COMPOSITION MIXTE A BASE DE TRAITEMENTS ET CONTROLES ORIENTES OBJET

Alain Bonardi

COMPOSITION MIXTE A BASE DE TRAITEMENTS ET CONTROLES ORIENTES OBJET EXEMPLE DE MISE EN ŒUVRE DANS *PIANOTRONICS 3*

Alain Bonardi
CICM – EA 1572
Université Paris 8 et IRCAM
alain.bonardi@univ-paris8.fr

RÉSUMÉ

Dans cet article, nous présentons notre travail de recherche sur l'utilisation du paradigme objet dans la création de musique mixte, allant de la conception du traitement temps-réel à son contrôle sous la forme de classes-instruments ou encore son déploiement dans l'activité de composition, menée par généralisation/spécialisation de ces classes. Nous illustrons le propos en suivant l'exemple de la composition de *Pianotronics 3*, pièce mixte pour piano et électronique.

1. INTRODUCTION

Dans le domaine de l'informatique musicale, les recherches mobilisant le paradigme objet ont été très nombreuses, depuis les propositions de langages de programmation, les modèles de représentation et jusqu'aux stratégies compositionnelles de certains musiciens contemporains[1]. Nous en donnons quelques exemples sans tentative d'exhaustivité.

Concernant les langages orientés objet, l'un des plus connus est *SuperCollider*, développé par James Mc Cartney, dont la première version date de 1996, qui est un langage pour la synthèse temps réel et la composition algorithmique [12]. Sa syntaxe est dans l'ensemble proche de celle des structures en C.

La question des langages orientés objet en informatique musicale a évidemment rencontré celle des modèles de représentation musicale [4]. Il n'est pas possible de citer ici tous les travaux dans le domaine, mentionnons ceux de Stephen Pope [8] auteur de l'environnement de composition, performance et analyse MODE/SMOKE en SmallTalk ; de leur côté, Xavier Rodet et Pierre Cointe ont proposé l'environnement FORMES pour la composition et la programmation temporelle des processus [9], et affirment : « il est significatif que la programmation orientée objet répond à la majorité des besoins en MCS [Music composition and synthesis] »[2]. Certains chercheurs ont produit des modélisations plus spécialisées comme l'environnement MusES [7], dédié à la représentation des connaissances en musique tonale. Un environnement plus récent comme OpenMusic [1] utilise également un modèle objet et permet de prendre en compte des classes créées par l'utilisateur.

Le domaine du temps-réel s'est lui aussi penché sur les modélisations orientées objet d'abord pour la représentation de sa propre mise en œuvre, autour du contrôle temps-réel du jeu de synthétiseurs sonores. Dans sa thèse parue en 1991 [5], Lounette M. Dyer établit une modélisation objet du jeu musical temps-réel avec contrôle interactif, et dans son annexe C compare son approche à celle de Max, non-orienté objet car implémenté en C. Dans un second temps, lorsque Max et PureData se sont établis comme références dans le domaine du traitement temps-réel, plusieurs bibliothèques ont proposé des environnements orientés objet venant s'y intégrer, comme *Odot* développée au CNMAT [6].

Le paradigme objet a également intéressé les compositeurs ; l'un des plus inspirés par cette approche est Horacio Vaggione qui y trouve un support aussi bien théorique que pratique ou d'implémentation à son approche compositionnelle, en construisant des réseaux d'objets [11] [10].

Le travail que nous présentons ici s'intéresse à une conception orientée objet du traitement temps réel et de son contrôle, par une approche classe-instance. Dans la première partie de l'article, nous exposerons la conception du traitement sonore dans cette perspective, en faisant appel au langage FAUST. Dans la deuxième partie, nous montrons comment nous avons créé des classes de contrôle du traitement sonore en Javascript, aboutissant à des modèles d'instruments temps réel. Enfin, dans la troisième partie, nous montrons comment la manière de composer avec cette partie informatique est liée à la conception de cette dernière. La notion d'héritage, prise au sens large acquiert alors une place importante, le travail compositionnel se faisant parfois par généralisation de comportements locaux et parfois par affinage successif de modèles que l'on va spécialiser au fur et à mesure. Tout au long de l'exposé, nous prendrons l'exemple de *Pianotronics 3* (2016), pièce

[1] Sans compter l'utilisation des langages objet comme C++, Java, etc. pour développer des applications musicales, aspect que nous ne prenons pas en compte ici.

[2] Le texte d'origine en anglais est : "It is significant that object-oriented programming matches most of the requirements of MCS", traduction de l'auteur.

mixte pour piano[3] et électronique temps-réel composée par l'auteur, qui a servi de terrain d'expérimentation, de création et de recherche.

2. CONCEPTION CLASSE-INSTANCE DU TRAITEMENT SONORE

Le traitement temps réel est conçu à partir d'un modèle unique de transformation sonore qui pourra être instancié autant de fois que nécessaire. Nous avons choisi de l'implémenter en code FAUST, à la fois pour les qualités classiques de ce langage en termes d'explicitation des processus et de finesse du rendu sonore, mais aussi pour la facilité à mettre en œuvre une instanciation multiple d'un traitement avec des instructions comme *par* (instanciation multiple parallèle). Dans le cas de *Pianotronics 3*, nous avons choisi un module associant une ligne à retard à un *pitch-shifter* à base d'effet Doppler, combinant ainsi un traitement temporel et un traitement fréquentiel[4]. Nous nous sommes fixé à 16 instances de ce bloc.

Chaque ligne à retard est constituée de deux *delays* superposés et déphasés d'une demi-période de *phasor*, (par le jeu de 2 enveloppes *env1* et *env2*) permettant de modifier la durée du retard sans clic (figure 1).

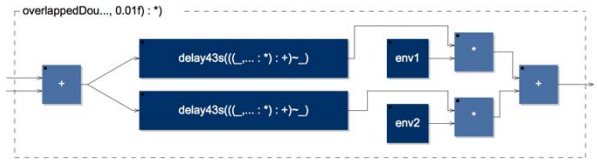

Figure 1. Bloc-diagramme issu du code FAUST de chaque ligne à retard double.

Chaque ligne à retard est envoyée ensuite vers un *pitch-shifter* (effet Doppler) pour permettre une transposition, avec une répartition possible (selon la valeur du paramètre *hvd – harmonizer versus delay*, comprise entre 0 et 1) entre son retardé transposé et son retardé direct (non transposé).

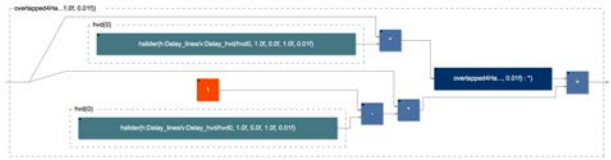

Figure 2. Bloc-diagramme issu du code FAUST de l'envoi de la ligne à retard vers le *pitch-shifter*.

[3] *Pianotronics 3* sera créée le 8 avril 2016 dans le cadre des concerts du Scrime à Bordeaux. Il s'agit d'ailleurs d'une commande de cette institution, dans le cadre de l'appel à miniatures de 2013 utilisant les technologies développées au LABRI. *Pianotronics 3* est une forme ouverte qui est pilotée par le séquenceur i-score développé au LABRI.

[4] Fréquentiel au sens musical (transposition du son avec une altération plus ou moins importante de son timbre), même si le traitement par effet Doppler est réalisé avec un retard variable croissant ou décroissant, signifiant que son implémentation est de nature temporelle.

Chaque *pitch-shifter* se compose de quatre transpositeurs élémentaires à effet Doppler, superposés avec un décalage d'un quart de période entre eux.

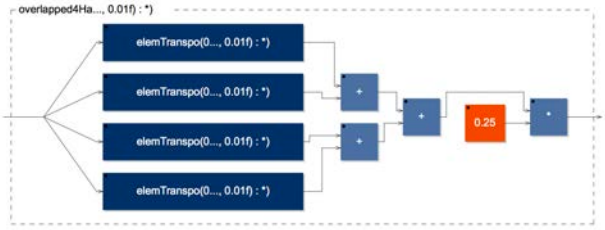

Figure 3. Bloc-diagramme issu du code FAUST de la structure du *pitch-shifter*, composé de quatre transpositeurs élémentaires déphasés.

Ce module d'ensemble {ligne à retard + *pitch-shifter*} est réitéré 16 fois dans le code FAUST. La sortie de chacun de ces modules peut être réinjectée vers l'entrée de n'importe quel autre, via une matrice 16 x 16.

L'architecture d'ensemble est ainsi donnée :

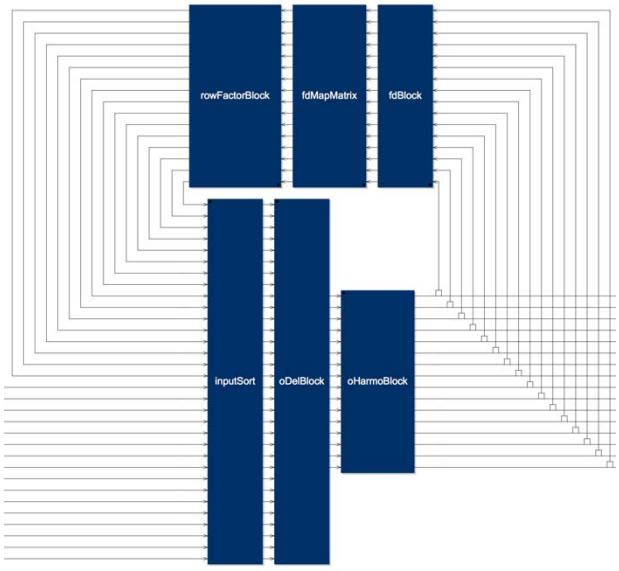

Figure 4. Bloc-diagramme issu du code FAUST du traitement d'ensemble.

Les différents blocs composant le traitement sont les suivants :

- *inputSort* : permutation des entrées pour associer l'entrée de la i*ème* ligne à la i*ème* réinjection la concernant.
- *oDelBlock* : bloc des 16 lignes à retard variable doubles (cf. figure 1).
- *oHarmoBlock* : bloc des 16 *pitch-shifters* (cf. figure 3).
- *fdBlock* : bloc de multiplicateurs des signaux en sortie utilisés pour la réinjection.
- *fdMapMatrix* : bloc permettant de dispatcher les signaux à réinjecter issus de *fdBlock* vers toutes les entrées en sommant ce qui revient vers chacune.
- *rowFactorBlock* : bloc d'atténuation de chaque somme de réinjections par entrée (qui permet d'éviter des signaux trop forts si plusieurs réinjections sont sommées sur la même entrée).

Même si FAUST est un langage fonctionnel qui n'est pas en soi orienté objet, l'idée est de disposer d'un traitement générique - en quelque sorte une « classe » {ligne à retard + *pitch-shifter*}, que l'on va instancier 16 fois avec des valeurs différentes. Les données membres de la « classe » modélisant chaque unité de traitement sont alors :

- le numéro identifiant de l'instance ;
- la durée du délai en milli-secondes ;
- le *feedback*, compris entre 0 et 0.99 ;
- l'indice de répartition entre son retardé direct et son retardé transposé (*HVD*, ie *Harmonizer versus Delay*), compris en 0 et 1 ;
- la transposition en *midicents* (de -2400 à +2400 midicents) ;
- le gain linéaire en entrée (de 0 à 4) ;
- le gain linéaire en sortie (de 0 à 4) ;

A côté du code FAUST, nous avons développé sur Max une interface de contrôle permettant de piloter de manière graphique le traitement temps-réel, reprenant les différents contrôles correspondant aux données membres évoquées ci-dessus, mais regroupées par nature d'information : toutes les durées des lignes à retard, puis tous les *feedbacks*, puis tous les indices de répartition, etc. En bas à droite se trouve la matrice de contrôle de la réinjection, indiquant sur chaque colonne i tous les modules de numéro j (sur la j^{ème} ligne) vers lesquels la réinjection se fait.

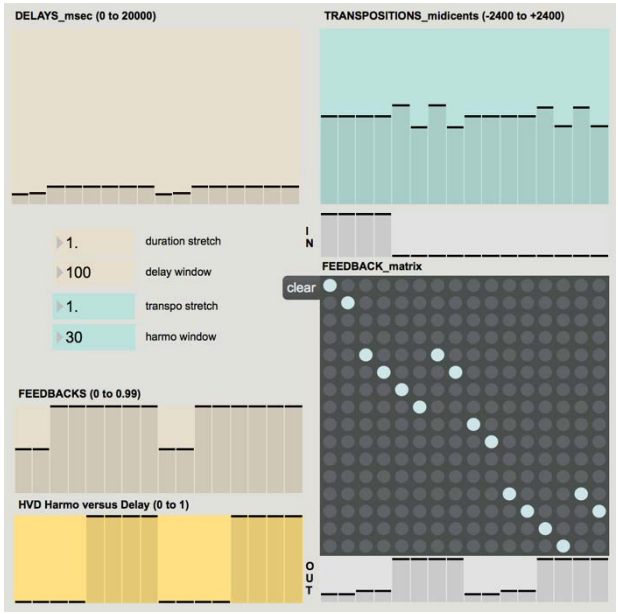

Figure 5. L'interface de contrôle du code FAUST.

L'interface permet de manipuler des données macroscopiques agissant sur les 16 instances simultanément, comme par exemple le paramètre *duration stretch*, qui permet de multiplier toutes les durées de ligne à retard par un facteur commun, ou *transpo stretch* qui agit de même sur les transpositions en *midicents*.

Cette interface prend également en charge les interpolations temporelles linéaires : les valeurs envoyées vers le module en FAUST peuvent être soit modifiées instantanément soit progressivement depuis la dernière valeur prise.

3. CONCEPTION ORIENTEE-OBJET DU CONTROLE DES TRANSFORMATIONS TEMPS-REEL

Considérant l'ensemble des informations caractérisant une unité {ligne à retard + *pitch-shifter*} comme la « classe » de base de notre traitement, nous nous intéressons maintenant à l'assemblage et l'imbrication de plusieurs de ces unités pour former des modules de traitement plus complexes, que nous nommerons ici des « instruments ». Nous avons choisi ce terme « instruments » car nous avons créé ces regroupements d'unités élémentaires en cherchant des rendus de transformation du son du piano ayant des ressemblances avec des instruments acoustiques, notamment des percussions.

Dans *Pianotronics 3*, nous avons implémenté le contrôle de l'ensemble du traitement temps réel sous la forme d'un programme Javascript. Le choix de ce langage est motivé par plusieurs raisons : facilité de mise en œuvre dans Max (objet *js*), lisibilité du code, approche orientée-objet, et possibilité ultérieure d'utiliser ce code par exemple dans un navigateur. Le code Javascript nous permet d'une part de construire les assemblages d'unités de traitement que nous avons appelés « instruments » et d'autre part de contrôler temporellement la conduite de l'électronique. Dans cet article, nous ne nous pencherons que sur le premier aspect[5].

Pour *Pianotronics 3*, nous avons créé 11 « instruments »[6] comportant chacun un certain nombre d'unités {ligne à retard + *pitch-shifter*} :

- *Bass2Octava* : 4 unités
- *HighCymbal* : 2 unités
- *Downglissando* : 3 unités
- *Timbales* : 4 unités
- *Quinteup* : 3 unités
- *Thickchord* : 10 unités
- *Quintedown* : 2 unités
- *Circularmotive* : 8 unités
- *Tremolo* : 2 unités
- *Reflectivebass* : 4 unités
- *Harmonicset* : 6 unités

Prenons le cas de l'instrument *Timbales*[7], qui comporte 4 unités dont les entrées et sorties sont associées par des réinjections. Le code du constructeur est construit ainsi :

[5] Le contrôle temporel utilise un détecteur d'attaque qui à certains moments de la partition évalue le tempo, et programme les tâches de mise à jour du contrôle de la partie électronique.

[6] Les « instruments » sont instanciés au début de chaque séquence de la pièce (qui en comporte 8) sur les 16 unités prévues dans le code FAUST. D'une séquence à l'autre, la distribution des « instruments » varie.

[7] baptisé ainsi car son rendu sonore dans le médium-grave du piano suggère le son d'une timbale.

```
var Timbales = function(index) {
	this.index = index;
	this.outLevel = 2.5;
	//
	setColumn(index, 2650, 0, 0.999, 1, 0, 0);
	setColumn(index+1, 3400, 0, 0.999, 1, 0, 0);
	setColumn(index+2, 0, -1730, 0, 1, 0, this.outLevel);
	setColumn(index+3, 0, -2100, 0, 1, 0, this.outLevel);
	//
	mFdMatColumnReset(index, 4);
	mFdMat(index, index, 1);
	mFdMat(index, index+2, 1);
	mFdMat(index+1, index+1, 1);
	mFdMat(index+1, index+3, 1);
	//display update
	updateMTapDisplays()
};
```

Les valeurs des colonnes de paramètres correspondent aux données membres de la classe de base de chaque unité et sont transmis par la fonction *setColumn* : numéro d'unité, durée du délai (milli-secondes), transposition (*midicents*), *feedback*, coefficient *HVD* (*Harmonizer versus Delay*), gain en entrée, gain en sortie. Les appels à la fonction *mFdMat* permettent de remplir la matrice de feedback : dans le code ci-dessus, la sortie de l'unité 0 renvoie du son vers les entrées 1 et 2.

Du côté des méthodes, nous avons défini plusieurs méthodes simples :

- *Timbales.prototype.on()* : son on en sortie des 4 unités de timbales.
- *Timbales.prototype.off()* : son off en sortie des 4 unités de timbales.
- *Timbales.prototype.inputOn() :* les entrées des 4 unités de timbales sont ouvertes.
- *Timbales.prototype.inputOff()* : les entrées des 4 unités de timbales sont fermées. Mais cela permet, si les sorties sont actives, de laisser entendre les réinjections entre les 4 lignes sans les alimenter par de nouveaux sons en entrée.

4. CONSEQUENCES DE L'APPROCHE ORIENTEE OBJET SUR LA COMPOSITION MIXTE

Ayant posé le cadre orienté objet de notre démarche au niveau de l'élaboration du traitement temps réel et de son contrôle, nous montrons maintenant les conséquences de cette approche dans la composition de musique mixte, autour de l'exemple de *Pianotronics 3*, sur trois niveaux : au niveau macroscopique de la construction d'ensemble du traitement par séquence musicale, au niveau de l'écriture du contrôle, et au niveau méthodologique.

Au niveau macroscopique, les 8 séquences de cette pièce sont définies en termes de traitements sonores par de simples instanciations des classes correspondant aux instruments choisis, l'ensemble donnant l'état et donc la nature musicale du patch dans son ensemble à un moment particulier. Donnons un exemple avec la séquence 0, qui est définie par les instanciations suivantes :

```
function seq0() {
	allReset();
	bass2oct0 = new Bass2octava(0);
	hcymbal0 = new Highcymbal(4);
	dgliss0 = new Downglissando(6);
	timb0 = new Timbales(9);
	…
}
```

Les arguments des constructeurs des différents instruments sont les index de début pour la première ligne de chaque instrument : ainsi l'instance *bass2oct0* de la classe *Bass2octava* (transposition à la double octave inférieure) va occuper 4 lignes en commençant à 0 (0, 1, 2, 3), alors que l'instance *hcymbal0* de la classe *Highcymbal* (son de cymbale dans le suraigu) va en occuper 2 en commençant à 4 (4, 5). Les 16 lignes du dispositif sont ainsi affectées, formant l'ensemble des traitements temps réel opérant dans une séquence.

Cette manière d'aborder le patch est très différente de celle de certains compositeurs, comme par exemple Philippe Manoury, qui conçoivent le patch comme un « orchestre » de transformations [3] comportant des modules spécialisés distincts (*harmonizers*, *samplers*, *phase-aligned formants*, etc.) dirigés par une partition le plus souvent sous la forme de listes de commandes attribuant des valeurs à des paramètres. Au contraire, notre démarche part d'un seul module générique, qui par le jeu des réglages globaux, ou locaux, et surtout par l'association mobile des traitements élémentaires sous la forme d'instruments permet des rendus sonores très variés. De plus, l'instanciation dynamique des classes modifie très rapidement et en quelques lignes de code la palette des effets affectés aux 16 unités.

Du côté de l'écriture du contrôle, le passage au paradigme objet permet de s'affranchir des listes statiques de valeurs du type *cue-list* ou *pattrstorage*, en confiant cette tâche à des fonctions en Javascript, rendant immédiatement ces commandes dynamiques et calculables, et pouvant faire appel à des structures de données. Nous avons choisi d'associer une fonction à chaque événement dans la partition mixte. Ainsi, à l'événement 10 (dans la séquence 0), deux[8] actions sont exécutées par la fonction *state10*, la première ouvrant les entrées du transpositeur à la quinte supérieure *quintu0*, la seconde fermant celles de la cymbale aigue *hcymbal0*.

```
function state10()
{
		quintu0.inputOn();
		hcymbal0.inputOff();
}
```

Enfin, au niveau méthodologique, cette approche nous a conduit en permanence à travailler soit en

[8] En fait, l'événement 10 comporte une autre action de programmation temporelle de l'événement suivant (11) : *nextStateBeatScheduled(8.2)* qui permet de le planifier dans 8,2 temps. Comme nous l'avons dit à la note 5, nous n'entrons pas dans le détail du contrôle temporel dans cet article.

généralisation, soit en spécialisation. Nous avons par exemple rassemblé sous une même classe plus générale deux instruments auparavant proches mais modélisés dans deux classes différentes, en gérant leurs spécificités individuelles par l'ajout de méthodes. Mais nous avons aussi avancé dans la composition par spécialisation, utilisant l'extrême souplesse du paradigme objet en Javascript. Par exemple, comme les séquences constituent des variations les unes des autres – la séquence 3 est une variation de la séquence 0, la 4 de la 1, etc., nous avons adapté les classes-instruments conçues pour les séquences 0, 1 et 2 en leur ajoutant des données membres ou des méthodes pour supporter les séquences 3, 4 et 5. Il s'agit d'une forme très souple (et presque implicite) d'héritage au sens objet, et qui correspond bien au mode de travail compositionnel entrepris, qui part du dispositif de traitement pour aller vers l'écriture instrumentale. Le langage Javascript, complètement intégré dans Max, sans compilation, s'avère très pratique pour une approche par prototypage et affinage itératif.

Citons enfin une dernière conséquence de cette approche dans son ensemble : avec un code FAUST générique, un programme Javascript organisant des instruments-classes et leur déploiement dans la pièce, nous rendons explicites à la fois les traitements temps réel [2] et leur organisation, facilitant l'archivage pérenne de l'ensemble du code de la pièce, sa transmission et son portage sur de futures plateformes. Ajoutons que le patch Max de *Pianotronics 3*, intégrant les traitements FAUST dans un objet généré *.mxo* et le contrôle dans un programme Javascript inséré dans un objet *js* est extrêmement simple à comprendre et mettre en œuvre.

5. CONCLUSION

Nous avons montré dans cet article comment nous avons organisé les différents composants d'une composition de musique mixte autour du paradigme objet. Les choix retenus : traitement temps réel générique déclinable à volonté, création d'instruments-classes agrégeant les instances par bloc et permettant leur contrôle, conception de la pièce sous la forme de variations se sont révélés pertinents et cohérents, permettant de mettre en œuvre le paradigme objet pleinement, à la fois comme mode de conception, comme mode d'implémentation et comme support de l'activité compositionnelle. Nous réfléchissons maintenant, pour une nouvelle composition, à la création de méta-classes d'instruments, associant plusieurs d'entre eux, et permettant des envois de sons des uns vers les autres, ce qui n'est pas possible dans l'architecture actuelle.

Nous souhaitons remercier Myriam Desainte-Catherine, György Kurtag Jr. et Pierre Cochard au Labri/Scrime à Bordeaux sans qui le travail de création de *Pianotronics 3* n'aurait pu avoir lieu.

6. REFERENCES

[1] Agon, C., Assayag, G., Delerue, O., Rueda, C., "Objects, Time and Constraints", *Actes de la conférence ICMC 1998*, Ann Arbor, USA, 1998.

[2] Bonardi, A. « Approches pratiques de la préservation/virtualisation des œuvres interactives mixtes : *En Echo* de Manoury », *Actes des Journées d'informatique musicale 2011*, Université Jean Monnet, Saint-Etienne, 2011.

[3] Bonardi, A., « Analyser l'orchestre numérique interactif dans les œuvres de Manoury ». *Actes du Colloque Analyser la musique mixte* (Sfam/Ircam, 2012), à paraître chez Delatour.

[4] Dannenberg, R., "Music Representation Issues, Techniques, and Systems", *Computer Music Journal*, 17(3), 1993, pages 20-30.

[5] Dyer, L., *An object-oriented real-time simulation of music performance using interactive control*, thèse de doctorat, Stanford University, 1991.

[6] Freed, A., MacCallum, J., Schmeder, A., "Dynamic, instance-based, object-oriented programming in Max/MSP using open sound control message delegation". In *Proceedings of the ICMC 2011 Conference*, Huddersfield, Angleterre, 2011.

[7] Pachet, F., Ramalho, G., Carrive, J., Cornic, G., Representing temporal musical objects and reasoning in the MusES system. *Journal of New Music Research*, 1996, (25) 3, pp. 252-275.

[8] Pope, S. T. Introduction to MODE: The Musical Object Development Environment. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*. In S. T. Pope, MIT Press, Boston, 1991, pages 83-106.

[9] Rodet, X., P. Cointe. FORMES: Composition and Scheduling of Processes. *Computer Music Journal* 8(3):32-50, Fall, 1984.

[10] Svidzinski, J., Bonardi, A., « Vers une théorie de la composition musicale numérique fondée sur des réseaux d'objets ». Dans *Actes des Journées d'Informatique Musicale 2015*, Montréal, 2015.

[11] Vaggione, H., « Représentations musicales numériques : temporalités, objets, contextes ». In *Manières de faire des sons*, L'Harmattan, Paris, 2010.

[12] Wilson, S., Cottle, D., Collins, N., *The SuperCollider Book*. The MIT Press, 2011.