rue pour créer un son collectif, une forme de collage sonore collaboratif. Si cette première initiative a été suivie de beaucoup d'autres, c'est le

[3] L'utilisation de la radio pour la production musicale avait déjà été abordé par John Cage dans *Imaginary Landscape* N°4 (1951) ou *Speech* (1955), Karlheinz Stockhausen s'était aussi intéressé au sujet avec notamment *Kurzwellen* (1968) ou *Spiral* (1968). Ces réalisations ne faisaient pas encore appel au public

développement des smartphones qui introduit un changement d'échelle dans les possibilités d'étendre l'interaction. Voyons quelques projets importants mettant en œuvre des smartphones.

***Dialtone,* A *Telesymphony*** en 2001 [54] qui est souvent citée comme la première œuvre qui ait utilisé des smartphones pour une performance musicale et qui ait démontré le potentiel du téléphone mobile comme interface. Benjamin Taylor dans son article "A History of the Audience as a Speaker Array" de 2017 [86] fait un tour d'horizon circonstancié de l'utilisation des smartphones comme en ensemble de haut-parleurs. L'idée d'utiliser les haut-parleurs des smartphones lors de performances a depuis largement été exploitée. Le projet COSIMA de l'*Ircam* [49] s'inscrit dans cette continuité en utilisant le smartphone comme un instrument de musique et en intégrant les derniers développements des technologies Web Audio du W3C. Les projets FAUST du GRAME [63] ou Kronos du Center for Music & Technology de l'académie Sibelius en Finlande [62] s'inscrivent aussi dans cette mouvance en proposant des outils de développement permettant de créer des instruments, entre autres pour smartphones.

Avec le développement des smartphones, les initiatives en matière de musique interactive ont naturellement décuplé. Nous avons choisi un nombre réduit d'exemples en nous basant sur leur pertinence vis-à-vis de nos travaux. La plupart des articles cités ici incluent des états de l'art. A ce titre l'article *Open Symphony: Creative Participation for Audiences of Live Music Performances* [93], propose de plus une classification des solutions utilisées par des performances *live* interactives. Cette classification est basée sur le niveau de participation, allant de partielle, où l'audience influence le contenu produit par des artistes, à totale, où il n'y a plus de différence entre artiste et audience. Nous reviendrons sur cette classification plus tard.

***Open Symphony*** [93] est un système récent (publication de 2017) interactif basé sur les technologies du Web. Il permet à l'audience de contrôler une partition à l'aide de mécanisme de vote. La partition est jouée par des musiciens. Nous verrons que comme pour *Open Symphony*, nos travaux proposent à l'audience d'agir comme un « méta-compositeur/chef d'orchestre ». Mais à la différence d'*Open Symphony*, nos travaux ne reposent pas uniquement sur un mécanisme de vote pour activer la partition en train de se jouer.

La plateforme ***massMobile*** [91] est un système client-serveur développé pour la participation en masse d'une audience lors d'une performance à partir de smartphones. Ce projet dans sa définition est très proche de notre contexte. Il a été utilisé pour des performances à grande échelle. L'architecture globale repose sur un serveur Java (SOAP et GlassFish) interfacé avec *MAX/MSP* [71] via des API développées pour le projet. Il s'agit d'un projet qui date de 2011, avec une architecture conforme aux technologies populaires à cette époque. Le système est capable de contrôler l'interface graphique des smartphones via *MAX/MSP*. L'architecture globale est néanmoins le seul point commun avec nos travaux. Nous verrons que nous avons un projet musical plus précis et que nous n'utilisons pas *MAX/MSP* comme outil de production du son. *MAX/MSP* est un outil aux fonctionnalités très riches mais de bas niveau. Nous cherchons à l'inverse à mettre en œuvre une méthode de création musicale assez précise et de haut niveau. Nous verrons aussi que notre projet, étant plus récent, repose sur JavaScript en matière de serveur et non Java.

***echobo*** [52] propose aussi d'utiliser les smartphones comme instrument de musique. C'est bien le haut-parleur du smartphone qui produit les sons. *Echobo* repose sur le principe d'un chef d'orchestre qui définit la structure musicale à haut-niveau (progression d'accord, échelle). A partir de cette structure l'audience peut jouer note à note, tout en étant globalement alignée harmoniquement. Nous sommes assez loin de notre projet musical, mais des similitudes existent notamment

sur le principe du chef d'orchestre. Notons une différence majeure sur la production du son. Nous verrons que nous n'utilisons pas les haut-parleurs des smartphones pour produire la musique, mais un système central. Au-delà de la solution, cet article est intéressant car il pose un certain nombre de principes à respecter pour qu'une performance interactive se déroule bien. 1) rendre la participation facile (accessibilité). 2) capter les actions de l'audience pour en faire une composition musicale (cohérence musicale). 3) permettre la participation sans avoir besoin de s'inscrire préalablement. 4) motiver l'audience à participer et maintenir la motivation. 5) donner à tous une vision claire de l'impact d'une action sur le résultat musical. Nous avons eu à nous confronter à ces principes lors de nos travaux sur l'interaction.

Le projet ***a.bel*** [15] propose d'utiliser le smartphone comme un instrument de musique piloté depuis un serveur. La synthèse sur le smartphone est réalisée à l'aide du logiciel PureData [81]. L'interface du smartphone se veut très simple, elle utilise des fonctionnalités comme l'accéléromètre. Le système a été utilisé à grande échelle par 700 personnes au Portugal en octobre 2015. Nous sommes ici dans une configuration proche d'*echobo* dans la mesure où le smartphone produit du son, mais aussi proche de *massMobile* en termes de création musicale car *PureData* et *MAX/MSP* ont la même origine et les mêmes fonctionnalités.

***Open Band*** [84] est un système collaboratif original qui propose d'utiliser l'alphabet pour produire de la musique expérimentale. Chaque participant peut produire un message qui est chargé dans un serveur pour être traduit dans une séquence de sons jouée sur le smartphone. Un contrôleur peut agir sur la musique produite en jouant sur le volume ou les fréquences audio par exemple. La musique produite est créée en associant un échantillon à chaque caractère ASCII ou à l'aide de synthèses produites avec des API Web Audio. Ce système possède quelques points communs avec le nôtre. Il propose une méthode de production de la musique et non pas un accès à un outil de bas niveau comme le fait *massMobile*. Il intègre aussi une notion de contrôleur. Comme dans le cas d'*echobo* le son est produit par les smartphones, contrairement à notre environnement. Par ailleurs, le principe de production de la musique à partir d'un alphabet est loin de nos préoccupations sur le lien entre langage informatique et composition.

***Performance Without Borders*** et ***Embodied iSound*** [31] sont deux exemples d'installations produites au *Contemporary Music Festival* de l'université de Plymouth en 2016. *Performance Without Borders* est basée sur une forme de dialogue entre un système central utilisant *MAX/MSP* et l'audience via un mécanisme de vote, dans ce sens il est assez proche de *massMobile*. « *Embodied iSound* » utilise des informations statistiques produites par des terminaux iOS de l'audience pour contrôler le son produit par un système quadriphonique. Bien que produite dans le même contexte les deux solutions sont très différentes. *Performance Without Borders* se situe dans la même mouvance que *massMobile* en termes d'architecture, même si la façon de gérer les votes est différente.

***Swarmed*** [41] est un projet de 2013. Il est intéressant à titre historique. Il met un fort accent sur l'architecture technique basée sur des technologies web et un réseau privatif Wifi. *Swarmed* fait tourner de façon concurrente 7 instruments centralisés qui peuvent être contrôlés par l'audience. Le son est produit par le logiciel *Csound* [51]. Nous avons utilisé une architecture proche de celle de *Swarmed.* Nous retrouvons un serveur central qui contrôle un ensemble d'instruments utilisant des clients HTML. Contrairement à notre plateforme expérimentale, *Swarmed* ne propose pas de logique de composition. Il fournit une suite d'instruments dont l'utilisation par l'audience est totalement libre.

Même s’ils n’adressent pas totalement nos problématiques, un certain nombre d’outils collaboratifs du marché méritent l’attention comme *Ableton Link* et des logiciels en ligne comme *Soundtrap* [55], *PiBox* [95], *Flat* et *TrackD* [96]. *Ableton Link* [2] permet de synchroniser plusieurs ordinateurs exécutant le programme *Ableton Live*. L’idée est donc bien de permettre à un groupe d’utilisateurs de produire une musique collectivement tout en ayant la même base de temps. Si nous sommes assez loin d’un contexte de musique collaborative pour une vaste audience, ces outils mettent en œuvre des solutions qui se rapprochent de nos travaux sur les interfaces homme-machine et plus précisément un élément de notre plateforme expérimentale, le séquenceur distribué. Cet outil propose une solution de création synchronisée d’éléments musicaux en petit groupe. Une autre différence majeure entre notre solution et *Ableton Link* est l’usage de micro-ordinateurs pour *Ableton* et de smartphones pour notre séquenceur distribué. Pour ce qui est des outils en ligne comme *Soundtrap*, nous avons affaire à des solutions de type *Digital Audio Workstation* (DAW, station de travail audio), distribuée. Ces solutions ne sont pas initialement destinées à produire de la musique collaborative Live, néanmoins rien n’empêche de les utiliser dans ce contexte.

Pour conclure ce chapitre, voici dans les grandes lignes une trame d’analyse proposée par l’article sur *Open Symphony* [93]. Cette trame a pour mérite de formaliser les axes de développement que peut envisager tout développeur de solution de musique collaborative. Nous l’utilisons pour positionner les travaux que nous avons sélectionnés.

- *Motivation de la participation créative de l'auditoire* : par exemple imitative (suiveur), compétitive, contributive, direction/chef d'orchestre. Les notions compétitive et contributive font partie de nos objectifs et de la plupart des projets évoqués : *Open Symphony, massMobile, echobo, a.bel, Frontier*. Nous ne retrouvons pas dans cette rubrique les projets à dominante technologique comme *Swarmed* par exemple.

- *Répartition de la création* : du niveau individuel au niveau collectif, comme les sous-groupes d'audience ou l'audience entière. Nous verrons que nous chercherons à proposer plusieurs scénarios qui peuvent aller de la cocréation en petits groupes à l’exécution d’une œuvre créée par un compositeur. *Open Symphony* entre aussi dans cette catégorie, ainsi que *massMobile* et Frontier.

- *Médiation de la création* : elle peut aller *d’indirecte*, comme dans le cas *Open Symphony*, où les interprètes suivent les instructions du public et créent les sons, jusqu’à la réalisation *sans médiation* où le public crée lui-même les sons comme dans les cas d’*echobo, a.bel* et *Open Band*. Bien qu’il ne s’agisse pas d’une plateforme de création collaborative, notons ici le travail réalisé par le *GRAME* [97] avec *INscore* [28] qui est un outil de notation complet et dynamique qui permet le contrôle en temps réel de la partition. Nos travaux auront pour objectifs d’être ouvert à des médiations indirecte et directe.

- *Niveau de création* : il dépend de la façon dont les participants agissent sur la performance. Par exemple, un public peut contribuer en modifiant l’exécution d’un projet musical défini, ce qui sera le plus souvent notre cas, ou cocréent complétement une performance ce que nous retrouvons dans les projets *Open Symphony, massMobile* et *echobo*.

- *Contraintes sur la création* : Il peut s’agir de choix sur un ensemble de propositions, comme c’est le cas pour nous lorsque le public active une « partition » conçue par un compositeur, comme dans *Open Symphony* où le public a le choix entre certaines options. Il peut aussi s’agir d’une liberté totale lorsque le public a le contrôle, ce qui est le cas des projets *Frontier, Open Band* et *a.bel.*

- *Modalités de participation créative* : parmi les modalités nous avons l'audition, la vision, ou la combinaison des deux comme le propose *Open Symphony*, où les actions du public influencent les visuels projetés et les sons produits par les interprètes. Nous avons aussi les cas où les membres du public produisent des sons et visuels avec leurs téléphones portables, ce que nous retrouvons dans le projet *echobo*, ou du texte dans le projet Open Band. Nous verrons que notre travail autour de l'interaction mettra en œuvre des visuels sous le contrôle de l'audience.

- *Moyen de participation créative :* Ce sera par exemple le corps et le mouvement, comme les gestes des mains et des bras dans le cas du projet *a.bel* par exemple. Nous aurons aussi des mécanismes de votes dans les projets *Open Symphony, massMobile* et *Frontier*, ou bien des expressions linguistiques comme dans *Open Band*.

- *Interfaces utilisateur* : Ce sont des interfaces Web dans notre cas, ou pour les projets *Open Symphony* et *Swarmed*. Ce peut être aussi des capteurs, comme pour le projet *a.bel*.

- *Localisation* : Il s'agit de l'organisation de l'audience. Elle pourra être colocalisée. Ce sera notre cas, où l'audience et les interprètes partagent physiquement le même espace. Elle pourra être dispersée géographiquement comme pour les solutions en ligne *Soundtrap*, *PiBox*, Flat et *TackD.*

- *Évolutivité* : Il s'agit de la capacité de la solution d'adresser de petits groupes, ce qui est cas de Frontier, et de passer à des auditoires importants ce que permettent *massMobile* ou *Open Symphony* et notre plateforme.

- *Structure à long terme* : Nous ajoutons cet axe à la série proposée par l'article sur *Open Symphony*. C'est un axe que nous estimons majeur et qui rejoint l'avertissement d'U. Eco quant au contrôle de la qualité de l'œuvre produite. Ce que l'on peut conclure de cette revue est que les développements autour des musiques interactives prennent assez peu en compte, voire pas du tout dans la liste des projets cités précédemment, la structure à long terme de l'œuvre musicale. Elles se concentrent le plus souvent sur des mécanismes d'improvisation collective assez peu contrôlés ou de production de musiques expérimentales. Nous verrons que notre démarche se démarque de ces développements en introduisant dans la musique interactive un discours musical structuré lié à l'expressivité d'un langage informatique particulier.

### 2.1.3 Les technologies du Web appliquées à la musique

Le choix des musiques interactives comme mise en œuvre de la durée nous conduit naturellement à créer des outils pour mettre en œuvre l'interaction. Les technologies du Web sont celles qui s'imposent de façon naturelle aujourd'hui. Il existe un grand nombre d'outils et de « frameworks » pour les développements d'applications Web. Nous avons utilisé *Hop.js* [79], qui est un environnement de programmation « multitier » pour JavaScript, développé à l'INRIA, compatible avec Node.js [69], conforme à l'implémentation JavaScript *EcmaScript 5.1* [22] et aux fonctionnalités *EcmaScript 6* [114]. Hop.js permet le développement de programmes comprenant à la fois le côté client et le côté serveur dans le même fichier source, ce qui permet de garantir la cohérence des exécutions de l'application sur le serveur et du côté client. Par ailleurs Hop.js possède plusieurs fonctionnalités qui facilitent le développement d'applications complexes. La simplicité avec laquelle se mettent en place les communications par *broadcast* en est une.

Passer en revue les technologies du Web est un chantier hors de propos ici, nous nous limiterons à aborder le thème de la musique. Le Web est devenu le média de diffusion de la musique. En quelques années l'industrie du disque s'est vue dépassée par les *Youtube*, *Spotify*, *Qobuz*, *SoundCloud*, *Deezer* et autres *Tidal*, *GrooveShark*. La Radio s'est appropriée les podcasts en créant le

puissant concept de radio à la demande. Les webradios ont connu un développement explosif jusqu'à ce que les sociétés de production réagissent en 2007. Les activités du web autour de la musique ne se limitent pas à la diffusion. Nous avons évoqué dans le chapitre sur les musiques interactives quelques outils de conception en ligne comme *Soundtrap* [98], *PiBox* [95], *Flat* [99] ou *TrackD* [96]. Nous pouvons ajouter à la liste des solutions utilisant le Web pour la musique, les studios en ligne comme *Audiotool* [100], *Audio Sauna* [101], *Soundation* [102], *Jam Studio* [103], *Ujam* [104]. Parmi les initiatives issues du Web, citons le projet *Magenta* de *Google* [33] dont le but est d'explorer les utilisations du « Machine Learning », avec la plateforme *Tensorflow* [34], pour la création artistique et musicale.

Parmi les initiatives techniques directement liées à la création musicale arrêtons-nous sur les *Web Audio API* portées par le *W3C* [105] (*World Wide Web Consortium*) dont le but est de définir et de promouvoir la compatibilité des technologies du Web. Les Web Audio API sont issues du constat que l'audio dans les navigateurs était encore récemment géré de façon non standard via des plugins *Flash* ou *QuickTime*. Il a donc été proposé d'intégrer des éléments audios dans la définition de HTML5 initialement pour lire de la musique en *streaming*. Cette première étape s'est cependant révélée insuffisante pour la mise en œuvre d'applications musicales plus complexes comme des jeux interactifs par exemple. Il a donc été décidé d'aller plus loin en introduisant des processus de mixage et de traitement du signal.

Le W3C ne propose pas des technologies mais des standards, les API ont donc été conçues de façon à être mises en œuvre par un moteur codé en C++ par exemple qui serait contrôlé par un langage comme JavaScript dans un navigateur. A ce jour les Web Audio API ne sont pas encore capables de fournir des fonctions aussi performantes que celles que l'on trouvera dans des logiciels de MAO comme *Reaper* [106] *ou Ableton Live* [2] (comme l'écriture optimisée *direct-to-disk* , la gestion des interfaces MIDI, le support des instruments VST…). Le W3C cependant promeut une architecture qui est capable d'évoluer vers des fonctions plus complexes que celles disponibles aujourd'hui.

Les Web Audio API sont conçues pour apporter les fonctions de base suivantes [82] :

- Routage modulaire pour la mise en œuvre d'architecture de mixage complexe.
- Traitements du son en 32 bits flottants.
- Lecture des sons avec une précision au niveau de l'échantillon avec une latence faible dans le but de fournir une précision rythmique nécessaire aux boites à rythme et séquenceurs.
- Automatisation des paramètres d'enveloppe, fade-in, fade-out, effet granulaires…
- Gestion de canaux dans les flux audio, pour les diviser ou les fusionner.
- Traitement de sources audio enregistrées ou live.
- Intégration des WebRTC (pour les communications en temps réel comme la voix sur IP).
- Synthèse de flux audio.
- Spatialisation de l'audio.
- Moteur de convolution pour des effets de réverbération par exemple.
- Compression dynamique.
- Différents filtres passe-bas, passe-haut…
- Effet de distorsion
- Fourniture d'oscillateurs

Ces éléments constituent déjà une liste de fonctions élémentaires permettant des développements riches. Notons cependant que les fonctions offertes par ces API sont de bas niveau. Elles sont destinées à des programmeurs avertis et ne sont pas accessibles à des musiciens non informaticiens.

Les travaux du W3C autour des Web Audio conférence sont accompagnés d'une conférence annuelle la *Web Audio Conference* dont la 5ème session en 2019 se tient à Trondheim en Norvège. Ces conférences sont l'occasion d'exposer différents types de mise en œuvre des Web Audio API. En 2016 la conférence WAC s'est tenue au Georgia Institute of Technology. Cette université a mis en ligne les 70 publications issues de la conférence [30] qui montre la diversité des applications. Pour faire court, nous retrouvons quasiment tous les grands thèmes de la MAO (Musique Assistée par Ordinateur), la conception de synthétiseur, de séquenceur MIDI, le « Live Coding », différents traitements du signal et effets, l'édition de partition, la pédagogie, la musique collaborative, la programmation visuelle, la micro-tonalité, etc. Même si aujourd'hui ces applications n'ont pas atteint la maturité des outils de MAO du marché, on peut imaginer que le navigateur Web puisse devenir une plateforme universelle pour la musique au même titre qu'il est devenu une plateforme universelle pour beaucoup d'applications métiers.

Les API Web Audio sont importants pour notre propos à plusieurs niveaux. Il s'agit d'un environnement dynamique dont émerge un grand nombre d'initiatives qui traitent d'aspects particuliers propres aux musiques interactives. Le projet COSIMA de l'IRCAM utilise des API Web Audio, le projet FAUST possède une traduction de ses programmes en Web Audio (Faust2WebAUdio). La veille technologique dans ce domaine est nécessaire. En ce qui concerne notre contribution au développement de solutions autour du Web, nous nous sommes concentrés sur la conception d'interfaces pour l'audience. La plupart de nos interfaces ont pour objectif de rendre la participation de l'audience la plus directe et la plus simple possible, à l'exception d'une : le séquenceur distribué, qui fait l'objet d'une section à part dans le chapitre sur l'interaction. Cette interface soulève des problèmes de synchronisation que nous avons traités en faisant appel aux horloges Web Audio.

## 2.2 L'expression du temps en musique

Notre problématique relève de l'expression du temps et de la durée en musique. Nous avons abordé l'expressivité des outils informatiques au sens large au début de ce chapitre sur l'état de l'art. Nous allons aborder à présent plus précisément les principaux moyens *d'expression du temps et de la durée* en musique à travers : la partition musicale et ses évolutions, quelques exemples représentatifs de MAO, les musiques improvisées, les musiques aléatoires, la vision particulière de Iannis Xenakis qui s'est exprimé sur ce sujet et les musiques interactives.

### 2.2.1 La partition et son évolution

Comme le cinéma la musique se perçoit dans le temps, ce qui n'est pas le cas de la peinture ou de la sculpture. La musique occidentale a défini un système de notation plus ou moins précis, sur un axe temporel duquel dépendent des évènements sonores (hauteur, timbre, intensité, articulation). Nous n'avons pas de date précise concernant les premières écritures musicales. Ce que l'on sait mieux dater est la genèse de la notation musicale telle que nous la connaissons en occident. Il est communément admis de considérer que les premiers spécimens de ce qui donneront nos partitions remontent au IXème siècle. Ce sont ce que l'on nomme les neumes, terme emprunté au grec ancien νεῦμα / neûma qui veut dire *signe*. Les premières versions de ce système de notation consistent en un ensemble de signes placés au-dessus du texte. La notation n'est pas précise, il s'agit plus d'un aide-mémoire que d'un outil de lecture. Un exemple est donné en Figure 1 : Extrait de l'Anthiphonarium offici n°390 (source wikipedia).

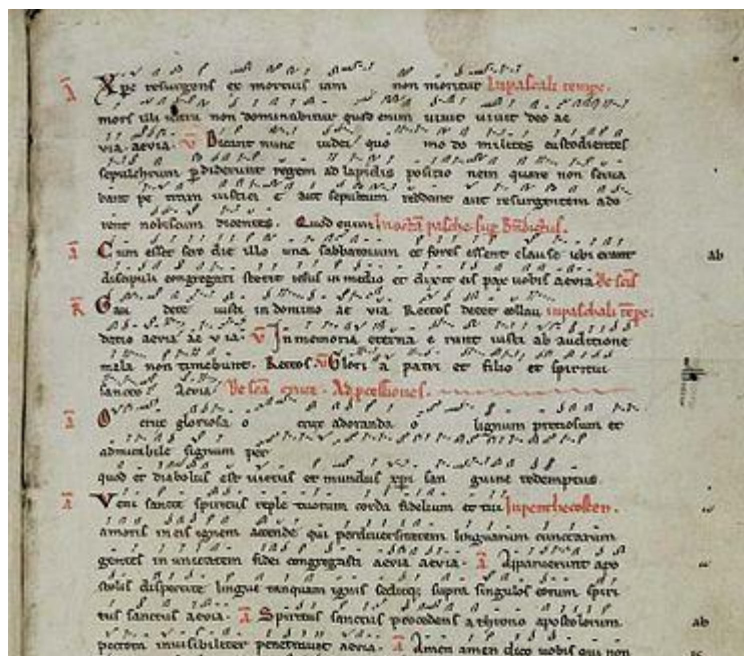


**Figure 1 : Extrait de l'Anthiphonarium offici n°390 (source wikipedia).**

Ces signes vont évoluer pour définir des formules mélodiques simples et aboutir ainsi au XIème siècle à un système solfégique formalisé par Guy d'Arezzo vers 1030, avec la portée à quatre lignes. Ce système de notation va encore évoluer jusqu'à la renaissance vers la représentation en portées que nous avons aujourd'hui. De cette genèse de la notation musicale, nous pouvons retenir l'association initiale faite entre des textes et des mélodies. La notation musicale n'apparait que comme un corollaire de l'écriture, elle en épouse la linéarité. Les éditeurs de partition récents de type *Finale*, *Sibélius* ou *Dorico,* reposent sur une représentation graphique conforme à l'écriture de partitions la plus traditionnelle. Ils n'impactent pas en profondeur la façon dont un compositeur écrit ses partitions sur papier de façon linéaire. L'expression du temps reste spatiale selon une lecture de gauche à droite des portées avec des symboles dédiés à des renvois divers. De la même façon, des DAW se composant d'un éditeur MIDI et d'un mixeur de pistes MIDI ou sonore (*Cubase, Reaper, Cakewalk*…) vont permettre de travailler plus près du son qu'un éditeur de partition, mais ces outils de composition utilisent une représentation temporelle d'événements sonores qui reste dans la lignée d'une partition. Dans les deux cas, la méthode de composition sera donc descriptive selon l'axe du temps, même si cet axe n'est pas rigide.

Dans cette lignée d'outils, *Antescofo* [19] créé à l'IRCAM est une solution originale qui se situe dans la continuité des représentations temporelles de la musique tout en assouplissant la dépendance au métronome, puisque c'est le musicien qui contrôle le déroulement. Antescofo est un environnement complet qui utilise MAX/MSP et qui propose un langage de programmation permettant d'associer des événements au déroulement séquentiel de la musique. Les réflexions à la base d'Antescofo ont plusieurs points communs avec les nôtres. Il s'agit d'une solution de musique interactive, et donc d'une forme d'introduction de la durée dans la MAO. Par ailleurs le langage de description qu'Antescofo associe aux partitions utilise des concepts de programmation proches de ceux que nous utilisons (programmation réactive synchrone). Il y a cependant des différences majeures dans nos approches du temps et de la durée. Antescofo s'inscrit dans la continuité directe de la partition « temporelle ». L'outil *AscoGraph* d'aide à la composition associé à Antescofo, a la forme d'un éditeur de partition. Contrairement à Antescofo nous verrons que nous n'exprimons

pas la durée en fonction d'un musicien, mais en fonction d'une audience et nous ne formalisons pas nécessairement la partition avec un éditeur de partition. Par ailleurs, Antescofo ne propose pas une méthode de composition particulière, alors que notre réflexion sur l'expression de la durée nous a conduit à poser des principes élémentaires de composition musicale.

### 2.2.2 Des exemples d'expression du temps en MAO

L'axe du temps mesuré, même imprécisément, est constitutif de notre façon d'écrire la musique depuis plus de mille ans. On imagine aisément qu'il n'est pas facile de changer un modèle ancré si profondément. Cependant dans la mouvance des mouvements d'avant-garde d'après la deuxième guerre mondiale et de l'apparition des ordinateurs, les systèmes de notations vont connaitre des développements aussi rapides que variés. L'article réalisé pour le groupe de travail *AFIM* (Association Francophone d'Informatique Musicale) en 2015 sous l'égide du *GRAME*, de *l'Ircam*, de *l'INA* et de *l'Université Paris-Sorbonne* donne un bon aperçu de ces développements [27]. Nous avons d'un côté l'automatisation de *l'édition et de la gravure musicale*, depuis le langage *DARMS* conçu dans les années 60, jusqu'aux outils d'édition musicale sur le Web. D'un autre côté, nous avons les outils dédiés à la MAO*,* mais pas spécifiquement à l'édition, qui incluent des solutions aussi variées que de *l'UPIC* [56] conçu dans les années 70/80 avec une représentation graphique du son, *Open Music* de *l'Ircam* [3], *MAX/MSP* [71] ou *Ableton Live* [2]. Ces outils comportent des outils de notation musicale qui ne relèvent pas, à proprement parler, de l'édition. Voyons quatre exemples fort différents, d'expression du temps en MAO.

Tout d'abord les outils de programmation *MAX/MSP* [71], *PureData* [81], où l'expressivité du temps fait l'objet de primitives dédiées (`timer`, `clocker`, `delay`, `metro` pour MAX/MSP par exemple) mais peut aussi exprimer la durée à travers la gestion d'événements (`bang`, `transport`, `when`, `timepoint` toujours pour MAX/MSP). Ces langages sont en théorie aptes à exprimer la durée sous forme des formes variées. Mais ils soulèvent les mêmes difficultés que des langages généralistes textuels. De plus, leur mise en œuvre est difficile. La courbe d'apprentissage de ces langages et de leurs environnements est longue car ils sont davantage basés sur l'accumulation de fonctionnalités que sur une réflexion de fond autour d'une syntaxe et d'une sémantique, ce qui ne les rend peu performant pour l'expression du temps et de la durée.

Le cas d'*Open Music* de l'IRCAM est différent. En proposant une boite à outils de fonctions algorithmiques au moyen d'un langage graphique, cet outil est destiné aux compositeurs qui veulent traiter des processus pouvant s'exprimer sous forme de calcul. Cet outil est donc fortement corrélé à des méthodes de composition qui utilisent des formes qui s'expriment au moyen d'algorithmes. Open Music possède des outils permettant d'exprimer le temps des horloges, avec l'utilisation d'un séquenceur. Mais il ne possède pas d'outil pour exprimer la durée, c'est-à-dire une interaction en temps réel avec un ou plusieurs individus.

Un autre exemple d'expression du temps en MAO est la solution *Ableton Live*, évoquée plus haut. Cet outil d'écriture, mais aussi d'improvisation sur scène, propose une méthode de composition graphique à base de « clips ». Cet outil intègre nativement des représentations assez simples du temps et d'une certaine façon de la durée. L'expression du temps se trouve à plusieurs niveaux, sous la forme :

- D'une pulsation globale, d'un métronome qui contrôle l'ensemble du système. Cette pulsation, sera contrôlable par des équipement MIDI ou programmée sous une forme graphique ;

- De différents paramètres associés à chaque clip. Paramètres qui sont basés sur des multiples ou divisions de la pulsation globale.
- D'automates relativement simples permettant d'automatiser l'enchainement de clips pour une piste, selon des critères sur le nombre de pulsations (il s'agit de la fonction « Follow Action » associée à un clip).

La mise en œuvre de la durée se fait grâce aux différentes fonctions spécialisées que propose l'interface à un musicien contrôlant le logiciel sur scène. Nous pouvons parler ici de représentation de la durée dans la mesure où ces contrôles définissent la façon dont le musicien va pouvoir agir. Même si le champ d'action est large, il n'est pas infini et certaines actions sont plus immédiates à mettre en œuvre que d'autres. Les modes d'expression de la durée induits par l'interface de contrôle auront donc forcément un impact sur le projet esthétique du musicien.

Le *Live Coding* [18] est un moyen d'associer musique algorithmique et image à improvisation. C'est une discipline issue du monde de *l'électro* dans les années 2000 qui connait un certain développement dans les milieux de la musique expérimentale, notamment autour de *l'Organisation Temporaire pour la Promotion de la Programmation Algorithmique* en Live créé en 2004. En ce qui concerne l'expression du temps, le Live Coding met en œuvre des langages interprétés plutôt destinés à des improvisations. Ceci signifie que le codage du temps et de la durée dans ces langages n'est pas une question fondamentale. La durée s'exprime dans l'acte même de programmation, puisqu'il est le produit vivant de la conscience du programmeur. Il existe plusieurs plateformes permettant cette forme de création musicale (*SuperCollider* [57], *Chuck* [89], *Sonic Pi*, etc.). *SuperCollider* est l'une des plus représentatives. Dans le même registre citons *Lissajous,* qui est une librairie JavaScript légère et qui permet de coder en direct à l'aide de la console d'un navigateur Web. Elle a fait l'objet d'une démonstration lors de la première édition de la conférence Web Audio [82] qui s'est tenue à l'*Ircam* en 2015. *Lissajous* peut bénéficier de l'utilisation d'un langage réactif tel que HipHop.js pour composer des constructions musicales élaborées. Il est similaire à *NNdef* [61] qui utilise la programmation réactive pour contrôler les instruments virtuels à l'aide de données issues de contrôleurs, d'applications mobiles ou de widgets en format graphique. Outre les différences techniques entre ces systèmes et le nôtre, notons que notre objectif est un outil de contrôle par une audience, de musiques qui doivent être produites par un ordinateur ou des musiciens, tandis que les plateformes de Live Coding sont des outils de synthèse du son par un programmeur. Ceci signifie que nos travaux ne s'inscrivent pas dans le contexte du Live Coding, bien que nous puissions utiliser des technologies de Live Coding pour la synthèse du son dans nos travaux.

### 2.2.3 Les musiques improvisées

Il n'est pas possible de regrouper ici l'ensemble des expressions du temps dans les musiques improvisées. Nous nous focaliserons sur des genres proches de nous comme le Jazz et les musiques moyen-orientales.

Le Jazz qui est un univers riche, sans borne précise, qui regroupe des pratiques collectives d'improvisation à base de grilles d'accords, ou des formes plus expérimentales comme le free-jazz qui évite tout repère d'échelle, d'accord et de pulsation rythmique. L'histoire du Jazz est en grande partie liée à la danse. L'élément le plus caractéristique de cette origine reste la quasi-universalité de la pulsation, à l'exception du free-jazz. Le temps en Jazz est donc structurellement soumis à un découpage régulier du temps. Un bon maintien de la pulsation fait partie des prérequis indispensables à une bonne section rythmique. On pourrait dire que la notion de temps mesuré est un élément constitutif du Jazz. De la même façon que la pulsation est rigoureusement maintenue, l'élément fédérateur des improvisations que sont les grilles d'accords est lui aussi soumis au temps mesuré.

Le Jazz pourrait donc être perçu dans sa structure temporelle comme extrêmement figé. Mais son ouverture se situe ailleurs, dans l'improvisation mélodique. En effet, l'improvisation est largement soumise à la durée au sens de Bergson. Son déroulement est sous le contrôle de la conscience non seulement de l'improvisateur mais aussi de celle du groupe qui soutien et participe à l'improvisation. La façon de manipuler les durées des notes, donc du rythme de la mélodie, est un processus qui ne s'exprime pas totalement selon un temps mesuré. Précisons bien « pas totalement » car la plupart des improvisations en Jazz sont constituées d'éléments de base que le musicien sollicite au gré de son humeur. Mais c'est la conscience du musicien qui décide de la mélodie, le temps qui la conditionne relève d'un acte de création selon un temps qui relève de la durée bergsonienne et non d'un temps mesuré. Il n'y a pas acte d'écriture et de représentation. La mélodie ne se construit pas sur un schéma décrit selon un axe du temps.

L'expression du temps suit un schéma du même ordre dans les musiques du Moyen-Orient. La différence majeure avec le Jazz, dans le cas des musiques du Moyen-Orient qui n'ont pas subi l'influence de la tonalité et l'harmonie occidentale, est la prédominance d'échelles modales et de motifs rythmiques. Bien que ce ne soit pas une grille d'accords qui ordonne le déroulement du temps, il peut être soumis au décompte des pulsations qui nous ramène sur le même terrain que le déroulement d'une grille. Mais d'autres scénarios sont possibles, comme la libre décision d'un musicien de changer de motif rythmique ou de mode. On pourrait dire que la liberté sous-jacente au système modal nous rapproche encore plus de la durée bergsonienne que ce que permet le Jazz des grilles d'accords. Notons que les Jazzmen ont repris en partie à leur compte ces principes orientaux avec ce que l'on nomme le *Jazz Modal*.

### 2.2.4 L'aléatoire dans la musique

L'introduction de phénomènes aléatoires ou indéterminés dans la conception et l'exécution de la musique remonterait au XVème siècle avec Johannes Ockberghem. On attribue aussi à Mozart un jeu de dés musical, qui permettait de combiner au hasard des éléments pour réaliser une pièce de musique. La musique aléatoire, devint même un courant de la musique occidentale de l'après deuxième guerre mondiale. Les principaux porteurs de ce mouvement sont John Cage et Earl Brown [80]. L'introduction du hasard dans la musique peut se faire de façons différentes. Il peut s'agir de dérouler au hasard des séquences musicales prédéfinies. Il peut s'agir aussi d'introduire des phénomènes probabilistes dans différentes phases de la conception d'une œuvre musicale comme l'a fait Iannis Xenakis [42] avec sa musique stochastique ou et comme le fait Philippe Manoury avec l'utilisation des chaines de Markov. Qu'en est-il de l'expression du temps dans des musiques faisant appel à l'aléatoire ? La question n'est pas simple a priori. En effet, on ne peut pas dire que l'expression du temps soit totalement mesurée si l'aléa peut influer le déroulement de l'œuvre. D'un autre côté il ne s'agit pas non plus de durée, puisque le mécanisme aléatoire n'est, par définition, pas conscient. Ceci signifie tout simplement qu'en introduisant de l'aléatoire dans une production musicale, le compositeur introduit une non-expression du temps, que ce soit le temps mesuré ou la durée. Ceci n'est d'ailleurs pas totalement contradictoire avec la quête d'une forme d'absolu au-delà du temps.

### 2.2.5 En-temps, hors-temps

Revenons à Xenakis qui est souvent cité dans les réflexions autour du temps en musique. Il propose trois notions relatives au temps : en-temps, hors-temps et temporalisation. Les architectures hors-temps sont les matériaux de base de la phase de conception de la musique au sens de Xenakis. Une gamme est une architecture hors-temps par exemple. Pour Xenakis ce sont les processus basés sur des calculs probabilistes et des théories mathématiques. La traduction de ces processus en partition

revient à produire de la musique en-temps. La temporalisation concerne la production de la musique en-temps à partir de la musique hors-temps. La distinction hors-temps/en-temps revient à mettre un accent particulier sur les processus utilisés pour la conception des matériaux de base de la musique. Les recherches et les travaux sur la micro-tonalité relèvent aussi de développements d'éléments hors-temps par exemple. Voici un extrait de l'article de Xenakis « Vers une Métamusique » [94] qui donne un assez bon résumé de sa position vis-à-vis du temps.

*« D'ailleurs, grâce au caractère métrique du temps, on peut le « munir » lui aussi d'une structure hors-temps, laissant finalement à la catégorie temporelle seule sa vraie nature, toute nue, celle de la réalité immédiate, du devenir instantané. Par-là, le temps pourrait être considéré comme un tableau noir (vide) sur lequel on inscrit des symboles et des relations, des architectures, des organismes abstraits. Du choc entre organismes-architectures et réalité instantanée, immédiate, naît la qualité primordiale de la conscience vécue. »*

Ce court extrait est assez révélateur de la distance qui sépare Xenakis de Bergson. Le temps pour Xenakis n'est envisagé que sous un *angle métrique*, donc du temps des horloges qui traduit dans l'instant une trame écrite. La dimension intuitive que Bergson attribue à la durée s'exprime chez Xenakis sous la forme d'un choc entre « organismes-architecture et une réalité instantanée ». Il n'est pas immédiat de faire la relation entre l'intuition selon Bergson et la métaphore de Xenakis dont les référents philosophiques se situent plutôt dans l'antiquité grecques avec Aristoxène et Pythagore.

### 2.2.6 Les musiques interactives

La présentation des musiques interactives ou collaboratives, celles qui concernent l'interaction avec une audience, occupe une place importante dans cette thèse. Elles ont donc fait l'objet d'un chapitre à part. Ce que nous traitons ici est leur rapport à l'expression du temps. Il est impossible d'avoir une vision complète de ce que se fait dans le domaine des musiques interactives. Le domaine a connu une expansion trop importante depuis l'arrivée des smartphones. Sur le périmètre que nous connaissons, on notera la dominante des musiques improvisées et expérimentales comme nous l'avons écrit dans le chapitre précédent. Les musiques interactives introduisent naturellement des formes de durées qui se rapprochent de la définition que Bergson propose de ce terme. La scénarisation, quand elle repose sur des mécanismes de vote par exemple, ce qui est un cas assez fréquent, est un processus ou la durée peut facilement primer sur le temps des horloges. On peut décider de déclencher une action, au bout de X votes et non pas au bout de Y secondes par exemple. Le temps de l'interaction devient le concurrent naturel du temps des horloges. De plus, le temps de l'interaction devient un sur-ensemble du temps des durées de chaque individu participant à la performance. Au mécanisme de comptage des horloges se substitue un autre moteur du temps, qui est celui de la conscience collective de l'audience. Nous voyons donc que dès que l'on se pose la question de musique sous l'angle de la durée, la musique interactive devient un candidat naturel pour la création.

On peut se poser la question de savoir quelles sont les incidences esthétiques issues de la dépendance du temps musical vis-à-vis de la conscience collective. Il s'agit là d'un sujet complexe, car sa mise en œuvre est difficile à la fois en termes d'écriture musicale (si l'on souhaite ne pas être dans un contexte de musique improvisée), mais aussi de mise en œuvre technique. Penser la musique sous l'angle de la durée nécessite des outils particuliers à la fois de conception mais aussi d'exécution. Les systèmes de notations issus de la tradition des neumes ne savent pas exprimer le temps sous l'angle d'une conscience collective. Nous allons à présent entrer plus dans le détail d'un

élément de base complémentaire des systèmes de notation et qui existe en MAO. Nous avons retenu cet élément pour la conception et l'exécution de nos performances interactives.

## 2.3 Les méthodes de composition utilisant des patterns

La durée s'exprime par des transitions. Plutôt que de dire qu'à « tel instant je dois faire ceci », nous dirons plutôt « quand ceci se produira, je ferais ceci. » Une transition suppose un changement d'état. Pour penser en durée, il faut donc pouvoir faire référence à des états. Dans la méthode d'écriture issue que nous proposons de développer, les états sont ce que nous nommerons des *patterns musicaux* ou plus simplement des *patterns*.

Les patterns sont les éléments musicaux de base de notre mise en œuvre de la durée. Il s'agit de phrases musicales ou de parties de phrases musicales dont les combinaisons en temps réel donnent naissance à la performance et sur lesquelles, comme nous le verrons par la suite, reposent les mécanismes d'interaction homme-machine. Les transitions se font dans la durée, les patterns sont les états qui eux se déroulent selon le temps des horloges. L'utilisation de patterns dans le domaine de la composition musicale peut être envisagée de différentes façons : celle de l'analyse musicale [45] et de la musique sérielle [53], celle des clips et celle du Jazz. Commençons par le point de vue de l'analyse musicale.

### 2.3.1 Thème, motif, cellule

L'analyse musicale consiste à étudier la structure, le contexte, les objectifs d'une œuvre musicale. Elle repose sur le paradigme posé à la fin du XIXème siècle qui considère que l'œuvre écrite est le cœur de toutes activité musicale. Elle a développé tout un arsenal théorique permettant de décortiquer la forme d'une œuvre. Parmi ceux-ci se trouvent les thèmes, les motifs et les cellules. L'idée de base est de décomposer la pièce de musique en éléments de plus en plus courts depuis la forme globale (sonate, variations, strophique...) aux éléments les plus simples, les cellules, qui sont les plus petits éléments rythmiques, mélodiques ou harmoniques que l'on peut isoler. Le sur-ensemble de la cellule est le motif. Le sur-ensemble du motif est le thème. Nous avons donc une vision structuraliste de l'œuvre musicale. Ce qui intéresse l'analyse musicale est donc à la fois la dynamique de l'œuvre (ses transitions) et ses formes (ses états). Même si le compositeur ne pense pas forcément son œuvre comme l'instance d'une forme. On voit qu'une œuvre musicale met en œuvre une forme de combinatoire d'éléments simples. Le terme *pattern* pourrait ainsi se décliner en thème, motif ou cellule selon le point de vue ou l'on se place dans la structure de l'œuvre.

Laurie Spiegel, compositrice américaine née et 1945, s'est posée dans les années 1980 la question de la structure musicale d'une façon proche de la nôtre. Elle propose de réfléchir à la forme de la musique électronique en partant de la notion de pattern. Dans son article *Manipulations of Musical Patterns* [83] elle propose de partir de la notion de pattern comme extension des notions que nous avons évoquées comme étant issues de l'analyse musicale. Elle propose de plus un catalogue de transformation des patterns : transposition, renversement, rotation, changement d'échelle (augmentation rythmique, changement d'échelle micro-tonale, changement de tempo…), interpolation, extrapolation, fragmentation, substitution, combinaison. Notons une certaine parenté de la pensée de Laurie Spiegel avec la technique sérielle d'Arnold Schoenberg. Les manipulations sur la série dodécaphonique telle que les a formalisées ce compositeur comprennent les transposition, rétrogradation, fragmentation et combinaison par exemple. Pierre Boulez ira plus loin encore avec le sérialisme total en introduisant des procédés combinatoires sur les rythmes, les intensités ou les attaques comme l'avait fait Olivier Messiaen avec son œuvre « Mode de valeurs et Intensités ». Notons aussi qu'Olivier Messiaen a introduit dans son vocabulaire les personnages rythmiques et

les chants d’oiseau qui démontrent une fois de plus que la composition par patterns fait partie des techniques de composition occidentales ayant une longue histoire.

### 2.3.2 Composition par clips

Sans lien avec l’analyse musicale ou la musique sérielle, le terme composition par clip fait référence aux outils qui se sont développés autour de la notion de boucle, de « step sequencer » et de « drum box », qu’en français l’on traduit le plus souvent par « boite à rythme ». Les « step sequencers » sont des outils qui permettent d’enregistrer et d’exécuter en boucle des séquences assez courtes de commandes MIDI, de séquences de tensions électriques ou de sons préenregistrés. Le principe de la boucle s’est développé très tôt avec les premiers synthétiseurs. Les musiques électroniques pour la danse, l’Electro, ont largement contribué à la popularisation des boucles.

En parallèle et de façon complémentaire au développement des boites à rythme se sont développés les séquenceurs qui sont des systèmes de notation et de contrôle qui ont connu leur essor avec la popularisation du standard MIDI. De la même famille que les boites à rythme, ils ne se limitent pas à la répétition de séquences en boucle. Originellement mise en œuvre sur des matériels dédiés ou dans des synthétiseurs, ils ont bénéficié du développement de la micro-informatique depuis les *Atari* et *Amiga* [64] des années 1980.

C’est dans les années 1990 avec *FL Studio* [7] (originellement *FruityLoops*) et un peu plus tard en 2001 avec *Ableton Live* que s’opère une forme de symbiose entre le principe de la boucle et celui du séquenceur. La boucle change de nom, elle devient le *clip*. Elle n’est plus limitée à la batterie comme l’étaient les boites à rythmes. L’informatique le permettant, le clip se limite plus à une boucle, il devient une entité musicale que l’on peut intégrer dans une composition longue. Lorsque la musique électronique rejoint celle des DJ, *Ableton* innove en faisant une synthèse de la logique du clip avec celle de la performance du DJ en adjoignant des PAD (*Push*) pour la scène. *Ableton Live* devient alors le produit phare de la musique électronique et pas uniquement sur scène. C’est ainsi que cet objet musical qu’est le clip est devenu un élément commun des musiciens Electro, Techno et autre genres musicaux populaires. Avec les évolutions permanentes des solutions à base de clip apportées par *Ableton* et ses concurrents, les échanges entre musiques savantes et musiques populaires, l’usage du clip s’est aussi largement répandu en musique contemporaine.

### 2.3.3 Patterns d’improvisation

En improvisation, et surtout en Jazz, le terme *pattern* synonyme de *lick* [74][29] fait partie du langage commun. La Figure 2 est un exemple de lick. Un lick fait référence à des cellules mélodiques que le Jazzman peut utiliser quand il le souhaite. Les patterns de Jazz sont liés à un contexte harmonique. Ceci signifie qu’un pattern est défini pour un enchainement fixe d’accords. La tonalité de l’enchainement d’accords n’étant pas figé, un Jazzman doit apprendre et mettre en œuvre ses patterns en les transposant. Une grande partie de l’apprentissage de l’improvisation Jazz repose sur l’apprentissage d’un catalogue de patterns qui constitue le vocabulaire de base du langage de chaque musicien.

**Figure 2 : Un exemple de Jazz Lick**

La variété des patterns qu’un musicien utilise et sa capacité à les enchainer sont donc une composante forte de la richesse de son style.

## 2.4 Conclusion de l’état de l’art

Dans les chapitres précédents nous avons abordé les principaux sujets pour lesquels notre recherche a apporté une contribution. Il s’agit en premier lieu des solutions pour les musiques interactives qui ne proposent en général pas de véritable méthode de composition, mais plutôt des outils d’aide à la mise en œuvre de composition. Il s’agit aussi de l’expression du temps et de la composition par pattern dont nous retrouverons plusieurs caractéristiques dans nos travaux. Pour les technologies du Web et les interfaces, le sujet est ouvert et suffisamment complexe pour mériter plus de développements. Nous l’avons en partie traité pour que notre plateforme soit opérationnelle. Cependant le développement des interfaces homme-machine ne constituent pas le problème central de la définition de la relation entre langage informatique et composition musicale. Bien qu’important, nous ne l’avons pas considéré comme une priorité absolue pour notre recherche.

Pour exposer nos travaux nous suivrons, dans la suite de ce document, un chemin proche de celui d’un compositeur créant une pièce avec notre plateforme. Sachant qu’il a construit son matériau de base sous forme de patterns, nous passerons à l’étape suivante qu’est la définition des orchestrations. Nous verrons qu’elles consistent à décrire des combinatoires complexes sous forme d’automates. Nous regarderons ensuite en quoi consiste la technique de programmation que nous avons retenue. Puis nous détaillerons comment nous avons réalisé des orchestrations au moyen de cette technique de programmation avant de traiter comment mettre en œuvre la durée sous forme d’interaction. En complément de ce parcours de composition, nous aborderons l’architecture technique puis nous exposerons les premières performances réalisées avec notre plateforme et donneront quelques perspectives qu’ouvrent nos travaux.

Mais avant de passer à la description du langage que nous avons choisi, restons dans un registre musical en mettant à l’épreuve le principe de base de la composition par pattern que nous avons traité précédemment. Quelles sont les limites et les opportunités de ce choix ?

# 3 EVALUATION OPERATIONNELLE DE LA COMPOSITION AVEC PATTERNS

**EMULER UNE IMPROVISATION JAZZ** 33

**EMULER AVEC DES TONALITES** 35

**EMULER LE STYLE CLASSIQUE** 36

**EMULER LE STYLE MODAL** 39

**ECRIRE SANS TONALITE** 39

**A LA SCHOENBERG** 40
**A LA MESSIAEN** 40
**COMPOSER AVEC DES SONS** 40

**CONCLUSION : LIMITES ET OPPORTUNITES** 41

> Toute rupture de l'organisation banale suppose un nouveau type d'organisation, laquelle est désordre par rapport à l'organisation précédente, mais ordre par rapport aux paramètres du nouveau discours. (L'œuvre ouverte, U. Eco, p.87)

Nous avons vu un rapide historique de ce que recouvre techniquement le terme pattern en musique dans le chapitre sur l'état de l'art et plus particulièrement dans la section dédiée à la composition par patterns. Nous avons vu que le concept n'est pas nouveau et qu'il peut être décliné en termes de thème, motif ou cellule de l'analyse musicale, de clip en Electro, en « lick » pour le Jazz, ou en matériaux de base pour la musique sérielle. Dans la mesure où nous avons choisi de baser notre réflexion sur la notion de *transitions* pouvant exprimer la *durée*, en passant par des *états* qui sont des patterns, nous devons explorer les opportunités et les limites musicales du choix de la composition par pattern, avant d'exposer en quoi nous lui avons donné une nouvelle dimension grâce à notre concept d'orchestration basé sur les automates complexes.

Une façon d'évaluer notre choix de la composition par pattern est de partir d'une situation extrême, d'étudier un *cas limite*. Pour nous, ce cas limite consistera à partir d'une vision « pessimiste » du comportement d'une audience, c'est-à-dire dont le comportement ne tiendrait pas compte des paramètres subjectifs liés à chacun des individus. Ce serait une audience « sans conscience », ce qui est évidemment une vision limite car négative et réductrice du comportement d'une audience. Une audience sans conscience peut se modéliser avec un processus aléatoire. Notre propos sera donc d'évaluer quel serait le résultat obtenu par la combinaison de patterns mélodiques activés de façon aléatoire dans un style musical prédéfini. La décision d'émuler des styles connus part du besoin d'avoir des éléments de comparaison simples. Le but de ces exercices de construction et de « tests » musicaux n'est pas d'aboutir à des œuvres originales. Nous avons cependant fait face à des résultats assez surprenants permettant d'imaginer que la méthode pouvait produire des réalisations originales qui laissent à penser que des créations musicales nouvelles sont possibles.

Notons que nous aurions pu éviter ici tout comportements aléatoires et nous contenter d'écrire de la musique à base de patterns de façon complétement statique en figeant les combinaisons. Les résultats, en étant plus stables, auraient certainement permis de travailler plus facilement avec des formes musicales bien définies, en revanche cette démarche aurait été plus lointaine de l'objectif d'interaction que nous nous sommes donné.

## 3.1 Emuler une improvisation Jazz

L'émulation d'une improvisation Jazz fait partie des objectifs les plus naturels pour une œuvre à base de patterns. En effet, la notion de « lick » est à notre disposition, elle est proche de nos patterns. La différence fondamentale entre un « lick » et un pattern dans notre méthode de composition est le mode de synchronisation. Dans notre cas, si les patterns sont sollicités, aléatoirement ou par une audience, sans contrainte forte sur le temps des horloges, leur exécution pourra être soumise à un principe de *synchronisation*. Nous entendons par synchronisation des patterns un mécanisme qui contrôle l'instant auquel un pattern commence à être joué après avoir été sollicité aléatoirement ou sélectionné par une audience. La synchronisation permet de garantir au compositeur

que les patterns qu'il a imaginés, sans être combinés de façon prédictible, pourront être *en phase*. Dans le cas de la musique interactive, nous verrons plus loin que la réalisation de cette mise en phase est définie par le compositeur et qu'elle dépend des choix faits par les membres de l'audience. Elle pourra aller de très contraignante, en fixant des intervalles de temps bien précis conformes aux longueurs des patterns par exemple, ou totalement absente. Cette synchronisation globale est ce qui définit ou non un premier niveau de cohérence de l'œuvre musicale dans notre méthode. C'est ce qui permet au compositeur de s'assurer qu'il ne se produira pas « n'importe quoi » lors de l'exécution de l'œuvre ou au contraire qu'il laisse libre cours aux départs des patterns sélectionnés par l'audience ou par des processus aléatoires. L'activation des « licks » par un musicien de Jazz, elle, n'est soumise à aucune contrainte autre que son bon vouloir.

En ce qui concerne l'exemple émulant une improvisation Jazz, nous avons commencé par définir une petite formation composée d'un piano, d'une percussion, d'une basse, d'un saxophone, d'une batterie et d'une guitare. Pour simplifier ce premier travail, nous avons choisi un style Jazz-Rock ou Jazz Fusion. C'est un style qui s'accommode bien d'une grande stabilité harmonique, et donc qui permet dans un premier temps de nous affranchir de la gestion des changements de tonalités, qui sont une question que nous verrons par la suite. Nous avons construit une maquette d'abord uniquement à l'aide d'*Ableton Live* (une autre DAW aurait aussi fait l'affaire) dont nous avons l'habitude. Pour ceci nous avons construit une sorte de base de données de patterns, tous de la même longueur, soit deux mesures à quatre temps. Pour un tempo de 110, un pattern dure donc 4,36 secondes, pour un tempo de 120 un pattern dure 4 secondes. Le choix de cette longueur est important. Il fait partie des critères à évaluer en matière de « qualité » de l'interaction et « qualité » musicale. Des patterns courts offrent plus de dynamique dans l'interaction, mais introduisent plus d'incertitude dans le résultat. Les patterns trop longs peuvent démotiver les participants. Le choix de la longueur des patterns doit aussi tenir compte de la dimension de l'audience prévue pour la performance. Des patterns trop courts avec une petite audience, risquent de donner un résultat haché. Les 4 secondes environ que nous avons choisis semblent à propos pour une audience de 20 personnes environ. Nous avons synchronisé les patterns sur leur durée.

Une fois choisie la formation et la longueur des patterns, nous avons cherché à tâtons à partir de combien de patterns la pièce semblait réaliste en introduisant un mécanisme d'enchainement aléatoire des patterns pour chaque instrument. C'est un processus simple à réaliser avec un automate trivial ou directement dans *Ableton Live*. Dans le style choisi et avec un tempo de 110, nous sommes arrivés à un résultat qui peut simuler une improvisation avec :

- 8 patterns au piano
- 6 patterns à la percussion
- 18 patterns au saxophone et le même nombre à la trompette
- 18 patterns à la batterie
- 4 patterns à la guitare
- 8 patterns à la basse

Le nombre relativement important de patterns au saxophone et à la trompette s'explique par les rôles de soliste de ces instruments. Le nombre de patterns à la batterie est lié au rôle important de cet instrument en Jazz-Rock et à la volonté de casser le côté mécanique de type boite à rythme auquel on risque d'aboutir avec des clips.

La façon de concevoir les patterns relève de la création musicale au sens habituel du terme. Il s'est agi d'inventer des portions de phrases musicales qui pouvaient s'enchainer dans n'importe quel ordre, ce qui était largement facilité par la stabilité harmonique. Nous avons essayé de ne pas

donner des structures trop symétriques aux patterns, et d'introduire fréquemment des respirations. De façon à illustrer le mécanisme, voici un exemple de combinaison de deux patterns :

donneront

qui pourraient être aussi

Une fois la phase de composition des patterns réalisée, nous avons lancé un processus de sélection aléatoire des patterns. Un résultat de ce processus « d'improvisation » peut être écouté à l'adresse : https://soundcloud.com/user-651713160/grand-loup.

## 3.2 Emuler avec des tonalités

Introduire des tonalités au sens musical dans un système de patterns n'est pas un projet évident. Dans une logique d'émulation d'un style, la façon de faire évoluer la tonalité dépend de ce style. Les cas les plus simples sont du type « Jazz ». J'entends par type « Jazz », les musiques qui suivent une trame harmonique figée, une grille répétitive. Ce qui est le cas de la majorité des musiques populaires occidentales.

Quand on veut aborder des genres musicaux plus complexes que les musiques populaires, les changements de tonalité (les modulations) peuvent suivre des règles plus ou moins compliquées et font appel à des mécanismes qui ne relèvent pas de la simple transposition mais qui relèvent aussi des structures mélodiques et de la forme de l'œuvre. Or gérer des structures mélodiques n'est pas nativement ce que l'on imagine de faire à partir de patterns choisis avec une certaine liberté par une audience, ou de façon aléatoire. Ceci est encore plus vrai en ce qui concerne la forme de l'œuvre. Nous verrons que ce qui concerne la forme relève du chapitre relatif à ce que nous appellerons l'orchestration. Nous avons quand même tenté l'opération.

Pour commencer à envisager les changements de tonalité, nous avons traité le cas des musiques de type « Jazz ». Dans ce cas nous avons expérimenté le fait de transposer et de convertir des patterns de façon modale en fonction de l'évolution des tonalités. Ce qui est proche de la façon dont on apprend à manipuler des échelles en improvisation Jazz. En reprenant les termes de l'harmonie fonctionnelle, nous dirons qu'un pattern dans le premier degré (fondamental) sera converti dans le mode Ionien, un pattern transposé dans le 2$^{\text{ème}}$ degré sera converti en mode Dorien, 3$^{\text{ème}}$ degré en mode Phrygien, 4$^{\text{ème}}$ degré en mode Lydien, 5$^{\text{ème}}$ degré en mode Mixolydien, 6$^{\text{ème}}$ degré en mode Éolien et 7$^{\text{ème}}$ degré en mode Locrien. Les conversions des patterns selon les modes est un *choix musical*, car il pourrait y avoir plusieurs façons de faire ces conversions. Notre façon de faire ne produira pas des résultats extrêmement originaux mais elle permet de s'assurer qu'il n'y aura pas d'incohérence harmonique entre les solos et la grille d'accords.

Voici le tableau de conversion de gamme diatonique en mode et la transposition sur le degré de la gamme diatonique correspondant. Nous voyons donc que toutes les notes de la transposition de la gamme diatonique de départ sur un degré de cette gamme, redonnent des notes dans la même gamme.

Conversion modale
Dorien
Phrygien
Lydien
Mixolydien
Aéolin
Locrien
Transposition
2ème degré
3ème degré
4ème degré
5ème degré
6ème degré
7ème degré

Par exemple dans une tonalité de Do majeur, ceci signifie qu'un pattern conçu dans un mode Ionien au-dessus d'un accord de Do majeur, une fois transposé sur le deuxième degré dans la même tonalité, sera compatible avec un accord de Ré mineur. On pourra aussi transposer le résultat de cette conversion pour changer de tonalité de base. Par exemple, en transposant le résultat un ton au-dessus, Ré majeur, notre pattern sera compatible avec l'accord du deuxième degré de Ré majeur, soit Mi mineur. Ainsi tout pattern conçu dans un environnement harmonique peut être adapté à un autre environnement harmonique.

Le tableau précédent sur les conversions de mode ne traite que la gamme diatonique. Le *traitement musical* évoqué précédemment ne concerne pas les notes étrangères à la tonalité. On peut choisir : soit de ne pas convertir ces notes, donc de les garder étrangères à la tonalité, soit de les convertir selon le cas en une note dans le mode, en dessous ou en dessus. Il n'y a pas de règle absolue pour ces conversions car l'introduction de notes étrangères à la tonalité (notes altérées) peuvent avoir des objectifs mélodiques différents qui ne toléreront pas tel ou tel choix de conversion. Ce sera au compositeur de définir le processus qui lui conviendra le mieux.

En faisant un exercice du même type de que précédemment dans un style middle Jazz, avec un volume équivalent de patterns déclenchés de façon aléatoire et des transpositions qui se déroulent de façon séquentielle et non aléatoire, nous obtenons une simulation du type https://soundcloud.com/user-651713160/hope. Le premier constat est que le modèle fonctionne assez bien, néanmoins il posera un certain nombre de questions sur l'interaction. Si l'on propose à l'audience de choisir les patterns en les écoutant, quelle sera la perception d'un auditeur dont le pattern choisi est converti et transposé ? C'est un point important à prendre en compte lors de la composition d'une œuvre interactive.

## 3.3 Emuler le style classique

En allant un cran plus loin dans l'application de la tonalité nous pouvons tenter d'aborder un style « classique » en référence aux musiques du 18ème siècle. L'exercice est plus difficile car la gestion des modulations est plus complexe qu'en Jazz, la notion de grille d'accord n'est pas aussi figée, elle dépend en partie de la forme de la pièce. La forme (sonate, fugue, passacaille, rondo, lied, …) est une notion qui est plus développée ici qu'en Jazz où la forme est généralement simple et répétitive. Elle se réduit en général en Jazz à ensemble de variations sur une suite d'accords pouvant comporter des notions de couplet refrain qui acceptent bien le processus aléatoire que nous utilisons pour nos activations de patterns de notes.

Le problème d'une émulation de musique classique est plus compliqué que dans le cas du Jazz, car peut-on envisager d'émuler un style classique sans définir une forme ? La réponse est qu'il sera impossible d'émuler une véritable forme sonate dont le principe n'est pas compatible avec des processus aléatoires sur des éléments mélodiques. On pourra cependant se rapprocher d'une suite

de variations. Nous traiterons la question de la forme dans le cadre du chapitre sur l'orchestration, nous verrons que nous pouvons créer des formes très fines et complexes avec notre méthode, mais pas dans les mêmes catégories que celles qu'a créées la musique dite « classique ». Dans l'exercice que nous nous proposons, qui ne s'intéresse qu'aux traitements de base des patterns, nous allons donc un peu oublier les questions de forme en considérant que nous allons créer des exemples de pièces libres, des sortes de fantaisies proches de l'improvisation au niveau mélodiques, mais figées au niveau harmonique. Nous allons en profiter pour introduire un nouveau mécanisme de transposition.

Si nous souhaitons introduire des schémas harmoniques, nous pouvons appliquer la méthode de conversion de patterns vue au chapitre précédent, mais aussi introduire une autre catégorie de patterns que nous appellerons des patterns harmoniques. Ces patterns harmoniques ne comportent pas de notes mais des indications de conversion modale et de transposition. Ils se superposent à des patterns de notes pour les transformer. On pourrait utiliser le terme « convoluer » inspiré du traitement du signal pour ce processus. Les patterns harmoniques nous permettent d'imaginer que l'orchestration, et donc l'interaction, puisse contrôler l'évolution harmonique de façon fine, dans des subdivisions de patterns, ce que nous ne faisions pas précédemment dans le cas des improvisations Jazz.

Pour illustrer la « convolution », voici un exemple de transposition automatique d'un pattern constitué de notes d'un accord de do majeur. Il y a une transposition à chaque blanche. Les patterns de notes et harmoniques durent deux mesures.

Les patterns harmoniques transposent les deux premiers temps vers le $6^{\text{ème}}$ degré sans changer le mode, puis vers le $2^{\text{ème}}$ degré sans changer de mode, puis vers le $7^{\text{ème}}$ degré sans changer de mode pour finir sur le $3^{\text{ème}}$ degré converti en mode dorien. Ceci correspond à ce que l'on nomme une marche harmonique. Ici la phrase mélodique a été choisie très simple pour bien illustrer la transposition.

Ce modèle est techniquement plus lourd à mettre en œuvre que la conversion globale d'un pattern, mais il est conceptuellement similaire.

Patterns vers Classique

Heidelein

**Figure 3 : Exemple de composition en style "classique"**

Dans les deux cas les transpositions automatiques demandent d'être prudent sur les tessitures. Il faut faire attention à ce que l'ambitus des patterns soit compatible avec les transpositions prévues ou mettre en place un mécanisme de contrôle et de recadrage de la tessiture.

Les patterns harmoniques posent néanmoins des questions sur l'interaction car si la conversion simple de patterns pouvait poser des problèmes de perception à l'audience, celui-ci en posera encore plus. En effet, seule la structure rythmique du pattern sera effectivement conservée. L'exemple « Patterns vers classique », dont on voit un extrait, a été généré avec 23 patterns de notes et 7 patterns de transpositions. Sur cet exemple dans les mesures 2 et 3 de la première portée, nous avons à l'origine le pattern suivant de deux mesures :

Ce pattern de notes est modifié sur ces mesures selon un pattern harmonique qui transpose la deuxième mesure du pattern, mesure 3 sur la partition, au $5^{ème}$ degré. En mesure 8 et 9 nous retrouvons ce même pattern transposé de façon différente, selon un pattern harmonique qui commence sur le $1^{er}$ degré, qui au bout d'une blanche transpose vers le $4^{ème}$ degré, puis au début de la mesure 8 transpose sur le $2^{ème}$ degré, et dans la deuxième partie de la même mesure transpose sur le $5^{ème}$ degré. Toute la pièce est explicable selon ce type de procédé.

## 3.4 Emuler le style modal

La méthode de transposition modale décrite ci-dessus pour le Jazz est particulièrement bien adaptée aux structures harmoniques des musiques modales du début du 20ème siècle. En démonstration, voici une courte pièce pour piano produite avec très peu de moyens : 7 patterns mélodiques, 1 pattern d'accompagnement et 1 pattern de basse. Les patterns font tous 2 mesures. Les transpositions des patterns de notes vers les modes enchainent de façon séquentielle : Eolien (la), Dorien (ré), Lydien (fa), Phrygien (mi), Eolien (la) et Locrien (si). Les 7 patterns mélodiques s'enchainent de façon aléatoire.

Vers Satie depuis Patterns

Heidelein

**Figure 4 : Exemple de composition modale**

Nous voyons bien la répétition des mêmes patterns des mesures 1 et 2 en mesure 3 et 4.

Il est facile de suivre la progression modale depuis la ligne de basse qui permet de voir le changement de mode toutes les deux mesures.

La pièce fait ici 20 mesures, mais nous aurions pu naturellement lui donner n'importe quelle longueur et en produire autant de versions que nous le souhaitions.

## 3.5 Ecrire sans tonalité

Nous avons abordé la question de la composition par patterns sous l'angle de la tonalité et de la modalité. Il est donc temps de se poser la question de ce que signifierait ce mode de composition pour des musiques donc le souci principal ne relève pas de ces logiques tonales ou modales. L'exercice est ici à la fois plus compliqué et plus simple.

Plus compliqué car il existe autant de méthodes de composition hors tonalité que de compositeurs. Depuis les mouvements du début du 20ème avec l'école de Vienne, et de l'avant-garde d'après la seconde guerre mondiale, la création d'un langage personnel, et donc de méthodes de composition personnelles, fait partie des fondements de la recherche esthétique de nombreux compositeurs. Il n'est donc plus possible de poser rapidement une question aussi générale que la composition par patterns hors tonalité, ce serait l'objet au moins d'une thèse en musicologie.

La question est aussi plus simple car les systèmes de composition hors tonalité n'étant pas contraints par un corpus de « règles » harmoniques, contrapuntiques ou formelles, quel est l'intérêt de se poser la question de la limite ? On pourrait donc immédiatement conclure que la composition par patterns peut être vue comme constituante d'un langage et qu'elle ne mérite donc en soi aucune justification à partir de méthodes existantes. Ceci ne doit pas justifier de ne pas continuer l'effort commencé avec les musiques tonales. Nous allons aborder quelques styles sous l'angle

forcément réducteur de notre logique de patterns. Ces styles ne se réduisent évidemment pas à notre approche. Il s'agit plutôt de voir comment nous pourrions appliquer certains éléments propres à ces styles à notre méthode de composition.

### 3.5.1 A la Schoenberg

Un des premiers systèmes, pensé en dehors de la tonalité, est le dodécaphonisme de Schoenberg. La théorie de Schoenberg n'est pas très complexe. Elle repose sur le choix initial d'un arrangement des 12 sons de la gamme chromatique que l'on qualifiera de « série ». Cette série sera soumise à des traitements simples (récurrence, renversement, récurrence du renversement, transposition) qui donneront un matériel de base constitué de 48 séries. L'école de Vienne avec Schoenberg, Berg et Webern va largement développer et argumenter autour de ces traitements de base en cherchant notamment des propriétés de symétrie à l'intérieur des séries. Arnold Schoenberg a expliqué sa démarche son ouvrage *Le style et l'Idée* [76], le chef d'orchestre et compositeur René Leibowitz a introduit ces réflexions en France notamment avec l'ouvrage *Introduction à la musique de douze sons* publié en 1949 [53].

Le rapprochement entre la série de 12 sons et les patterns est tentante. On imagine facilement de construire des patterns de façon quasi systématique à partir des 48 versions d'une série. De fait, la méthode dodécaphonique peut s'appliquer à la composition par patterns au même titre qu'une méthode tonale, excepté qu'il y a assez peu de règles strictes en dehors de celles sur la constitution de la série. Schoenberg préconisait d'éviter les implications tonales par exemple, Alban Berg ne sera pas aussi strict que son professeur à ce sujet, alors qu'Anton Webern appliquera la règle à la lettre. Pourquoi donc ne pas penser une musique dodécaphonique sous forme de patterns ?

### 3.5.2 A la Messiaen

La façon de composer d'Olivier Messiaen repose en partie sur des éléments qui relèvent de patterns. Ce sont les chants d'oiseaux et les personnages rythmiques [58]. De plus son travail sur les modes à transposition limité se prêtent bien aux traitements par transpositions présentés précédemment. Nous avons utilisé ces modes dans l'exemple : https://soundcloud.com/user-651713160/opus1-skini-v2-echelle.

### 3.5.3 Composer avec des sons

Nous avons jusque-là parlé uniquement de patterns exprimés sous la forme de note et des traitements possibles sur ces notes. Mais il est simple de concrétiser des patterns sous forme de sons enregistrés, comme celui de la Figure 5, d'échantillons sonores ou de contrôles sur des paramètres de synthèse du son. Les possibilités offertes ici sont celles de l'outil utilisé pour produire les sons, la *Digital Audio Workstation*. Comme nous l'avons vu dans le chapitre sur l'état de l'art, la composition par clips dans le domaine de l'Electro fait souvent appel à ce type composition. Nous avons expérimenté ce type d'écriture dans le projet Fabrique à Musique supporté par le CIRM et la SACEM, où les collégiens qui ont conçu leur pièce de musique ont travaillé plutôt sur des ambiances sonores que des mélodies. Un extrait sonore est disponible à l'adresse : https://www.cirm-manca.org/actualite-fiche.php?ac=1273.

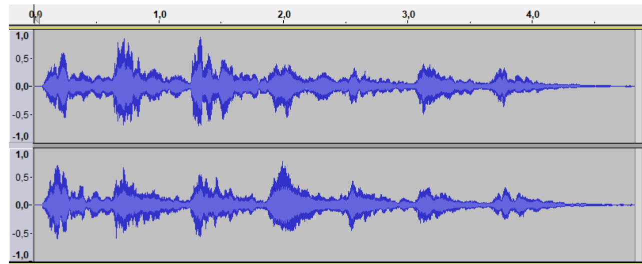


**Figure 5: Un pattern sous forme de son**

## 3.6 Conclusion : limites et opportunités

Nous avons évoqué ici quelques traitements possibles de patterns dans des contextes musicaux connus sur la base de l'activation aléatoire de patterns avec des trames harmoniques fixes. Ces essais se positionnent de façon intermédiaire entre musique complétement écrite, et musique aléatoire. Ce choix intermédiaire répond au besoin de mettre en avant des processus musicaux qui se rapprochent de ce que nous allons rencontrer lors de la création de pièces dans une démarche d'interaction et d'orchestration selon notre méthode de composition. La réflexion sur la composition à base de patterns proposée ici n'a pas donc pour objectif de refaire en moins bien ce qui a été fait, mais plutôt de concrétiser et d'imager les limites d'un exercice et surtout de donner une idée de son potentiel.

Comme première opportunité, en reprenant l'article de Laurie Spiegel, nous voyons que l'éventail des possibilités est plus large que ce que nous avons évoqué ici. Nous avons évoqué l'utilisation d'une part de patterns fixes, c'est-à-dire sans traitement en temps-réel, nous avons d'autre part évoqué la possibilité de modifier des patterns en temps-réel au moyen de patterns harmoniques. Nous ne sommes pas allés plus loin dans nos travaux, mais il est possible d'introduire d'autres types de traitements en temps réel sur les patterns, qu'ils soient sous forme de notes ou de sons. Le traitement en temps-réel des notes est un sujet tout à fait abordable avec notre plateforme, qui est naturellement bien adaptée au traitement algorithmique de données MIDI. Parmi les traitements applicables aux patterns, notons aussi la création par l'audience qui fait l'objet d'un chapitre à part sur le séquenceur distribué.

Comme deuxième axe d'opportunité, nous avons le traitement du son. Les possibilités sont infinies dans la mesure où il est possible de contrôler des synthétiseurs ou autre processeur de son via les protocoles MIDI ou OSC. Par ailleurs, nous avons évoqué ici les patterns au sens musical du terme, mais notons que des patterns peuvent aussi contrôler la lumière ou des robots de scène de la même façon que des patterns musicaux. Nous avons fait quelques essais de contrôle DMX de la lumière au travers de commandes OSC, ceci ne pose aucune difficulté technique importante.

Nous voyons donc que l'éventail des traitements possibles n'est pas limité d'un point de vue algorithmique. Il le sera néanmoins par d'autres facteurs dans notre cas. Le premier est lié à l'interaction avec l'audience. Pour qu'elle soit possible, c'est-à-dire motivante pour les participants, le compositeur doit imaginer des scénarios où l'audience puisse se repérer, comprendre son rôle. Plus les traitements sur les patterns seront sophistiqués, plus l'interaction devra trouver les moyens originaux de faire ressentir ces traitements à l'audience.

Il existe une autre limite. La composition par patterns revient à avoir une approche de type pointilliste de l'œuvre musicale. Ceci a un impact sur les formes musicales que va générer notre méthode de composition. Nous avons vu que le comportement non-déterministe de l'audience fait que notre système à base de patterns n'est pas adapté, ou du moins facilement adaptable, à une vision contrapuntique de la musique, ou à une forme sonate par exemple. Nous n'avons pas encore trouvé le

bon moyen de mettre en œuvre une fugue. La superposition structurée de mélodies de façon aléatoire ou de développements mélodiques sont des problèmes difficiles. Il peut probablement y avoir des solutions en modifiant les mécanismes de synchronisation des patterns, en utilisant des réseaux de neurones, mais il s'agit de sujets de recherche à part entière en dehors de notre propos. Nous pouvons conclure qu'une méthode de composition à base de patterns n'est pas un moyen d'émuler les formes musicales classiques, même s'il est possible de s'en inspirer et de créer des atmosphères qui se rapprochent d'univers musicaux « habituels ». En revanche, dans le chapitre sur l'orchestration nous proposerons de mettre en œuvre un type nouveau de formes complexes, qui se rapprocheront de la logique des « œuvres ouvertes ».

# 4 PENSER L’ORCHESTRATION PAR PATTERN ET AUTOMATE

> Considéré sous son aspect poétique, cet art s'enseigne aussi peu que celui de trouver de beaux chants, de belles successions d'accords et des formes rhythmiques originales et puissantes. (Traité d'instrumentation et d'orchestration, H. Berlioz, p.2)

Dans le chapitre précédent nous avons tenté d'évaluer le potentiel de création offert par une approche d'une œuvre musicale sous l'angle de combinaisons aléatoires de patterns. Cette approche ne faisait pas appel à une méthode d'organisation particulièrement évoluée des patterns. Nous considérions qu'ils étaient disponibles et qu'il suffisait de les jouer de façon aléatoire ou faiblement organisée. Nous allons aborder à présent l'élément structurant de notre méthode de composition qui consiste à définir une organisation des patterns. Organisation que nous aurons à exprimer au moyen d'un langage informatique adapté. Nous appellerons cette organisation une *orchestration*. Nous verrons qu'elle vise à contrôler les *combinaisons possibles de patterns* à un instant donné. Dans ce chapitre, nous définirons tout d'abord ce que nous entendons par *orchestration*. Nous aborderons ensuite les principes sur lesquels elle repose : l'*interaction* et les *groupes de patterns*. Puis nous verrons trois exemples de description d'orchestration de plus en plus riches avant d'évoquer deux façons de contrôler leurs mises en œuvre. Commençons tout d'abord par définir ce que l'on entend par *orchestration*.

Dans le langage musical courant, l'orchestration est la description dans le temps (des horloges) des instruments qui interprètent une pièce de musique. On peut par exemple organiser les instruments par groupes (violons, altos, violoncelles, contrebasses, cors, percussions, etc.), le compositeur choisira quel groupe sera actif pour tel ou tel élément de la partition. Certain compositeur, comme Richard Wagner, écrivait d'abord une version piano/chant et passait ensuite dans une phase d'orchestration. Il s'agit d'un domaine bien identifié dans l'apprentissage de la musique qui fait l'objet d'une littérature spécialisée comme le traité de Vincent d'Indy [88], de Charles Koechlin [48] ou celui d'Hector Berlioz [14], ou dans le monde du Jazz celui d'Ivan Julien [44] par exemple. Pour les informaticiens l'orchestration a un autre sens. Il s'agit de gérer des processus automatiques d'organisation, de coordination de systèmes. Plus spécifiquement dans le contexte du Web, il s'agit des techniques employées pour coordonner différents services. Une requête web, assez simple pour un utilisateur, comme une recherche de vols entre deux destinations, peut en réalité faire appel à plusieurs services d'interrogation de compagnies aériennes différentes qui utilisent des serveurs totalement différents. Pour que l'information puisse être présentée à l'utilisateur, le service qui a lancé la requête initiale va devoir attendre et gérer les résultats des différents serveurs. C'est cette gestion que l'on nomme *orchestration informatique*.

Nous voyons donc que les deux sens du terme, n'ont en commun que le souci de coordonner des acteurs, d'un côté des instruments de musique, de l'autre des services informatiques. Les fonctionnements dans le temps des deux systèmes n'ont rien en commun. Celui de l'orchestration musicale relève de la gestion d'un flux selon un tempo, celui de l'informatique consiste à gérer des successions d'événements sans contrainte d'horloge. Notre méthode de composition opère une forme de synthèse entre ces deux approches. Il va s'agir de gérer des événements et pas seulement le temps, dans le but de contrôler des instruments ou des groupes d'instruments. Pour comprendre ce qu'est notre orchestration, nous devons partir des notions d'*interaction* et de *pattern*. Nous avons déjà

traité la question des patterns précédemment, voyons en quoi consiste l'interaction dans notre cas, avant de justifier l'organisation en groupes des patterns.

Notre contexte musical est celui d'un lieu regroupant un ensemble d'individus venu pour participer à une performance musicale. Cet ensemble d'individus constitue une audience. Le principe de l'interaction est de permettre à cette audience de *sélectionner* des patterns qui seront *joués* par un équipement électronique ou des musiciens. Ceci signifie qu'en fonction de certaines circonstances définies par le compositeur, l'audience aura accès à des patterns préconçus aux moyens de terminaux (smartphones, tablettes…) et que la *mise à disposition des patterns pour l'audience* doit être signalée et motivée par des interfaces homme-machine appropriées. Nous appellerons cette mise à disposition *activation*. Nous verrons plus en détail la mise en œuvre de l'interaction dans un chapitre dédié (Mise en œuvre de la durée, p.95), mais les principes d'*activation* et de *sélection* par l'audience sont suffisants pour comprendre comment s'organise notre orchestration.

L'orchestration consiste d'abord à organiser les patterns en *groupes*, puis à définir une façon de gérer ces groupes. On pourrait dire que l'organisation des patterns en groupes relève de l'orchestration musicale et que la façon de gérer les groupes relève de l'orchestration informatique. Nous avons choisi d'organiser nos patterns en groupes, qui ne correspondent pas forcément à des groupes d'instruments. Voici une explication à ce choix : Nous avons déjà vu, dans le chapitre précédent sur la composition par pattern, les principales caractéristiques des patterns et les possibilités de traitement que l'on peut leur appliquer. Dans ce chapitre nous n'avions considéré que des scénarios aléatoires très simples d'activation. Les combinaisons de patterns activés dans un contexte collaboratif donnent accès à un grand nombre de possibilités. En effet pour N patterns nous aurons $2^N$ combinaisons de patterns possibles à un instant. Pour 10 patterns nous aurons $2^{10} = 1024$ combinaisons, pour 50 patterns $10^{15}$ combinaisons, pour 100 patterns $10^{30}$ combinaisons, etc. La définition des groupes patterns, en constituant la première étape de l'orchestration, est une façon de réduire le nombre de combinaisons de patterns possibles à un instant. Si nous prenons 3 groupes de I, J et K éléments avec $N=I+J+K$ nous réduirons le nombre de combinaisons de $2^N$ à $2^I+2^J+2^K$. Dans le cas de 10 patterns et 3 groupes de 2, 3 et 5 éléments nous passerons de 1024 combinaisons à 44 par exemple. Le compositeur en groupant les patterns introduit un premier niveau de contrôle sur les possibilités musicales de son travail à partir duquel il va pouvoir organiser la mise à disposition des groupes pour l'audience, c'est-à-dire l'orchestration dans une dimension *informatique*.

A partir de la notion de groupe de patterns, il s'agit de penser une orchestration informatique. Doit-on passer par une étape préliminaire de conception, si oui laquelle ? En effet il ne nous a pas semblé très simple au premier abord de commencer à définir une orchestration directement avec un programme informatique. Les compositeurs sont habitués à écrire des orchestrations sous forme de partitions et non de programmes. Pourrait-on donc imaginer une représentation qui nous servirait de spécification au codage informatique et qui soit en phase avec son expressivité ? Voici une première tentative de représentation que nous avons expérimentée :

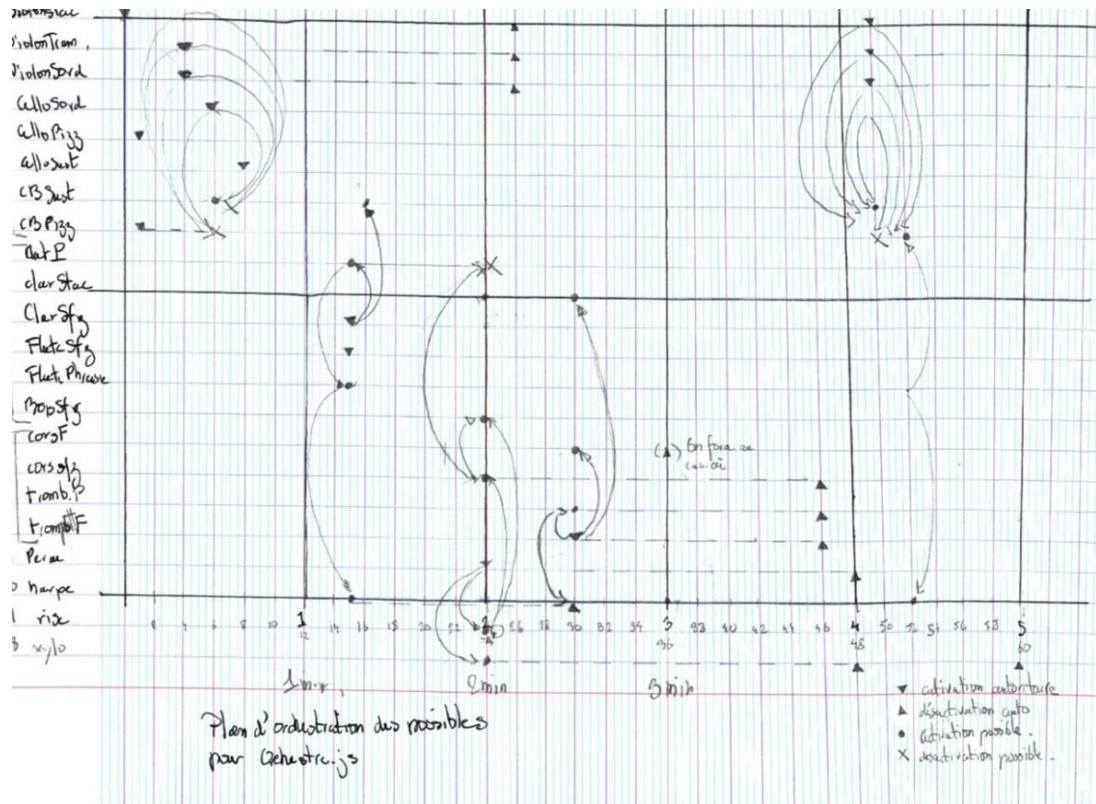



**Figure 6 : Une orchestration sur papier**

Ce schéma a des points communs avec une partition, cependant l'axe horizontal n'est pas le temps mais une façon de représenter des séquences d'événements, alors que l'axe vertical les représente en parallèle. Les évènements sont les activations des groupes de patterns nommés en marge de l'axe vertical. Il s'agit des ensembles de patterns que nous pourrons mettre à la disposition de l'audience en les *activant* et cacher à l'audience en les *désactivant*. Nous avons appelé ce document un « plan d'orchestration des possibles », car cette description n'est pas déterministe. Elle exprime *un champ de possibilités offertes à l'audience*, dans ce sens elle n'est pas une partition traditionnelle avec type portée et mesures. Rien n'est acquis sur la façon dont ces possibilités seront accueillies par l'audience. Il s'agit de la formalisation d'une *durée,* de façon non temporelle, qui ne se concrétisera qu'à travers une interaction. Il se pourra que certains groupes soient très actifs, d'autres non. Le compositeur pourra donner des poids aux groupes en jouant sur le nombre de patterns qu'ils contiennent par exemple, mais il n'est pas maître du résultat finalement produit. Bien que se déroulant dans un cadre, la performance sera unique, c'est-à-dire pratiquement impossible à reproduire exactement.

Dans ce premier essai nous représentons les « activations autoritaires » de groupe avec un triangle vers le bas. Une « activation autoritaire » est la mise à disposition d'un groupe sans tenir compte du comportement de l'audience. La « désactivation autoritaire » suit le même principe mais en faisant disparaitre un groupe du champ de vision de l'audience. L'activation possible représentée par des petits cercles, signifie que l'on activera un groupe pour donner suite à un certain nombre de sélections de patterns dans un groupe. C'est pour cela que les cercles sont suivis de flèches partant d'un groupe *activé* et pointant vers un ou plusieurs groupes *activables*.

A plusieurs titres cette représentation n'est qu'une étape intermédiaire. Elle introduit certaines restrictions sur l'expression de la durée issues de la représentation d'un axe horizontal quasi-

temporel. La première est sa difficulté à exprimer des opérations logiques sur des combinaisons d'événements. Il n'est pas très facile de signifier que : nous souhaitions que tel groupe soit activé quand X sélections sont issues du groupe A, *et* que Y sélections sont issues du groupe B, *ou bien* que nous ayons vu apparaitre Z sélections du groupe C indépendamment de ce qui a pu se passer dans les groupes A ou B. Par ailleurs, il est naturellement possible de diviser une orchestration en plusieurs moments, de définir une hiérarchie dans la structure de l'orchestration, de façon à pouvoir contrôler simplement des pans entiers. Nous avons donné un nom à un premier niveau de ces sous-orchestrations, ce sont des *sessions*. Penser l'orchestration sous forme de sessions est un moyen de remettre en cause son déroulement séquentiel, d'imaginer que des pans entiers de l'orchestration puissent se dérouler dans des ordres dépendant de l'interaction, et même se dérouler en parallèle. La combinaison de sessions par des sessions ouvre le champ du possible à un niveau de complexité difficilement imaginables. Une pièce peut en effet, par des mécanismes de combinatoires complexes, produire des résultats surprenants, mais musicalement cohérents.

Pour faire suite à notre première tentative, nous avons affiné notre approche en tenant compte de cette idée de session. Voici une autre représentation plus récente (Figure 7: Une orchestration avec sessions) qui, de plus, tient compte de la différence entre groupes *répétitifs* et *réservoirs* (sound tanks). Les *groupes répétitifs sont* ceux dont l'ensemble des patterns est disponible pour l'audience quand ce groupe est activé, c'est-à-dire mis à la disposition de l'audience comme nous l'avons vu précédemment. Les *réservoirs* sont des ensembles de patterns qui se réduisent au fur et à mesure que les patterns sont sélectionnés. Les réservoirs sont une façon de contrôler la non-répétition de pattern. Nous reverrons la pièce Opus1 (Figure 7: Une orchestration avec sessions) sous son angle informatique plus tard. Elle inclue trois sessions indépendantes. Les flèchent avec des chiffres indiquent les nombres d'occurrences de signaux pour que des groupes soient activés. Les lignes sans flèches indiquent les limites du délai durant lequel un groupe est actifs. Les lignes verticales permettent d'indiquer l'ensemble des groupes dépendant d'un délai. Les réservoirs se distinguent graphiquement des groupes répétitifs. Cette représentation introduit de plus une notion de combinaison logique pour les activations. Dans la « session scale » nous avons par exemple un « et » logique (*and*) sur la fin complète des réservoirs *TrumpetsScale*, *HornsScale* et *TrombonesScale* avant l'activation de *PianoScale*. Notons que le temps est un événement comme un autre indiqué par des flèches en pointillé et qu'il est décompté en *ticks*.

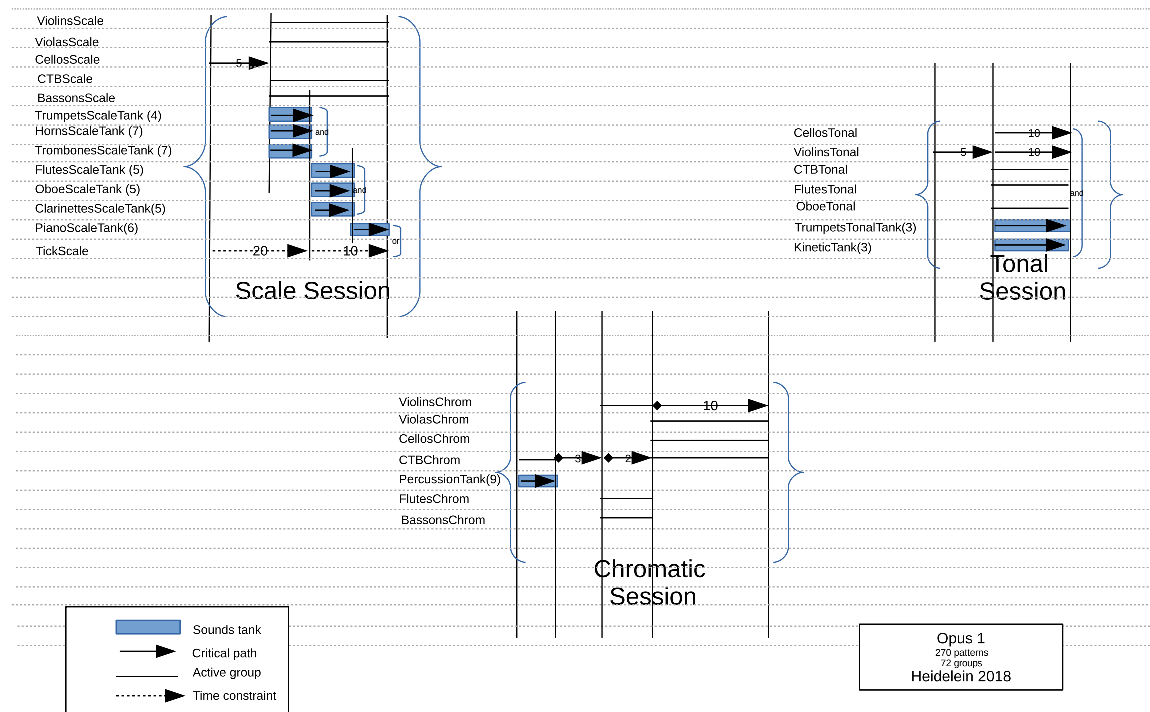



**Figure 7: Une orchestration avec sessions**

Cette représentation permet de visualiser les chemins possibles pour une pièce et notamment de s'assurer qu'il y aura bien une fin. Elle est une façon d'appréhender une œuvre sous une forme facilement compréhensible. Elle permet aussi de faire une relation visuelle rapide avec une partition qui serait générée par le système Skini. Mais comme nous l'avons vu, elle n'est pas entièrement satisfaisante pour un compositeur qui souhaiterait introduire des conditions logiques complexes sur des événements ou des ensembles d'événements.

Figure 8: Orchestration Opus2, donne une autre orchestration un peu plus riche que la précédente. Elle met en œuvre des organisations différentes pour quatre sessions distinguées par des ambiances que nous avons associées à des couleurs, bleu, rouge, noir et jaune. Nous retrouvons ces noms de sessions dans les noms des groupes de patterns (*AltosBleu*, *ViolonsRouge*,…). Ces couleurs correspondent à des matériaux musicaux différents. Les patterns de la session bleue sont construits de façon « libre atonale », ceux de la session rouge sur un mode à transposition limités (demi-ton, ton) avec une fondamentale en do, ceux de la session noire sont construit sur une base dodécaphonique, et ceux de la session jaune sur une base de mode à transposition limités (demi-ton, ton) avec une fondamentale en sol. Cette pièce est un essai avec des patterns longs de 16 noires pour trois sessions et des patterns plus courts pour une session. Les tempos sont modifiés pour chacune des sessions. Nous voyons pour chaque session une ligne *transposition*, qui signifie que l'orchestration va contrôler des mécanismes de transposition pour un ou plusieurs instruments. Pour la session noire par exemple nous voyons que dès le début l'orchestration met en œuvre une transposition pour le piano à chaque *tick*. Ici la durée du *tick* n'est pas visible dans le document, mais elle a été fixée à la durée des patterns. A chaque *tick* nous transposons le piano d'un demi-ton modulo 6, c'est-à-dire jusqu'à une quinte diminuée pour revenir à la « tonalité » d'origine.

Parmi les nouveautés introduites dans cette orchestration nous avons, pour la session jaune que nous avons dans la partie inférieure du schéma, trois séquences associées à trois variables qui seront choisies aléatoirement (aléas 0, 1 et 2). Pour chacun de ces aléas nous définissons un déroulement un peu différent et un tempo différent. Ceci signifie qu'à chaque exécution de la pièce, au-delà de sélections différentes des patterns par une audience, nous aurons un résultat qui peut être sensiblement différent.

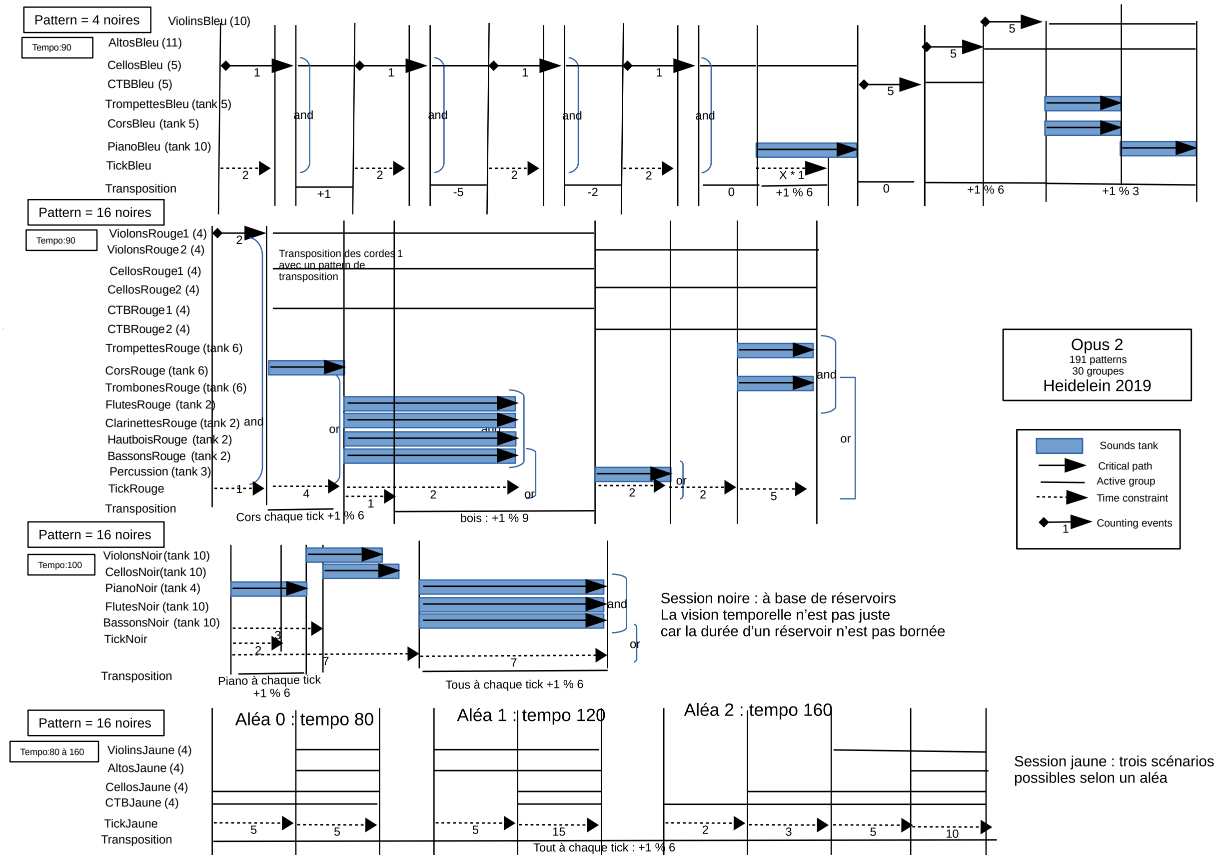



**Figure 8: Orchestration Opus2**

Nous avons précisé que pour enrichir encore l'orchestration nous pouvions aussi décider de mélanger les sessions. Par exemple décider que l'occurrence d'un ou plusieurs patterns particuliers va déclencher une autre session en parallèle. Ce type d'orchestration est plus difficile à exprimer graphiquement, mais elle s'exprime bien « oralement » et aussi avec le langage informatique que nous avons choisi. Pour démontrer ce principe, imaginons de commencer notre orchestration par une session jaune. A la sélection d'un pattern bien précis au violoncelle nous lancerons la session noire. Sans entrer dans la mise en œuvre de ce scénario, la Figure 9 est un extrait d'une partition obtenue à partir d'une exécution de cette orchestration qui montre que l'opération est possible. De la mesure 1 à 16 nous voyons entrer d'abord les violons. Il est facile d'identifier les patterns qui durent quatre mesures. Puis entrent les altos, puis les violoncelles et les contrebasses. Nous sommes bien dans le cas de l'aléa 1 de la session jaune. En mesure 17 nous voyons apparaitre le piano qui joue un pattern dodécaphonique de quatre mesures. Il s'agit bien du réservoir « piano » du début de la session noire qui s'exécute en parallèle à la session jaune. Ce court exemple est aussi l'occasion de montrer que notre plateforme permet de représenter, sous forme d'une partition, une exécution complète d'une orchestration et de vérifier qu'elle correspond bien au projet initial.

Parmi les éléments autres que les sessions, qui permettent d'enrichir le déroulement d'une orchestration, il est possible d'introduire des patterns de longueurs différentes dans un groupe de patterns. Ceci permet par exemple de laisser l'audience créer des phrases musicales moins « systématiques » que celles construites avec des patterns de même durée. Il est aussi possible de demander à l'orchestration de contrôler des « accelerandi » ou « rallentendi ». L'ensemble de ces possibilités de contrôle offre une large palette au compositeur.

Opus2-2, instance 2

**Figure 9: Exemple de superposition de sessions**

Si notre plateforme d'expérimentation permet au compositeur de contrôler graphiquement son orchestration, nous verrons qu'elle peut aussi générer automatiquement des simulations d'exécutions « sonores ». Celles-ci permettent de vérifier, dans une certaine mesure, la façon dont une orchestration peut varier d'une exécution à l'autre et ainsi de contrôler avant une exécution sur scène la pertinence musicale du projet et sa conformité avec une description graphique. Pour illustrer ceci, à l'adresse « soundcloud.com/user-651713160/opus2-2-5-instances » se trouvent cinq exécutions à la file de l'orchestration Opus2 avec des superpositions de sessions. Il serait fastidieux de détailler cet exemple sonore, mais notons que les exécutions donnent des résultats musicalement différents qui se traduisent notamment par des durées globales variables. La première exécution dure 50 secondes, la deuxième s'arrête à 3min27sec et dure donc 2min37sec, la troisième s'arrête à 4min13sec pour une durée de 46sec, la quatrième s'arrête à 6min13sec pour une durée de 2min et la dernière s'arrête à la fin de l'exemple soit 8min11s pour une durée de 1min58sec. Ceci montre que la durée de l'exécution dépend de la mise en œuvre de l'orchestration. Si une performance est impossible à reproduire exactement de façon automatique, ceci sera plus vrai encore avec une audience. Cette impossibilité de reproduire une œuvre lorsqu'elle sera concrétisée par un ensemble d'individus (donc de consciences) est significative d'une forme de mise en œuvre de la durée selon Bergson.

Nous avons posé les bases musicales de notre recherche : patterns et orchestration. Il s'agit maintenant d'aborder la mise en œuvre d'une pièce musicale avant qu'elle puisse prendre vie avec une audience. Pour passer à cette mise en œuvre, une étape préliminaire est nécessaire. Il s'agit d'expliquer comment et pourquoi nous avons choisi un langage informatique particulier proposant une expressivité capable d'actualiser ce passage. Pour cela il nous fallait répondre aux trois points de la question : Quel est le langage dont l'expressivité correspondait à nos orchestrations mais qui soit aussi suffisamment malléable pour permettre des mécanismes d'essais-erreurs rapide et qui permettrait de plus une mise en œuvre efficace de l'interaction avec des audiences ?

# 5 LA PROGRAMMATION AVEC HIPHOP.JS

PRINCIPE DE LA PROGRAMMATION REACTIVE SYNCHRONE 56

PROGRAMMER AVEC HIPHOP.JS 56

PENSER A L'ARCHITECTURE 56
HIPHOP.JS ET LE TEMPS : DELAI ET INSTANT 57
LES INSTRUCTIONS HIPHOP.JS 58
ACTIVER UN MODULE DEPUIS JAVASCRIPT 60
PREEMPTION 61
CONSTRUCTION DYNAMIQUE DE PROGRAMME 61
HYPOTHESE INSTANTANEE 62

HIPHOP.JS VS. JAVASCRIPT 63

Il est possible de diviser les techniques de programmation entre les systèmes dit *réactifs* et *interactifs,* et ceux dit *transformationnels* [36]. Les systèmes *réactifs* et *interactifs* sont ceux qui traitent des *événements* discrets, systèmes que l'on nomme aussi *automates*, sans se préoccuper du temps des horloges, en opposition aux systèmes transformationnels qui sont caractérisés par un début et une fin. Nos orchestrations ressemblent beaucoup à des *automates* traitant des événements discrets comme des sélections de patterns, des activations/désactivations de groupes de patterns, des signaux d'horloge, etc. Comme les systèmes *réactifs* sont bien adaptés à ce type de programmation, il est naturel de nous orienter dans leur direction. Parmi les formalismes pour représenter des systèmes réactifs nous avons historiquement les réseaux de Petri conçus dans les années 1960 par Carl Adam Petri [68]. Les *réseaux de Petri* fournissent des outils à la fois graphiques et mathématiques dont le but est de modéliser et de vérifier le comportement d'*automates*. Toujours dans le domaine des automates nous avons *Grafcet* [20] (Graphe Fonctionnel de Commande des Étapes et Transitions) lui-même issu des réseaux de Petri. *Grafcet* est un langage graphique utilisé par l'informatique industrielle et non un outil mathématique. La représentation Grafcet fait l'objet d'une normalisation par l'UTE (Union Technique de l'électricité). Dans le domaine de la modélisation d'événements nous avons aussi les diagrammes d'activités *UML*, dont le but est de représenter des déclenchements d'actions en fonctions d'états d'un système. Ces diagrammes sont utilisés pour modéliser des traitements parallèles (*threads* ou processus), ou représenter des *workflows*.

Les réseaux de Petri sont un outil mathématique, Grafcet est dédié aux automates industriels, UML est surtout un outil de spécification. Ces solutions bien que conformes à nos besoins de programmation d'automates, ne répondent pas à nos critères de malléabilité et de développement d'interfaces avec une audience. En revanche, dans les années 80 et 90 sont apparus en France trois langages adaptés à la programmation des automates : *Esterel* [10], *Lustre* [35] et *Signal* [50]. *Lustre* et *Signal* ont été conçus pour le traitement de flots de données. *Esterel* a été conçu pour le contrôle de flot. *Esterel* et *Lustre* ont été commercialisés par la société *Esterel Technologie* dans un environnement intégré nommé *SCADE* (*Safety Critical Application Development Environment*) [17]. *SCADE* est utilisé pour le développement de systèmes critiques dans le domaine du transport ou de l'énergie. D'autres langages sont apparus à la suite du trio *Esterel*, *Lustre* et *Signal*, comme *Lucid Synchrone* [16] qui traitent des flots de données comme *Lustre*, ou *ReactiveC* [11] et *ReactiveML* [70] dans la lignée d'*Esterel*. *Esterel*, *Lustre* et *Signal* appartiennent à la catégorie des langages synchrones. C'est-à-dire que les programmes écrits dans ces langages s'exécutent sous forme de réactions instantanées. Lorsqu'un ou plusieurs événements discrets se produisent, le programme donne son nouvel état immédiatement. Les événements discrets perçus et ceux que le programme produit, sont exprimés par des signaux reçus et émis. Ces langages, bien que très différents, offrent en général une malléabilité bien supérieure aux langages Grafcet et UML, mais aucun n'intègre simplement des technologies permettant d'interagir avec une audience. Comme nous l'avons vu dans l'état de l'art, les technologies du Web sont les candidats naturels dans ce domaine. Or il n'existe à notre connaissance qu'un seul langage combinant la programmation des automates et développement Web : le langage *HipHop.js* qui met en œuvre d'une nouvelle façon la sémantique du langage *Esterel* au sein de la programmation en JavaScript. C'est donc vers ce langage que nous nous sommes orientés.

Nous allons dans ce chapitre aborder les principaux principes de programmation avec HipHop.js puis nous pourrons démontrer en quoi ce langage est, pour nos orchestrations, nettement plus malléable qu'un langage généraliste du Web. Commençons par quelques principes de base de la programmation réactive synchrone.

## 5.1 Principe de la programmation réactive synchrone

Nous allons voir en quoi consiste la programmation réactive synchrone avec HipHop.js en utilisant des exemples musicaux qui n'ont pas d'importance pour la compréhension du langage, mais nous semblent appropriés dans notre contexte.

HipHop.js est un « langage dédié » (Domain Specific Language, DSL) qui met en œuvre une variante du langage *Esterel* sur une plateforme JavaScript. L'objectif de HipHop.js est de gérer simultanément des tâches synchrones et asynchrones. Un programme écrit en HipHop.js est compilé en JavaScript, en l'occurrence dans un environnement Hop.js. Ecrire un programme HipHop.js consiste à utiliser une syntaxe particulière à l'intérieur d'un programme JavaScript. Ceci rend son usage plus simple que les versions d'*Esterel* compatibles avec le langage C par exemple. Celles-ci nécessitaient d'intégrer « manuellement » l'automate issu de la compilation du programme Esterel dans l'environnement d'une application écrite en langage C. Ici donc tout se passe dans un environnement unique. Avant d'aborder le langage HipHop.js, il y a quelques principes à saisir.

- Le premier est qu'il faut raisonner comme si le programme HipHop.js se déroulait instantanément à chaque réaction, c'est pour cela que nous parlons de **programmation synchrone**. Si des instructions en HipHop.js sont « parallèles » ou « séquentielles », ce sera d'un point de vue logique et non pas selon l'horloge du processeur. Ceci est une vue abstraite du programme, en réalité l'exécution d'un programme HipHop.js consommera du temps processeur et ne se déroulera pas en temps nul. Mais nous devons raisonner comme si cela était le cas et aborder la question des performances comme un autre sujet, plus tard.
- Le deuxième principe de base est que HipHop.js fonctionne avec des **signaux** et non des variables. Cette notion de signal n'existe pas en JavaScript. Un signal correspond à un événement, il peut avoir une valeur. On peut l'émettre, le recevoir, l'attendre. La plupart des instructions HipHop.js traitent de signaux.

En vertu de ces deux principes de base, utiliser HipHop.js consiste à faire appel à un programme HipHop.js, que l'on appelle un **module réactif** en passant des signaux correspondant à des événements associés à des interactions avec des utilisateurs, des évènements réseaux, des timeouts, etc. L'appel d'un module HipHop.js et la réponse (en temps logique nul) qu'il apporte s'appelle une **réaction**. D'où le terme **programmation réactive** fréquemment employé pour la programmation Esterel/HipHop.js.

Ces principes ont pour vertu de permettre la définition d'instructions possédant une sémantique déterministe et donc permettant le développement de compilateurs efficaces et fiables, et surtout permettant de développer des programmes compréhensibles.

## 5.2 Programmer avec HipHop.js

### 5.2.1 Penser à l'architecture

La programmation HipHop.js consiste à écrire des modules qui seront activés par un programme JavaScript avec lequel il communique à l'aide de signaux. Les programmes Hiphops.js sont des sortes de boîtes noires, invoquées par un programme principal dans lequel ils sont encapsulés. Il est important de bien saisir qu'un programme HipHop.js ne fait rien tout seul. Ce que l'on lui demandera de traiter doit être défini dans un contexte global, avec une vision complète de l'architecture du logiciel à obtenir.

Un des points fort de HipHop.js est son intégration avec JavaScript. L'encapsulation de HipHop.js dans un programme permet d'écrire le programme principal et le module HipHop.js dans le même fichier source, et ainsi de pouvoir intégrer des commandes Javascript à l'intérieur d'un module HipHop.js. Cependant, la tâche qui consiste à cerner les problèmes que l'on peut résoudre avec HipHop.js avant de se lancer dans la programmation, demande un peu d'expérience. Penser un programme sous forme d'automates peut remettre en cause les habitudes d'un programmeur formé à des langages généralistes comme Javascript, Java, C++ ou Lisp.

### 5.2.2 HipHop.js et le temps : délai et instant

Nous avons beaucoup écrit sur le temps dans les précédents chapitres, et notamment sur la représentation du temps avec les langages synchrones. Or il y a un terme fréquemment utilisé par les programmeurs HipHop.js qui n'est pas toujours facile de saisir, c'est le terme de *délai*. Dans le langage commun, qui est celui des horloges, un délai est « le temps accordé pour faire quelque chose » (Dictionnaire Larousse). C'est aussi le temps (des horloges) compris entre deux événements. Le délai de livraison est le temps qui s'écoule entre un engagement de livrer une marchandise et la date effective de livraison par exemple. Or le terme délai en programmation HipHop.js n'a pas exactement ce sens. Il a un sens plus général qui rend possible une mise en œuvre de la notion de durée à la Bergson. Un délai en HipHop.js est un événement ou une conjonction d'évènements. Ces évènements peuvent relever du temps des horloges, mais pas nécessairement. Un délai peut être l'aboutissement d'un trajet, le résultat de l'action de quelqu'un dans une assistance… en fait tout ce qui peut s'exprimer au moyen d'un signal ou d'une combinaison logique de signaux. Nous sommes donc bien à un niveau d'abstraction au-dessus du temps des horloges, c'est ce qui rend HipHop.js apte à s'occuper des durées au sens de Bergson.

Nous nous sommes permis d'insister sur ce point car dans la littérature sur la programmation synchrone nous rencontrons fréquemment des explications faisant référence au temps, qui sont difficiles à comprendre si l'on ne fait pas abstraction du temps des horloges. Parmi ces explications complexes il y a la différence entre délai standard, délai immédiat et délai compté. Ce que l'on appelle un délai standard, est l'occurrence d'un événement qui se traduit par la valeur logique « vrai » d'un signal ou le résultat « vrai » de l'évaluation d'une expression logique JavaScript ou HipHop.js. HipHop.js sera à l'écoute de cet événement dès qu'une instruction le concernant apparaitra une première fois. Le programme sera à l'écoute, mais pour qu'il traite effectivement l'évènement une première fois, il faudra attendre une réaction de l'automate. En parler *esterellien* on dira « qu'un délai démarre quand une instruction temporelle qui le traite démarre, et il se finira dans un instant ultérieur ». La première réaction passée, le signal est traité à chaque réaction. Si l'on souhaite qu'un évènement soit traité dès la première réaction, sans attendre une réaction ultérieure de l'automate, il faudra le qualifier « d'immédiat ». Un délai compté est simple à comprendre, c'est le nombre de fois que l'occurrence d'un évènement a été comptée par une suite de réactions provoquées sur l'automate HipHop.js.

Un terme intéressant en parler *esterellien* est celui d'*instant*. L'instant de HipHop.js est un bon moyen de faciliter la mise en œuvre de la durée sous une forme inspirée de Bergson. Un instant, n'est pas une indication d'horloge mais bien une réaction provoquée sur l'automate HipHop.js qui peut provenir d'une conscience.

Nous pensons que cette petite mise au point permettra d'être plus à l'aise dans la lecture de la littérature sur Esterel ou HipHop.js, et notamment de rendre compréhensible une expression du type « Un délai immédiat peut s'écouler instantanément » qui dans le langage commun est difficilement compréhensible. En effet dans le langage commun, comment un délai peut-il être

immédiat ? Si immédiat signifie de « durée nulle », l'idée n'est pas facile à saisir car ce qui définit un délai c'est bien qu'il s'écoule un temps mesurable différent de zéro entre un début et une fin. Dans notre langage de tous les jours, s'il n'y a pas de temps qui s'écoule, il n'y a pas non plus de délai. Mais en fait cette phrase signifie que « l'occurrence d'un évènement, peut être prise en compte sans attendre que l'automate ait été activé une nouvelle fois ». On rencontre aussi l'expression « terminaison d'un délai » qui est l'équivalent de « l'occurrence d'un évènement ». Voyons à présent un premier exemple de programme HipHop.js.

### 5.2.3 Les instructions HipHop.js

Le langage HipHop.js et son utilisation sont décrits dans la Thèse de Colin Vidal sur la *Programmation web réactive* [87]. Depuis cette thèse, il y a eu des modifications mineures de la syntaxe. Les instructions sont actuellement en minuscules. Les expressions sur les signaux ont changé de forme, plutôt que `NOW(identifier)` nous avons `identifier.now`, de même pour `identifier.preval`, ou `identifier.val`. Il y a aussi quelques changements sur le positionnement de parenthèses au lieu de `AWAIT COUNT(3, NOW(s))`, nous avons `await count (3, s.now)`.

Si le compilateur HipHop.js a connu et connait encore des évolutions, la syntaxe est basée sur *Esterel* V5 et reste conforme à sa définition. Nous ne reprendrons donc pas ici la description complète faite par Colin Vidal dans sa thèse ou celle de Gérard Berry dans le document *The Esterel v5 Language Primer* [8]. Nous renvoyons le lecteur à ces deux ouvrages indispensables à tout programmeurs HipHop.js. Nous procéderons plutôt par exemples de programmes développés pour notre plateforme. Voici un extrait d'un programme de contrôle d'une orchestration. Dans un premier temps nous ne nous focaliserons pas sur l'aspect musical de ce programme que nous aborderons dans les chapitres suivants, mais sur le langage HipHop.js.

```
1.  hiphop module trajetModule3(in start, stop, tick, abletonON)
2.    implements ${nuceraInt.creationInterfaces(par.groupesDesSons[2])} {
3.      signal tempsNucera=0;
4.      loop {
5.          abort(stop.now) {
6.              await immediate (start.now);
7.              hop {
8.                  console.log("--Automate des possibles Nucera");
9.              }
10.             fork{
11.                 every immediate (tick.now) {
12.                     fork{
13.                         hop { gcs.setTickOnControler(tempsNucera.nowval); }
14.                     }par{
15.                         emit tempsNucera(tempsNucera.preval + 1);
16.                     }
17.                 }
18.             }par{
19.                 run usine(...);
20.                 run conscience(...);
21.                 run maladie(...);
22.                 run arretUsine(...);
23.                 run guerison(...);
24.             }
25.         }
26.         hop { console.log("--Arret d'Automate des possibles Nucera");
27.             ableton.cleanQueues();
28.             oscMidiLocal.convertAndActivateClipAbleton(300);
29.             // Commande d'arrêt global
30.         }
31.         emit tempsNucera(0);
```

```
32.     }
33. }
34. exports.trajetModule3 = trajetModule3;
```

La première ligne introduit le module dont le nom est `trajetModule3()`. Il est déclaré avec 4 signaux, `start`, `stop`, `tick` et `abletonON`. Le signal `start` est spécifiquement déclaré comme un signal entrant, les autres par défaut sont entrants et sortants, soit `inout`. La ligne 2 est un peu plus complexe, retenons pour le moment qu'il s'agit d'un autre moyen de déclarer des signaux traités par le module. En ligne 3, nous avons un signal local `tempsNucera` initialisé à 0. `Local` signifie que ce signal n'est pas vu en dehors du module. En ligne 4, nous rencontrons la première instruction du module : `loop`. Le corps du code délimité par les accolades suivant `loop` sera répété automatiquement lorsque le déroulement séquentiel des instructions du corps sera arrivé au bout. Le corps à l'intérieur des accolades correspondant à `loop{}` peut être interrompu `par` l'instruction en ligne 5, `abort`. Cette instruction agit sur un corps délimité par des accolades de la même manière que `loop`. La commande `abort`, interrompt le contenu de son corps quand le signal placé en paramètre est vrai, c'est-à-dire quand le signal `stop` a été émis par le programme principal et que celui-ci provoque une réaction. On évalue si le signal `stop` a été émis en utilisant la syntaxe `stop.now`. Avec la commande `abort`, la préemption est dite forte, c'est-à-dire que lorsque l'évènement `stop` se produit le corps du `abort` n'est pas exécuté. Si pour une raison quelconque nous avions souhaité que le corps du `abort` soit exécuté une dernière fois, lors de l'occurrence de l'évènement `stop`, nous aurions utilisé l'instruction `weakabort`. Dans ce cas nous disons que la préemption est faible.

La première instruction dans le corps du `abort` est `await`. Tant que le signal `start` n'a pas été reçu, cette instruction bloque le *thread* dans son déroulement séquentiel en ligne 6. `immediate` signifie que nous avons décidé de prendre en compte le signal `start` au moment même où nous rencontrons l'instruction `await`. Sans `immediate` ce n'est pas ce qui se produit. La première fois que dans un module HipHop.js nous rencontrons une instruction `await` sur un signal, le signal n'est pas évalué. Il le sera à chaque réaction suivante. Comme nous l'avons vu précédemment avec l'instruction `abort`, si nous souhaitons évaluer un signal quand nous le rencontrons une première fois dans un `await`, il faut le préciser avec l'instruction `immediate`.

En ligne 7 nous avons une instruction fort pratique, `hop{}`, qui permet d'insérer du code JavaScript directement à l'intérieur d'un module.

La ligne 10 introduit un des concepts phares de HipHop.js, le parallélisme. L'instruction `fork{}` va de pair avec l'instruction `par{}` pour « parallèle ». Chacune des instructions `fork` et `par` est suivie d'un corps entre accolade. Les deux corps sont exécutés en parallèle à chaque réaction. L'ensemble des instructions `fork` et `par` associées se termine quand l'ensemble des corps se terminent. Ici le `fork` de la ligne 10 se terminera quand nous aurons atteint la ligne 17 **et** la ligne 24.

En ligne 11, nous rencontrons l'instruction `every{}`. Le corps d'un `every` est sollicité à chaque occurrence de l'évènement en paramètre, ici `tick.now`. Comme pour `await`, nous devons ajouter `immediate` si nous souhaitons activer `every` dès la première fois que nous rencontrons `tick.now`.

La prochaine nouveauté est en ligne 15, avec la commande `emit`. Il s'agit de l'émission d'un signal, ici `tempsNucera` auquel nous assignons une valeur `tempsNucera.preval + 1`. L'émission d'un signal peut être vue comme une diffusion (broadcast) de ce signal à l'intérieur du programme HipHop.js. Ici, le signal `tempsNucera` n'est pas réutilisé à l'intérieur de ce module, mais il pourra l'être par d'autres sous-modules. Notons ici la référence à `preval` qui donne la valeur du signal obtenu

lors de la réaction précédente. La ligne 15, `emit tempsNucera(tempsNucera.preval + 1),` est une façon d'incrémenter un compteur.

De la ligne 19 à 23 nous avons affaire au déroulement séquentiel de cinq sous modules, déclarés selon le même procédé que celui que nous regardons. Les trois points sont une façon commode de signifier que nous passons en paramètre du module appelé tous les signaux de notre module. Quand nous arrivons en fin de notre premier `fork`, nous passons à la ligne 26 où tout s'enchaine dans la même réaction pour reprendre au début de `loop{}` en ligne 5. Notons que la ligne 34 est une instruction d'export purement JavaScript qui permet à des modules JavaScript autres que celui qui comprend ce programme, d'accéder à notre module HipHop.js. Cet exemple de module n'est pas exhaustif en termes d'instructions HipHop.js, mais il est déjà représentatif de la logique de fonctionnement de ce langage.

### 5.2.4 Activer un module depuis Javascript

Comme nous l'avons vu, l'exécution d'un module HipHop.js est fractionnée en une succession de réactions déclenchées par des événements externes associés aux interactions des utilisateurs, aux événements du réseau, aux temporisations, à des événements multimédias, etc. Un programme HipHop.js est organisé sous la forme d'un ou plusieurs modules qui sont chargés dans une machine réactive (ou machine en bref). Prenons un automate orchestral très simple comme celui-ci où nous avons des signaux d'entrée qui correspondraient à des actions de sélection d'un pattern par l'audience, `VioloncelleIN` et `BasseIN`, et où nous avons des signaux de sortie `VioloncelleOUT`, `BasseOUT` et `ViolonOUT` qui correspondent à des activations, c'est-à-dire la mise à disposition pour l'audience des patterns des violoncelles, basses et violons.

```
1.  "use hiphop";
2.  "use strict";
3.
4.  hiphop module orchestration(in VioloncelleIN, in BasseIN,
5.     out VioloncelleOUT, out BasseOUT, out ViolonOUT) {
6.     emit VioloncelleOUT(true);
7.     emit BasseOUT(true);
8.     emit ViolonOUT(false);
9.
10.    fork {
11.       await(VioloncelleIN.now);
12.    } par {
13.       await(BasseIN.now);
14.    }
15.    emit ViolonOUT(true);
16. }
17.
```

Le code source JavaScript suivant montre comment charger un module en machine, comment l'utiliser et comment établir une connexion entre HipHop.js et le reste de l'application Web.

```
1.  hiphop machine mach(orchestration);
2.  mach.addEventListener("VioloncelleOUT",
3.   evt => console.log("Violoncelle", evt.nowval));
4.  mach.addEventListener("BasseOUT", evt => console.log("Basse", evt.nowval));
5.  mach.addEventListener("ViolonOUT", evt => console.log("Violon", evt.nowval));
6.
7.  mach.react({ VioloncelleIN: 1 });
8.  mach.react({ VioloncelleIN: 0, BasseIN: 2 });
```

En ligne 1, la machine est chargée à partir du module HipHop.js. Dans les lignes 2 à 5, trois « listeners » sont attachés à cette machine. Il s'agit de fonctions Hop.js qui seront appelées après chaque réaction où les signaux `VioloncelleOUT`, `BasseOUT`, ou `ViolonOUT` seront émis. Dans les lignes 7 et 8 deux réactions sont exécutées. La première émet un signal `VioloncelleIN`, et la seconde émet les signaux `VioloncelleIN` et `BasseIN`. Ceci montre la façon traditionnelle dont Hop.js et HipHop.js interagissent. C'est bien Hop.js qui fait réagir la machine HipHop.js, par exemple après le clic d'un utilisateur ou une temporisation. HipHop.js déclenche les « listeners » de Hop.js à la fin de la réaction, avec l'émission du signal. C'est en ces échanges, entre les exécutions de Hop.js et HipHop.js, que consiste l'exécution globale d'un programme.

### 5.2.5 Préemption

La préemption est un concept fondamental de HipHop.js. Pour l'illustrer, modifions l'orchestration précédente pour que, comme auparavant, à chaque fois qu'au moins un violoncelle et une basse ont été tous deux sélectionnés une fois, le public se voit proposer des violons. Mais, après chaque percussion, nous désactiverons le violon. Un nouveau violoncelle et une nouvelle basse doivent être sélectionnés une seconde fois avant de rendre de nouveau le violon accessible par le public et donc à nouveau visible sur l'interface acteur. Cette nouvelle orchestration pourrait être mise en œuvre comme suit :

```
1.  "use hiphop";
2.  "use strict";
3.
4.  hiphop module orchestration(
5.   in VioloncelleIN, in BasseIN, in PercussionIN,
6.   out ViolonOUT, out VioloncelleOUT, BasseOUT) {
7.      emit BasseOUT(true);
8.      emit VioloncelleOUT(true);
9.
10.    do {
11.       emit ViolonOUT(false);
12.       fork {
13.          await(VioloncelleIN.now);
14.       } par {
15.          await(BasseIN.now);
16.       }
17.       emit ViolonOUT(true);
18.    } every(PercussionIN.now)
19. }
```

Les lignes 7 à 9 initient le contexte en activant la basse, le violoncelle et le violon. La construction `do/every` met en œuvre la préemption. Dans cette nouvelle version, `fork/par` s'exécute comme dans l'exemple de module d'orchestration précédent, mais en attendant le violoncelle ou la basse. Chaque fois qu'un événement de percussion se produit, la construction du `fork` est préemptée, c'est-à-dire qu'elle est stoppée quel que soit son état, et un nouveau violoncelle et une nouvelle basse sont de nouveau attendus.

### 5.2.6 Construction dynamique de programme

L'exécution d'un programme HipHop.js se fait en plusieurs étapes. Lors de la première phase d'exécution du programme JavaScript, le programme HipHop.js est prétraité et compilé en engendrant un circuit électronique [9]. La deuxième étape consiste à exécuter ce circuit, l'extension syntaxique implémentée dans Hop.js pour HipHop.js donne l'illusion d'un langage statique mais tous les

programmes HipHop.js sont bien le résultat d'une exécution JavaScript. La syntaxe HipHop.js supporte la création dynamique de programmes. Au cours de la phase de prétraitement, les expressions ${...} intégrées dans les programmes HipHop.js sont évaluées et le résultat de cette évaluation est inséré dans le programme. Par exemple, le module d'orchestration présenté précédemment pourrait être implémenté sous la forme :

```
1.  "use hiphop";
2.  "use strict";
3.
4.  const branche1 = hiphop await(VioloncelleIN.now);
5.  const branche2 = hiphop await(BasseIN.now);
6.
7.  hiphop module orchestration(
8.   in VioloncelleIN, in BasseIN, in PercussionIN,
9.   out ViolonOUT, out VioloncelleOUT, BasseOUT) {
10.     emit BasseOUT(true);
11.     emit VioloncelleOUT(true);
12.     emit ViolonOUT(false);
13.    fork {
14.       ${branche1}
15.    } par {
16.       ${branche2}
17.    }
18.    emit ViolonOUT(true);
19. }
```

Comme nous le verrons, cette fonctionnalité simplifie la programmation des modules qui ont de nombreux signaux d'entrée et de sortie pour représenter les groupes de patterns utilisés dans une orchestration.

### 5.2.7 Hypothèse instantanée

Les expressions HipHop.js sont évaluées instantanément, c'est-à-dire pendant la réaction. La conception de HipHop.js favorise la modularité car l'évaluation des expressions Hop.js n'introduit pas de pauses implicites qui peuvent désynchroniser les modules. Ceci impose une étape de compilation à l'avance pour calculer le graphe de dépendance du signal. Cette étape est nécessaire à la planification des instructions élémentaires pendant la réaction. Prenons, par exemple, le module suivant :

```
1.  hiphop module autreOrchestration(
2.     in PercussionIN, in ViolonIN,
3.     out ViolonOUT, VioloncelleOUT, PercussionOUT,
4.     inout cordes) {
5.     emit PercussionOUT(true);
6.     emit VioloncelleOUT(true);
7.     emit ViolonOUT(true);
8.
9.     fork {
10.       if (!PercussioIN.now) emit cordes;
11.    } par {
12.       if (cordes.now) emit VioloncelleOUT(false);
13.    }
14. }
```

Nous avons introduit ici une nouvelle instruction `if` qui permet d'évaluer une expression HipHop.js, de tester l'état d'un signal ou d'une combinaison de signaux au moment de la réaction. Sans une analyse préalable du programme, le prédicat `PercussionIN.now`, dans l'énoncé du `if` (ligne 10), ne peut être résolu qu'à la fin de la réaction car l'absence d'un événement ne peut être observée qu'après la réaction. C'est pourquoi les signaux `cordes` et `VioloncelleOUT` ne peuvent pas être émis instantanément. Le calcul de la dépendance a le bénéfice de permettre l'exécution de ce module en une seule réaction (ici il montre que puisque la percussion n'a pas d'émetteurs potentiels, elle est absente dans la réaction), mais il a aussi l'inconvénient de rejeter des programmes pour lesquels le graphe de dépendance contient des cycles, qui sont parfois des erreurs difficiles à corriger par le programmeur.

## 5.3 HipHop.js vs. JavaScript

Nous avons justifié l'utilisation de HipHop.js sur le principe qu'il s'agit d'un langage combinant programmation d'automate et programmation Web. Nous sommes partis pour cela du postulat qu'une orchestration ressemblait beaucoup à un automate. Mais il est naturel de nous demander si notre choix est effectivement justifié et si nous n'aurions pas pu tout simplement utiliser un langage généraliste du Web, c'est-à-dire JavaScript. Maintenant que nous avons vu les principes de base de la programmation synchrone, nous allons pouvoir étoffer notre argumentation en prenant un exemple simple de codage d'un scénario proche d'une *orchestration*. Nous allons programmer cet exemple en Javascript et en HipHop.js afin de comparer les deux approches.

Nous avons introduit l'orchestration au chapitre 4. Celle-ci consiste à permettre à l'audience de sélectionner des patterns à un moment donné, en fonction de certaines circonstances définies par le compositeur. Ainsi l'audience aura accès à certains patterns et pas à d'autres. Considérons que le compositeur ait défini des groupes de patterns, par exemple un groupe de patterns de violons, un groupe de patterns de violoncelles, un groupe de patterns de percussions. Peu nous importe le nombre de patterns dans chacun de ces groupes pour le moment. L'orchestration consistera à *activer* ou *désactiver* des groupes selon des événements. C'est-à-dire rendre accessible ou inaccessible à l'audience des groupes de patterns selon des règles définies préalablement par le compositeur. L'accessibilité à un groupe se traduira par une information sur les terminaux des individus dans l'audience, comme l'affichage d'une liste de patterns disponibles.

Le schéma suivant (Figure 10: Un exemple simple d'orchestration) est la spécification d'une orchestration très simple dans la lignée de ce que nous avons vu au chapitre 4. Nous avons trois groupes de patterns, *Violons*, *Violoncelles* et *Percussions* sur l'axe vertical. Sur l'axe horizontal, nous avons la durée, au sens de déroulement d'événements produits par des individus, ou en fonction du temps. Un triangle vers le bas signifie que le groupe est *rendu actif* pour l'audience, un rectangle vers le haut signifie que le groupe est *rendu inactif*. En reprenant quelques principes du chapitre 4, une flèche pleine signifie qu'un groupe est actif. Une ligne en pointillés représente un compteur d'événements, auquel correspond un signal se terminant par « IN ». « 5 x VioloncellesIN » sur la flèche du haut, signifie que nous attendons 5 occurrences d'un événement *VioloncellesIN* pour invalider le groupe Violoncelles à partir de la fin du déroulement de 5 *ticks* d'horloge.

Le *tick* d'horloge est un signal produit par la plateforme. Nous reviendrons sur ce signal dans le chapitre sur la mise en œuvre de l'orchestration Skini.

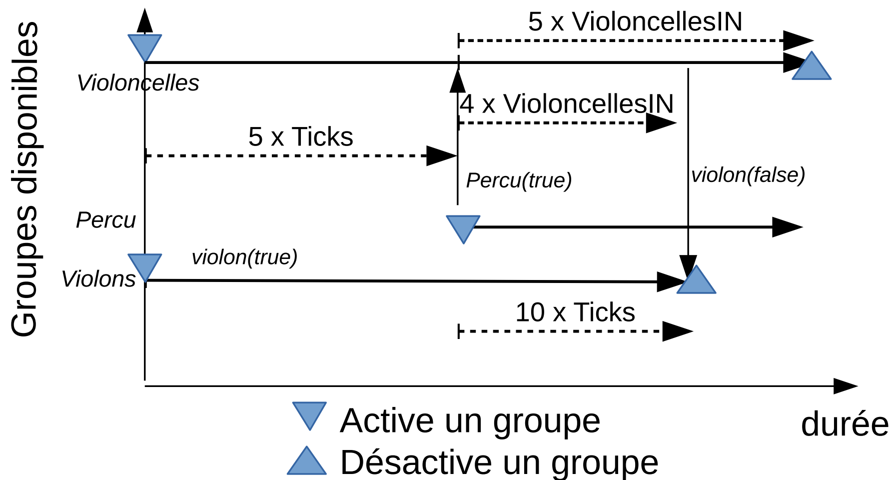



**Figure 10: Un exemple simple d'orchestration**

Un événement plus compliqué est celui de l'invalidation des violons, celui se produira au bout de 10 *ticks* comptés depuis le début des percussions ou lorsque nous aurons eu 4 sélections de violoncelle, soit 4 occurrences du signal *ViolincellesIN*, à la fin des 5 premiers *ticks*. Les percussions sont autorisées au bout de 5 ticks. Voici la programmation de ce scénario en HipHop.js, où nous traduisons la mise à disposition des groupes, leur autorisation, par l'`emit` d'un signal se terminant par « OUT » correspondant au groupe avec une valeur booléenne, vrai pour « autoriser », faux pour « invalider ».

```
1.  "use hiphop";
2.
3.  hiphop module orchestration(in tick,
4.      in PercussionIN, in ViolonsIN, in ViolincellsIN,
5.      out PercussionOUT, out ViolonsOUT, out ViolincellesOUT) {
6.      emit ViolonsOUT(true);
7.      emit ViolincellesOUT(true);
8.
9.      await count(5, tick.now);
10.     emit PercussionOUT(true);
11.     await (PercussionIN.now);
12.     fork {
13.         await count(5, ViolincellesIN.now);
14.         emit ViolincellesOUT(false);
15.     } par {
16.         await count(4, ViolincellesIN.now) || count(10, tick.now);
17.         emit ViolonsOUT(false);
18.     }
19. }
```

Nous retrouvons dans ce code les instructions que nous avions vues précédemment avec une nouveauté ligne 16 où l'instruction `await` peut se déclencher selon deux événements : 4 occurrences de `ViolincellesIN` ou 10 `ticks`. Notre spécification a donc été codée en une dizaine de lignes. Nous voyons qu'il serait simple de totalement modifier ce scénario en ajoutant ou déplaçant des instructions `await` ou `emit` de signaux.

Implémentons à présent ce scénario en JavaScript.

```
1.  let tick = 0;
2.  let percuState = false;
3.  let celloState = false;
4.  let celloCount = 0;
5.  let celloTick = 0;
6.
7.  function orchestration() {
8.     if (++tick === 5) {
9.        open("Percussion");
10.       percuState = true;
11.    }
12.    if (percuState && event("Percussion")) {
13.       celloState = true;
14.       percuState = false;
15.       celloCount = 0;
16.    }
17.    if (celloState && event("Cello")) {
18.       if (celloCount++ === 0)
19.      celloTick = tick;
20.       if (celloCount === 4
21.       || tick-celloTick === 10 ) {
22.      close("Violon");
23.       }
24.       if (celloCount === 5)
25.      close("Cello");
26.    }
27. }
28.
29. // Lance l'orchestration
30. open("Violon");
31. open("Cello");
32. setInterval(score, beat);
```

Nous avons choisi d'exprimer l'accessibilité à un groupe par une fonction `open(`*groupe*`)`, et l'inaccessibilité par une fonction `close(`*groupe*`)`. Ceci correspondrait à *`emit`* *groupe*`(true)` et `emit` *groupe*`(false)` de HipHop.js. Nous avons une fonction *event*`()` dont le rôle sera de vérifier l'état des commandes passées par l'audience. Comme nous nous intéressons seulement à l'automate ici, peu importe la manière dont nous aurons mis en œuvre cette fonction.

La programmation de cet exemple n'est pas compliquée, mais nous pouvons observer que même pour une orchestration aussi simple et brève, la fonction *orchestration*`()` et les variables sont déjà profondément enchevêtrées. La compréhension du programme n'est pas si simple. Nous pouvons aussi observer que changer l'orchestration exigerait probablement d'ajouter de nouvelles variables globales et de mettre à jour le corps entier de la fonction. Lorsque l'orchestration devient complexe (et certaines orchestrations sont très complexes), ceci devient irréalisable. La simplicité de la version HipHop.js parle d'elle-même. Dans la version HipHop.js, la distance entre l'expression artistique de l'orchestration et le programme est beaucoup plus courte qu'avec le programme JavaScript. Avec HipHop.js, toutes les lignes du programme sont directement liées aux actions de l'orchestration. Les imbrications de contraintes de l'orchestration sont directement associées aux constructions HipHop.js. Par exemple, il est évident et intuitif qu'après une percussion, le programme se termine après 5 `VioloncellesIN`. Ce n'est pas aussi visible dans la version JavaScript. Enfin, et probablement plus important encore, la simplicité du programme HipHop.js le rend beaucoup plus facile à modifier que son équivalent JavaScript.

Illustrons cette capacité à composer et à adapter les programmes HipHop.js existants à de nouveaux comportements avec un nouvel exemple. Supposons que le module d'orchestration ne doive pas durer plus de 30 *ticks* après un événement de percussion, mais qu'il puisse être réutilisé comme l'implémentation existante que nous allons simplement faire fonctionner sous le contrôle d'un autre module qui servira de superviseur. Nous utiliserons la construction d'échappement sur un label (ligne 7 ci-dessous) pour plafonner le nombre de ticks que le module d'orchestration exécute. En ligne 8, le module d'orchestration original (la syntaxe "..." signifie que les signaux de l'orchestration du module sont liés aux signaux des mêmes noms qui se trouve dans le module `OrchestrationLabel`) est exécuté en parallèle avec une branche qui attend le premier événement de percussion (ligne 10) et ensuite 30 ticks avant de terminer l'orchestration (ligne 12).

```
1.  "use hiphop";
2.
3.  hiphop module OrchestrationLabel(in tick,
4.     in PercussionIN, in ViolonsIN, in VioloncellsIN,
5.     out PercussionOUT, out ViolonsOUT, out ViolocellesOUT) {
6.
7.     end: fork {
8.        run orchestration(...);
9.     } par {
10.       await PercusionIN.now();
11.       await count(30, tick.now);
12.       break end;
13.    }
14.    emit ViolinsOUT(false);
15.    emit VioloncellesOUT(false);
16.    emit PercussionOUT(false);
17. }
```

Cette nouvelle version ne partage aucun élément d'implémentation avec la version originale. Il n'est pas nécessaire d'être au courant des détails de mise en œuvre du module Orchestration pour mettre en œuvre le nouveau module `OrchestrationLabel`. Il en serait tout autrement avec JavaScript. Cet exemple montre que les composants existants peuvent être réutilisés et assemblés dans des situations plus complexes impliquant la préemption et des exécutions parallèles, sans modifier la base de code existante. C'est un autre atout essentiel de HipHop.js. Si nous avions implémenté cette version avec JavaScript, nous aurions probablement réutilisé les variables globales `tick` et `percuState`, rendant dépendante l'une de l'autre l'implémentation des deux modules.

Maintenant que les principes de HipHop.js ont été exposés, et que son usage est motivé, nous allons pouvoir aborder la dimension musicale de l'expressivité de ce langage, en expliquant notamment le sens orchestral des signaux et modules que nous avons vus dans ce chapitre sous une forme abstraite. Mais nous devons avant cela expliquer dans quel contexte technique se met en œuvre une orchestration pour comprendre sur quoi agit l'orchestration.

# 6 REALISATION D’ORCHESTRATIONS PAR AUTOMATE

STRUCTURE DE LA PLATEFORME D’EXPERIMENTATION 69

LA MATRICE DES POSSIBLES 70
LES FILES D’ATTENTES 70
LA MISE EN ŒUVRE DU TEMPS DES HORLOGES 71

PROGRAMMATION DE PERFORMANCES 72

GESTION DES SELECTIONS DE PATTERNS 72
GROUPES REPETITIFS ET RESERVOIRS 73
CONFIGURATION DES PATTERNS ET GROUPES 76
L’USINE 78
GRAND LOUP 80
OPUS 1 82
OPUS 2 86
UNE AUTRE ORCHESTRATION DE L’OPUS1 88

CONCLUSIONS SUR LA PROGRAMMATION DES ORCHESTRATIONS 91

> En somme, l'auteur offre à l'interprète une œuvre à achever. Il ignore de quelle manière précise elle se réalisera, mais il sait qu'elle restera son œuvre ; au terme du dialogue interprétatif, se concrétisera une forme organisée par un autre, mais une forme dont il reste l'auteur. Son rôle consiste à proposer des possibilités déjà rationnelles, orientées et dotées de certaines exigences organiques qui déterminent leur développement. (L'œuvre ouverte, U. Eco, p.34)

A partir des principes présentés dans les chapitres précédents, nous pouvons à présent nous intéresser à des réalisations complètes et voir concrètement comment l'expressivité de HipHop.js est adaptée à la mise en œuvre d'orchestrations. Nous prendrons 4 exemples dans des contextes musicaux différents, en partant de l'orchestration la plus simple pour aller vers la plus complexe. Mais avant d'étudier des orchestrations complètes, nous devons aborder la structure de la plateforme et quelques principes importants concernant sa mise en œuvre.

## 6.1 Structure de la plateforme d'expérimentation

Le schéma, Figure 11: Vue fonctionnelle d'une performance donne une représentation logique de la plateforme qui va nous aider à en comprendre les principaux éléments. Nous avons :

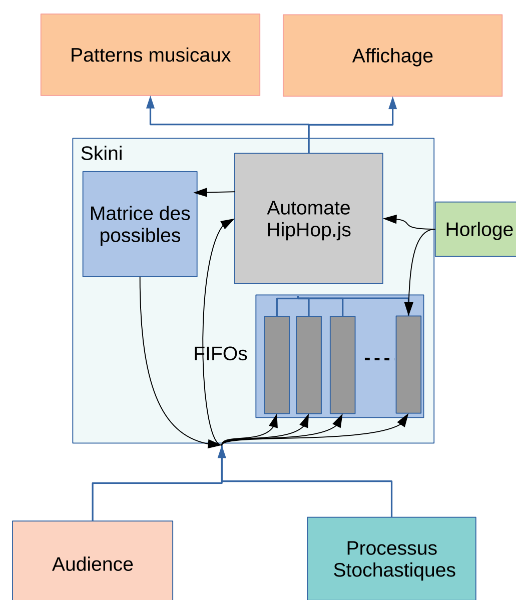



**Figure 11: Vue fonctionnelle d'une performance**

- Le logiciel Skini qui inclut : le dialogue avec l'audience ; *l'automate d'orchestration HipHop.js* qui nous intéresse plus particulièrement ; la *matrice des possibles* qui correspond à l'état de l'orchestration ; la *gestion de file d'attente* (FIFO) au paragraphe 6.1.2, qui permet notamment de garantir un certain niveau de qualité musicale ; et le contrôle de l'affichage.
- Les patterns musicaux qui sont les *objets* que Skini contrôle via l'orchestration et qui sont sélectionnés par l'audience avant d'être joués.
- L'audience qui donne vie à la performance en sélectionnant des patterns et en agissant sur le déroulement de la pièce de musique.
- Les processus stochastiques qui sont un moyen pour le compositeur d'évaluer son travail avant la performance, au moyen d'un outil dont le comportement correspond à celui d'une audience un peu particulière. Cet outil, le simulateur, fait l'objet d'un chapitre dédié.

- Une horloge, positionnée à l'extérieur de la plateforme logicielle comme un élément agissant sur l'orchestration, au même titre que l'audience ou des processus stochastiques.
- L'affichage qui concerne les différents moyens d'informer les acteurs de la performances, audience, compositeur, musiciens, etc.

Nous allons voir plus en détail les éléments de cette architecture qui permettent la mise en œuvre logicielle de la plateforme Skini, c'est à dire : la matrice des possibles ; les files d'attente ; la mise en œuvre du temps ; et la gestion de la sélection des patterns. La section traitant de la configuration des patterns et de groupes de patterns permettra de comprendre comment se font les liens entre les patterns, les groupes et les instruments qui jouerons les patterns.

### 6.1.1 La matrice des possibles

Pour mettre en œuvre une orchestration nous avons besoin de sa représentation informatique, mise à jour au fur et à mesure du déroulement de la pièce de musique. La représentation dynamique de l'orchestration est ce que nous appelons la *matrice des possibles*. Cette expression signifie que nous avons une matrice qui donne *l'état des groupes de patterns* (actif ou inactif, via des variables logiques vraies ou fausses). Ces groupes peuvent *potentiellement* être sollicités par l'audience. Il s'agit donc bien de la description d'opportunités offertes à l'audience. Nous avons une matrice, car sur un axe nous avons les groupes de patterns et sur un autre axe nous avons des groupes de personnes. En effet, Skini offre la possibilité de rendre disponibles des groupes de patterns à des *sous-groupes d'individus* dans l'audience sans que les autres sous-groupes y aient accès. Ceci est une fonctionnalité dont l'objectif et de permettre des scénarios variés, comme la définition de sous-groupes ayant des rôles différents dans le déroulement de la performance, ou même de mettre des sous-groupes en concurrence pour des performances imaginées sous forme de jeu. Techniquement, la gestion de l'orchestration revient à mettre à jour cette matrice en fonction du déroulement de la performance et des réactions du public. La partie HipHop.js de Skini agit essentiellement sur cette matrice en mettant à jour ses éléments en fonction des signaux entrants venant de l'interaction.

### 6.1.2 Les files d'attentes

Un des principes de base du système Skini est de considérer qu'un instrument n'est capable de jouer qu'un seul pattern à la fois. Ce principe a été fixé pour plusieurs raisons :

- Nous voulons que les membres de l'audience aient conscience de leur contribution. Il s'agit donc de mettre en œuvre toutes des sélections de l'audience de façon individualisée. Par conséquent si plusieurs patterns se superposent pour un même instrument, il devient difficile voire impossible de l'isoler pour celui qui l'a sélectionné.
- Le compositeur, qui exerce un contrôle de l'orchestration via HipHop.js, verrait sa tâche fortement compliquée s'il lui fallait prendre en compte non seulement les combinaisons de patterns, mais de plus leurs superpositions.
- Nous souhaitons que le compositeur contrôle l'instant où un pattern sera joué. Ce qui signifie qu'un pattern n'est pas joué au moment où il est sélectionné, mais suivant un processus de synchronisation qui peut introduire des délais.

Nous avons choisi de répondre à ces contraintes au moyen de files d'attente par instrument. La Figure 12: *Les files d'attente de patterns*, décrit le principe des files d'attente. Schématiquement, les patterns entrent dans chaque file par le bas et Skini joue de façon régulière la ligne du haut. Une fois qu'un pattern est lu, il est supprimé de la file. La file est décalée vers le haut et à la prochaine occurrence *d'un cycle de lecture* le processus recommence. Il s'agit d'un mécanisme *First In First*

*Out* (FIFO). Pendant ce processus de lecture, l'audience peut à tout moment ajouter des patterns dans les files. Ce mécanisme a la vertu de permettre une parfaite synchronisation entre les patterns puisqu'il est possible de contrôler leurs départs selon un même cycle.

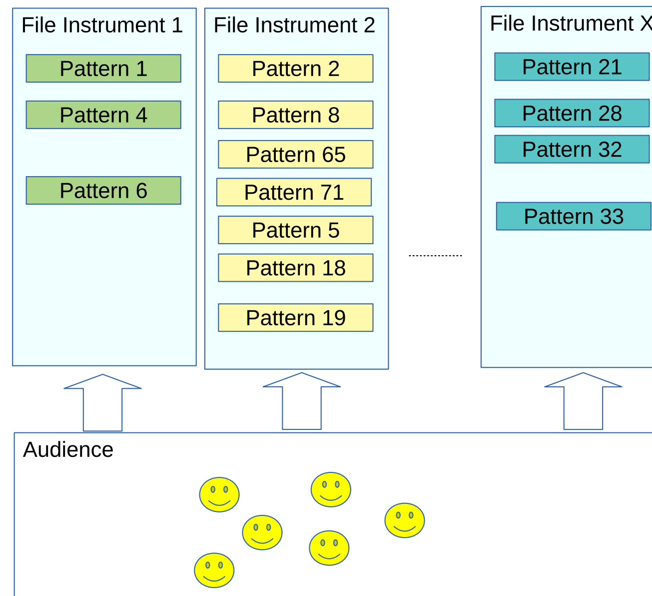



**Figure 12: Les files d'attente de patterns**

Les vertus de ces files d'attente sont donc de donner une bonne visibilité musicale et de garantir une bonne synchronisation, la contrepartie se traduit dans la difficulté de faire comprendre ce mécanisme à l'audience. Nous avons l'habitude que nos actions soient prises en compte instantanément sur nos terminaux. Il faudra donc informer de la prise en compte instantanée d'une sélection de pattern, et de son exécution potentiellement différée.

### 6.1.3 La mise en œuvre du temps des horloges

Un des éléments de contrôle de l'orchestration est le temps des horloges. Le temps des horloges peut d'une part être pris en compte par l'orchestration, nous parlerons « d'actions du temps sur l'orchestration » et d'autre part l'orchestration peut définir sa façon d'interpréter le temps, il s'agit « du contrôle de la mesure du temps par l'orchestration et les patterns ». Les actions du temps sur l'orchestration se font à l'aide de signaux d'associés à des horloges. Nous avons rencontré de tels signaux dans les exemples précédents. Ce sont les signaux *tick*[4]. La mise en œuvre du temps et donc l'émission de ces signaux est concrètement réalisée à partir du système qui produit le son. Il s'agit pour nous d'une *Digital Audio Workstation (DAW)*. Il pourrait s'agir d'un chef d'orchestre pour une utilisation de Skini avec des musiciens. De plus, le temps agit sur la performance en parallèle du déroulement de l'orchestration via une pulsation, liée au tempo, qui définit le rythme de lecture des files d'attente.

[4] Notons ici qu'il est tout à fait possible d'introduire d'autre type d'événements dans l'orchestration que ceux liés aux sélections de l'audience ou le temps. Dans la mesure où il sera possible d'interfacer une information venant d'un capteur quelconque avec HipHop.js et d'y assigner un signal. Cette information pourra être prise en compte par le compositeur. Nous avons par exemple utilisé notre système pour la réception de signaux MIDI venant d'une vidéo.

Ceci signifie que le temps des horloges a plusieurs rôles, que le compositeur et l'audience via l'orchestration pourront contrôler de différentes façons en fixant :

- un *tempo* global;
- une valeur (que nous nommons *timer division*) qui définit la longueur du *tick* en multiple de la pulsation ;
- pour *chaque pattern* sa longueur en fonction de la pulsation définit par le tempo. La durée d'un pattern définit le cycle de lecture du pattern suivant pour le même instrument.

Une audience peut modifier radicalement le comportement temporel au cours d'une performance via l'orchestration, en agissant sur le *tempo* et le paramètre *timer division.*

En ce qui concerne la mise en œuvre technique, prenons le cas le plus simple où la production du son est réalisée par une DAW. La méthode que nous avons utilisée consiste à utiliser le temps métronomique produit par la DAW, de façon à contrôler les instants d'émission du signal *tick*. Concrètement, Skini utilise l'horloge MIDI générée par la DAW pour émettre le signal *tick*. L'horloge MIDI a une fréquence de 24 pulsations par temps métronomique (24 ppqn, *pulse per quarter note*), qui peut apporter suffisamment de finesse dans le contrôle. Comme les durées des patterns dépendent de la même horloge MIDI, il est possible de contrôler le temps métronomique depuis la DAW durant le déroulement de l'orchestration, nous voyons donc que le contrôle du temps est globalement cohérent et contrôlable avec la précision d'une synchronisation MIDI.

## 6.2 Programmation de performances

### 6.2.1 Gestion des sélections de patterns

La sélection d'un pattern par un membre de l'audience se traduit par la réception d'un signal dans l'orchestration. Nous avons choisi de nommer ce signal du nom du groupe de patterns auquel appartient le pattern, suivi des caractères « IN ». Par défaut, l'orchestration reçoit des signaux « IN », au moment de la sélection et non pas au moment où le pattern se joue. En effet, entre la sélection et l'action de jouer un pattern, il y a un délai introduit par les files d'attente. Il est cependant possible de paramétrer le système pour que les signaux soient émis au moment où le pattern est joué et non pas au moment où le pattern sélectionné est mis en file d'attente. Cette différence de comportement du système aura un impact sur le déroulement de l'orchestration à cause des files d'attente. Supposons que nous conditionnions le passage à une étape suivante à la réception de X signaux IN pour un groupe. Dans le cas d'une émission des signaux IN à la sélection, la condition pourra être remplie avant que tous les patterns aient été joués, ce qui ne sera pas le cas dans l'émission des signaux au moment où les patterns sont joués. Les comportements de l'orchestration obtenus dans ces deux cas peuvent être très différents.

Par ailleurs, notons que Skini offre la possibilité de générer des signaux au *moment où un pattern particulier est joué*. Il est donc possible au-delà d'un travail sur les groupes, de traiter des signaux émis à l'instant où un pattern spécifique se joue. Le principe consiste à définir un groupe avec un seul pattern et en paramétrant le système pour recevoir un signal « IN » au moment où le pattern se joue. Nous pouvons obtenir le même résultat avec une DAW capable d'émettre un signal lors de l'activation d'un pattern. Le choix de l'une ou l'autre méthode est une affaire de goût.

Nous savons à présent comment sont déclarés les patterns et les groupes de patterns, en quoi consiste l'activation et la désactivation de groupes de patterns au moyen de signaux émis par HipHop.js qui vont agir sur la matrice des possibles. Nous savons comment gérer le temps des horloges,

comment HipHop.js est informé des sélections de l’audience et comment ces sélections sont traitées par Skini sous forme de files d’attente. Nous avons aussi vu une distinction subtile sur la prise en compte de la sélection, immédiate ou différée. Nous avons vu qu’il est donc possible de définir en complément du langage HipHop.js un ensemble d’éléments permettant la mise en œuvre des orchestrations.

### 6.2.2 Groupes répétitifs et réservoirs

Comme nous l’avons vu au chapitre 4, l’orchestration consiste à définir des groupes de patterns qui seront *activés* et *désactivés* par cette orchestration. L’*activation* consiste à signifier à l’audience à travers une interface Web accessible depuis des smartphones, des tablettes ou des ordinateurs quels sont les patterns disponibles. Les membres de l’audience peuvent alors *sélectionner* des patterns qui seront joués par des synthétiseurs ou des musiciens.

Il y a peu de contraintes pour la conception d’un groupe. Un groupe peut contenir un ou plusieurs patterns. On peut retrouver le même pattern dans plusieurs groupes. On peut mettre des patterns d’instruments différents dans un même groupe.

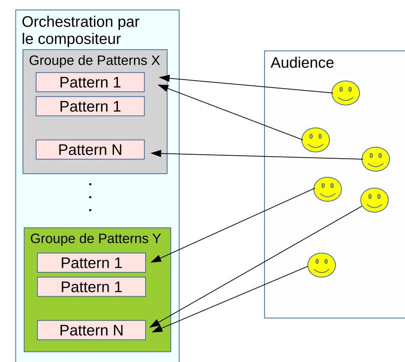



**Figure 13: Pattern et audience**

Nous avons introduit rapidement au chapitre 4, deux catégories de groupe : la catégorie *groupe répétitif* ou la catégorie *réservoir*. Comme nous l’avons vu dans ce chapitre, un groupe *répétitif* est un ensemble de patterns qui peuvent être sélectionnés à la demande de l’audience sans contrainte. Un *réservoir* est un ensemble de patterns où chaque pattern sélectionné est enlevé de l’ensemble. Dans un groupe répétitif, le même pattern peut être sélectionné plusieurs fois, dans un réservoir le même pattern ne peut être sélectionné qu’une seule fois. Il est aisé de saisir de façon intuitive l’intérêt de l’un ou l’autre modèle. Pour un groupe de patterns conséquent et utilisé pour maintenir une atmosphère, un groupe répétitif est bien adapté. Pour des patterns en petit nombre qui doivent avoir un rôle de ponctuation du discours, de type percussion explosive, dont le but et de provoquer une rupture de contexte par exemple, on préfèrera un réservoir pour éviter que les patterns se répètent et posent une atmosphère stable.

Nous avons expérimenté ces deux types de groupes, on peut en imaginer d’autres comme des réservoirs autorisant un nombre limité de répétitions par exemple. Ces groupes de patterns sont le matériel de base de l’orchestration. Comme nous l’avons vu, l’*activation* d’un groupe signifie qu’il devient accessible à l’audience et qu’avec une interface homme-machine adaptée, les individus de l’audience pourront demander l’activation d’un pattern appartenant à ce groupe. Les activations sont déclenchées en HipHop.js à l’aide des signaux se terminant par « OUT ». Nous en avons croisées dans les exemples précédents liés au langage HipHop.js.

Comme nous l’avons vu, l’art de l’orchestration est de définir quelles seront les règles d’activation et de désactivation des groupes de patterns. Pour définir ces règles, le compositeur devra tenir compte de ce que pourrait faire l’audience, donc *gérer* des durées au sens où ce qui définira le déroulement sera la perception de l’œuvre en cours par l’audience, donc les consciences qui constituent l’audience. Nous sommes dans une forme de durée proche de la définition de Bergson au service de l’œuvre ouverte d’Eco [23]. La principale façon de tenir compte du comportement de

l'audience que nous avons imaginée, est de prendre en compte l'émission de signaux liés aux groupes de patterns comme nous l'avons vu au chapitre 4. *Un signal correspondant à un groupe de patterns sera émis à chaque sélection par l'audience d'un pattern dans ce groupe*. La mise en œuvre se fait au moyen des signaux se terminant par « IN » dans les exemples précédents. Notons que si le compositeur tient explicitement à identifier un pattern particulier et non pas un groupe, il lui suffit de définir un groupe avec un seul pattern. Nous avons retenu l'idée de groupe car elle permet une bonne ouverture et n'est pas restrictive puisqu'un groupe peut être un singleton. Voyons comment ces groupes se mettent en œuvre avec HipHop.js.

#### *6.2.2.1 Le codage des groupes répétitifs*

Pour mettre en œuvre un « groupe répétitif » (que nommerons simplement « groupe » pour faire plus court) nous avons besoin de trois choses : la liste des patterns qui constitue ce groupe, un signal d'activation/désactivation, un signal correspondant à une demande de pattern de ce groupe par l'audience. Ces différents signaux sont construits à partir de fichiers de configuration associés à chaque pièce musicale. Rappelons-nous l'instruction `implements` vue dans nos exemples au chapitre 5.2, Programmer avec HipHop.js.

```
1. hiphop module trajetModule3(in start, stop, tick, abletonON)
2.   implements ${nuceraInt.creationInterfaces(par.groupesDesSons[2])} {
```

Cette instruction permet d'intégrer des signaux dans un module en suivant le modèle exposé dans la construction dynamique de programme. Voici le code de la fonction `creationInterfaces()` :

```
1. "use hiphop"
2. "use hopscript"
3.
4. const hh = require( "hiphop" );
5.
6.  function creationInterfaces(groupes) {
7.        return <hh.interface>
8.            ${ groupes.map(  function(k) {
9.                     return <hh.signal name=${ k[0] + "OUT"}
10.                            direction="out" value=${ [false, -1] }  />;
11.                }
12.            )}
13.             ${ groupes.map(  function(k) {
14.                     return <hh.signal name=${ k[0] + "IN"}
15.                              direction="in" value=${ [0] }  /> ;
16.                }
17.            )}
18.        </hh.interface>;
19. }
20. exports.creationInterfaces = creationInterfaces;
```

Nous utilisons dans cette fonction une syntaxe intermédiaire de HipHop.js, sous un format XML. La balise `interface` est un élément de syntaxe utilisée par l'instruction `implements` pour prendre en compte une liste de signaux. Nous voyons qu'à partir du tableau `groupes` nous créons, avec l'instruction JavaScript `map()` deux types de signaux pour chaque élément du tableau. Un signal :

*nomDuGroupe*`OUT([false, -1]);`

Où la valeur false, signifie que le groupe est par défaut inactif, -1 signifie qu'aucun groupe d'utilisateurs n'est concerné par ce signal par défaut. Et un signal :

`nomDuGroupeIN();`

qui sera reçu par l'orchestration à chaque fois que quelqu'un dans l'audience aura demandé un pattern du groupe *nomDuGroup*.

L'activation d'un groupe répétitif se traduira donc par une commande du type :

`emit groupeOUT([true, 255]);`

sa désactivation par :

`emit groupeOUT([false, 255]);`

le 255 signifie que tous les groupes sont concernés, nous aurions pu préciser un sous-groupe avec une valeur inférieure à 255.

#### 6.2.2.2 Le codage des réservoirs

La mise en place d'un réservoir suit le même processus qu'un groupe répétitif pour la création des signaux. La différence est dans sa gestion à l'intérieur de l'orchestration. Il s'agit à l'intérieur d'un réservoir de désactiver chaque pattern après sa sélection par l'audience. Un réservoir est donc techniquement un sur-ensemble de groupes répétitifs avec dans chaque groupe répétitif un seul pattern. Voici la fonction de création d'un réservoir :

```
1.  function makeReservoir(instrument) {
2.   return hiphop
3.     laTrappe: {
4.     abort immediate (stopReservoir.now) {
5.       // Début demake réservoir
6.
7.       ${instrument.map( function(val) {
8.           return  <hh.emit ${ val +  "OUT" }  value=${ [true,255] } />
9.         })
10.      }
11.      hop { gcs.informSelecteurOnMenuChange(255,instrument[0], true);      }
12.
13.      ${makeAwait(instrument)}
14.      // Fin naturelle du réservoir
15.      break  laTrappe;
16.    }
17.    ${instrument.map( function(val) {
18.          return  <hh.emit ${ val +  "OUT" }  value=${ [false,255] } />
19.        })
20.     }
21.     hop { gcs.informSelecteurOnMenuChange(255,instrument[0], false); }
22.     hop {
23.        // Abort du réservoir
24.        // et message la gestion des réservoirs dans l'affichage de la partition
25.        hop.broadcast('killTank', instrument[0] );
26.     }
27.   }
28. }
```

Cette fonction reçoit un tableau contenant la liste des instruments. Nous avons ici le principe de génération de code rencontré précédemment pour les interfaces avec le code HipHop.js dans son format XML. Nous retrouvons la syntaxe HipHop.js précédée de balises hh. Le `abort` de la ligne 4 introduit un moyen de stopper le déroulement d'un réservoir en cours de performance. Le module faisant appel au réservoir pourra à tout moment émettre le signal `stopReservoir` qui activera le

abort et mettra fin au réservoir. En ligne 25 nous voyons apparaitre une commande Hop.js, `hop.broadcast()`. Nous utilisons régulièrement cette commande lorsqu'il s'agit de diffuser une information vers plusieurs clients. Ici l'information diffusée concerne l'affichage de l'orchestration que nous verrons plus loin.

Voici le code de la fonction `makeAwait()` appelé en ligne 13 :

```
1. function  makeAwait(instrument) {
2.   return <hh.parallel>
3.     ${instrument.map( function(val) {
4.       var code =
5.         <hh.sequence>
6.           <hh.await ${ val +  "IN" } />
7.           <hh.emit ${ val +  "OUT" }  value=${ [false,255] } />
8.           <hh.atom apply=${function() {
9.               gcs.informSelecteurOnMenuChange(255, val, false); }
10.            } />
11.        </hh.sequence>
12.      return code;
13.      })
14.    }
15.  </hh.parallel>
16. }
```

Il s'agit une fois de plus de code généré dynamiquement qui met en parallèle des commandes `await` (ligne 6) sur des sélections de patterns et désactive avec `emit` (ligne 7) le « groupe » composé d'un seul élément qui correspond à ce pattern.

Nous avons rencontré plusieurs fois :

```
hop {gcs.informSelecteurOnMenuChange(255,instrument[0], true);}
```

C'est une commande nécessaire à la mise à jour de l'affichage sur les terminaux clients. Cette commande n'est pas lancée de façon automatique pour permettre au compositeur de contextualiser le message correspondant à l'activation d'un groupe de patterns.

### 6.2.3 Configuration des patterns et groupes

Nous avons vu comment programmer les groupes répétitifs et les réservoirs. Nous allons voir comment se font les liens entre patterns, instruments et groupes à l'aide d'un fichier de *configuration des patterns* et d'un fichier de *configuration générale des groupes de patterns*. Le compositeur devra créer ces fichiers pour chaque pièce avant de se lancer dans la phase de programmation de l'orchestration.

Voici un extrait d'un fichier de configuration des patterns (Tableau 1: Configuration des patterns) pour une pièce que nous verrons un peu plus loin.

| Index | Index stop | Flag | Nom du pattern | Nom du son | Instru-ment | Groupe | Longueur |
|---|---|---|---|---|---|---|---|
| 10 | 300 | 0 | PolyUsine1 | PolyUsine1 | 2 | 0 | 8 |
| 11 | 300 | 0 | PolyUsine2 | PolyUsine2 | 2 | 0 | 8 |
| 12 | 300 | 0 | PolyUsine3 | PolyUsine3 | 2 | 0 | 8 |
| 13 | 300 | 0 | PolyUsine4 | PolyUsine4 | 2 | 0 | 8 |
| 16 | 300 | 0 | FM8Usine1 | FM8Usine1 | 3 | 1 | 8 |
| 17 | 300 | 0 | FM8Usine2 | FM8Usine2 | 3 | 1 | 8 |
| 18 | 300 | 0 | FM8Usine3 | FM8Usine3 | 3 | 1 | 8 |

**Tableau 1: Configuration des patterns**

Chaque ligne décrit un pattern. Chaque pattern possède un index. Cet index sera converti par la plateforme Skini en une commande « note ON » au format MIDI qui sera envoyée via une passerelle à la DAW pour déclencher le départ du pattern, comme nous le verrons au chapitre 8. Notons que l'index n'est pas l'équivalent direct de la note MIDI qui sera envoyée à la DAW. L'index ne prend pas en compte la notion de canal MIDI, ni la notion de vélocité, il n'est pas limité à 128 comme une note MIDI. Un mécanisme de conversion transparent pour le compositeur traduit l'index en note et canal. L'« Index Stop » fonctionne comme l'index du pattern. Il permet d'arrêter le pattern. Cette valeur dépend de la mise en œuvre de la DAW. En général un pattern se termine de lui-même, mais cette commande peut être utilisée dans des situations particulières où le compositeur décide d'interrompre certains patterns en cours d'exécution. La colonne « Flag » est utilisée par le système en cours d'exécution, elle ne concerne pas le paramétrage avant le début de la pièce. La colonne « nom du pattern » parle d'elle-même. La colonne « nom du son » donne le nom du fichier son qui sera utilisé par le client pour écouter le pattern localement sur son terminal. Dans notre exemple nous avons les mêmes noms pour le son et le pattern, mais ce n'est pas obligatoire. Le nom du pattern sera utilisé pour l'affichage sur les terminaux de l'audience. La colonne longueur donne la longueur du pattern en pulsations métronomiques. Il est possible d'avoir des patterns de différentes durées. La colonne « instrument » permet de définir sur quel instrument le pattern doit être joué. Nous voyons ici que les quatre premiers patterns du tableau sont destinés au même instrument. La colonne groupe défini un index de groupe de patterns. A l'aide de cet index, et d'un autre fichier de configuration général que nous allons voir ci-dessous qui inclue les groupes de patterns, nous allons pouvoir faire le lien avec l'orchestration HipHop.js.

Le fichier de configuration générale qui traite des groupes de patterns est construit sur le modèle donné en exemple ci-dessous (Tableau 2: Configuration des groupes de patterns). Il s'agit d'un tableau de tableaux.

```
1.  [  // Nucera
2.    ["PolyUsine", 0, "group", 15, 100, 17, [103,120,132], []  ],
3.    ["FM8Usine", 1, "group", 130, 100, 17, [103,120,132], [0] ],
4.    ["MassiveUsine", 2, "group", 270, 100, 17, [103,120,132], [1] ],
5.    ["PrismUsine", 3, "group", 400, 100, 17, [103,120,132], [2] ],
6.    ["LPSUsine", 4, "group", 540, 100, 17, [103,120,132], [3] ],
7.    ["AbsynthUsine", 5, "group", 666, 100, 17, [103,120,132], [4] ],
8.    ["LoopUsine", 6, "group", 793, 100, 17, [103,120,132], [5] ],
9.
10.   ["PolyConscience", 7, "group", 15, 240, 17, [103,120,132], [] ],
11.   ["FM8Conscience", 8, "group", 130, 240, 17, [103,120,132], [7] ],
12.   ["MassiveConscience", 9, "group", 270, 240, 17, [103,120,132], [8] ],
13.   ["PrismConscience", 10, "group", 400, 240, 17, [103,120,132], [9] ],
14.   ["AbsynthConscience", 11, "group", 540, 240, 17, [103,120,132], [10] ],
15.
16.   ["PolyMaladie", 12, "group", 15, 370, 17, [103,120,132], [] ],
17.   ["FM8Maladie", 13, "group", 130, 370, 17, [103,120,132], [12] ],
18.   ["MassiveMaladie", 14, "group", 270, 370, 17, [103,120,132], [13] ],
19.   ["PrismMaladie", 15, "group", 400, 370, 17, [103,120,132], [14] ],
20.   ["GiantMaladie", 16, "group", 540, 370, 17, [103,120,132], [15] ],
21.   ["AbsynthMaladie", 17, "group", 666, 370, 17, [103,120,132], [16] ],
22.
23.   ["MassiveArret", 18, "group", 15, 510, 17, [103,120,132], [] ],
24.   ["LPSArret", 19, "group", 130, 510, 17, [103,120,132], [18] ],
25.   ["AbsynthArret", 20, "group", 270, 510, 17, [103,120,132], [19] ],
26.
27.   ["PolyGuerison", 21, "group", 15, 630, 17, [103,120,132], [] ],
28.   ["FM8Guerison", 22, "group", 130, 630, 17, [103,120,132], [21] ],
29.   ["MassiveGuerison", 23, "group", 250, 630, 17, [103,120,132], [22] ],
```

```
30.   ["PrismGuerison", 24, "group", 380, 630, 17, [103,120,132], [23] ],
31.   ["LPSGuerison", 25, "group", 500, 630, 17, [103,120,132], [24] ],
32.   ["GiantGuerison", 26, "group", 620, 630, 17, [103,120,132], [25] ],
33.   ["AbsynthGuerison", 27, "group", 740, 630, 17, [103,120,132], [26] ],
34.   ["LoopGuerison", 28, "group", 873, 630, 17, [103,120,132], [27] ]
35. ]
```

**Tableau 2: Configuration des groupes de patterns**

Nous avons ici pour chaque ligne la description d'un groupe. Le premier élément de la ligne donne le nom du groupe. L'index du groupe vu ci-dessus dans le fichier de configuration des patterns correspond au deuxième élément d'une ligne. Nous voyons que le pattern PolyUsine1 appartient au groupe 0 dans le fichier de configuration des patterns. Nous retrouvons ce 0 dans le deuxième champ d'une ligne du tableau. Nous voyons en ligne 2, où le premier champ est la chaine de caractère « PolyUsine », dans le deuxième champ le numéro du groupe, ici 0. C'est à partir du premier champ que sont créés les signaux associés à un groupe de patterns OUT et IN vus précédemment dans le chapitre sur les principes de l'orchestration. Les autres informations sont relatives à l'affichage, nous y reviendrons dans le chapitre qui traite des interfaces avec l'audience. Elles sont sans importance pour l'orchestration.

### 6.2.4 L'usine

Passons à présent à des réalisations complètes. La pièce interactive « L'Usine » a été réalisée avec des Lycéens du Collège Nucéra à Nice, dans le cadre d'un projet supporté par le CIRM [107]. Il s'agit d'une commande d'œuvre faite par la SACEM dans le cadre des projets « Fabriques à Musique ». Les cent patterns de cette pièce ont été conçus par les lycéens avec un outil qui sera présenté plus tard dans ce document, le séquenceur distribué. Nous nous intéressons ici à l'orchestration réalisée à partir de ces patterns. Comme base de l'orchestration nous avons la trame de l'histoire que les lycéens ont inventée comme fil conducteur de leur pièce de musique. Il s'agit d'un scénario futuriste, où le patron d'une usine polluante, qui se soucie peu de son impact sur l'environnement, fait face à la maladie de son fils, maladie provoquée par la pollution. Cette crise provoque une crise de conscience de ce patron qui décide d'arrêter son usine. L'histoire se décompose donc en plusieurs scènes :

- Le contexte de l'usine, le côté futuriste, la pollution,
- La conscience qui tente de parler au patron,
- La maladie du fils qui apparait,
- L'arrêt de l'usine,
- La guérison du fils.

La pièce est disponible à l'adresse : https://soundcloud.com/user-651713160/lusine-mai-2019. Voici le code du module principal, c'est-à-dire celui qui décrit le déroulement global de la pièce.

```
1. hiphop module trajetModule3(in start, stop, tick, abletonON)
2.   implements ${nuceraInt.creationInterfaces(par.groupesDesSons[2])}{
3.     signal tempsNucera=0;
4.     loop {
5.       abort(stop.now) {
6.          await immediate (start.now);
7.          hop {
8.            console.log("--Automate des possibles Nucera");
9.          }
10.         fork{
11.            every immediate (tick.now) {
```

```
12.                 fork{
13.                   hop { gcs.setTickOnControler(tempsNucera.nowval); }
14.                 }par{
15.                   emit tempsNucera(tempsNucera.preval + 1);
16.                 }
17.               }
18.             }par{
19.               run usine(...);
20.               run conscience(...);
21.               run maladie(...);
22.               run arretUsine(...);
23.               run guerison(...);
24.             }
25.           }
26.           hop { console.log("--Arret d'Automate des possibles Nucera");
27.           ableton.cleanQueues();
28.           oscMidiLocal.convertAndActivateClipAbleton(300);//Commande d'arrêt global
29.        }
30.        emit tempsNucera(0);
31.      }
32. }
33. exports.trajetModule3 = trajetModule3;
```

Le nom de ce module, `trajetModule3`, n'a pas grande importance ici, c'est un nom générique qui correspond à une position dans une interface client destinée au « maître du jeu », le chef d'orchestre. Le code est globalement le même que ceux que nous avons déjà rencontrés. Nous avons vue toutes les commandes HipHop.js. Notons ici la présence du signal `tempsNucera.` Il s'agit d'un signal utilisé pour donner des informations sur le déroulement de la pièce en fonction du tempo. Ce n'est pas une information d'horloge puisqu'il est possible de modifier le tempo au cours de la pièce. Ceci se rapproche d'un décompte de mesures. Cette information est diffusée en ligne 14 via une commande JavaScript.

Ce qui concerne l'orchestration se trouve des lignes 19 à 23. Nous avons le descriptif séquentiel des 5 scènes de la narration. Prenons à titre d'exemple le premier module `usine()`, les autres modules sont structurellement très proches, car l'objectif de l'orchestration de cette pièce est de donner un cadre au 24 lycéens qui, en l'occurrence, sont les seuls à avoir accès aux patterns.

```
1.  hiphop module usine(tick)
2.  implements ${nuceraInt.creationInterfaces(par.groupesDesSons[2])}  {
3.
4.      hop { console.log("-- USINE --"); }
5.      emit FM8UsineOUT([true, 255]);
6.      hop { gcs.informSelecteurOnMenuChange(255, "FM8Usine", true); }
7.
8.      await  count(10, tick.now);
9.      emit PrismUsineOUT([true, 255]);
10.     hop { gcs.informSelecteurOnMenuChange(255, "PrismUsine", true); }
11.
12.     await  count(3, PrismUsineIN.now);
13.     emit LPSUsineOUT([true, 255]);
14.     hop { gcs.informSelecteurOnMenuChange(255, "LPSUsine", true); }
15.
16.     await  count(4, LPSUsineIN.now);
17.     emit PolyUsineOUT([true, 255]);
18.     hop { gcs.informSelecteurOnMenuChange(255, "PolyUsine", true); }
19.
20.     await  count(4, tick.now);
21.     emit MassiveUsineOUT([true, 255]);
22.     emit PrismUsineOUT([false, 255]);
```

```
23.     hop { gcs.informSelecteurOnMenuChange(255, "MassiveUsine", true); }
24.
25.     await  count(4, MassiveUsineIN.now);
26.     emit AbsynthUsineOUT([true, 255]);
27.     emit FM8UsineOUT([false, 255]);
28.     hop { gcs.informSelecteurOnMenuChange(255, "MassiveUsine", true); }
29.
30.     await  count(4, tick.now);
31.     emit LPSUsineOUT([false, 255]);
32.     emit LoopUsineOUT([true, 255]);
33.     hop { gcs.informSelecteurOnMenuChange(255, "LoopUsine", true); }
34.
35.     await  count(5, tick.now);
36.     emit LoopUsineOUT([false, 255]);
37.     emit PolyUsineOUT([false, 255]);
38.     emit MassiveUsineOUT([false, 255]);
39.     emit AbsynthUsineOUT([false, 255]);
40.     hop {
41.             gcs.informSelecteurOnMenuChange(255, "Usine", false);
42.             console.log("--Fin Usine");
43.     }
44. }
```

Nous voyons que ce module se déroule selon une séquence d'instructions `await`. Nous avons un premier type `d'await` sur le temps avec par exemple les lignes 8, 20 ou 30. Nous avons un deuxième type `d'await` sur des évènements comme en ligne 12, 16 ou 25. Les instructions `await` sur les `ticks` sont une façon de garantir un délai en fonction du tempo, les autres conditionnent le déroulement de l'évolution du discours musical. On retrouve régulièrement la commande `gcs.informSelecteurOnMenuChange()` qui met à jour l'affichage des clients.

L'orchestration ici ne fait appel qu'à des groupes répétitifs de patterns, nous n'avons pas de réservoir. Le déroulement est séquentiel selon le temps ou des événements. Il s'agit de la forme d'orchestration la plus simple qui soit.

### 6.2.5 Grand Loup

Nous avons déjà évoqué cette pièce dans le chapitre sur l'émulation d'une pièce Jazz. Ce précédent chapitre avait pour objectif de démontrer le principe d'une composition par patterns, nous n'avions pas abordé la dimension orchestrale, ni interactive de ce projet. Voici le code du module principal :

```
1.  hiphop module trajetModule3(in start, stop, tick, abletonON)
2.  implements ${grandloupInt.creationInterfaces(par.groupesDesSons[2])} {
3.      signal tempsGrandloup=0;
4.      loop {
5.          abort(stop.now) {
6.              await immediate (start.now);
7.              hop { console.log("--Démarrage automate des possibles");}
8.              emit GLguitareOUT([true, 255]);
9.              emit GLpercuOUT([true, 255]);
10.             emit GLsaxoOUT([true, 255]);
11.             fork{
12.                every immediate (tick.now){
13.                   fork{
14.                      hop {gcs.setTickOnControler(tempsGrandloup.nowval);}
15.                   }par{
16.                      emit tempsGrandloup(tempsGrandloup.preval + 1);
17.                   }
18.                }
19.
```

```
20.                 /****************************************
21.                         SEQUENCEUR D'ORCHESTRATION
22.                  ****************************************/
23.                  }par{
24.                     if      (GLguitareIN.now) { run trajet1( ... );}
25.                     else if (GLpercuIN.now) { run trajet2( ... );}
26.                     else if (GLsaxoIN.now)  { run trajet3( ... );}
27.                     run trajetFinal( ... );
28.                     hop {
29.                        console.log("-- Fin automate grandloup");
30.                        ableton.cleanQueues();
31.                       }
32.                   }par{
33.                        run harmony(...);
34.                   }
35.         } // ABORT
36.         hop {
37.             console.log("--Arret d'Automate grandloup");
38.             ableton.cleanQueues();
39.         }
40.         emit tempsGrandloup(0);
41.     } // LOOP
42. }
43. exports.trajetModule3 = trajetModule3;
```

Il est globalement proche de celui de « l'Usine », comme différences majeures nous avons les trois instructions `emit` des lignes 8, 9 et 10 qui vont de pair avec les 3 tests lignes 24, 25 et 26. Ceci est une façon de lancer une des 3 orchestrations trajet1, trajet2 ou trajet3 en fonction du premier pattern sélectionné par l'audience. Ceci est un exemple d'action de l'audience qui peut avoir un impact majeur sur le déroulement d'une pièce. Une autre différence majeure est le module `harmony()` en ligne 33. Ce module tourne en parallèle du déroulement des trajets. Il pilote le déroulement d'une structure harmonique qui modifiera les patterns selon les mécanismes de transpositions que nous avons vus au chapitre sur la composition par patterns. Voici le code du module `harmony()` :

```
1.  hiphop module harmony(tick) {
2.      loop {
3.          hop {
4.              transposeAll(0);
5.              mixolydienSax(false);
6.          }
7.          await count(4, tick.now);
8.
9.          hop { transposeAll(-2); }
10.         await count(4, tick.now);
11.
12.         hop {
13.             transposeAll(0);
14.             mixolydienSax(true);
15.         }
16.         await count(4, tick.now);
17.
18.         hop { transposeAll(2); }
19.         await count(4, tick.now);
20.     }
21. }
```

Nous sommes dans le cas d'une grille très simple puisqu'il s'agit de changer de contexte harmonique tous les 4 ticks. La ligne 14 active la conversion de la mélodie du pattern selon le mode

mixolydien. La fonction de transposition (lignes 4, 9, 13 et 18) pilote la transposition en demi-tons. Nous voyons que nous passons alternativement de la tonalité de base, à un ton en dessous avec un retour à la tonalité de base pour passer un ton au-dessus. Ce schéma se déroulera quelques soient les actions de l'audience.

Voici un exemple de « trajet » appelé par `trajetModule3` :

```
1. hiphop module trajet1(tick)
2. implements ${grandloupInt.creationInterfaces(par.groupesDesSons[2])}  {
3.
4.     hop { console.log("-- DEBUT TRAJET 1 --"); }
5.     await count (4, GLguitareIN.now);
6.     emit GLpianoOUT([true, 255]);
7.     hop { gcs.informSelecteurOnMenuChange(255,"Piano",true); }
8.
9.     await count (2, GLpianoIN.now);
10.    emit GLsaxoOUT([true, 255]);
11.    hop { gcs.informSelecteurOnMenuChange(255,"Saxo", true); }
12.
13.    await count(10, GLsaxoIN.now);
14.    emit GLsaxoOUT([false, 255]);
15.    hop { gcs.informSelecteurOnMenuChange(255,"Saxo",false); }
16.
17.    await count(2, tick.now);
18.    emit GLtrompetteOUT([true, 255]);
19.    emit GLbasseOUT([true, 255]);
20.    hop { gcs.informSelecteurOnMenuChange(255,"Trompette",true); }
21.
22.    await count(10, GLtrompetteIN.now);
23.    emit GLtrompetteOUT([false, 255]);
24.    hop { gcs.informSelecteurOnMenuChange(255,"Trompette", false); }
25.
26.    hop { console.log("-- FIN TRAJET 1 --"); }
27. }
```

Nous voyons qu'il s'agit du déroulement séquentiel d'un scénario similaire à ceux du projet « L'Usine ».

### 6.2.6 Opus 1

Voici à présent un exemple de pièce qui utilise de façon assez intensive les réservoirs. Il s'agit d'une pièce qui se rapproche de la description donnée avec la Figure 7: Une orchestration avec sessions. Cette pièce est composée de 270 patterns et 72 groupes de patterns. Une simulation de cette pièce est audible depuis l'adresse https://soundcloud.com/user-651713160/opus1-skini-v2-chrom.

Le module principal, ci-dessous est le même que pour les exemples précédents, nous avons seulement ajouté un nouveau module intermédiaire en ligne 19.

```
1. hiphop module trajetModule1(in start, stop, tick, abletonON)
2. implements ${opus1Int.creationInterfaces(par.groupesDesSons[0])} {
3.   signal temps=0, size;
4.   loop {
5.       abort(stop.now) {
6.           await immediate (start.now);
7.           hop {  console.log("--Démarrage automate des possibles Opus 1");}
8.           fork {
9.               every immediate (tick.now){
10.                  fork{
11.                      emit temps(temps.preval + 1);
```

```
12.                   }par{
13.                       hop { // Pour suivre le temps sur le controleur
14.                       gcs.setTickOnControler(temps.nowval);
15.                       }
16.                   }
17.               }
18.           }par{
19.               run journey( ... );
20.           }
21.       }
22.       hop {
23.           console.log("--Arret d'Automate Opus 1");
24.           ableton.cleanQueues();
25.       }
26.       emit temps(0);
27.   }
28. }
29. exports.trajetModule1 = trajetModule1;
```

Voici le module journey() qui décrit le comportement global de la pièce :

```
1.  hiphop module journey(tick)
2.  implements ${opus1Int.creationInterfaces(par.groupesDesSons[0])}  {
3.      signal choixHasard =0, theEnd;
4.      hop { console.log("-- DEBUT JOURNEY --"); }
5.
6.      loop {
7.          hop {hop.broadcast('removeSceneScore', 1 );}
8.          hop {hop.broadcast('removeSceneScore', 2 );}
9.          hop {hop.broadcast('removeSceneScore', 3 );}
10.
11.         hop {hop.broadcast('refreshSceneScore');}
12.         hop {hop.broadcast('addSceneScore', 5 );}
13.         hop {hop.broadcast('alertInfoScoreON', "Messiaen, Britten ou Shosta-
    kovitch");}
14.
15.         emit MessiaenOUT([true, 255]);
16.         emit ShostakovichOUT([true, 255]);
17.         emit BrittenOUT([true, 255]);
18.
19.         weakabort(theEnd.now){
20.             fork{
21.                 await count ( 2, ShostakovichIN.now);
22.                 hop {hop.broadcast('alertInfoScoreOFF');}
23.                 hop {hop.broadcast('addSceneScore', 2 );}
24.                 emit ShostakovichOUT([false, 255]);
25.                 emit MessiaenOUT([false, 255]);
26.                 emit BrittenOUT([false, 255]);
27.                 run sessionChromatique(...);
28.                 run sessionEchelle(...);
29.                 run sessionTonale(...);
30.                 emit theEnd();
31.             }par{
32.                 await count ( 2, MessiaenIN.now);
33.                 hop {hop.broadcast('alertInfoScoreOFF');}
34.                 hop {hop.broadcast('addSceneScore', 1 );}
35.                 emit ShostakovichOUT([false, 255]);
36.                 emit MessiaenOUT([false, 255]);
37.                 emit BrittenOUT([false, 255]);
38.                 run sessionEchelle(...);
39.                 run sessionChromatique(...);
40.                 run sessionTonale(...);
```

```
41.                 emit theEnd();
42.             }par{
43.                 await count ( 2, BrittenIN.now);
44.                 hop {hop.broadcast('alertInfoScoreOFF');}
45.                 hop {hop.broadcast('addSceneScore', 3 );}
46.                 emit ShostakovichOUT([false, 255]);
47.                 emit MessiaenOUT([false, 255]);
48.                 emit BrittenOUT([false, 255]);
49.                 run sessionTonale(...);
50.                 run sessionChromatique(...);
51.                 run sessionEchelle(...);
52.                 emit theEnd();
53.             }
54.         }
55.         hop {console.log("-- FIN JOURNEY --");}
56.         hop {hop.broadcast('alertInfoScoreON', "FIN");}
57.     }
58. }
```

Sur les lignes 7 à 12 nous avons des *broadcasts* Hop.js en JavaScript destinés au contrôle de l'affichage de la partition sur un grand écran. Nous verrons ce type d'affichage quand nous traiterons la question des interfaces homme-machine. Sur les lignes 15 à 17 nous avons un type de message que nous n'avons pas encore rencontré. Il s'agit de messages dont le rôle est non plus de mettre à disposition des groupes de patterns, mais de présenter des choix possibles qui seront proposés à l'audience avec la même interface que celle utilisée pour la sélection des patterns. Ces choix concernent des décisions sur des orientations possibles dans le déroulement de la partition, comme de choisir une session, l'activation de tel ou tel groupe, etc. Nous avons un exemple d'affichage d'un choix sur la partition en grand écran en ligne 13. Le `weakabort` de la ligne 19 encadre plusieurs déroulements possibles de la pièce en fonction des choix proposés avec les émissions de signaux des lignes 15 à 17. Quand deux sélections se sont produites sur l'un des signaux `ShostakovichIN`, `MessiaenIN` ou `BrittenIN`, nous basculons dans des environnements différents. Tout d'abord en adaptant l'affichage au choix (lignes 22, 23, 33, 34, 44 et 45) puis en désactivant les choix précédents (lignes 24 à 26, 35 à 37 et 46 à 48). Nous avons par la suite, trois déroulements différents de trois sessions (lignes 27 à 29, 38 à 40, 49 à 51). Le corps du `weakabort` est détruit en fin de séquence des sessions avec le signal `theEnd`. Ce module est un exemple de la façon dont on peut laisser à l'audience le choix du déroulement macroscopique d'une pièce. Il s'agit d'un mécanisme de vote, très simple à mettre en œuvre avec HopHop.js.

Prenons à présent l'exemple d'une session faisant appel aux réservoirs :

```
1.  hiphop module sessionEchelle(tick)
2.  implements ${opus1Int.creationInterfaces(par.groupesDesSons[0]) } {
3.    signal stopReservoirsEchelle, stopCuivreEchelle, tickEchelle =0;
4.    laTrappe:{
5.        fork{
6.          every(tick.now) {
7.             emit tickEchelle();
8.          }
9.        }par{
10.       hop { console.log("-- DEBUT SESSION ECHELLE --"); }
11.       emit cellosEchelleOUT([true, 255]);
12.       hop { gcs.informSelecteurOnMenuChange(255,"cellosEchelle", true); }
13.
14.       await count (4, cellosEchelleIN.now);
15.       emit violonsEchelleOUT([true, 255]);
16.       emit altosEchelleOUT([true, 255]);
```

```
17.       emit ctrebassesEchelleOUT([true, 255]);
18.       emit bassonsEchelleOUT([true, 255]);
19.       hop { gcs.informSelecteurOnMenuChange(255,"violonsEchelle", true); }
20.
21.       weakabort(stopReservoirsEchelle.now) {
22.         fork {
23.           fork{
24.             run resevoirTrompettesEchelle(...,
25.              stopReservoir as stopCuivreEchelle);
26.           }par{
27.             run resevoirCorsEchelle(...,
28.              stopReservoir as stopCuivreEchelle);
29.           }par{
30.             run resevoirTrombonesEchelle(...,
31.              stopReservoir as stopCuivreEchelle);
32.           }par{
33.             run resevoirRise(...,
34.              stopReservoir as stopCuivreEchelle);
35.           }
36.           fork{
37.             run resevoirFlutesEchelle(...,
38.              stopReservoir as stopReservoirsEchelle);
39.           }par{
40.             run resevoirHautboisEchelle(...,
41.              stopReservoir as stopReservoirsEchelle);
42.           }par{
43.             run resevoirClarinettesEchelle(...,
44.               stopReservoir as stopReservoirsEchelle);
45.           }
46.           // Après les réservoirs ci-dessus
47.           run resevoirPianoEchelle(...,
48.               stopReservoir as stopReservoirsEchelle);
49.         }par{
50.           // Garde fou temporel pour controler le temps d'exécution
51.           await count (20, tickEchelle.now);
52.           emit stopCuivreEchelle();
53.
54.           await count (10, tickEchelle.now);
55.           emit stopCuivreEchelle();
56.           emit violonsEchelleOUT([false, 255]);
57.           emit altosEchelleOUT([false, 255]);
58.           emit cellosEchelleOUT([false, 255]);
59.           emit ctrebassesEchelleOUT([false, 255]);
60.           emit bassonsEchelleOUT([false, 255]);
61.           emit stopReservoirsEchelle();
62.           hop { gcs.informSelecteurOnMenuChange(255,"Echelle", false); }
63.           break laTrappe;
64.         }
65.       }
66.       hop { gcs.informSelecteurOnMenuChange(255,"Echelle", false); }
67.       hop { console.log("-- FIN NATURELLE SESSION ECHELLE --"); }
68.     }
69.   }
70. }
```

Regardons plus précisément l’orchestration mise en place dans le corps du weakabort ligne 21. Nous avons un premier fork de la ligne 23 à la ligne 35. Nous avons d’abord 4 réservoirs activés en parallèle, *trompettes*, *cors*, *trombones* et *rise*. Ceci signifie que nous ne passerons en ligne 36 que lorsque tous les réservoirs auront été vidés. De la ligne 36 à 45 nous avons un nouvel ensemble de réservoirs en parallèle, flutes, hautbois et clarinettes. De la même façon nous n’aurons le réservoir piano que lorsque tous les réservoirs de bois auront été vidés. En parallèle de l’exécution des

réservoirs nous avons un garde-fou sur le temps en ligne 51. Ici quoiqu'il arrive, au bout de 20 `tickEchelles`, nous arrêterons les réservoirs de cuivre puis nous attendrons encore 10 `tickEchelles` pour tuer le corps du `weakabort` et sortir de la trappe référencée en ligne 4. Les `emits` des lignes 56 à 60 permettent de désactiver les groupes activés en début de session lignes 15 à 18.

Voici la façon dont le réservoir *Trompettes* avait été créé à l'aide du code vu p.75 :

```
45. hiphop module resevoirTrompettesEchelle(tick,stopReservoir)
46. implements ${opus1Int.creationInterfaces(par.groupesDesSons[0])}    {
47.     ${makeReservoir( [ "trompettesEchelle1", "trompettesEchelle2",
48.         "trompettesEchelle3","trompettesEchelle4"])};
49. }
```

Dans cette session nous voyons que les groupes répétitifs de patterns de cordes restent actifs du début à la fin. C'est-à-dire que l'audience a la possibilité de jouer des cordes pendant toute la durée de la pièce. En revanche, nous n'aurons jamais de superposition de cuivres et de bois puisqu'ils ne peuvent que se succéder. Le piano ne viendra qu'à la fin, s'il reste suffisamment de temps, car au bout de 20 + 10 ticks, tout s'arrêtera.

### 6.2.7 Opus 2

Nous ne verrons pas le détail complet de cette pièce de 189 patterns qui reprend la plupart des techniques déjà évoquées. Elle se décompose en 4 sessions, 2 sessions mettant l'accent sur l'utilisation des modes à transposition limitée, une session atonale libre et une session dodécaphonique.

Voici un extrait mettant en avant dans un autre contexte les mécanismes de transpositions vus dans le chapitre sur la composition par patterns et dans l'orchestration de Grand Loup.

```
1.  hiphop module sessionAtonale(tick,setTimerDivision,resetMatriceDesPossibles)
2.  implements ${opus2Int.creationInterfaces(par.groupesDesSons[0]) } {
3.      signal stopReservoir;
4.  
5.      hop{ console.log("-- DEBUT SESSION ATONALE --"); }
6.      emit setTimerDivision(4); // 4 noires = 1 mesure
7.      hop{transposeAll(0);}
8.  
9.      emit  cellosAtonalOUT([true, 255]);
10.     hop { gcs.informSelecteurOnMenuChange(255,"cellosAtonal", true); }
11. 
12.     await (cellosAtonalIN.now);
13.     hop{ transpose(CCTransposeCellos, 1);}
14. 
15.     await (cellosAtonalIN.now);
16.     hop{ transpose(CCTransposeCellos, -5);}
17. 
18.     await (cellosAtonalIN.now);
19.     hop{ transpose(CCTransposeCellos, -2);}
20. 
21.     await (cellosAtonalIN.now);
22.     hop{ transpose(CCTransposeCellos, 0);}
23. 
24.     seq1:{
25.         fork{
26.             run resevoirPianoAtonal(...);
27.             break seq1;
28.         }par{
29.             every (tick.now){
30.                 hop{
```

```
31.                     transposition = (transposition+1)% 6;
32.                     transpose(CCTransposeCellos, transposition);
33.                 }
34.             }
35.         }
36.     }
37.
38.     hop{
39.         transpose(CCTransposeCellos, 0);
40.         transposition = 0;
41.     }
42.
43.     seq2:{
44.         fork{
45.           emit  cellosAtonalOUT([false, 255]);
46.           hop{gcs.informSelecteurOnMenuChange(255,"cellosAtonal", false);}
47.
48.           emit  contrebassesAtonalOUT([true, 255]);
49.           hop{gcs.informSelecteurOnMenuChange(255,"contrebassesAtonal",true);}
50.           await count (5, contrebassesAtonalIN.now);
51.
52.           emit  altosAtonalOUT([true, 255]);
53.           hop{ gcs.informSelecteurOnMenuChange(255,"altosAtonal", true); }
54.           await count (5, altosAtonalIN.now);
55.
56.           emit  contrebassesAtonalOUT([false, 255]);
57.           hop{gcs.informSelecteurOnMenuChange(255,"contrebasseAtonal",false);}
58.
59.           emit  violonsAtonalOUT([true, 255]);
60.           hop{ gcs.informSelecteurOnMenuChange(255,"violonsAtonal", true);}
61.           await count (5, violonsAtonalIN.now);
62.
63.           break seq2;
64.         }par{
65.             every (tick.now){
66.   // Transposition contrôlée par hiphop, donc au niveau du pattern complet
67.                 hop {
68.                     transposition = (transposition+1)% 6;
69.                     transposeAll(transposition);
70.                 }
71.             }
72.         }
73.     }
74.
75.     trans:{
76.         fork{
77.             fork{
78.                 run resevoirTrompettesAtonal(...);
79.             }par{
80.                 run resevoirCorsAtonal(...);
81.             }
82.             break trans;
83.         }par{
84.             every (tick.now){
85.  // Transposition contrôlée par hiphop, avec donc une transposition par tick.
86.                 hop {
87.                     transposition = (transposition+1)% 3;
88.                     transposeAll(transposition);
89.                 }
90.             }
91.         }
92.     }
93.     hop{ gcs.informSelecteurOnMenuChange(255,"Fin", true); }
```

```
94.     hop{hop.broadcast('alertInfoScoreON', "FIN");}
95.
96.     emit  violonsAtonalOUT([false, 255]);
97.     emit  altosAtonalOUT([false, 255]);
98.     emit  contrebassesAtonalOUT([false, 255]);
99.     emit resetMatriceDesPossibles();
100.           hop{
101.                   ableton.cleanQueues();
102.                   console.log("-- FIN SESSION ATONALE --");
103.           }
104.       }
```

En ligne 6 nous avons introduit la possibilité de changer en cours de route les indications de mesure et donc la longueur des ticks.

De la ligne 5 à 22 nous avons une séquence basée sur des violoncelles avec des transpositions qui dépendent d'actions de l'audience. Des lignes 24 à 32 nous avons un réservoir de patterns de piano qui tourne en parallèle à une transposition en cycle des violoncelles. A partir de la ligne 43 nous avons une nouvelle séquence. De la ligne 45 à 61 nous avons une orchestration de cordes transposée systématiquement par les lignes 65 à 70. Nous avons le même type de scénario qui s'enchaine en ligne 75. Nous avons vu donc ici une utilisation systématique des transpositions qui donne effectivement une forte sensation de mouvement, on peut entendre une simulation sonore utilisant des transpositions à l'adresse https://soundcloud.com/user-651713160/opus2-2-5-instances.

### 6.2.8 Une autre orchestration de l'Opus1

Parmi les orchestrations possibles, nous allons aborder ici un autre type de déroulement qui peut s'exprimer sous une forme de cartographie de trajets possibles. Prenons le schéma Figure 14 *: Chemins pour Opus1 version 2*, qui représente des trajets. En bleu nous avons les groupes répétitifs de patterns, en rouge nous avons les réservoirs de patterns. On passe d'un groupe de patterns à un autre en suivant les flèches. Les flèches en gras signifient des choix d'orientation possibles. Nous avons des flèches en gras quand plusieurs chemins sont laissés au libre choix de l'audience. Les numéros sur les flèches nous servirons de repère dans le code HipHop.js, ils désignent les choix.

Nous commençons notre parcours avec des violoncelles seuls (Cellos), au bout d'un certain nombre d'occurrences des violoncelles, nous proposons trois chemins possibles à l'audience, aller vers les altos, aller vers les contrebasses, ou aller vers les trompettes. Si nous choisissons les altos, nous irons vers les violons au bout de quelques occurrences. Il se passera la même chose si nous passons par les contrebasses. En revanche, les trompettes nous feront passer par le réservoir des cors avant de nous proposer un choix entre piano et trombones. Le reste du schéma est facile à interpréter. Sur ces principes, nous voyons que ce dessin exprime plusieurs scénarios possibles pour la même pièce et que ces scénarios dépendent de choix proposés à l'audience lors de certains moments clés. La décision de tester ce type de scénario est issue de réflexions sur l'interaction. Nous adresserons les questions liées à l'interaction dans un chapitre spécifique, ce qui nous intéresse ici est la façon dont nous allons coder ce type de scénario en HipHop.js.

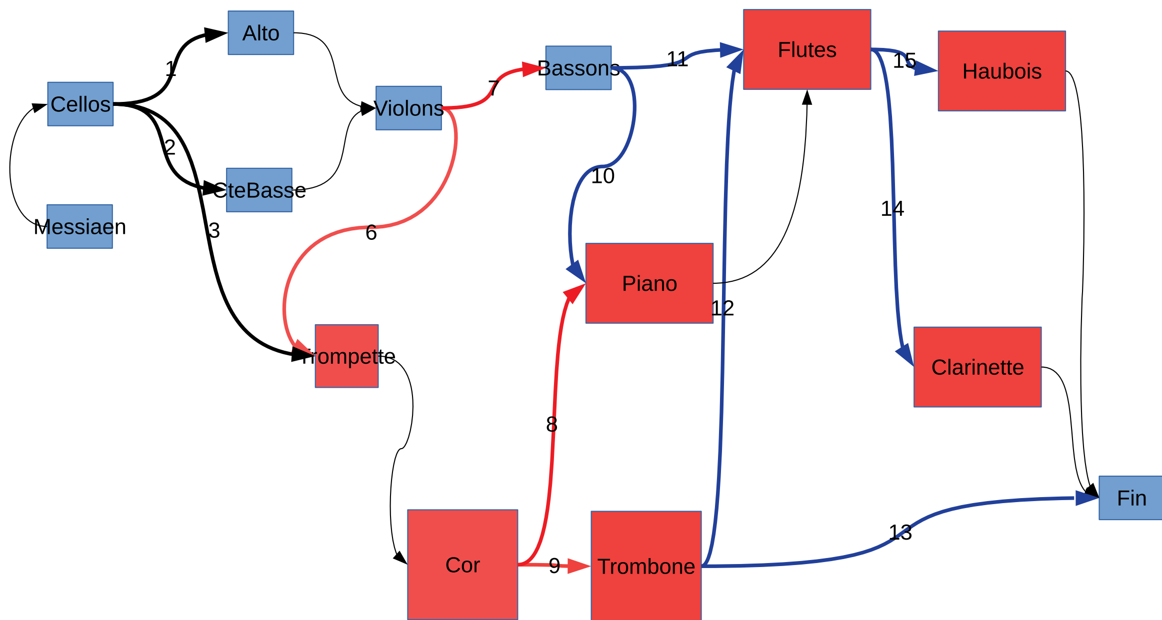



**Figure 14 : Chemins pour Opus1 version 2**

Voici ci-dessous le codage d'une partie de cet automate. Dans les commentaires du code HipHop.js nous avons introduit des indications qui permettent de suivre les déroulements :

- -> 1 (flèche suivie d'un chiffre) signifie une condition pour activer la transition 1. Ici ce sont des instructions `await` sur des instruments.
- 7 -> (chiffre suivi d'une flèche) signifie une préparation de la transition 7, via un choix possible faisant suite à l'émission d'un signal OUT.

Les lignes 19 à 21 émettent trois messages permettant trois choix. La ligne 25 informe l'audience sur les choix possibles. Chaque choix fait l'objet du corps d'une trappe. En ligne 27, nous avons une trappe pour le traitement des trois choix possibles : alto, contrebasse ou trompettes. Chacun des corps en parallèle dans cette trappe traite un des chemins possibles, ligne 29 pour alto, ligne 48 pour contrebasse et ligne 67 pour trompette. Ces chemins peuvent conduire à activer des groupes comme en ligne 35 ou 39. En ligne 42 nous avons l'attente des violons avant de proposer un nouveau choix entre trompette et basson. Dans cette orchestration les cordes entrent progressivement et ne sont jamais désactivées avant la fin entre les lignes 92 à 96. Les cuivres, bois et piano étant des réservoirs peuvent s'épuiser et disparaitre de l'orchestration. Le résultat probable sera donc un certain maintien des cordes avec des évolutions possibles sur la façon dont les autres instruments vont apparaitre.

```
1.  hiphop module sessionEchelle(tick, setTimerDivision)
2.  implements ${opus1Int.creationInterfaces(par.groupesDesSons[0])} {
3.    signal stopReservoirsEchelle, stopCuivreEchelle, tickEchelle =0;
4.
5.    laTrappe:{
6.      fork{
7.        every(tick.now) {
8.          emit tickEchelle();
9.        }
```

```
10.     }par{
11.       hop { console.log("-- DEBUT SESSION ECHELLE --"); }
12.       emit setTimerDivision(4);
13.       emit cellosEchelleOUT([true, 255]);
14.       hop { gcs.informSelecteurOnMenuChange(255,"cellosEchelle", true); }
15.
16.       await count (4, cellosEchelleIN.now);
17.
18.       // Proposition de choix
19.       emit AltosOUT([true, 255]); // 1->
20.       emit ContrebassesOUT([true, 255]); // 2->
21.       emit TrompettesOUT([true, 255]); // 3 ->
22.       hop { gcs.informSelecteurOnMenuChange(255,"Choix", true); }
23.
24.       // Le premier choix qui atteint 5 votes gagne
25.       hop {hop.broadcast('alertInfoScoreON',
                  "Le premier choisi 5 fois gagne");}
26.
27.       AtloCtbTromp:{
28.         fork{
29.           await count (5, AltosIN.now); // -> 1
30.           emit AltosOUT([false, 255]);
31.           emit ContrebassesOUT([false, 255]);
32.           emit TrompettesOUT([false, 255]);
33.           hop {hop.broadcast('alertInfoScoreOFF');}
34.
35.           emit altosEchelleOUT([true, 255]);
36.           hop { gcs.informSelecteurOnMenuChange(255,"altosEchelle", true); }
37.           await count (5, altosEchelleIN.now); // On attend qq altos
38.
39.           emit violonsEchelleOUT([true, 255]);
40.           hop { gcs.informSelecteurOnMenuChange(255,"violonsEchelle", true); }
41.
42.           await count (5, violonsEchelleIN.now);
43.           emit BassonsOUT([true, 255]); // 7 ->
44.           emit TrompettesOUT([true, 255]); // 6 ->
45.           hop { gcs.informSelecteurOnMenuChange(255,"Choix", true); }
46.           break AtloCtbTromp;
47.         }par{
48.           await count (5, ContrebassesIN.now); // -> 2
49.           emit AltosOUT([false, 255]);
50.           emit ContrebassesOUT([false, 255]);
51.           emit TrompettesOUT([false, 255]);
52.           hop {hop.broadcast('alertInfoScoreOFF');}
53.
54.           emit ctrebassesEchelleOUT([true, 255]);
55.           hop{gcs.informSelecteurOnMenuChange(255,"ctrebassesEchelle", true);}
56.           await count (5, ctrebassesEchelleIN.now);
57.
58.           emit violonsEchelleOUT([true, 255]);
59.           hop { gcs.informSelecteurOnMenuChange(255,"violonsEchelle", true); }
60.
61.           await count (5, violonsEchelleIN.now);
62.           emit BassonsOUT([true, 255]); // 7 ->
63.           emit TrompettesOUT([true, 255]); // 6 ->
64.           hop { gcs.informSelecteurOnMenuChange(255,"Choix", true); }
65.           break AtloCtbTromp;
66.         }par{
67.           await count (5, TrompettesIN.now); // -> 3
68.           emit AltosOUT([false, 255]);
69.           emit ContrebassesOUT([false, 255]);
70.           emit TrompettesOUT([false, 255]);
71.           hop {hop.broadcast('alertInfoScoreOFF');}
```

```
72.
73.             run resevoirTrompettesEchelle(..., stopReservoir as stopCuivreEchelle);
74.             run resevoirCorsEchelle(..., stopReservoir as stopCuivreEchelle);
75.             emit PianoOUT([true, 255]); //  8 ->
76.             emit TrombonesOUT([true, 255]); //  9 ->
77.             hop { gcs.informSelecteurOnMenuChange(255,"Choix", true); }
78.             break AtloCtbTromp;
79.           }
80.         }
81.
82.         // En plus des cellosEchelle.
83.         // Ici on a soit des altoEchelle + violonsEchelle
84.         // soit des contrebassesEchelle + violonsEchelle
85.         // soit des trompettesEchelle,
86.
87.       ...
88.
89.
90.       }
91.     }
92.     emit altosEchelleOUT([false, 255]);
93.     emit violonsEchelleOUT([false, 255]);
94.     emit cellosEchelleOUT([false, 255]);
95.     emit ctrebassesEchelleOUT([false, 255]);
96.     emit bassonsEchelleOUT([false, 255]);
97.     hop { gcs.informSelecteurOnMenuChange(255,"Fin", true); }
98.     hop {hop.broadcast('alertInfoScoreON', "FIN");}
99. }
```

Un exemple sonore de 53 secondes du début de cette orchestration est disponible à l'adresse :

https://soundcloud.com/user-651713160/exemple-session-echelle-opus1

Il s'agit d'une simulation. Au début nous entendons les violoncelles seuls qui ont été mis à disposition avec l'instruction `emit` de la ligne 13. Les altos apparaissent à la 12ème seconde après que l'instruction `await` de la ligne 16 ait compté les quatre patterns de violoncelles et que le choix proposé dans les lignes 19 à 21 se soit fait au hasard de la simulation sur les altos. Ceci nous a conduit à la ligne 35. Nous passons ensuite rapidement aux violons en seconde 15. Le simulateur a donc été rapide sur la sélection des 5 patterns de violons attendus en ligne 37. En effet, l'automate prend ici en compte les sélections et non le fait que les patterns aient été joués. Les bassons rendus disponibles en ligne 43 apparaissent à la 33ème seconde, soit 18 secondes après l'apparition des violons. Ceci correspond effectivement à un peu plus de la durée des 5 patterns attendus en ligne 42. Nous n'irons pas plus loin dans le déroulement sonore de cette pièce, mais nous voyons que l'analyse de l'écoute correspond bien à l'automate HipHop.js.

## 6.3 Conclusions sur la programmation des orchestrations

Dans les paragraphes précédents, nous avons vu que nous avions utilisé la plupart des instructions[5] HipHop.js. Si nous nous contentons de traduire une représentation graphique en HipHop.js, il est possible de réaliser des orchestrations avec un nombre réduit d'instructions. Des orchestrations simples, donnant des résultats musicalement satisfaisants, peuvent se construire avec des

[5] `await`, `count`, `immediate`, `emit`, `abort`, `weakabort`, `loop`, `fork`, `par`, `hop`, `every`, `break`, `val`, `preval`, `now`, `yield` et `run`. Parmi les commandes que nous n'avons pas rencontrées dans ces exemples il reste essentiellement `sustain` et `suspend` que nous utilisons pour des cas complexes de dépendances entre sessions.

instructions `emit` et `await`, en séquence ou en parallèle, entourées d'instructions `abort` par exemple. Dans ce cas, ce qui caractérise notre programmation d'orchestration n'est donc pas la complexité de la programmation HipHop.js. Les automates que nous avons vus dans ce document ne sont pas très difficiles à comprendre avec la syntaxe de HipHop.js. En revanche, nos programmes destinés à des orchestrations riches se traduisent par des automates avec un nombre d'états très important (jusqu'à 10 000 pourl'Opus2). Ce qui signifie que sous une apparente simplicité de structure du programme, l'automate généré est très complexe et serait ingérable avec un langage généraliste. Cette simplicité d'expression est un des apports importants de l'expressivité de HipHop.js à notre projet. Ce volume élevé d'états est lié à l'utilisation d'un nombre important de signaux. Dans le cas de l'Opus 1 par exemple, nous avions 72 groupes de patterns, ce qui donne 144 signaux OUT et IN à gérer pour un seul automate d'orchestration. Ceci nous a posé initialement des problèmes de performance, car si l'automate est censé s'exécuter en temps nul d'un point de vue théorique, l'activation d'un automate avec beaucoup d'états peut prendre un temps non négligeable au regard des contraintes de déroulement de l'orchestration. Pour que le comportement de l'automate soit perçu en temps nul, il faudra que son temps d'exécution soit inférieur à l'intervalle de temps qui sépare deux appels. Les appels à l'automate d'orchestration sont réalisés à un intervalle de temps qui est défini comme un multiple de la pulsation du métronome. Le métronome est fixé par le *tempo* de la pièce musicale. Comme nos orchestrations produisent des automates avec beaucoup d'états, au début de nos travaux les durées d'exécution des automates n'étaient pas négligeables vis-à-vis des cycles d'appel. Dans un premier temps ceci a limité le type de traitement que l'orchestration pouvait contrôler. Fort heureusement, le compilateur s'est considérablement amélioré, rendant possibles certains scénarios impossibles initialement. Néanmoins si la simplicité d'expression d'automates volumineux est un point important, il n'est qu'une première étape dans la relation circulaire entre outil informatique et création musicale. HipHop.js avec son catalogue d'instruction est aussi source de propositions. Ainsi, initialement nous n'avions pas utilisé les commandes `sustain` et `suspend`. Nous n'avions pas trouvé un usage immédiat de ces instructions pour traduire nos représentations graphiques. C'est en nous interrogeant sur leur sens que nous nous sommes demandé ce qui pourrait être *suspendu* ou *maintenu* dans le déroulement d'une orchestration. C'est de cette question que nous est venue l'idée de contrôler des interactions entre sessions en prenant en compte des événements très particuliers, comme des sélections de patterns uniques pouvant perturber en profondeur le déroulement d'une session en introduisant une autre session en parallèle, en interrompant ou pas celle en cours. Nous n'avons pas donné d'exemple de ce type de codage pour ne pas alourdir nos présentations de programmation d'orchestration, nous avions cependant évoqué leur existence en commentaire de la Figure 9: Exemple de superposition de sessions, p.51. En allant encore plus loin sur les enchevêtrements de sessions, pourquoi ne pas imaginer des événements particuliers ayant des probabilités faibles de se produire, mais qui remettraient en cause l'enchaînements des sessions, les interrompant, les réinitialisant, les mettant en parallèles, etc. Ainsi ces deux instructions nous permettent d'imaginer des scénarios originaux qui auront pour caractéristique de ne pas pouvoir s'exprimer graphiquement. Nous pouvons aussi imaginer qu'en mettant en œuvre des scénarios variés de combinaison de sessions, nous ouvrions d'autres pistes musicales nécessitant de penser d'autres formes d'algorithmes, continuant ainsi le processus dialectique entre outil et création. Toujours dans le registre de l'impact de l'outil sur la création, la sémantique de HipHop.js, issue d'Esterel, sous-entend qu'il sera possible de faire des preuves sur l'orchestration et notamment de vérifier que certains scénarios sont impossibles. La mise en œuvre de « preuves », c'est-à-dire la vérification formelle que certains états sont possibles ou non, sera certainement un compagnon d'un nouveau type pour le compositeur de musique interactive selon nos principes. Imaginons qu'un compositeur souhaite être certain qu'un groupe de flûte ne puisse

pas être sélectionner avec un groupe de trombone. Cette vérification sera pratiquement impossible à faire « à la main » pour une orchestration un peu complexe. Un outil de preuve pourra la faire.

Pour conclure ce chapitre sur la programmation, nous pouvons constater que HipHop.js est en l'état proche d'un DSL musical. Nous avons démontré que l'expressivité de HipHop.js est adaptée à notre objectif et qu'elle est aussi une source de propositions pour le compositeur. Il serait néanmoins possible de concevoir une évolution de HipHop.js encore plus adaptée à la musique en travaillant sur une intégration plus approfondie avec les mécanismes d'interaction et de simulation que nous verrons par la suite. On peut aussi imaginer une traduction vers HipHop.js d'une représentation graphique de l'orchestration. Cette traduction n'offrirait pas le même champ de possibilités qu'une programmation en HipHop.js, mais rendrait notre méthode accessible à des musiciens non informaticiens. En effet, la souplesse qu'apporte HipHop.js ne doit pas faire oublier que programmer avec un langage réactif « à la Esterel » peut devenir difficile dès que l'automate se complexifie. La gestion des cycles de causalité est une des difficultés inhérentes à la programmation HipHop.js que nous avons rencontrée régulièrement et qui est difficile à maîtriser.

Nous avons posé les bases de notre méthode de composition et vu sa relation avec la programmation HipHop.js, il s'agit à présent de l'évaluer musicalement et donc de mettre en œuvre la *durée*, en donnant vie aux orchestrations. C'est ce que nous allons voir dans les chapitres suivants tout d'abord au travers des interfaces avec une audience, puis à l'aide d'un outil de simulation.

# 7 MISE EN ŒUVRE DE LA DUREE

L'INTERACTION 98

LES NIVEAUX D'INTERACTION 99
LES INTERFACES 101
LE SEQUENCEUR DISTRIBUE 110

LE SIMULATEUR 114

CONCLUSIONS SUR L'INTERACTION 118

> A l'intérieur d'un désordre, nous instaurons des systèmes de probabilités purement provisoires, hypothétiques, complémentaires d'autres systèmes que, dans le même temps ou par la suite, nous pourrons également assumer ; ce faisant nous jouissons précisément de l'équiprobabilité de tous ces systèmes, et de la disponibilité ouverte du processus pris dans son ensemble. (L'œuvre ouverte, U. Eco, p.106)

Dans les chapitres précédents nous avons abordé la composition par patterns et une façon d'organiser l'accès aux patterns via une orchestration. Notre système a besoin de plus d'un organe qui l'anime, d'une forme de concrétisation de la *durée*. Autrement dit, la mécanique de l'orchestration a besoin d'un moteur. Le moteur que nous avons retenu, dès l'origine de notre recherche, est une audience. C'est au travers d'interactions homme-machine qu'une œuvre prend vie et que la durée au sens de Bergson devient une réalité. Ce recours à une interaction humaine élargie à l'audience part du principe que plusieurs consciences réunies apporteront à une œuvre musicale une dimension différente de celle conçue uniquement par un esprit solitaire, et produite par un groupe de musiciens n'ayant de liens qu'entre eux. Pour construire nos interfaces avec l'audience nous devons définir quels sont les principaux critères qui vont conduire notre démarche. Il s'agira tout d'abord de trouver une façon de motiver l'audience, c'est-à-dire de lui faire prendre conscience de son rôle dans le déroulement de la performance pour lui donner envie de « jouer le jeu » et de maintenir son attention. Elle devra être informée le plus clairement possible des actions qu'elle va pouvoir immédiatement déclencher et avoir un retour explicite et fiable concernant les effets de ses actions. Il s'agit d'éviter que des actions soient sans effets ou que l'effet ne soit pas perçu par celui qui l'a demandé. Il faudra aussi trouver le moyen de lui donner des perspectives sur le déroulement à long terme de la performance et la façon dont elle va pouvoir l'orienter. C'est parce qu'il y aura un ou plusieurs buts à atteindre que l'audience aura envie de participer. De plus il faudra que les interfaces soient suffisamment simples pour ne pas demander un temps d'apprentissage long ou un effort particulier. Nous verrons que, même si le sujet n'est pas clos, nos interfaces ont été améliorées en fonction des performances réalisées de façon à respecter ces critères : clarté des choix, fiabilité, vision long terme et simplicité. Nous verrons aussi que l'interaction met en jeu l'audience mais aussi d'autres acteurs qu'il nous faudra identifier, comprendre ou imaginer les rôles.

En complément de cette réflexion sur les interfaces, pour évaluer son travail le compositeur doit donc pouvoir idéalement faire appel à une audience. Ce n'est pas une opération simple. Il fallait donc trouver un moyen de faire fonctionner l'œuvre dans un mode dégradé qui permette d'avoir une idée de ce qui pourrait se passer avec une véritable audience avant de réaliser une performance. C'est ce besoin de simuler le comportement d'une audience pour le compositeur qui nous a amené à proposer un « moteur aléatoire ». C'est-à-dire un programme qui se comporterait plus ou moins comme une audience. Et de façon à mettre à l'épreuve la pièce de musique, une audience qui serait la moins constructive possible, qui choisirait les patterns complétement au hasard. C'est

ce que nous avons réalisé avec notre moteur aléatoire qui est une forme de simulateur d'audience. Nous traiterons de l'interaction avec l'audience avant d'aborder le moteur aléatoire.

## 7.1 L'interaction

> Comme dans l'univers einsteinien, le refus d'une expérience privilégiée n'implique pas le chaos des relations, mais la règle qui permet leur organisation. L'œuvre en mouvement rend possible une multiplicité d'interventions personnelles, mais non pas de façon amorphe et vers n'importe quelle intervention. Elle est une invitation, non pas nécessitante ni univoque mais orientée, à une insertion relativement libre dans un monde qui reste voulu par celui de l'auteur. (L'œuvre ouverte, U. Eco, p.34)

Lorsque nous nous sommes posé la question de définir une méthode de composition en confrontation avec la programmation informatique et avec le langage HipHop.js, la piste de l'interaction avec l'audience nous a semblé naturelle comme expression de la durée. Les technologies du Web sont par définition des technologies dédiées à l'interaction. Cela dit, pour pouvoir expérimenter il fallait définir un type d'interaction en gardant en tête nos principes de clarté des choix, fiabilité, vision long terme et simplicité. Un des éléments structurants a été pour nous le choix du mode production du son. Nous avons vu que plusieurs travaux sur la musique interactive se concentrent sur les terminaux, c'est-à-dire sur des façons de produire le son à l'aide des terminaux. Plusieurs projets comme *echobo*, *a.bel*, ou *Swarmed* travaillent dans ce sens. Il s'agit d'une piste que nous avons décidé de ne pas suivre pour une raison simple. Les limites imposées par les haut-parleurs des smartphones ou tablettes ne permettent pas d'obtenir une qualité de son satisfaisante dans le contexte d'une performance en concert avec plusieurs dizaines de participants. En effet, les sons de chacun des terminaux ne s'additionnent pas comme les haut-parleurs d'une enceinte. Bien que l'analogie soit tentante, on ne peut pas considérer un ensemble de haut-parleurs de smartphones comme un réseau contrôlable de haut-parleurs. Chaque individu dans une salle ne percevra que son propre terminal et quelques-uns autour de lui dans un rayon de quelques mètres. De plus, la bande passante de ces haut-parleurs est dépendante de la qualité des smartphones, et vue leurs dimensions, les basses fréquences sont fortement atténuées. Nous nous sommes donc rattachés au mouvement initié par *massMobile* [91] ou *Open Symphony* [93] en considérant que ce ne serait pas les terminaux qui produiraient la musique mais un système central de bonne qualité ou bien des musiciens. Ceci ne signifie pas que le haut-parleur du terminal soit exclu du système, son rôle ne sera pas de produire la musique mais plutôt de permettre à chacun d'écouter et d'évaluer des patterns localement.

Le deuxième élément qui conditionne fortement la structure de l'interaction est notre décision d'utiliser une technique de composition basée sur des patterns. Ceci signifie que nous travaillons à partir d'un matériel de base identifiable, contrairement à l'utilisation de processus travaillant directement sur le son comme le proposent *massMobile*, *Performance Without Border* ou *Embodied iSound* par exemple. L'avantage de travailler sur un matériel de base identifiable est de permettre

aux individus dans l'audience de se repérer plus facilement qu'au travers de processus de traitement du son. Il est par exemple plus simple pour une personne de reconnaitre un pattern dans un environnement sonore, et pour le système d'associer un pattern à un individu, que de repérer durant une performance que le filtrage que j'ai choisi est actif sur un son que je ne connais pas. En fait, plus le traitement acoustique mis à la disposition de l'audience sera de bas niveau, plus il sera difficile à identifier par les individus. Le pattern est un événement acoustique de haut-niveau plus facile à mettre en scène qu'un processus.

Nous avons fait le choix d'un système centralisé et de la production de musique à base de patterns contrôlée par un langage réactif synchrone, nous avons posé des critères de fonctionnement sur les interfaces pour l'audience. Mais avec quoi l'audience va-t-elle interagir ? Et de quelles interactions peut dépendre la relation qu'aura l'audience avec notre système de production de la musique ? Avant d'aller plus loin nous devons réfléchir aux éléments qui vont constituer l'interaction. Quels sont-ils ? Lesquels sont utiles et dans quels buts ?

### 7.1.1 Les niveaux d'interaction

Pour comprendre ce que signifie « interagir » pour notre système, plusieurs questions se posent. Quelles sont les parties prenantes, quelles peuvent être leurs relations et en quoi ces relations peuvent enrichir la performance ?

Le premier constat est qu'une pièce de musique interactive est un système avec au minimum 5 types de participants :

- L'individu.
- Le groupe, l'audience.
- Un maitre du jeu qui peut être le compositeur. Le maitre du jeu est celui qui exerce un contrôle sur le système automatique de gestion de l'interaction.
- Le système informatique.
- L'organe de production du son.

Peuvent s'y ajouter :

- Des instrumentistes, à la place ou avec l'organe de production du son.
- Un ingénieur du son.

Dans une version simple de Skini, nous aurons 13 types de communications possibles, contre deux pour un soliste en concert (soliste-> audience, audience->soliste), 4 pour un orchestre avec un chef (chef -> orchestre, orchestre -> chef, orchestre -> public, public - > orchestre). L'interaction introduit donc un niveau de complexité supplémentaire dans la communication par rapport à la façon habituelle de produire de la musique. Nous pensons qu'un travail sur l'interaction devrait être accompagné d'un minimum de réflexion sur chacun de ces types de communication de façon à évaluer leurs impacts sur la performance et le bien-fondé de leur utilisation pour notre projet.

1) Individu -> Individu
2) Individu -> groupe
3) Individu -> Maitre du jeu
4) Individu -> Système
5) Individu -> Organe de production du son
6) Maitre du jeu -> groupe
7) Maitre du jeu -> Individu

8) Maitre du jeu -> Système
9) Maitre du jeu -> Organe de production du son
10) Système -> groupe
11) Système -> Individu
12) Système -> Maitre du jeu
13) Système -> Organe de production du son

Avec des musiciens nous pourrions ajouter :

1) Individu -> Instrumentistes
2) Maitre du jeu -> Instrumentistes
3) Système -> Instrumentistes
4) Instrumentistes -> groupe
5) Instrumentistes -> Individu
6) Instrumentistes -> Maitre du jeu
7) Instrumentistes -> Système
8) Instrumentistes -> Organe de production du son (autre que l'instrument)
9) Instrumentistes -> Instrumentistes

#### *7.1.1.1 A l'initiative de l'Individu*

Notons que les deux types de communication (Individu vers Individu, Individu vers groupe) n'ont pas d'action directe sur la production du son. L'interaction entre individus est en général exclue du spectacle. L'usage veut que l'on ne parle pas à son voisin tout simplement pour ne pas perturber le spectacle. Néanmoins si le média n'est plus la parole, mais un terminal, il peut être utilisé sans interférer avec le son. Nous n'avons pas utilisé ce type d'interaction dans nos expériences pour le moment car il ne nous parait pas immédiatement lié à la question de la production de musique à base de patterns et automates.

La communication de chaque individu vers le groupe correspondrait à ce que nous proposent les réseaux sociaux de type *tweet*, mais il peut s'agir aussi d'informations produites par les initiatives de chacun. Nous verrons que notre système diffuse effectivement les choix réalisés par chaque membre de l'audience.

La communication des individus vers le maitre du jeu peut permettre de tenir compte de retours du public sur le déroulement de la performance comme le niveau sonore, l'ennui… Pour les mêmes raisons que lors des échanges entre individus, nous n'avons pas mis en œuvre ce type d'interaction.

Dans notre cas, la communication de l'individu vers le système est constituée des messages que les clients HTML envoient au serveur.

La communication des individus vers l'organe de production du son pourrait sembler hors sujet, nous verrons qu'elle ne l'est pas quand on souhaite donner plus de contrôle sur les patterns à certains membres de l'audience. L'interface « séquenceur distribué » que nous verrons plus loin se situe dans ce scénario d'interaction.

#### *7.1.1.2 A l'initiative du Maitre du Jeu*

La communication du maitre du jeu vers l'audience est une façon de donner des directives en fonction du déroulement de la performance. On peut imaginer des scénarios qui utiliseraient ce type de communication du maitre du jeu vers un individu en particulier. Nous n'avons pas encore exploité ce type de scénario.

La communication du maitre du jeu vers le système, est ce que nous appelons le *contrôle* qui fait l'objet d'une interface dédiée.

La communication du maitre du jeu vers l'organe de production du son relève aussi du contrôle, mais plus précisément sur l'activation de synthétiseurs, le contrôles des niveaux, etc. Nous avons réalisé des essais techniquement concluant à ce niveau, mais ils ne sont pas en relation avec la méthode de composition que nous étudions.

La communication du maitre du jeu vers des instrumentistes, revient à introduire le rôle habituel du chef d'orchestre que nous n'avons pas traité.

#### *7.1.1.3 Initiative au Système*

La communication du système vers le groupe correspond aux informations qui sont diffusées sur l'état du spectacle et son déroulé par exemple. Skini utilise largement ce canal.

Du système vers Individu nous aurons des informations plus personnelles sur l'état du spectacle et son déroulé. Dans notre système, il s'agira d'informer un individu sur la bonne exécution de ses requêtes.

La communication du système vers le maitre du jeu, comprend les informations détaillées sur l'état du spectacle et son déroulé, ainsi que les retours des contrôles.

La communication du système vers l'organe de production du son correspond au dépilement des files d'attente et donc aux commandes d'exécution des patterns.

La communication du système vers les instrumentistes est similaire à la précédente mais avec des interfaces différentes.

#### *7.1.1.4 Les niveaux d'interaction de Skini*

Cette approche systématique des niveaux d'interaction nous permet de voir que Skini met actuellement en place la plupart des modes de communication possibles à l'exception des communications : entre individus, individus vers l'audience (tweet), du maitre du jeu vers individus, du maitre du jeu vers l'audience, avec des instrumentistes. La principale conclusion est que nous n'avons pas épuisé les ressources disponibles en termes de niveau d'interaction. Il reste encore des espaces à défricher dans ce domaine.

### 7.1.2 Les interfaces

La plateforme Skini possède 7 types d'interfaces issues des résultats de différentes performances et expérimentations :

1) Une interface « acteur », qui est celle qu'utilise l'audience. Cette interface a connu plusieurs versions donnant des visions différentes et des méthodes différentes de sélection des patterns. Nous nous étions axés initialement sur une sélection par critères en combinant des ambiances et des types d'instruments. Cette méthode a été abandonnée pour plusieurs raisons. Elle n'était pas déterministe et donc pas fiable, au fur et à mesure que la pièce évoluait le panel de patterns était modifié mais l'audience n'en avait pas une vue claire. Ceci se traduisait par une difficulté à retrouver un pattern particulier, ce qui donnait l'impression que la disponibilité des patterns était aléatoire. Nous nous sommes orientés sur une présentation plus simple et fiable donnant une vision complète des groupes de patterns disponibles. Si cette interface répond à nos critères, il ne s'agit pas d'un chantier clos. Il est tout à fait imaginable et souhaitable de proposer de nouvelles interfaces « acteur ».

2) Une interface « contrôleur » qui est celle du maitre du jeu, de celui qui suit le déroulement de la performance, qui peut la démarrer ou l'arrêter. Cette interface est dédiée à un « expert » du système. Elle ne pose donc pas de contrainte ergonomique particulière.
3) Une interface « scrutateur » qui n'a pas encore été utilisée en concert et qui peut avoir deux fonctions : récolter des avis sur l'orchestration en cours ou agir sur l'orchestration en cours. Il s'agit d'une interface complémentaire à celle de l'acteur. Nous pensons qu'elle est plus légère que l'acteur, mais pas plus simple ni complète. Elle est intéressante car elle conditionne certains cas d'usage différents de ceux dépendant de l'interface « acteur ».
4) Le « configurateur » qui permet le paramétrage rapide des commandes MIDI dans la *Digital Audio Workstation*. Elle n'est utilisée que par le compositeur.
5) Une « interface globale » qui donne des informations à l'ensemble de l'audience. L'écran du smartphone ou de la tablette n'est pas suffisant pour exprimer le déroulement d'une orchestration et permettre à l'audience de se situer dans le trajet que suit l'orchestration et donc de comprendre le rôle qu'elle peut jouer. Le rôle de l'interface globale et de donner à tous la même vision du déroulement de la pièce sur un grand écran et de satisfaire notre critère sur la vision à long terme.
6) Une interface pour « musicien » qui permet la production du son non seulement au moyen d'une DAW mais aussi en sollicitant des musiciens pour jouer des patterns ou participer d'une façon ou d'une autre à la performance.
7) Une « interface de conception de patterns » qui fait l'objet d'une section dédiée le « séquenceur distribué ». Cette interface nettement plus complexe que les précédentes, est issues de nos premières expériences et du besoin de nous adresser à des compétences musicales de niveaux très différents au sein d'une audience.

Le chantier des interfaces s'est avéré faire partie d'un domaine de recherche à part entière sur les interfaces homme-machine. Les solutions que nous proposons ici ne sont pas définitives. Nous imaginons que les interfaces devront évoluer non seulement sur leur aspect ergonomique mais aussi dans leur relation avec les orchestrations. Si les solutions que nous avons mises en place sont « généralistes » et sont assez peu dépendantes du style de musique et de l'orchestration, on peut imaginer qu'une interface pour un public dans un opéra soit sensiblement différente d'une interface pour un public de concert Electro dans une cave de Berlin, ou d'une interface pour les visiteurs d'un musée d'art contemporain.

### *7.1.2.1 L'acteur*

L'interface acteur a connu trois versions principales. Elle fait partie de la catégorie « individu vers système ». La version actuelle a été conçue dans le but d'être concise et proche de l'organisation par pattern. Elle est une vue détaillée des groupes de patterns contrôlés par la matrice des possibles. La liste sur fond bleu, dans la copie d'écran de smartphone de la Figure 15 : L'interface "acteur", est la liste des patterns disponibles. Nous avons une couleur différente pour chaque groupe de patterns, nous n'en voyons qu'un sur cet écran. Au-dessus de cette liste nous recevons des informations venant de l'orchestration concernant l'activation et la désactivation des groupes de patterns. A chacun des éléments de la liste est associé un délai en seconde qui correspond au temps d'attente avant que le pattern puisse être joué, ceci est conforme à notre souci de clarté. Ce délai est proportionnel à la longueur de la file d'attente associée à l'instrument qui doit jouer le pattern. En cliquant sur un pattern de la liste, on le sélectionne ce qui permet soit de l'écouter en cliquant sur le bouton « écouter » ou de le charger dans la file d'attente en cliquant « jouer ». Quand un pattern est chargé dans la file d'attente, il est joué lors de la prochaine synchronisation s'il n'y a pas d'attente, sinon une barre de temps qui décompte le temps d'attente apparait en dessous des boutons. Au moment où le pattern est effectivement joué, le terminal vibre et flashe de façon à avertir celui qui avait sélectionné le pattern. Les informations sous la barre horizontale concernent les demandes faites par les autres acteurs dans l'audience. Nous avons le nom du pattern suivi du pseudo de l'acteur entre parenthèses. Le programme de l'acteur est écrit essentiellement en JavaScript. Il comporte une partie écrite en HipHop.js, pour la gestion des boutons « écouter » et « stop », qui permet d'interrompre l'écoute du pattern sur le smartphone avant qu'il ait été joué entièrement. Cette interface répond à la plupart de nos critères, nous pensons cependant que son aspect mériterait d'être retravaillé.

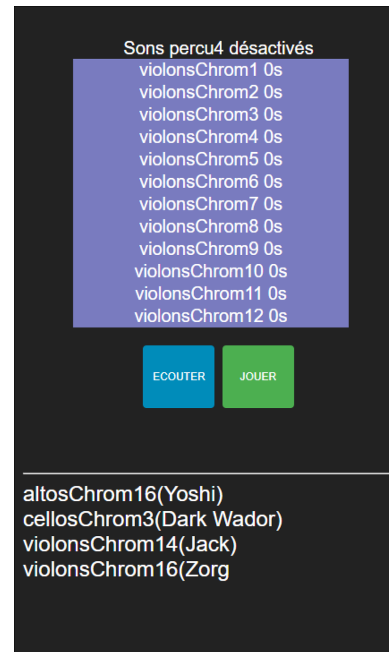



**Figure 15 : L'interface "acteur"**

### *7.1.2.2 HipHop.js pour l'acteur*

Profitons de notre description de l'acteur pour faire une digression relative à l'utilisation de HipHop.js dans le contexte d'une application Web. Notons que le contrôle de l'écoute d'un fichier son constitue déjà un automate non trivial et un cas d'usage intéressant de HipHop.js. Il faut en effet gérer le cas où le fichier son se termine naturellement et le cas où l'utilisateur décide de l'arrêter en route. Il faut en même temps remplacer le bouton « écouter » par un bouton « stop » au démarrage de la lecture. Dans les deux cas où le fichier son s'arrête, en arrivant à sa fin ou en étant interrompu, nous devons remplacer le bouton « stop » qui a remplacé le bouton « écouter », par le bouton « écouter ». Arrêtons-nous un instant sur ce code qui introduit une fonctionnalité spécifique à HipHop.js :

```
1.  function makeListenMachine() {
2.    var audio = new Audio();
3.    var endedListener = null;
4.
5.    function startSound(src, thisIsdone) {
6.      audio.removeEventListener("error", endedListener);
7.      audio.removeEventListener("ended", endedListener);
```

```
8.     endedListener = function() {
9.       console.log("play fini", src, thisIsdone);
10.      thisIsdone();
11.    }
12.    audio.addEventListener("error", endedListener);
13.    audio.addEventListener("ended", endedListener);
14.    audio.src = src;
15.    audio.play();
16.  }
17.
18.  hiphop module automate(in start, stop, out  playing = false) {
19.    every immediate (start.now){
20.      abort (stop.now) {
21.        emit playing(true);
22.        async {
23.          startSound(start.nowval, () => {this.notify(); this.react()} );
24.        } kill {
25.          audio.pause();
26.        }
27.      }
28.      emit playing(false);
29.    }
30.  }
31.
32.  listenMachine = new hh.ReactiveMachine(automate, "automate");
33.
34.  listenMachine.addEventListener("playing", function(evt) {
35.    if (evt.signalValue) {
36.      document.getElementById( "buttonEcouter").style.display = "none";
37.      document.getElementById( "buttonStop").style.display = "inline";
38.    } else {
39.      document.getElementById( "buttonEcouter").style.display = "inline";
40.      document.getElementById( "buttonStop").style.display = "none";
41.    }
42.  });
43.
44.  return listenMachine;
45. }
```

La partie qui nous intéresse le plus dans ce programme se trouve entre les lignes 18 et 30. Le module comprend un premier corps qui est activé à chaque clic sur le bouton « écouter ». Ce clic génère un signal `start`. A chaque signal `start` nous entrons dans le corps d'un `abort` en ligne 20 qui sera détruit quand le bouton `stop` sera activé. L'émission du signal `playing` permet de gérer l'affichage au moyen du listener des lignes 34 à 41.

L'instruction `async` est associée à un corps qui va se mettre en attente d'une réaction de l'automate provoquée par un événement asynchrone à l'aide des instructions `this.notify()` et `this.react()`. C'est une façon d'intégrer des comportements asynchrones dans un programme synchrone HipHop.js. Dans notre code, `this.notify()` et `this.react()` suivent un chemin un peu complexe. Ce sont des instructions à l'intérieur d'une fonction passée en deuxième paramètre de la fonction `startSound()` en ligne 23. Elles seront activées quand le callback passé en paramètre du *listener* audio qui est créé dans la fonction `startSound()`, sera activé. Les callbacks sont mis en place en lignes 12 et 13. Les callbacks sont activés de façon asynchrone quand le fichier son a été lu entièrement, selon le paramètre « ended » ligne 12 ou s'il se produit une erreur de lecture selon le paramètre « error » ligne 13. Dans notre code, lorsque le fichier a été entièrement lu, ou qu'une erreur de lecture s'est produite nous passons en ligne 28 et terminons « naturellement » le corps de `l'abort`. Si l'utilisateur décide d'interrompre la lecture, `l'abort` de la ligne 20 détruit son corps,

ce qui active le corps inclu dans l'instruction `kill` de la ligne 24. L'instruction ligne 25 interrompt la lecture en cours du fichier son.

### *7.1.2.3 Le contrôleur*

Il s'agit de l'interface qui donne une vision et un contrôle global de l'évolution de l'orchestration à travers la visualisation en temps-réel de la matrice des possibles. Cette interface fait partie de la catégorie « maître du jeu vers système ». Le contrôleur peut aussi agir sur la matrice des possibles, ce qui fait de lui potentiellement un orchestrateur « live ». Voici un exemple de contrôleur pour une pièce comportant deux groupes de personnes dans l'audience (sur les deux lignes) et huit groupes de patterns. La conception de cette interface n'est pas soumise aux mêmes contraintes que l'interface acteur. Elle s'adresse à une personne, une forme de chef d'orchestre ou de maître du jeu. Elle doit essentiellement satisfaire les critères de lisibilité et fiabilité sans enjeu fort sur la qualité de la conception graphique.

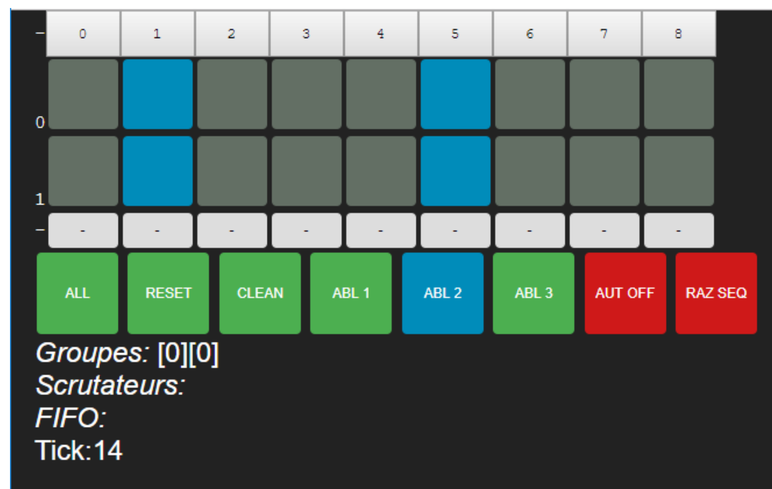



**Figure 16 : L'interface du contrôleur**

Une case bleue signifie que le groupe de patterns est actif, une case grise qu'il est inactif. Les numéros de groupe de patterns sont ceux définis dans le fichier de configuration de la pièce que nous avons vu p.76. En cliquant sur une case, on change d'état le groupe de patterns associé. En cliquant sur un numéro de pattern, on change toute la colonne et donc tous les groupes. Le bouton « all » permet d'activer tous les groupes de patterns pour tous les groupes d'utilisateurs. Le bouton « reset » désactive tous les groupes de patterns pour tous les groupes d'utilisateurs. Le bouton « clean » permet de vider toutes des files d'attente. Les boutons ABL1, ABL2, ABL3 permettent de sélectionner trois orchestrations différentes. Nous avons prévu de pouvoir charger trois automates, donc trois orchestrations, en même temps dans une session Skini. Ceci peut permettre de tester trois orchestrations sur la même base de patterns sans avoir besoin de relancer une session Skini par exemple. Une fois cliqué, le boutons « auto OFF », se transforme en bouton « auto ON ». Il permet de lancer ou d'arrêter l'automate, c'est lui qui émet les signaux start et stop de l'automate HipHop.js. Le bouton « RAZ SEQ » est utilisé pour une remise à zéro de tous les séquenceurs distribués que nous verrons dans un chapitre ultérieur. Les autres champs concernent des informations sur le déroulement de la performance. « Groupes » nous donne le nombre de participants dans chaque groupe de participants. « Scrutateurs » nous donne le nombre de clients de type « Scrutateurs » que nous verrons dans le chapitre suivant. « FIFO » donne un état sur la charge des files d'attente. « Tick » est le compteur du temps au rythme du tempo pour la pièce en cours.

Le contrôleur est l'interface nécessaire dans toutes les configurations de Skini, que ce soit en simulation ou pour une performance. Si l'application Skini tourne et que le simulateur est lancé, rien ne se passera tant que le contrôleur n'aura pas sélectionné un automate et activé « AUTO ON » par exemple. Le contrôleur communique avec le serveur au moyen de websockets.

#### *7.1.2.4 Le scrutateur*

L'interface *scrutateur* est issue des conclusions de nos premiers concerts avec Skini. Le comportement de l'audience n'est pas régulier, nous avons des individus très actifs, d'autres moins. Nous avons des approches différentes vis-à-vis de l'interaction, certains préfèrent jouer avec le système, d'autres préfèrent rester spectateur.

Initialement le scrutateur s'adressait à des personnes souhaitant donner un avis sans pour autant être acteur, il s'agissait de mettre l'accent sur la *simplicité* au détriment de la *fiabilité*. En effet les retours des actions des scrutateurs sont moins visibles que ceux des acteurs. L'idée était d'associer les scrutateurs au chef d'orchestre/maitre de jeu donc de se positionner dans la catégorie des interfaces du type « individu vers maître du jeu ». C'est-à-dire que les scrutateurs émettaient des avis que le chef d'orchestre pouvait prendre ou non en compte en agissant sur le contrôleur. Le scrutateur n'a pas été conçu initialement pour dialoguer avec l'automate HipHop.js. Il s'agissait d'un élément pouvant contribuer à la modification manuelle de l'orchestration sur scène.

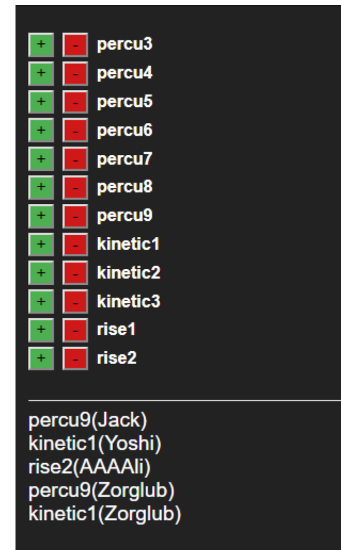



**Figure 17 : Le scrutateur**

Une deuxième étape pour le scrutateur a été de lui permettre d'agir sur la matrice des possibles au même titre que le contrôleur, donc passer dans la catégorie « individu vers système ». Ceci permet à certaines personnes de l'audience d'influer sur l'orchestration en cours en même temps qu'une orchestration sans pour autant interagir informatiquement avec elle via HipHop.js.

La Figure 17 : Le scrutateur, donne un exemple d'interface. Chaque ligne correspond à un groupe de patterns. Le « + » signifie que l'utilisateur souhaite que l'on active le groupe, le « - » qu'on le désactive. Les informations sont reçues sur le contrôleur, et selon le paramétrage du système, activent ou non la matrice des possibles.

Le scrutateur n'est pas d'une utilisation facile. Il ne fonctionne que dans des cas précis où il est aisé pour quelqu'un dans l'audience de repérer des groupes de sons. Par ailleurs il est en « conflit » avec l'automate puisqu'il peut décider de choix différents. Il offre des opportunités intéressantes, notamment dans un usage conjoint avec le simulateur. Il étend considérablement la capacité d'intervention de l'audience sur le déroulement de la pièce. Plus qu'un complément de l'acteur, le scrutateur doit être utilisé par le compositeur dans des contextes assez différents de ceux que l'on peut imaginer dans des orchestrations HipHop.js activées par l'audience, comme nous les avons vues dans les exemples précédents d'orchestration. Plus orienté vers des travaux sur l'interaction que sur l'orchestration avec HipHop.js, l'utilisation systématique du scrutateur est une piste de recherche à part entière que nous n'avons pas encore exploitée intensivement.

#### *7.1.2.5 Le configurateur*

Le configurateur fait partie de la famille des outils de contrôle du compositeur, de la catégorie « maître du jeu vers système ». Il permet d'envoyer directement des commandes MIDI durant une session Skini en respectant les conventions des fichiers de configuration. Notamment la numérotation des commandes qui est limité dans la norme MIDI qui n'autorise que 128 commandes sur un canal et que 16 canaux sur un port MIDI. Pour éviter de gérer des changements de canaux dans définition des patterns, Skini convertit automatiquement la commande Skini « note » en une commande MIDI comprenant la note et le canal. Skini autorise jusqu'à 2048 commandes de patterns, soit tout ce qui peut être transmis sur 16 canaux de 128 notes.

Le configurateur permet aussi d'émettre des Control Change MIDI immédiatement depuis Skini. Ceci permet d'intégrer rapidement des commandes vers la *Digital Audio Workstation* concernant les transpositions ou d'autres paramètres de contrôle de synthétiseur par exemple.

#### *7.1.2.6 L'interface globale*

Une des grandes difficultés des systèmes de musique interactive est la motivation du public. Depuis que la musique n'a plus comme fonction essentielle de renforcer la cohésion d'un groupe au moyen de la danse ou de la cérémonie religieuse, nous avons pris l'habitude d'une relation personnelle et passive à un événement musical. La plupart du temps, nous assistons à un concert, nous n'y participons pas. Nous sommes donc culturellement assez peu enclins à intervenir sur un déroulement. Il faut donc trouver les moyens non seulement de faciliter l'accès à la participation, mais aussi trouver des façons de motiver cette participation pour que le concept de *durée* prenne tout son sens. Hormis l'effet découverte qui peut fonctionner pendant un moment, il faut trouver des moyens pour que l'audience se sente le plus possible partie prenante de la création et de l'exécution de l'œuvre. Nous avons vu que le client « acteur » donnait des informations sur l'état de la pièce en temps-réel, mais notre sentiment est que cela n'est pas suffisant. Pour se sentir concernée, l'audience a besoin de plus que l'effet de surprise et l'état du système. Il lui faut des perspectives et qu'elle soit consciente de son rôle à jouer sur le devenir de la performance. Le smartphone est un superbe outil d'interaction, mais la taille de son écran est une vraie contrainte, d'autant plus qu'il est déjà utilisé par la sélection des patterns ou des choix d'orchestration. Nous avons donc estimé que l'utilisation d'un média de communication de la catégorie « système vers le groupe » était nécessaire. Dans la mesure où nous sommes dans un contexte de performance en salle, l'utilisation d'un grand écran, ou de plusieurs écrans, est apparue comme une solution naturelle et simple à mettre en œuvre. Néanmoins une fois le média défini, nous sommes encore loin d'avoir trouvé ce qui pourrait motiver l'audience.

**Figure 18 : Texte et pattern**

Nous avons pensé qu'il y avait ici plusieurs pistes à explorer. Une qui consisterait à introduire une autre dimension artistique à la performance, à travers du

texte ou des images. Une autre piste qui consisterait à penser l'orchestration sous la forme d'un scénario, peut-être même d'un jeu, dont le déroulement serait affiché sur le grand écran.

Parmi les scénarios qui consistent à donner une autre dimension, il y en a un assez simple à mettre en œuvre et que nous avons testé. Il consiste à associer à chaque pattern non pas un texte correspondant à une description de pattern, mais une courte phrase. Chaque fois qu'un pattern est activé, la phrase lui correspondant est affichée sur l'écran comme un texte qui défile. De cette façon la sélection de pattern construit un texte qui est partagé par l'ensemble de l'audience. L'idée est donc de faire évoluer l'orchestration en fonction de la perception littéraire des textes associés aux patterns. Ici le compositeur peut s'associer ou non avec un librettiste pour la conception de son œuvre. Nous avons mis en œuvre cette fonctionnalité à l'aide de *Processing* [73]. La Figure 18 : Texte et pattern, est une copie de grand écran où le texte est généré dynamiquement en fonction des patterns.

Parmi les autres pistes du type « scénario », nous avons mis en place une méthode d'affichage de l'orchestration, qui permet de visualiser les groupes de patterns et des liaisons entre les groupes. Ainsi l'audience peut voir l'ensemble ou une partie de l'orchestration, comprendre les chemins possibles dans le déroulement et même agir sur les directions que peut prendre la pièce. Nous avons rencontré des paramètres liés à l'affichage de l'orchestration dans le chapitre sur la « configuration des patterns et des groupes » p.76. Nous avons aussi rencontré des éléments concernant les trajets possibles dans l'orchestration dans les exemples de code de l'orchestration de l'opus1, comme par exemple :

```
59.                    hop {hop.broadcast('alertInfoScoreOFF');}
60.                    hop {hop.broadcast('addSceneScore', 2 );}
```

où la ligne 59 désactive l'affichage de la partition, et où la ligne 60 commande l'affichage d'une scène particulière correspondant à une orchestration. Pour mettre en œuvre cet affichage de type scénario nous avons utilisé un client HTML. Ainsi ce client peut être utilisé sur n'importe quel terminal et grand écran associé à un navigateur HTML.

La Figure 19: Un affichage de scénario d'orchestration, montre une orchestration en train de se dérouler. En l'occurrence, il s'agit de la session « échelle » de l'opus1. Les groupes répétitifs ont des bords arrondis, les réservoirs des bords en angle droit. Les réservoirs rétrécissent en fonction de la disparition des patterns. On voit sur la figure que « trompettesEchelles1 » a disparu par exemple. Les groupes répétitifs ont leurs bordures qui s'épaississent en fonction du nombre de sollicitations. Sur la figure, le groupe « cellosEchelle » a un bord qui est plus épais qu'au début de la performance. Les groupes de patterns sont noirs quand ils sont inactifs. Notons que dans la partie inférieure de l'écran, l'affichage donne un état des files d'attente qui peut aider à voir la dynamique en cours.

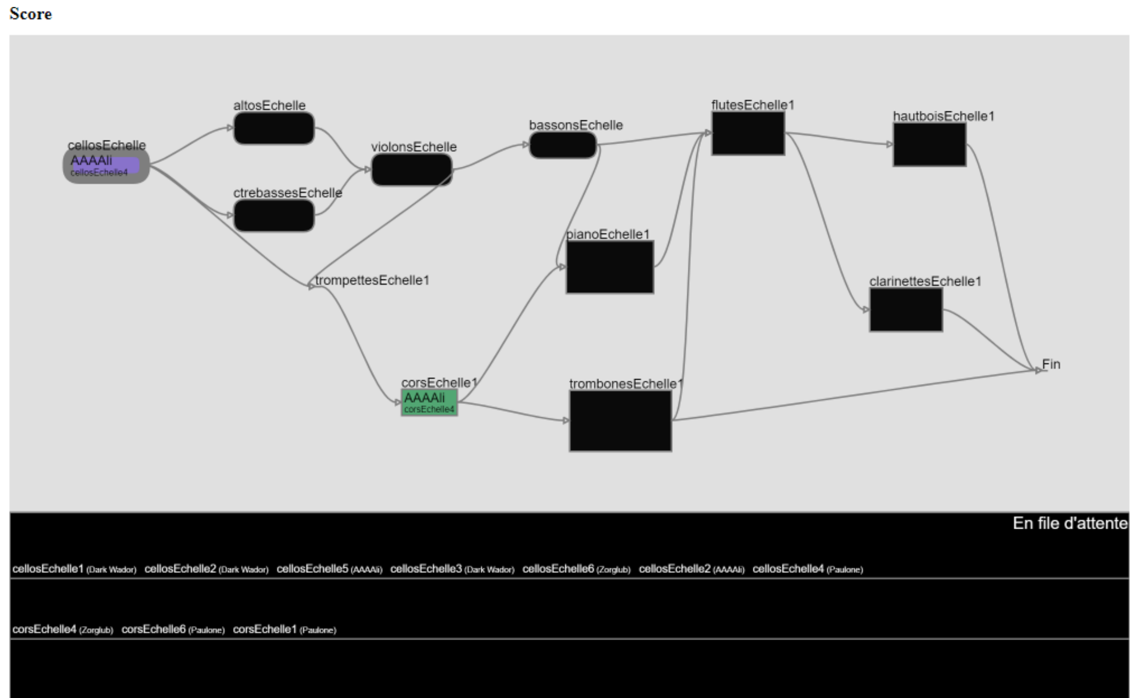



**Figure 19: Un affichage de scénario d'orchestration**

Techniquement, la définition de l'affichage de l'orchestration se fait au moyen du fichier de configuration que nous avons évoqué dans le chapitre « Réalisation d'orchestrations par automate », lors de la configuration des groupes de patterns. Ce mode d'affichage peut être utilisé pour faire passer des messages d'information ou poser des questions à l'audience. Nous avons un exemple de ce type d'usage dans le code de l'orchestration de l'Opus1 dans le chapitre sur les réalisations d'orchestrations en section Opus 1, p.82.

#### *7.1.2.7 Les musiciens*

Nous avons vu que notre plateforme expérimentale pouvait s'adresser à une audience en interaction avec des musiciens. Le dialogue avec des musiciens repose globalement sur les mêmes principes que ceux exposés pour activer une DAW. Nous avons cependant dans ce cas quelques différences. La première est que les patterns s'expriment sous formes de partitions ou d'instructions pour les musiciens. Ceux-ci doivent donc être munis chacun d'un terminal, tablette ou smartphone par exemple, capable de se connecter au serveur Skini pour afficher les patterns. L'autre différence relève de la gestion des files d'attente. Contrairement à un ordinateur un musicien a besoin de savoir ce qu'il doit jouer ou faire un peu en avance. Il n'est donc pas possible d'afficher un pattern juste au moment où celui-ci démarre. Dans le paramétrage de la pièce, le compositeur devra définir un décompte en pulsations avant que le premier élément d'une file d'attente soit joué. Si une file d'attente possède plus d'un élément, nous affichons le pattern en attente, c'est-à-dire celui qui vient juste après

**Figure 20: Interface pour musicien**

celui en cours. Quand un pattern est terminé, l'affichage change pour mettre sur la première ligne le pattern à jouer et en dessous le suivant. La Figure 20, donne l'exemple d'un pattern en cours de lecture et du pattern qui le suit. L'interface affiche la pulsation et le décompte des pulsations de la mesure en cours (« 2 » dans la figure). A chaque début de pattern l'écran flash pour bien signifier le départ. Pour le décompte précédent le premier pattern de la file d'attente, nous avons choisi de changer la couleur du fond pour bien signifier au musicien qu'il s'agit du premier pattern. Notre interface est volontairement la plus simple possible, elle ne joue que sur des couleurs de fond et un métronome-compteur contrôlé par le serveur. Le choix de n'afficher que deux patterns, relève aussi de cette volonté de simplicité.

L'utilisation de cette interface n'exclut pas la participation d'un chef d'orchestre. Il peut en effet y avoir de légers décalages dans les métronomes, provoqués par les latences du réseau. Nous n'avons pas mis en place de synchronisation de type NTP pour nos premiers essais, ce sera une piste de développement à creuser. Le chef d'orchestre peut donc avoir pour rôle de synchroniser les musiciens tout en agissant sur les nuances. Notons que pour qu'un chef d'orchestre puisse agir sur le tempo, il sera nécessaire d'adjoindre un autre outil du type Antescofo par exemple.

L'utilisation de notre plateforme avec des musiciens est parfaitement compatible avec son utilisation associée à une DAW. Il est donc possible de créer des pièces mixtes. Naturellement l'utilisation de la plateforme sans électronique et sans contrôle dynamique du tempo rend inutile l'adjonction d'une DAW à la plateforme. Celle-ci peut fonctionner alors uniquement avec Hop.js et HipHop.js.

En termes d'interaction, la conception d'orchestrations pour des musiciens laisse présager des cas d'usage intéressants. On peut imaginer des patterns extrêmement précis dans leur définition, mais aussi des patterns plus libres donnant des indications générales comme des listes d'accords pour des musiciens de Jazz, des indications d'atmosphères, des définitions de modes ou de types de rythmes pour des musiques du monde, des textes à lire, etc.

### 7.1.3 Le séquenceur distribué

Les travaux concernant ce que nous nommons le *séquenceur distribué* font suite au premier spectacle réalisé avec Skini, Golem, dans le cadre du MANCA 2017. Lors de cette première expérience d'interaction avec le public, nous avions enregistré le comportement des participants sous forme de traces de façon à pouvoir dégager des éléments sur leurs comportements. Cette analyse fait l'objet d'un autre chapitre, mais reprenons ici une conclusion majeure. *Les individus dans l'audience peuvent avoir des comportements très différents correspondant à des niveaux de compréhension variables et des connaissances musicales très diverses*. Il serait donc naturel de penser à des solutions d'interactions correspondant à ces différences de profils. Nous nous sommes donc posé la question : serait-il possible de laisser l'audience produire des patterns ? Comment pourrait-on s'y prendre ? C'est ainsi que nous avons eu l'idée de proposer un outil de création de patterns, dédié au public : le séquenceur distribué, qui est un outil de la catégorie « individu vers l'organe de production du son ».

Imaginer un outil de création pour l'audience posait plusieurs types de question. Quelle forme donner à l'interface ? Quelle méthode proposer pour l'édition et l'écoute des patterns ? Quelle serait la bonne façon d'intégrer les patterns dans une performance ?

Nous nous sommes tout d'abord attachés à résoudre la question de l'interface et de l'édition pour aborder la question de l'intégration dans une performance dans un deuxième temps. Il s'agissait de construire une interface ne nécessitant pas un long temps d'apprentissage. Dans le domaine de l'édition de patterns relativement courts, de nombreux travaux intéressants ont été réalisés depuis

les concepteurs de boite à rythme jusqu'aux logiciels à base de clip, ou les matériels *Push* d'*Ableton* ou *Maschine* de *Native Instruments* [77] par exemple. Sans chercher à reproduire un produit du marché, nous avons mis en œuvre certains concepts éprouvés, pouvant fonctionner sur un smartphone ou une tablette.

La deuxième question importante était celle de comment intégrer des patterns conçus par l'audience dans une performance ? Et tout d'abord comment synchroniser ces patterns ? Car si le comportement de Skini, son orchestration, dépend de durées définies par l'audience, nous avons vu que les patterns eux suivaient la voix traditionnelle de l'assujétissement au tempo et donc à une horloge. Deux problèmes de synchronisation se posaient donc. Le premier est celui de l'exécution des patterns sur le système central, le deuxième est celui de la synchronisation des éditeurs sur les terminaux. Notons ici que nous ne souhaitions pas remettre en cause le principe de base de la plateforme Skini, qui est que les terminaux ne sont pas les producteurs du son, mais des contrôleurs. Après plusieurs semaines de tâtonnements nous avons conçu un système où les patterns sont stockés et exécutés par le serveur, et où l'affichage des terminaux est synchronisé par une version allégée du protocole *Network Time Protocol* (NTP). Nous avons mise en œuvre une solution proche de celle décrite dans l'article [49] qui traite de la synchronisation de terminaux mobiles au moyen des Web Audio API.

La synchronisation en place, l'exécution des patterns sur le système central ne va pas sans poser quelques problèmes. En effet, la question de l'écoute en cours d'édition demeure. Ce n'est pas un problème insoluble, il est possible d'intégrer des échantillons sonores que le terminal pourrait jouer en local pour que le concepteur puisse écouter son travail avant de le diffuser. C'est néanmoins un travail assez lourd, car il suppose d'avoir des échantillons qui correspondent aux sons produits par le système central. Il faut donc reproduire sous forme d'échantillons tous les instruments que le musicien souhaite mettre à la disposition directe de l'audience. Nous avons préféré dans un premier temps valider nos concepts avant de nous lancer dans ce travail fastidieux. Le séquenceur distribué est composé de deux éléments principaux : le client qui comprend l'interface de l'utilisateur et le serveur qui exécute et affiche les patterns en cours de conception.

#### *7.1.3.1 Le client*

Initialement nous ne savions pas quel serait le niveau de finesse pertinent pour un outil de ce type. Il s'agissait en effet de trouver un équilibre entre un système accessible et suffisamment riche pour permettre de créer des résultats musicaux non triviaux. Nous avons abouti[6] à une interface du client du séquenceur qui ressemble à un PAD (Figure 21 : L'interface du séquenceur distribué), dont chacune des cases représente une note. La durée affectée à une note est choisie à partir des symboles, noire, croche, double-croche… La dimension du PAD est redéfinie à chaque changement de durée de note. Ceci est un des points différentiant d'un PAD matériel qui garde naturellement une taille fixe. Il est possible d'assigner une vélocité à une note au moyen d'un curseur.

L'utilisateur peut choisir un instrument parmi une liste prédéfinie (ici drum, lead, bass, solo, flute, riser). En changeant d'instrument le pattern affiché demeure ce qui permet de le jouer dans différentes configurations sonores. Le bouton « erase » efface le pattern. La colonne de gauche affiche des caractères signifiant les hauteurs. Elle est paramétrée pour chaque instrument. Ici nous voyons des noms de note en notation germanique, on pourrait avoir un texte décrivant les éléments d'une

[6] En matière de développement, la partie client représente 1300 lignes de code JavaScript. Elle a fait l'objet de plusieurs versions de façon à évaluer plusieurs mécanismes de synchronisation de l'affichage avec le serveur.

percussion comme « tom », « clap », « caisse claire3 … Le compositeur est libre de la définition de la configuration. La colonne verte est la barre de temps, qui indique où en est le serveur dans l'exécution du pattern. Le bouton OFF/ON permet de lancer l'exécution du pattern sur le serveur. Cette interface permet d'enregistrer et de charger des patterns.

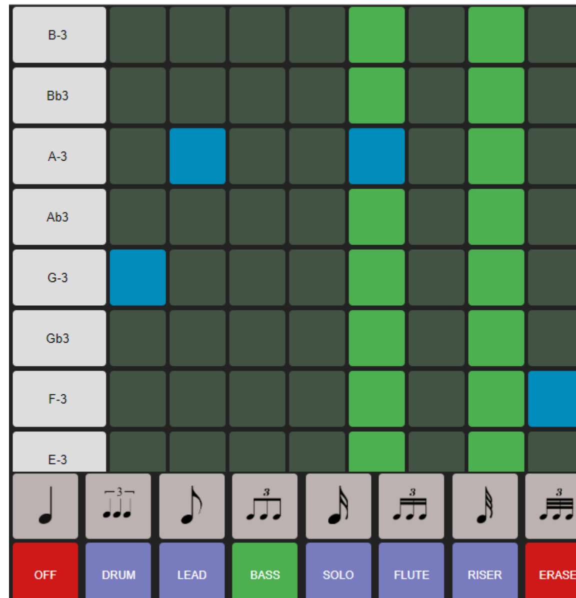



**Figure 21 : L'interface du séquenceur distribué**

Notons que ce type de PAD introduit une contrainte sur la définition des durées. Il n'est pas encore possible d'avoir des notes avec des durées supérieures à la noire. C'est une limitation qui peut être dépassée par un mécanisme de sélection des durées plus complexe que celui présenté ici. Les différents tests que nous avons réalisés avec la plateforme n'ont pas encore mis en évidence que cette limitation était problématique. La vélocité des notes jouées peut-être assignée pour chaque note par un curseur. Il est possible d'enregistrer une séquence et d'en charger une autre.

Ces fonctions de base sont suffisantes pour créer des patterns ayant déjà un bon niveau de complexité. Nous avons pu constater cependant qu'il s'agit d'un outil qui demande une certaine formation musicale. Pouvoir décrire une séquence musicale en termes de durée et de hauteur, même sans une écriture solfégique complexe n'est pas à la portée de tous. Cette interface fonctionne bien dans deux cas : quand la performance est organisée de façon à laisser le temps de comprendre et de tester l'utilisation du PAD, quand les utilisateurs ont déjà manipulé un PAD, ce qui se produit assez régulièrement pour les générations Y et Z. Nous verrons dans les retours d'expérience que cette interface constitue un outil pédagogique efficace.

#### *7.1.3.2 Le serveur*

Le serveur du séquenceur a plusieurs rôles :

- C'est lui qui stocke et exécute les patterns produits par les clients. C'est-à-dire que c'est lui qui possède les informations MIDI de pilotage des synthétiseurs. Il n'y a pas de dialogue direct entre les clients et les synthétiseurs. Après de nombreux essais avec des scénarios de contrôle différents, dont le contrôle des synthétiseurs directement par les clients en les synchronisant, nous avons retenu un mode de fonctionnement centralisé car il était le plus simple et le plus fiable.

- C'est lui qui gère l'affichage global des patterns en cours de conception et d'exécution. Il nous a paru important que tous les participants aient une vue de ce que chacun produisait. Le serveur affiche donc une sorte de partition en cours de conception qui donne une vue des patterns superposés (Figure 22 : Affichage du serveur séquenceur). Chaque participant s'identifie par un code de couleur correspondant au pseudo qu'il a saisi au moment de se loguer sur le système. Un pattern en cours d'exécution est suivi par un train de bulles graphiques qui signifie son activité. Une barre de temps défile de façon synchrone avec la barre de temps du client. Voici une vue affichée par le serveur.

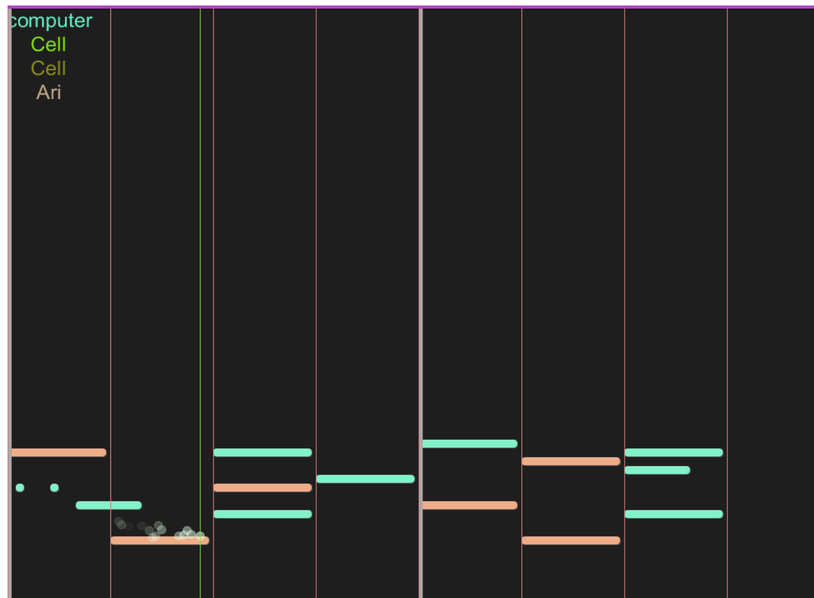



**Figure 22 : Affichage du serveur séquenceur**

Il y a ici deux mesures à quatre temps, deux utilisateurs ont saisi des notes, on les différencie par les couleurs. On devine que l'utilisateur « Ari » est actif car des particules s'affichent au-dessus de la note en rose du deuxième temps de la première mesure.

### *7.1.3.3 Evolutions du séquenceur distribué*

Initialement conçu dans le but d'ajouter un niveau d'interface dans une performance Skini avec une audience nombreuse, le séquenceur distribué a développé une histoire personnelle pour s'éloigner de notre question initiale sur l'utilisation des langages réactifs synchrones. Ceci pour plusieurs raisons. D'une part grâce à la participation d'un stagiaire de l'ENS (Thibaud Ardoin) qui a optimisé les mécanismes de synchronisation et qui a pressenti que le séquenceur de notre plateforme pouvait être une application en soi. D'autre part, parce que sollicité par le CIRM pour un projet pédagogique pour la SACEM, nous avons décidé d'utiliser le séquenceur distribué non pas dans un contexte de performance, mais dans la phase de conception d'une création musicale. L'idée faisant son chemin, nous nous sommes demandé si ce développement pouvait avoir un quelconque intérêt commercial. Nous avons alors pu bénéficier du stage d'une étudiante en école de commerce IPAG/Politecnico de Turin (Ariana Corvi) qui a réalisé une première étude comparative et d'opportunité de la solution.

Le séquenceur distribué n'est pas un outil de masse. Il est destiné à des groupe de 5 ou 6 individus au maximum, non pas pour des raisons techniques, mais pour permettre une bonne synergie entre les participants. Ceci signifie que dans un spectacle de masse, on n'attribuera les séquenceurs

distribués qu'à des individus motivés et comprenant les principes de la conception d'un pattern en boucle à partir d'un PAD.

Nous avons pu constater qu'en petit groupe l'outil fonctionnait correctement. Lors du projet Fabrique à Musique de la SACEM, nous avons rencontré des difficultés liées à l'utilisation d'une interface relativement complexe, mais aussi liées à l'écriture musicale par des scolaires de 6$^{ème}$. Du côté de notre approche marketing nous avons eu des retours encourageants, même s'il reste un certain nombre de point d'ergonomie à revoir, comme l'affichage des hauteurs, la saisie des durées longues ou des intensités.

Même si nous nous sommes éloignés de notre question initiale relative à l'expressivité du temps au moyen d'un langage réactif synchrone avec cette interface du type « individu vers organe de production du son », un retour est possible vers une interface du type « individu vers système ». Il est en effet possible d'intégrer le séquenceur distribué dans une orchestration en intégrant les patterns du séquenceur distribué dans des groupes répétitifs ou des réservoirs mis à la disposition d'une audience. Ceci permet d'imaginer qu'il est possible de mettre en œuvre des performances initialement sans pattern où la création complète de la pièce se ferait sur scène. Si cette idée n'est pas extrêmement difficile à mettre en œuvre du côté des patterns, car il s'agit d'enregistrer les patterns sous plusieurs formes pour les mettre à la disposition de l'audience. Elle pose néanmoins la question de savoir s'il est possible d'imaginer une orchestration sur scène ? Techniquement HipHop.js permet d'intégrer du code en cours d'exécution, il est donc possible d'imaginer d'enrichir une orchestration minimaliste en cours de performance. La difficulté réside dans la façon dont l'audience, ou du moins certains membres de l'audience, pourrait agir sur l'orchestration en temps-réel. Nous n'avons pas encore trouvé de solution satisfaisante à cette difficulté. La position extrême d'ouverture totale, nous conduit dans une approche musicale très différente de notre projet initial. On peut cependant imaginer des étapes intermédiaires où une pièce pourrait intégrer, en plus de ses propres patterns, d'autres conçus par des membres de l'audience, où l'audience modifierait la structure de l'orchestration, etc.

Le séquenceur distribué nous a ouvert des pistes de recherche nouvelles qui, tout en étant conformes à nos travaux sur l'interaction, nous ont un peu éloigné de la mise en œuvre d'une performance à l'aide d'une orchestration. Pour revenir à notre sujet principal, nous avons vu en début de ce chapitre sur la mise en œuvre de la durée, que nous avions besoin d'une solution permettant au compositeur d'avoir une première évaluation de ce que pourrait être la performance qu'il conçoit. Nous allons à présent traiter ce besoin de simulation.

## 7.2 Le simulateur

Le simulateur est un outil important de notre méthode de composition et de la phase d'expérimentation de l'utilisation de HipHop.js. Il est possible et nécessaire de vérifier le déroulement d'une orchestration *manuellement*. Le compositeur peut devenir un exécutant de son orchestration en activant une ou plusieurs interfaces utilisées par l'audience, mais il sera limité dans cet exercice. Le principe du moteur aléatoire, ou simulateur de Skini, est de passer à une échelle supérieure et de créer un mécanisme se comportant comme une audience qui sélectionnerait les patterns de façon aléatoire, une forme d'audience sans conscience. Bien qu'il ne s'agisse pas d'une véritable mise en œuvre de la durée, ce simulateur doit utiliser exactement les mêmes protocoles de communication avec le serveur qu'une personne dans l'audience.

Nous avons choisi de faire reposer son fonctionnement sur quatre principes :

- Interroger le serveur de façon aléatoire sur une plage de temps fixée.
- Sélectionner de façon aléatoire un pattern parmi ceux disponibles.
- Ne choisir d'activer que des patterns disponibles dans un temps inférieur à un délai d'attente fixé.
- Ne pas resélectionner le même pattern immédiatement. Attendre d'avoir réalisé deux autres sélections avant de sélectionner de nouveau un même pattern.

L'algorithme de simulation suit donc le chemin :

1. Initialisation. Elle se fait par un message transmis par le serveur au lancement de l'automate.
2. Demande de la liste des patterns disponibles par le simulateur.
3. Transmission par le serveur de la liste des patterns en cours.
4. Présélection au hasard d'un pattern dans la liste.
5. Demande par le simulateur du délai d'attente au serveur pour le pattern présélectionné.
6. Transmission par le serveur du délai pour le pattern présélectionné. Comme nous l'avons vu au sujet des « files d'attente » p.70, un pattern n'est pas nécessairement activé à l'instant même où le demande un client. Rappelons que les informations sur les délais d'attente sont transmises systématiquement à l'audience pour chaque pattern.
7. Si le délai est en dessous de la limite, le pattern est effectivement sélectionné à un instant choisi au hasard sur une plage fixée en paramètre. Le pattern sera effectivement activé par le serveur quand ce sera son tour dans la file d'attente.
8. Retour en 2 à la demande de la liste des patterns disponibles

Nous voyons que le comportement du simulateur est lié à deux paramètres :

- La limite de la durée d'attente associée à un pattern présélectionné.
- L'instant de la sélection effective par le simulateur.

L'activité du simulateur et donc ce qui correspond à une taille d'audience, dépend de la plage de temps sur laquelle sera choisie de façon aléatoire l'instant de l'activation. Cet instant est calculé de la façon suivante entre une tempoMax et une tempoMin :

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

Plus la valeur de tempoMax sera faible, plus le comportement se rapprochera d'une audience nombreuse. Le tempoMin est un moyen d'éviter des successions d'activations trop rapides. La limite de la durée d'attente part du principe qu'une personne dans l'audience ne demandera pas l'activation d'un pattern qui demanderait trop d'attente avant d'être joué.

Il n'y a pas de règle absolue sur la façon d'ajuster les paramètres. Le simulateur déroule son comportement sur une console.

```
--Trop attendre sec :  32   pour  vldo-1
--tempoInstant msec :  297
WS Recu : listClips 1ere ligne: vldo-I -12 Nombre clip dispo: 21
--Trop attendre sec :  32   pour  vldo-I -12
--tempoInstant msec :  180
WS Recu : listClips 1ere ligne: vldo-I -12 Nombre clip dispo: 21
Durée précédant le son ci-dessous: 180
startClip: 11  : cedo-9 par AAAAli
--tempoInstant msec :  410
WS Recu : listClips 1ere ligne: vldo-I -12 Nombre clip dispo: 21
--Trop attendre sec :  32   pour  vldo-2
--tempoInstant msec :  160
WS Recu : listClips 1ere ligne: vldo-I -12 Nombre clip dispo: 21
Durée précédant le son ci-dessous: 160
startClip: 12  : cedo-VI par Jack
--tempoInstant msec :  322
WS Recu : listClips 1ere ligne: vldo-I -12 Nombre clip dispo: 21
Durée précédant le son ci-dessous: 322
startClip: 13  : cedo-X par Paulone
--tempoInstant msec :  226
WS Recu : listClips 1ere ligne: vldo-I -12 Nombre clip dispo: 21
--Trop attendre sec :  32   pour  vldo-1
--tempoInstant msec :  450
WS Recu : listClips 1ere ligne: vldo-I -12 Nombre clip dispo: 21
Durée précédant le son ci-dessous: 450
startClip: 14  : pido-12 par Dark Wador
--tempoInstant msec :  486
WS Recu : listClips 1ere ligne: vldo-I -12 Nombre clip dispo: 21
--Trop attendre sec :  32   pour  vldo-III
```

C'est en suivant l'évolution des différents paramètres du système que le compositeur peut modifier les réglages du simulateur et évaluer le résultat de sa composition en fonction de différentes tailles d'audience. En situation réelle pour une audience de 100 personnes nous avons environ une activation par seconde par exemple. La console du simulateur permet de vérifier que les activations se font plus ou moins dans ce délai.

Le simulateur produit un résultat qui peut être enregistré et comparé à un schéma d'orchestration tel que ceux que nous avons vus au chapitre 4. La figure ci-dessous, Résultat d'une simulation dans Ableton Live, nous donne une visualisation issue d'une simulation de l'Opus1 enregistrée dans l'écran arrangement d'*Ableton Live*. Chaque ligne correspond à un instrument. Chaque petit bloc correspond à un pattern. Nous voyons bien les trois sessions dont les patterns sont identifiés par des couleurs différentes. Les patterns bleus correspondent à la session échelle, les patterns orange à la session chromatique et les verts à la session tonale.

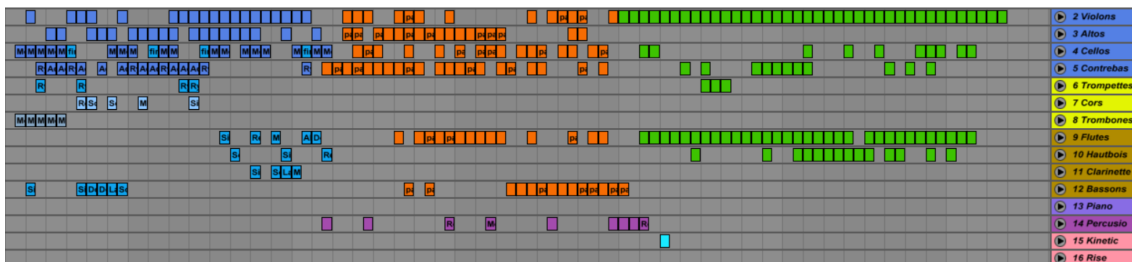



**Figure 23: Résultat d'une simulation dans Ableton Live**

Pour une analyse encore plus fine du résultat il est possible de produire une partition. En Figure 24: Extrait d'une simulation de l'Opus1, nous avons un extrait de la partition produite par une simulation de l'Opus1.

Skini n°1

Heidelein

Flûte
Hautbois
Trompette en Sib
Violon
Violoncelle
Contrebasse

Fl.
Hb.
Trp. (Sib)
Vln.
Alt.
Vlc.
Cb.

©Heidelein 2018

**Figure 24: Extrait d’une simulation de l'Opus1**

Il s’agit ici de début de la session tonale qui commence par l’attente de cinq sélections de patterns de violons comme nous le voyons sur la figure p.49 (Figure 7: Une orchestration avec sessions). Il s’agit bien de cinq *sélections* et non pas cinq *exécutions*. En effet, nous voyons que le hautbois et la flutes démarrent en mesure 3, alors que nous n’avons eu que l’*exécution* de 2 patterns de violons et non cinq. Ceci correspond au fait que le simulateur a déjà sélectionné 5 patterns de violons au bout de 2 mesures. Ceci est tout à fait normal pour un simulateur paramétré de façon à réagir rapidement. Si nous avions voulu attendre effectivement l’exécution des cinq patterns de violons, il aurait fallu paramétrer Skini différemment pour qu’il prenne en compte les exécutions et non les sélections. Si nous restons au niveau des violons, nous voyons que la « session tonale » dure le temps de la sélection de 15 patterns de violons (5 + 10 dans le schéma de l’orchestration) dont 10 sont en parallèle à 10 patterns de violoncelles et deux réservoirs (trompettes et Kinetic). En mesure 23 au violoncelle et en mesure 25 à la flute, les patterns ne font plus partie de la session tonale. Nous voyons qu’en effet nous avons rempli la condition logique des 10 patterns de violons « et » 10 patterns de violoncelles « et » fin des réservoirs. En effet, 12 patterns de violoncelles ont été joués avant ces mesures et le réservoir de trompettes a bien été vidé de ses trois patterns, mesures 8, 9 et 10. (Pour alléger la partition, nous n’avons pas affiché Kinetik).

Le simulateur se comporte donc comme un moteur capable d’activer le système Skini. Ceci signifie aussi qu’il peut fonctionner en même temps qu’une performance en interaction avec une audience. Il existe deux cas d’usage du simulateur durant une performance :

1) Renforcer une audience trop petite : il est possible dans certains cas de compléter une audience trop peu nombreuse par rapport à l’ampleur d’une orchestration, en injectant ponctuellement des phénomènes aléatoires. Ceci revient à déresponsabiliser en partie

l'audience. Il s'agit donc d'une manœuvre à utiliser avec précaution ou dans l'objectif particulier d'introduire une dimension chaotique.
2) Changer le rôle de l'audience : Le principe initial de la plateforme est de laisser le choix des patterns à l'audience selon une orchestration bien définie. Il est aussi possible de mettre en place un scénario totalement différent qui consiste à laisser le contrôle de l'orchestration à l'audience et l'activation des patterns au simulateur. Il s'agit d'un cas d'usage simple à mettre en œuvre, qui fait peu ou pas appel à HipHop.js, et qui peut permettre à un petit groupe de créer en « live » une pièce complète uniquement en sélectionnant des groupes d'instruments.

Nous avons encore un cas d'usage du simulateur hors performance qui est de produire de la musique automatiquement avec un contrôle « manuel » de l'orchestration via le « scrutateur ». C'est-à-dire en limitant les interactions à très peu d'individus, voir un seul. Ceci est un axe probablement intéressant pour des situations interactives du type jeu vidéo, visite de musée ou d'expositions. Des situations où le comportement de quelques individus contrôlerait l'orchestration via des capteurs de position ou de mouvement.

## 7.3 Conclusions sur l'interaction

En ce qui concerne l'interaction avec l'audience, nous ne considérons pas nos réflexions comme abouties pour plusieurs raisons :

D'une part notre cheminement dans ce domaine a été essentiellement expérimental dans le but de répondre à quelques principes de base dont : la *clarté*, pour une visibilité conforme à la musique en train de se faire ; la *fiabilité*, pour qu'à chaque action corresponde une même réponse du système ; la *lisibilité*, pour qu'à chaque action le système retourne une information ; la *simplicité* pour un apprentissage rapide ; la *vision à long terme* pour comprendre le sens global de la démarche musicale. Cette approche expérimentale nécessiterait encore quelques réalisations pour être validée et affinée. Il reste aussi à prendre en compte la qualité du graphisme des interfaces, qui aurait pu être un des critères de départ sur lequel nous avons choisi de ne pas mettre la priorité pour différentes raisons. Essentiellement parce qu'il s'agit d'un sujet difficile assez loin de notre recherche.

D'autre part, nous n'avons pas expérimenté tous les niveaux d'interactions possibles car, comme nous l'avons vu en début de chapitre, ils deviennent nombreux dès que le système s'enrichit. Parmi les pistes intéressantes non explorées nous avons les « interactions entre individus ou groupes d'individus » qui, en relation avec le séquenceur distribué, devraient créer des opportunités intéressantes en matière de création en petits groupes.

Par ailleurs, parmi les éléments qui complexifient l'interaction avec l'audience, nous avons le délai possible pour une activation de pattern lorsque nous avons plusieurs demandes sur un même instrument. Nous avons pris l'habitude de systèmes qui agissent assez rapidement à la suite de nos actions. Il n'est pas facile de faire comprendre que la prise en compte des sélections, et donc des choix de chacun, ne sont pas toujours instantanées. Faire prendre conscience à chacun qu'il fait partie d'un groupe de personnes et que ce groupe est soumis à des règles collectives est certainement un des défis de nos travaux. L'interaction reste donc un chantier ouvert qui bénéficie pleinement des apports de la souplesse de la programmation Web et de JavaScript.

En complément des questions d'interaction, nous avons traité le besoin de simulation. Au-delà de notre projet initial de validation d'un travail orchestration dans un contexte d'interaction avec

l’audience, c’est en mettant en œuvre un outil de simulation que nous avons réalisé que le moteur aléatoire produisait des versions cohérentes de nos pièces. C’est ainsi qu’en nous éloignant de notre projet initial lié à la durée, nous sommes entrés indirectement dans l’univers des systèmes de *musiques génératives*, c’est-à-dire des systèmes capables de produire des pièces musicales sans intervention humaine, sous forme de composition automatique ou de composition algorithmique [5,24,46]. La musique générative est un vaste domaine de recherche, fort actif. Dans son article « A Functional Taxonomy of Music Generation Systems » [40] l’auteur nous donne un état de l’art récent sur les systèmes de musique générative. Notons aussi l’important article de J. Ferandez et F. Visco qui donne un panorama très large des travaux autour de l’Intelligence Artificielle appliquée à la composition algorithmique [25]. Plus spécifiquement en ce qui concerne l’utilisation de « l’apprentissage profond » ou « deep learning » dans l’article [13] « Music Generation by Deep Learning - Challenges and Directions » de 2018 les auteurs font un point détaillé des technologies de réseau de neurones et des enjeux des différentes architectures (Restricted Boltzmann Machine, Recurrent Neural Network, Long Short Term, Generative Adversarial Networks…). L’utilisation de HipHop.js, et donc des automates, nous rattacherait à la famille des musiques « génératives combinatoires » à base de « synthèse concaténative » [47,85]. La composition à partir de combinaisons possède une longue histoire remontant à *l’Organum Mathematicum* d’Athanasius Kircher [21,26] au 17ème siècle qui permettait, entre autres, de créer des pièces de musique à partir de phrases musicales combinées de façon aléatoire, une sorte de Skini fortement simplifié avec son simulateur. Au 18ème siècle apparaissent les *Musikalisches Würfelspiel*, ou jeux de dés musicaux qui permettent de composer des pièces à partir de combinaisons aléatoires de fragments (l’équivalent de nos patterns). Les formes ouvertes des années 60, dont parle U. Eco, sont parfois rattachées à ce mouvement. Néanmoins l’introduction d’automates complexes comme support de création me semble un domaine peu exploré. La combinatoire se définit souvent comme l’étude de la configuration de collections finies d’objets, or la création d’automates est une façon de traiter des combinaisons mais en définissant un ordre qui n’est pas purement mathématique.

Notre système faisant appel à des processus stochastique, il est assez logique de le comparer à l’approche de I. Xenakis qui a largement basé ses travaux sur des techniques probabilistes comme nous l’avons vu au paragraphe « en-temps, hors-temps », p.26. Le simulateur de Skini repose sur un moteur aléatoire mais il ne se situe pas au même niveau que les processus utilisés par Xenakis et tel qu’il les décrit dans son ouvrage « Musique Formelle » [43]. En schématisant nos approches, nous pourrions dire que Xenakis utilise des processus stochastiques complexes pour créer les matériaux « hors-temps » de ses pièces et que la réalisation « en-temps » du geste musical est le travail subjectif du compositeur, qui ne suit alors pas de processus stochastique. C’est d’ailleurs ce qui fait l’intérêt des pièces de Xenakis. La simulation Skini inversement est basée sur des structures « hors-temps » totalement subjectives, les patterns d’une part et l’orchestration d’autre part. Mais le geste musical est concrétisé par un processus stochastique qui active un automate complexe. Aléatoire et subjectivité se combinent dans les deux cas en utilisant des savoir-faire scientifiques, mais pas les mêmes, ni de la même façon.

Nous avons traité l’ensemble des éléments nécessaires pour mettre en œuvre la durée à partir de l’expressivité du langage HipHop.js. Nous avons vu que pour rendre effective la durée il était nécessaire d’introduire des concepts complémentaires à ce langage. Il s’agit des patterns, des groupes de patterns, de différents mécanismes de synchronisation et d’interaction qui ont été complétés par un simulateur. Nous avons vu comment réunir ces éléments, nous avons aussi éprouvé quelques principes de composition par patterns et vu comment réaliser des simulations. Il nous reste à voir comment nous avons validé nos concepts à travers la mise en œuvre de performances

« réelles », c’est à dire avec de véritables audiences. Avant d’aborder nos retours d’expérience voyons comment nous avons organisé la plateforme expérimentale destinée à de véritables performances.

# 8 ARCHITECTURE TECHNIQUE

LES SERVICES WEB 124

LES COMMANDES DU SON 124

LA DIGITAL AUDIO WORKSTATION 125

L’AFFICHAGE 125

L’ARCHITECTURE DU SEQUENCEUR DISTRIBUE 125

LE RESEAU 125

CONCLUSION SUR L’ARCHITECTURE TECHNIQUE 126

Dans les chapitres précédents nous avons vu comment d'un besoin d'expressivité de la durée, dans un sens proche de celui que Bergson donne à ce terme, nous avions cherché à mettre au point une méthode de composition pour la musique interactive basée sur les concepts de pattern, de groupe de patterns et d'orchestration. L'orchestration repose sur le langage réactif synchrone HipHop.js complètement intégré dans une plateforme de développement Web, Hop.js, utilisant le langage JavaScript. L'ensemble de ces concepts prend vie au moyen d'une audience pouvant agir sur la composition musicale via des interfaces Web. Nous avons vu aussi que le compositeur peut vérifier en partie la cohérence de son travail via un simulateur. La mise à l'épreuve de la méthode de composition issue de notre recherche nécessitait la réalisation d'une plateforme permettant la réalisation de performances interactives à grande échelle et de simulations. Ce chapitre expose l'architecture de cette plateforme baptisée Skini.

La Figure 25: Architecture Skini, donne une vision globale de l'organisation du système. Nous avons les différents clients HTML5 pour l'audience, le séquenceur distribué, les acteurs et les scrutateurs. Ce sont des applications écrites en HTML5 et JavaScript en utilisant les fonctionnalités multi-tiers de Hop.js. C'est-à-dire la possibilité d'écrire les programmes des clients comme des services développés sur le serveur. Hop.js permet d'éviter le découpage des programmes entre client et serveur comme il est d'usage de le faire avec Node.js [69] par exemple. Ceci facilite l'écriture et le débogage. De plus Hop.js offre un certain nombre de services de communication très simples à mettre en œuvre comme le *broadcast* qui diffuse un message à tous les clients connectés au serveur par exemple.

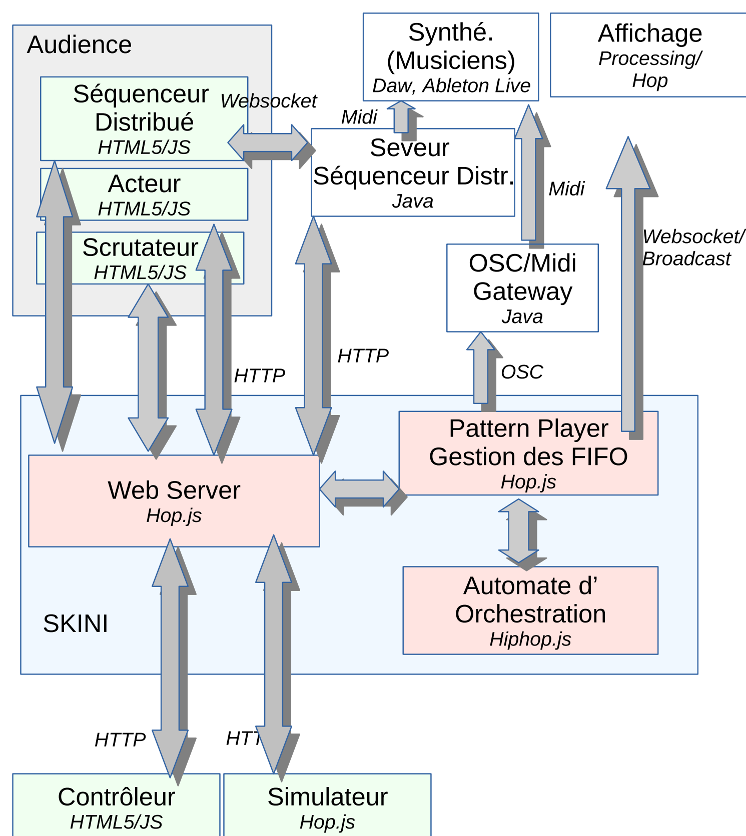



**Figure 25: Architecture Skini**

Le contrôleur est développé de la même façon que les clients pour l'audience. Le simulateur est une application Hop.js. Il n'est donc pas embarqué dans un client HTML. Nous avions précédemment vu une image de la console utilisée pour le simulateur.

## 8.1 Les service web

L'application Skini est un serveur web, écrit en Hop.js. Il a cinq fonctions principales :

1. alimenter les différents clients,
2. gérer l'orchestration,
3. gérer les files d'attente (FIFO),
4. émettre les commandes vers la production du son,
5. et transmettre les différentes informations d'affichages globaux.

La communication entre le serveur et les différents clients utilise les mécanismes de *websocket* [90] et de *broadcast* de Hop.js. Le module central du serveur est la gestion des messages échangés au moyen des *websockets*. Le fonctionnement de Skini peut donc être compris à partir des protocoles utilisés entre le serveur et les différents types de clients.

## 8.2 Les commandes du son

Les communications entre le serveur Skini et la production du son, ici une *Digital Audio Workstation* (DAW) utilisent le protocole *Open Sound Control* [92] (OSC) qui est une couche de présentation développée au-dessus de TCP/IP pour transmettre des messages de contrôle. Il s'agit d'un protocole développé par le CNMAT [6] comme alternative à MIDI. Nous utilisons ce protocole non seulement car Hop.js ne possède pas de librairie permettant d'envoyer directement des commandes MIDI, mais surtout parce que la mise en œuvre d'OSC sur le serveur facilite la distribution de Skini sur plusieurs plateformes connectées en TCP/UDP/IP. Avec OSC nous pouvons mettre en œuvre le serveur sur une machine Linux et les autres fonctions sur des machines de type *Windows* ou *macOs*. Cette architecture nécessite soit une DAW qui comprend OSC, soit une passerelle OSC-MIDI car toutes les DAW comprennent MIDI.

Après des investigations sur les passerelles OSC/MIDI disponibles sur le marché nous avons décidé de développer la nôtre. En effet, les passerelles OSC/MIDI que nous avons trouvées étaient compliquées à mettre en œuvre et dépendantes du système d'exploitation. Nous avons choisi de développer la passerelle OSC/MIDI (*Gateway*) avec la plateforme de développement *Processing* [73] initialement créée par le MIT, d'une part parce qu'elle est multiplateforme et que notre développement est ainsi facilement transportable sur Linux, Windows, ou macOs, d'autre parce que *Processing* possède des bibliothèques gérant les deux protocoles OSC et MIDI ainsi que la communication à base de *websockets*. Notons que *Processing* est un outil initialement plutôt destiné au développement graphique. Nous donc avons aussi utilisé cet outil, qui est basé sur Java, pour nos affichages. De plus, il possède aussi une version compatible avec JavaScript qui permet d'étendre nos développements à des navigateurs Web.

Cette passerelle joue un rôle important dans le fonctionnement général de Skini. La communication OSC/MIDI est bidirectionnelle. Hop.js comme les plateformes JavaScript du type de Node.js ne sont pas très stables au niveau temporel. Nous avons donc choisi de gérer la synchronisation à l'aide des signaux « MIDI sync » émis par la DAW. C'est donc la passerelle qui reçoit les signaux de synchronisation et retransmet une synchronisation adaptée au serveur Skini à l'aide de *websockets*. Ceci offre

aussi la possibilité de modifier le tempo de la DAW en temps réel en parfaite synchronisation avec Skini et le séquenceur distribué.

Le séquenceur distribué est fonctionnellement différent de la passerelle OSC/MIDI. Pour simplifier l'architecture logicielle les deux fonctionnent au moyen d'un seul programme. L'ensemble est mis en œuvre avec un code assez compact de 1000 lignes environ.

## 8.3 La Digital Audio Workstation

La DAW est la plateforme qui héberge les synthétiseurs et les patterns dans leur format MIDI ou fichier son. Nous l'utilisons aussi comme outil de conception des patterns. Nous avons choisi *Ableton Live* pour nos projets, un autre outil comme *Reaper* avec *Playtime* ferait aussi l'affaire, comme toute DAW capable de lancer des clips au moyen de commandes MIDI. De plus, *Ableton Live* apporte nativement des fonctions évoluées de contrôle MIDI, de mise en œuvre des transpositions de patterns, et de contrôle en *Live* avec l'interface *Push*. *Ableton Live* intègre *MAX/MSP* [71] qui est un outil des plus diffusés et des plus puissants actuellement sur le marché des DAW.

Bien que nous utilisions *Ableton Live* nous avons fait des essais sans cet outil en gérant directement les patterns depuis la passerelle OSC/Midi. Cet essai concluant d'un point de vue technique montre qu'une DAW n'est pas absolument nécessaire au fonctionnement de Skini. Nous n'avons cependant pas trop investigué dans ce sens, car il ne nous a pas semblé crucial d'un point de vue conceptuelle de passer un temps important pour refaire ce que les DAW du marché font déjà. De plus, activer les patterns depuis *Processing* signifie que les patterns ont été créés par ailleurs, soit avec un éditeur MIDI, soit une DAW.

## 8.4 L'affichage

Nous avons mis en œuvre deux méthodes d'affichage d'informations globales sur le déroulement de l'orchestration. Une première version sur *Processing* dans le cas de l'affichage de texte et une version en JavaScript avec la librairie P5.js [108] pour l'affichage dynamique des orchestrations. Il s'est agi ici de mettre en œuvre des techniques de programmation graphique qui n'ont aucune relation avec Hop.js ou HipHop.js. La programmation à l'aide de *Processing* est assez compacte, notre client d'affichage de l'orchestration occupe environ 1200 lignes de code.

## 8.5 L'architecture du séquenceur distribué

Le séquenceur distribué s'intègre complétement dans l'architecture de Skini. Le client charge une page HTML à partir du serveur Hop.js. Les échanges sur le contenu des patterns et la synchronisation se font à l'aide de *websockets* qui dialoguent essentiellement avec le séquenceur développé au moyen de *Processing* [73] en Java. La grande différence avec la simple activation de patterns est qu'ici nous avons des contraintes de type « temps-réel » et qu'il n'est pas possible de gérer ces contraintes en Javascript. C'est pour cela que l'essentiel des échanges du séquenceur avec le serveur est géré par *Processing.* Ceci donne des résultats satisfaisant pour des traitements temps-réels graphiques et sonores. Les échanges avec Hop.js concernent la configuration générale du contexte, le nombre de mesures allouées aux patterns, le tempo et le chiffrage des mesures.

## 8.6 Le réseau

Le système repose sur un réseau TCP/IP. Il n'est pas très gourmand en bande passante car les communications consistent en échanges de messages utilisant les *websockets* et en requêtes HTML. Les

seuls transferts de fichiers concernent les fichiers son des patterns lors de la pré-écoute sur les clients. Dans nos réalisations chacun de ces fichiers occupe quelques centaines de kilo-octets.

Lors de nos performances, nous avons utilisé un réseau Wifi fermé pour l'audience, avec un serveur DHCP et un serveur DNS. Le serveur DNS n'est pas indispensable, il simplifie l'accès aux serveurs. Si le système est décomposé en plusieurs serveurs, nous avons toujours interconnectés ceux-ci en Gigabit Ethernet pour réduire les latences. Bien que nous ayons utilisé un réseau Wifi, il ne serait pas difficile de mettre en œuvre Skini sur un réseau public, en 4G par exemple. Nous ne l'avons pas concrètement fait essentiellement parce que la couverture des salles de spectacle par les réseaux publics n'est absolument pas garantie. Nous avons même pris l'habitude d'installer nous-même notre réseau Wifi pour éviter de dépendre d'une infrastructure dont nous n'aurions pas le contrôle. Nous pensons que l'utilisation de réseaux mobiles sera plutôt adaptée à des performances dans des lieux publics « ouverts » du type hall de gare ou dans la rue.

## 8.7 Conclusion sur l'architecture technique

Skini est un système relativement complexe à mettre en œuvre. Il fait appel d'un point de vue informatique à des univers techniques différents, demandant des expertises techniques sans grandes relations : MAO, informatique distribuée et système d'exploitation. Bien que Skini ne soit lié à aucune DAW particulière, nous avons essentiellement utilisé *Ableton Live*, et nous avons de nombreuses fois fait appel à des fonctions de ce logiciel, notamment pour la gestion par l'orchestration des transpositions et des modes. Néanmoins, ces fonctions ne sont pas exceptionnelles et d'autres outils les fourniraient. Dans notre cas, la DAW héberge des synthétiseurs et échantillonneurs logiciels qui ont chacun des comportements spécifiques. La mise en place de la plateforme de production du son a été une activité fortement chronophage. Nous n'avons pas eu l'occasion d'utiliser Skini avec beaucoup d'interprètes, mais la mise en œuvre technique de ce scénario est assez simple à réaliser techniquement. Elle le sera sans doute moins du côté d'un orchestre.

La partie système d'exploitation n'était a priori pas très simple car Hop.js a été conçu sous Linux et peut-être porté sur *macOS*. Or à ce jour *Ableton Live* comme les principaux logiciels de MAO ne tournent pas sous Linux. Ne souhaitant pas brider Skini en rendant la plateforme dépendante de *macOS*, nous avons réalisé plusieurs essais plus ou moins fructueux de portage de Skini sous Windows10 à travers des machines virtuelles [1] ou docker [4]. En dernière instance le problème a été résolu de façon élégante par *Microsoft* qui a porté *Linux* de façon native sous *Windows10* (*Window Subsystem for Linux* [109]). Il est donc possible de faire fonctionner Skini aussi bien sur PC *Windows* que sur *macOS* avec les mêmes outils de MAO et des performances équivalentes. Pour des raisons de portage similaires, nous avons choisi *Processing* comme passerelle OSC-MIDI. Cet outil fonctionne aussi bien sur *Linux* que *macOS* ou *Windows*.

Du côté des clients nous avons rencontré assez peu de problèmes de système d'exploitation si ce n'est avec les terminaux Apple. Les anciennes générations d'iOS ne connaissent pas certaines commandes JavaScript et les dernières versions d'iOS interdisent l'utilisation des fonctions utilisant les horloges des API Web Audio que nous utilisons pour le protocole de synchronisation du séquenceur distribué. En tout état de cause, la synchronisation du séquenceur distribué est un peu moins bonne sur *iOS* que sur *Android*. Point qui n'est cependant pas bloquant.

Nous allons à présent aborder la phase finale de l'évaluation de la mise en œuvre de HipHop.js comme support de notre méthode de composition. Il s'agit donc de présenter et analyser quelques performances réalisées à partir de la plateforme technique que nous avons décrite.

# 9 LES RETOURS D'EXPERIENCES

LE GOLEM 129

LE 8 DECEMBRE 2017 130
LE 9 DECEMBRE 2017 132
CONCLUSION DU GOLEM 135

LA FABRIQUE A MUSIQUE 136

L'USINE 136
CONCLUSION DE LA FABRIQUE A MUSIQUE 137

D'AUTRES PERFORMANCES 138

CONCLUSION DES RETOURS D'EXPERIENCE 138

Nous avons eu l’opportunité d’utiliser notre plateforme dans des contextes différents durant nos travaux. Nous entrerons dans le détail de deux productions se situant à des étapes différentes de notre réflexion. Une a été réalisée au tout début de notre projet où nous n’avions pas encore clairement finalisé notre méthode de composition. L’autre réalisation importante est plus récente, elle met en œuvre la totalité de notre méthode dans un contexte pédagogique.

Au-delà de ces deux cas d’usage nous avons eu la possibilité de mettre en pratique nos concepts Skini dans d’autres contextes, comme lors d’une performance réalisée pendant la conférence Synchron en 2018 [110] avec une audience d’une trentaine de personnes, lors de la fête de la science 2018, la fête de l’Inria en 2018, ou le séminaire IFIP à Nice en novembre 2019. Pour la fête de la science et la fête de l’INRIA, il ne s’agissait pas de performances à proprement parler, mais plutôt de la mise à disposition de la plateforme dans un espace ouvert où le public venait « jouer » avec la plateforme en petits groupes de moins de 10 personnes, ce qui constitue un cas d’usage encore différent de ceux évoqués ci-dessus.

## 9.1 Le Golem

Le spectacle Golem [37], est une allégorie du mythe bien connu du Golem [78] créé par la communauté juive de Prague au XVIe siècle. Le spectacle a été représenté deux fois dans le cadre du festival de musique contemporaine MANCA à Nice organisé par le CIRM.

La scène pour Golem était composée :

- d’un calligraphe [111], dont la table de travail était projetée sur un grand écran. Deux microphones captaient le son produit par le calame (l'outil d'écriture) sur le papier et le son produit par un résonateur sur lequel le papier était posé (l'Epistophone). Des effets en temps réel transformaient ces sons ;

- d’une chanteuse soprano dont le rôle était de chanter et de dire des textes. La voix de la soprano était captée et traitée en temps réel ;

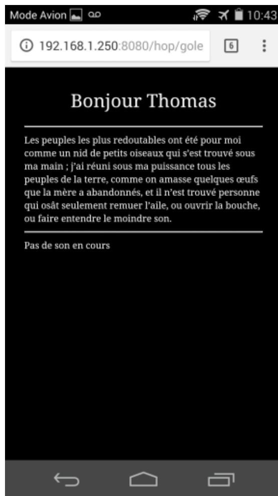
- un « plateau technique » avec un « contrôleur » de l'autre côté de la scène vis-à-vis du calligraphe. Son rôle était de lancer les effets correspondant aux différentes scènes, et de contrôler le déroulement du spectacle et les niveaux sonores pendant l'interaction. L'opposition entre le calligraphe et le contrôleur sur place était un élément de l'allégorie ;

- d’un grand écran permettant de voir les réalisations du calligraphe et de donner des informations à l’audience ;

- d’une effigie du Golem qui a été utilisée pour représenter les différentes phases de sa vie.

Le spectacle comprenait deux sessions d'interactions de 4 et 5 minutes chacune utilisant une orchestration HipHop.js simple. Nous avons présenté le spectacle à deux reprises, avec deux catégories de spectateurs différentes. Le 8 décembre 2017, avec deux classes de collégiens, et le 9 décembre 2017 avec un panel plus classique de 150 spectateurs. Avant le spectacle, quelques instructions ont été données au public sur la façon d’accéder au réseau Wi-Fi privé et comment se connecter au serveur. Les principales caractéristiques de l'interface ont été présentées. Le rôle du public dans l'histoire du Golem a également été expliqué.

### 9.1.1 Le 8 décembre 2017




Nous avions 42 élèves de 12 ans très motivés par l'idée d'utiliser un smartphone pendant un spectacle. En effet, bien que le smartphone soit un appareil important pour les pré-adolescents, son usage est interdit dans les écoles. Cette occasion exceptionnelle d'utiliser des smartphones dans un contexte scolaire a certainement contribué à une participation active à la fois durant le spectacle et aussi durant la séance de questions et réponses qui a suivi. Dès leur entrée en salle, tous les élèves se sont pressés d'accéder au système. Ce à quoi nous nous nous attendions car ils avaient été préparés à cette séance par leurs professeurs de musique.

L'interface du smartphone avait trois rôles : donner des informations sur le déroulement du spectacle comme les textes chantés ou dits ; permettre la présélection, c'est-à-dire l'interrogation du serveur à partir d'une série de critères ; sélectionner des patterns durant les sessions interactives. Pendant ce spectacle, nous avons eu deux sessions d'interactions de 4 et 5 minutes chacune. Cela a généré 4890 événements. Il pouvait s'agir d'une présélection de patterns, ou de la sélection d'un pattern pour qu'il soit diffusé sur le système de sonorisation de la salle de concert. Nous avons eu 2400 événements en session 1 ; 2490 en session 2 dont 689 activations de patterns (228 en session 1 ; 461 en session 2) ce qui fait une moyenne de 1,3 activation de pattern par seconde. Les graphiques suivants donnent l'activité par utilisateur sur les deux sessions. Les collégiens sont associés chacun à un numéro sur l'axe horizontal. Sur le graphisme de l'activité en session 1, nous voyons un participant très actif, qui n'était en fait pas un élève mais l'enseignant.

**Figure 26 : L'acteur pour le Golem**

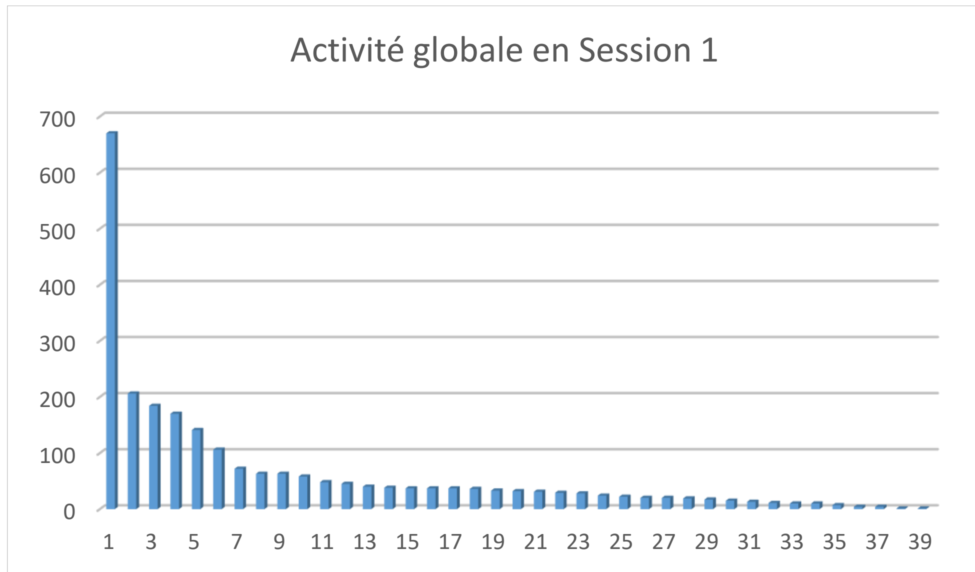



**Figure 27 : Activité session 1**

Les Figure 28 : Activations session 1 et Figure 29 : Activations session 2, donnent la répartition de l'activation de patterns par participant pour chaque session. Il s'agit de l'indicateur le plus

représentatif de l'impact sur le son produit. Nous voyons que la répartition n'est pas plate du tout. Cela signifie que nous avons des comportements très différents au sein de l'audience.

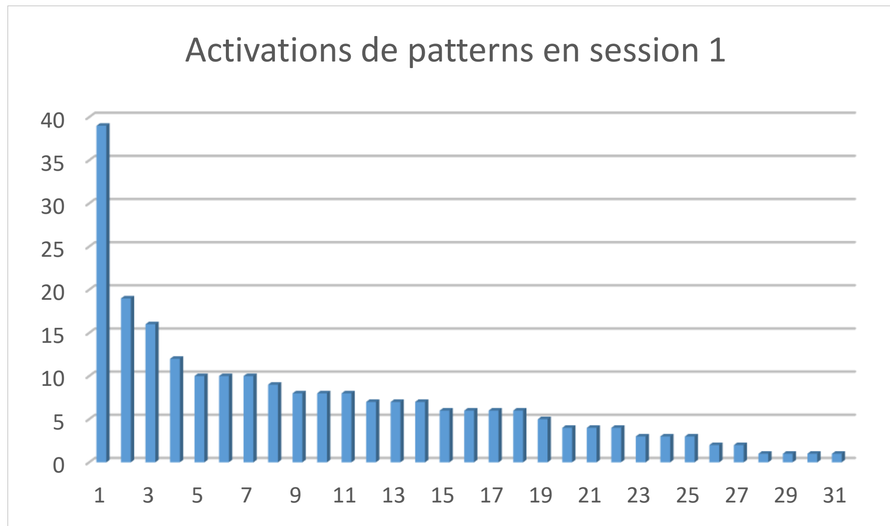



**Figure 28 : Activations session 1**

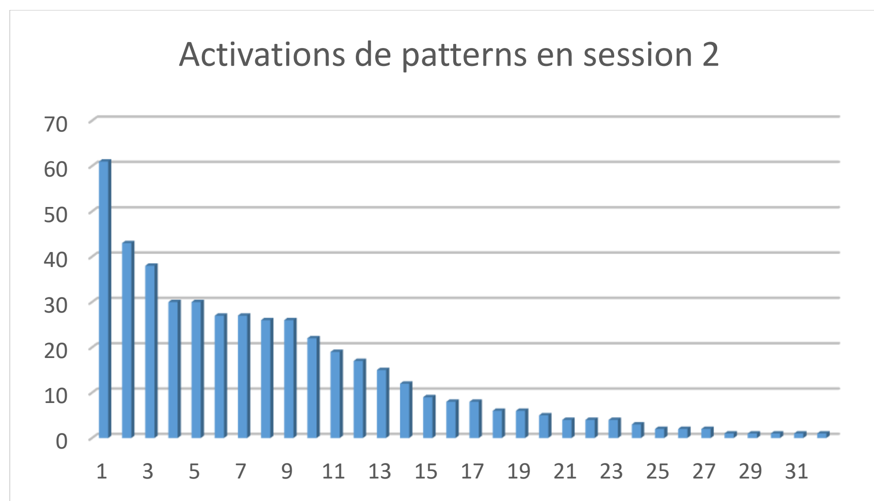



**Figure 29 : Activations session 2**

Cette répartition irrégulière de l'activité est un point important que nous avons pris en compte dans les développements des interfaces que nous avons rencontrées précédemment. Ceci nous a incité à proposer plusieurs interfaces selon une typologie de participants car nous voyons ici de façon évidente que le public n'est pas une entité uniforme.

Dans cette première version de Skini, nous avions mis en place une protection simple contre les répétitions systématiques de patterns qui interdisait plus de 3 répétitions immédiates d'un même pattern. L'interface acteur pour ce spectacle était assez différente de celle présentée dans ce document (voir la Figure 26 : L'acteur pour le Golem). Il était alors assez facile de demander de façon répétée l'activation d'un même pattern. C'est un point que nous avons corrigé par la suite, néanmoins ces répétitions sont intéressantes car elles peuvent avoir plusieurs origines : une mauvaise

compréhension de l'interface, un manque de retour clair à l'action demandée ou même une tentative de pirater le système.

Dans le schéma suivant (Figure 30 : Détection des répétitions) qui donne le nombre de fois que nous avons détecté une répétition pour un même utilisateur en session 2, nous voyons des comportements étranges, avec neuf spectateurs produisant beaucoup de répétitions et surtout un en particulier détecté avec 855 répétitions. Bien qu'il ait été très actif, cet individu n'a pas activé beaucoup de patterns (38). Nous supposons qu'il n'avait pas compris le mécanisme des files d'attente exposé en début du spectacle. Le second individu a le même profil avec 456 répétitions et une activité modérée en termes d'activation de patterns.

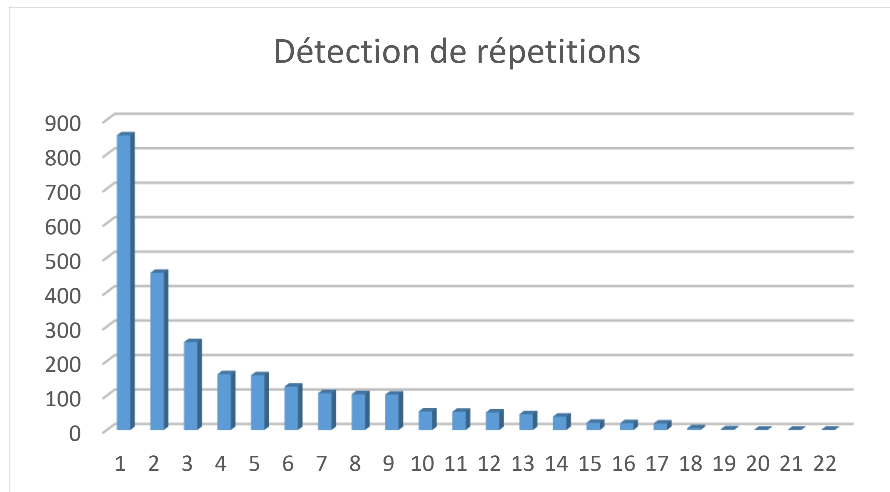



**Figure 30 : Détection des répétitions**

### 9.1.2 Le 9 décembre 2017

Nous avions 150 personnes présentes pour la performance. Il s'agissait du public traditionnel du festival MANCA. Les spectateurs étaient pour la plupart des adultes habitués à assister à des concerts de musique contemporaine. Au cours du spectacle, 90 spectateurs ont effectivement participé à la performance. Cela signifie que 33 % d'entre eux n'ont pas essayé ou n'ont pas réussi à se connecter au serveur pour plusieurs raisons, comme l'incompatibilité des terminaux ou une mauvaise compréhension du processus. (Les anciens iOS ne supportaient pas les dernières évolutions JavaScript utilisées par Hop par exemple). Comme pour les collégiens, le public a été informé avant le spectacle que des problèmes pouvaient survenir avec certains appareils.

Pendant ce spectacle, nous avons eu deux sessions d'interactions de 3 et 6 minutes chacune. Cela a généré 4099 événements (2171 en session 1 ; 1928 en session 2), dont 754 événements sonores générés par le public, soit une moyenne de 1,4 événement sonore par seconde.

L'image suivante (Figure 31 : Activité globale grand public session 1) donne le nombre d'événements générés par chaque utilisateur. Pour la première session, nous voyons qu'il y a 3 utilisateurs générant plus de 60 événements. Pour la session 2, (Figure 32 : Activité globale grand public session 2), nous avons encore 3 utilisateurs qui génèrent cette fois plus de 100 évènements et 69 utilisateurs pour plus de 20 événements.

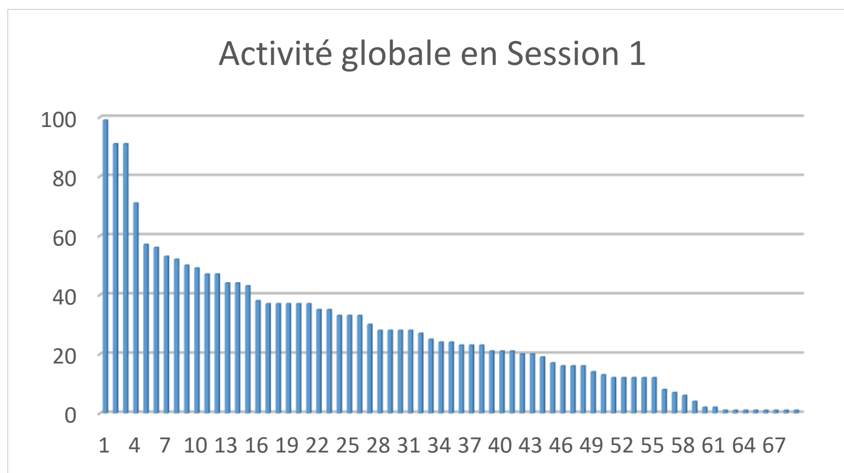



**Figure 31 : Activité globale grand public session 1**

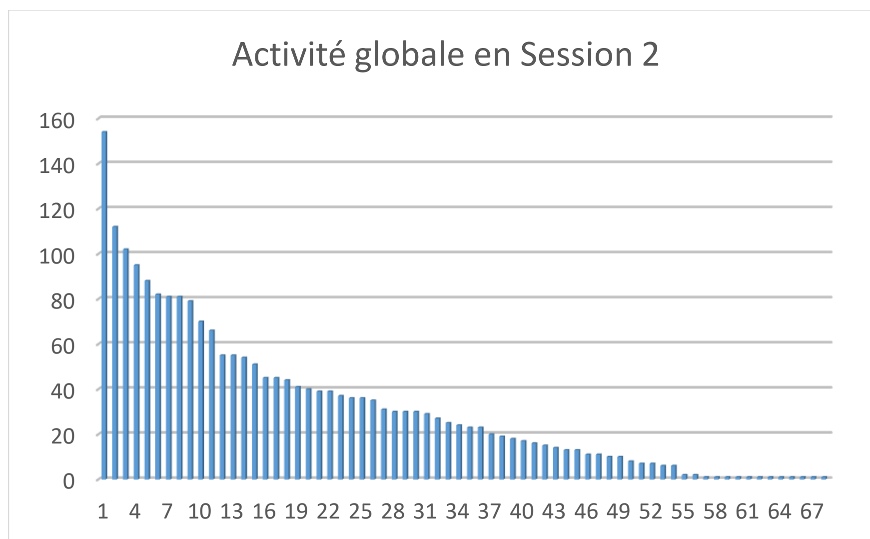



**Figure 32 : Activité globale grand public session 2**

Nous avons eu 754 activations de patterns (345 en session 1 ; 409 en session 2) pendant le spectacle. Les graphiques, Figure 33 : Activation grand public session 1 et Figure 34 : Activations grand public session 2, donnent la répartition par utilisateur. Comme nous l'avons dit, il s'agit de l'activité la plus représentative de l'impact sur le son produit. Nous voyons que globalement 19 utilisateurs ont contribué plus de 10 fois à l'activation de patterns.

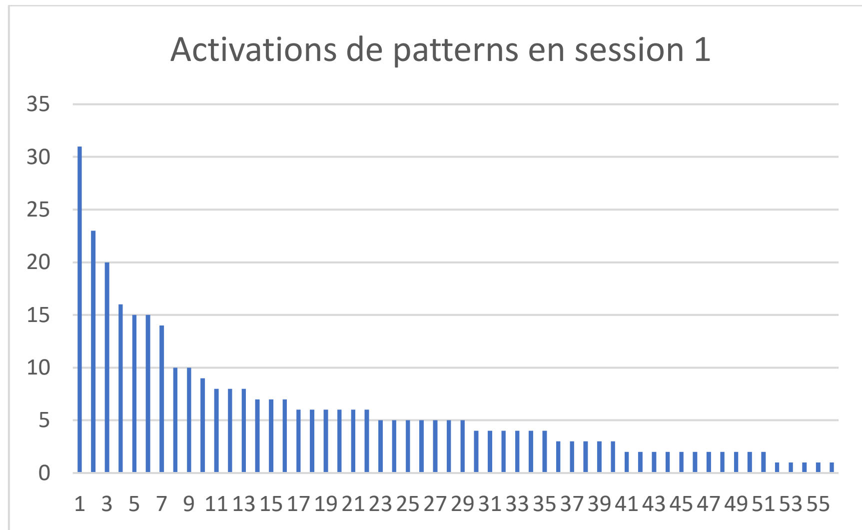



**Figure 33 : Activation grand public session 1**

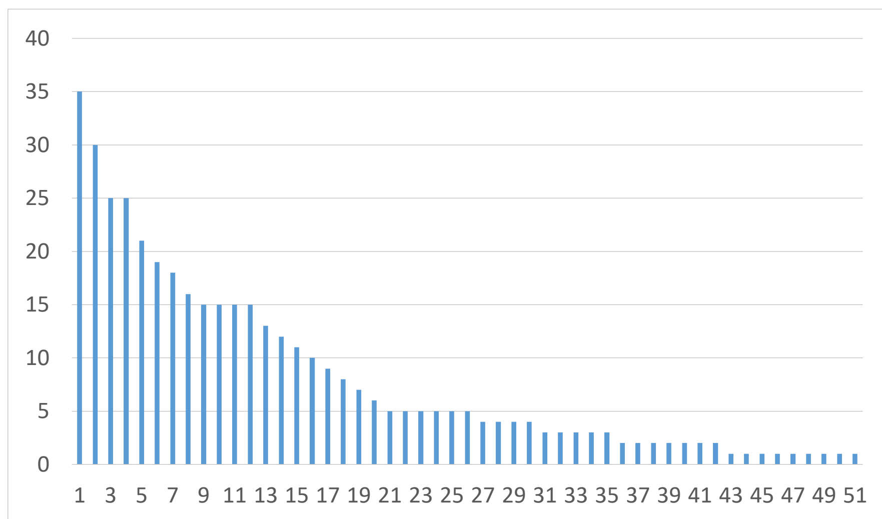



**Figure 34 : Activations grand public session 2**

Comme ce fut le cas avec les élèves, nous avons eu quelques comportements étranges comme nous le voyons sur le graphique Figure 35 : Répétitions grand public, avec des détections de répétitions durant la deuxième session. Le premier participant est globalement l'utilisateur le plus actif en termes de répétitions avec 1046 détections. Il a généré 54 événements et 38 activations de patterns durant cette deuxième session. Comme précédemment il peut s'agir de quelqu'un qui, soit n'a pas compris le processus, soit qui a des problèmes avec son interface, soit qui veut « pirater » le système.

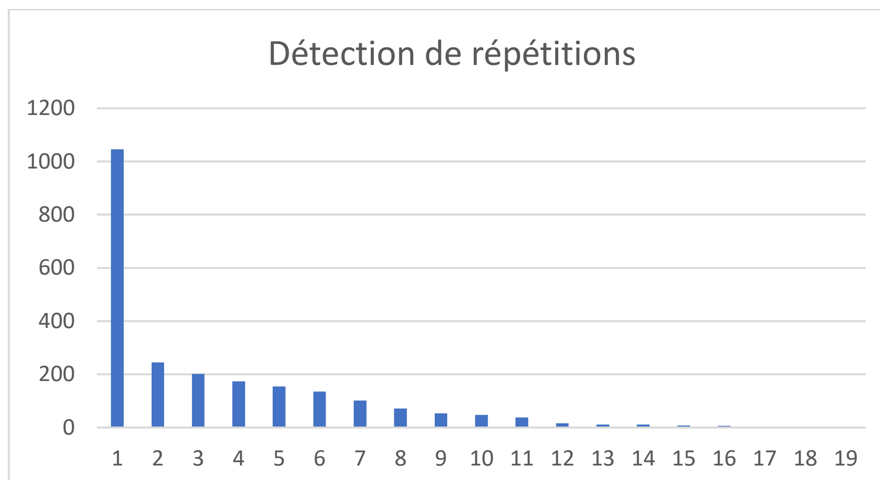



**Figure 35 : Répétitions grand public**

Le deuxième participant est également intéressant en tant que spectateur particulièrement actif. Il a été détecté plus de 200 fois lors de la deuxième session. Son activité est néanmoins tout à fait raisonnable avec 64 événements et 17 activations de clips. Globalement, sur 64 utilisateurs ayant été détectés, 12 d'entre eux ont été détectés comme ayant cliqué plus de 11 fois de façon répétitive.

### 9.1.3 Conclusion du Golem

Le premier point à noter est que le taux de participation était élevé et que le public (soit les jeunes élèves le 8 décembre ou les adultes le 9 décembre) a participé avec enthousiasme au spectacle. Ceci est attesté par le haut niveau d'activité au cours des différentes sessions interactives. Même si nos travaux en étaient encore à leurs balbutiements, nous avons constaté que les deux publics ont été motivés par l'événement comme en témoigne la session de questions-réponses après les deux spectacles. Nous avons reçu beaucoup de commentaires pour améliorer l'interface et le spectacle lui-même.

En ce qui concerne les différents publics, il n'est pas surprenant de constater que les élèves étaient en proportion beaucoup plus actifs que le public standard avec un taux de connexion bien meilleur. Plus surprenant, nous constatons que la répartition des activités n'est pas si différente entre les deux populations. Dans les deux types d'audience, le niveau global de participation montre le même type de différences dans le comportement des spectateurs.

En termes de réseau et de connexions, plusieurs facteurs sont entrés en ligne de compte. Nous pouvons constater que la population jeune n'a eu aucune difficulté à se connecter au serveur et que seulement 66 % d'un public plus classique a franchi cette première étape. Pour donner suite à ce spectacle nous avons envisagé et réalisé les améliorations suivantes :

L''interface des acteurs devait être repensée. Les algorithmes utilisés pour la sélection des patterns et la gestion des files d'attente étaient trop opaques pour le public, notamment parce que la sélection semblait non déterministe. Cela a induit plusieurs comportements différents, comme la répétition systématique des commandes, et bien souvent la sélection et le lancement de patterns sans les écouter. Nous avons traité ce problème en :

- présentant une vue globale graphique des files d'attente ;
- proposant une navigation plus simple dans les patterns disponibles ;

- améliorant la gestion des files d'attente et le suivi du déroulement en affichant l'orchestration.

Les « logs » des spectacles ont révélé un point très important : comme il existe différents comportements, il nous a semblé judicieux de proposer différentes interfaces en fonction du profil du participant que nous avons vues dans le chapitre sur l'interaction. Cette première évaluation nous a permis de valider que l'environnement Hops.js/HipHop.js était suffisamment stable, que l'interaction nécessaire à une expression réelle de la durée devait être améliorée pour aller plus loin dans la validation d'une méthode de composition faisant appel à l'expressivité de HipHop.js.

Nous avons peu parlé de l'orchestration des séquences interactives dans la présentation de cette performance car certains concepts comme la structuration en sessions, ou la mise en place de synchronisation MIDI n'avaient pas encore été définis. L'orchestration était donc réduite à des comportements simples d'activation de groupes de patterns sans contraintes. Le projet présenté ci-dessous fait appel à une configuration mettant en œuvre l'ensemble des concepts de notre méthode.

## 9.2 La fabrique à musique

La « Fabrique à Musique » est un appel à projet de la SACEM (Société des Auteurs Compositeurs et Editeurs de Musique [112]) destiné aux organisations culturelles. Cet appel à projet a pour but de promouvoir la collaboration entre des artistes et des écoles secondaires. L'objectif est de sensibiliser les enfants à la création musicale, l'écriture et la composition. Grâce au Conservatoire de Nice et au CIRM, Skini et un compositeur de musique ont été sélectionnés. Notre proposition avec Skini a été de créer un travail musical complet en 8 sessions de 2 heures avec une classe de 6ème, soit 25 jeunes élèves (12 ans). Le projet s'est déroulé entre décembre 2018 et mai 2019. Le résultat a été présenté officiellement lors d'un concert au Conservatoire de Musique de Nice le 24 mai 2019.

### 9.2.1 L'usine

Le projet consistait à aider les élèves à créer une histoire et à l'illustrer d'une composition musicale de leur choix. Les élèves ont conçu l'illustration musicale avec Skini par groupes de 5 ou 6. Ce projet était plus qu'une simple performance, il a été pensé avec le professeur de musique comme une approche pédagogique complète. Dans ce contexte, la plateforme Skini n'a pas été utilisée pour interpréter une pièce de musique créée par un seul compositeur, mais comme un outil de création collaborative.

Tout d'abord, les élèves ont dû découvrir les concepts de base de Skini en jouant avec l'interface acteur à l'aide d'une pièce interactive simple. Puis avec leur professeur de littérature, ils ont imaginé une histoire sur le thème de la pollution qu'ils ont baptisée « L'Usine ». La conception des patterns a été réalisée en petits groupes à l'aide du séquenceur distribué. Lors de la phase de conception, chaque élève devait choisir un synthétiseur différent dans un panel proposé par le compositeur. Chaque élève pouvait ainsi exécuter son pattern en même temps que ses camarades ou demander que l'on écoute son travail sans entendre les autres patterns. Ceci était un moyen d'échanger sur les essais et d'alterner l'écoute collective d'un pattern en particulier avec la superposition des patterns en cours de conception. Quand un pattern donnait satisfaction à son créateur, il était enregistré. Les 5 séances de création, où se succédaient 3 ou 4 groupes de 5 à 6 élèves, étaient chacune axées sur une étape de la narration. L'illustration sonore était donc conduite par un thème, une atmosphère qu'ils avaient eux-mêmes proposée.

L'orchestration, étant un concept un peu difficile à comprendre pour des enfants, a été principalement proposée par le compositeur et validée par la classe. Enfin, pendant le spectacle, les élèves ont donné vie à leur travail musical en temps réel. Figure 36 : L'orchestration de l'Usine, correspond à l'affichage de la pièce durant la performance. Chaque ligne correspond à une étape de la narration. Cet affichage permettait aux élèves de savoir quand ils devaient intervenir. En effet nous avions convenu que chaque étape serait activée par un petit groupe, sauf pour le final où tous devaient participer.

**Figure 36 : L'orchestration de l'Usine**

Techniquement, le travail de conception a été réalisé au moyen de tablettes sur lesquels les élèves ont utilisé le séquenceur distribué.

### 9.2.2 Conclusion de la fabrique à musique

Il s'agissait de la première utilisation de Skini dans un contexte éducatif. Ce projet a été bien soutenu par le professeur de musique du collège, dont la contribution au bon déroulement des différentes séances de création a été importante. Cet accueil positif est probablement dû à l'adéquation entre l'outil et la méthodologie pédagogique utilisée dans cette école qui accueille une population immigrante considérable qui peut avoir à apprendre une langue étrangère et à s'adapter à un nouvel environnement. Les activités créatives sont mises en valeur et les enfants y sont très réceptifs. Ils se sont bien investis dans le projet et n'ont eu aucun problème pour manipuler l'outil. L'aspect ludique de l'outil a facilité le processus créatif ; cependant, il aurait fallu plus de temps pour permettre aux enfants de bien comprendre les notions de durée de note, de gammes, etc.

La mise en œuvre de l'orchestration a été définie en collaboration avec les élèves. A partir du scénario proposé nous avons pu constater que la programmation en HipHop.js a été effectivement rapide et que nous n'avions utilisé qu'une partie des possibilités offertes par HipHop.js. En effet, les événements pris en compte pour le déroulement étaient des combinaisons de sélections de patterns et de chronomètres. Ce projet nous a permis de mesurer l'importance de l'affichage global de l'orchestration, et de donner une vision structurée du déroulement. La « gestion de la motivation » de l'audience était ici en revanche fortement simplifiée car l'interaction ne concernait que les 25 concepteurs de la pièce. Le reste de l'audience était passif. Nous étions donc dans un cas d'usage mélangeant spectacle et performance collaborative.

Un des principaux constats issus de ce projet est que le modèle proposé par Skini fonctionne dans des situations très diverses, depuis l’interaction avec une large audience comme pour le Golem, à un travail de création en petits groupes avec le séquenceur distribué.

## 9.3 D’autres performances

Les autres performances évoquées en début de chapitre ont permis la mise en œuvre des pièces Grand Loup, Opus1 et Opus2 présentées dans les exemples de programmation de performances. Ces performances ont été réalisées avec des audiences de 20 à 30 personnes. Nous avons pu constater que les réalisations concrètes de ces pièces étaient assez conformes à ce que nous avions vu lors des simulations. Parmi les différences notables avec une simulation, nous avons noté des variations sensibles de l’activité de l’audience dans le temps. Ces performances ont eu besoin d’une période de démarrage où l’audience a découvert les interfaces. À la suite de cette phase initiale, l’activité de sélection prend un rythme qui fluctue, mais qui est bien absorbé par le mécanisme des files d’attente. Ces performances nous ont permis de prendre conscience que la fluctuation de la participation est un paramètre à prendre en compte dans la phase de conception de la performance. Le simulateur peut être paramétré de façon à prendre en compte différentes tailles d’audience. Il serait donc possible de le faire évoluer pour modifier ces paramètres dynamiquement pendant une session de simulation, de façon à anticiper certaines fluctuations possibles du déroulement d’une performance.

L’Opus1 est un exemple de pièce dont l’orchestration a fortement évolué en fonction des retours durant les performances évoquées ici. Nous avons vu deux phases de cette évolution dans le chapitre 6, Réalisation d’orchestrations par automate. La première phase décrite dans le paragraphe « Opus1 », p.82, est différente de celle décrite dans le paragraphe « Une autre orchestration de l’Opus1 », p.88. L’élément majeur faisant cette différence est le scénario proposé à l’audience. La deuxième version introduit la possibilité de choisir des chemins dans le déroulement. Le point majeur à retenir de cette évolution est que le mécanisme de vote introduit un ralentissement des sélections de patterns. Ceci n’a pas été perçu par l’audience grâce aux files d’attente qui ont pour effet de lisser les fluctuations. Cette évolution de l’Opus1 est intéressante à deux niveaux techniques. Le premier est qu’elle n’aurait pas été facile avec un langage généraliste. Le deuxième est que l’intégration de HipHop.sj dans JavaScript via Hop.js a permis de faire évoluer rapidement non seulement l’orchestration mais aussi le mode d’affichage.

## 9.4 Conclusion des retours d’expérience

Nous avons eu la chance de pouvoir réaliser quelques performances pour valider les principes de notre méthode composition et ainsi mettre à l’épreuve la programmation réactive synchrone sur le terrain. Nous ne pensons cependant pas avoir terminé cette validation. En effet, il nous faudra plus de performances, déclinées avec des audiences différentes et selon d’autres styles de musique. Il nous faudra aussi imaginer de nouveaux scénarios mettant en jeu des audiences dans des contextes qui ne soient pas des concerts. Il est fort probable que nos interfaces adaptées à un contexte expérimental, ne soient pas adaptées à une soirée Electro ou une performance de Jazz. Il reste aussi des travaux intéressants à réaliser autour de la scénarisation de la performance ou l’utilisation de l’interface « scrutateur ». Les recherches en *ludologie* [75], la science du jeu, seraient certainement des apports importants pour la mise en œuvre de scénarios adaptés à des audiences variées.

Du côté de l’interaction, la façon de penser les interfaces et leurs évolutions ont été revues au fur et à mesure des performances. La première expérience réalisée avec le spectacle Golem s’est bien

déroulée grâce à plusieurs paramètres : un effet « surprise », la bonne intégration dans une narration et la brièveté des sessions d'interaction qui a pallié le manque de vision long terme de l'interaction. A partir de ce spectacle nous avons réalisé des modifications profondes dans notre méthode de composition comme l'introduction d'automates plus complexes, mais aussi refondu l'interface « acteur », mis en place une meilleure visibilité des files d'attentes et le contrôle des « abus ». Les performances en plus petits comités nous ont conduits à travailler la vision long-terme de l'orchestration pour aboutir dans les derniers spectacles à l'utilisation de systèmes de vote en même temps que les sélections de patterns.

Si nous avons rendu opérationnels la plupart de nos concepts initiaux, il est évident qu'il reste un travail important à réaliser sur l'esthétique des interfaces. Il reste aussi un chantier sur la conception d'autres types d'interface « acteur ». La version actuelle a le mérite d'être proche des concepts de base, mais c'est aussi un défaut car elle suppose que l'audience ait compris l'architecture logique de la plateforme. La pratique montre que cette compréhension n'est pas immédiate, même pour des informaticiens.

Les travaux dans le contexte pédagogique ont été la source de modifications régulières du séquenceur distribué que nous ne considérons pas dans un état suffisamment finalisé. Il lui manque la capacité de faire entendre localement les patterns lors de leur conception. Dans sa version actuelle il n'est utilisable que pour des petits groupes bien conscients d'un travail collectif, ce qui limite considérablement son usage.

# 10 CONCLUSION ET PERSPECTIVES

Nous sommes partis d'une problématique liée à l'expression du temps, qui est une question fondamentale pour la musique et qui est à l'origine de la programmation réactive synchrone. A partir des concepts de temps objectif et de durée de Bergson, nous avons conclu que nous avions besoin d'une solution de mise en œuvre de la durée, en complément d'un moyen de l'exprimer. La solution que nous avons retenue pour mettre en œuvre la durée est la musique interactive. Pour évaluer concrètement l'expressivité de HipHop.js, il nous a fallu développer une plateforme de musique interactive, Skini, et adresser un certain nombre de questions relatives aux interfaces homme-machine.

Nous avons vu à travers la mise en œuvre de la plateforme Skini, que même si l'expressivité du langage HipHop.js, n'est en elle-même pas suffisante pour exprimer de la musique, elle permet de définir une méthode originale de composition à base de patterns et d'orchestrations de ces patterns. Cette méthode permet de créer des pièces de musique proches des œuvres ouvertes au sens d'U. Eco. Nous avons abouti à un système expérimental mais opérationnel, puisqu'il a permis plusieurs réalisations concrètes validant nos hypothèses.

Notre méthode de composition peut être utilisée de plusieurs façons. La plus simple est de traduire en HipHop.js une description graphique de l'orchestration. Il s'agit d'une approche naturelle, car elle est dans la continuité directe de la façon dont la musique est écrite depuis des siècles. Une manière plus originale et aussi plus difficile car elle nécessite de bien saisir les concepts de base de la programmation synchrone, consiste à penser l'orchestration directement en HipHop.js avec toute la richesse combinatoire que peuvent offrir des automates complexes.

Voici quelques pistes de développement et de recherche que nous avons abordées pour la plupart dans deux publications, une dans le domaine musical pour la conférence *New Interfaces for Musical Expression* (NIME 2019) [67] et une à paraitre dans le domaine de la programmation informatique pour le journal *The Art, Science, and Engineering of Programming* [66]. Certaines relèvent d'améliorations ou d'extensions de fonctions existantes pour « le séquenceur distribué », d'autres sont des axes de recherche à part entière comme l'utilisation de Skini pour la « musique générative » et pour « le contrôle de la scène », ou bien comme la mise en œuvre d'un « Computer Systems for Expressive Music Performance ».

**Un DSL Musical**

Nous avons vu que HipHop.js, accompagné des concepts de pattern et de groupe de patterns, donnait des résultats à la fois en musique interactive et en musique générative. L'expressivité de ce langage informatique est donc suffisante pour notre objectif, mais son utilisation par un musicien non informaticien est complexe. Nous avons vu aussi que la traduction « manuelle » d'une représentation graphique d'une orchestration en HipHop.js était un processus assez simple. Il serait donc assez naturel d'envisager une traduction automatique des représentations graphiques d'orchestrations en HipHop.js. Celle-ci n'utiliserait pas toutes l'expressivité de HipHop.js, mais faciliterait l'accès à notre méthode de composition.

**Musique générative**

Nous avons abordé cette question dans le chapitre dédié à la simulation. Nous avons vu que la simulation d'une orchestration pouvait produire simplement une pièce musicale faisant appel à des mécanismes aléatoires pour les activations de patterns. Le processus de sélection aléatoire repose sur une fonction de randomisation propre à JavaScript. Il est possible d'investiguer davantage autour des processus aléatoires, en utilisant des chaines de Markov par exemple. La réflexion autour

de l'association de processus stochastiques et d'automates d'orchestration est une piste de recherche intéressante qui pourrait trouver des cas d'usage en dehors de la scène pour des environnements sonores de jeux, de salles d'exposition ou autres lieux cherchant à dégager une couleur musicale originale.

**Contrôle de la scène**

Tout événement pouvant se décrire sous forme de patterns au sens de Skini peut faire l'objet d'une orchestration Skini. Nous avons largement traité le cas de la musique, mais le principe du pattern s'applique très bien à la lumière par exemple. Nous avons pu constater que les logiciels de contrôle DMX (Digital Multiplex [59]) étaient assez proches des séquenceurs MIDI dans leur logique. Nous avons fait des essais avec le logiciel QLC+ [113] par exemple, qui utilise des notions similaires à des patterns (des scènes pour QLC+). Parmi les organes de contrôle de la scène nous pouvons imaginer de piloter sous forme de patterns des robots, des séquences vidéo ou des algorithmes de synthèse d'image. La réalisation de travaux dans ce sens relève d'une réflexion préliminaire sur ce que signifie l'utilisation de patterns dans ces contextes mais leurs mises en œuvre à partir de notre méthode de composition ne devraient pas présenter de grandes difficultés.

**Séquenceur distribué**

Nous avons utilisé le séquenceur distribué dans deux contextes, comme outil de conception pour les scolaires et comme application « ludique » pour un groupe de 5 ou 6 personnes. Nous n'avons pas encore intégré le séquenceur en live dans une orchestration. Ceci pose en effet des problèmes de fond, comme par exemple la synergie à créer entre orchestration et patterns. Doit-on partir d'une orchestration prédéfinie, et laisser la libre conception des patterns ? Doit-on laisser l'orchestration ouverte ? Il s'agit d'un chantier de conception de musique interactive à part entière, qui offre des possibilités de scénario à inventer. La seule limite facilement concevable est que ce type de système ne s'adressera pas à une large audience de novices. En revanche on peut imaginer des usages pédagogiques autour de travaux collaboratifs et pas uniquement pour des scolaires. Techniquement l'intégration du séquenceur distribué dans une orchestration pose des questions intéressantes d'ergonomie sur la façon de stocker et d'indexer les patterns en cours de création.

**Evolution de l'interaction**

Nous avons décrit dans ce document un ensemble d'interfaces clients. Nous ne pensons pas avoir fait le tour de la question. La réflexion autour de la bonne façon de gérer l'interaction entre acteurs et orchestration est un domaine de recherche à part entière. Nous sommes convaincus qu'un travail est à faire autour de la relation entre Skini et les jeux. La « ludologie » [75], la science des jeux, est une discipline qui devrait apporter nombre d'idées intéressantes sur la façon de concevoir nos interfaces. Notons aussi que nous avons abordé une interface un peu particulière, le scrutateur, dont nous n'avons pas encore fait un usage intensif. Nous pensons qu'il y a des pistes intéressantes à creuser autour de cette interface, notamment en conjonction avec un moteur aléatoire. Ceci devrait permettre d'imaginer des scénarios de musiques collaboratives en petits groupes, très différents de ce que nous pouvons imaginer avec le séquenceur distribué.

**Plateforme pédagogique**

Nous avons utilisé Skini dans un environnement pédagogique. La mise en œuvre de la plateforme a été réalisée par nos soins. Nous ne pensons pas qu'il soit encore très simple de mettre en œuvre Skini pour un professeur de musique sans compétence informatique. La promotion de Skini en pédagogie demandera une simplification de l'architecture du système avec une utilisation privilégiée d'outils *open source* pour la synthèse du son notamment.

**Evolution du système**

Il reste un certain nombre de points relatifs au système à améliorer ou à mettre en œuvre comme une simplification de l'architecture en remplaçant *Processing* par Hop.js. Bien qu'il ne s'agisse pas d'une question fondamentale pour le fonctionnement de Skini, en termes de maintenance du système, il serait cohérent de limiter les outils de développement et par conséquent de remplacer *Processing* par Hop.js. En effet, Hop.js est capable de dépasser les contraintes de temps de réponse dû à la boucle d'événements JavaScript par l'utilisation de *threads*. Il serait tout à fait possible à Hop.js de gérer avec des temps de réponse suffisants un bon séquencement des commandes MIDI issues des files d'attente. Dans un autre registre, nous avons évoqué la possibilité d'utiliser Skini sur un réseau opérateur. Ceci reviendrait à mettre la solution sur le *cloud*. Cette opération nécessite de résoudre tous les problèmes de sécurité inhérents à ce type de solution. Le séquenceur distribué devrait être complétement revu dans ce sens.

**Computer Systems for Expressive Music Performance**

Dans le cas d'un usage de notre plateforme pour le contrôle de synthétiseurs ou d'échantillonneurs commandés par MIDI, il est possible d'améliorer le rendu du système en introduisant un traitement sur les articulations et les vélocités des commandes MIDI. Ceci permettrait d'obtenir des phrasés plus réalistes ou plus intéressants à partir des résultats finaux issus de la concaténation des patterns. Il s'agirait donc d'ajouter une sorte de post-processeur MIDI entre l'exécution des patterns et la synthèse du son. Ce type de traitement est le sujet de nombreuses recherches, référencées sous les termes *Computer Systems for Expressive Music Performance* (CSEMP), qui se sont développées en parallèles des travaux sur les musiques génératives [60][12]. Il s'agit d'un domaine d'application assez naturel des techniques d'apprentissage profond [46]. La mise en place d'un processeur CSEMP dans Skini pourrait aussi bénéficier d'informations issues de l'orchestration et rendre la solution encore plus crédible dans des contextes de musiques génératives appliquées aux jeux vidéo ou à l'audiovisuel par exemple.

# 11 GLOSSAIRE

**Architecture Multitier** : Il s'agit d'un terme informatique synonyme d'architecture logicielle « client-serveur ». C'est-à-dire d'architecture où le traitement des données, la présentation des données et la gestion de l'application sont clairement séparés. Les applications utilisées sur le Web sont construites sur ce type d'architecture.

**Bug, débugger** : La notion de « bug » remonte aux premiers ordinateurs dont le fonctionnement pouvait être perturbé par la présence d'insectes attirés par la chaleur des composants électroniques. Ces insectes pouvaient produire des courts-circuits et bloquer les ordinateurs. L'action de retirer ces insectes est restée dans le vocabulaire informatique pour signifier la recherche d'erreurs de programmation. Les outils destinés à la recherche des erreurs de programmation s'appellent des « debuggers ».

**Composition Assistée par Ordinateur, CAO :** Il s'agit des méthodes de composition utilisant des algorithmes sous forme de programmes informatiques. La CAO ne produit pas toujours une musique prête à jouer par des synthétiseurs ou des musiciens. Elle peut servir à créer un matériau de base utilisé par le compositeur. Un des exemples est la façon dont I. Xenakis créait certains matériaux de base à partir de processus stochastiques produits par des programmes informatiques pour écrire manuellement ses partitions.

**Editeur de partitions :** Ce sont des logiciels destinés initialement à l'écriture et l'impression de partitions. Les plus connus sont Finale et Sibelius. Ils se sont enrichis pour devenir de véritables DAW.

**DAW, Digital Audio Workstation :** Une DAW, en français « station audionumérique » est un système informatique destiné à la création et à la manipulation de la musique sous format numérique. De systèmes initialement coûteux et complexes, les DAW se sont popularisées avec le développement des micro-ordinateurs. Elles peuvent prendre des formes diverses, allant de stations autonomes dédiées à des logiciels pour micro-ordinateurs.

**Génie logiciel** : Ce terme regroupe les méthodes de travail et les bonnes pratiques qu'utilisent les informaticiens qui développent des programmes. Ceci inclut notamment les méthodes de spécifications, les méthodes de programmation qui dépendent des langages informatiques, les outils de test et de maintenance des logiciels.

**Lick :** Pour les musiques actuelles (blue, jazz, rock…), un « lick » est une courte phrase musicale utilisée pour construire une ligne mélodique. Les « licks » sont souvent l'objet de catalogues utilisés pour l'apprentissage de l'improvisation.

**Modes musicaux, musique modale :** La musique modale consiste à construire le discours musical à l'aide d'échelles de notes clairement définies. En musique occidentale une terminologie s'est imposée au cours du temps, plus ou moins inspirée de la Grèce antique. On parle fréquemment de modes ionien, dorien, phrygien, lydien, mixolydien, éolien et locrien. Ils sont construits sur la base d'une gamme diatonique qui est lue à partir d'une note de départ dans cette gamme. Le mode dorien sur une gamme diatonique en do « do, ré, mi, fa, sol, la, si, do » sera « ré, mi, fa, sol, la si, do, ré » par exemple.

**MIDI, Musical Instrument Digital Interface** : Il s'agit d'un protocole créé dans les années 80. Il est destiné à standardiser les communications entre des instruments électroniques (synthétiseurs, échantillonneurs, boites à rythmes …), des contrôleurs, des séquenceurs et des ordinateurs. Il comprend la description de la connectique, la description des messages échangés par les équipements et la description du format de fichiers permettant de stocker et rejouer des pièces de musique.

**Musique Assistée par Ordinateur, MAO** : Ce terme regroupe l'ensemble des techniques relatives à a production musicale à l'aide d'ordinateur. A la différence de la CAO, la MAO concerne le plus souvent la production du résultat sonore à l'aide d'ordinateurs.

**OSC, Open Sound Control** : OSC est la description d'un format de données conçu pour la transmission de données en temps réel dans le contexte de performances musicales. Ce format a été défini par le CNMAT de l'Université de Berkeley en Californie dans le but d'apporter une alternative plus puissante au standard MIDI. OSC repose sur les protocoles UDP ou TCP très largement utilisés par Internet.

# REFERENCES BIBLIOGRAPHIQUES

[1] Aini Abdul Kadir, Xun Xu, et Enrico Hämmerle. 2011. Virtual machine tools and virtual machining-A technological review. *Robotics and Computer-Integrated Manufacturing*. DOI:https://doi.org/10.1016/j.rcim.2010.10.003

[2] Ableton. 2018. Ableton Live 10 - Ableton. *Ableton*. Consulté 10 mai 2020 à l'adresse https://www.ableton.com/fr/live/

[3] Patricia Alessandrini. 2006. A computer-assisted analysis of rhythmic periodicity applied to two metric versions of Luciano Berio's Sequenza VII. In *Proceedings of Sound and Music Computing Conference, SMC 2006*.

[4] Charles Anderson. 2015. Docker. *IEEE Software*. DOI:https://doi.org/10.1109/MS.2015.62

[5] Moreno Andreatta. 2009. *Théories de la composition musicale au xxe siècle*. Editions Symétrie.

[6] Richard Andrews. 2008. Studio report: UC Berkeley Center for New Music and Audio Technologies (CNMAT). In *International Computer Music Conference, ICMC 2008*.

[7] J Scott Bell. 2005. FL Studio. *Computer (Long. Beach. Calif).* (2005).

[8] G Berry. 1997. The {Esterel} v5 Language Primer. (1997).

[9] G Berry. 2008. Esterel v7 for the Hardware Designer draft 1.1. (2008).

[10] Gérard Berry. 2000. The Foundations of Esterel. In *Proof, Language, and Interaction*, G. Plotkin, C. Stirling et and M. Tofte (éd.). MIT Press. DOI:https://doi.org/10.7551/mitpress/5641.003.0021

[11] Frédéric Boussinot. 1991. Reactive C: An extension of C to program reactive systems. *Softw. Pract. Exp.* (1991). DOI:https://doi.org/10.1002/spe.4380210406

[12] Roberto Bresin et Anders Friberg. 2012. Evaluation of Computer Systems for Expressive Music Performance. In *Guide to Computing for Expressive Music Performance*. 181-203. DOI:https://doi.org/10.1007/978-1-4471-4123-5_7

[13] Jean Pierre Briot et François Pachet. 2018. Deep learning for music generation: challenges and directions. *Neural Computing and Applications*. DOI:https://doi.org/10.1007/s00521-018-3813-6

[14] Clive Brown. 2003. {Berlioz's Orchestration Treatise}. *Journal of Musicological Research*.

[15] Alexandre Resende Clément, Filipe Ribeiro, Rui Rodrigues, et Rui Penha. 2016. Bridging the gap between performers and the audience using networked smartphones : the a . bel system. *Proc. ICLI 16* Levin 2001 (2016).

[16] Jean Louis Colaço, Grégoire Hamon, Alain Girault, et Marc Pouzet. 2004. Towards a higher-order synchronous data-flow language. In *EMSOFT 2004 - Fourth ACM International Conference on Embedded Software*.

[17] Jean Louis Colaço, Bruno Pagano, et Marc Pouzet. 2018. SCADE 6: A formal language for embedded critical software development. In *Proceedings - 11th International Symposium on Theoretical Aspects of Software Engineering, TASE 2017*. DOI:https://doi.org/10.1109/TASE.2017.8285623

[18] Nick Collins, Alex McLean, Julian Rohrhuber, et Adrian Ward. 2003. Live coding in laptop performance. *Organised Sound* (2003). DOI:https://doi.org/10.1017/S135577180300030X

[19] Arshia Cont. 2008. Antescofo: Anticipatory synchronization and control of interactive parameters in computer music. In *International Computer Music Conference, ICMC 2008*.

[20] René David. 1995. Grafcet: A Powerful Tool for Specification of Logic Controllers. *IEEE Trans. Control Syst. Technol.* (1995). DOI:https://doi.org/10.1109/87.406973

[21] F. Daxecker. 2000. Der Jesuit Athanasius Kircher und sein Organum mathematicum. *Gesnerus* (2000).

[22] Ecma International. 2011. *ECMA-262 ECMAScript Language Specification. ECMA-262 5.1 Edition / June 2011.*

[23] Umberto Eco. 1965. *L'oeuvre ouverte* (Points éd.).

[24] Karlheinz Essl. 2007. Algorithmic composition. In *The Cambridge Companion to Electronic Music*. DOI:https://doi.org/10.1017/CCOL9780521868617.008

[25] Jose David Fernández et Francisco Vico. 2013. Ai methods in algorithmic composition: A comprehensive survey. *J. Artif. Intell. Res.* 48, (2013), 513-582. DOI:https://doi.org/10.1613/jair.3908

[26] Paula Findlen. 2004. *Athanasius Kircher: The last man who knew everything*. DOI:https://doi.org/10.4324/9780203643884

[27] Dominique Fober, Jean Bresson, Pierre Couprie, et I N A Grm. 2015. Les nouveaux espaces de la notation musicale. (2015).

[28] Dominique Fober, Yann Orlarey, et Stéphane Letz. 2012. INScore An Environment for the Design of Live Music Scores. In *Proceedings of the Linux Audio Conference*.

[29] Klaus Frieler, Martin Pfleiderer, Wolf Georg Zaddach, et Jakob Abeßer. 2015. Midlevel analysis of monophonic jazz solos: A new approach to the study of improvisation. *Music. Sci.* (2015). DOI:https://doi.org/10.1177/1029864916636440

[30] GeorgiaTech. WAC 2016. Consulté à l'adresse https://smartech.gatech.edu/handle/1853/54577/browse?type=title

[31] Marcelo Gimenes, Pierre-Emmanuel Largeron, et Eduardo Miranda. 2016. Frontiers: Expanding Musical Imagination With Audience Participation. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, 350-354. Consulté à l'adresse http://www.nime.org/proceedings/2016/nime2016_paper0068.pdf

[32] Francis Golffing et Umberto Eco. 1963. Opera aperta. *Books Abroad* (1963). DOI:https://doi.org/10.2307/40117464

[33] Google. Magenta. Consulté 10 mai 2020 à l'adresse https://magenta.tensorflow.org/

[34] Google. Tensorflow. Consulté 10 mai 2020 à l'adresse https://www.tensorflow.org/

[35] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, et Daniel Pilaud. 1991. The Synchronous Data Flow Programming Language Lustre. *Proc. IEEE* 79, 9 (1991), 1305-1320. DOI:https://doi.org/10.1109/5.97300

[36] D. Harel et A. Pnueli. 1985. On the Development of Reactive Systems. In *Logics and Models of Concurrent Systems*. DOI:https://doi.org/10.1007/978-3-642-82453-1_17

[37] Heidelein. Golem. Consulté 10 mai 2020 à l'adresse https://www.youtube.com/watch?v=MuZfpSgsIPo

[38] Henri Bergson. 1922. *Durée et Simultanéité*. Presses universitaires de France.

[39] Henri Bergson. 1934. *La Pensée et le Mouvant*. Presses universitaires de France.

[40] Dorien Herremans, Ching-Hua Chuan, et Elaine Chew. 2017. A Functional Taxonomy of Music Generation Systems. *ACM Comput. Surv.* (2017). DOI:https://doi.org/10.1145/3108242

[41] Abram Hindle. 2013. Swarmed: Captive portals, mobile devices, and audience participation in multi-user music performance. *Proc. Int. Conf. New Interfaces Music. Expr.* (2013), 174-179.

[42] Peter Hoffmann. 2009. Music Out of Nothing? A Rigorous Approach to Algorithmic Composition by Iannis Xenakis. *October* (2009).

[43] I. Xenakis. 1963. *Musiques formelles*. Editions R, Paris.

[44] Ivan Julien. 2005. *Traité de l'arrangement*. Media Musique.

[45] Ivanka Stoïanova. 1996. *Manuel d'analyse musicale*. Minerve.

[46] Maria Klara Jedrzejewska, Adrian Zjawinski, et Bartlomiej Stasiak. 2018. Generating musical expression of MIDI music with LSTM neural network. In *Proceedings - 2018 11th International Conference on Human System Interaction, HSI 2018*. DOI:https://doi.org/10.1109/HSI.2018.8431033

[47] Stefan Kersten et Rafael Ramirez. 2008. Concatenative Synthesis of Expressive Saxophone Performance. In *Sound and Music Computing Conference*.

[48] Charles Kœchlin. 1954. *Traité de l'orchestration Vol. 1*. Eschig.

[49] Jean-Philippe Lambert, Sébastien Robaszkiewicz, et Norbert Schnell. 2016. Synchronisation for Distributed Audio Rendering over Heterogeneous Devices, in HTML5. *Proc. 2nd Web Audio Conf.* (2016), 6-11.

[50] Phillip A. Laplante et Seppo J. Ovaska. 2011. Programming Languages for Real-Time Systems. In *Real-Time Systems Design and Analysis*. DOI:https://doi.org/10.1002/9781118136607.ch4

[51] Victor Lazzarini, Steven Yi, John Ffitch, Joachim Heintz, Øyvind Brandtsegg, et Iain McCurdy. 2016. *Csound: A sound and music computing system*. DOI:https://doi.org/10.1007/978-3-319-45370-5

[52] Sang Won Lee et Jason Freeman. 2013. echobo: A mobile music instrument designed for audience to play. *Proc. Int. Conf. New Interfaces Music. Expr.* 1001, (2013), 42121-48109.

[53] René Leibowitz. 1949. *Introduction à la musique de douze sons*. L'Arche.

[54] G. Levin. 2001. Dialtone: a Telesymphony premiere. *Ars Electron. Lynz Austria* (2001).

[55] Fredrik Lind et Andrew MacPherson. 2017. Soundtrap: A collaborative music studio with Web Audio. *Wac 2017* (2017), 3-4. Consulté à l'adresse https://qmro.qmul.ac.uk/xmlui/handle/123456789/26162

[56] Henning Lohner. 1986. The UPIC System: A User's Report. *Comput. Music J.* (1986). DOI:https://doi.org/10.2307/3680095

[57] James McCartney. 2002. Rethinking the computer music language: SuperCollider. *Comput. Music J.* (2002). DOI:https://doi.org/10.1162/014892602320991383

[58] Olivier Messiaen. 1994. *Traité de rythme, de couleur, et d'ornithologie*. Alphonse Leduc.

[59] Miguel Ricardo Pérez Pereira et Anderson Sebastián Salinas. 2017. Control of DMX lighting systems (digital multiplex) based on audible signals. *Rev. Vínculos* (2017). DOI:https://doi.org/10.14483/2322939X.13791

[60] Eduardo R. Miranda, Alexis Kirke, et Qijun Zhang. 2010. Artificial Evolution of Expressive Performance of Music: An Imitative Multi-Agent Systems Approach. *Comput. Music J.* (2010). DOI:https://doi.org/10.1162/comj.2010.34.1.80

[61] Miguel Cerdeira Negrão. 2018. NNdef: livecoding digital musical instruments in SuperCollider using functional reactive programming. DOI:https://doi.org/10.1145/3242903.3242905

[62] Vesa Norilo. 2011. Introducing Kronos A Novel Approach to Signal Processing Languages. *Proc. Linux Audio Conf.* January 2011 (2011).

[63] Y. Orlarey, S. Letz, et D. Fober. 2008. Multicore technologies in Jack and Faust. In *International Computer Music Conference, ICMC 2008*.

[64] Tetsuro Oshimi et Toshio Itabashi. 1988. PERSONAL COMPUTERS. *Mitsubishi Electr. Adv.* (1988).

[65] Thiti Panya-In, Jarernchai Chonpairot, et Manop Wisutthipat. 2015. The role of Jose

Maceda in the music research circle. *Wacana Seni* 14, (2015), 167-185.

[66] Bertrand Petit et Serrano Manuel. 2020. Skini: Reactive Programming for Interactive Structured Music. *Art, Sci. Eng. Program.* (2020).

[67] Bertrand Petit et Manuel Serrano. 2019. Composing and Performing Interactive Music using the HipHop . js language. In *New Interfaces for Musical Expression 2019*, 71--76. Consulté à l'adresse http://www.nime.org/proceedings/2019/nime2019%5C_paper014.pdf

[68] Carl Petri et Wolfgang Reisig. 2008. Petri net. *Scholarpedia* (2008). DOI:https://doi.org/10.4249/scholarpedia.6477

[69] Robert Prediger, Ralph Winzinger, Robert Prediger, et Ralph Winzinger. 2015. Node.js. In *Node.js*. DOI:https://doi.org/10.3139/9783446437586.fm

[70] Riccardo R. Pucella. 1998. Reactive programming in Standard ML. In *Proceedings of the IEEE International Conference on Computer Languages*. DOI:https://doi.org/10.1109/ICCL.1998.674156

[71] Miller Puckette. 2002. Max at seventeen. *Comput. Music J.* (2002). DOI:https://doi.org/10.1162/014892602320991356

[72] Miller Puckette. 2011. Étude de cas sur les logiciels pour artistes : Max / MSP et Pure Data. *Art++* (2011), 179-191.

[73] Casey Reas et Ben Fry. 2006. Processing: Programming for the media arts. *AI Soc.* (2006). DOI:https://doi.org/10.1007/s00146-006-0050-9

[74] Monk Rowe. 2011. Thoughts on Improvisation. *Jazz Backstory*.

[75] Patrick Schmoll. 2011. Sciences du jeu : état des lieux et perspectives. *Rev. des Sci. Soc.* (2011).

[76] Boris Schwarz, Leonard Stein, et Leo Black. 1976. Style and Idea. Selected Writings of Arnold Schoenberg. *Notes* (1976). DOI:https://doi.org/10.2307/897990

[77] Jonathan Segel. 2006. Native Instruments: Reaktor 3 Software SynthesizerLen Sasso: Native Instruments Reaktor 3—Wizoo Guide. *Comput. Music J.* (2006). DOI:https://doi.org/10.1162/comj.2003.27.1.109

[78] Michael A. Sells et Moshe Idel. 1995. Idel, « Golem: Jewish Magical and Mystical Traditions »Golem: Jewish Magical and Mystical Traditions on the Artificial Anthropoid. *Jewish Q. Rev.* (1995). DOI:https://doi.org/10.2307/1454744

[79] Manuel Serrano et Vincent Prunet. 2016. A glimpse of Hopjs. *Proc. 21st ACM SIGPLAN Int. Conf. Funct. Program. - ICFP 2016* (2016). DOI:https://doi.org/10.1145/2951913.2951916

[80] Patricia Shaw. 2002. Experimental music: Cage and beyond. *Musicology Australia*. DOI:https://doi.org/10.1080/08145857.2002.10416002

[81] Stephen Sinclair et Marcelo Mortensen Wanderley. 2007. Using PureData to

control a haptically-enabled virtual environment. *Proc. PureData Conv. ’07* (2007).

[82] Boris Smus. 2013. Web Audio API. *Mozilla Developer Network*.

[83] Laurie Spiegel. 1981. Manipulations of Musical Patterns. *Proc. Symp. Small Comput. Arts* 393 (1981), 19-22.

[84] Ariane Stolfi, Mathieu Barthet, Fábio Goródscy, Antonio Deusany, et Fernando Iazzetta. 2017. Open band: Audience Creative Participation Using Web Audio Synthesis. *Proc. Web Audio Conf.* (2017). Consulté à l’adresse http://eecs.qmul.ac.uk/~keno/11.pdf

[85] Bob L. Sturm. 2006. Adaptive Concatenative Sound Synthesis and Its Application to Micromontage Composition. *Comput. Music J.* 30, 4 (2006), 46-66. DOI:https://doi.org/10.1162/comj.2006.30.4.46

[86] Benjamin Taylor. 2017. A History of the Audience as a Speaker Array. di, 4 (2017), 481-486.

[87] Colin Vidal. 2018. *Programmation Web Typée*. Consulté à l’adresse http://hal.archives-ouvertes.fr/tel-00640566/

[88] Vincent d’Indy. 1902. *Cours de composition musicale*. Durand et Cie.

[89] Ge Wang. 2008. The ChucK Audio Programming Language “ A Strongly-timed and On-the-fly Environ / mentality.

[90] Vanessa Wang, Frank Salim, Peter Moskovits, Vanessa Wang, Frank Salim, et Peter Moskovits. 2013. The WebSocket Protocol. In *The Definitive Guide to HTML5 WebSocket*. DOI:https://doi.org/10.1007/978-1-4302-4741-8_3

[91] Nathan Weitzner, Jason Freeman, Stephen Garrett, et Yan-ling Chen. 2012. massMobile – an Audience Participation Framework. *NIME 2012 Proc. Int. Conf. New Interfaces Music. Expr.* (2012), 92-95. Consulté à l’adresse http://vhosts.eecs.umich.edu/nime2012/Proceedings/papers/128_Final_Manuscript.pdf

[92] Matthew Wright. 2005. Open Sound Control: An enabling technology for musical networking. *Organised Sound* (2005). DOI:https://doi.org/10.1017/S1355771805000932

[93] Yongmeng Wu, Leshao Zhang, Nick Bryan-Kinns, et Mathieu Barthet. 2017. Open Symphony: Creative Participation for Audiences of Live Music Performances. *IEEE Multimed.* (2017). DOI:https://doi.org/10.1109/MMUL.2017.19

[94] Xenakis. 1967. Vers une Métamusique. *La Nef* 29, (1967), 117-140.

[95] PiBox Music. Consulté 10 mai 2020 à l’adresse https://pibox.com/

[96] TrackD. Consulté 10 mai 2020 à l’adresse https://trackdmusic.com/

[97] GRAME. Consulté 10 mai 2020 à l’adresse http://www.grame.fr/

[98] SoundTrap. Consulté 10 mai 2020 à l'adresse https://www.soundtrap.com/

[99] Flat. Consulté 9 mai 2020 à l'adresse https://flat.io/fr

[100] Audiotool. Consulté 10 mai 2020 à l'adresse https://www.audiotool.com

[101] Audio Sauna. Consulté 10 mai 2020 à l'adresse http://www.audiosauna.com/

[102] Soundation. Consulté 10 mai 2020 à l'adresse https://soundation.com/

[103] Jam Studio. Consulté 10 mai 2020 à l'adresse http://www.jamstudio.com/Studio/index.htm

[104] Ujam. Consulté 10 mai 2020 à l'adresse https://www.ujam.com/

[105] World Wide Web Consortium. Consulté 10 mai 2020 à l'adresse https://www.w3.org/

[106] Reaper. Consulté 10 mai 2020 à l'adresse https://www.reaper.fm/

[107] CIRM. Consulté 10 mai 2020 à l'adresse https://www.cirm-manca.org/

[108] P5.js. Consulté 10 mai 2020 à l'adresse https://p5js.org/

[109] Window Subsystem for Linux. Consulté 10 mai 2020 à l'adresse https://docs.microsoft.com/fr-fr/windows/wsl/install-win10

[110] Synchron 2018. Consulté 6 mars 2020 à l'adresse https://project.inria.fr/synchron2018/fr/

[111] Frank Lalou. Consulté 10 mai 2020 à l'adresse https://www.lalou.net

[112] SACEM. Consulté 10 mai 2020 à l'adresse https://www.sacem.fr/

[113] QLC+. Consulté 10 mai 2020 à l'adresse https://www.qlcplus.org/

[114] 2016. Understanding ECMAScript 6. *Netw. Secur.* (2016). DOI:https://doi.org/10.1016/s1353-4858(16)30112-x