# OMClouds, petits nuages de contrainte dans OpenMusic

Charlotte Truchet, Gérard Assayag, Philippe Codognet

# OMClouds, petits nuages de contraintes dans OpenMusic

Charlotte Truchet, Gérard Assayag, Philippe Codognet
IRCAM, 1 place Igor Stravinsky, Paris, France
LIP6, Université de Paris 6, 8, rue du Capitaine Scott, Paris, France
{truchet, assayag }@ircam.fr, Philippe.Codognet@lip6.fr

April 1, 2003

**Abstract**

Cet article décrit OMClouds, une librairie OpenMusic. OMClouds permet de modéliser et résoudre visuellement des problèmes de contraintes dans OpenMusic. Elle a été réalisée d'après une collaboration avec des compositeurs sur des problmes musicaux. Elle utilise un algorithme de recherche locale, la recherche adaptative, bien adapté à une utilisation musicale. OMClouds donne la possibilité d'écrire visuellement un CSP, et surtout d'éditer des résultats partiels ou approchés pendant la résolution.

**Mots clefs** Composition assistée par ordinateur, programmation par contraintes, OpenMusic, recherche locale, recherche adaptative

## 1 Introduction

Cet article présente OMClouds, une librairie pour la programmation par contraintes pour la composition assistée par ordinateur (CAO). OMClouds est une librairie du logiciel OpenMusic, langage de programmation visuelle, fonctionnelle et objet pour la CAO développé à l'IRCAM par Gérard Assayag et Carlos Agon (voir [1] et [2]).

La Programmation par Contraintes (PLC) est un paradigme de programmation qui connaît depuis quelques années un succès certain. Une contrainte est tout simplement une relation logique portant sur un ensemble de variables (inconnues d'un problème, variant dans un domaine précis). Cela permet de spécifier une information partielle sur les valeurs que les variables peuvent prendre, "contraignant" ainsi cet espace de valeurs. L'ensemble variables, domaines, contraintes forme un CSP, Constraint Satisfaction Problem. L'idée de base de la programmation par contraintes est d'intégrer ce concept de contrainte dans un langage de haut-niveau pour pouvoir programmer avec cette notion d'information partielle. Une fois le CSP écrit dans un langage adéquat (modélisation), il est passé à un solver qui se charge d'affecter aux variables des valeurs satisfaisant les contraintes (résolution). La PLC trouve naturellement une place en

musique, comme le montrent les nombreux travaux sur l'harmonisation automatique ([7], [3], [15] ou [12]). En composition contemporaine, il existe déjà deux solvers intégrés à OpenMusic, PWConstraints [9] et Situation [13], basés sur une exploration systématique des domaines (backtracking avec forward checking).

Pour concevoir OMClouds, nous avons commencé par étudier une douzaine de problèmes musicaux posés par des compositeurs liés à l'IRCAM, et trois problèmes d'analyse musicale en coopération avec Marc Chemillier [4]. Ces problèmes ont été modélisés et résolus en collaboration avec les compositeurs. Par exemple, nous avons étudié un problème de classement d'accords : à partir d'une suite d'accords fixée, il faut permuter ces accords de manière  maximiser le nombre de notes communes d'un accord à l'autre. Une autre problème, dû à Gilbert Nouno et Mickael Jarrell (dans Droben, pour contrebasse, ensemble et électronique), portait sur des gestes musicaux : une note de départ étant donne, il faut trouver une suite d'intervalles qui permette de rester dans une échelle fixée, avec des contraintes de cadrinalité sur les intervalles choisis. Mauro Lanza a proposé un problème portant sur des rythmes asynchrones (dans Aschenblume, pour ensemble, ou Burger Time, pour tuba et électronique). Fabien Lévy a utilisé un problème de génération d'accords sur un domaine continu de frquences, avec des contraintes fixant le nombre de notes communes d'un accord à l'autre (dans Coïncidences, pour orchestre). Georges Bloch (dans Empreinte sonore de la fondation Bayeler) a proposé un problème d'harmonisation automatique de canons rythmiques, où il fallait minimiser une certaine distance d'un accord à l'autre, et minimiser une deuxième distance portant sur les fondamentales virtuelles des accords. Nous ne décrivons ici ces CSPs que de manière très succinte.

En revanche, il est important de détailler les conclusions à tirer sur la structure générale d'un problème de contraintes en composition contemporaine, qui ont servi à définir l'architecture de OMClouds.

Il faut d'abord souligner la grande variété des problèmes rencontrés, du point de vue des structures musicales utilisées : accords, notes, rythmes, tempi, harmonie, gestes. D'un point de vue informatique, ce sont toujours des domaines finis. Mais d'un point de vue musical, cela signifie que l'on doit pouvoir modéliser n'importe quel objet musical. Par ailleurs, si dans certains problèmes les variables sont directement des objets musicaux visibles sur la partition (notes, accords par exemple), dans d'autres elles sont en amont de la notation musicale, ie il faut un calcul entre les variables et le résultat musical que l'on peut éditer ou jouer (ceux de Georges Bloch ou Gilbert Nouno par exemple). Pour ces deux raisons, nous nous abstenons de donner un sens musical aux variables, elles sont simplement considérées comme des nombres entiers.

Deuxièmement, les contraintes sont également très variées : on trouve bien sûr des contraintes arithmétiques, des égalités et inégalités, mais aussi des contraintes plus difficiles à traiter comme des alldifferent (les valeurs doivent toutes tre diffrentes deux  deux), ou encore des contraintes de haut niveau comme des contraintes analogues à des contraintes de capacité ou bien des contraintes de cardinalité. Inutile donc d'espérer optimiser le solver pour un type de contraintes particulier. Quelque soit la méthode choisie, elle doit pouvoir traiter cette variété dans le langage de contraintes

: il faut prévoir des fonctions de filtrage pour un solver complet ou des fonctions de coût pour un solver par recherche locale. En revanche on constate que pour la majorité des problèmes rencontrés, les contraintes sont homogènes sur les variables, elles portent de manière identique sur toutes les variables. Plus précisément, elles sont souvent posées entre deux variables successives, plus rarement avec un ordre supérieur à un (d'une variable aux deux suivantes, aux trois suivantes, etc).

Troisème point, le but lui-même varie : on peut avoir des problèmes d'optimisation, de résolution (trouver une solution ou quelques unes), génération (trouver toutes les solutions). Souvent les problèmes (celui Mauro Lanza par exemple) sont même surcontraints, et le but est alors de trouver une solution approchée.

Dernière remarque, le but n'est pas toujours de résoudre le CSP. En informatique, un solver de contraintes est utilisé pour donner une ou des solutions. Ce principe, problème $\rightarrow$ solution, ne s'applique pourtant pas vraiment dans notre cas. D'abord, dans le cas des CSPs surcontraints, une stricte résolution consisterait à répondre "non". Ce n'est évidemment pas satisfaisant. Mais plus généralement, d'une part les solutions des CSPs en CAO sont toujours réécrites par le compositeur en fonction de critères esthétiques non formalisables, et d'autre part une solution approchée mais "qui sonne bien" sera préférée à une solution exacte "qui ne sonne pas". Pour citer Mauro Lanza, qui a utilisé des solutions approchées à son problème sur les rythmes, "Dans 90 pour cent des cas, la solution avec le moins d'erreurs etait la bonne". Il faut donc, plutôt que résoudre classiquement le CSP, considérer le solver comme faisant à la demande des propositions de solutions, éventuellement partielles ou approchées.

# 2 Recherche adaptative

## 2.1 Principe général

Cette méthode est due à Philippe Codognet et Daniel Diaz [5]. Elle fait partie des techniques de recherche locale, qui ont depuis une dizaine d'années largement prouvé leur efficacité (voir [14], ou [8]). Le principe en recherche locale est de parcourir l'espace de recherche de manière non exhaustive, en se guidant par une mesure de la qualité de la configuration courante. On peut résumer grossièrement ce type d'algorithme par : intialisation aléatoire, puis itérativement exploration d'un voisinage, recherche d'une meilleure configuration, remplacement. Cela suppose d'avoir une mesure de la qualité de la configuration courante, ce qui est fait en représentant les contraintes par une fonction de coût, qui sert à la recherche d'une meilleure configuration.

L'algorithme de recherche adaptative fonctionne sur ce principe, mais en affinant la notion de coût. Il s'agit de tirer le maximum d'information à partir des contraintes, au niveau de chaque variable et non de la configuration globale. On utilise pour cela une projection des coûts sur chaque variable (la méthode la plus simple consistant à prendre les coûts des contraintes où la variable figure). Cela permet de sélectionner à chaque itération la variable la plus mauvaise. Nous remplaçons l'étape "explo-

ration du voisinage" par le calcul des coûts de chaque variable, la sélection de la plus chère, et l'exploration du domaine de cette variable pour trouver une meilleure valeur. Cette méthode ne se limite pas à un type particulier de contraintes, mais il faut pour chaque contrainte définir une fonction de coût associée, qui mesure à quel point la contrainte est violée pour une configuration donnée (et vaut évidemment 0 si la contrainte est satisfaite). On peut prendre par exemple pour une contrainte arithmétique $|X - Y| \leq C$ la fonction $max(0, |X - Y| - C)$.

Pour éviter d'être pris dans un minimum local, on ajoute une mémoire adaptative à la manière de la recherche tabu : quand une variable mène à un minimum local (aucune configuration du voisinage ne permet d'améliorer le coût), elle est marquée Tabu et ne peut être sélectionnée pendant un certain nombre d'itérations.

D'un point de vue informatique, la recherche adaptative a montré son efficacité sur un ensemble de benchmarks sur les problèmes classiques des carrés magiques et des N-reines. Les résultats sont comparés à Localizer [10, 11], qui propose un langage général de définition d'heuristiques en recherche locale pour des problèmes de Recherche Oprationnelle ou des CSPs. La recherche adaptative donne des résultats très satisfaisants (voir [6] pour l'ensemble des benchmarks).

## 2.2 Algorithme

On notera $\epsilon$ le seuil d'arrêt, normalement fixé à 0, $l_{tabu}$ la longueur de la liste Tabu, $V_+$ la plus mauvaise variable (pour une itération).

1. Initialisation aléatoire

   *Repeat*

2. *Calcul* des coûts de toutes les variables, sauf celles marquées Tabu. Sélection de la plus chère, $V_+$.
3. *Test* du coût global en remplaçant $V_+$ par toutes les valeurs de son domaine. Sélection de $v'$ la meilleure.
4. *Si* à la fin de l'exploration du domaine, aucune valeur n'améliore le coût global, *alors* $V_+$ est marquée tabu pendant $l_{tabu}$ itérations. *Sinon*, $V_+$ est instanciée à $v'$.
5. *Si* toutes les variables sont tabu *alors* réinitialisation aléatoire.

   *Until* l'erreur globale est inférieure à $\epsilon$.

## 2.3 Application musicale

Cette méthode répond particulièrement bien à notre cahier des charges. D'abord, les variables sont toujours instanciées, et la valeur des instanciations progresse jusqu'au minima locaux. On peut donc toujours renvoyer des résultats partiels (soit les minima locaux, soit le meilleur minimum local trouvé, soit l'instanciation courante). Cela permet d'utiliser la PLC non plus seulement pour résoudre un problème, mais aussi pour donner à la demande des solutions éventuellement partielles ou approchées. Ce principe est exploité dans OMClouds pour éditer des résultats partiels.

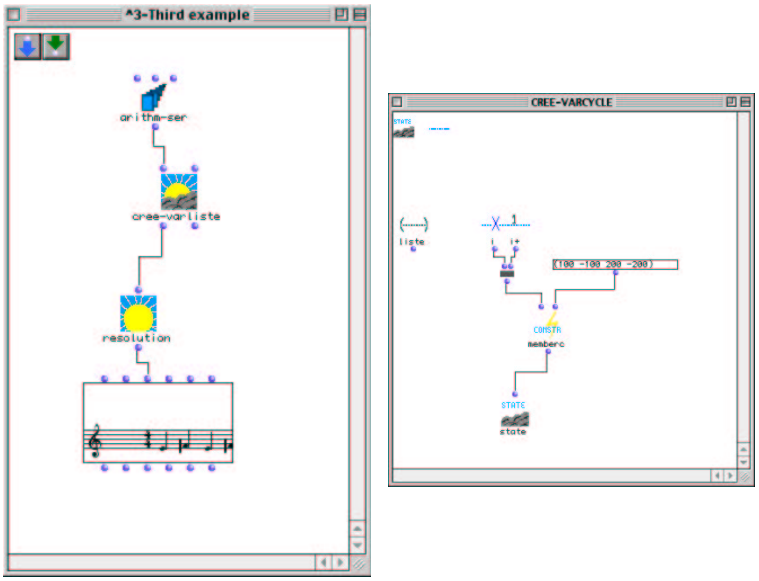


Figure 1: Un problème de contraintes dans OMClouds. L'instance d'asvarliste est créée par la boîte cree-varliste, avec en entrée le domaine, ici une liste de valeurs midi, et le nombre de variables. La définition des contraintes, à droite, est faite dans le patch associé à cette boîte. Le asvarliste est ensuite passé à la boîte résolution, qui donne une solution en sortie.

Il traite naturellement les solutions approchées. Dans notre implémentation, la condition d'arrêt est que la somme des coût pour toutes les variables soit inférieure à un nombre fixé à $\epsilon$. On obtient les solutions exactes en prenant $\epsilon = 0$, et des solutions approchées pour $\epsilon > 0$.

Par ailleurs, la représentation des contraintes par des coûts apporte une souplesse supplémentaire au programme. On peut très simplement donner plus d'importance à certaines contraintes, et inversement laisser une tolérance sur d'autres. Il suffit de pondérer leurs fonctions de coût, voire de jouer avec la condition d'arrêt en modifiant $\epsilon$.

# 3 OMClouds

OMClouds est une librairie intégrée dans la version 4.6.5 d'OpenMusic, disponible au forum d'avril 2003.

## 3.1 Modélisation du problème

Les problèmes sont définis grâce à des classes *ad hoc*, qui contiennent l'ensemble des données nécessaires à la modélisation et à la résolution. OMClouds fournit trois sortes de CSPs :

1. **liste** les variables sont placées simplement dans une liste

2. **cycle** les variables sont toujours placées dans une liste, mais on considère qu'elles forment un cycle, donc la dernière variable a pour suivante la première variable. Cela revient à considérer les indices modulo le nombre de variables.
3. **permutation** Le domaine de valeurs possibles est l'ensemble des permutations sur les variables. Dans la résolution, on change l'étape "exploration du domaine de $V_+$" par "exploration quand on permute $V_+$ avec chaque autre variable".

Pour pouvoir modéliser les trois sortes de problèmes, on a en fait trois classes : asvarliste, asvarcycle, et asvarperm. Dans la suite, on détaillera la modélisation pour asvarliste. L'architecture d'OMClouds est semblable pour les trois classes.

Leurs slots sont :

1. valeurs les variables étant toujours instanciées, on stocke ici l'état courant du système
2. domaine qui contient le domaine des variable (sauf pour asvarperm)
3. erreurs stocke les erreurs variables par variable
4. erreur-global stocke l'erreur globale de la configuration (somme des erreurs)
5. tabu stocke la liste des variables marquées tabu, et le nombre d'itérations avant qu'elles ne sortent
6. longueur nombre de variables
7. contraintes contient la définition des fonctions de coût pour les variables
8. cont-glob contient la fonction de coût pour l'ensemble de la configuration, somme des contraintes

OMClouds permet de créer visuellement, à la manière d'OpenMusic, des instances de ces classes pour modéliser un problème musical, avec les fonctions cree-varliste, cree-varperm et cree-varcycle. Elles sont représentées visuellement par des boîtes particulières, instances de la classe asbox, qui ont notamment un patch associé pour la définition des contraintes. Les deux entrées représentent les domaines et le nombre de variables. Ces boîtes ont deux sorties, la première renvoie une instance de la classe concernée, la deuxième peut être utilisée pendant la résolution et renvoie uniquement les valeurs courantes des variables.

## 3.2 Définition des contraintes

Les contraintes sont définies dans le patch (classe OM varpanel) attaché à la boîte asbox. Ce patch est le même pour les trois types de CSP (même si bien sûr les contraintes s'appliquent différemment). Il contient plusieurs entrées et sorties prédéfinies, qui sont des sous-classes des classes input et output d'OpenMusic.

L'entrée var-i est unique. Elle représente la variable courante, nous noterons $i$ son indice dans la suite. On peut ajouter des entrées var-i+ avec un bouton adéquat (en haut à gauche). Elles représentent les

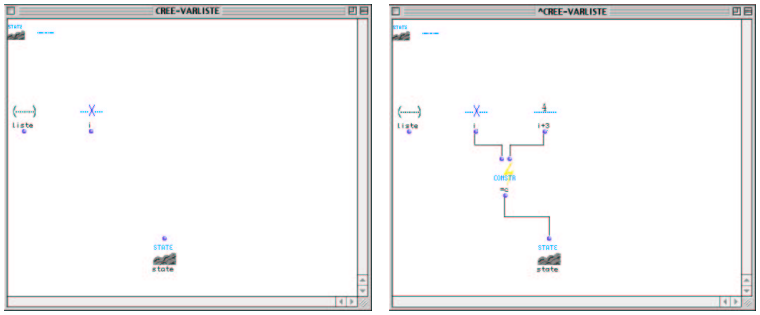


Figure 2: Définition des contraintes, à gauche, le patch par défaut. A droite, un exemple avec comme seule contrainte que chaque variable $V_i$ soit égale à $V_{i+4}$.

variables suivant la variable courante. On peut ajouter autant de var-i+ que nécessaire, qui représenteront successivement les variables d'indices $i+1$, $i+2$, $i+3$, etc. Ainsi, pour écrire une contrainte sur deux variables qui se suivent, on ajoutera la variable d'indice $i+1$ pour poser la contrainte entre var-i et l'instance de var-i+ ainsi créée, qui représente la variable d'indice $i+1$.

OMClouds intègre plusieurs primitives de contraintes : =c, equalc pour l'égalité, <c, <=c, notequalc pour les inégalités, andc, orc pour le opérateurs logiques, minimizec, memberc, alldiffc et cardc pour des contraintes de plus haut niveau. Elles sont associées à des fonctions de coût prédéfinies.

Les sorties varstate servent à poser les contraintes. On peut en ajouter autant que nécessaire. Il suffit de connecter la contrainte à la sortie. Une remarque importante : les contraintes peuvent être des fonctions prédéfinies mais ce n'est pas obligatoire. Un utilisateur peut vouloir définir ses propres fonctions de coût, par exemple un cas très fréquent est de minimiser une certaine distance sur les variables. Il lui suffit alors de programmer visuellement la distance dans le patch de contraintes et de la connecter à une sortie varstate.

Quand l'utilisateur ferme le patch, les fonctions de coûts associées aux contraintes sont automatiquement définies. Elles sont placées dans le slot contraintes de l'instance d'asvarliste correspondant. La génération de code est sensiblement la même que pour un patch OpenMusic, à ceci près que l'on traduit les primitives de contraintes par leurs fonctions de coût. La fonction ainsi générée a toujours deux paramètres, l'indice courant $i$ et la configuration courante $l$. Pour asvarliste, l'entrée var-i sont traduites par $(nthil)$, c'est-à-dire en CLOS la valeur de la variable. L'entrée var-liste par $l$ (quand elle est utilisée), les entrées var-i+ ne sont pas des paramètres mais sont traduites dans le code par $(nth(+i\ldots))$, c'est-à-dire en CLOS la valeur des variables suivantes. Pour les cycles, tous les indices sont pris modulo le nombre de variables.

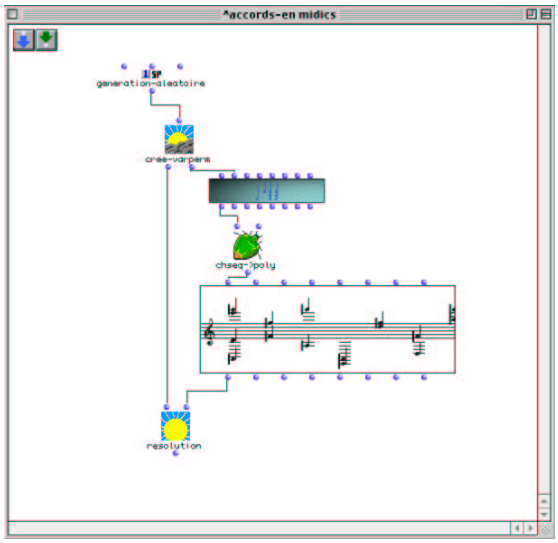


Figure 3: OMClouds permet d'exploiter des résultats partiels. Ici le problème du classement d'accords. On récupère les variables, suites de valeurs midi, que l'on place dans un objet chord-seq. La fonction chordseq->poly représentée par l'insecte sert à lier les notes pour mieux voir le résultat. L'ensemble de ce calcul est actualisé à chaque minimum local.

## 3.3 Résolution

L'algorithme de recherche adaptative est implémenté dans la méthode résolution. Elle a par défaut une entrée, qui prend une instance d'asvarliste. Ses autres entrées sont optionnelles. La deuxième sert à l'édition de résultats approchés, voir ci-dessous. La troisième est la longueur tabu $l_{tabu}$, qui est fixée par défaut au nombre de variables. On peut la changer soit pour accélerer la résolution, soit pour forcer la recherche autour des minima locaux. La quatrième est la condition d'arrêt $\epsilon$, qui est fixée par défaut à 0. On peut la changer soit pour obliger le solver à ne pas terminer, et a renvoyer indéfiniment des solutions (il suffit de la mettre à $-1$ par exemple), soit pour au contraire arrêter la recherche à partir d'un certain seuil si l'on sait que le problème n'a pas de solutions. La dernière entrée optionnelle est le nombre d'itérations avant de réinitialiser le solver.

Plus intéressant, la première entrée optionnelle sert à exploiter les résultats partiels rencontrés lors de la recherche. Quand un minimum local est trouvé, la boîte résolution fait deux mises à jour : d'abord, elle actualise les "meilleures valeurs" dans la deuxième sortie de la boîte asvarliste, ensuite elle réévalue l'entrée optionnelle.

### 3.4 Conclusions

OMClouds est intégrée à la version 4.6.5 d'OpenMusic. Il est prévu de lui ajouter des fonctionnalités : d'abord, de permettre à l'utilisateur de ralentir ou stopper la résolution de manière à voir plus facilement les résultats intermédiaires. Ensuite, nous envisageons d'ajouter des couleurs dans les éditeurs musicaux pour distinguer les variables dont l'erreur est nulle des autres. Il y a cependant un problème pour savoir à quelles variables correspondent quels objets musicaux. Enfin, nous prévoyons d'utiliser la liste tabu pour réduire le problème en cours de résolution. Par exemple, si l'utilisateur est satisfait par la première moitié d'un résultat intermédiaire, mais pas par la suite, il serait utile de pouvoir bloquer la première moitié de l'instanciation et de continuer à résoudre sur la deuxième. D'un point de vue CSP, il faudrait redéfinir et résoudre le sous-CSP correspondant. On peut très facilement le faire en recherche adaptative puisque ce sont les variables qui sont marquées tabu : il suffit de forcer les premières variables dans la liste tabu, de manière à ce qu'elles ne soient jamais sélectionnées. L'avantage est de ne pas avoir à ré-instancier le CSP.

Concernant la composition assistée par ordinateur, l'apport le plus important de OMClouds est certainement l'édition des résultats intermédiaires, et le contrle donn l'utilisateur sur la rsolution. Au contraire d'une boîte opaque qui renvoie des résultats au bout d'un temps indéfinis, résultats pouvant être "il n'y a pas de solutions", OMClouds fournit des propositions. Nous pensons que cette manière très souple de voir la programmation par contraintes est beaucoup mieux adaptée à la composition, et correspond la philosophie de calcul musical interactif dfendue par OpenMusic

## References


[1] A. Agon. *An environment for computer assisted composition.* Thèse de doctorat, IRCAM-UniversitÈ de Paris VI, 1998.

[2] G. Assayag, C. R. M. Laurson, C. Agon, and O. Delerue. Computer assisted composition at ircam : Patchwork & openmusic. *Computer Music Journal*, 1999.

[3] P. Ballesta. *Contraintes et objets, clefs de voûte d'un outil d'aide à la composition.* Editions Hermès, 1998.

[4] M. Chemillier and C. Truchet. Analyse musicale et contraintes. *Journées d'Informatique Musicale*, 2002.

[5] P. Codognet and D. Diaz. Yet another local search method for constraint solving. *LNCS 2246, SAGA 2001, first Symposium on Stochastic Algorithms : Foundations and Applications*, 2001.

[6] P. Codognet, D. Diaz, and C. Truchet. The adaptive search method for constraint solving and its application to musical csps. *International Worksop on Heuristics*, 2002.

[7] K. Ebcioglu. *An expert system for harmonizing chorals in the style of J.-C. Bach.* AAAI Press, 1987.

[8] J.-K. Hao, P. Galinier, and M. Habib. Métaheuristiques pour l'optimisation combinatoire et l'affectation sous contraintes. *Journal of Heuristics*, 1998.

[9] M. Laurson. Patchwork : a visual programming language and some musical applications. *Sibelius Academy*, 1996.
[10] L. Michel and P. V. Hentenryck. Localizer : a modeling language for local search. *In proc. CP'97, 3rd International Conference on Principles and Practice of Constraint Programming*, 1997.
[11] L. Michel and P. V. Hentenryck. Localizer. *Constraints*, 2000.
[12] F. Pachet and P. Roy. Musical harmonization with constraints : a survey. *Constraints*, 2001.
[13] C. Rueda, M. Laurson, G. Bloch, and G. Assayag. Integrating constraint programming in visual musical composition languages. *Proceedings of ECAI'98*, 1998.
[14] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. *AAAI'92*, pages 440–446, 1992.
[15] C. P. Tsang and M. Aitken. Harmonizing music as a discipline of constraint logic programming. *ICMC*, pages 61–64, 1991.