Numéros / n° 3 - automne 2013

« Comment développer des outils généraux pour l'étude des instruments de musique numériques ? »

Frédéric Dufeu

Résumé

La musicologie des pratiques électroacoustiques reposant sur des dispositifs numériques peut largement s'appuyer sur l'étude des environnements informatiques eux-mêmes. À partir d'un prototype de visualisation et de navigation de programmes Max écrit en JavaScript (1), cet article interroge la possibilité et l'utilité de développer des outils permettant une investigation organologique dépassant le cadre d'une seule étude de cas.

Introduction

« Les instruments de musique doivent à la diversité ou à l'étrangeté de leurs formes, également à la complexité des phénomènes qui s'y produisent, d'avoir suscité une attention que le musicologue partage avec d'autres spécialistes, physicien, archéologue, ethnologue. » (Schaeffner, 1980). Les propos d'André Schaeffner sur les instruments mécanico-acoustiques peuvent-ils être transposés au domaine des outils de création musicale ayant recours à l'informatique ? Si les développements de nouvelles lutheries reposant sur les technologies numériques ont depuis plus d'un demi-siècle appelé des compétences extrêmement diversifiées (2), leur histoire relativement récente ne leur a sans doute pas encore permis de recevoir un intérêt aussi large que celui décrit par l'ethnologue pour un ensemble aux origines très anciennes (Voir notamment Schaeffner, 1936). Pourtant, le « bouillonnement organologique » contemporain décrit par Bernard Stiegler (2003, p. 12) est d'ores et déjà l'objet d'un certain nombre de travaux de recherche. Une part importante des études des dispositifs technologiques dédiés à la musique est motivée par l'urgence des questions de préservation de la création contemporaine, auxquelles la *Revue francophone d'informatique musicale* a consacré son deuxième numéro (voir également Lemouton, Bonardi, Ciavarella, 2013). D'autres publications sont plus directement concernées par l'analyse musicale des oeuvres dont la composition ou l'interprétation reposent sur la technologie (voir notamment Battier, 2003 ; Baudouin, 2007 ; Dahan, 2007). Entreprises de préservation et musicologie des pratiques électroacoustiques peuvent ainsi se nourrir mutuellement à partir d'une volonté commune de comprendre et transmettre (voir par exemple de Sousa Dias, 2007) le comportement des instruments de musique numériques (3).

Selon Stiegler, « [l']histoire de la musique est d'abord (au moins chronologiquement) celle de ses instruments, et il n'y a pas de musique sans dispositifs instrumentaux, dont l'évolution conditionne celle de la musique, et inversement : c'est ce que Schaeffner appela la "pression des instruments" (4). » (Stiegler, 2003, p. 11) Dans le prolongement de cette perspective, il se conçoit que l'analyse des processus de création des oeuvres électroacoustiques recourant au numérique soit étroitement liée à l'étude des programmes informatiques constituant les dispositifs par lesquelles elles sont réalisées (Dufeu, 2010). Une approche organologique des environnements développés pour mettre en jeu des comportements instrumentaux particuliers à une situation compositionnelle ou interprétative donnée doit permettre de saisir l'ensemble du potentiel expressif dans lequel le musicien se situe, au-delà des choix qu'il opère pour l'actualisation de l'oeuvre, tels qu'ils peuvent par exemple être documentés par l'enregistrement d'une performance. Si la grande spécificité des programmes développés pour chaque projet esthétique implique une démarche d'étude hautement individualisée, l'objectif de cet article est de soulever la possibilité de méthodes applicables à un corpus et non à une seule oeuvre (5). Pour amorcer la réflexion sur l'élaboration

d'outils musicologiques d'investigation des instruments numériques dont la portée dépasse le cadre d'une seule analyse, nous présentons un dispositif prototypé en JavaScript pour Max, un environnement d'informatique musicale très largement répandu dans la création contemporaine. À partir d'un principe trivial d'information de l'ampleur du programme devant être approché, la présentation de ses applications à des fins musicologiques et de ses limites dans un tel contexte conduit à soulever l'importance, pour l'implémentation d'outils d'analyse visant une certaine généralité, de la capacité d'un langage informatique à renseigner sur ses propres instanciations.

1. De la quantification du programme à un mode synoptique de représentation

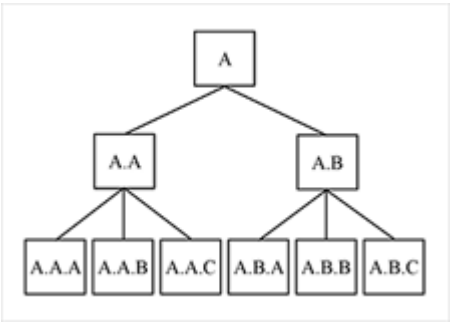
1. 1. Mesure de la taille d'un programme Max

Le prototype que nous présentons a été initialement motivé par une question simple : comment mesurer l'ampleur d'un programme dédié à une performance particulière ? Dans le cadre de nos recherches visant l'étude de plusieurs oeuvres dont l'exécution repose sur des environnements informatiques temps réel programmés en Max (6), un moyen d'évaluer rapidement la complexité d'un *patch* (7) s'est imposé. Contrairement aux langages de programmation textuelle, avec lesquels la lecture linéaire du code peut donner une première mesure de l'importance du travail d'analyse à accomplir, l'encapsulation récursive d'ensembles de fonctions dans des *subpatchers* (8) rend toute vision générale immédiate du programme difficile, les cas servant à une interactivité approfondie entre instrument ou voix et dispositif électroacoustique étant généralement constitués de très nombreux objets répartis dans différentes régions du *patch* parent. Au-delà de la taille en Mégaoctets du document Max lui-même, le nombre de *subpatchers* et d'objets est un premier niveau d'information utile. Dans Max, l'intégration d'un code en JavaScript permet de parcourir chaque *subpatcher* et d'en recenser le nombre d'objets. Par exemple, le *patch* d'exécution de *Jupiter* (1987) (9) pour flûte et électronique temps-réel de Philippe Manoury contient 14 290 objets Max, répartis dans 444 *subpatchers* ; le *patch* d'exécution de *Partita I* (2006) (10) pour alto et électronique du même compositeur contient 40 111 objets Max, répartis dans 1 749 *subpatchers*.

1. 2. Visualisation synoptique de la structure du *patch*

Cette première étape de quantification a été prolongée par l'implémentation d'un mode de visualisation, toujours en JavaScript (11), de la structure du *patch*. Ce mode de visualisation repose sur les principes suivants : soit le *patch* parent dit A, contenant deux *subpatchers* dits A.A et A.B, eux-mêmes contenant chacun trois *subpatchers* dits respectivement A.A.A, A.A.B, A.A.C et A.B.A, A.B.B, A.B.C. Le *patch* global sera représenté selon le schéma de la figure 1.

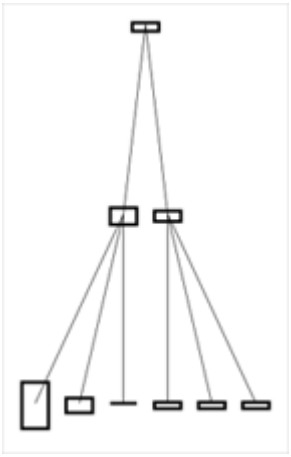
Figure 1. Principe schématique du mode de visualisation d'un *patch* simple



À partir de ce schéma général, la dimension verticale de chaque rectangle représente la taille en nombre d'objets du *subpatcher* correspondant. Si A ne contient que ses deux *subpatchers* A.A et A.B, A.A trois objets en plus de ses trois *subpatchers*, A.B aucun objet supplémentaire, et si les six *subpatchers* de troisième niveau contiennent respectivement 20, 5, 0, 1, 1 et 1 objets, le prototype donne la visualisation

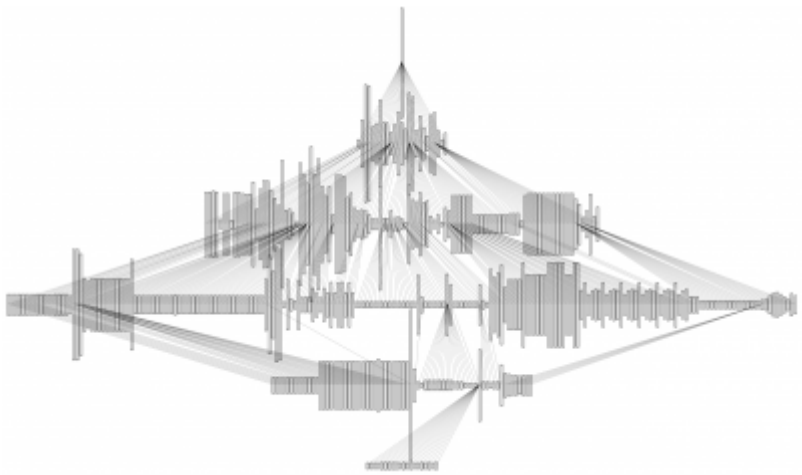
de la figure 2.

Figure 2. Visualisation effective d'un *patch* simple, prenant en compte le nombre d'objets contenus par les *subpatchers*



Avec cette implémentation, un *patch* réellement utilisé en situation instrumentale est visualisé selon la figure 3.

Figure 3. Visualisation synoptique de l'ensemble du *patch* de *Jupiter* de Philippe Manoury



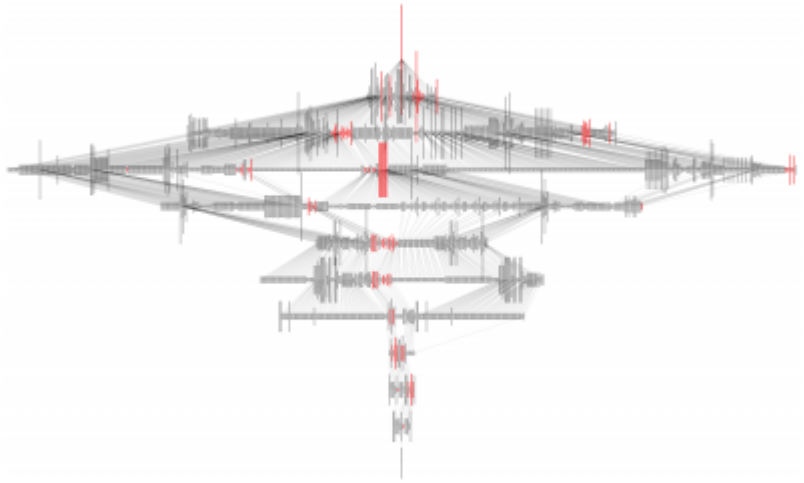
1. 3. Différenciation des types d'objets

Cette visualisation offre un aperçu synoptique de la structure d'un *patch* réel. Mais la taille des *subpatchers*, représentée ici en nombre d'objets, pose le problème d'une hiérarchisation des composants. Dans Max, des objets comme *panel* [(12)] ou *print* [(13)] n'ont en principe aucune incidence sur le comportement instrumental. Inversement, des objets comme *coll*, *detonate*, *js*, ou *qlist* peuvent eux-mêmes contenir un très grand nombre d'informations pertinentes pour l'analyse musicale. Au stade actuel d'implémentation de notre prototype, un objet *panel* et un objet *coll* ont la même incidence sur la visualisation de la structure du programme.

Dans Max, JavaScript permet de retourner pour chaque objet recensé le nom de la classe à laquelle il appartient [(14)]. Une différenciation visuelle entre *subpatchers* destinés aux opérations audionumériques et ceux ne traitant pas d'audio pouvant être utile, notre programme permet de repérer les noms d'objets se terminant par le caractère *tilde* (« ~ ») et de colorer en rouge les *subpatchers* en contenant au moins un.

Cette méthode n'est pas entièrement robuste, puisque des abstractions ou *externals* [(15)] audio peuvent ne pas avoir cette terminaison seulement conventionnelle et inversement, mais elle permet de donner une indication de la proportion des parties du programme dédiées au traitement du signal sonore avec une bonne fiabilité dans les cas d'un usage orthodoxe de Max. La figure 4 montre la visualisation colorée du *patch* de *Partita I*.

Figure 4. Visualisation colorée de l'ensemble du *patch* de *Partita I* de Philippe Manoury (en rouge, les *subpatchers* contenant au moins un objet théoriquement audio)



Comme pour la grande majorité des dispositifs programmés en Max que nous avons explorés avec ce mode de visualisation, il apparaît que la proportion de *subpatchers* effectuant de la synthèse ou du traitement de signal audio est relativement faible en regard de l'ensemble du *patch.*

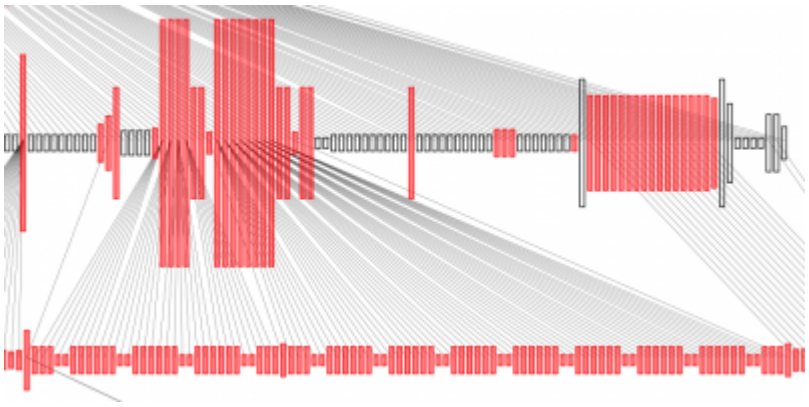
2. Applications analytiques des représentations de programmes Max

Pour conjoindre la visualisation à l'analyse du programme lui-même, notre prototype implémente, toujours en JavaScript, la possibilité pour l'utilisateur de cliquer sur un objet pour ouvrir le *subpatcher* correspondant. Les hypothèses émises par la seule observation de notre mode de représentation peuvent ainsi être vérifiées immédiatement à l'aide d'une navigabilité efficace.

2. 1. Redondances au sein d'un *patch*

Pour un même *patch* de grande envergure, la représentation synoptique permet de repérer aisément des régularités visuelles qui peuvent révéler des redondances fonctionnelles. La figure 5 montre un détail de la représentation du *patch* Max de *Neptune* (1991) [(16)] pour trois percussionnistes et électronique temps-réel de Manoury.

Figure 5. Détail du *patch* de *Neptune* de Philippe Manoury

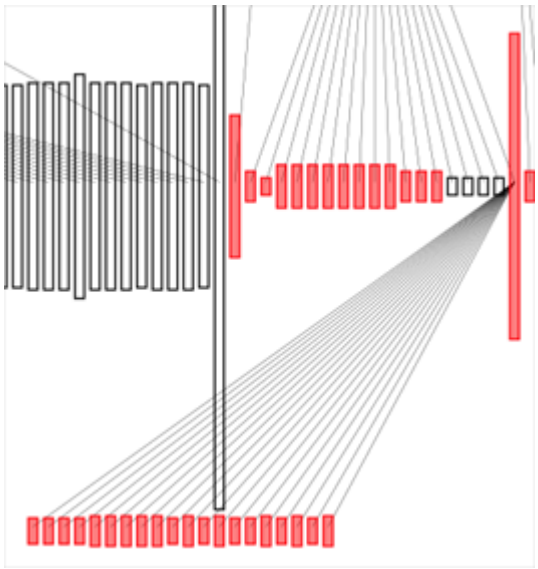


Le motif répété pratiquement à l'identique sur le rang inférieur de la figure (de gauche à droite : trois objets de petite taille, deux de très petite taille, trois objets de petite taille) correspond à la présence d'abstractions (six objets *osc1~* permettant la lecture d'une table d'onde et deux objets *tab1~* stockant cette table dans un objet *buffer~*) encapsulées dans douze abstractions de plus haut niveau, représentées par les lignes verticales de plus grande taille sur le rang supérieur, qui correspondent aux douze modules de synthèse *Phase Aligned Formants* (*PAF*) employés dans *Neptune*. Pour un *patch* de grande envergure, le repérage d'abstractions permis par cette visualisation favorise l'identification de sous-ensembles identiques et de réduire l'investigation sur les éléments réellement différenciés.

2. 2. Sous-programmes communs à différents *patches*

La possibilité de copier-coller l'objet de visualisation et de navigation dans n'importe quel programme facilite une approche comparative de *patches* différents. La figure 6 montre, au rang inférieur, un sous-ensemble d'objets audio correspondant à un module de réverbération intégré à la spatialisation du programme de *Jupiter*.

Figure 6. Détail du *patch* de *Jupiter* de Philippe Manoury



La représentation synoptique des quatre *patches* correspondant aux oeuvres du cycle *Sonus ex machina* (*Jupiter*, *Pluton*, *La partition du ciel et de l'enfer*, *Neptune*) de Manoury fait apparaître la présence de ce sous-ensemble dans les quatre oeuvres. L'observation de *patches* à partir de notre prototype de visualisation permet, comme avec cet exemple, une approche comparative de programmes distincts mais susceptibles d'appartenir à un ensemble cohérent : *patches* écrits pour un même cycle d'oeuvres, par un même compositeur, par un même réalisateur en informatique musicale, par plusieurs compositeurs ayant collaboré sur un plan technique, ou encore différentes versions d'un *patch* au service d'une même

situation.

Conclusion

Le prototype de visualisation et de navigation de *patches* Max présenté dans cet article constitue un premier niveau de développement d'outils d'investigation d'instruments numériques suffisamment généraux pour dépasser l'analyse d'une oeuvre. Son implémentation simple permet d'ores et déjà de soulever des aspects utiles pour l'étude des comportements de dispositifs auxquels le musicologue peut avoir accès, et certaines de ses limites doivent susciter une réflexion sur la possibilité de développer une hiérarchisation efficace des classes d'objets présents dans un *patch* Max. Celles de ces classes qui peuvent elles-mêmes être porteuses de structures de données informatiquement ou musicalement pertinentes devront être l'objet d'un protocole adapté à la mesure de leur possible complexité. Par ailleurs, la possibilité d'établir un filtre rejetant des objets ayant pour fonction le commentaire, la documentation et l'habillage visuel des programmes, non négligeables dans les *patches* que nous avons pu étudier et pourtant insignifiants sur le plan du strict comportement instrumental, appellent un travail d'étude du langage Max lui-même et d'établissement de catégories de classes adaptées à l'investigation musicologique. Dans un article sur le calcul, la programmation et la création, Yann Orlarey (2009) soulève six « dimensions cognitives des langages de programmation » (p. 345) : l'abstraction, la modularité, la consistance, la lisibilité, l'expressivité et la viscosité. Dans le cadre de notre recherche, une autre dimension peut apparaître, que nous pourrions nommer *auto-documentabilité* : la capacité d'un langage à renseigner sur ses propres instanciations. Cette capacité, manifeste à travers la présentation de notre prototype qui bénéficie de la possibilité de connaître les classes d'objets présentes dans un *patch* Max, a ses limites. Comme nous l'avons vu, il est par exemple impossible de certifier algorithmiquement qu'un objet traite ou non de signal audio. Il n'existe également aucune fonction JavaScript permettant de fournir les arguments d'un objet, ce qui serait très utile pour détailler les relations entre objets *send* et *receive* ou entre les *buffers* audio et leurs différents enregistreurs et lecteurs. Notre implémentation voit donc ses perspectives de développement limitées par le langage dans lequel elle s'inscrit. Il n'en demeure pas moins que les ressources offertes par Max permettent dans une large mesure de mettre en oeuvre des outils généraux d'analyse des *patches* élaborés par les acteurs de la communauté musicale. Exemplifiée par le prototype de visualisation synoptique et de navigation présenté à travers cet article, une telle approche à vocation organologique et musicologique pourrait voir sa portée évaluée par une application aux dispositifs de création programmés dans tous les langages de l'informatique musicale.

---

1. L'abstraction Max et le code JavaScript, utilisables avec Max 6, sont attachés à cet article. Le paquet est téléchargeable en annexe.

2. Comme cela peut apparaître dans ce témoignage de John Chowning sur ses premiers essais à l'*Artificial Intelligence Laboratory* de l'université Stanford : « [...] dans cet environnement de l'intelligence artificielle, se trouvaient des ingénieurs, des mathématiciens, des psychologues, des linguistes, des philosophes, toutes sortes de personnes très compétentes à qui je pouvais poser des questions à tout moment [...]. Alors j'ai profité de cet environnement exceptionnel, que j'ai utilisé comme une aubaine pour acquérir les connaissances scientifiques et techniques dont j'avais besoin. Je posais des questions à l'un, puis à l'autre ; et finalement j'ai réussi à progresser dans mes expériences sur l'espace, avec l'aide de techniciens, bien sûr. Assez rapidement, j'ai pu reproduire les mouvements gauche / droite, des cercles et même des déplacements en quatre canaux. Et, ce faisant, petit à petit, j'ai aussi appris à programmer. » (Chowning, Gayou, 2005, p. 10).

3. Sans ignorer les réserves de Claude Cadoz sur le statut instrumental des dispositifs technologiques de production sonore (voir notamment Cadoz, 1999, p. 89), nous adoptons ici une définition large de l'instrument de musique numérique, telle que nous l'avons formulée en 2008 : « Un instrument de musique numérique peut être défini de manière très générale par sa constitution ? la présence d'une interface d'accès gestuel, d'un environnement informatique et d'une interface de sortie sonore ? et sa destination : l'exécution musicale. » (Dufeu, 2008).

4. (Schaeffner, 1936), pagination non précisée par Stiegler.

5. Remarquons que des outils généraux d'analyse de programmes sont déjà développés et exploités par

certains réalisateurs en informatique musicale, comme par exemple des scripts permettant de lister les bibliothèques de fichiers nécessaires au fonctionnement d'un dispositif. Ces outils sont généralement destinés à un usage privé.

6. Notamment avec le projet TaCEM (*Technology and Creativity in Electroacoustic Music*, 2012-2015), initié en Angleterre par Michael Clarke (University of Huddersfield) et Peter Manning (Durham University), qui se donne pour objectif l'étude de huit oeuvres du répertoire électroacoustique (voir Clarke, Dufeu, Manning, 2013). La présentation du projet TaCEM est consultable à l'adresse suivante : http://www.hud.ac.uk/research/researchcentres/tacem/ (consulté le 18 octobre 2013).

7. Nous désignons par *patch* un programme réalisé à l'aide d'un environnement logiciel permettant de connecter les objets fonctionnels de manière graphique, comme Max ou Pure Data, par opposition aux programmes directement codés en texte, comme avec Csound ou SuperCollider.

8. Sous-programmes. Nous conservons l'expression anglaise dans la suite de l'article.

9. Le *patch* mentionné dans cet article est nommé Jupiter2007beta.pat, dont les commentaires intégrés mentionnent une performance à Séoul, le 11 novembre 2006. Ce document et la librairie attachée nous ont aimablement été communiqués par Alain Jacquinot, alors directeur de la production à l'Ircam, en septembre 2009.

10. Le *patch* mentionné dans cet article est nommé PARTITA2011-24.maxpat. Ce document et la librairie attachée nous ont aimablement été communiqués par Christophe Lebreton, réalisateur en informatique musicale à Grame, Centre national de création musicale de Lyon, en mai 2012.

11. En employant un objet JSUI (JavaScript User Interface) et la librairie MGraphics disponible dans Max 6.

12. L'objet *panel* a généralement une fonction seulement décorative (affichage d'un panneau coloré).

13. L'objet *print* permet d'afficher des informations dans la fenêtre Max.

14. C'est d'ailleurs ce qui permet d'établir l'analyse récursive de *patches* présentée ici : les *subpatchers* et abstractions appartiennent à la classe « patcher ».

15. Les *externals* sont des objets développés en C par des tiers et intégrés à Max (voir Lyon, 2012).

16. Le *patch* mentionné dans cet article est nommé NEPTUNE2003.msp, dont les commentaires intégrés mentionnent une performance à la Cité de la musique, le 16 juin 2003. Ce document et la librairie attachée nous ont aimablement été communiqués par Alain Jacquinot en septembre 2009.

Pour citer ce document:
Frédéric Dufeu, « Comment développer des outils généraux pour l'étude des instruments de musique numériques ? », *RFIM* [En ligne], Numéros, n° 3 - automne 2013, Mis à jour le 06/12/2013
URL: http://revues.mshparisnord.org/rfim/index.php?id=255