Outils informatiques et analyse musicale: représenter le routing de Jupiter

Maxence Larrieu

OUTILS INFORMATIQUES ET ANALYSE MUSICALE : REPRÉSENTER LE ROUTING DE *JUPITER*

Maxence Larrieu
LISAA
Université Paris-Est Marne-la-Vallée
maxence@larri.eu
orcid.org/0000-0002-1834-3007

RÉSUMÉ

Cet article traite du routing de la pièce *Jupiter* (1987), pour flûte et électronique en temps-réel, composée par Philippe Manoury avec l'assistance de Miller Puckette. Le routing de cette pièce, l'interconnexion entre les modules audio, pose un problème particulier pour l'analyste qui souhaite s'appuyer sur le « document électronique de la pièce », le *patch*. Le problème tient dans le fait qu'il n'est pas possible de savoir quels sont les modules actifs et leur embranchement à un passage donné. Dans le cadre de notre thèse nous avons été confrontés à ce problème et nous avons développé une solution *ad hoc,* laquelle est l'objet du présent article. Nous avons développé des programmes informatiques qui simulent le fonctionnement du patch, et cela exclusivement à partir des *qlist* – les fichiers alphanumériques qui intègrent les données de synthèse. De cette simulation nous avons obtenu des représentations qui rendent compte de l'état du routing sur toute la durée de la pièce. Après avoir présenté la pièce et ces programmes informatiques, nous expliquons comment nous avons utilisé ces résultats dans le cadre de notre analyse musicale.

1. INTRODUCTION

Jupiter, pour flûte et électronique en temps-réel, est une pièce pionnière du répertoire d'informatique musicale. Elle ouvre en effet le célèbre cycle *Sonus Ex Machina*, avec lequel Philippe Manoury a développé les premières réflexions d'une « musique en temps-réel » [7][1]. Composée dans les années 1985 avec l'aide de Miller Puckette, elle est créée en 1987 par Pierre-André Valade dans l'espace de projection de l'Ircam, lors d'un concert qui célèbre les 10 ans de l'institut[2].

Dans notre doctorat portant sur l'analyse des musiques d'informatique [4], nous avons travaillé conséquemment sur cette pièce. Celle-ci a notamment été choisie pour ses possibilités d'interconnexion entre modules, ce que nous désignons avec le terme *routing*, emprunté à l'ingénierie audio. En effet la composition de *Jupiter* se fait avec la possibilité de configurer un réseau d'une dizaine de modules tout au long de la pièce. Cette possibilité engendre néanmoins un contre-coup important : même avec le patch « sous les yeux », il n'est pas possible pour l'analyste de définir précisément quels sont les modules actifs à un passage donné. Ce problème n'est pas nouveau pour la communauté d'informatique musicale puisqu'il a été pointé en 2013 par Alain Bonardi sur une autre pièce du compositeur, *En Écho* (2003), dans le cadre d'un travail de reconstruction (cf. [1] et [2]).

Afin de mener à bien notre analyse nous avons dû trouver une solution à ce problème de routing. L'idée a été de simuler le fonctionnement du patch à l'aide uniquement des fichiers *qlist* et, durant cette simulation, d'en extraire les informations qui nous sont nécessaires. Nous avons conçu des programmes informatiques *ad hoc* qui scannent les 12 fichiers *qlist* de l'archive et qui simulent, à l'aide d'une forte compréhension du patch[3], le fonctionnement du routing. De cette simulation nous avons ensuite extrait des *chaînes de routing,* des interconnections entre modules, avec lesquelles il nous a été possible de répondre à la question : quels sont les modules qui produisent ce qui est perçu ? De là, nous avons pu réaliser notre analyse de *Jupiter* en croisant l'écoute et les connaissances acquises sur la production.

Dans cet article nous présentons le routing de *Jupiter* et le problème qui se pose à l'analyste, puis nous décrivons les trois programmes informatiques réalisés qui permettent de faire face à ce problème. Enfin nous présentons l'usage que nous avons fait des résultats dans le cadre de notre analyse musicale.

[1] Pour une introduction sur *Jupiter* voir [4], [6] et [8].

[2] Pour ses 40 ans en 2017, l'Ircam présente et diffuse les pièces de ce concert sur son site web (https://www.ircam.fr/article/detail/sons-dessus-dessous-28-retour-sur-le-10e-anniversaire/, 18 janvier 2018)

[3] Cette compréhension a abouti par exemple à une table qui décrit toutes les varibles du patch (242). cf. datapipes.okfnlabs.org/csv/html/?url=https://raw.githubusercontent.com/Podatus/Jupiter/master/jupiterVars/jupiterVars.csv, (visité le 30 mars 2018).

2. MODULES ET ROUTING DE *JUPITER*

2.1. Le patch, les modules

Pour analyser *Jupiter* nous avons utilisé un patch Pure Data[4], diffusé dans un article ambitieux qui a fait date, *New Public-Domain Realizations of Standard Pieces for Instruments and Live Electronics* de Miller Puckette [9]. Dans ce patch nous pouvons différencier plusieurs modules audio, dont les noms sont, pour la plupart, familiers. Il y a d'abord des modules de synthèse : Sampler, Additive, Chapo, Paf[5] ; puis des modules de traitements : Harmonizer, Frequency Shifter, Reverb, Noise et un Spatialisateur. La table suivante donne un aperçu de ces modules avec différentes données que nous utilisons par la suite.

Module	Type	Groupe	Nb signaux	Abbréviation
Flûte	synth		1	d
Sampler	synth		4	t
Additive	synth	osc	4	o
Chapo	synth	osc	4	o
Paf	synth	osc	1	o
Harmonizer	treat		1	h
FreqShift	treat		2	f
Reverb	treat		2	r
Noise	treat		1	n
Spat			4	4

Table 1. Données sur les modules du routing[6]

2.2. Le routing

Jupiter est une pièce singulière notamment pour le routing : il est flexible et la composition se fait en partie par sa définition dans le temps de la pièce. Cette possibilité de configurer à loisir le routing est permise d'abord par le « suiveur de partition ». Centrale à *Jupiter* – et plus largement à la musique en temps-réel – il permet d'articuler des événements électroniques avec l'interprétation qu'effectue le flûtiste. De plus, la flexibilité du routing s'explique par son implémentation : les 5 modules de synthèse sont tous reliés aux 4 modules de traitement, et parallèlement ces derniers sont tous reliés entre eux ; enfin, tous ces modules sont reliés au Spat. La configuration du routing passe alors par un ensemble de variables qui permet de doser la quantité de signal (en valeur MIDI) dans chacune de ces connexions. Cette flexibilité a un prix : une configuration totale du routing nécessite en théorie[7] 41 variables[8].

3. LE PROBLÈME DU ROUTING

3.1. Aperçu

Nous pouvons présenter le problème en imaginant la situation suivante : soit un musicologue qui souhaite analyser *Jupiter* à l'aide (notamment) du patch. Dans le travail à réaliser nous pouvons différencier deux étapes : une où il va effectuer des lectures du patch – de sorte à identifier et comprendre les 9 modules implémentés ; et une autre où il va se confronter à la manifestation sonore de la pièce. Le problème survient dans la seconde étape : confronté au sonore de la pièce il va avoir besoin de savoir quels sont les modules qui produisent le signal perçu ; quels sont ceux qui expliquent ce qui est écouté ? Or, comme nous allons le voir, il n'est pas possible de répondre directement à cette question.

3.2. Utilisation du routing

Le compositeur a utilisé le routing dans le temps de la pièce, « au fil des événements », et non de configuration type en configuration type. Par exemple le Frequency Shifter (FS par la suite) est souvent utilisé pour venir perturber un « tapis » maintenu par le Paf et/ou la Reverb (*e.g.* le début de la section IV) : sur un passage le compositeur va placer le FS sur la production de ces derniers modules puis quelques événements après cette connexion va être supprimée, puis, encore quelques événements après, il va par exemple activer l'envoi de la Reverb vers l'Harmonizer, puis de ce dernier vers le FS, etc. Avec cet exemple nous souhaitons montrer que le routing possède une écriture *persistante* : une connexion reste dans son état tant qu'elle n'est pas modifiée (ré-écrite). Ainsi, l'analyste qui souhaite savoir la configuration du routing à tel événement devra remonter la vingtaine de variables[9] du routing. Ceci reste envisageable dans les premières minutes, mais devient impossible sur la demi-heure de la pièce.

Le problème auquel est confronté l'analyste vient d'une part des 21 variables utilisées pour configurer le routing, et de l'autre de leur utilisation intensive ; la flexibilité du routing appelle en contre-partie une *difficulté de représentation.*

[4] Ce patch est couramment nommé « la version de Miller Puckette » par la communauté. Soulignons que Miller Puckette comme Philippe Manoury nous ont assuré que ce patch a été utilisé en concert.

[5] Nous utilisons des majuscules afin de différencier ces modules des méthodes de synthèse qu'ils sous-tendent.

[6] Le lecteur spécialiste aura remarqué que nous n'intégrons pas le module de traitement nommé phasor. En effet celui-ci n'est pas relié au routing : il est appliqué tout au long de la pièce sur la flûte et diffusé dans la stéréophonie.

[7] En théorie, car tous ce que contient le patch n'est pas nécessairement utilisé.

[8] 5 modules de synthèse reliés aux 4 modules de traitements ; 4 modules de traitement reliés entre eux ; et les 9 modules reliés au Spat.

[9] Toutes les connexions entre modules ne sont pas utilisées par le compositeur, nous avons ainsi trouvé dans les *qlist* l'usage de 21 variables dédiées au routing.

4. UNE SOLUTION

Puisque le patch et les *qlist* sont seules responsables de l'électronique, de ce qui est perçu, il doit être possible de rendre compte de ces embranchements entre modules. Une telle prise de conscience est par exemple présente chez Alain Bonardi, dans le cadre d'un travail de reconstruction de la pièce *En Echo* (1993). Le patch est exécuté, écouté, et les connexions entre modules sont représentées dans le patch, graphiquement, à l'aide d'un « diagramme fonctionnel » [2]. Cette solution convient, mais elle ne nous permet pas de connaître, hors temps, des connexions présentes à un événement donné ; elle n'amène pas un support qui rende compte du routing en dehors du temps de la pièce.

Nous présentons ci-après une solution voisine où le patch à été simulé uniquement à partir des *qlist* – *i.e.* sans computation audio – et d'un ensemble de règles. Cette solution est faite de trois étapes : la première relève les pré-états des modules ; la deuxième les connexions prescrites ; la dernière utilise les précédents résultats et en déduit les connexions effectives entre modules.

Précisons avant de commencer que nous avons travaillé avec une précision à l'échelle des événements, sans tenir compte des quelques occurrences de connexions qui se font avec des temps de lissage.

Aussi, les programmes que nous avons conçus se basent sur un typage des modules, ceux de synthèse et ceux de traitement. Afin de ne pas alourdir la suite de notre écrit nous utilisons les expressions « les *synth* » et « les *treat* » pour désigner réciproquement les modules de synthèse, qui produisent des signaux, et ceux de traitement, qui les modifient.

Enfin, pour plus de clarté, nous avons été amenés à différencier le routing *interne* de celui *externe* ; le premier fait référence aux connexions entre modules, le second aux connexions vers le système de diffusion – une stéréophonie classique au-devant de la scène et une quadriphonie qui entoure le public.

4.1. Les Pré-états des modules

Cette première étape permet de savoir si les modules sont actifs ou non. Mais il ne peut s'agir ici que d'un état premier (pré-état), puisque avant de pouvoir connaître l'état (final) il est nécessaire de connaître l'acheminement du signal à la diffusion[10].

Pour les *synth,* cette étape consiste à savoir s'ils produisent du signal ; pour les *treat,* s'ils sont aptes à en produire.

Pour les *synth* et à l'exception du Paf, nous avons procédé à une détection manuelle à l'aide des *qlist*, de la partition et de l'écoute ; pour les *treat*, à une détection automatique avec un programme informatique.

[10] Comme l'a montré Alain Bonardi [2] il est en effet possible que des embranchements n'aboutissent pas – comme une conséquence de la difficulté de représentation du routing.

4.1.1. Détection manuelle

Pour le Sampler nous pouvons aisément différencier deux utilisations. Une à partir d'écriture dans les *qlist* avec la variable *addsamp* et une autre à partir de lecture de fichiers (*e.g. ost6-start*) où les données sont lues et affectées aux modules de synthèse – à l'instar d'un lecteur de données MIDI. Pour la première utilisation nous avons lu les *qlist* et appliqué la règle suivante : si une des voies est active durant l'événement alors le module est actif, et inversement. Le Sampler est en effet particulier puisque le signal produit est nécessairement limité dans le temps. Pour la seconde utilisation nous avons eu recours à la partition et à l'écoute. La détection est aisée puisque cet usage du Sampler est très marqué, il correspond en effet aux *ostinatos*, éléments très saillants de la pièce.

Pour les modules Additive et Chapo nous avons aussi lu les *qlist*. Simplement, à la différence du Sampler ils sont inactifs quand *toutes* leurs voies possèdent une amplitude nulle.

4.1.2. Détection automatique

Pour le Paf et les *treat* nous avons réalisé un programme – *a_preStateOfMods.pde*[11] – qui déduit automatiquement les états à partir des *qlist*. Cette déduction est faite à chaque événement, qu'il y ait ou non de nouvelles écritures. La figure ci-dessous illustre le fonctionnement du programme : il reçoit en entrée tous les fichiers *qlist,* puis, après application de règles, il construit une table – nommée *preStatesOfMods.tsv* – qui représente sous forme booléenne les états des modules, événement par événement.

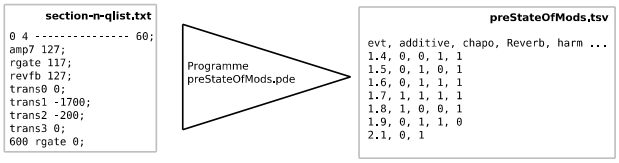

Figure 1. Illustration du programme de détection des pré-états des modules.

Tous les modules à traiter possèdent une structure de données commune (*e.g.* nom, type, état, variables). En utilisant la Programmation Orientée Objet (POO par suite) nous avons alors créé une classe, nommée Module, qui définit cette structure de données. Une seconde classe attachée à la précédente est aussi nécessaire. Nommée DictList elle a pour fonction de relier une variable à une liste de nombres ; par exemple la chaîne de caractère *fpos* à toutes les occurrences de cette variable.

[11] Nos programmes sont conçus avec le langage Processing. Ils sont disponibles avec les résultats à l'adresse suivante : github.com/Podatus/Jupiter (visité le 30 mars 2018).

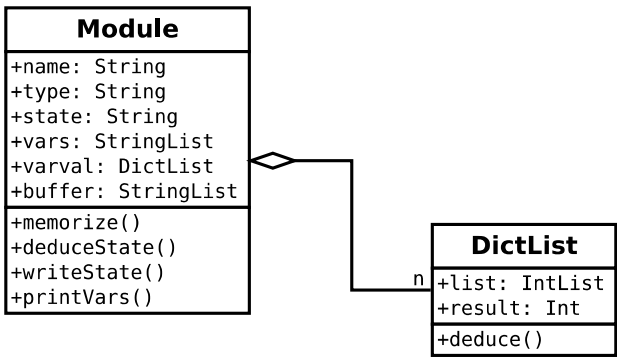

Figure 2. Schéma UML[12] de la structuration du premier programme.

Le programme s'initie en créant une instance de la classe Module par modules à analyser. Le *n* du précédent schéma précise que le nombre de variables attachés aux modules n'est pas fixe, comme en témoigne la table ci-dessous.

Module	Type	Variables
paf	synth	amp1, amp2, amp3, amp4, amp5, amp6, amp7, amp8
reverb	treat	rout
harm	treat	hamp1, hamp2, ham3, ham4
freqshift	treat	fpos, fneg

Table 2. Données des modules utilisées pour les instances du premier programme.

Les états des modules sont alors déduits à partir de trois règles, que nous pouvons ainsi présenter : si durant un événement la variable de sortie d'un Module

- a eu au moins une occurrence supérieure à 0, alors le module est actif ;
- n'a eu que des occurrences égales à 0, alors le module est inactif ;
- n'a eu aucune occurrence, alors le module reste dans l'état précédent.

Dans cette simulation les difficultés rencontrées viennent des variables de substitutions, *i.e.* des variables de plus haut niveau qui affectent un ensemble de variables, par exemple *amptutti6* affecte l'amplitude des 6 premières voies du Paf, *amp1, amp2 … amp6.*

4.2. Extraire le routing

La deuxième étape extrait les données de routing, interne comme externe. Si la première étape permettait de répondre à la question « quels modules produisent (au niveau du patch) du signal ? », la deuxième permet de répondre à la suivante « quelles sont les connexions présentes entre modules ?».

[12] Unified Modelling Language

Cette étape est réalisée automatiquement à l'aide d'un programme – *b_modsConnections.pde* – assez similaire au précédent. C'est en effet la même structuration en POO qui est utilisée, mais cette fois-ci avec une instanciation pour tous les modules, la flûte comprise, et bien évidemment avec toutes les variables de routing comme le présente la table ci-dessous.

Module	Type	Variables
flute	synth	dtor, dtoh, dtof, dton, dto4
sampler	treat	ttor, ttoh, ttof, tton, tto4, tto2
osc	treat	otor, otoh, otof, oton, oto4, oto2, noise1, noise2, noise3, noise4, noise5
reverb	treat	rtoh, rtof, rton, rto4, rto2
harm	treat	htor, htof, hton, hto4, hto2
freqshift	treat	ftor, ftoh, fton, fto4, fto2
noise	treat	ntor, ntoh, ntof, nto4, nto2

Table 3. Données des modules utilisées pour créer les instances du deuxième programme.

Pour chaque événement le programme extrait les données du routing puis déduit les connexions entre modules. Le routing est ensuite représenté à l'aide d'une table – *modsConnections.tsv* –, avec une ligne pour chaque événement et une colonne pour chacun des modules. Enfin, le routing est indiqué à partir des envois. Par exemple, si la Reverb est connectée à l'Harmonizer à tel événement le programme inscrit *rtoh* dans la colonne Reverb à l'événement correspondant.

Dans cette étape les difficultés rencontrées viennent du groupement de module « osc » – pour oscillateur – qui regroupe Paf, Additive et Chapo. Une difficulté supplémentaire se présente à cause de la connexion du Paf vers le Noise, qui se fait à la fois à l'échelle du groupement de module – la variable *oton* – et à l'échelle des canaux du Paf – les variables *noise1, noise2 … noise5.* Enfin, il y a une variable de substitution à considérer qui affecte et le Sampler et le groupement de module « osc », *ston* affecte *tton* et *oton.*

4.3. Déduire les états des modules et le routing

La dernière étape permet de répondre à la question initiale, « quels sont les modules actifs à tel événement et quels sont leurs connexions ? », soit de rendre compte de la production. La réponse est obtenue intégralement à l'aide d'un programme – *c_stateAndRouting.pde*. Il croise les résultats des précédentes étapes et déduit, à l'aide de règles, les connexions et les états des modules. C'est une étape conséquente puisqu'il nous a été nécessaire de concevoir un algorithme qui traverse le réseau du routing.

4.3.1. *Fonctionnement*

Le programme utilise de nouveau la POO avec une classe Module, qui définit la structure de données et les méthodes communes aux instances.

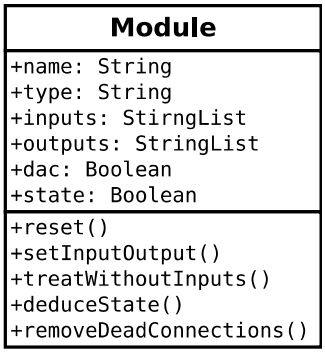

Figure 3. La classe Module du troisième programme.

Les attributs *inputs* et *outputs* permettent de stocker les différentes connexions des modules ; les attributs booléens *dac* et *state* de stocker la connexion à la diffusion et l'état du module.

Le programme crée une instance pour les 8 modules (Flûte, Sampler, Osc, Reverb, Harm, FreqShift, Noise et Spat) puis croise, événement par événement, les résultats des précédentes étapes. Les modules actifs sont d'abord sélectionnés – la table *preSatesOfMods.tsv* – puis les connexions entre modules sont affectées aux attributs *inputs* et *outputs* des modules correspondants. Par exemple, la lecture de *rtoh* dans la table *modsConnections.tsv* à un événement donné ajoute le terme *harm* à l'attribut *outputs* de l'instance *reverb* et le terme *reverb* à l'attribut *intput* de l'instance *harm.* De cette façon, il est possible d'avoir une représentation exhaustive du routing interne. Par exemple, dans la conception du programme il nous a été nécessaire d'avoir un retour sur l'état du routing ; nous avons opté pour la représentation suivante :

```
flute out(reverb)
noise in(osc,reverb) out(spat)
spat in(noise,osc,reverb)
osc out(noise,spat)
reverb in(flute) out(noise,spat)
```

4.3.2. *Règles*

Une fois les connexions établies deux règles ont été appliquées afin d'obtenir un routing cohérent :

- Si un module n'a aucune sortie, ni *dac* ni routing interne, alors il est inactif ;
- Si un module, hormis la Reverb[13], est un *treat* et n'a aucune entrée, alors il est inactif[14].

Ensuite, les états finaux des modules sont déduits – la méthode *deduceState()* – et les connexions mortes sont retirées – la méthode *removeDeadConnections().*

4.3.3. *Chaînes de routing et algorithme*

La précédente représentation du routing est nettement trop lourde pour rendre compte du routing sur la demi-heure de la pièce. Afin de l'alléger nous avons pensé le routing à partir de chaînes, d'interconnexions entre modules. Une chaîne commence nécessairement par un *synth* et finit soit dans un *dac* soit dans le Spat. La transformation de la précédente représentation du routing à une nouvelle, avec ce principe de chaînes, a été réalisée par un algorithme. Ce dernier doit, tout en mémorisant leurs noms et leurs successions, traverser toutes les instances actives interconnectées jusqu'à arriver à une connexion au *dac* ou au Spat. La représentation finale que nous avons choisie est la suivante, avec en haut le numéro de la section et l'événement, puis les chaînes de routing, et enfin les modules reliés au *dac.*

```
10.12
flute>spat
sampler>reverb
sampler>harm>noise>spat
sampler>harm>reverb
dac : harm,reverb,sampler
```

Cette représentation est elle-même utilisée dans le programme pour comparer le routing d'un événement à l'autre afin de savoir s'il y a eu un changement ou non. Le fichier final – *routingChains.txt*[15] – de notre solution contient donc cette représentation pour chaque changement de routing. Avec un saut de ligne avant les événements, il contient environ 1700 lignes.

4.3.4. *Difficultés*

La principale difficulté a été de concevoir l'algorithme. Celui-ci n'est pas évident puisqu'il doit être récursif : le chemin à parcourir d'instance en instance n'est pas connu à l'avance mais se dessine au fur et à mesure de la traversée des instances. Dans le précédent exemple une fois la chaîne *sampler>reverb* trouvée, il faut remonter au Sampler pour investir l'autre connexion qui va à l'Harmonizer, et ainsi de suite. Lorsque l'algorithme se trouve à une instance donnée, toutes les connexions sont donc à placer dans un tampon, et de même pour le chemin qui a mené à cette instance.

[13] La Reverb possède en effet un mode infini avec lequel du signal est produit même s'il n'y a rien en entrée.

[14] La méthode *treatWithoutInputs()*

[15] cf. github.com/Podatus/Jupiter/blob/master/routing/c_statesAndRouting/routingChains.txt, (visité le 30 mars 2018).

5. UTILISATION DES RÉSULTATS POUR L'ANALYSE MUSICALE

Nous présentons ci-après l'usage que nous avons fait des précédents résultats lors de l'analyse musicale. Ces résultats nous ont servi à deux reprises. D'abord lors des premiers contacts avec le sonore de la pièce, de sorte à avoir un aperçu global, et ensuite, plus substantiellement, de sorte à expliquer ce que nous percevons.

5.1. Les états des modules

La première utilisation concerne exclusivement les états des Modules. Dans différentes analyses écrites de *Jupiter* nous avons constaté une pratique récurrente, celle d'éclairer les 30 minutes de la pièce avec l'enchaînement des modules actifs. Une représentation classique en deux dimensions est ainsi souvent donnée : la durée est présente sur l'horizontale avec les 13 sections, et les modules sont placés verticalement. L'éclairage consiste à donner les modules qui sont actifs pour les différentes sections (*e.g.* Andrew May [8]). Évidemment, avec notre précédent fichier il est possible d'aller plus loin : nous avons d'une part une discrétisation temporelle plus fine, celle des événements, et d'autre part nous avons une indication des modules actifs qui est basée sur une simulation du patch, qui dépasse donc l'écoute.

Nous avons donc prolongé la précédente représentation avec nos résultats. Le temps est présent en abscisse selon la succession des événements et non des minutes, et nous avons ajouté les segmentations de la partition : dessous le numéro des événements sont présents les lettres des sous-sections puis, dessous, le numéro des sections. Représenter plus de 500 événements demande cependant une taille conséquente ; nous plaçons donc ci-après un extrait de cette représentation, et nous la diffusons en pleine largeur sur le Web[16].

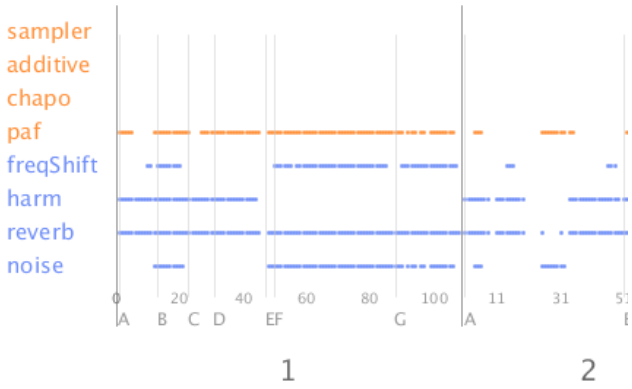

Figure 4. Extrait de la représentation des états des Modules.

[16] DOI : 10.6084/m9.figshare.5817459.v1

Avec la version intégrale nous remarquons de prime abord que les modules de traitement sont amplement utilisés. De plus, accompagnée d'une écoute cette représentation permet d'effectuer deux groupements de sections : la 6 et la 12, avec l'utilisation du Chapo, et un autre groupe plus vaste 1, 2, 4, 7 et 13 où l'on trouve les même modules.

Il faut cependant rester très prudent avec une telle représentation[17]. Il est en effet primordial de la relier à l'écoute et/ou à la partition, sans quoi il serait possible d'emprunter des chemins de compréhension déviants.

5.2. Le routing

Enfin notre analyse s'est fortement appuyée sur les chaînes de routing précédemment identifiées. Au fil de la lecture de ce fichier et de la pratique de notre analyse nous nous sommes aperçus que la précision obtenue était trop grande comparée à l'usage du routing qui a été fait dans la composition. C'est typique par exemple pour le Noise qui n'est pas un module « support de la composition », mais davantage un module de mixage : il vient brouiller un signal qui est déjà diffusé. Cela se retrouve aussi dans les nombreuses connexions des modules au *dac*. On s'aperçoit qu'il y a un *usage* récurrent des *treat* qui consiste moins à créer une nouvelle sonorité qu'a venir enrichir, brouiller, une déjà diffusée. Cet usage-là doit être dépassé par l'analyse musicale. Une analyse qui se baserait sur les connexions entre le Noise et la Reverb (en dehors du mode infini) et le FS resterait en effet lacunaire. Dans le même temps, et c'est fondamental, cet usage-là participe à l'empreinte sonore de *Jupiter*. Les ignorer serait une erreur, comme en atteste leur forte présence sur la durée de la pièce. Finalement, cet usage des *treat* s'éclaire bien avec une métaphore que le compositeur nous a transmise : « les modules de synthèse de l'époque sont comme des autoroutes, droites et monotones ; les modules de traitements permettent de faire face à cette uniformité de sorte que l'on ne s'ennuie pas dans l'écoute »[18].

Le fichier texte avec les chaînes de routing nous a ainsi été utile avec une lecture experte, de sorte à dépasser cette trop grande quantité, en relevant dans les changements ceux qui sont importants de ceux qui ne le sont pas. Avec ce fichier, nous avons pu éclairer des sonorités très surprenantes, comme l'événement 14 de la section 12, qui s'expliquent, en plus de la configuration des modules, par le routing, comme le présente la figure ci-dessous.

[17] Pour l'anecdote, on se souviendra que la première réaction de Philippe Manoury quand nous lui avions présenté cette représentation graphique fût « qu'est-ce que vous faite de cela ? »

[18] Métaphore retranscrite lors d'un entretien avec le compositeur à l'Ircam en novembre 2016.

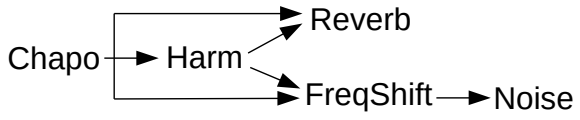

Figure 5. Exemple d'une configuration du routing. Événement 14, section 12.

6. DISCUSSION

À l'issue de ce travail se pose la question de l'adaptabilité de notre démarche à d'autres pièces. Précisons d'abord que les programmes développés ne sont pour l'heure fonctionnels que sur la pièce *Jupiter*, ils ont en effet été construits avec toutes les idiosyncrasies du patch[19], de sorte justement à les dépasser. Les pièces qui demanderaient le moins de remaniement des programmes sont bien évidemment celles du cycle *Sonus Ex Machina*, car composée par les mêmes personnes. Ensuite, en dehors d'une application rapide sur d'autres pièces, il nous semble qu'il y a dans ce travail une approche originale que nous pouvons éclairer ci-après mais qui mériterait d'être approfondie ultérieurement. Notre approche nous paraît singulière dans la mesure où nous utilisons les *qlist* dans le but d'éclairer la pièce dans sa durée, c'est-à-dire au-delà de la microstructure. Tous les *qlist* ont en effet été utilisés de sorte à éclairer – dans cet article – les embranchements des modules. Partir du postulat que toutes les informations relatives à l'électronique, sur la durée de la pièce, sont présentes dans le patch et les *qlist* est peut-être évident, en revanche, ce qui l'est moins, c'est d'aller vers cette densité et de l'exploiter dans une perspective d'analyse musicale : les *qlist* se situent à un bas niveau d'abstraction, 8000 lignes : utiliser cette quantité nécessite un pas conceptuel. Nous avons franchi celui-ci en deux temps : d'abord une forte compréhension de tous les modules et du fonctionnement du patch, ensuite une représentation de ces modules (POO dans notre cas) et du fonctionnement du patch dans des programmes informatiques.
Notre approche nous paraît donc singulière puisqu'elle va exploiter, à l'aide de représentations informatiques sur mesures, les données bas niveau présentes dans les *qlist*, dans le but d'en extraire des informations de plus haut niveau, éclairantes pour l'analyse musicale sur la durée de la pièce.

7. CONCLUSION

Dans cet article nous avons traité du problème du routing qui survient dans l'analyse musicale de *Jupiter*. Le problème tient dans le fait qu'il n'est pas possible pour l'analyste de connaître précisément les modules responsables de la production électronique à un passage donné. Nous avons expliqué la solution que nous avons réalisée dans le cadre de notre thèse [4]. Nous sommes arrivés à une représentation exhaustive et hors temps du routing, sans laquelle nous n'aurions pu développer notre analyse musicale. Les documents obtenus, confrontés à l'écoute, nous ont permis de relever une spécificité de la pièce : son empreinte sonore ne s'explique pas par un ou deux modules de traitement, mais par un réseau non linéaire de modules – dans le sens où des modules intermédiaires diffusent directement dans le *dac* et/ou le Spat –, qui évolue au fil des événements.

Enfin, la méthode que nous avons réalisée pour affronter le problème du routing nous paraît aussi originale pour l'analyse musicale. En effet avec une forte compréhension du patch nous avons pu simuler son fonctionnement, et cela en dehors de la computation audio. C'est une pratique originale puisque d'une part elle aboutit à un document qui est en dehors du temps de l'actualisation de la musique – ce qui autorise une saisie *in extenso –,* et d'autre part puisque nous avons utilisé l'outil informatique pour construire des représentations à partir d'un bas niveau d'abstraction (les *qlist*). Cette pratique pourrait aussi s'avérer féconde pour l'analyse d'autres pièces d'informatique musicale.

[19] Pensons par exemple aux variables de substiution (*e.g. ston* qui affecte *tton* et *oton)* ou plus fortement aux connexions du Paf vers le Noïse qui se font à l'échelle des voies (*noise1, noise2 …)*. Tout cela doit se retrouver dans les programmes

8. REMERCIEMENT

Je remercie Philippe Manoury pour les échanges que nous avons eus.

9. RÉFÉRENCES

[1] Bonardi, A. "Analyser et représenter la production du son dans les œuvres mixtes. Exemple du patch Max/MSP de *En Echo* de Philippe Manoury" Présentation orale dans le cadre du séminaire *Analyse par segmentation des musiques électroacoustiques,* Saint-Étienne, 2013.

[2] Bonardi, A. "Analyser l'orchestre électronique interactif dans les œuvres de Manoury", *Analyser la musique mixte*, Delatour, Sampzon, 2017. hal-01575296

[3] Clarke, M. "Jonathan Harvey's *Mortuos Plango, Vivo Voco", Analytical methods of electroacoustic music,* dir. Mary Symony. Routledge, London, 2006.

[4] Di Scipio, A. "An Analysis of Jean-Claude Risset's Contours", Journal of New Music Research, vol.29, n°1, p 1-21, 2000.

[5] Dufeu, F. "Comment développer des outils généraux pour l'étude des instruments de musique

numériques ? Un prototype en JavaScript pour l'investigation de programmes Max", Revue Francophone d'Informatique Musicale, n°3, 2013. revues.mshparisnord.org/rfim/index.php?id=255 (visité le 30 mars 2018)

[6] Goody, J. *La raison graphique. La domestication de la pensée sauvage*, Édition de Minuit, Paris, 1979.

[7] Larrieu, M. "Analyse des musiques d'informatique : vers une intégration de l'artefact. Proposition théorique et application sur *Jupiter* (1987) de Philippe Manoury". Thèse, Université Paris-Est Marne-la-Vallée, 2018. tel-01757277.

[8] Lemouton, S. "Vingt ans de pratique de la Réalisation en Informatique Musicale". Mémoire de Master 2, Université Paris-Est Marne-la-Vallée, 2012.

[9] Manoury, P., Battier, M., Bonardi, A., Lemouton, S. *Les musiques électroniques de Philippe Manoury*, Ircam, Paris, 2003.

[10] Manoury, P. "Considérations (toujours actuelles) sur l'état de la musique en temps réel", l'Étincelle, n°3, Ircam, Paris, 2007.

[11] May, A. "Philippe Manoury's Jupiter", *Analytical methods of electroacoustic music*, dir. Mary Symony. Routledge, London, 2006.

[12] Puckette, M. "New Public-Domain Realizations of Standard Pieces for Instruments and Live Electronics", International Computer Music Conference, Havana, Cuba, 2001.

[13] Puckette, M. *The theory and technique of electronic music*, World Scientific Publishing Co Inc, Singapore, 2006.

[14] Risset, J.-C. "Problème posés par l'analyse d'oeuvres musicales dont la réalisation fait appel à l'informatique"*, Analyse et création musicales : actes du Troisième congrès Européen d'Analyse Musicale (Montpellier, 1995)*. Harmattan, Paris, 2001.

[15] Will, U. "La baguette magique de l'ethnomusicologue : repenser la notation et l'analyse de la musique", *Cahiers de musique traditionnelles,* n° 12, 1999. journals.openedition.org/ethnomusicologie/671 (visité le 30 mars 2018).

[16] Zattra, L. "Génétiques de la computer music", *Genèses musicales*, dir. Nicolas Donin, A. Grésillon, J.-L. Lebrave. Presse universitaire de Paris-Sorbonne, Paris, 2015.