

Interaction temps-réel/temps différé

Élaboration d'un modèle formel de Max
et implémentation d'une bibliothèque OSC
pour OpenMusic

Carl Seleborg

Mémoire de stage de DEA ATIAM
Année 2003–2004
Juin 2004

Université Aix-Marseille II

Sous la direction de M. Carlos Agon
Équipe des Représentations Musicales
Ircam — Centre Georges Pompidou

Interaction temps-réel/temps différé
Elaboration d'un modèle formel de Max et implémentation
d'une bibliothèque OSC pour OpenMusic

Stage de DEA ATIAM
Faculté des sciences de Luminy
Université de la Méditerranée — Aix-Marseille II

Sous la direction de M. Carlos Agon
Equipe des Représentations Musicales
Ircam — Centre Georges Pompidou, Paris

Copyright © 2004, Carl Seleborg

54 pages.

Ce document a été mis en page avec \LaTeX . Aucun animal n'a
été maltraité, blessé ou tué durant la rédaction de ce document.

Remerciements

Mes remerciements vont à Carlos Agon pour avoir su me guider à travers un sujet parfois difficile à appréhender et pour m'avoir appris énormément de choses ; à Georges Bloch, qui nous a proposé un exemple musical pour concrétiser nos idées et nos réalisations techniques ; à Gérard Assayag pour m'avoir très bien accueilli dans son équipe et aussi pour avoir mis à ma disposition son tourne-disques ; à Philippe Manoury pour l'entretien qu'il m'a accordé ; aux enseignants du DEA ATIAM pour leurs cours si riches et si intéressants ; à Jean-Brice Godet et à Cyril Laurier qui ont su insuffler une excellente ambiance au bureau A107 ; aux autres élèves de la promotion pour la très sympathique année que j'ai passée en leur compagnie ; à Cyrille Defaye pour avoir bouleversé ma vision des secrétariats de formations universitaires ; et enfin à tout le personnel de l'Ircam qui depuis plus d'un an maintenant m'a toujours très gentiment et très agréablement accueilli au sein de ses locaux.

Table des matières

Introduction	2
Problématique	2
Proposition	3
Méthode	3
Structure du document	4
1 Interaction temps-réel/temps différé	5
1.1 Motivations et problèmes	5
1.1.1 L'importance de l'interaction temps-réel/temps différé . .	5
1.1.2 Problèmes	6
1.2 Présentation d'OpenMusic	7
1.3 Présentation de Max	9
2 Une étude théorique et formelle de Max	13
2.1 Langage de programmation	14
2.2 Le patch Max : un système réactif	15
2.3 Introduction aux CSP	18
2.3.1 Processus et événements	18
2.3.2 Traces	20
2.3.3 Communications et canaux	20
2.4 Syntaxe du modèle formel de Max	20
2.4.1 Objets réactifs et connexions	20
2.4.2 Données	23
2.5 Sémantique du modèle formel	24
2.5.1 Objets réactifs	24
2.5.2 Communication entre objets réactifs	25
2.6 Abstractions	25
2.7 Temps et ordonnancement	26
2.8 Grandes familles d'objets	27
2.8.1 Objets conventionnels	27
2.8.2 Opérateurs	28
2.9 Primitives	28
2.9.1 Objets conventionnels	29
2.9.2 Opérateurs	30
2.9.3 La primitive [lisp]	31
3 La bibliothèque Moscou	34
3.1 Présentation d'OpenSound Control	34
3.2 Moscou : OSC dans OpenMusic	36
4 Un exemple musical avec Georges Bloch	41
4.1 Principe	41
4.2 Implémentation	43
Conclusion	50
Travail réalisé	50
Perspectives d'avenir	50

Table des figures

1	Exemple d'un patch OpenMusic du compositeur Tristan Murail.	8
2	L'exemple le plus connu de patch Max : un synthétiseur à modulation de fréquence.	10
3	Un exemple de synthèse vocale avec jMax.	11
4	Un exemple de patch Pure Data.	12
5	Implémentation d'un algorithme de tri à bulles sous Max.	16
6	Notation graphique d'un objet réactif.	21
7	Notation graphique d'une connexion.	22
8	Notation graphique d'un patch.	22
9	Notation graphique d'une primitive.	22
10	Deux patches en parallèle.	23
11	Deux patches en série.	23
12	Patch rebouclé sur lui-même.	23
13	Construction de l'abstraction myAbstraction.	26
14	Exemple partiel d'espace de noms pour un synthétiseur virtuel.	35
15	Envoi du message /synth/voices/1/osc 440.	37
16	Réception d'un <i>bundle</i>	37
17	Partie Max du dispositif <i>Fast Feedback</i> .	39
18	Partie OpenMusic du dispositif <i>Fast Feedback</i> .	40
19	Patch Max principal et mécanisme de détection de respiration.	45
20	Patch Max en charge de la détermination de la texture estrada.	46
21	Patches Max en charge de la détermination de la fondamentale Hindemith.	47
22	Patch OpenMusic du dispositif de Georges Bloch.	48
23	Patch Max d'affichage des informations de texture et de fondamentale.	49
24	Schéma du "tunnel du temps" affiché à l'utilisateur pendant qu'il joue.	49

Introduction

Problématique

Une dichotomie historique. . . . Dans l'histoire de l'informatique musicale et du développement des outils informatiques, les choix d'implémentation importants ont été conditionnés à la fois par les possibilités techniques des ordinateurs et équipements de l'époque et par les idées proposées par les développeurs ou par les utilisateurs. Nous pourrions, pour illustrer ce propos, citer Miller Puckette, le créateur de Max, dans [37] : « *Max took its modern shape during a heated, excited period of interaction and ferment among a small group of researchers, composers, and performers at IRCAM during the period 1985–1990; writing Max would probably not have been possible in less stimulating and demanding surroundings. In the same way, Pd's development a decade later would not have been possible without the participation of the artists and other researchers of the Global Visual Music Project, of which I was a part. My work on Max and its implementations has been in essence an attempt to capture these encounters in software, and a study of Max will succeed best if it considers the design issues and the artistic issues together.* »

Dans ce contexte, une dichotomie est née entre les outils temps-réel et les outils de composition assistée par ordinateurs, qui ressemble à la dichotomie entre compositeur et chef d'orchestre. La voix naturelle fut d'abord de se dire que ces deux types d'outils ne vivent pas dans le même monde temporel : la composition se fait dans son propre temps, le temps du compositeur n'est pas celui de son œuvre. Peut-être écrit-il le début, puis le milieu, puis la fin, puis revient sur le début, puis retouche la fin, puis le milieu, et ainsi de suite ? Il peut prendre des semaines pour écrire une simple mesure. Le monde du temps-réel, lui, n'est pas si indulgent : le temps du musicien doit être au plus aussi long que celui de la musique. Pas question de revenir sur ce qui a été joué : ça n'existe déjà plus que dans la mémoire de l'auditeur.

L'autre idée qui s'imposa naturellement fut celle de la nature du travail de composition. Traditionnellement, la composition est un travail sur des structures musicales, devant aboutir à la création d'une partition. On manipule notes, accords, mélodies, harmonies, orchestres et solistes, et ces manipulations voient leurs résultats se figer sur la portée finale. De l'autre côté, le travail du chef d'orchestre est de diriger l'exécution d'une œuvre. La baguette a remplacé la plume et les structures sont déjà figées. Il faut maintenant manipuler les joueurs, les *tempi*, les cadences et tous les autres détails de l'exécution.

Ainsi, le développement des outils a dans la plupart des cas naturellement suivi celui des concepts. Les outils de composition assistée par ordinateurs [6, 43, 33, 15] se sont tournés vers le calcul symbolique et ont peu à peu abouti à l'utilisation du langage Lisp [30], dont c'est le domaine d'excellence. De leur côté, les outils de temps-réel se sont développés autour de mécanismes d'ordonnancement temporel et ont abouti à la création d'un paradigme à part, sur lequel nous reviendrons plus tard dans ce document.

. . . rattrapée par la technique et les idées. Aujourd'hui, la donne change. Grisée par la fulgurante progression technique de ces dernières années, l'imagination a cassé certaines frontières qu'on pensait naturelles hier.

C'est ainsi qu'est arrivée l'idée de marier temps-réel et calcul symbolique.

Si il y a quelques années, le calcul symbolique était trop lent pour suivre le rythme imperturbable du temps-réel, et si ce dernier n'avait été utilisé que pour contrôler des appareils MIDI [31] ou à la rigueur, des paramètres de synthèse sonore [27], on commence à trouver de nouvelles idées pour lesquelles il serait intéressant de faire collaborer les deux mondes. Aujourd'hui, ce ne sont plus les performances des ordinateurs qui sont un frein au développement de ces idées, mais bien l'héritage historique. Aucun utilisateur des outils des deux mondes ne souhaite s'en séparer : il faut plutôt les faire travailler ensemble.

Mais comment faire pour intégrer des systèmes temps-réel dans le monde du calcul symbolique, ce pour quoi ils n'ont pas été conçus, et comment faire pour surmonter l'incapacité des systèmes de calcul symboliques à satisfaire des contraintes élémentaires de temps-réel ?

Proposition

Afin de respecter un environnement de travail déjà existant, et aussi par souci de pragmatisme (développer un nouveau système qui soit à la fois temps-réel et capable de calcul symbolique, et qui soit aussi riche en extensions que ceux qui existent déjà prendrait, au bas mot, quelques années), nous proposons de résoudre ce problème, ou du moins d'apporter quelques premiers éléments de réponse, en travaillant sur une *stratégie de communication* entre les deux mondes.

Nous allons ainsi tenter de jeter les bonnes passerelles entre les deux mondes, entre Max et OpenMusic (chacun étant l'un des grands représentants de sa catégorie), afin que chacun continue de faire ce pour quoi il a été fait, mais que les deux systèmes communiquent entre eux, créant ainsi un tout plus puissant que la somme de ses parties.

Nous fonderons notre travail sur l'élaboration préalable d'un formalisme des systèmes comme Max, jMax ou Pure Data. Par ce travail qui, à notre connaissance n'a jamais été fait auparavant, nous espérons doter la communauté des utilisateurs de Max d'un vocabulaire mieux partagé et mieux défini, et d'un cadre sémantique et syntaxique sur lequel les prochains travaux de recherche pourront se baser (ou qu'ils pourront rejeter, si il s'avérait qu'ils n'étaient pas utiles, au profit de propositions meilleures). Nous espérons que suite à nos travaux, les gens, en parlant de Max, emploieront les termes "objets réactifs", "objets conventionnels", "opérateurs", "entrée réactive", etc. (tous ces termes sont introduits dans le § 2, p. 13).

Méthode

La nature très différente de Max et d'OpenMusic nous a amené à choisir une approche théorique et formelle de ce problème. En effet, si on sait bien sur quels principes théoriques repose OpenMusic (§ 1.2, p. 7), on ne connaît pas aussi bien la nature exacte des systèmes tels que Max (§ 1.3, p. 9), et nous échappe ainsi la clé de leur lien avec le temps. Nous nous sommes donc attachés à donner à Max et à ses cousins un formalisme, ce qui représente la plus grosse partie de ce document.

Partant de ce formalisme, nous y avons inclus un mécanisme d'exécution de code Lisp, symbolisant ainsi la collaboration avec OpenMusic. Vint ensuite une réalisation pratique de cette collaboration. Le CNMAT, à l'université de

Berkeley, à développé des objets pour Max permettant de communiquer via le protocole *Open Sound Control* (§ 3.1, p. 34) ; nous avons développé une bibliothèque similaire pour OpenMusic, baptisée *Moscou*.

Nous nous sommes ainsi donné la théorie et la pratique pour faire collaborer Max et OpenMusic.

Structure du document

Ce document est structuré en quatre grandes parties, outre la présente introduction et la conclusion. Premièrement, nous présentons le sujet de l'interaction temps-réel/temps différé, avec ce qu'il comporte de problématiques, de projets et de tâtonnements, et nous présentons au lecteur non averti les logiciels OpenMusic et Max. Puis, la seconde partie de ce document expose notre petite étude théorique de Max et le formalisme que nous en proposons. La troisième partie est consacrée à la bibliothèque Moscou et à son implémentation sous OpenMusic. Enfin, la quatrième partie détaille la réalisation d'un exemple musical, proposé par et mis au point avec l'aide de Georges Bloch, illustrant l'utilisation des idées et outils exposés dans ce document.

1 Interaction temps-réel/temps différé

Ainsi que nous l'avons vu en introduction, les deux mondes du temps-réel et du temps différé ont évolué parallèlement l'un à l'autre, sans concurrence et sans interférence, sauf à quelques exceptions près. Et bien qu'il y ait eu des tentatives d'implémentation de systèmes (comme par exemple Varèse [21]), les possibilités formelles et théoriques de combiner les deux activités du calcul symbolique et de l'interaction temps-réel en un seul système n'avaient pas été explorées plus avant.

Comment la technique et les idées peuvent-elles guider de tels travaux ? Comment les principes peuvent-ils ressortir des besoins ?

Dans cette partie, nous exposerons les idées et solutions de plusieurs musiciens et chercheurs, dont, entre autres, Gérard Assayag, Philippe Manoury et Georges Bloch. Nous présenterons également les principaux outils (issus du monde de la recherche en informatique musicale) destinés au temps-réel et à la composition assistée par ordinateur, à savoir Max et OpenMusic respectivement — si le lecteur connaît déjà ces logiciels, il n'apprendra rien dans ces deux derniers paragraphes et pourra passer directement à la suite du document.

1.1 Motivations et problèmes

1.1.1 L'importance de l'interaction temps-réel/temps différé

Il y a quelques années, la dichotomie dont nous avons déjà amplement parlé était très nette : ou bien on *contrôlait* une musique pendant son déroulement, auquel cas on faisait du temps-réel, ou bien on composait de la musique (que ce soit par un humain qui use d'algorithmes et de processus informatiques pour composer ou que ce soit même un algorithme ou un processus informatique qui compose une musique validée ensuite par un humain) et c'était du domaine du temps différé.

Mais avec la maturité croissante à la fois des idées, des pratiques et de la technique, on a vu naître une intrication entre ces deux domaines : on s'est mis à vouloir utiliser les techniques du temps différé dans un contexte de temps-réel. Partant de ce contexte, une nouvelle décomposition a vu le jour : analyser, représenter ou générer la musique en temps-réel.

Analyse musicale. L'analyse musicale poussée requiert des capacités d'abstractions élevées de la part du système utilisé, comme les montre les travaux de ce domaine, qui font appel au système OpenMusic.

A notre connaissance, très peu de projets d'analyse musicale en temps réel ont été réalisés ; il est vrai qu'on n'en perçoit pas immédiatement l'intérêt. Aussi nous permettons-nous d'émettre une possible idée d'application, dans le domaine de l'improvisation ; on pourrait imaginer un outil qui, analysant à la volée ce que joue un musicien improvisant, suggère à ce dernier de changer de tonalité ou de réintégrer un motif mélodique sur lequel il était parti mais qu'il aura peut-être perdu en cours de route. En somme, une sorte d'outil pédagogique pour enrichir ou mieux structurer des improvisations.

Représentation musicale. La représentation de la musique est un domaine de recherche à part entière (comme en témoigne le nom de notre équipe d'accueil,

cf. [7]). La visualisation d'une partition de musique à partir de données musicales "objectives" comme les informations MIDI par exemple relève d'un ensemble de problèmes qui ont trait à la représentation symbolique. [28, 18]

Le § 4, p. 41 expose un exemple d'application où OpenMusic est utilisé pour générer pendant une improvisation une représentation symbolique de ce qui a été joué.

Philippe Manoury, pour la pièce $\Omega\nu\ \Omega\nu\rho\omega\nu$, en préparation au moment où nous écrivons ce document, utilise dans Max un ensemble de métronomes réglés sur des mesures différentes, et chacun associé à une hauteur de note. Chaque métronome déclenche ainsi une note particulière. De ce processus, très clairement déterministe, M. Manoury souhaitait créer une partition ; mais limité par les faibles capacités symboliques de Max, il dut transcrire les notes ainsi générées dans un fichier MIDI et rouvrir celui-ci dans OpenMusic pour quantifier les notes et en faire une portée. Clairement, une meilleure interaction entre les deux lui aurait été utile.

Génération automatique de musique. Les problèmes de génération automatique de musique (on parle aussi de composition algorithmique ou d'auto-composition [14, 13]) sont nombreux et titillent les esprits des savants et des musiciens depuis déjà de nombreux siècles [29].

De nombreuses propositions ont été faites, et on distingue ainsi trois grandes catégories d'approches : les méthodes stochastiques, utilisées par I. Xenakis [42], les méthodes d'intelligence artificielles et les méthodes par règles (ou grammaires de substitution).

Mis à part la composition stochastique, qui ne requiert que la possibilité de générer des valeurs aléatoires, la composition algorithmique fait appel à des structures de données symboliques et musicales de haut niveau, voir même à des réseaux de neurones. Ces processus nécessitent donc comme support un système d'un haut niveau d'abstraction de données.

Dans la famille des méthodes par règles, on retrouve par exemple les grammaires de substitution de Jazz pour la génération de grilles harmoniques durant une improvisation (projet OMax), de Marc Chemillier et Gérard Assayag [5], se basant sur les travaux de Mark Steedman [38].

1.1.2 Problèmes

Dans la réalisation de tels expérimentations, les problèmes apparaissent dans la mesure où l'on pousse les outils existants comme Max et OpenMusic dans des directions dans lesquelles il n'était pas prévu qu'ils soient emmenés : le temps-réel pour OpenMusic, et le calcul symbolique pour Max.

M. Manoury déplore par exemple l'absence de notion de temps musical dans Max : on y trouve des objets capables de générer des signaux à des intervalles de temps très précis, donnés en millisecondes, mais exprimer la durée d'un processus en une unité de temps relative devient plus compliqué. Il suggère qu'on y ajoute des structures temporelles adaptées au temps musical.

OpenMusic est tout à fait capable de gérer des structures symboliques requises pour ces calculs, mais ne saurait garantir une quelconque efficacité en terme de vitesse de calcul, si bien qu'il n'offre aucune garantie quelconque de temps-réel.

On sera donc tenté par deux approches : la première consisterait à créer un système depuis le départ prévu pour le calcul symbolique et pour le temps-réel. Techniquement, ce serait la solution la plus propre, mais on perdrait alors le capital important que représente les milliers de travaux déjà réalisés sur les outils existants, avec les extensions, les bibliothèques et les expériences qui vont avec. La seconde solution verrait naître la collaboration entre les systèmes existants, chacun mettant à disposition de l'autre ses points forts.

On se retrouve alors confronté à un problème difficile : si donner à Max, par un moyen quelconque des capacités de calcul symbolique (ou du moins proposer aux utilisateurs de Max une *interface* vers un tel calcul, ce qui sera notre approche un peu plus tard dans ce document)¹, ne représente pas de réelle difficulté technique, faire en sorte qu'OpenMusic respecte des contraintes de temps-réel est un tout autre labeur. Nous verrons plus en détail dans le § 4, p. 41, les problèmes soulevés par cet obstacle.

1.2 Présentation d'OpenMusic

OpenMusic [6, 1] est un environnement de programmation visuel pour compositeurs basé sur Macintosh Common Lisp [39]. En successeur de *PatchWork* [25], il étend le paradigme visuel de son ancêtre avec la programmation orientée objet et un contrôle de haut niveau sur le matériau musical symbolique. La représentation symbolique de la musique est le domaine privilégié d'OpenMusic : bien qu'il s'agisse d'un environnement de programmation parfaitement général, il est fourni avec un ensemble de classes et de fonctions spécialement développées pour la manipulation d'informations musicales symboliques.

Visuellement, OpenMusic représente les objets et les fonctions par des icônes, que l'utilisateur dispose sur une zone de travail (appelée *patcher*) et qu'il interconnecte par des fils. L'ensemble ainsi constitué s'appelle un *patch* ; par ces patches, l'utilisateur peut créer ou modifier des classes, programmer des fonctions (par le biais du mécanisme d'abstraction qui permet d'utiliser un patch comme n'importe quel autre objet ou fonction), ou simplement expérimenter avec les objets existants pour construire, étape par étape, son programme. La figure 1 montre un exemple de patch OpenMusic.

OpenMusic intègre de nombreuses bibliothèques, qui elles-mêmes fournissent de nombreuses classes et fonctions liées à la création musicale. Ainsi retrouve-t-on des classes `note`, `rest`, `chord`, `voice`, `sound`, etc., des fonctions liées à la création de partitions comme `finale-export` pour l'exportation de partitions vers *Finale* ou `omquantify` pour la quantification. On retrouve encore des bibliothèques de résolutions de contraintes, des bibliothèques d'interface vers des logiciels de synthèse sonore comme CSound ou AudioSculpt et même bientôt, une bibliothèque pour la communication par OSC avec d'autres logiciels (cf. § 3.2, p. 36).

¹Par souci d'équité, nous signalons que jMax (cf. § 1.3, p. 9) a intégré dans ses données des objets implémentant des structures de plus haut niveau que la simple liste. Ces objets sont accessibles depuis peu dans Max grâce à la bibliothèque FTM, voir <http://http://www.ircam.fr/equipes/temps-reel/maxmsp/ftm.html> et [17].

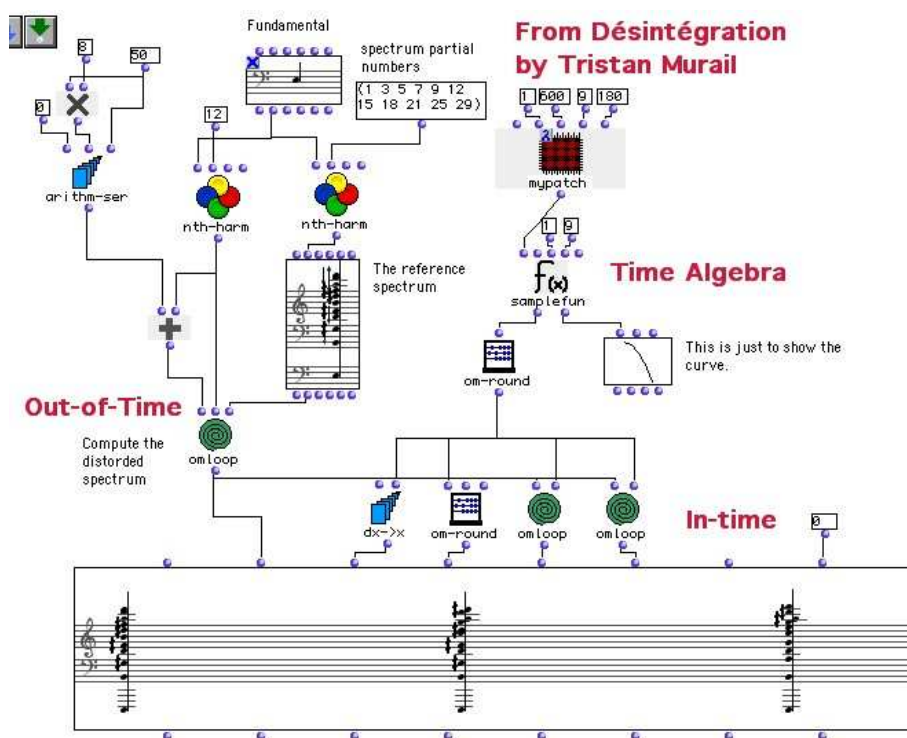


FIG. 1 – Exemple d'un patch OpenMusic du compositeur Tristan Murail.

1.3 Présentation de Max

Max est un logiciel créé par Miller Puckette à l'Ircam dans les années 1980 et aujourd'hui toujours développé par la société américaine *Cycling'74*.

Nous allons voir plus loin (page 13) qu'il est difficile de dire exactement ce qu'est Max, et nous donnerons ici, comme tout le monde l'a fait auparavant, notre propre interprétation, que nous laisserons volontairement informelle, afin de ne pas empiéter sur le discours de la partie consacrée, justement, à ce problème (le § 2).

Or donc, Max est un environnement graphique destiné au temps-réel dans lequel l'utilisateur assemble entre elles des boîtes qui s'envoient mutuellement des données, chaque boîte représentant un traitement particulier à effectuer sur ou avec ces données. Ces dernières sont principalement de nature musicale, ou du moins destinées à contrôler un processus musical ; elles proviennent le plus souvent d'entrées MIDI du système ou d'éléments graphiques que l'utilisateur peut manipuler.

Ceci ne décrit que l'aspect "contrôle" de Max. Plusieurs extensions ont été ajoutées, de manières à ouvrir Max au traitement d'autres informations : MSP² est l'extension qui intègre le traitement audio dans Max, Jitter permet de manipuler des images et permet de faire des spectacles visuels en temps-réel. Aujourd'hui, on parle le plus souvent de Max/MSP, et Jitter est souvent livré avec.

Comme pour OpenMusic, la "feuille de travail" sur laquelle on assemble les boîtes dans Max s'appelle un *patch*. Max est muni d'un mécanisme d'abstraction, certes un peu limité mais suffisant dans la plupart des cas, qui permet de réutiliser des patches dans d'autres patches.

Max n'est en réalité qu'une implémentation d'un paradigme particulier, qui a vu deux autres implémentations naître quelques années plus tard, chacune avec ses forces et ses faiblesses. Ainsi, jMax (figure 3) fut présenté en 1999 l'équipe temps-réel de l'Ircam, dirigée par François Déchelle [16], insistant sur une séparation entre l'interface graphique, écrite en Java, et le moteur de calcul, FTS, écrit en C, ce dernier intégrant par ailleurs tout un système objet (parmi ces objets, on retrouve certaines structures de données comme les vecteurs ou les dictionnaires) [17]. Quelques années plus tôt, c'était Miller Puckette lui-même qui présentait une autre implémentation de son système, nommée Pure Data ou PD [36] (figure 4), qui devait entre autres s'attaquer aux faiblesses de Max dans le domaine des structures de données.

²Miller Puckette avait nommé son logiciel en hommage à Max Mathews, celui que beaucoup considèrent comme le père de l'informatique musicale. On est amusé de constater que David Zicarelli a choisi de nommer son extension "Max Signal Processing", dont les initiales sont les mêmes que celles de Miller S. Puckette.

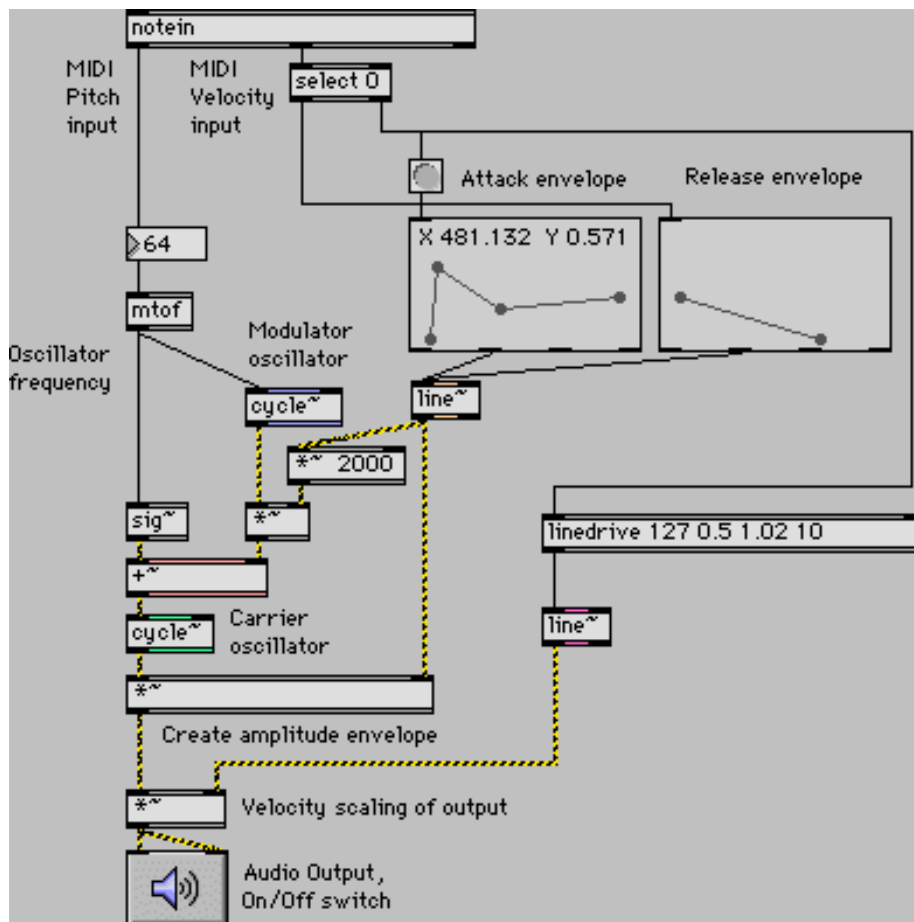


FIG. 2 – L'exemple le plus connu de patch Max : un synthétiseur à modulation de fréquence.

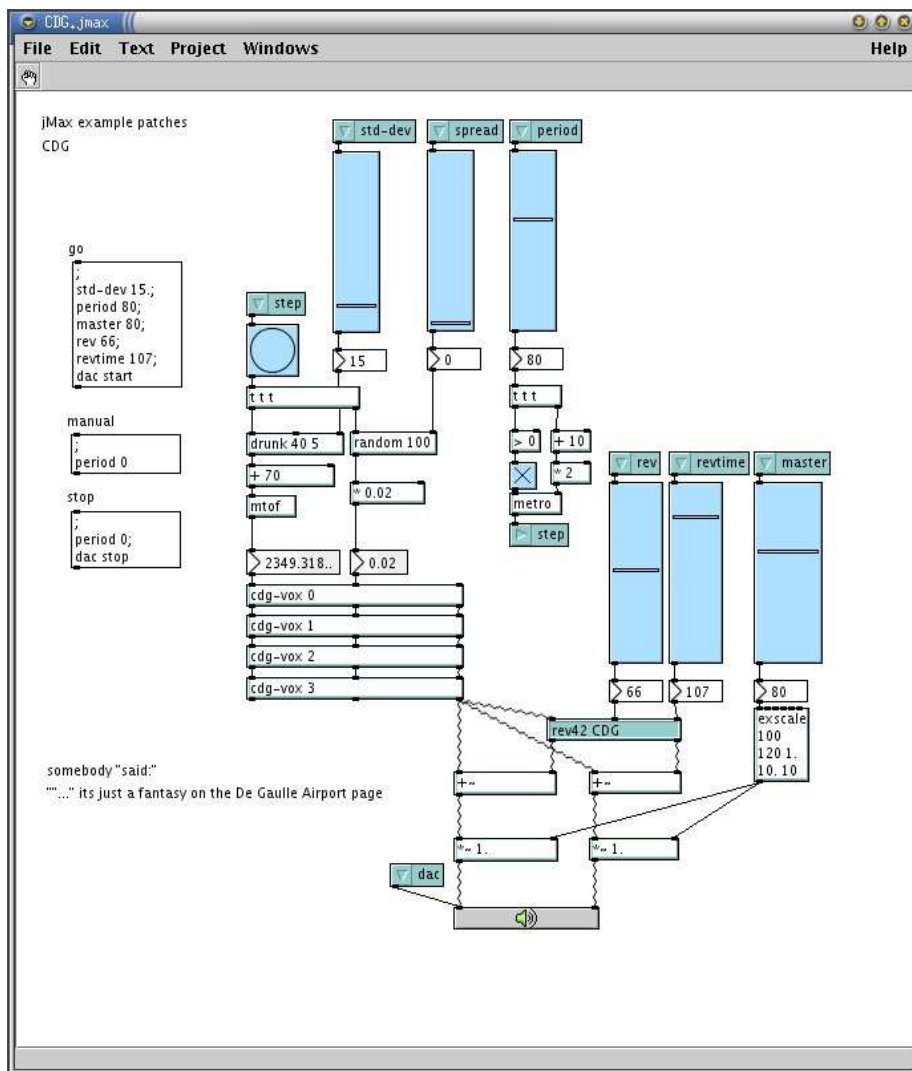


FIG. 3 – Un exemple de synthèse vocale avec jMax.

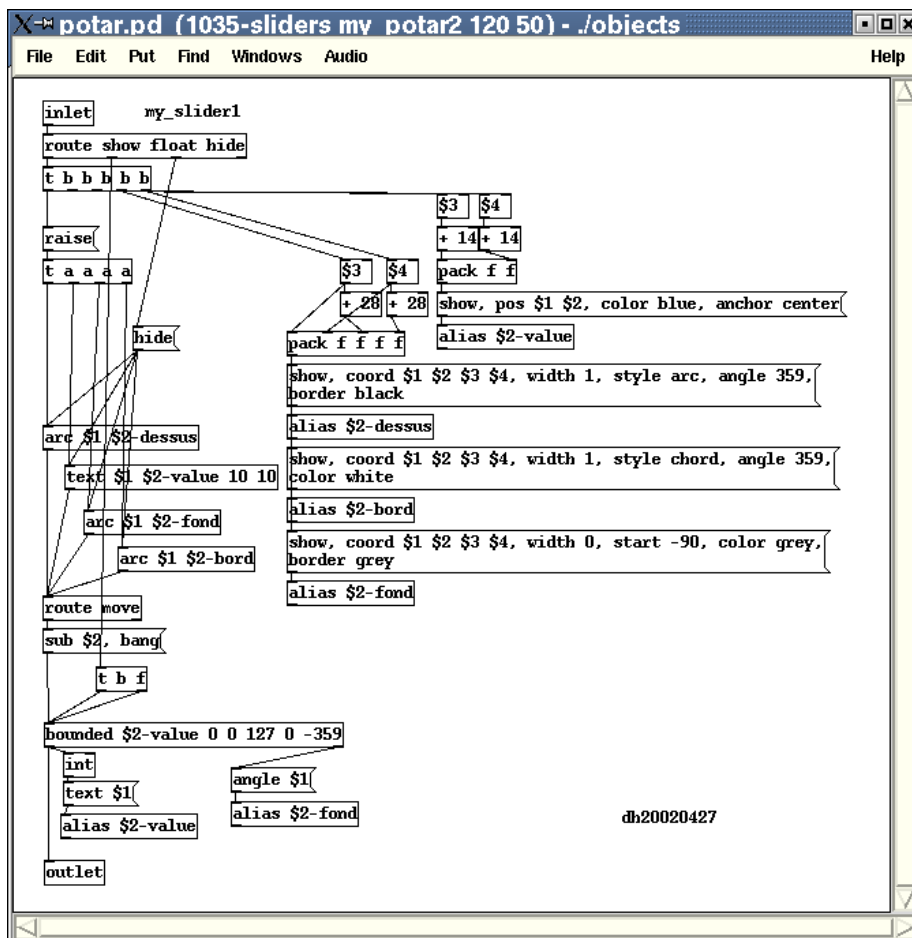


FIG. 4 – Un exemple de patch Pure Data.

2 Une étude théorique et formelle de Max

Dans cette partie, nous présentons un premier formalisme des systèmes Max, jMax, Pure Data et autres clones. Ce formalisme se base syntaxiquement et sémantiquement en partie sur celui des CSP (*Communicating Sequential Processes*), proposé par C.A.R. Hoare en 1985, et formalisant les communications entre processus indépendants. Nous discutons sur la nature de Max en tant que langage de programmation et nous montrons qu'un patch Max implémente un système réactif.

*Three currently supported computer programs —
Max/MSP, jmax, and Pd — can be considered as
extended implementations of a common paradigm I
refer to here as “Max.”*
— Miller Puckette

Des fondements théoriques de Max. Max est un programme très particulier, développé selon de bonnes idées techniques, mais à notre connaissance, aucun document explicitant ses fondements théoriques n'a été publié. Cela n'empêche pas Max de fonctionner très bien et d'être apprécié des utilisateurs, ce qui n'est pas toujours le cas d'autres systèmes basés sur des modèles formels théoriques bien définis — or il nous semble que le degré de qualité d'un logiciel se mesure le mieux au niveau de satisfaction de ses utilisateurs.

Mais ce manque de fondation théorique transparait très vite lorsqu'on lit les différents documents écrits sur Max et en particulier la première caractérisation donnée du logiciel par les auteurs. En voici quelques exemples :

- « *Max views a performance as a collection of independent objects which communicate by passing messages* » [35]
- « *Max is a data-flow language with a graphical interface that lets one manipulate the flow of data (numbers, symbols and lists) through “a patch”.* » [19]
- « *Max was an attempt to make a screen-based patching language that could imitate the modalities of a patchable analog synthesizer.* » [36]
- « *Max, though not claimed as such by its inventor, Miller Puckette, is often depicted as a visual programming language.* » [16]
- « *The Max paradigm can be described as a way of combining pre-designed building blocks into configurations useful for real-time computer music performance. This includes a protocol for scheduling control- and audio-rate computations, an approach to modularization and component intercommunication, and a graphical representation and editor for patches.* » [37]
- « *Max/MSP is a graphical environment for music, audio, and multimedia.* » (Page de présentation de Max/MSP sur le site Internet de Cycling'74).

Chaque document donne sa propre définition de ce qu'est Max — on retrouve les notions de “visuel” et de “patch”, mais dans l'ensemble cela reste métapho-

rique, et on constate le manque de caractérisation de ce qu'est ce système dans sa nature.

Partie contrôle. Ce document se concentre uniquement sur l'aspect contrôle de Max : les diverses extensions ajoutées par la suite pour le traitement de flux audio, comme MSP, mais aussi les parties audio de jMax et de PureData, ne seront pas considérées, car le paradigme en est très différent.

2.1 Langage de programmation

Une question fréquemment posée par les utilisateurs de Max et par les acteurs du domaine de l'informatique musicale est : Max est-il un langage de programmation ?

Nous rejetons d'emblée la question de la programmation en C de nouvelles primitives : en effet, ceci est une caractéristique d'implémentation, pas une caractéristique du paradigme lui-même. Si dans la pratique, grâce au C, on peut "tout faire" avec Max, en restant dans le modèle visuel que propose le système de prime abord, on est bel et bien limité aux primitives déjà existantes qu'on nous propose.

Donner une réponse définitive à la question posée dans ce paragraphe dépasserait le cadre de ce document ; ce serait un gros travail théorique que de montrer que Max est équivalent à un autre système dont on sait qu'il implémente effectivement un langage de programmation.

Certes, la question est importante pour donner un aperçu complet des fondations théoriques de Max, mais il est de notre avis qu'elle n'est peut-être pas la plus *pertinente*. Nous nous expliquons.

Une brève étude de la littérature scientifique consacrée à Max (et notamment l'article [19]) permet de relever un certain nombre d'arguments tendant à montrer que Max n'est pas un langage de programmation. En voici certains :

- Max ne propose pas de syntaxe particulière pour écrire des boucles ;
- il n'est pas possible de définir un patch récursivement ;
- Max ne connaît pas la notion de variable ;
- les structures de données (de base) de Max sont très pauvres ;
- Max n'est plus aussi pratique lorsque les programmes deviennent grands ;
- le paradigme visuel utilisé ne se prête pas bien à la programmation d'une manière générale.

Notre sentiment est que ces arguments n'en sont pas vraiment. Mis à part pour la récursivité (que l'on peut toutefois remplacer par une structure opérations itératives programmée "à la main"), on peut dire exactement la même chose des différents langages d'assemblage classiques, dont personne ne doute qu'ils sont des langages de programmation.

En fait, il nous semble que la comparaison avec les langages d'assemblage est intéressante : ces derniers n'offrent pas de structure ou d'instruction de haut niveau, mais en y mettant l'énergie nécessaire, on peut toujours finir par implémenter ce qu'on veut, quitte à essayer le feu nourri des critiques des langages de bas niveau qui diront tous en chœur que d'autres langages existent qui permettent de faire exactement telle ou telle tâche facilement. De la même manière qu'on n'implémente pas un système d'exploitation en Lisp, ni un système de contrôle d'une centrale nucléaire en T_EX, ni un utilitaire de reconnaissance de modèles de texte en assembleur, on n'utilise pas Max pour faire ce pour quoi il

n'a pas été prévu. La figure 5, page 16, en montre un exemple : il s'agit d'un patch implémentant un tri à bulle (l'un des algorithmes de tris les plus simples qui soient) sur un tableau (objet *table* dans Max), avec uniquement quelques primitives de base. Même écrit en assembleur, avec les primitives d'accès à la mémoire et de copie de données, ce simple tri prendrait sans doute moins d'instructions que l'implémentation ici proposée. Ceci nous montre bien que le tri de tableaux n'est pas ce pour quoi Max a été créé.

La question de savoir si Max est effectivement un langage de programmation nous semble donc de ce fait moins importante que la question suivante : en quoi la nature théorique de Max influe-t-elle sur son expressivité et sur sa pertinence dans la résolution de problèmes donnés ?

Cette question trouvera un début de réponse dans notre exposé du modèle formel du paradigme de Max, mais sa réponse définitive est laissée, à l'instar de celle posée au début de cette section, à ceux qui souhaiteront approfondir après nous l'étude théorique de Max.

2.2 Le patch Max : un système réactif

Nous commençons, avant de donner un aperçu théorique du fonctionnement interne d'un patch Max, par donner un aperçu de sa nature externe, c'est-à-dire de dire *ce que c'est* avant d'expliquer *comment il fonctionne*.

Résumons rapidement les caractéristiques d'un patch :

- Un patch reçoit des valeurs depuis un dispositif (logiciel ou matériel) extérieur, traite ces valeurs puis renvoie d'autres données, calculées à partir des premières, à un dispositif extérieur (le même, ou un autre).
- On attend d'un patch qu'il réagisse vite aux données d'entrée. Un patch qui a pour fonction de doubler à l'octave une note jouée sur un synthétiseur doit émettre le doublon instantanément, sans que le musicien ne perçoive de délai.
- On part du principe, implicite, qu'un patch se comporte de manière cohérente tout au long de la durée de son utilisation. Plus précisément, on souhaite qu'un patch soit déterministe : mis à part dans le cas d'utilisation de générateurs de nombres aléatoires, une même séquence de valeurs d'entrées devrait toujours donner la même séquence de valeurs de sorties.

Ces trois caractéristiques rappellent très fortement les *systèmes réactifs* (cf. [10, 11, 8, 9]).

Systèmes réactifs et systèmes interactifs. La plupart des programmes existants aujourd'hui sont des systèmes dits interactifs, en ce sens où ils sont les maîtres du jeu pour ce qui est de la séquence d'opérations à effectuer. C'est typiquement le cas d'un logiciel de traitement de texte : même si c'est l'utilisateur qui détermine les caractères à ajouter au texte qu'il compose, dès qu'il souhaite sauvegarder le texte, l'éditeur ouvre une nouvelle fenêtre pour demander à l'utilisateur le nom d'un fichier pour l'enregistrement.

Le 14 mars 1991, au Alice Tully Hall à New York, Georges Pludermacher s'installait au piano — un Yamaha Disklavier — pour interpréter *Trois esquisses en duo* de Jean-Claude Risset. Quelle aurait été son sentiment si le patch censé l'accompagner pendant son jeu lui avait subitement demandé s'il désirait enregistrer le concert dans un quelconque fichier sur le disque ?

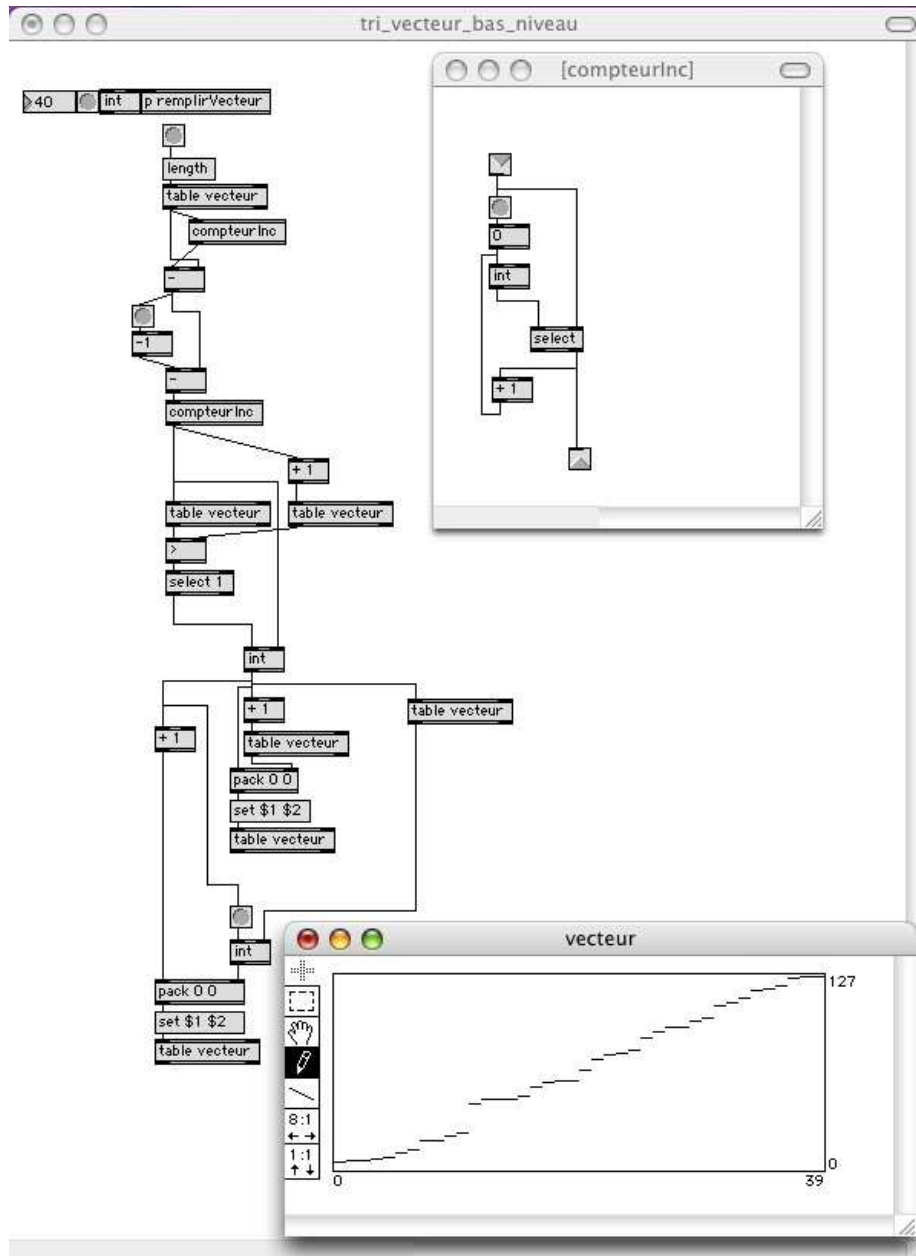


FIG. 5 – Implémentation d'un algorithme de tri à bulles sous Max. Seul le compteur [compteurInc] a été abstrait, tout le reste de l'algorithme est programmé en un patch, en n'utilisant que les primitives les plus simples.

Fort heureusement, ce n'est pas ce qui s'est passé. M. Pludermacher à joué, et le patch a docilement suivi son exécution pour produire la sienne. C'est une différence majeure entre les systèmes réactifs et les systèmes interactifs : contrairement à ces derniers, les premiers ne contrôlent pas la conversation, ils sont tenus de la suivre, donc de suivre leur environnement.

C'est exactement ce qu'on attend d'un patch : il doit être réactif, et non interactif, car il est piloté entièrement par l'environnement qui l'entoure (l'ensemble des appareils de contrôle du musicien, par exemple), et à la vitesse de celui-ci.

Déterminisme. Dans [8], G. Berry donne la définition suivante du déterminisme : « *We say that an interactive or reactive program is deterministic if its behavior only depends on its (timed) inputs.* »

Outre le rôle (actif ou passif) dans la conversation, une autre grande différence entre les systèmes réactifs et les systèmes interactifs se situe au niveau du déterminisme. Puisqu'un système réactif doit suivre et réagir à la conversation, il n'a pas de choix à faire : il est guidé par les informations qu'il reçoit, et de ce fait, s'il reçoit deux fois la même séquence d'informations, il doit reproduire deux fois le même comportement et renvoyer deux fois la même série de valeur à son environnement. On constatera donc fort logiquement que la plupart des systèmes réactifs existants sont déterministes, car ils deviennent ainsi bien plus simples à modéliser et à analyser.

Une notion de temps-réel. Il existe de nombreuses définitions de la notion de temps-réel, en fonction des domaines d'application et en fonction des exigences qu'on place sur le système considéré. Dans le domaine de l'informatique, on considère qu'un système est temps-réel dès lors qu'on connaît une limite supérieure à son temps d'exécution pour une tâche donnée. Il s'agit là d'une contrainte très forte, qui impose souvent des choix d'implémentation très complexes et difficiles à mettre en œuvre.

Dans le domaine de l'informatique musicale, on adopte souvent une notion plus souple du temps-réel, et on considère qu'un système correspond à cette caractéristique dès lors qu'il est capable d'exécuter ses tâches musicales sans prendre de retard (notable) par rapport à la musique jouée, ou encore s'il est assez véloce pour suivre le rythme imposé par les processus dépendants du temps, comme par exemple les cartes audio qui doivent régulièrement trouver dans leurs mémoires tampon des échantillons numériques à convertir en signal analogique.

Les systèmes comme Max et ses cousins correspondent plutôt à cette seconde définition. Ils n'offrent aucune garantie qu'un patch pourra réagir en un temps borné lors de la réception d'une note MIDI, mais ils ont été développés de manière à rendre le temps de réaction d'un patch le plus faible possible, utilisant de nombreuses techniques d'optimisation classiques et moins contraignantes. Un patch peut être en retard sur la musique si le traitement qu'il implémente est trop complexe, mais cela est rare, et le fait d'avoir relaxé la contrainte sur le temps-réel rend le logiciel beaucoup plus souple et simple à utiliser.

Objets réactifs. En conclusion, nous profiterons de ce paragraphe sur les systèmes réactifs pour préciser un point de vocabulaire important concernant

les “objets” dont on parle souvent en parlant de Max, jMax ou PureData, ou d’autres systèmes équivalents.

Quelque soit leur implémentation sous-jacente, il ne s’agit pas, du point de vue de l’utilisateur, d’objets au sens de la programmation orientée objets. Les systèmes ci-dessus ne donnent pas la capacité de redéfinir de nouveaux objets par héritage, ni ne proposent aucune des autres fonctionnalités propres aux langages orientés objets (polymorphisme, méthodes virtuelles, surcharge d’opérateurs, etc., cf section 2.1, p. 14.)

Max et ses cousins ne sont donc pas des langages objets, mais comme le propose David François Huynh dans [24], des systèmes « *basés sur des objets* ».

Afin de ne pas cultiver cette confusion, mais dans le souci de conserver ce terme d’objet auquel les utilisateurs sont habitués, nous décidons de qualifier ces objets de *réactifs* ; et partout où nous omettrons ce qualificatif, il sera implicite, sauf mention contraire.

Un patch Max, par construction, pourra être qualifié de *réseau d’objets réactifs* ou encore de *système réactif*, ce qui ferait de Max un *environnement de construction de systèmes réactifs musicaux*.

2.3 Introduction aux CSP

Nous proposons dans ce paragraphe une très brève introduction aux CSP, sur lesquels se fonde en partie la syntaxe et la sémantique de notre formalisme. Nous n’introduisons ici que ce qui est nécessaire à la compréhension du reste du document. En outre, cette introduction ne se veut pas parfaitement rigoureuse, mais plutôt pédagogique et simple à comprendre ; le lecteur est invité à consulter [22] pour une définition formelle et rigoureuse de toutes les notions présentées ci-après.

2.3.1 Processus et événements

Processus. Hoare donne un premier aperçu de ce qu’est un processus en ces termes : « *objects in the world around us, which act and interact with us and with each other in accordance with some characteristic pattern of behaviour.* » ([22], p. 1). Il s’agit donc de décrire une interaction entre objets indépendants. On ne donne pas de définition plus formelle de ce qu’est la nature d’un processus ; leur fonction est par contre essentielle, puisqu’il s’agit des primitives du calcul présenté dans ce document, tout comme les λ -expressions sont les primitives du λ -calcul.

On adopte la convention de nommer les processus par des noms écrits en majuscules, comme par exemple *MACHINEASOUS*.

Événements. L’élément central de l’interaction entre processus, ou entre un processus et son environnement, est *l’événement* (qu’on peut également appeler *action*). Les processus vont ainsi réagir à des événements et vont en déclencher d’autres en réaction. Les noms de classes d’événements sont écrits en minuscules, comme dans *boutonrouge*.

À titre d’exemple, on peut imaginer qu’une *MACHINEASOUS* produise ou réagisse aux événements suivants : *piece1euro*, *piece50cents*, *piece10cents*, *levier*, *jackpot*. En effet, vu de l’extérieur de la machine à sous, on ne peut qu’y insérer des pièces, baisser le levier et éventuellement gagner le *jackpot*. Ce petit exemple

montre qu'on ne s'intéresse souvent qu'à l'interface que le processus propose au monde qui l'entoure : son fonctionnement interne est souvent ignoré.

L'ensemble des événements définis pour un processus donné P est appelé *alphabet* de ce processus, et se note αP .

Le fait de s'engager dans un événement ne prend *aucun temps* : il s'agit toujours d'une action instantanée.

Notation préfixe. Si x est un événement et P un processus, on définit

$$(x \rightarrow P)$$

(prononcé "*x puis P*") comme le processus qui s'engage d'abord dans l'action x puis se comporte exactement comme le processus P . Cette notation n'est valide que si $x \in \alpha P$.

Il est ainsi possible de définir un processus de manière récursive :

$$CLOCK : (tick \rightarrow CLOCK)$$

Cette définition est plus courte qu'une définition plus explicite :

$$CLOCK : tick \rightarrow tick \rightarrow tick \rightarrow \dots$$

Expressions. Dans la notation des CSP, il est possible d'écrire des expressions en tant qu'événements. La sémantique d'une telle expression est alors souvent triviale. Ainsi par exemple, le processus suivant :

$$((a := b + c) \rightarrow P)$$

commence par donner à a la valeur de $b+c$, puis se comporte comme le processus P . La notation des CSP telle qu'elle est définie par Hoare est relativement souple.

Choix. Lorsqu'un processus peut suivre plusieurs "branches" d'exécution, on utilise l'opérateur de choix "|". A titre d'exemple, une pièce de monnaie *PIECE* qu'on lance peut tomber soit sur *pile*, soit sur *face*. On notera :

$$PIECE : (pile \rightarrow PIECE \mid face \rightarrow PIECE).$$

Processus et événements particuliers. On définit deux processus particuliers :

- $STOP_A$ est un processus qui ne s'engage dans aucun des événements de l'alphabet A . Ceci s'apparente au comportement d'un objet qui tombe en panne et qui doit être réparé avant de pouvoir interagir à nouveau avec le monde qui l'entoure. On pourra également le noter plus simplement $STOP$.
- $SKIP$ décrit un processus qui termine de manière correcte son travail. Un processus qui s'engage dans un certain nombre d'événements pour finir par se comporter comme $SKIP$ est un processus qui a terminé de manière satisfaisante un cycle d'opérations et est prêt à recommencer ce cycle.

On définit également l'événement \checkmark qui indique une terminaison positive d'un processus.

Construction de processus. Grâce aux définitions données ci-avant, nous sommes finalement en mesure de donner une définition par construction des processus :

$$P ::= STOP \mid SKIP \mid (P) \mid a \rightarrow P \mid P \mid P .$$

où (P) dénote l'utilisation de parenthèses de manière similaire à l'usage des parenthèses dans l'écriture mathématique, et où $a \in \alpha P$.

Notons que dans [22], Hoare définit bien d'autres opérateurs, mais nous n'en avons pas besoin, et par souci de simplicité, nous ne les mentionnons donc pas ici.

Similitudes avec la programmation. Le formalisme de Hoare s'inspire très fortement de la pratique de la programmation ; il donne d'ailleurs de nombreux exemples d'implémentation dans un langage proche du Lisp. Ainsi, il utilise par exemple des clauses *if ... then ... else* ou encore les concepts de fonctions, qu'il note sous la forme $\lambda x \bullet f(x)$.

2.3.2 Traces

La trace d'un processus renseigne sur la séquence d'actions dans laquelle le processus s'est engagé.

Une trace se note comme une liste de noms d'événements séparés par des virgules, et délimitée par les signes $\langle \rangle$. Nous donnons, pour notre *MACHINEASOUS* un exemple de trace :

$$\langle piece1euro, levier, piece50cents, piece50cents, levier, jackpot \rangle .$$

Le nombre d'éléments d'une trace t se note $\#t$, et le i -ème élément de t est $t[i - 1]$ (la numérotation des éléments commençant à 0). La concaténation de deux traces se note $t_1 \frown t_2$.

La seule trace possible du processus *SKIP* est $\langle \checkmark \rangle$.

2.3.3 Communications et canaux

Les processus communiquent entre eux en s'envoyant des événements, qui transitent dans des *canaux de communication*. On note en minuscules les noms de ces canaux.

La transmission d'une valeur x sur le canal c se note $c!x$, et la réception d'une valeur y sur un canal d se note $d?y$. Si on ne tient pas compte du sens de la communication, on note $c.x$ la transmission de x sur c .

2.4 Syntaxe du modèle formel de Max

2.4.1 Objets réactifs et connexions

Objets réactifs primitifs. Un objet réactif primitif (ou plus simplement *primitive*) se note graphiquement comme le montre la figure 6.

Cette figure 6 montre en outre la notation retenue pour les entrées et les sorties d'un objet réactif, sous forme de petits rectangles dessinés sur le haut (pour les entrées) ou sur le bas (pour les sorties) de la boîte.

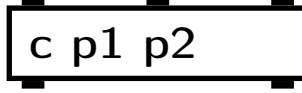


FIG. 6 – Notation graphique d’un objet réactif, de classe c , de paramètres $p1$ et $p2$ et comportant trois entrées et deux sorties.

Cependant, le nombre de sorties et d’entrées étant fixe pour une classe et un ensemble de paramètres d’initialisation donnés³, nous noterons l’objet réactif plus simplement $\boxed{c \ p1 \ p2}$, ou encore $[c \ p1 \ p2]$ dans un contexte textuel.

Un objet réactif primitif de notre formalisme est un processus au sens des CSP. On note \mathcal{P} l’ensemble de tous les processus définissables par le formalisme des CSP.

Ces objets réactifs, même s’ils sont des processus, ont quelques propriétés que n’ont pas les processus “classiques”. Nous notons donc \mathcal{M} l’ensemble des processus dont on peut dire qu’ils sont des objets réactifs.

Pour tout objet réactif Ω de \mathcal{M} , on note :

- $S(\Omega)$ le symbole donnant la classe de l’objet.
- $P(\Omega)$ la liste des paramètres d’initialisation de l’objet.
- $I(\Omega)$ la liste des canaux d’entrée de Ω . Pour chaque entrée i de Ω , on note α_i son alphabet, c’est-à-dire l’ensemble des valeurs que l’objet peut recevoir sur ce canal.
- $O(\Omega)$ la liste des canaux de sortie de Ω . On note α_o l’alphabet de la sortie o .
- $\alpha\Omega = \bigcup_{i \in I(\Omega)} \alpha_i \quad \cup \quad \bigcup_{o \in O(\Omega)} \alpha_o$ l’alphabet de Ω .

Connexions. On note graphiquement la connexion entre une sortie o et une entrée i par un simple trait allant de l’une à l’autre, comme le montre la figure 7.

³En effet, nous considérons que lorsque l’utilisateur modifie les paramètres d’initialisation d’un objet, il change d’objet, et le nombre d’entrées/sorties du nouvel objet sera déterminé par ces nouveaux paramètres. Nous ne connaissons aucun objet de Max qui puisse changer dynamiquement le nombre de ses entrées ou sorties, en recevant un tel ordre sur l’une de ses entrées.

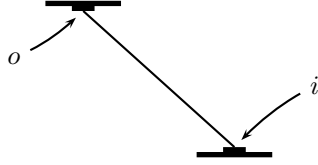


FIG. 7 – Notation graphique d’une connexion.

On peut également noter textuellement $o \rightsquigarrow i$, ou encore $\Omega_1.o \rightsquigarrow \Omega_2.i$ (si $o \in O(\Omega_1)$ et $i \in I(\Omega_2)$ — cette notation permet de lever les ambiguïtés si deux entrées ou sorties portent le même nom.)

Patches. Un patch est un ensemble d’objets réactifs, éventuellement connectés entre eux. Il peut se noter de manière simplifiée (sans faire apparaître les objets ou patches qui le composent) d’une façon similaire aux primitives, mais sans nom ou paramètre d’initialisation, comme le montre la figure 8.



FIG. 8 – Notation graphique d’un patch.

Un patch se définit récursivement par les règles syntaxiques graphiques suivantes :

- Un objet primitif (figure 9) ;

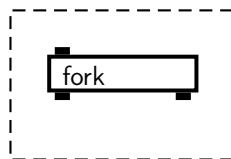


FIG. 9 – Notation graphique d’une primitive.

- Une association de deux patches en parallèle, c’est-à-dire deux patches qui ne sont pas connectés entre eux (figure 10) ;

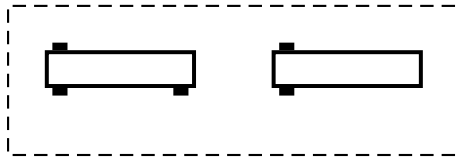


FIG. 10 – Deux patches en parallèle.

- Une association de deux patches en série, c'est-à-dire deux patches connectés de telle sorte qu'une ou plusieurs sorties de l'un des deux soient connectées à une ou plusieurs entrées de l'autre patch (figure 11).

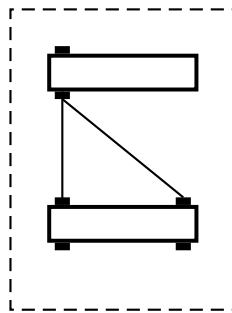


FIG. 11 – Deux patches en série.

- Un patch rebouclé sur lui-même, dont une sortie est connectée à une entrée (figure 12) ;

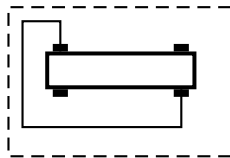


FIG. 12 – Patch rebouclé sur lui-même. Ici, c'est la deuxième sortie qui est rebouclée sur la première entrée.

Comportement des objets réactifs. Les comportements des objets réactifs sont vus comme des processus au sens des CSP, et nous utiliserons en conséquence les notations définies par Hoare pour définir ces comportements.

2.4.2 Données

Atomes. Les différentes implémentations du système de que nous décrivons ici (Max, PureData, jMax, etc.) utilisent des ensemble de données *atomiques* différents. Pour simplifier et aussi pour aller à l'essentiel, nous allons définir formellement un ensemble de données que les objets de notre formalisme pourront se transmettre via leurs canaux de communication.

Nous notons \mathbb{S} l'ensemble des symboles utilisables dans un patch. Ces symboles sont des chaînes de caractères pour lesquels on définit l'identité comme étant l'égalité caractère par caractère : si deux symboles s'écrivent de la même façon, il s'agit alors du même symbole. On notera les symboles précédés d'une double apostrophe : *"bang*.

Enfin, nous introduisons la valeur particulière *nil*, dont la signification est à peu près : "aucune valeur".

Nous pouvons ainsi définir \mathbb{A} l'ensemble des atomes existants :

$$\mathbb{A} = \mathbb{S} \cup \mathbb{R} \cup \{nil\}$$

Listes. Notre formalisme définit aussi un ensemble de données non atomique, sous formes de liste ; l'ensemble des listes se note \mathbb{L} .

Une liste se note $l = (e_0, e_2, \dots, e_n)$ avec $\forall i : 0 \leq i \leq n \bullet e_i \in \mathbb{A}$. On note $\#l$ la longueur de la liste, l_0 le premier élément de la liste et $l[i - 1]$ le i -ème élément de la liste, la numérotation commençant à 0.

Les listes sont bien sûr ordonnées, et on définit l'égalité entre deux listes de la manière suivante :

$$l_1 = l_2 \quad \equiv \quad (\#l_1 = \#l_2 \quad \wedge \quad \forall i : 0 \leq i < \#l_1 \bullet l_1[i] = l_2[i]).$$

On note $\mathbb{D} = \mathbb{A} \cup \mathbb{L}$ l'ensemble des données échangeables entre deux objets.

2.5 Sémantique du modèle formel

2.5.1 Objets réactifs

Primitives. Le comportement des primitives se définit suivant la notation des expressions gardées (*guarded expressions*) des CSP, et commencent par la réception d'une valeur sur un canal d'entrée et se terminent, si tout va bien, par le processus *SKIP* qui indique une terminaison sans échec. Par exemple, si Ω est un objet qui envoie immédiatement sur sa sortie *out* les valeurs qu'il reçoit en entrée sur *in*, on pourra définir son comportement par :

$$\Omega : in?x \rightarrow out!x \rightarrow SKIP.$$

Patches. Un patch se définit par construction de primitives ou d'autres patches. L'ensemble des entrées d'un patch est alors l'union des entrées des primitives et patches qui le composent (y compris celles qui sont déjà connectées à des sorties). De même pour les sorties. L'alphabet du patch est l'union des alphabets de ses entrées.

On note θ le patch total qu'on construit ainsi ; θ est toujours le patch de "plus haut niveau".

Si deux patches P_1 et P_2 sont construits en série ou en parallèle, ou si un patch est rebouclé sur lui-même (ces trois cas de figure correspondent aux trois notations définies plus haut, p. 22), le comportement du patch ainsi défini sera déterminé par les connexions ; il n'est pas possible de déterminer *a priori* un "ordre d'exécution".

2.5.2 Communication entre objets réactifs

Connexion. Il est impossible de connecter un canal de sortie à un autre canal de sortie, ni de connecter un canal d'entrée à un autre canal d'entrée.

Une entrée i et une sortie o ne peuvent être connectées ensemble que si $\alpha o \subset \alpha i$.

Si aucun alphabet n'est précisé pour un canal, on considérera que son alphabet est \mathbb{D} .

Communication. Soit un objet réactif Ω doté d'un canal de sortie o . Nous donnons la définition suivante à l'action $o!x$:

$$\forall i : \{j \mid j \in I(\theta) \wedge o \mapsto j\} \bullet i!x$$

Notre sémantique de cette action diffère de celle donnée par Hoare. Si un objet effectue une transmission $o!x$, alors son action est suspendue, et la valeur x est transmise séquentiellement à tous les objets ayant un ou plusieurs canaux d'entrée connectés à o . Chaque objet finit de réagir à la réception de cette valeur avant de “passer la main” à l'objet suivant qui doit recevoir cette même valeur.

Cette façon de faire correspond à une “exécution en profondeur” du patch, où chaque objet qui reçoit une valeur est “activé”, traite complètement la valeur qu'il a reçue et se termine, avant qu'une autre objet, connecté à la même sortie o ne puisse à son tour réagir à la valeur. Bien sûr, rien n'empêche un objet qui reçoit une valeur d'envoyer une autre valeur sur l'un de ses canaux de sortie avant de terminer son travail, se trouvant ainsi lui-même bloqué jusqu'à ce que cette transmission soit entièrement traitée par les objets “d'en-dessous”.

Prenons un exemple :

$$\begin{aligned} \Omega_1 & : \text{action1} \rightarrow o!x \rightarrow \text{action2} \rightarrow \text{SKIP} \\ \Omega_2 & : i?x \rightarrow \text{action3} \rightarrow \text{SKIP} \end{aligned}$$

Si $o \mapsto i$, alors la trace d'exécution de ce patch serait

$$\langle \text{action1}, o!x, i?x, \text{action3}, \surd_{\Omega_2}, \text{action2}, \surd_{\Omega_1} \rangle.$$

Si plusieurs objets sont connectés à la sortie o de Ω_1 , aucun ordre n'est déterminé pour savoir quel objet recevra la valeur en premier. Par contre, cet ordre, s'il n'est pas déterminé est néanmoins immuable. Les objets recevront toujours la valeur envoyée par Ω_1 dans le même ordre.⁴

2.6 Abstractions

Une abstraction est un patch P contenant un ou plusieurs objets [inlet] et/ou [outlet] (définis page 30) et auquel on donne un nom. On note $inlets(P)$ la liste des objets [inlet] de P , et $outlets(P)$ ses objets [outlet]. Tous les objets [inlet] doivent comporter un numéro de paramètre distinct, tout comme les objets [outlet]. La liste des entrées de P est alors composée des entrées des objets de $inlets(P)$ dans l'ordre croissant de leurs indices (ordre dans lequel ils sont classés dans $inlets(P)$), et de même pour les sorties de P .

⁴Les utilisateurs de Max sont habitués à ordonner les objets en fonction de leur position à l'écran, mais jMax et PureData n'utilisent pas cette règle et recourent à un ordre d'exécution arbitraire.

Le nom $S(P)$ qu'on donne à l'abstraction permet d'identifier ce modèle de patch (composition et entrées/sorties), et permet d'instancier plusieurs objets similaires qui seront autant de répliques de cette abstraction. La différence entre chacune de ces instances réside par la suite dans les valeurs contenues dans les mémoires des objets qui les composent.

Ainsi, une abstraction P se définit comme suit :

- $P \in \mathcal{M}$;
- $S(P) \in \mathbb{S}$;
- $I(P) = (I(\text{inlets}(P)[0])_0, \dots, I(\text{inlets}(P)[N])_0)$ avec $N = \#\text{inlets}(P)$;
- $O(P) = (O(\text{outlets}(P)[0])_0, \dots, O(\text{outlets}(P)[M])_0)$ avec $M = \#\text{outlets}(P)$;
- Le comportement de P est défini par sa construction (i.e. par les objets qui le composent et par leurs interconnexions).

La figure 13 donne un exemple d'abstraction.

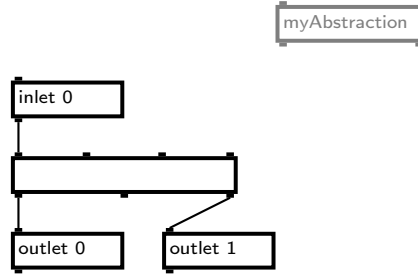


FIG. 13 – Construction de l'abstraction myAbstraction.

2.7 Temps et ordonnancement

Notre modèle définit une machine infiniment rapide, dans laquelle les opérations ne prennent aucun temps.

Cependant, les utilisateurs de Max et de ses cousins sont très dépendants de la possibilité de pouvoir déclencher des événements dans le futur. Il nous faut donc un modèle du temps qui passe et un mécanisme pour inscrire des événements dans le futur.

Nous définissons le temps comme une variable discrète positive t dont la valeur ne diminue jamais au cours de la vie d'un patch.

Nous nous dotons également d'une fonction *schedule* qui permet d'inscrire l'envoi d'une valeur sur un canal à un instant donné t' ; plus précisément, la valeur donnée ne sera pas envoyée *avant* la date t' , et on espère qu'elle sera envoyée le plus tôt possible après. Cette granularité du temps nous est donnée par la valeur δt , constante au cours de la durée de vie du patch.

$$\text{schedule}(o, v, t') := \lambda o, v, t \bullet T := T \cup \{(o, v, t')\}$$

où T est l'ensemble des triplets (*valeur, canal de sortie, date*) décrivant l'envoi de valeurs v sur des canaux de sortie o à des instants dans le futur, pas avant l'instant t' mais avant l'instant $t' + \delta t$.

Le mécanisme par lequel ces événements futurs sont déclenchés s'appelle le *scheduler*, qui est un processus activé dès le début de la vie d'un patch :

$$\begin{aligned}
SCHEMULER & : \textit{startWait} \rightarrow \textit{stopWait} \rightarrow (t := t + \delta t) \\
& \rightarrow (\forall (o, v, t') : \{T \mid t' \leq t\} \bullet o.v; T := T \setminus (o, v, t')) \\
& \rightarrow SCHEMULER
\end{aligned}$$

La période d'attente entre les événements *startWait* et *stopWait* (Hoare précise que les événements ne prenant aucun temps, une période de temps se représente par deux événements indiquant le début et la fin de la période) représente le passage physique du temps, à une vitesse proportionnelle à la vitesse logique du temps dans le patch. En général, on souhaite que cette période d'attente corresponde à la durée δt pour faire ce qu'on appelle du "temps-réel".

2.8 Grandes familles d'objets

Si tous les objets primitifs présentés en paragraphe 2.9 peuvent être modélisés uniquement grâce au formalisme tel qu'il a été présenté dans les parties précédentes, on observe facilement dans ces objets de grandes familles de comportements qui laissent à penser qu'on peut pour un certain nombre d'entre eux factoriser leurs points communs.

C'est le cas notamment pour les objets qui suivent les conventions très largement répandues parmi les objets dans les logiciels Max, jMax ou PureData, comme par exemple le fait que l'entrée la plus à gauche est celle qui déclenche la réaction de l'objet, ou encore la manière de répondre à la réception du symbole "bang".

Nous détaillons dans les deux sous-paragraphes qui suivent deux grandes familles d'objets ayant des caractéristiques comportementales communes.

2.8.1 Objets conventionnels

Nous appelons \mathcal{S} l'ensemble des objets de \mathcal{M} qui respectent quelques conventions simples :

1. Chaque objet est associé à une *action* ;
2. Chaque entrée est munie d'une *mémoire* qui permet de retenir la dernière valeur reçue sur cette entrée ;
3. Certaines entrées sont dites *réactives*, et déclenchent l'action dès qu'elles reçoivent une valeur ;
4. L'action peut être déclenchée également par la réception du symbole "bang" dans la première entrée.

On définira ainsi, pour tout objet Ω de \mathcal{S} :

- $R(\Omega) \subseteq I(\Omega)$ le sous-ensemble des entrées réactives.
- Pour toute entrée i dans $I(\Omega)$, $m(i)$ est la mémoire de cette entrée, et conserve sa valeur dans le temps jusqu'à une prochaine affectation.
- $B(\Omega)$ est l'action associée à l'objet. Il s'agit d'une fonction qui sera exécutée à chaque réception d'une valeur sur une entrée réactive ou du symbole "bang" sur la première entrée.

Le comportement des objets Ω de \mathcal{S} est défini comme suit :

$$\begin{aligned}
\Omega & : (i_0?x \rightarrow (\text{if } x = \text{"bang"} \text{ then } B(\Omega) \text{ else} \\
& \quad (m(i_0) := x) \rightarrow (\text{if } i_0 \in R(\Omega) \text{ then } B(\Omega))) \\
& \quad \rightarrow \text{SKIP}) \\
& i_1?x \rightarrow (m(i_1) := x) \rightarrow (\text{if } i_1 \in R(\Omega) \text{ then } B(\Omega) \rightarrow \text{SKIP}) \\
& i_2?x \rightarrow (m(i_2) := x) \rightarrow (\text{if } i_2 \in R(\Omega) \text{ then } B(\Omega) \rightarrow \text{SKIP}) \\
& \dots \\
& i_{n-1}?x \rightarrow (m(i_{n-1}) := x) \rightarrow (\text{if } i_{n-1} \in R(\Omega) \text{ then } B(\Omega) \rightarrow \text{SKIP})
\end{aligned}$$

La fonction $B(\Omega)$ est chargée d'émettre des valeurs en sortie en fonction des valeurs des mémoires des canaux d'entrée. Comme il est possible que l'objet reçoive comme première valeur en entrée le symbole "bang" sur le premier canal, chaque mémoire doit être correctement initialisée pour que l'objet soit dans un état cohérent. Cette initialisation se note sous forme d'exposant ("*i*init") sur le nom du canal lors de la définition de la liste des entrées $I(\Omega)$. Toutes les valeurs initiales doivent être précisées.

2.8.2 Opérateurs

De nombreux objets habituels de Max, jMax ou PureData correspondraient à la catégorie des objets standards mais sont encore plus fortement conventionnels. Il s'agit notamment de toutes les fonctions arithmétiques ou logiques, qui respectent les conventions suivantes, en plus de celles déjà établies pour les objets conventionnels :

1. Seul le premier canal d'entrée est réactif ;
2. Il n'y a qu'un canal de sortie ;
3. La fonction d'action se résume à l'application d'une fonction mathématique à l'ensemble des valeurs des mémoires des entrées et à l'envoi du résultat de cette fonction sur le canal de sortie.

Pour ces objets-là, qu'on regroupe dans l'ensemble des opérateurs de n arguments $\mathcal{E}^n \subset \mathcal{S}$, on définit :

- $\#I(\Omega) = n$;
- $R(\Omega) = (I(\Omega)_0)$;
- $\#O(\Omega) = 1$;
- $f_\Omega : \alpha I(\Omega)[0] \times \dots \times \alpha I(\Omega)[n-1] \rightarrow \alpha O(\Omega)[0]$ est la fonction appliquée aux valeurs des mémoires des entrées de l'objet ;
- $B = \lambda\Omega \bullet o!f(m(I(\Omega)[0]), \dots, m(I(\Omega)[n-1]))$

2.9 Primitives

Nous définissons enfin l'ensemble des objets réactifs primitifs de notre modèle formel, grâce auxquels il est possible de redéfinir la plupart des objets "classiques" de Max ; les objets faisant appel à des entrées/sorties (comme les objets MIDI par exemple) peuvent être définis à partir du moment où l'on suppose qu'à côté de l'ordonnanceur vivent d'autres processus en charge des communications

avec les périphériques de l'ordinateur, et auxquels sont automatiquement reliés les objets tels que `midin`, `notein` ou encore `MouseState`, pour n'en citer que quelques uns inspirés des objets réactifs fournis dans Max.

2.9.1 Objets conventionnels

[bang]

Cet objet est très utilisé dans Max et ses cousins pour déclencher des actions. Tout ce qu'il fait est d'envoyer sur son canal de sortie le symbole `"bang`

$$\begin{aligned} [\text{bang}] \in \mathcal{M} \quad I([\text{bang}]) &= (i) \quad O([\text{bang}]) = (o) \\ \alpha o &= \{\text{"bang"}\} \\ [\text{bang}] : i?x \rightarrow o!\text{"bang"} &\rightarrow \text{SKIP} \end{aligned}$$

[fork]

Cet objet permet de forcer l'ordre dans lequel deux objets vont recevoir la valeur émise en sortie d'un troisième objet. Dans Max, cet objet porte le nom de `[trigger]` et est un peu plus évolué que la version que nous en donnons ici. Sous jMax, il est présenté graphiquement comme une fourche montrant une entrée reliée à deux sorties ou plus.

$$\begin{aligned} [\text{fork}] \in \mathcal{M} \quad I([\text{fork}]) &= (i) \quad O([\text{fork}]) = (o_1, o_2) \\ [\text{fork}] : i?x \rightarrow o_2!x \rightarrow o_1!x &\rightarrow \text{SKIP} \end{aligned}$$

[gate]

L'objet `[gate]` permet de bloquer ou de laisser passer des valeurs de \mathbb{D} .

$$\begin{aligned} [\text{gate}] \in \mathcal{S} \quad I([\text{gate}]) &= (\text{state}^0, \text{data}^0) \quad R([\text{gate}]) = (\text{data}) \quad O([\text{gate}]) = (o) \\ B([\text{gate}]) &= \lambda \bullet \text{if } m(\text{state}) \neq 0 \text{ then } o!m(\text{data}) \end{aligned}$$

[pipe]

L'objet `[pipe]` est la seule primitive qui permette de projeter un événement dans le futur. C'est donc la seule primitive temporelle de notre modèle.

$$\begin{aligned} [\text{pipe } n] \in \mathcal{S} \quad I([\text{pipe } n]) &= (\text{data}^0, \text{delay}^n) \quad R([\text{pipe } n]) = (\text{data}) \\ O([\text{pipe } n]) &= (o) \quad B([\text{pipe } n]) = \lambda \bullet \text{schedule}(o, m(\text{data}), m(\text{delay})) \end{aligned}$$

[inlet]

L'objet [inlet n] sert à définir un canal d'entrée pour une abstraction.

$$[\text{inlet } n] \in \mathcal{M} \quad I([\text{inlet } n]) = (i_n) \quad O([\text{inlet } n]) = (o_n)$$

$$[\text{inlet } n] : i_n ? x \rightarrow o_n ! x \rightarrow \text{SKIP}$$

On donne la même définition pour l'objet [outlet], qui définit une sortie pour une abstraction.

[identity]

Cet objet représente une mémoire ; Max l'appelle [int] ou [float]. Sa définition est un peu différente, car elle teste le canal sur lequel à lieu la communication pour décider s'il faut envoyer la valeur en sortie ou non. Une valeur reçue par i_1 provoque l'envoi immédiat de cette même valeur en sortie, alors qu'une valeur reçue en i_2 ne fait que mettre en mémoire la valeur.

$$[\text{identity}] \in \mathcal{S} \quad I([\text{identity}]) = (i_1^0, i_2^0) \\ R([\text{identity}]) = (i_1, i_2) \quad O([\text{identity}]) = (o)$$

$$B([\text{identity}]) = \lambda \bullet \begin{array}{l} \text{if } i_2.x \text{ then} \\ \quad m(i_1) := x \\ \text{else} \\ \quad o!x \end{array}$$

2.9.2 Opérateurs

[+]

C'est l'opérateur d'addition, qui calcule la somme de deux nombres et renvoie le résultat du calcul.

$$[+] \in \mathcal{E}^2 \quad f_{[+]}(x, y) = x + y \\ \alpha I([+]) [0] = \alpha I([+]) [1] = \alpha O([+]) [0] = \mathbb{R}$$

On définit de manière similaire les opérateurs arithmétiques [-], [×], [/] et [%].

[<]

On définit ensuite l'opérateur de comparaison [$<$].

$$[<] \in \mathcal{E}^2 \quad f_{[<]}(x, y) = \begin{cases} 1 & \text{si } x < y \\ 0 & \text{sinon.} \end{cases}$$

$$\alpha I([<])[0] = \alpha I([<])[1] = \alpha O([<])[0] = \mathbb{R}$$

De la même manière, on définit les opérateurs [\leq], [$>$], [\geq], [$=$] et [\neq].

[\wedge]

On définit enfin l'opérateur logique de conjonction (“et logique”) :

$$[\wedge] \in \mathcal{E}^2 \quad f_{[\wedge]} = \begin{cases} 1 & \text{si } x \neq 0 \wedge y \neq 0 \\ 0 & \text{sinon.} \end{cases}$$

$$\alpha I([\wedge])[0] = \alpha I([\wedge])[1] = \alpha O([\wedge])[0] = \mathbb{R}$$

L'opérateur logique de disjonction [\vee] se définit pareillement.

2.9.3 La primitive [lisp]

L'objet primitif [lisp] est la pierre angulaire de l'interaction entre notre système (formel) temps-réel et un système temps-différé comme OpenMusic.

D'une manière générale, cette primitive [lisp] permet d'exécuter du code Lisp depuis l'environnement de notre système temps-réel qui est largement dépourvu des capacités symboliques du Lisp.

Appel de fonction. La primitive [lisp] joue le rôle d'enveloppe pour un appel de fonction distant. De ce fait, cette primitive semblerait appartenir à la classe d'objets \mathcal{E} , mais nous verrons que cela n'est pas possible — elle n'est en réalité qu'un élément de \mathcal{M} . Elle prend comme paramètre d'initialisation le nom d'une fonction Lisp à appeler. Selon la fonction Lisp donnée, le nombre d'entrées de l'objet est ajusté de manière à correspondre à chacun des paramètres qu'elle attend — la valeur par défaut de chacun de ces arguments est fixée à *nil*. L'unique sortie de cet objet est utilisée pour renvoyer le résultat de l'application de la fonction.

Les données de l'appel de fonction sont envoyées (par un mécanisme abstrait), lorsque la première entrée, qui est l'entrée réactive, reçoit une donnée, à une application prédéterminée implémentant un interpréteur Lisp afin que celle-ci puisse exécuter l'application de fonction et nous en renvoyer le résultat. Ce résultat est alors envoyé sur la sortie de notre objet [lisp].

Correspondance de types. Les données atomiques que manipule notre modèle sont assez proches de certaines données que manipule le langage Lisp. Cependant, on ne peut établir de bijection entre l'ensemble des données de Max et celui de Lisp, ce dernier étant bien plus exhaustif la plupart du temps, et autorisant surtout la construction de listes de données non atomiques, ce qui n'est pas le cas pour notre modèle, où seules les listes “plates”, où toutes les données sont atomiques, sont autorisées.

L'objet [lisp] est donc susceptible d'aboutir sur un état *STOP* si les données renvoyées par la fonction appelée ne sont pas compatibles avec les éléments de \mathbb{D} .

Description formelle. Nous donnons maintenant la description formelle de cette primitive. Comme le nombre d'entrées dépend de la fonction qu'on enveloppe, nous appellerons ce paramètre `fun` et nous supposons qu'elle prend n paramètres en argument.

Nous définissons $r([\text{lisp fun}])$ un canal de `[lisp fun]` par lequel l'application distante pourra nous communiquer sa valeur. Ce canal n'est ni dans $I([\text{lisp fun}])$, ni dans $O([\text{lisp fun}])$, il est *privé*.

$$[\text{lisp fun}] \in \mathcal{M} \quad I([\text{lisp fun}]) = (i_0^{nil}, \dots, i_{n-1}^{nil}) \quad O([\text{lisp fun}]) = (o)$$

$$\alpha I([\text{lisp fun}])[0] = \dots = \alpha I([\text{lisp fun}])[n-1] = \alpha O([\text{lisp fun}])[0] = \mathbb{D}$$

```

[lisp fun] :  $i_0?x \rightarrow (m(i_0) := x) \rightarrow Q(\text{fun}, m(i_0), \dots, m(i_{n-1}))$ 
               $\rightarrow [\text{lisp fun}]$ 
              |  $i_1?x \rightarrow (m(i_1) := x) \rightarrow [\text{lisp fun}]$ 
              ...
              |  $i_{n-1}?x \rightarrow (m(i_{n-1}) := x) \rightarrow [\text{lisp fun}]$ 
              |  $r([\text{lisp fun}]?reply \rightarrow (\text{if } reply \in \mathbb{D} \text{ then}$ 
                   $o!reply \rightarrow [\text{lisp fun}]$ 
              else
                  STOP)
```

Sémantique de l'appel. La sémantique exacte du processus

$$Q(\text{fun}, m(i_0), \dots, m(i_{n-1}))$$

n'est pas précisée : cette sémantique est établie par le système destinataire de cet appel de fonction distant.

Temps d'exécution. Le canal spécial $r([\text{lisp fun}])$ est destiné à recevoir la réponse de la part du système distant devant traiter l'appel de fonction. Mais il n'est pas possible de savoir quand cette réponse sera reçue. La très grosse différence entre l'objet `[lisp]` et les autres primitives de notre modèle est donc la réactivité : théoriquement, les réactions des autres primitives sont instantanées. Ce n'est pas du tout le cas de celle-ci, car le système implémentant l'interpréteur Lisp n'offre aucune garantie (dans la pratique, ces interpréteurs sont même plutôt lents en comparaison des implémentations de Max, n'étant en général pas prévus pour le temps-réel).

Nous pouvons envisager plusieurs scénarios pour savoir comment gérer ce temps de réponse.

1. Si aucune échéance n'existe, il suffit d'attendre une réponse du système distant (notons alors que cette attente peut être infiniment longue).
2. Si il y a une échéance, là encore plusieurs cas sont possibles :
 - (a) Si le calcul effectué par la fonction `fun` est atomique, c'est-à-dire qu'aucun résultat n'existe avant la fin du calcul, alors il faut prévoir une valeur par défaut à utiliser en lieu et place de la réponse

attendue. Cette valeur par défaut doit bien sûr être connue dans notre environnement temps-réel, seul à même de pouvoir la fournir à la bonne date.

- (b) Si à tout instant du calcul, une réponse existe (comme c'est le cas par exemple pour des algorithmes de recherche locale), alors à une date $d - \delta$, située quelque peu avant la date d de l'échéance, on peut demander au système distant de nous fournir la solution qu'il a actuellement. Il faudra néanmoins prévoir une réponse par défaut comme dans le cas précédent, puisqu'il est impossible de garantir que le système distant puisse interpréter la requête et renvoyer la réponse dans l'intervalle de temps δ .
- (c) Si aucune valeur par défaut n'est satisfaisante, il faut signaler une erreur.

Ainsi, de manière fort naturelle, dès qu'on demande à un système qui n'est pas aussi contraint que nous-mêmes, la fiabilité devient un point crucial à gérer. Notre définition sémantique de ce processus [lisp] ne contient aucun mécanisme pour résoudre ce type de problèmes. En fait, ces deux problèmes sont complètement indépendants, et le choix de la méthode de résolution à utiliser se fait au cas par cas.

3 La bibliothèque Moscou

Nous avons développé les outils nécessaires à la communication efficace depuis et vers OpenMusic en se basant sur le protocole *OpenSound Control*, dit OSC [41], sous forme de bibliothèque OpenMusic : la bibliothèque *Moscou*. Dans cette partie, nous commençons par présenter brièvement le protocole OSC, puis nous détaillons le travail effectué pour la création de cette bibliothèque.

3.1 Présentation d'OpenSound Control

OpenSound Control est un protocole de communication entre ordinateurs, synthétiseurs et autres appareils multimédia, développé par le *Center for New Music and Audio Technologies*, ou CNMAT, à Berkeley, *University of California*. Ce protocole, qui spécifie une manière de formater des données pour l'échange d'information, est conçu pour reposer sur les protocoles de transport de données sur réseaux actuels. Il ne précise donc pas comment *envoyer* les données, mais comment les *organiser*.

Le message comme unité de base. OSC fonde son principe sur l'envoi d'un message d'une application (le client) à une autre (le serveur). Un message se présente sous la forme d'un sélecteur, suivi d'un certain nombre d'arguments. La sémantique d'un tel message est de déclencher chez le serveur une action correspondant au sélecteur donné, paramétrée par les valeurs des arguments du message, ce qui correspond à un appel de fonction dans de très nombreux langages de programmation.

Hiérarchie de noms. Le protocole suppose que le serveur, destinataire de messages, est organisé sous forme hiérarchique en conteneurs et méthodes. Les conteneurs peuvent être vu comme des objets ou des répertoires, et les feuilles de cette arborescence hiérarchique sont les méthodes à appeler. La racine s'appelle "/", et chaque nom de conteneur, jusqu'à la méthode, est séparé du précédent par le même caractère "/".

Prenons l'exemple d'un synthétiseur virtuel, comprenant entre autres deux oscillateurs. Chaque oscillateur prévoit une méthode pour modifier sa fréquence d'oscillation et une méthode pour modifier sa phase. Voici alors un extrait de l'espace de noms de notre synthétiseur :

Si depuis une application on souhaite modifier la fréquence du second oscillateur, on peut envoyer au synthétiseur le message

```
/osc2/freq 359.5
```

Le protocole OSC ne spécifie pas de profondeur ou de complexité maximale pour cet espace de noms, dont la gestion est laissée à la discrétion du serveur.

Types des arguments d'un message. Dans la version 1.0 de la spécification d'OSC, il est précisé que chaque message doit être encodé accompagné d'une chaîne de caractères spéciale indiquant à l'application recevant le message les types des arguments des messages. En effet, les arguments d'un message OSC peuvent être des entiers sur 32 bits, des flottants sur 32 bits, des chaînes de caractères (au format des chaînes de caractères en C) ou des données binaires.

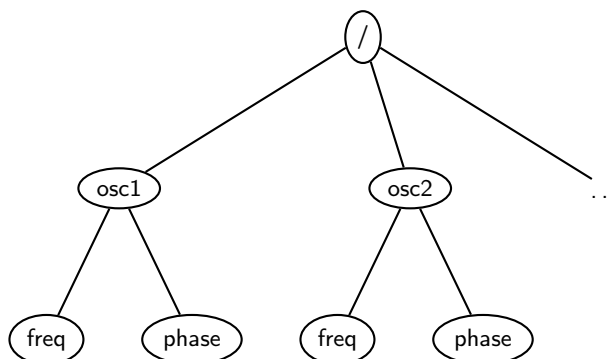


FIG. 14 – Exemple partiel d’espace de noms pour un synthétiseur virtuel.

Chacun de ces types est identifié par une lettre particulière : dans le cas des quatre types cités ci-avant, ces lettres sont respectivement “i”, “f”, “s” et “b” (pour BLOB, ou *Binary Large Object*, un terme souvent utilisé dans la littérature des bases de données pour désigner des données binaires dont la structure et la sémantiques sont inconnues de l’application).

Cette chaîne de caractères, qui commence obligatoirement par un caractère “virgule”, n’est pas transmise à l’objet destinataire du message : il n’est utilisé que par le code chargé de la réception de paquets OSC par le réseau pour recomposer la liste des arguments.

La spécification prévoit également une liste d’identifiant pour d’autres types courants, mais précise que les applications implémentant OSC ne sont pas tenues de les reconnaître et peuvent ignorer les messages qui les contiennent.

Reconnaissance de chaînes. Afin de rendre l’utilisation d’OSC plus simple, le protocole prévoit un mécanisme similaire aux expressions régulières pour préciser un modèle de nom qui correspondrait à plusieurs noms dans l’espace de noms du serveur. La syntaxe est très proche des expressions régulières classiques : “*” correspond à une séquence quelconque de zéro, un ou plusieurs caractères, “?” correspond à un seul caractère quelconque, etc. les caractères normaux correspondant à eux-mêmes.

Ainsi, une application qui souhaiterait régler à 440 Hz les fréquences des deux oscillateurs de notre synthétiseur fictif peut envoyer le message suivant :

/osc?/freq 440

La méthode `freq` sera alors appelée avec 440 comme argument pour chacun des deux oscillateurs de notre synthétiseur, puisque leurs deux noms, “osc1” et “osc2” correspondent à l’expression “osc?”.

Envois groupés. Pour mieux utiliser la bande passante du réseau, un client peut grouper les messages à envoyer à un serveur en paquets, appelés *bundles* dans la spécification d’OSC. Ceci permet de limiter la surcharge due aux informations encodées avec un message pour chaque envoi.

Étiquette temporelle. OSC prévoit que chaque *bundle* soit tamponné d'une date de prise d'effet, c'est-à-dire d'une valeur qui indique au serveur qui a reçu un ensemble de messages le moment où ces messages doivent être exécutés.

Une valeur particulière (la valeur hexadécimale `0x0000000000000001`) à une signification spéciale : elle indique qu'un message doit être exécuté immédiatement.

3.2 Moscou : OSC dans OpenMusic

Moscou est une nouvelle bibliothèque pour OpenMusic qui permet l'émission et la réception de messages *via* le protocole OSC.

Code existant. Soucieux de rendre à César ce qui appartient à César, nous précisons que cette bibliothèque se base sur un code écrit à l'origine par Carlos Agon. Notre travail a été de reprendre ce code, le corriger et le rendre plus robuste, puis de l'empaqueter dans une bibliothèque OpenMusic, en suivant les règles de l'art pour ce qui est de la création de telles bibliothèques.

Open Transport. Puisque le protocole OSC laisse à l'implémenteur le choix du protocole de transport réseau à utiliser, la plupart des implémentations choisissent le protocole le plus simple et le plus approprié, le protocole UDP, défini dans [34] et [12]. C'est le cas pour *Moscou*, qui utilise les services de la bibliothèque *Open Transport* [4], fournie par Apple avec son système MacOS X, pour faire transiter les messages OSC dans des paquets UDP.

Envoi et réception de messages. *Moscou* fournit pour l'envoi et la réception de messages un service minimaliste mais suffisant dans la plupart des cas. Sont fournies les fonctions `moscou:send` et `moscou:receive`, dont les figures 15 et 16 montrent deux exemples simples d'utilisation.

La fonction `moscou:send` prend trois éléments en paramètres : un *bundle*, c'est-à-dire une liste de messages (*Moscou* ne gère pour le moment que des *bundles*), une adresse IP de destinataire sous forme de chaîne de caractères, et un numéro de port sur la machine de destination. Pour envoyer un message à une application s'exécutant sur la même machine, on peut employer l'adresse "127.0.0.1". Il n'existe pas de numéro de port de réception réservé pour le protocole OSC, on doit donc s'assurer de ne pas utiliser un numéro de port déjà occupé.

La fonction `moscou:receive` ne prend que deux arguments en paramètre : un numéro de port de réception (ce qui signifie que plusieurs patches OpenMusic indépendant peuvent écouter des messages OSC en même temps, pour peu qu'ils se mettent d'accord sur les ports à utiliser) et un patch à exécuter lors de la réception d'un *bundle*. Ce patch doit avoir deux entrées : la première reçoit l'étiquette temporelle du *bundle*, et la seconde reçoit la liste des messages qui composent le *bundle*.

En outre, `moscou:receive` se comporte légèrement différemment des fonctions habituelles. En effet, lorsque l'utilisateur essaye d'évaluer la boîte `receive`, celle-ci plutôt que d'évaluer directement ses arguments et de renvoyer le résultat de son évaluation passe à l'état "écoute", état matérialisé par un cadre de couleur autour de la fonction (comme on le voit sur la figure 16). Ce n'est que

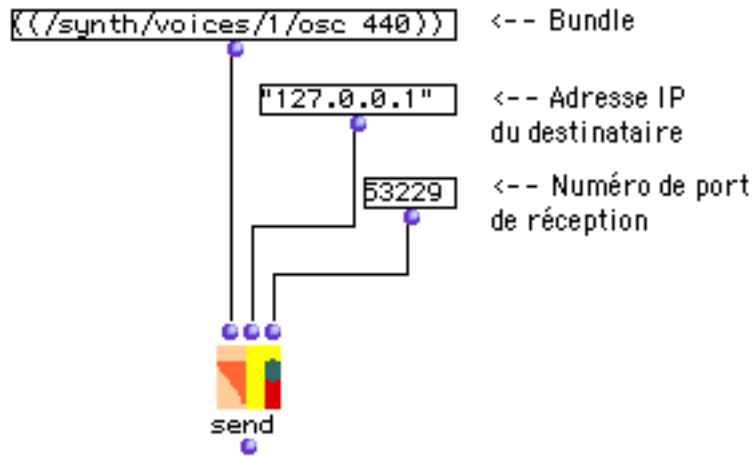


FIG. 15 – Envoi du message `/synth/voices/1/osc 440` à l'adresse IP `127.0.0.1` sur le port `53229`.

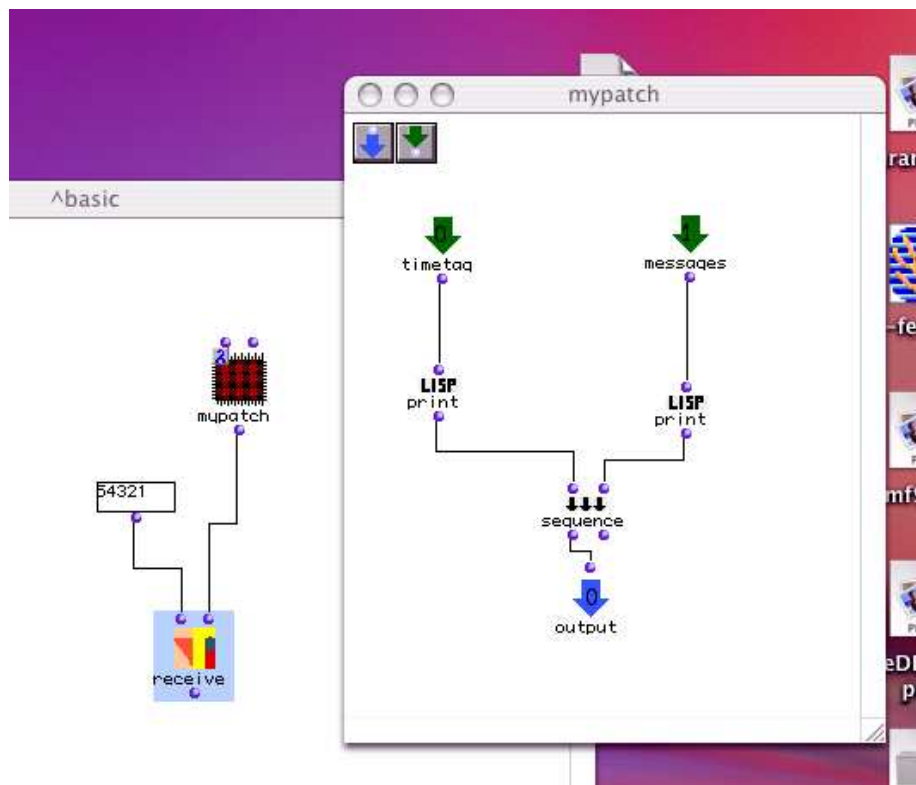


FIG. 16 – Réception d'un *bundle* sur le port `54321`. Le patch invoqué lors de la réception est affiché à côté.

dans cet état-là que les messages OSC seront effectivement reçus et passés au patch donné en paramètre.

Un exemple très simple : “Fast Feedback”. Afin d’illustrer la communication par OSC entre Max et OpenMusic, nous avons implémenté un exemple très simple, baptisé *Fast Feedback* (cf. figures 17 et 18, pages 39 et 40), et dont le principe est le suivant :

1. Max choisit aléatoirement une fréquence, et déclenche une sinusoïde pure à cette fréquence (patch [synth]). Puis, la fréquence est envoyée par OSC à OpenMusic (patch [sendToOM]).
2. OpenMusic reçoit cette fréquence par un message OSC et la multiplie par $\frac{2}{3}$. Le résultat est renvoyé immédiatement à Max dans un autre message OSC (patches ^renvoi et ^fast-feedback)
3. Max, en recevant le message d’OpenMusic, met la fréquence de la sinusoïde qu’il était en train de jouer à la nouvelle fréquence reçue (patch [receiveFromOM])
4. Un objet [timer] dans Max relance l’étape 1. à des intervalles réguliers (patch fast-feedback).

Cet exemple, simple à outrance, a pour principal mérite de donner un aperçu du temps de communication entre Max et OpenMusic par OSC, que nous avons évalué, sur une communication interne à notre propre machine, un iMac G4 cadencé à 1Ghz, à 0.25 ms environ. Des comparaisons avec d’autres outils, et notamment ceux fournis par le CNMAT pour tester les communications OSC (`sendOSC` et `dumpOSC`) nous ont montré que c’est l’implémentation d’OSC sous OpenMusic qui est responsable de la plupart des délais. Cependant, notre sentiment est qu’il est difficile de faire mieux que cela, étant donné que cette implémentation passe le plus clair de son temps à faire appel à des fonctions écrites en C de la bibliothèque *Open Transport*.

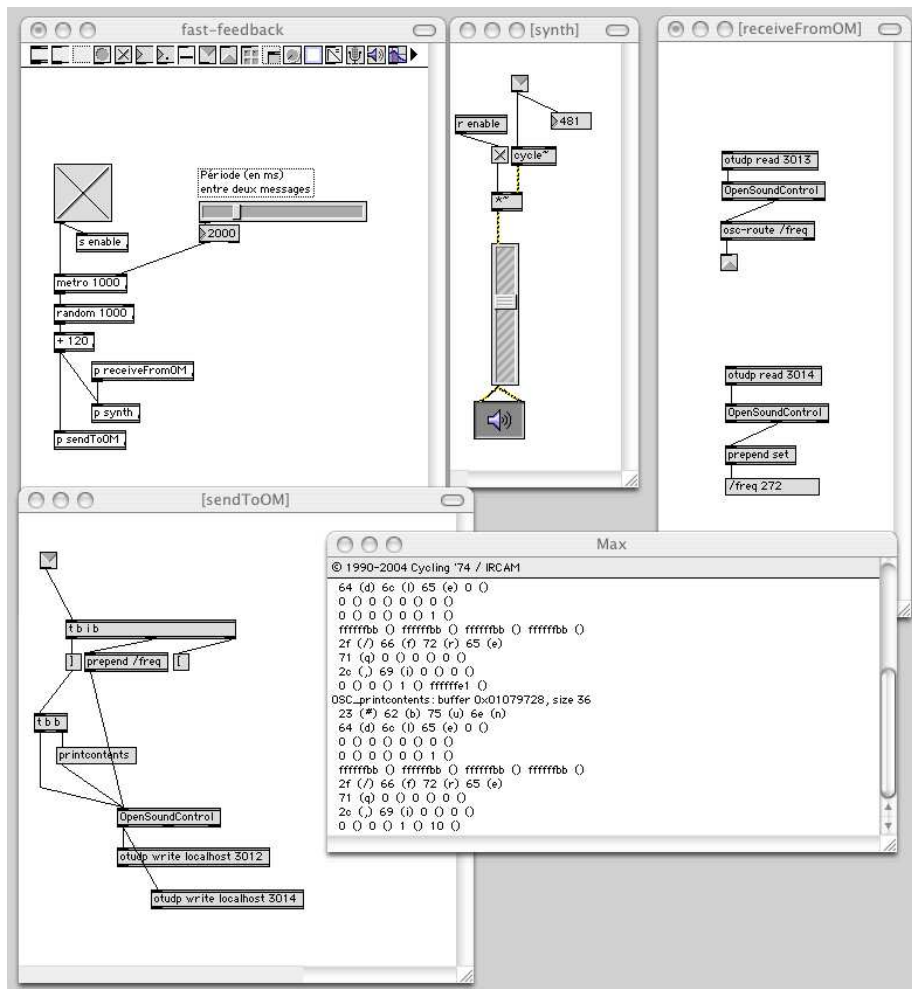


FIG. 17 – Partie Max du dispositif *Fast Feedback*.

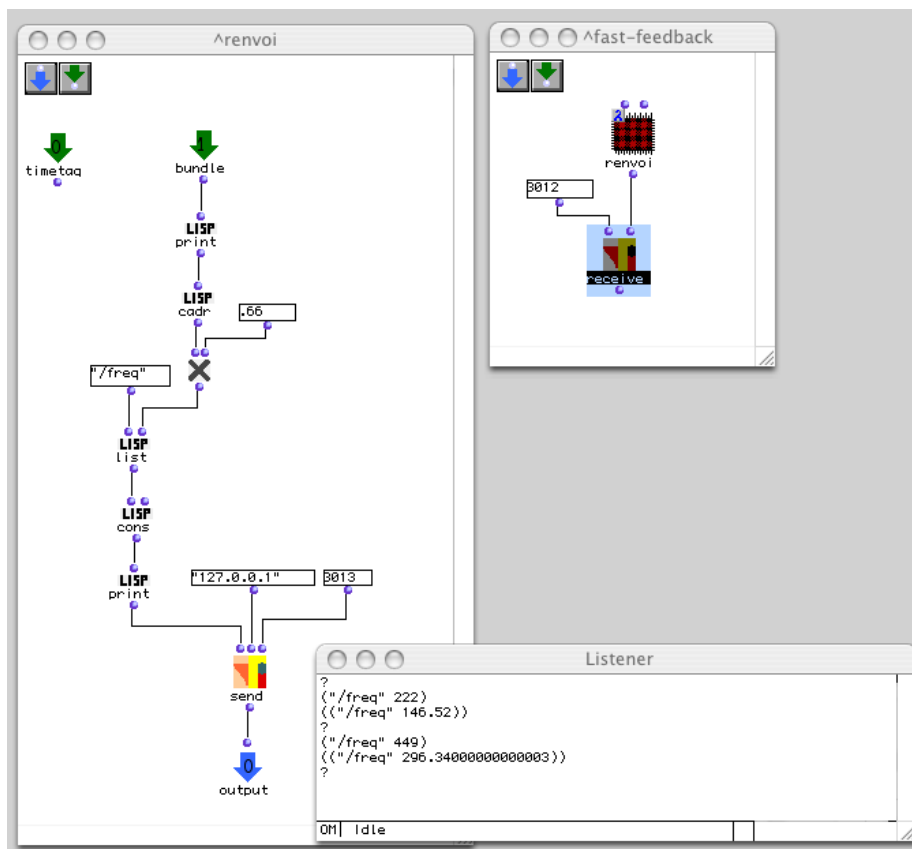


FIG. 18 – Partie OpenMusic du dispositif *Fast Feedback*.

4 Un exemple musical avec Georges Bloch

Afin de tester la validité de notre implémentation d’OSC et de mieux connaître les problématiques liées à l’utilisation d’OpenMusic dans un contexte temps-réel, nous avons tenu, avec Carlos Agon, à réaliser un dispositif musical qui tire parti des possibilités de Max et d’OpenMusic à la fois. C’est le compositeur Georges Bloch qui nous a fourni les idées et qui nous a aidé dans la réalisation de ce dispositif ; de nombreuses idées dans le paragraphe qui suit sont issues d’un texte de sa part intitulé *Du temps-réel au temps divin : la liaison Max-OM*.

4.1 Principe

Dans l’idée que nous propose Georges Bloch, il s’agit de fournir au musicien improvisant une “mémoire” de ce qu’il a joué, et de formuler des propositions pour la suite du jeu, en fonction d’une stratégie compositionnelle. La machine analyse les paramètres d’improvisation et calcule ses propositions grâce à sa mémoire de ce qui a été joué, information codée dans des structures musicales abstraites.

Segmentation et agrégation. Le jeu du musicien est découpé en segments par la détection de “respirations” (pause dans le jeu d’une durée de 100ms), et chaque segment est ainsi agrégé en accord. Ce type de segmentation convient bien pour les instruments mélodiques comme les instruments à vent. Cette méthode peut par ailleurs être raffinée pour ne pas être perturbée par les notes jouées *staccato* par exemple.

Analyse harmonique et analyse de fondamentale. Les paramètres de l’analyse que nous cherchons à extraire de l’improvisation sont la fondamentale Hindemith et la texture Estrada.

L’algorithme de détection de fondamentale virtuelle décrit par Paul Hindemith consiste à détecter dans un accord, la quinte la plus grave. S’il n’y a pas de quinte, l’algorithme sélectionne simplement la note la plus grave de l’accord.

L’analyse de texture, telle qu’elle a été définie par le compositeur mexicain Julio Estrada, tente de déterminer les intervalles présents dans l’accord, toutes les notes étant ramenées à l’octave, et indépendamment de leur ordre. Cette suite d’intervalles est ensuite réduite à une forme canonique dans laquelle les intervalles sont classés par ordre croissant et où leur somme vaut 12 (si on divise la gamme en 12 — cette analyse convient ainsi également pour les systèmes microtonaux de tempérament égal). Ainsi, la note simple a pour texture (12), l’accord de *do* mineur a pour texture (3 – 4 – 5) et la gamme chromatique a pour texture (1 – 1 – 1 – 1 – 1 – 1 – 1 – 1 – 1 – 1 – 1 – 1).

Si Max permet de déterminer par lui-même la fondamentale Hindemith et la texture Estrada d’un segment, la stratégie compositionnelle mise en œuvre derrière requiert une “vue d’en haut” de ce qui a été joué et fait appel à des structures musicales abstraites, ce qui nécessite d’utiliser OpenMusic.

Dimension temporelle. La composition exige une maîtrise temporelle : OpenMusic doit donc recevoir les informations de texture et de fondamentale accompagnées d’une étiquette temporelle, laquelle est déjà intégrée dans le protocole

OSC. De plus, afin de pouvoir prendre des décisions compositionnelles, il faut se doter de fenêtres temporelles d’analyse de plus ou moins grande échelle, chaque échelle étant un conteneur hiérarchique des échelles inférieures. Ces analyses à plus ou moins grande échelle nous permettront ensuite de délimiter des “régions” harmoniques.

Topologies. Les décisions sont prises à partir de critères topologiques sur ces données d’analyse, c’est-à-dire qu’une texture ou une fondamentale se voit associer une position, par laquelle il sera possible de déterminer des distances entre les données.

<i>do</i>	0
<i>sol</i>	1
<i>ré</i>	2
<i>la</i>	3
<i>mi</i>	4
<i>si</i>	5
<i>fa#</i>	6
<i>do#</i>	7
<i>sol#</i>	8
<i>ré#</i>	9
<i>la#</i>	10
<i>fa</i>	11

TAB. 1 – Le cycle des quintes et les valeurs de notes associées.

Le cycle des quintes que nous venons de citer en exemple convient bien pour les fondamentales Hindemith, en associant à chaque note son indice de position sur le cycle, comme le montre le tableau 1.

La distance entre deux textures Estrada est légèrement plus complexe à déterminer. Elle est la soustraction entre le nombre d’intervalles dans la texture la plus longue d’une part et le nombre d’intervalles communs entre les deux textures d’autre part. On a alors bien $dist(X, X) = 0$, quel que soit X . La position d’une texture peut alors être déterminée comme une distance par rapport à une texture de référence (une texture arbitraire, ou la texture la plus utilisée, ou la première texture utilisée, etc.)

Prise de décision. Finalement, un agrégat de données a pour un segment est constitué :

- d’une durée $d(a)$ de segment ;
- d’une valeur $h(a)$ de fondamentale Hindemith ;
- d’une valeur $e(a)$ de texture Estrada (calculée comme une distance par rapport à une texture de référence).

Il s’agit ensuite de déterminer si oui ou non, un groupe de données d’analyse forme une région ou non. Cette décision se fonde à la fois sur des critères topologiques et sur des critères temporels. Rappelons aussi qu’il s’agit de délimiter ces régions à plusieurs niveaux hiérarchiques. Nous allons détailler ci-après les critères de décision pour déterminer si un groupe G d’agrégats définit une région pour l’analyse de texture Estrada et pour un niveau hiérarchique de régions. Le même procédé pourra être appliqué pour les autres niveaux hiérarchiques et pour les fondamentales Hindemith.

Pour un niveau hiérarchique N de région, nous définissons $varMax(N)$ un seuil maximal de variance et $durMin(N)$ une durée minimum.

Nous calculons ensuite m la moyenne des valeurs de textures de chaque agrégat du groupe, pondérées par les durées des segments correspondants, et v la variance du groupe :

$$m = \frac{\sum_{a \in G} d(a) \times e(a)}{\sum_{a \in G} d(a)} \quad v = \frac{\sum_{a \in G} |m - d(a) \times e(a)|}{\sum_{a \in G} d(a)}.$$

Si $v < varMax(N)$ et $\sum_{a \in G} d(a) > durMin(N)$, alors on peut dire que ce groupe G définit une région de niveau hiérarchique N .

Affichage. En affichant des cycles des quintes répartis dans le temps, on peut avoir les résultats instantanés des analyses ainsi que des indications visuelles représentant les régions à différents niveaux hiérarchiques.

La fondamentale Hindemith s'affiche très naturellement sur une portée musicale.

4.2 Implémentation

Max, réception des notes et première analyse. Les informations entrent et sortent *via* Max : il reçoit les informations de jeu par le port MIDI et affiche à la fin la représentation graphique de la texture Estrada et de la fondamentale Hindemith.

Max fournit un objet nommé [borax] qui permet d'obtenir une foule d'informations sur les notes MIDI jouées au clavier. Cet objet, utilisé en conjonction avec un objet [timer] permet de détecter les respirations, c'est-à-dire les périodes sans note de plus de 100ms. Ce mécanisme est implémenté dans le patch [C_entrée_clavier2], qui envoie un "bang sur sa seconde sortie dès qu'une respiration est détectée.

La texture Estrada est déterminée par le patch [C_estrada], qui se contente de récupérer l'ensemble des notes MIDI jouées dans un tableau (l'objet [coll]) et, lorsqu'il reçoit un "bang de respiration calcule la texture en calculant la liste des intervalles entre les notes (cette liste peut être de longueur variable, mais la somme de ses éléments fait 12).

La fondamentale Hindemith est légèrement plus difficile à calculer ; le patch [hindemith_fixe] et son sous-patch [loop_fixe] sont en conséquence plus complexes, ce dernier appliquant un algorithme itératif de recherche de minimum pour déterminer la fondamentale qui correspond le mieux à l'accord courant.

Une fois la fondamentale Hindemith et la texture Estrada calculées, un message est créé pour être envoyé à OpenMusic par OSC. Ce message est composé de la manière suivante :

- Une date d'envoi du message ;
- une date de réponse attendue (celle-ci n'est pas prise en compte à l'heure ou nous écrivons ces lignes) ;
- le numéro de note⁵ de la fondamentale Hindemith ;
- la liste des valeurs composant la texture Estrada de l'accord.

⁵Comme dans la plupart des cas, il s'agit d'un numéro de note MIDI.

OpenMusic, prise de décision et représentation graphique. OpenMusic se charge ensuite de faire les calculs de distance Estrada et de calculer les données d’analyse pour déterminer les régions (cf. figure 22⁶). En plus de ces analyses, décrites dans les paragraphes précédents, l’implémentation (qui, malheureusement, ne sera pas terminée au moment de conclure ce mémoire) calcule un vecteur de probabilités de passer d’une tonalité à une autre : le vecteur, affiché dans le coin inférieur droit de la figure 23 (page 49), dont les valeurs initiales sont arbitraires et évoluent au cours du jeu pour refléter les choix passés du musicien, montre 12 valeurs qui sont, dans l’ordre, les probabilités de passer dans les tonalités des 12 demi-tons au-dessus de celle dans laquelle on se trouve actuellement. Ainsi par exemple, si l’analyse détecte une tonalité en *fa*, la quatrième valeur du vecteur donnera la probabilité de passer en *la*, si le musicien se conforme aux probabilités initiales et à ses choix passés.

Il crée également, grâce à ses fonctionnalités de représentations musicales, les images qui serviront à montrer la progression de l’analyse. Ces images sont sauvegardées dans un fichier, dont le nom est renvoyé à Max.

Retour dans Max : affichage des résultats d’analyse. Le nom du fichier ainsi généré est renvoyé à Max, qui peut ainsi l’afficher. Ce dernier utilise Jitter (et en particulier l’objet [jit.pwindow]) pour charger et afficher ces images. Dans certains patches non encore implémentés à l’heure où nous écrivons ces lignes, des objets Jitter permettant de dessiner des polygones 3D par OpenGL sont utilisés pour tracer en 3D le “tunnel du temps” (expression de Georges Bloch), dont nous donnons un aperçu en figure 24.

⁶Le patch de la figure 22 utilise une version plus ancienne de la bibliothèque Moscou, où les patches destinés à recevoir les messages OSC ne recevaient pas les étiquettes temporelles et n’avaient donc qu’une entrée, contrairement à ce qui est expliqué dans le § 3.2, p. 36.

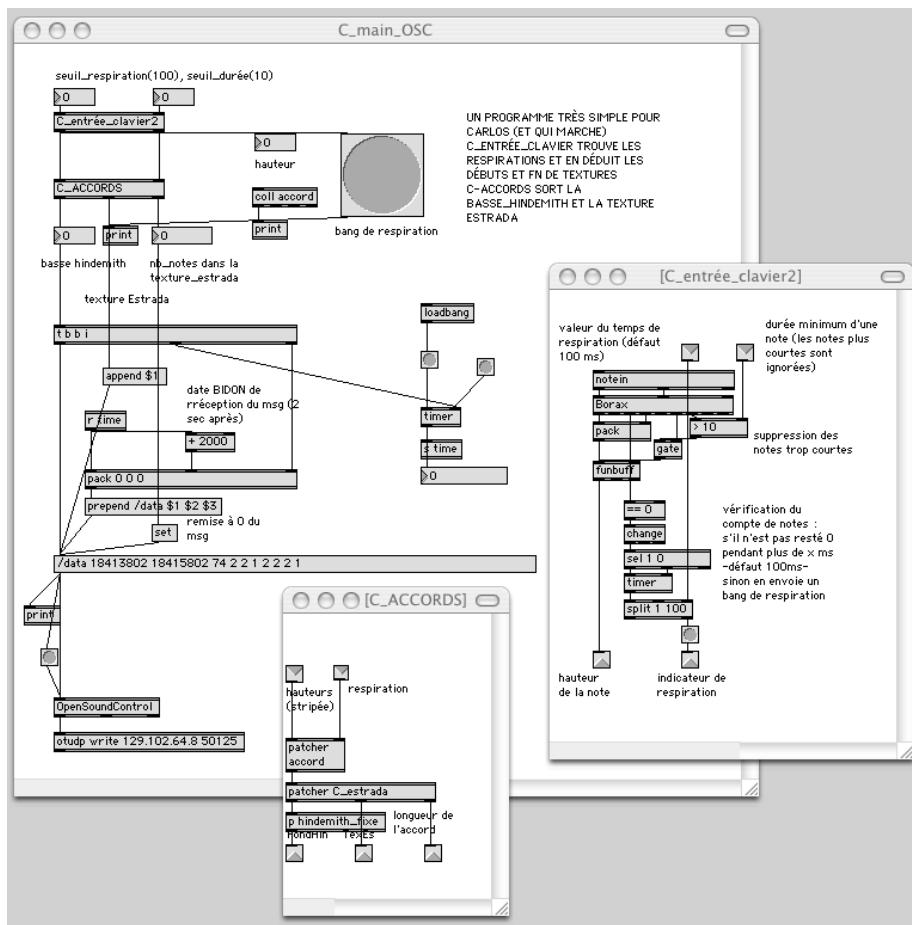


FIG. 19 – Patch Max principal et mécanisme de détection de respiration.

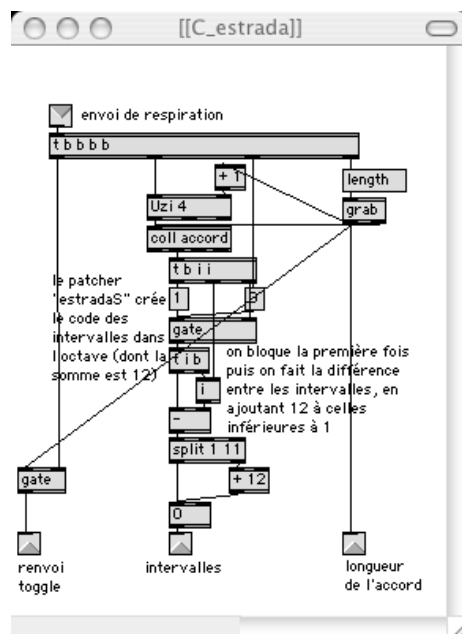


FIG. 20 – Patch Max en charge de la détermination de la texture estrada.

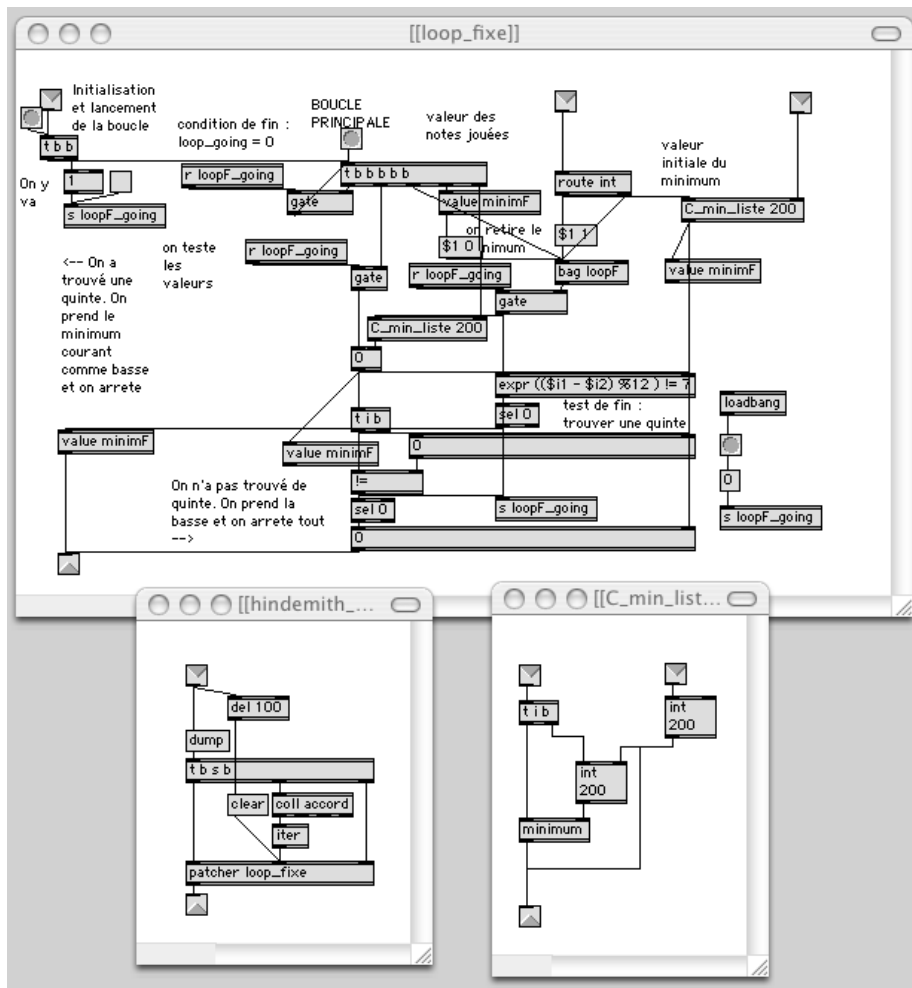


FIG. 21 – Patches Max en charge de la détermination de la fondamentale Hindemith.

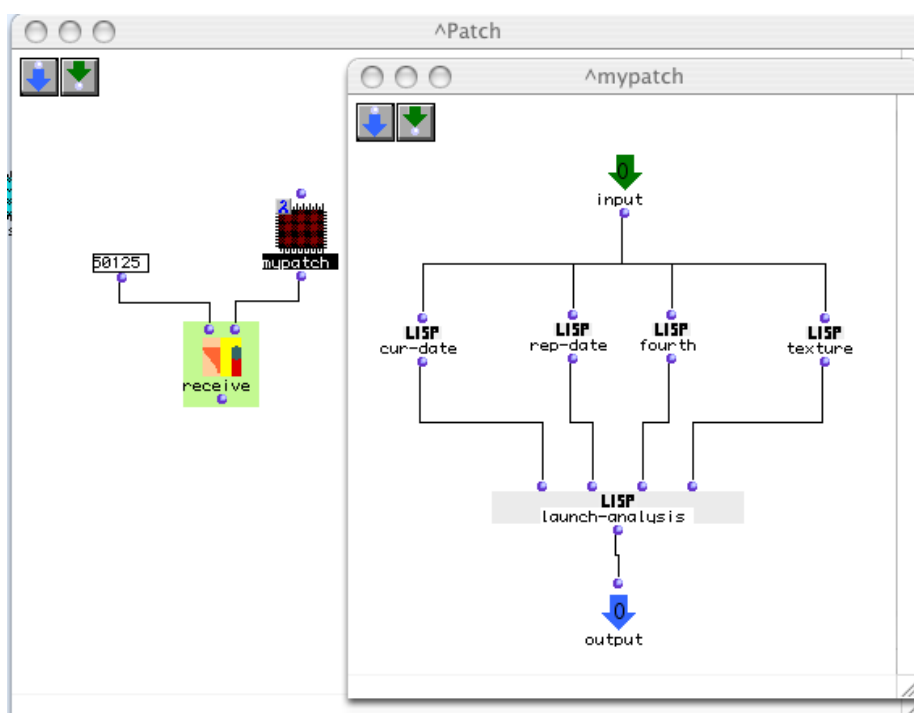


FIG. 22 – Patch OpenMusic du dispositif de Georges Bloch.

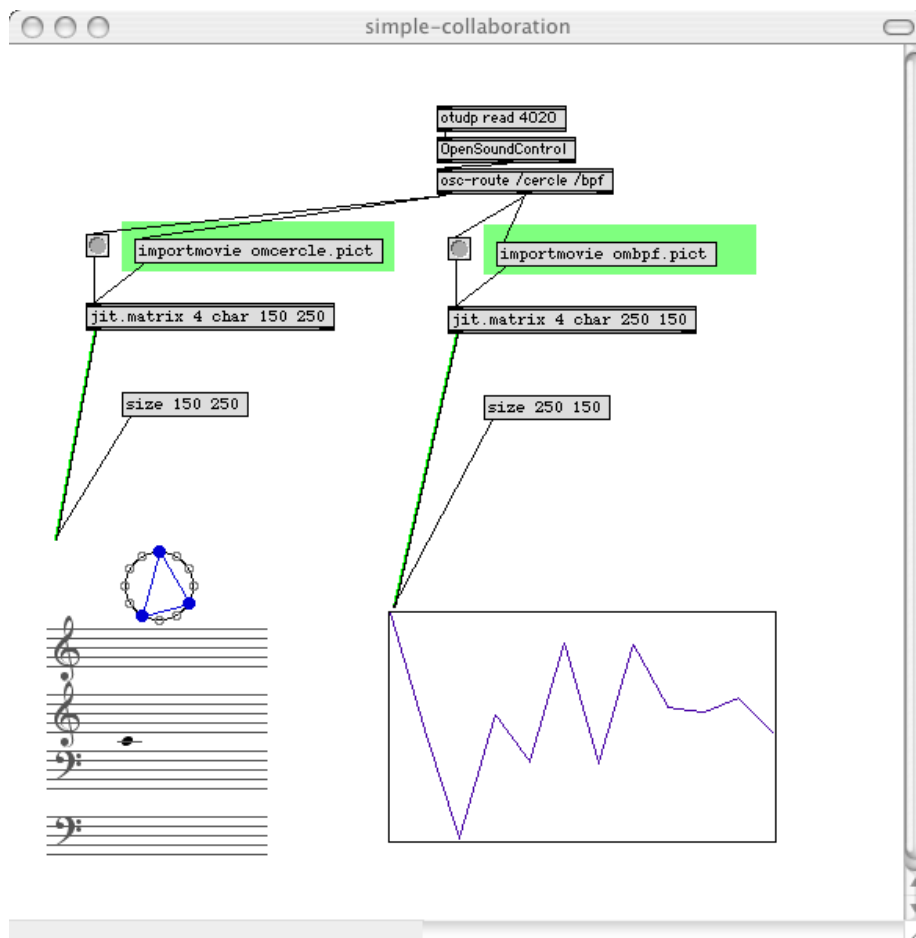


FIG. 23 – Patch Max d’affichage des informations de texture et de fondamentale.

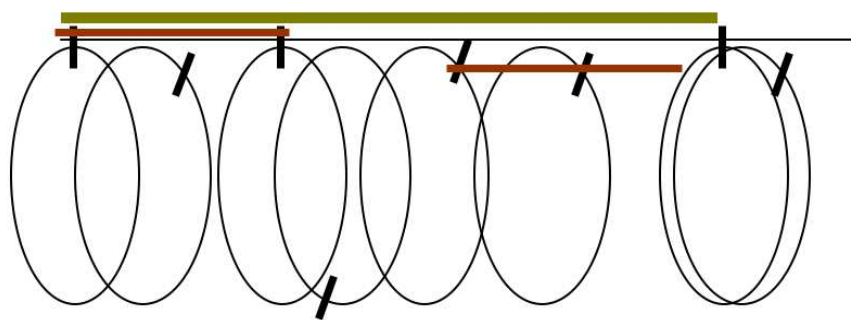


FIG. 24 – Schéma du “tunnel du temps” affichant les régions harmoniques à l’utilisateur pendant qu’il joue. La ligne horizontale fine représente la moyenne des régions. La ligne verte (la plus haute sur la figure) et les deux lignes rouges représentent les régions à deux échelles différentes.

Conclusion

Travail réalisé

Par notre travail, nous espérons ainsi avoir apporté quelques idées utiles à la communauté de la recherche en informatique musicale dans le domaine de l'interaction entre temps-réel et temps différé, dont nous donnons, dans la première partie de ce document et sans être exhaustifs, un petit aperçu des besoins de cette interaction, des problèmes qu'elles soulevait et des limites des solutions existantes ; nous nous sommes pour cela entretenus avec Philippe Manoury.

Notre principale réalisation a été de définir un modèle formel de la partie contrôle de Max. Nous avons étudié brièvement la caractérisation de Max comme un langage de programmation, puis celle du patch Max comme un système réactif, mais nous avons surtout défini un formalisme avec sa syntaxe et sa sémantique ; nous avons en outre modélisé le passage du temps et le mécanisme d'ordonnancement temporel, nous avons établi deux grandes familles d'objets remarquables, puis nous avons listé les primitives de ce modèle et introduit un lien vers OpenMusic au moyen d'une primitive [lisp]. Ce modèle formel a été pour nous l'occasion de proposer un ensemble de notations et de termes qui permettraient, nous l'espérons, aux utilisateurs de Max et aux chercheurs qui s'y intéressent d'avoir enfin un langage commun et un cadre formel pour décrire leurs travaux.

Nous avons implémenté la bibliothèque *Moscou* pour OpenMusic, permettant ainsi l'envoi et la réception de messages par OSC dans cet environnement.

Cette bibliothèque nous a permis de concrétiser un exemple musical proposé par Georges Bloch, et dont nous avons détaillé les principes et l'implémentation dans ce document.

Perspectives d'avenir

Nous espérons que la communauté donnera suite à nos travaux, ne serait-ce que parce que nous savons que bien des choses peuvent être améliorées. Ainsi, par exemple, notre modèle de Max ne prend pas en compte le mécanisme de messages, pourtant bien connu des utilisateurs de Max. Notre modèle, intentionnellement, se montre assez pauvre en primitives, car nous avons tenté d'isoler les composants les plus indispensables. Mais de ce fait, notre modèle manque de capacités de constructions de listes (qui donneraient accès à la création de messages), et ne fait pas référence aux objets graphiques de Max, qui sont une source importante de données.

Nous sommes cependant enthousiasmés par les classes remarquables que nous avons pu isoler, et par l'intégration, grâce à ces classes, d'un procédé d'appel de fonction externe en Lisp. Nous espérons que cette façon de procéder sera généralisée à d'autres langages de scripts ; elle pourrait permettre de définir des interfaces entre Max et de nombreux langages dynamiques comme Perl ou Python, donnant ainsi à ces langages procéduraux une "enveloppe" réactive.

La bibliothèque *Moscou* est encore incomplète, bien que suffisante pour la plupart des tâches. Il reste encore à implémenter le "routage" OSC, mécanisme par lequel on peut sélectionner automatiquement les messages à traiter avec l'aide de modèles de textes. En outre, des tests de performances pourraient révéler les points faibles de la bibliothèque, qui se révèle parfois lente pour

l'envoi de données nombreuses.

Enfin, nous espérons voir un cadre théorique pour l'interaction entre temps-réel et temps différé se développer, afin de bénéficier aux compositeurs qui voudraient s'y essayer mais qui se trouvent bloqués par l'inefficacité ou l'inélégance des procédés mis en œuvre.

Nous ne pouvons savoir aujourd'hui quelles idées viendront demain aux compositeurs, et à quels genres nouveaux d'interactions ils voudront s'essayer. C'est notre souhait que les outils et les fondations théoriques développés autour de cette collaboration entre temps-réel et temps différé prenne son essor, afin de rallier deux mondes fortement complémentaires qu'on a longtemps vu comme incompatibles.

Références

- [1] Carlos Augusto Agon Amado. *OpenMusic, un Langage Visuel pour la Composition Assistée par Ordinateur*, chapter 2. Représentations Musicales, Ircam, 1998.
- [2] C. André, H. Boufaïed, and S. Dissoubray. Synccharts : un modèle graphique synchrone pour systèmes réactifs complexes, janvier 1998. Real-Time Systems 1998.
- [3] Charles André. Synccharts : A visual representation of reactive behaviors, April 1996.
- [4] *Networking with Open Transport*. Apple Computing, Inc., November 1997. URL <http://developer.apple.com/documentation/mac/NetworkingOT/NetworkingWOT-2.html>.
- [5] G. Assayag, Dubnov, and Delerue. Guessing the composer's mind : applying universal prediction to musical style. In *Proceedings of the International Computer Music Conference, 1999*.
- [6] Gérard Assayag and Carlos Agon. OpenMusic architecture. In *ICMC 96*, Hong Kong, 1996.
- [7] Gérard Assayag and Camilo Rueda. Représentations musicales. *Revue Intermedia*, septembre 1993.
- [8] Gérard Berry. Real-time programming : Special purpose or general purpose languages. *Rapports de recherche INRIA*, 1989.
- [9] Gérard Berry. The foundations of Esterel, 1997.
- [10] Gérard Berry, Philippe Couronné, and Georges Gonthier. Systèmes réactifs et programmation synchrone. *Rapports de recherche INRIA*, 1986.
- [11] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language : Design, semantics, implementation. 1988.
- [12] R. Braden. RFC 1122 : Requirements for Internet Hosts — Communication Layers, October 1989. URL <http://www.ietf.org/rfc/rfc1122.txt>.
- [13] Kristine H. Burns. *Algorithmic Composition : a Definition*. Florida International University, 1997.
- [14] David Cope. *New Directions in Music*. Waveland Press, Inc., 2000.
- [15] S. Letz D. Fober. Les logiciels d'aide à la composition musicale. In Grame, editor, *Actes des Rencontres Musicales Pluridisciplinaires*, pages 13–19, 1999.
- [16] François Déchelle, Riccardo Borghesi, Maurizio De Cecco, Enzo Maggi, Butch Rovin, and Norbert Schnell. jMax : An environment for real-time musical applications. *Computer Music Journal*, 23(3) :50–58, Fall 1999.

- [17] François Déchelle, Maurizio de Cecco, Enzo Maggi, and Norbert Schnell. jmax recent developments. In *ICMC : International Computer Music Conference*, Pekin, Chine, Septembre 1999.
- [18] P. Desain and H. Honing. The quantization problem : traditional and connectionist approaches. In M. Balaban, K. Ebcioglu, and O. Laske, editors, *Understanding Music with AI : Perspectives on Music Cognition*, pages 448–463. MIT Press, 1992.
- [19] Peter Desain and Henkjan Honing. Putting Max in perspective. *Computer Music Journal*, 17(2) :3–11, 1993.
- [20] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9) : 1305–1320, September 1991.
- [21] Peter Hanappe. *Design and implementation of an integrated environment for music composition and synthesis*. PhD thesis, Université de Paris VI, 1999.
- [22] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.
- [23] Bernard Houssais. The synchronous programming language SIGNAL — a tutorial, April 2002. IRISA. ESPRESSO Project.
- [24] David François Huynh. *Mystique : An imperative reactive language*. 2003.
- [25] M. Laurson. Patchwork, a visual programming language and some musical applications. In *Studia Musica*, number 6, Helsinki, 1996.
- [26] Luc Léonard and Guy Leduc. A formal definition of time in lotos. *Formal Aspects of Computing*, 1998.
- [27] Eric Lindemann, Michel Starkier, and François Déchelle. The ircam musical workstation : Hardware overview and signal processing features. In Stephen ARNOLD et Graham HAIR, editor, *ICMC : International Computer Music Conference*, Glasgow, Ecosse, Septembre 1990. URL <http://mediatheque.ircam.fr/articles/textes/Lindemann90a>.
- [28] H. C. Longuet-Higgins. *Mental Processes. Studies in Cognitive Science*. MIT Press, 1987.
- [29] John A. Maurer IV. A brief history of algorithmic composition. <http://ccrma-www.stanford.edu/~blackrse/algorithm.html>, winter 1999.
- [30] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. <http://www-formal.stanford.edu/jmc/recursive.html>, April 1960.
- [31] *The Complete MIDI 1.0 Detailed Specification*. MIDI Manufacturers Association, November 2001.

- [32] Yann Orlary, Dominique Fober, and Stephane Letz. Syntactical and Semantical Aspects of Faust. *Soft Computing*, 2004.
- [33] Stephen Travis Pope. Fifteen years of computer assisted composition. <http://www.cnmat.berkeley.edu/~stp/>, 1995.
- [34] J. Postel. RFC 768 : User Datagram Protocol, 1980. URL <http://www.ietf.org/rfc/rfc768.txt>.
- [35] Miller Puckette. The patcher. In *ICMC 88*, 1988.
- [36] Miller Puckette. Pure data : another integrated computer music environment. In *Second Intercollege Computer Music Concerts*, pages 37–41, 1996.
- [37] Miller Puckette. Max at seventeen. *Computer Music Journal*, 26 :4 :31–43, Winter 2002.
- [38] Mark J. Steedman. A generative grammar for jazz chord sequences, 1984.
- [39] Guy L. Steel Jr. *Common Lisp, The language*. Digital Press, second edition edition, 1990.
- [40] R. D. Tennent. Introduction to reactive systems, 2000.
- [41] Matt Wright. Opensound control specification v1.0, April 2004. URL <http://www.cnmat.berkeley.edu/OpenSoundControl/OSC-spec.html>.
- [42] Iannis Xenakis. *Formalized Music. Thought and Mathematics in Composition*. Indiana University Press, 1971.
- [43] S. Letz Y. Orlarey, D. Fober. Elody : a java+midishare based music composition environment. In ICMA, editor, *Proceedings of the International Computer Music Conference*, pages 391–394, 1997.
- [44] David Zicarelli, Gregory Taylor, Jeremy Bernstein, Adam Schabtach, and Richard Dudas. *Max 4.3 Tutorials and Topics Manual*. Cycling '74, 2003.
- [45] David Zicarelli, Gregory Taylor, Joshua Kit Clayton, Jhno, and Richard Dudas. *Max 4.3 Reference Manual*. Cycling '74, 2003.