

Application de la programmation par contraintes relationnelles à l'analyse et à la composition musicales

Mémoire présenté par
Sascha Van Cauwelaert
en vue de l'obtention du titre de
Master ingénieur civil en informatique
option **Artificial Intelligence**

Lecteurs : Charles Pecheur
Carlos Agon

Promoteur : Peter Van Roy
Co-Promoteur : Gérard Assayag



Université catholique de Louvain
Année académique 2010–2011

Résumé

Le paradigme de la programmation par contraintes a déjà fait ses preuves dans le domaine de la composition et de l'analyse musicales. Cependant, il n'est pas aisé de travailler globalement à ce jour ; les systèmes actuels permettent de représenter les paramètres musicaux indépendamment des autres uniquement. Afin de résoudre ce problème, ce travail propose d'utiliser un nouveau type de variable : les relations. Ce document décrit ce qui a pu être réalisé à ce niveau.

De manière théorique, nous montrons que les relations apportent beaucoup en permettant comme attendu de travailler à un niveau global de la musique. La modélisation de certains problèmes est également plus simple et plus concise. D'autres n'auraient peut-être même pas été envisagés.

Par ailleurs, les relations peuvent avoir d'autres applications dans le domaine musical, comme la transformation de partitions complètes. De manière générale, les relations permettent sans doute beaucoup plus que ce qui est présenté dans ce document : elles permettent en théorie de créer des structures de grande taille et très complexes, sans rendre la modélisation ardue et confuse. Dans la suite, il faudrait donc que les compositeurs et les informaticiens exploitent le travail apporté ici afin de découvrir quels sont les réels impacts esthétiques, ainsi que d'autres moyens d'utiliser les relations.

Finalement, il serait également intéressant d'envisager l'utilisation des relations dans le domaine acoustique.

« *La dissonance picturale et musicale d'aujourd'hui
n'est rien d'autre que la consonance de demain.* »
Vassily Kandinsky.

Remerciements

Je remercie avant tout Gustavo Gutiérrez pour son aide inestimable ainsi que pour le temps qu'il m'a consacré durant cette année. Je remercie également le promoteur de ce mémoire, Peter Van Roy, pour son appui, pour l'apprentissage acquis et pour m'avoir permis de travailler dans ce domaine. Ensuite, je tenais à remercier les membres de l'IRCAM qui m'ont aidé dans l'étude de leur spécialité, notamment Jean Bresson et Carlos Agon. Je suis de plus reconnaissant envers Torsten Anders, sans qui ma compréhension de certains problèmes serait en-dessous de ce qu'elle est aujourd'hui. Je remercie également Yves Jaradin, pour certaines discussions intéressantes que nous avons eues durant cette année académique.

Finalement, je porte une attention particulière envers mes proches, qui m'ont toujours soutenu dans tout ce que j'ai entrepris.

Table des matières

Introduction	1
I Fondements	3
1 Bases logicielles	5
1.1 OpenMusic	5
1.2 Gecode	7
1.2.1 Programmation par contraintes	7
1.2.2 Description générale de Gecode	10
1.2.3 Résolution du problème de la série tous-intervalles avec Gecode	11
1.3 Interface entre Gecode et Common Lisp : GeLisp	15
1.3.1 Résolution du problème de la série tous-intervalles avec GeLisp	17
2 Relations dans la programmation par contraintes	23
2.1 Représentation des ensembles d'entiers	23
2.2 Relations	24
2.2.1 Théorie	25
2.2.2 Relations dans Gecode	31
II Programmation par contraintes et musique	35
Avant-propos	37
3 Etat de l'art	41
3.1 Généralités	41
3.2 Approche déclarative de la composition	43
3.3 Classes de problèmes solubles	44
4 Apport des relations	47
4.1 En théorie	47
4.1.1 Représentation d'une partition complète par une relation	47
4.1.2 Représentation d'une transformation d'une partition par une relation	50

4.1.3	Relations utilisées de manière générale	55
4.1.4	Utilisation des relations ground	55
4.1.5	Nouvelles classes de problèmes abordées et contraintes associées	57
4.1.6	Remarque sur le gain apporté par les relations	58
4.2	Cas pratiques	60
4.2.1	Recherche de parties communes de deux partitions	61
4.2.2	Répartition d'une pièce musicale entre différents instruments	62
4.2.3	Exemples d'applications simples	65
4.3	Le problème du contexte de partition inaccessible revisité	65
III	Evaluation et perspectives	71
	Conclusion	81
	Annexes	85
A	Informations complémentaires	I
A.1	Utilisation de code Common Lisp dans OpenMusic	I
A.2	Description de la résolution du problème des n reines avec Gecode	II
A.3	Utilisation des entiers, des booléens et des ensembles d'entiers en tant que variables de décision dans Gecode	IV
A.4	Utilisation pratique des relations dans Gecode	VIII
A.5	Deux systèmes existants : Situation et OMRC	XI
B	Description détaillée de GeLisp	XVII
B.1	Génération de l'interface	XVII
B.1.1	Architecture de GeLisp	XVII
B.1.2	Implémentation de GeLisp à l'aide de SWIG	XXII
B.2	Utilisation de l'interface	XXV
B.2.1	Dépendances nécessaires	XXV
B.2.2	Emploi effectif de l'interface	XXVI
B.2.3	Utilisation de Gecode à partir d'OpenMusic : Récapitulatif	XXVIII
C	Codes source	XXXI
D	Fichiers contenus sur le CD-ROM annexe	LVII

Introduction

La musique est le premier des arts pour lequel on a tenté de marier pensée scientifique et création artistique. Pythagore déjà, l'expliquait selon les mathématiques.

De nos jours, la programmation par contraintes est utilisée afin de résoudre des problèmes du domaine musical. Ce paradigme, utilisé pour résoudre des problèmes combinatoires de tailles conséquentes, permet des résultats assez concluants dans ce domaine. Par exemple, la génération d'une mélodie régie par certaines contraintes : intervalles mélodiques ou harmoniques entre les notes, motifs rythmiques, ...

Cependant, la programmation par contraintes ne permet aujourd'hui qu'une modélisation très locale de la musique. Les différents paramètres musicaux (hauteur, durée, intensité, ...) ne peuvent être représentés qu'indépendamment les uns des autres. Il n'est en effet généralement possible que d'utiliser et de contraindre des entiers, des booléens et des ensembles d'entiers.

Représenter et contraindre une pièce globalement selon des critères esthétiques n'est alors pas immédiat. Par exemple, imposer l'emploi de leitmotifs ou de variations sur un thème ne sont pas des tâches triviales.

Posséder une représentation de la musique pour laquelle les paramètres sont connexes est nécessaire. En effet, dans la musique, rien n'est totalement indépendant.

Dans ce travail, nous proposons donc l'utilisation d'un nouveau type de variables : les relations. Celles-ci, permettant de représenter un ensemble de liens entre des entiers, semble être un moyen de pallier au problème.

Ce qui est présenté ici peut se diviser en trois niveaux.

Premièrement, il a fallu permettre, de manière purement technique, l'utilisation de la programmation par contraintes munie des relations dans un environnement musical. La première partie de ce document décrit comment ceci a été réalisé. Elle décrit également comment le concept de relation s'est inscrit dans la programmation par contraintes de façon générale.

Le second niveau décrit tout d'abord le domaine étudié tel qu'il est aujourd'hui. Ensuite, il fournit un ensemble de possibilités et de concepts de base pour l'utilisation des relations dans le domaine musical. Les deux idées générales sont :

- La représentation globale d'une partition complète
- La représentation globale d'une transformation

Il est même possible d'aller plus loin : les relations permettent en fait de lier tout ce qui est représentable par des entiers dans la musique.

Tout ce qui est décrit dans ce document peut être utilisé conjointement avec ce qui existe déjà

aujourd'hui.

Le troisième niveau est quant à lui toujours à réaliser. Il devrait fournir un ensemble d'outils pour les compositeurs. Il faudra également démontrer grâce à des résultats concrets la preuve de l'apport pratique de ce travail : dans notre cas, un ensemble de pièces musicales.

Première partie

Fondements

Chapitre 1

Bases logicielles

Ce premier chapitre décrit les deux outils sur lesquels se base ce mémoire : OpenMusic et Gecode. A la base, ces deux outils n'ont pas été prévus pour être utilisés ensemble. Nous décrivons donc finalement comment nous avons pu rendre ceci possible.

1.1 OpenMusic

OpenMusic est un langage de programmation basé sur CommonLisp/CLOS¹, dont l'utilisation permet la composition assistée par ordinateur et le traitement du son. Le paradigme de ce langage est la programmation visuelle (ou graphique), c'est-à-dire « un langage de programmation dans lequel les programmes sont écrits par assemblage d'éléments graphiques. Sa syntaxe concrète est composée de symboles graphiques et de textes, qui sont disposés spatialement pour former des programmes. » [1] Ce langage a été initialement développé à l'IRCAM² par Gérard Assayag, Carlos Agon et Jean Bresson (d'autres membres de l'équipe nommée « Représentation Musicales » ont depuis également apporté leur contribution à ce projet). Si les développeurs de ce langage ont choisi le paradigme de programmation graphique, c'est avant tout afin de permettre aux personnes qui ne sont pas a priori des habitués de la programmation (comme les compositeurs) de s'adapter le plus vite possible. Cependant, il est également possible d'étendre ce langage à souhait, en ajoutant de nouvelles classes et méthodes écrites en Common Lisp³.

Dans cette section, nous décrivons OpenMusic de manière générale, le but de ce mémoire n'étant pas une étude approfondie de celui-ci. Afin de trouver plus d'informations à son sujet, il suffit de consulter le site qui lui est consacré [2].

1. Common Lisp Object System. Dans la suite de ce document, nous ne mentionnerons que le langage Common Lisp, car bien qu'OpenMusic utilise le paradigme de la programmation orientée objet en Common Lisp, nous ne l'avons pas utilisé au moment du développement du code Common Lisp de ce travail. Remarque : Lisp désigne une famille de langages et non un langage.

2. Institut de Recherche et Coordination Acoustique/Musique

3. La manière d'utiliser le langage Common Lisp dans OpenMusic est décrit dans l'annexe A.1

Vue d'ensemble d'OpenMusic

OpenMusic, en plus d'être un langage de programmation graphique, est aussi et surtout un environnement pour l'aide à la composition musicale par ordinateur. Dans cet environnement, deux fenêtres de base sont mises à disposition par défaut : le *workspace* et le *listener*.

Comme dans d'autres systèmes de travail tels qu'Eclipse, le workspace d'OpenMusic est un environnement de travail, qui se résume à un gestionnaire de fichiers sous forme d'icônes. Il est évidemment tout à fait possible de posséder plusieurs workspaces, mais un seul peut être utilisé à la fois. Comme dans tout système de fichiers, le workspace permet un classement hiérarchique des fichiers à l'aide de dossiers.

De plus, étant donné qu'OpenMusic est basé sur Common Lisp/CLOS, un listener spécifique est fourni. Celui-ci sert entre autres à afficher le résultat des évaluations. Un workspace et un listener d'OpenMusic sont respectivement illustrés dans les figures 1.1 et 1.2.

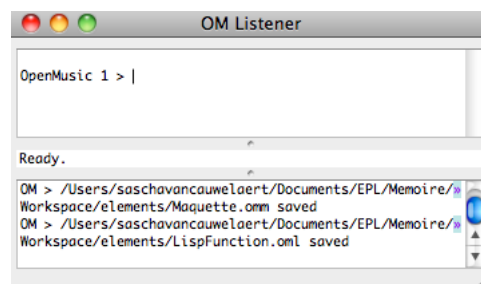
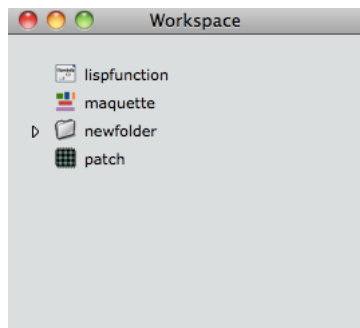


FIGURE 1.1 – Workspace d'OpenMusic. FIGURE 1.2 – Listener d'OpenMusic.

Bien qu'il soit également possible d'écrire un programme en ligne de code avec OpenMusic, il est à la base destiné à être utilisé pour écrire des programmes dans son langage visuel. Un des objets principaux de ce langage est le *patch* (exemple à la figure 1.3). Celui-ci est l'équivalent du corps d'un programme pour les langages textuels. Afin de le définir, il faut l'éditer sous la forme d'un conteneur (un cadre graphique), dans lequel on place les instructions du langage, sous forme d'icônes.

Chaque icône possède des entrées et sorties, représentées graphiquement par des points, respectivement au-dessus et en-dessous de l'icône. Le langage graphique pouvant être utilisé de manière hiérarchique, il est également possible de spécifier des entrées et sorties pour le patch. Ceci permettra de réutiliser ce corps de programme de manière fonctionnelle. On pourra donc l'utiliser par la suite à un plus haut niveau hiérarchique.

Afin d'utiliser la sortie d'une instruction en tant qu'entrée d'une autre, il suffit de les connecter entre elles par une ligne. Un programme écrit dans ce langage est donc un ensemble d'icônes, certaines d'entre elles étant connectées.

L'évaluation d'une instruction-icône a pour effet d'évaluer tout d'abord toutes les instruction-icône la précédant, c'est-à-dire celles dont la(les) sortie(s) sont connectée(s) à son(ses) entrée(s). Cette évaluation se fait de manière récursive, jusqu'à ce qu'il n'y ait plus d'entrées à évaluer.

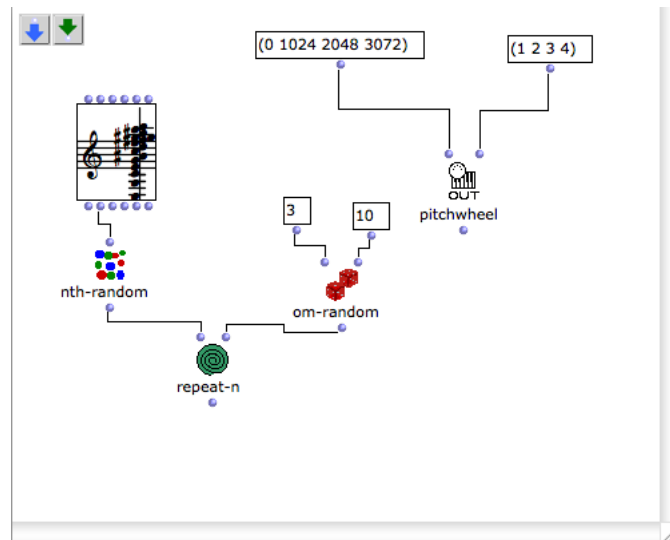


FIGURE 1.3 – Patch d’OpenMusic.

OpenMusic possède un ensemble de paquetages. Comme pour les langages Common Lisp et Java, ce sont des bibliothèques de classes et de fonctions génériques disponibles. Ils ont également une structure hiérarchique (voir figure 1.4). L’ensemble des classes et des fonctions contenues à l’intérieur peut également être exploré (exemple à la figure 1.5). Il est également possible d’avoir une vue en arbre des classes du paquetage (exemple à la figure 1.6).

OpenMusic étant basé également sur CLOS, le paradigme orienté objet est utilisé. Il est possible de créer de nouvelles classes héritant d’autres, et donc d’étendre le système selon ses besoins.

Beaucoup de librairies sont également à disposition dans l’environnement OpenMusic, notamment afin de faire de la programmation par contraintes dans le domaine musical. On peut également les trouver dans l’environnement des paquetages.

Finalement, des conteneurs spéciaux existent : les **éditeurs musicaux**. Ceux-ci ne font pas que contenir de l’information, mais ils l’interprètent également et en donnent une représentation globale. Certains éditeurs en donnent une vision sous forme de partition (comme la classe *chord* (voir figure 1.7)). D’autres permettent d’exprimer l’information dans le domaine du son, sous forme d’onde par exemple.

1.2 Gecode

1.2.1 Programmation par contraintes

« *Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming : the user states the problem, the computer solves it.* »

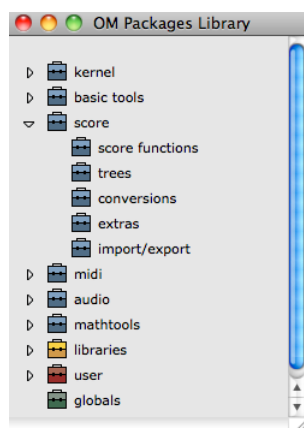


FIGURE 1.4 – Paquetages et bibliothèques d'OpenMusic.

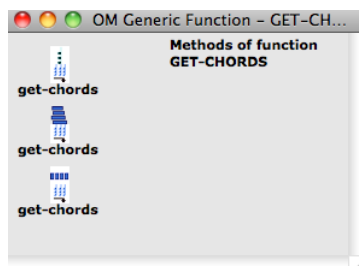


FIGURE 1.5 – Fonctions génériques d'OpenMusic.

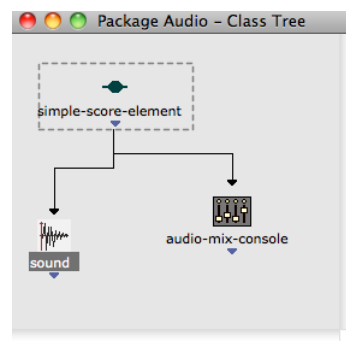


FIGURE 1.6 – Arbre de classes d'OpenMusic.

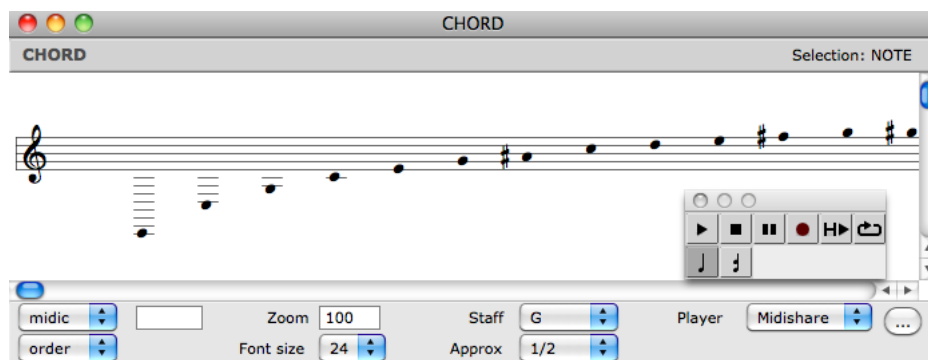


FIGURE 1.7 – Exemple d'éditeur musical d'OpenMusic.

Eugene Freuder

Le but de cette courte sous-section est de donner une description générale de ce qu'est la programmation par contraintes. Dans le cas où le lecteur ne connaît pas ce paradigme de programmation, il pourra alors se plonger plus facilement dans la suite de ce document. Dans le cas contraire, cette sous-section servira au pire de rappel.

La programmation par contraintes permet de résoudre des problèmes combinatoires, parfois de très grande taille. Dans ce paradigme, on divise généralement un programme en une partie de modélisation et en une partie de résolution :

- Dans la partie **modélisation**, on déclare un ensemble de variables. Les variables utilisées et nécessaires pour la modélisation sont appelées des **variables de décision**. Lors de la déclaration de ces variables, on leur donne un **domaine fini**. Ces variables sont appelées

des variables de décision car on décide en fonction des valeurs assignables à ces variables si oui ou non les **contraintes** qui sont imposées sur celle-ci peuvent être respectées. Imposer les contraintes est l'étape suivante de la modélisation. L'imposition d'une contrainte sur un ensemble de variables⁴ a pour effet d'exécuter un **propagateur** - l'implémentation effective de la contrainte (qui elle fait partie du monde logique) - à certains moments. L'exécution d'un propagateur a pour effet de modifier le domaine des variables concernées par la contrainte. Une fois les variables déclarées et les contraintes imposées sur ces variables, le problème est modélisé et prêt à être résolu. Ce problème modélisé de la façon décrite ci-avant est ce qu'on appelle un **problème de satisfaction de contraintes**.

- Dans la partie **résolution**, l'ordinateur démarre d'un problème de satisfaction de contraintes et modifie le domaine des variables en faisant des « branchements » afin d'arriver à un autre problème de satisfaction de contraintes. Un branchement consiste à réduire le domaine d'une variable. Le fait de réduire le domaine d'une variable a pour effet l'exécution des propagateurs associés à cette variable.

Par une ou plusieurs exécutions de ce processus de passage d'un problème de satisfaction de contraintes à un autre, on peut arriver à trois situations différentes :

- une ou plusieurs variables n'ont plus qu'une seule valeur dans leur domaine. Ces variables se voient alors assignées la valeur unique comprise dans leur domaine. On parle alors de **solution partielle**.
- toutes les variables n'ont plus qu'une seule valeur dans leur domaine. Toutes les valeurs ont donc une valeur assignée et on parle de **solution totale**. Le problème initialement modélisé est alors résolu. Le but initial est en effet atteint : assigner à chaque variable une valeur de son domaine, sans qu'aucune contrainte ne soit violée.
- une des variables possède un domaine vide. On parle alors d'**échec**.

Le processus de passage d'un problème de satisfaction de contraintes à d'autres afin de trouver une solution totale définit un arbre de recherche (voir figure 1.8) : chaque noeud est un problème de satisfaction de contraintes et chaque branche consiste à un branchement d'une variable. Le choix des variables et des branchements sur ces variables dépendent d'heuristiques de recherche définie par le programmeur et/ou par le système. Il est important de remarquer que la programmation par contraintes fait de la recherche complète dans l'arbre. Ceci impose le fait que pour une variable donnée, le domaine de cette variable dans un problème de satisfaction de contraintes donné (c'est-à-dire à un noeud donné de l'arbre de recherche) est exactement égal à l'union des domaines associés à cette variable dans les sous-problèmes de satisfaction de contraintes (les noeuds fils de ce noeud).

De manière interne, la recherche peut se faire de deux manières différentes lorsqu'on passe d'un noeud à un autre. Soit en copiant le noeud père et en le modifiant par la suite (« copy-

4. à partir de ce point et jusqu'à la fin de cette sous-section, nous faisons un abus de langage en parlant de « variables » à la place de « variables de décision » car bien qu'il existe d'autres types de variables dans la programmation par contraintes, ce sont surtout celles-là qui sont importantes dans ce paradigme. De plus, ceci permet de raccourcir le discours sans effet négatif.

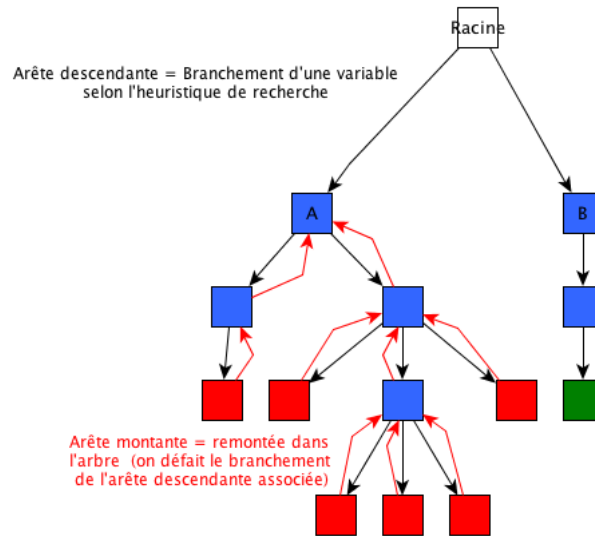


FIGURE 1.8 – Arbre de recherche. Les nœuds bleus sont des solutions partielles. Les nœuds rouges sont des échecs. Le nœud vert est une solution totale. On peut voir qu'une heuristique de recherche préférant le nœud B au nœud A trouvera plus vite la solution du nœud vert.

based »), soit en modifiant directement le nœud père mais en conservant une trace des modifications (« trail-based »).

La première possibilité prend évidemment plus de place mais demande moins de temps de calcul lorsqu'on remonte dans l'arbre de recherche (« backtracking »). Ajoutons que réaliser chaque copie prend également un certain temps. Le « copy-based » est donc surtout intéressant lorsque les domaines sont fort élargés à chaque nœud de l'arbre de recherche.

Gecode, ne permet que de faire du « copy-based »⁵. La librairie *CHOCO* [3] pour le langage Java permet de faire les deux, en utilisant le « trail-based » par défaut.

1.2.2 Description générale de Gecode

Gecode est une librairie écrite en C++ et développée essentiellement par Guido Tack, Christian Schulte et Mikael Z. Lagerkvist. Elle permet le développement de systèmes basés sur les contraintes et fournit un solveur de contraintes avec des performances de l'état de l'art en matière de programmation par contraintes. De plus, cette librairie est, de par son architecture, aisément modulable et extensible (en l'occurrence, ce travail se base sur l'une des extensions de Gecode, que nous décrirons dans le chapitre 2).

5. En fait, Gecode permet également de faire un compromis entre le trailing et la copie pure. Il utilise pour cela une technique nommée « Recalcul » [37]. Nous mentionnons ici cette technique car d'après Torsten Anders, les problèmes de satisfaction de problèmes musicaux sont souvent hautement complexes : ces problèmes nécessitent beaucoup d'informations et génèrent de profonds arbres de recherche. Le recalcul est une technique essentielle pour résoudre de grands problèmes qui sinon nécessiteraient une demande de mémoire excessive [23].

La figure 1.9 donne une représentation d'un **espace Gecode**.

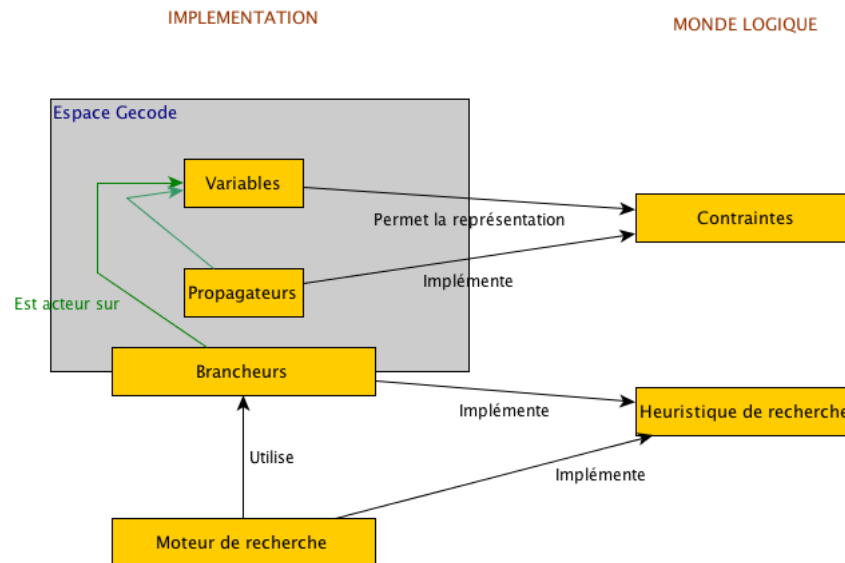


FIGURE 1.9 – Espace dans Gecode. Les brancheurs sont placés sur la frontière de l'espace afin de montrer qu'ils font le lien entre celui-ci et le moteur de recherche.

Dans Gecode, un espace est destiné à contenir les variables de décision, les contraintes, et les *brancheurs*. Un brancheur permet de définir les heuristiques de choix des variables et des valeurs pour la recherche. L'ensemble des brancheurs décrivent donc l'arbre de recherche.

Des *moteurs de recherche* existent également. Ce sont eux qui vont faire appel aux brancheurs, afin de pouvoir créer un nouveau noeud dans l'arbre de recherche. Ce sont donc eux qui réalisent véritablement la recherche. Ils définissent également une stratégie de recherche, comme par exemple la *recherche en profondeur la plus à gauche*.

Il est aussi possible de leur passer des options. Par exemple, on peut utiliser un objet *Stop* qui permet d'imposer une limite à la recherche : le nombre de noeuds ou d'échecs, par exemple.

1.2.3 Résolution du problème de la série tous-intervalles avec Gecode

Afin d'illustrer le fonctionnement général de Gecode⁶, nous allons maintenant expliquer comment résoudre un problème assez connu de la programmation par contraintes appliquée à la musique. Celui-ci est assez simple à formuler mais ne se résout pas de manière triviale.

6. Le lecteur intéressé par une description détaillée du problème des n -reines avec Gecode peut consulter l'annexe A.2. La description de l'utilisation de Gecode pour la résolution d'un problème de satisfaction de contraintes y est plus complète.

Le but est de créer une série dodécaphonique dans laquelle les intervalles entre les notes successives sont tous distincts : la série **tous-intervalles**. Peu de contraintes sont nécessaires à la modélisation de ce problème. Dans la définition que nous donnons, nous parlons de classes de hauteur plutôt que de hauteur de notes, car, une fois une solution trouvée, elle peut être transposée afin de faire commencer la série sur n'importe quelle note de la gamme chromatique. Le résultat reste alors évidemment correct.

Voici la définition formelle de ce problème :

– Les variables de décision sont :

- *pitchClasses* : un tableau de 12 variables entières, dont le domaine est $\{0, \dots, 11\}$.
- *intervals* : un tableau de 11 variables entières, dont le domaine est $\{1, \dots, 11\}$.

– Les 5 contraintes nécessaires sont :

- $\bigwedge_{i=0}^{10} \text{inversionalEquivalentInterval}(\text{pitchClasses}_i, \text{pitchClasses}_{i+1}, \text{intervals}_i)$
- $\text{distinct}(\text{pitchClasses})$
- $\text{distinct}(\text{intervals})$
- $\text{pitchClasses}_0 = 0$
- $\text{pitchClasses}_{11} = 6$

Le tableau de 12 variables entières *pitchClasses* est donc destiné à contenir les 12 classes de hauteur de la série dodécaphonique. Le tableau *intervals* contiendra les intervalles successifs de ces 12 classes de hauteur.

Par définition, dans une série dodécaphonique les 12 notes sont différentes. Nous utilisons donc la contrainte **distinct** sur le tableau *pitchClasses*. De même, au vu de la définition de la série tous-intervalles, nous posons cette contrainte sur le tableau *intervals*.

Ensuite, afin de **casser la symétrie**⁷ du problème, nous empêchons de trouver des solutions équivalentes, **à une transposition près**, en imposant la valeur assignée à la première classe de hauteur (dans ce cas 0).

Dans la programmation par contraintes, une technique permettant également de réduire l'arbre de recherche est l'utilisation de **contraintes redondantes**. Celles-ci sont des contraintes implicitement présentes dû au système de contraintes déjà appliquées, mais dont l'application explicite permet d'ajouter un propagateur qui réduira de manière effective l'arbre de recherche. Dans notre cas, nous savons que l'intervalle entre la première et la dernière classe de hauteur est obligatoirement un triton. Nous imposons donc la valeur de la dernière classe de hauteur à 6. Ceci est justifié dans [23]

La première contrainte est quant à elle un peu plus compliquée. Nous la décrivons donc un peu plus longuement dans ce qui suit.

7. C'est-à-dire réduire l'arbre de recherche en empêchant de trouver des solutions symétriquement égales.

Dans la musique occidentale et bien tempérée⁸, on définit un **intervalle renversé** comme suit :

Si un intervalle i ascendant (respectivement descendant) va de la note a vers la note b , l'intervalle renversé i_{inv} de i est un intervalle descendant (respectivement ascendant) allant de a vers b . Par exemple, à partir de l'intervalle ascendant de tierce majeure entre les notes fa et la, on obtient l'intervalle renversé de sixte mineure en descendant de fa vers la (voir l'image 1.10). De plus, on remarque la propriété suivante : la somme de la **distance** d'un intervalle et de la **distance** de son renversement forme une octave juste (à un multiple d'octave près).

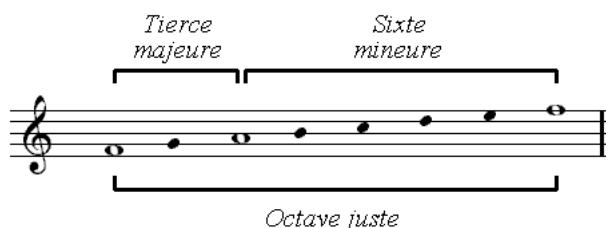


FIGURE 1.10 – Intervalle et intervalle renversé. (image tirée de [4])

Dans la musique (et plus spécialement dans la **théorie musicale des ensembles**), on définit le concept d'**équivalence inverse**, pour laquelle des ensembles de hauteurs (dont font partie les intervalles) sont égaux s'ils sont renversés. Dans l'exemple précédent, ceci signifie que l'intervalle ascendant entre fa et la est équivalent à l'intervalle descendant entre fa et la. La première contrainte du problème a pour but d'exprimer que ce concept est d'application.

Pour ce faire, nous allons utiliser la propriété exprimée ci-dessus au sujet de la somme de la distance d'un intervalle et de la distance de son intervalle renversé. En utilisant des valeurs MIDI pour exprimer les valeurs des intervalles (un intervalle ascendant et un intervalle descendant étant respectivement positif et négatif), cette propriété peut être formalisée comme suit :

$$i - i_{inv} = 12.n \quad (i, i_{inv}, n \in \mathbb{Z}) \quad (1.1)$$

ou encore,

$$i \equiv i_{inv} \pmod{12} \quad (1.2)$$

Autrement dit, i et i_{inv} sont **congrus modulo 12**. L'opération modulo 12 forme donc une relation d'équivalence pour i et i_{inv} . Appliquer cette opération aux intervalles permet donc de les rassembler en classes d'équivalence, et donc d'exprimer l'équivalence inverse des intervalles. On définit donc un intervalle entre deux notes successives comme leur différence de hauteur, modulo 12.

8. Selon le tempérament égal de Carl Philipp Emanuel Bach.

Formellement,

$$\text{inversionalEquivalentInterval}(PC1, PC2, Interval) \stackrel{\text{def}}{=} \\ Interval = (PC2 - PC1) \pmod{12}$$

Puisque cette équivalence est imposée et que tous les intervalles sont distincts, on peut s'assurer d'avoir des solutions avec des intervalles distincts, même s'ils sont renversés.

Ce problème étant défini, nous pouvons maintenant l'implémenter⁹.

Dans Gecode, un problème de satisfaction de contraintes est nécessairement encapsulé dans une classe fille de la classe *Space*, la classe qui encapsule le concept d'espace, décrit dans la sous-section 1.2.2.

Les variables de décision sont quant à elles définies en tant que variables d'instance de la classe fille. Les domaines des variables sont imposés dans le constructeur de cette classe. C'est également dans celui-ci qu'on impose les contraintes et qu'on définit les brancheurs à utiliser :

```

for(int i = 0 ; i < 11 ; i++)
{
    //establish needed coefficients
    int coeffTable[12] = {0};
    coeffTable[i]=-1;
    coeffTable[i+1]=1;

    //diffpitchshifted(i) = pitchClasses(i+1) - pitchClasses(i) + 12
    //gecode doesn't manage negative modulo dividend
    rel(*this, pitchClasses[i+1] - pitchClasses[i] + 12 == diffpitchshifted[i
    ]);

    //intervals(i) = (pitchClasses(i+1) - pitchClasses(i) + 12) mod 12
    mod(*this, diffpitchshifted[i], divisor, intervals[i]);
}

//dodecaphonic serie
distinct(*this, pitchClasses);

//by definition
distinct(*this, intervals);

//symmetry break (avoid transposition) : pitchClasses(0) = 0
dom(*this, pitchClasses[0], 0);

//redundant constraint (allows to prune more) ; pitchClasses(11) = 6
dom(*this, pitchClasses[11], 6);

//we only need the values of those variables

```

9. L'intégralité du code est fournie dans l'annexe C.

```
branch(*this, pitchClasses, INT_VAR_SIZE_MIN, INT_VAL_MIN);
branch(*this, intervals, INT_VAR_SIZE_MIN, INT_VAL_MIN);
```

Le moteur de recherche est quant à lui totalement indépendant du modèle. Il n'est donc pas défini dans la classe. Celle-ci est utilisée par le moteur de recherche afin de résoudre le problème désiré.

```
Script::run<AllIntervalsSerie,DFS,Options>(opt);
```

Voici une solution obtenue avec notre implémentation de Gecode. Certaines données statistiques sont données par Gecode également.

```
All intervals serie
    {0, 1, 3, 2, 7, 10, 8, 4, 11, 5, 9, 6}
    {1, 2, 11, 5, 3, 10, 8, 7, 6, 4, 9}

Initial
    propagators: 46
    branchers:   2

Summary
    runtime:      0.001 (1.343000 ms)
    solutions:    1
    propagations: 2072
    nodes:        98
    failures:     45
    peak depth:   13
    peak memory:  51 KB
```

Dans la sous-section 1.3.1, nous donnerons des résultats de ce problème dans l'environnement musical OpenMusic.

1.3 Interface entre Gecode et Common Lisp : GeLisp

Dans ce chapitre, nous avons vu qu'OpenMusic était basé sur Common Lisp. Nous avons également vu qu'il était possible d'exécuter du code Common Lisp à partir de l'environnement OpenMusic. Il suffit donc de permettre l'appel de routines de la librairie Gecode à partir d'un environnement Common Lisp afin de permettre l'accès à Gecode à partir d'OpenMusic. Dans notre cas, cet environnement est LispWorks, un outil de développement multi-plateforme pour le langage Common Lisp ANSI. Il s'agit en effet de l'outil utilisé par les développeurs d'OpenMusic.

Nous décrivons ici de manière globale comment GeLisp¹⁰, notre interface entre Gecode et

10. Le nom GeLisp provient d'une interface existante entre la version 2 de Gecode et Common Lisp, aujourd'hui obsolète. De gros changements ont en effet eu lieu au niveau de l'implémentation entre la version 2 et la version 3 de Gecode. Les développeurs de cette interface sont Mauricio Toro Bermudez et Camilo Rueda.

Common Lisp, a été réalisée.

GeLisp a été créée à l'aide de SWIG¹¹ [5]. SWIG est un compilateur d'interfaces qui connectent des programmes écrits en C et C++ avec un ensemble de langages de haut niveau. Concrètement, SWIG nous a permis de générer de manière (relativement) automatique le code d'interfaçage. Ce code est généré dans le langage Common Lisp, ce qui permet donc à LispWorks de l'interpréter, et donc d'utiliser la librairie Gecode. Ce code généré aurait très bien pu être écrit directement, mais l'utilisation de SWIG a permis plusieurs avantages non-négligeables :

- Un gain de temps car il n'a pas fallu développer le code directement.
- L'automatisme de la génération de l'interface permet une uniformité de celle-ci et d'éviter des erreurs de programmation.
- Il est assez aisé de maintenir cette interface. On peut facilement étendre celle-ci afin de rester consistant par rapport à la librairie Gecode en fonction de son évolution. Par exemple, l'ajout d'un propagateur dans Gecode ne nécessite que quelques lignes de code supplémentaires afin de pouvoir l'utiliser dans l'environnement LispWorks.

Cette facilité de maintenance est également due à l'architecture de l'interface. Nous avons essayé d'être le plus générique possible¹².

Une vue générale des différents fichiers¹³ et des mécanismes nécessaires à la génération de l'interface est donnée dans la figure 1.11.

Citons brièvement le rôle des fichiers sources :

- *gelisp.hpp* et *gelisp.cpp* implémentent le coeur de l'interface. Ils définissent les classes C++ qui permettent à l'utilisateur de modéliser un problème quelconque à partir de Common Lisp.
- *gecheader.hpp* contient les en-têtes de toutes les fonctions, classes ... de Gecode que SWIG rend utilisable à l'utilisateur Common Lisp.
- *gelisp.i* est un fichier utilisé par SWIG dans lequel nous donnons des directives quant à la génération de l'interface.

Le fichier *gelisp.lisp* contient toutes les fonctions étrangères permettant l'utilisation de Gecode à partir de Common Lisp. Les deux derniers fichiers sont décrits dans l'annexe B pour le lecteur intéressé.

GeLisp possède l'avantage de ne pas imposer à l'utilisateur de créer une classe pour son problème particulier, comme il devrait le faire avec Gecode. Tout ce qui lui est demandé est de créer un espace à l'aide d'une fonction. Ensuite, il peut ajouter des variables, des contraintes et

11. Simplified Wrapper and Interface Generator

12. Le lecteur intéressé peut se rendre à l'annexe B, dans laquelle se trouve une description détaillée de la manière dont l'interface a été créée. Remarquons qu'actuellement, nous ne fournissons de moyen de l'utiliser sur d'autres machines. Cependant, ceci pourrait être réalisé sans trop de problèmes dans le futur.

13. Certains d'entre eux sont fournis dans l'annexe C et les autres sont présents dans l'annexe D.

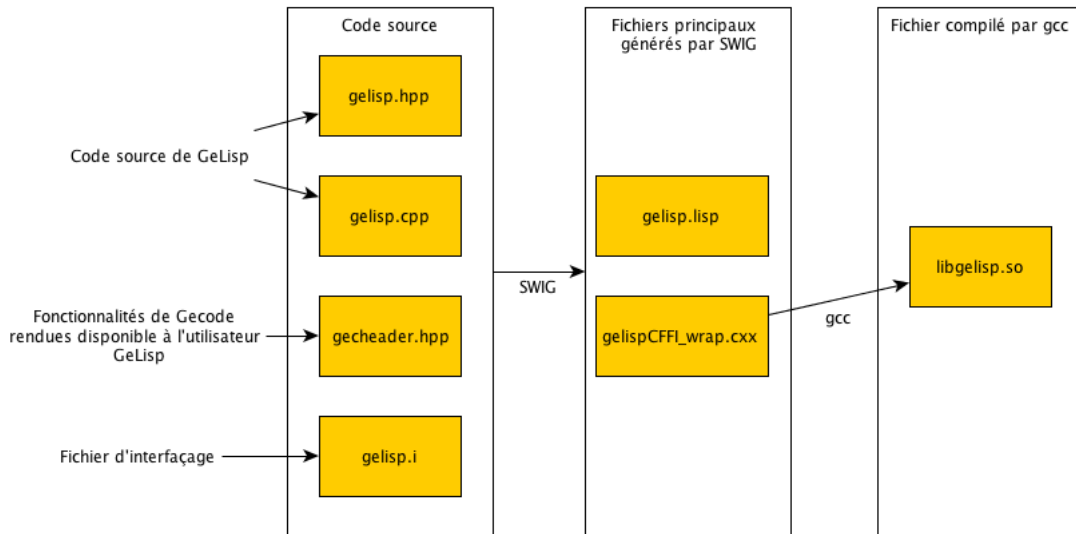


FIGURE 1.11 – Vue globale de la génération de l’interface.

des brancheurs selon ses besoins.

La recherche est également simplement faite à l’aide d’un appel de fonction.

Comme nous l’avons dit, GeLisp peut facilement être étendue et maintenue : par exemple, l’ajout d’un propagateur dans Gecode nécessite simplement l’ajout de son en-tête dans le fichier *gecheader.hpp* pour être utilisable à l’aide de GeLisp.

Pour pouvoir utiliser GeLisp, il faut disposer d’une librairie Common Lisp, nommée CFFI¹⁴ [6]. Elle fournit des outils afin d’interfacer un langage étranger au langage Common Lisp (dans notre cas, le C++). Cette librairie nous permet, à partir de LispWorks, d’allouer et de désallouer de la mémoire étrangère, ainsi que d’appeler des fonctions étrangères au langage Common Lisp.

Finalement, nous pouvons affirmer que GeLisp fournit le même potentiel de **modélisation** que Gecode. Seuls certains sucres syntaxiques n’ont pas été interfacés. Par contre, d’autres outils ne sont pas fournis, notamment au niveau de la recherche.

1.3.1 Résolution du problème de la série tous-intervalles avec GeLisp

Nous donnons maintenant la résolution du problème défini dans la sous-section 1.2.3, à l’aide de GeLisp. Nous détaillons un peu plus la manière de procéder et donnons des résultats pratiques, obtenus grâce à GeLisp dans OpenMusic.¹⁵

14. Common Foreign Function Interface

15. Le code Common Lisp permettant de résoudre ce problème à l’aide de GeLisp est fourni à l’annexe C. Le code utilisé dans OpenMusic y est quasiment identique.

Voici comment nous modélisons le problème, en l’implémentant dans une *Lisp function* dans OpenMusic. Après avoir défini l’espace et les deux tableaux de variables de décision, nous devons définir des variables auxiliaires permettant d’exprimer la contrainte *inversionalEquivalentInterval* :

```
; only used for mod divisor
(setq divisor (cl-user::G1Space_newIntVar_minmax sp 12 12))
; only used to express the mod constraint. It will contains the true pitch
  differences.
(setq diffpitch (cl-user::new_IntVarList_minmax sp 11 -11 11))
;used because domain must be shifted because Gecode doesn't manage modulo with
  negative dividend
(setq diffpitchshifted (cl-user::new_IntVarList_minmax sp 11 1 23))
```

Nous imposons ensuite les différentes contraintes requises pour le problème :

```
;coeff used to shift pitch differences
(setq coeffListForShift (new_IntList_sizeAndDefaultValue 2 1))

;inversionalEquivalentInterval
(loop for x from 0 to 10 do

  ; establish needed coefficients
  (setq coeffList (new_IntList_sizeAndDefaultValue 12 0))
  (IntList_setVar coeffList x -1)
  (IntList_setVar coeffList (+ x 1) 1)

  ; diffpitch(i) = pitchClasses(i+1) - pitchClasses(i)
  (linear_onIntArgsAndIVAAndIntVar_default sp (IntList_get coeffList) (
    IntVarList_get pitchClasses) :IRT_EQ (IntVarList_getVar diffpitch x))

  ; diffpitchshifted(i) = diffpitch(i) + 12
  (setq tempVarList (new_IntVarList))
  (IntVarList_add tempVarList (IntVarList_getVar diffpitch x))
  (IntVarList_add tempVarList divisor)
  (linear_onIntArgsAndIVAAndIntVar_default sp (IntList_get coeffListForShift
    ) (IntVarList_get tempVarList) :IRT_EQ (IntVarList_getVar
    diffpitchshifted x))

  ; intervals(i) = diffpitchshifted(i) mod 12
  (mod_default sp (IntVarList_getVar diffpitchshifted x) divisor (
    IntVarList_getVar intervals x))

)

;distinctPitchClasses
(distinct_default sp (IntVarList_get pitchClasses))

;distinctIntervals
(distinct_default sp (IntVarList_get intervals))

;symmetry break (avoid transposition) : pitchClasses(0) = 0
```

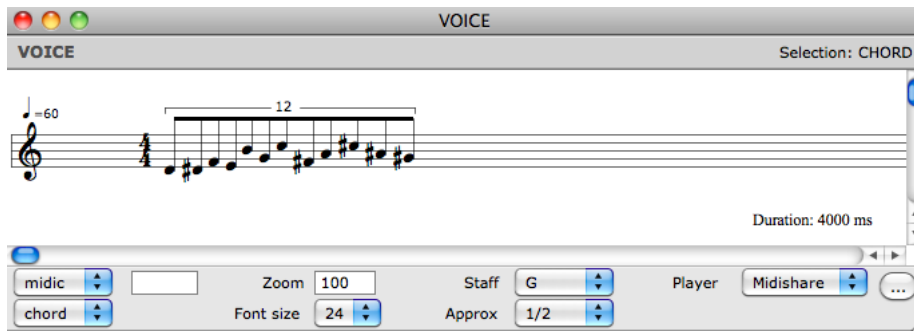



FIGURE 1.13 – Résultat du problème de la série tous-intervalles obtenu à l’aide de GeLisp utilisé dans OpenMusic.

$$\bigwedge_{i=0}^{10} intervals_i + intervals_{10-i} = 12$$

En effet, puisque la somme de la distance d’un intervalle et la distance de son renversement vaut 12¹⁷, cette contrainte suffit pour avoir une série symétrique. Exprimer ceci à l’aide de GeLisp ne demande que quelques lignes de code supplémentaires.

```
; constraint to have a symmetric serie : intervals(i)+intervals(11-1- i)=12
(setq tempVarList (new_IntVarList))
(IntVarList_add tempVarList (IntVarList_getVar intervals x))
(IntVarList_add tempVarList (IntVarList_getVar intervals (- 10 x)))
(linear_onIntArgsAndIVAAndIntVar_default sp (IntList_get coeffListForShift) (
  IntVarList_get tempVarList) :IRTEQ divisor)
```

Un des résultats est illustré à la figure 1.14.

On peut déjà remarquer que la programmation par contraintes permet de ne pas perdre trop de temps à réfléchir à la manière d’implémenter un problème et à la manière d’obtenir sa solution, mais de se consacrer à ce qu’on veut de la musique qu’on compose. On impose simplement de manière déclarative ce qu’on désire de la composition, en laissant l’ordinateur s’occuper de la résolution du problème.

Il faut cependant remarquer que pour des problèmes difficiles à résoudre, c’est-à-dire dont la recherche prend du temps pour l’ordinateur, il faut pouvoir réfléchir à la manière dont le processus de recherche est réalisé de manière interne. Il ne faut donc pas voir uniquement la programmation par contraintes comme un outil « boîte noire ».

17. voir sous-section 1.2.3

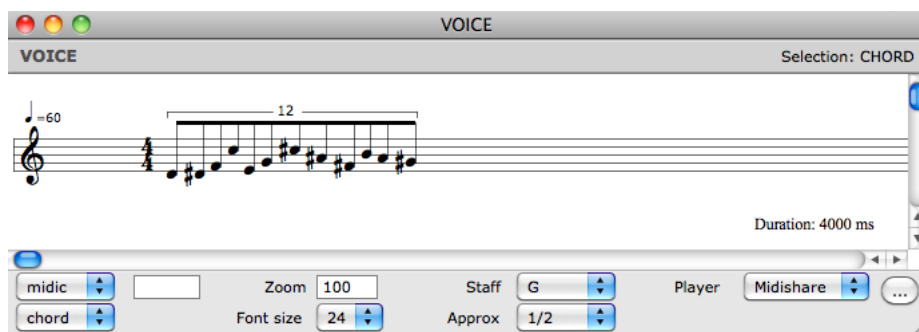


FIGURE 1.14 – Résultat du problème de la série tous-intervalles obtenu à l’aide de GeLisp utilisé dans OpenMusic. La contrainte de symétrie est ici imposée. Il s’agit de la même solution que celle que Torsten Anders présente dans sa thèse (bien qu’une note enharmonique et un saut d’octave les différencient).

Chapitre 2

Relations dans la programmation par contraintes

A ce jour, dans la plupart des systèmes permettant la programmation par contraintes existants, 3 concepts peuvent être représentés directement par une variable : les entiers, les booléens et les ensembles d'entiers¹. Dans ce chapitre, nous présentons d'abord brièvement comment les ensembles d'entiers sont représentés dans Gecode. Ceci sert en effet de base pour comprendre comment cette représentation a été étendue dans le cadre des relations. Ensuite, nous expliquons comment le concept de relation - au sens mathématique² - peut être utilisé dans la programmation par contrainte.

2.1 Représentation des ensembles d'entiers

La taille du domaine d'une variable entière³ dont le domaine peut contenir les valeurs allant de 1 à n est au pire égale à n .

Une variable représentant un ensemble d'entiers est une variable dont le domaine est un ensemble d'ensemble d'entiers. Or, le nombre de sous-ensembles de l'ensemble $\{1, \dots, n\}$ est égal à 2^n . La borne maximale de la taille du domaine d'une variable représentant un ensemble d'entiers est donc exponentielle en fonction de n . Cette complexité spatiale rend impraticable la représentation du domaine de ce type de variables par un ensemble des valeurs possibles pour cette variable.

C'est la raison pour laquelle Gecode (comme d'autres solveurs de contraintes, comme Comet [7] et JaCoP [8]) approxime le domaine de ces variables par un intervalle d'ensembles $[l\dots u]$, l et u étant respectivement les bornes inférieure et supérieure du domaine.

La borne inférieure, communément connue sous le nom de **plus grande borne inférieure** ou **glb** (de l'anglais), est l'ensemble de tous les entiers qui sont obligatoirement dans l'ensemble d'entiers représenté par la variable. La borne supérieure (connue sous le nom de **plus petite**

1. Nous renvoyons le lecteur intéressé par la manière de les utiliser dans Gecode à l'annexe A.3.

2. Plus précisément au sens algèbre relationnelle.

3. ou booléenne, qui n'est qu'un cas particulier de variable entière possédant un domaine au pire de taille 2.

borne supérieure ou **lub**), est l'ensemble de tous les entiers qui peuvent faire partie de l'ensemble d'entiers représenté par la variable (il faut donc avoir $glb \subseteq lub$).

Cette représentation prend évidemment beaucoup moins de mémoire qu'une représentation du domaine par l'ensemble de chacun de ses éléments (chaque élément du domaine est un ensemble d'entiers) ; seuls deux ensembles d'entiers sont en effet nécessaires. Cependant, elle ne permet qu'une approximation du domaine : par exemple, en reprenant l'exemple du guide des développeurs de Gecode [28], l'ensemble

$$\{\{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}\}$$

ne peut être représenté exactement par un intervalle. L'intervalle le plus proche permettant de représenter ce domaine est $[\{1, 2, 3\}]$. Comme d'autres systèmes, Gecode précise la représentation d'un domaine grâce à une borne inférieure et une borne supérieure pour la cardinalité du domaine. $[i \dots j]$ est la notation pour exprimer que la cardinalité est au minimum i et au maximum j . Dans notre exemple, ajouter $\#[1 \dots 2]$ à l'intervalle de bornes $[\{1, 2, 3\}]$ permet de représenter le domaine de manière exacte.

Au vu de ce qui précède, il est clair que les propagateurs ne peuvent faire de la domaine-consistance⁴ au niveau des variables représentant un ensemble d'entiers.

2.2 Relations

Dans cette section, nous décrivons un nouveau type de variables, actuellement développé par Gustavo Gutiérrez : les variables représentant des **relations**. Les relations peuvent être vues comme une généralisation des ensembles d'entiers. En effet, un ensemble d'entiers peut être vu comme une relation d'arité 1, c'est-à-dire un ensemble de tuples d'arité 1. En étendant cette idée à des ensembles de tuples d'arité n , on en arrive au concept de relation.

L'utilisation des relations n'est en fait pas tout à fait nouvelle dans le paradigme de la programmation par contraintes. Par exemple, le langage ESRA permet de modéliser un problème en utilisant des relations [30]. Cependant, jusqu'à présent, les systèmes permettant d'utiliser des relations se basent sur des traductions et des re-formulations de contraintes. Dans ce qui est présenté ici, il n'y a en aucun cas de traduction vers d'autres types de variables déjà connus ou de re-formulation de contraintes. Les contraintes interagissent directement avec des variables représentant des relations.

Dans cette section, nous décrivons tout d'abord ce système de contraintes dans le domaine relationnel de manière théorique et générale. Nous expliquons ensuite dans quelle mesure elles sont intégrées dans Gecode.

4. Toutes les valeurs qui sont inconsistantes en considérant **localement** les contraintes imposées des variables sont retirées de leur domaine. Il s'agit du meilleur élagage de domaine pouvant être réalisé en considérant les variables localement. D'autres types de consistance existent. Remarquons également que la domaine-consistance n'implique pas la consistance. [29]

2.2.1 Théorie

Dans cette sous-section⁵, nous décrivons la formalisation des relations dans un système de contraintes. La description est faite de manière incrémentale, chaque point étant nécessaire afin de développer les suivants.

Relation « ground »

Dans la théorie des ensembles et dans la logique, une relation *k-aire* est une propriété que des *k-tuples* d'éléments possèdent. Une relation *k-aire* peut être représentée par un ensemble de *k-tuples* pour lesquels la propriété est vraie. Une relation *k-aire* est donc une liste non-ordonnée de *k-tuples*, un *k-tuple* étant ici une liste ordonnée de *k* éléments.

Les relations définies comme ci-dessus doivent être distinguées des variables de décision représentant une relation. A partir de ce point, le terme employé pour les premières sera **relation ground**.

Nous définissons tout d'abord ici l'ensemble des éléments pouvant être présents dans un tuple :

$$\mathcal{U} = \{x : 0 \leq x < 2^K\}$$

K étant un entier arbitraire. Cette restriction est due à des raisons d'implémentations.

On définit ensuite R_n , l'espace des relations d'arité n dont les éléments sont des tuples ne pouvant contenir que des éléments de \mathcal{U} . Autrement dit :

$$R_n = \mathcal{P}(\mathcal{U}^n)$$

où \mathcal{U}^n est appelé l'**univers**.

Une relation $P \in R_n$ peut être vue comme un prédicat booléen sur \mathcal{U}^n :

$$P(t_1, \dots, t_n) \iff (t_1, \dots, t_n) \in P$$

Ceci étant dit, nous pouvons maintenant définir les opérations de base de négation et de conjonction d'une relation ground.

Négation ($\neg P$). Soit $P \in R_n$ une relation, sa négation est définie comme étant :

$$(\neg P)(t_1, \dots, t_n) \iff \neg(P(t_1, \dots, t_n))$$

En voyant P comme une relation (au sens ensemble de tuples) plutôt que comme un prédicat, cette opération se trouve être l'opération de complément d'un ensemble :

$$\neg P = \mathcal{U}^n \setminus P$$

5. Le contenu de cette sous-section se base largement sur le document écrit par Gustavo Gutiérrez [30].

Conjonction ($P \wedge Q$). Soit $P, Q \in R_n$ deux relations. Leur conjonction est définie comme étant :

$$(P \wedge Q)(t_1, \dots, t_n) \iff P(t_1, \dots, t_n) \wedge Q(t_1, \dots, t_n)$$

En considérant P et Q comme des relations, cette opération est l'intersection ensembliste :

$$P \wedge Q = P \cap Q$$

Les opérations de négation et de conjonction permettent de définir les autres opérations logiques, comme par exemple la *disjonction* et l'implication (respectivement équivalentes à l'*union* et à l'*inclusion* ensemblistes).

En effet, on a, si P et Q sont deux prédicats, les équivalences suivantes :

$$P \vee Q \equiv \neg(\neg P \wedge \neg Q)$$

$$P \Rightarrow Q \equiv \neg(P \wedge \neg Q)$$

Les opérations précédentes nécessitent des relations de mêmes arités et produisent une relation d'arité égale à celle de leurs opérands. D'autres opérations n'ayant pas cette propriété existent, comme le *produit cartésien*, la *quantification universelle* ou la *permutation*. Ces opérations ne sont pas présentées ici mais sont définies dans le document [30].

Domaine d'une relation

Une **variable relationnelle** est une variable de décision représentant une relation. Dans la programmation par contraintes, une variable de décision possède toujours un domaine permettant d'approximer l'information qu'elle représente ; le domaine d'une variable relationnelle peut être contraint, comme il est possible de le faire avec les autres types de variable de décision (entier, booléen, ensemble d'entiers). Les relations ground étant établies, nous pouvons maintenant définir ce domaine.

Une variable relationnelle X désigne une relation faisant partie d'un ensemble D_X de relations possibles :

$$D_X = \{S \in R_n : LB \subseteq S \subseteq UB\}$$

De la même manière que le domaine d'une variable représentant un ensemble d'entiers est approximé par deux ensembles d'entiers (glb et lub⁶), on approxime D_X par LB et UB . X peut alors prendre n'importe quelle valeur étant un sur-ensemble de LB et étant un sous-ensemble de UB . Puisqu'il ne s'agit que d'une approximation, il faut également spécifier que la valeur assignée à X doit faire partie du domaine D_X , c'est-à-dire que $X \in D_X$.

$LB \in R_n$ est la plus grande (au sens maximal pour la propriété d'inclusion \subseteq) des relations ground dont on **sait** qu'elle est incluse dans la valeur assignée à X . Autrement dit, il n'existe pas de relation ground G telle que $LB \subset G$ et dont on a la certitude que $G \subseteq X$.

6. voir la section 2.1

$UB \in R_n$ est la plus petite (au sens minimal pour la propriété d'inclusion \subseteq) des relations ground qui **peut** être assignée à X . Ou encore, il n'existe pas de relation ground G telle que $UB \subset G$ telle qu'il est possible que $G \subseteq X$.

Nous utiliserons la notation $glb(D_X)$ et $lub(D_X)$ pour désigner respectivement LB et UB . Les prédicats suivants permettent de raisonner sur l'état d'un tuple d'une variable relationnelle X :

$$\begin{aligned} (t_1, \dots, t_n) \in X &\iff (t_1, \dots, t_n) \in glb(D_X) \\ (t_1, \dots, t_n) \notin X &\iff (t_1, \dots, t_n) \notin lub(D_X) \end{aligned}$$

Tout comme dans le cadre des ensembles d'entiers, la propriété suivante est toujours maintenue : $glb(D_X) \subseteq lub(D_X)$.

Le domaine approximé peut être modifié en lui appliquant des opérations. Pour un problème donné, ces opérations permettent de préciser de plus en plus le domaine approximé.

Formellement, $D_X^{(i)}$ étant l'approximation du domaine calculé par la i -ème opération, on a les propriétés suivantes :

$$glb(D_X^{(i+1)}) \supseteq glb(D_X^{(i)})$$

et

$$lub(D_X^{(i+1)}) \subseteq lub(D_X^{(i)})$$

Dit autrement, à chaque opération, la plus grande borne inférieure du domaine peut uniquement comprendre plus d'éléments (de tuples) ou rester égale, et la plus petite borne supérieure peut uniquement comprendre moins de tuples ou rester égale. Dès lors, chaque opération peut au mieux améliorer l'approximation du domaine D_X . Lorsqu'on a $glb(D_X) = lub(D_X)$, D_X est un singleton. En effet, il n'existe alors plus qu'une seule relation représentée par D_X .

La différence (au sens de différence dans la théorie des ensembles) entre $lub(D_X)$ et $glb(D_X)$ est la relation qui comprend tous les tuples étant **inconnus** dans le domaine, c'est-à-dire dont on ignore s'ils font effectivement partie de la relation représentée par X ou non. Ils peuvent donc être ajoutés à la relation ground $glb(D_X)$ ou retirés de la relation ground $lub(D_X)$, en fonction des opérations appliquées à D_X . Le prédicat suivant permet de définir cet ensemble :

$$(t_1, \dots, t_n) \in unk(D_X) \iff (t_1, \dots, t_n) \in lub(D_X) \wedge (t_1, \dots, t_n) \notin glb(D_X)$$

En plus de cette relation (ground) $unk(D_X)$, on peut également définir la relation ground des tuples hors des bornes, dit *oob* (de l'anglais *out of bounds*). Cette relation peut être définie en fonction de la relation UB et de \mathcal{U}^n (l'univers de la relation X). Pour $X \in R_n$, en définissant le prédicat suivant :

$$(t_1, \dots, t_n) \in oob(D_X) \iff (t_1, \dots, t_n) \notin lub(D_X) \wedge (t_1, \dots, t_n) \in \mathcal{U}^n$$

De cette définition, on s'aperçoit que $lub(D_X) = \overline{oob(D_X)}$ et donc que si un tuple est retiré de la relation $lub(D_X)$, il est ajouté à la relation $oob(D_X)$. Ceci permet de ne raisonner qu'en

terme d'**ajouts** de tuples à des relations ground ($glb(D_X)$ et $oob(D_X)$) suite à l'application d'une opération sur le domaine d'une variable relationnelle. De manière générale, représenter le domaine en partie en terme de la relation ground $oob(D_X)$ n'est pas praticable, au vu de la grande cardinalité que celle-ci peut prendre. Cependant, les structures de données utilisées de manière interne par Gustavo Gutiérrez justifient ce choix ⁷.

Opérations sur les domaines relationnels

Les opérations ayant été présentées dans le point traitant des relations ground peuvent être étendues aux opérations sur les **domaines relationnels** ⁸. Trois valeurs sont utilisées ici pour désigner l'état d'un tuple t par rapport à une relation X :

- Si on est certain que $t \in X$, on utilise la valeur **T**. Autrement dit, $t \in glb(D_X)$.
- Si on est certain que $t \notin X$, on utilise la valeur **F**. Autrement dit, $t \in oob(D_X)$.
- Dans les autres cas, on ignore l'appartenance ou non de t à X , et on utilise la valeur **U**. Autrement dit, $t \in unk(D_X)$.

L'image 2.1 illustre la manière dont les opérations de complément, d'union et d'intersection sont étendues dans le cadre des domaines relationnels. Les états des tuples dans le/les domaines relationnels passé(s) en paramètre à l'opération sont situés dans l'/les en-tête(s) des colonnes (et des lignes) des différents tableaux. Il est important de noter que ces opérations ont bien pour opérandes des domaines relationnels et non des tuples.

\neg	T	U	F	\vee	T	U	F	\wedge	T	U	F
F	U	T	T	T	T	T	T	T	T	U	F
U	T	U	U	U	T	U	U	U	U	U	F
F	T	U	F	F	T	U	F	F	F	F	F
(A) Complément				(B) Union	(dis-			(C) Intersection	(conjonction)		
(négation)				jonction)				(conjonction)			

FIGURE 2.1 – Extension des opérations de complément (négation), d'union (disjonction) et d'intersection (conjonction) aux domaines relationnels.

En plus des opérations logiques définies pour les domaines relationnels, deux opérations supplémentaires sont définies : l'opération *merge* et l'opération *status*. L'opération *status* ne sera pas redéfinie ici mais l'est bien dans le document [30].

L'opération *merge* prend deux domaines relationnels en paramètre et les combine. Cette opération produit un domaine vide dans le cas où l'état d'un tuple dans un des deux arguments est en contradiction avec l'état dans l'autre. Autrement dit, si pour deux variables relationnelles

7. Ces justifications sortent du cadre de ce mémoire.

8. On entend ici par « domaine relationnel » le domaine d'une variable relationnelle, c'est-à-dire l'approximation de l'ensemble des relations possibles représentées par une variable relationnelle.

X et Y et un tuple t , on a que :

$$(t \in glb(D_X) \wedge t \in oob(D_Y)) \vee (t \in oob(D_X) \wedge t \in glb(D_Y))$$

L'image 2.2 présente la définition de cette opération. L'état **X** désigne le cas dans lequel le domaine relationnel obtenu est vide.

	T	U	F
T	T	T	X
U	T	U	F
F	X	F	F

FIGURE 2.2 – Opération merge sur un domaine relationnel

Un domaine relationnel (c'est-à-dire l'approximation du domaine réel de la relation représentée par la variable relationnelle) vide est équivalent à un domaine vide pour la relation représentée. Dans la programmation par contraintes, un domaine vide pour une variable de décision signifie que les contraintes imposées sur cette variable ne sont pas satisfaisables et donc que le problème de satisfaction de contraintes ne possède pas de solution.

Cette opération se distingue de l'opération d'union, qui produit le meilleur domaine relationnel comprenant toutes les relations comprises dans les domaines représentés par les opérandes.

Ces opérations sur les domaines relationnels permettent de définir des contraintes sur les variables relationnelles, qu'on peut encore appeler des *contraintes relationnelles*. Elles permettent en effet de créer de nouveaux domaines relationnels à partir d'anciens.

Contraintes noyau

Les contraintes noyau sont les opérations de base permettant d'élaguer le domaine de variables relationnelles (c'est-à-dire retirer des valeurs du domaine. Dans notre cas, ce sont des relations ground). Deux contraintes noyau sont définies ici : l'**inclusion d'une relation ground** et l'**exclusion d'une relation ground**.

Inclusion d'une relation ground. Au niveau des variables, inclure une relation ground $G \in R_n$ dans une variable relationnelle $X \in R_n$ se définit comme suit :

$$include(X, G) \equiv \forall t \in G \implies t \in X$$

Si D_X est le domaine relationnel de la variable relationnelle X , alors nous pouvons définir l'inclusion d'une relation ground au niveau des domaines comme suit :

$$D_{incl} = include(D_X, G) = merge(D_X, G \vee Unk)$$

D_{incl} est ici le nouveau domaine relationnel de la variable X . Unk définit quant à lui la relation $\{\forall t \in \mathcal{U}^n : Unk(t) = \mathbf{U}\}$ (c'est-à-dire que pour cette relation, tous les tuples de l'univers (\mathcal{U}^n) considéré ont pour statut la valeur **U**).

Faire l'union d'une relation ground G et de la relation Unk a pour effet de créer un domaine relationnel D_{new} tel que :

$$(glb(D_{new}) = G) \wedge (unk(D_{new}) = \mathcal{U} \setminus G) \wedge (oob(D_{new}) = \emptyset)$$

Dès lors, à partir de la définition de l'opération $merge$, on s'aperçoit que si l'opération ne retourne pas un domaine relationnel vide, on a alors que $glb(D_X) \subseteq glb(D_{incl})$. Dit autrement, on ne peut qu'obtenir une relation dont le domaine est vide ou éventuellement ajouter des tuples à la borne inférieure du domaine relationnel de la variable X .

Exclusion d'une relation ground. Au niveau des variables, exclure une relation ground $G \in R_n$ dans une variable relationnelle $X \in R_n$ se définit comme suit :

$$exclude(X, G) \equiv \forall t \in G \implies t \notin X$$

Comme pour l'inclusion, si D_X est le domaine relationnel de la variable relationnelle X , alors nous pouvons définir l'exclusion d'une relation ground au niveau des domaines comme suit :

$$D_{excl} = exclude(D_X, G) = merge(D_X, \neg G \wedge Unk)$$

Encore une fois, D_{excl} est le nouveau domaine relationnel de la variable X . Faire l'intersection du complément d'une relation ground G et de la relation Unk a pour effet de créer un domaine relationnel D_{new} tel que :

$$(glb(D_{new}) = \emptyset) \wedge (unk(D_{new}) = \mathcal{U} \setminus G) \wedge (oob(D_{new}) = G)$$

Cette fois, si l'opération $merge$ ne retourne pas un domaine relationnel vide, on a alors que $glb(D_{excl}) \subseteq glb(D_X)$. On obtient donc soit une relation dont le domaine est vide, soit on retire éventuellement des tuples de la borne supérieure du domaine relationnel de X .

Exemple pour l'inclusion. En supposant l'univers suivant :

$$\mathcal{U} = \{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 0, 2 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 0 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle\}$$

Soit le domaine relationnel D_R :

$$D_R = \{\langle 0, 0 \rangle\} \dots \{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 0, 2 \rangle, \langle 1, 2 \rangle\}$$

Si l'on veut inclure la relation ground $G = \{\langle 0, 0 \rangle, \langle 0, 1 \rangle\}$ dans D_R , on a alors :

$$G \vee Unk = \begin{cases} \{\langle 0, 0 \rangle, \langle 0, 1 \rangle\} & \text{borne inférieure} \\ \mathcal{U} \setminus \{\langle 0, 0 \rangle, \langle 0, 1 \rangle\} & \text{inconnus} \end{cases}$$

Et donc :

$$merge(D_R, G \vee Unk) = \begin{cases} \{\langle 0, 0 \rangle, \langle 0, 1 \rangle\} & \text{borne inférieure} \\ \{\langle 0, 2 \rangle, \langle 1, 2 \rangle\} & \text{inconnus} \\ \{\langle 1, 0 \rangle, \langle 1, 1 \rangle, \langle 2, 0 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle\} & \text{hors des bornes} \end{cases}$$

Exemple pour l'exclusion. Considérons le même univers \mathcal{U} et le même domaine relationnel D_R que dans l'exemple précédent. Si l'on veut exclure la relation ground $G = \{\langle 0, 1 \rangle, \langle 1, 0 \rangle\}$ de D_R , on a successivement :

$$\neg G = \mathcal{U} \setminus G$$

$$\neg G \wedge \text{Unk} = \begin{cases} \mathcal{U} \setminus G & \text{inconnus} \\ \{\langle 0, 1 \rangle, \langle 1, 0 \rangle\} & \text{hors des bornes} \end{cases}$$

$$\text{merge}(D_R, \neg G \wedge \text{Unk}) = \begin{cases} \{\langle 0, 0 \rangle\} & \text{borne inférieure} \\ \{\langle 0, 2 \rangle, \langle 1, 2 \rangle\} & \text{inconnus} \\ \{\langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle, \langle 2, 0 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle\} & \text{hors des bornes} \end{cases}$$

On peut voir qu'il est possible de créer un domaine relationnel D_V à partir de deux relations ground Lb et Ub (représentant respectivement les bornes inférieure et supérieure), en utilisant l'opération merge :

$$D_V = \text{merge}(Lb \vee \text{Unk}, Ub \wedge \text{Unk})$$

De manière générale, il est habituel de définir des contraintes noyau selon l'inclusion et l'exclusion d'*éléments du domaine*. Dans le cas présent, ceci équivaut à l'inclusion et l'exclusion de tuples. Cette approche a d'abord été utilisée en pratique⁹, mais la granularité au niveau des tuples a conduit à de gros problèmes de performances¹⁰. La granularité a donc été définie au niveau des relations, en représentant le domaine d'une variable relationnelle différemment qu'à l'aide de deux relations ground représentées par un ensemble de tuples. De plus, l'utilisation de relations ground permet un meilleur niveau d'abstraction pour la formulation de contraintes de plus haut niveau. On peut voir l'utilisation de relations ground comme une généralisation de celle des tuples, étant donné que la granularité au niveau des tuples est toujours possible en utilisant des relations ground ne contenant qu'un seul tuple.

Remarque sur la consistance des domaines. On peut constater que la manière d'approximer le domaine d'une relation par un domaine relationnel conduit au même niveau de consistance *maximal* que pour les ensembles d'entiers dans Gecode : la **borne consistance**. En effet, les contraintes noyau ne permettent d'élaguer que les bornes inférieure et supérieure des domaines. Les contraintes de plus haut niveau étant définies en fonction des contraintes noyau, il n'est pas possible de permettre un élagage supérieur à celui de la borne consistance.

2.2.2 Relations dans Gecode

Maintenant que nous avons expliqué la base théorique pour l'utilisation de variables et de contraintes relationnelles, nous pouvons décrire ce qu'il est possible de réaliser avec le système

9. par Gustavo Gutiérrez

10. Parcourir l'ensemble des éléments d'une relation ground pour la modifier est inefficace.

qui a été construit : l'extension de la librairie Gecode pour la **programmation par contraintes relationnelles**¹¹.

Ce système est encore à ce jour en construction et est donc relativement minimal. Nous divisons donc ce qui suit en 3 parties : le système existant, les ajouts dans le futur proche et les ajouts hypothétiques, dans un avenir à plus long terme.

Système existant Dans l'implémentation actuelle, il est déjà possible de construire des tuples, des relations ground et des variables relationnelles. Ceux-ci peuvent avoir une relation maximale de 8, à cause de la représentation interne de celles-ci¹². Remarquons également que les nombres négatifs ne peuvent être utilisés.

Il existent déjà quelques propagateurs :

Si A , B et C sont des variables relationnelles de même arité n , les propagateurs pouvant être utilisés pour le moment sont les suivants :

- * $A = B$. Exprimé en terme de tuples : $\forall t \in \mathcal{U}^n : t \in A \iff t \in B$
- * $A = \overline{B}$. Exprimé en terme de tuples : $\forall t \in \mathcal{U}^n : t \in A \iff t \notin B$
- * $A \cap B = C$. Exprimé en terme de tuples : $\forall t \in \mathcal{U}^n : t \in C \iff t \in A \wedge t \in B$
- * $A \cup B = C$. Exprimé en terme de tuples : $\forall t \in \mathcal{U}^n : t \in C \iff t \in A \vee t \in B$
- * $A \subseteq B$. Exprimé en terme de tuples : $\forall t \in \mathcal{U}^n : t \in A \implies t \in B$
- * $A \cap B = \emptyset$. Exprimé en terme de tuples : $\forall t \in A : t \notin B \wedge \forall t \in B : t \notin A$
- * $\forall t \in \mathcal{U}^n : t \in A \implies t \in B \iff t \in C$

Le seul brancheur existant à ce jour est un brancheur « naïf » : il choisit un tuple dans l'ensemble des tuples inconnus de la relation et crée deux noeuds fils dans l'arbre de recherche, l'un incluant le tuple et l'autre l'excluant.

Futur proche Deux propagateurs essentiels sont en construction actuellement. Ils ne sont donc pas disponibles et n'ont pas été utilisés en pratique dans ce travail. Nous les mentionnons ici en premier car nous en parlons dans la suite de ce document.

Afin de voir plus facilement l'effet de ceux-ci, nous représentons une relation par un tableau. La table 2.1 illustre ceci. Dans cet exemple, la relation est d'arité 3 et les composantes sont nommées c_1 , c_2 et c_3 . Les composantes et les tuples peuvent encore être vues respectivement comme les colonnes et les lignes du tableau.

Le premier des deux propagateurs est exprimé comme :

$$A = \pi_D(B)$$

11. Pour le lecteur intéressé, l'utilisation pratique actuelle des relations dans Gecode est décrite dans l'annexe A.4

12. La raison de ceci sort du cadre de ce mémoire, et ne sera donc pas expliquée ici.

c_1	c_2	c_3
\dots	\dots	\dots

TABLE 2.1 – Représentation d’une relation d’arité 3 par un tableau.

où D est un ensemble de **composantes**, lui-même sous-ensembles des composantes de B . Si B est une relation d’arité n , appliquer ce propagateur revient à imposer que A est égal à la relation d’arité $m \leq n$ telle que :

- $m = \#D$, ou encore $A \in \mathcal{U}^D$
- toutes les « colonnes » de B qui ont pour nom un des éléments de D sont présentes dans le tableau représentant la relation A

Si B est la relation représentée par le tableau 2.1, et qu’on impose :

$$A = \pi_{\{c_1, c_3\}}(B)$$

on force A à être une relation binaire telle qu’elle contient les deux colonnes de B dont les noms sont c_1 et c_3 . A contient donc des sous-tuples des tuples contenus dans B .

Ce propagateur correspond à l’opération de **projection** de l’algèbre relationnelle.

Afin de définir formellement le second propagateur, nous devons tout d’abord définir l’opération de produit cartésien entre deux relations :

$$A \times B = \{(t_a, t_b) : t_a \in A, t_b \in B\}$$

Concrètement, cette opération crée toutes les concaténations possibles entre les tuples de A et de B . Si A et B sont respectivement d’arité p et q , $A \times B$ sera d’arité $p + q$.

Le second propagateur est exprimé comme :

$$A \bowtie_j B = C$$

Et peut être défini par :

$$A \times \mathcal{U}^{q-j} \wedge \mathcal{U}^{p-j} \times B = C$$

où p et q sont respectivement les arités de A et B . Remarquons qu’on a : $C \in \mathcal{U}^{p+q-j}$.

Illustrons ceci par un exemple simple. En joignant les deux tableaux illustrés dans 2.2, par l’opération \bowtie_2 , on obtient le tableau (la relation) 2.3.

Cet exemple illustre le cas pour une relation ground, mais nous parlons bien ici d’un propagateur. Il correspond à l’opération de **jointure** de l’algèbre relationnelle¹³.

13. A strictement parler, la jointure naturelle de l’algèbre relationnelle possède une définition plus générale que celle-ci.

1	23	4	2	5	6	77
11	22	5	6	1000	6	42
11	22	4	2	4	2	7
300	400	4	5			

TABLE 2.2 – Opérandes pour l'opération \bowtie_2

1	23	4	2	7
11	22	4	2	7
11	22	5	6	77

TABLE 2.3 – Résultat de l'opération \bowtie_2

De manière informelle, $A \bowtie_j B = C$ impose donc à la relation C de contenir uniquement tous les tuples de A et B qui peuvent être assemblés par un sous-tuple d'arité j (dans la partie « droite » pour A et dans la partie « gauche » pour B) qui leur est commun.

Afin de pouvoir travailler au niveau de toutes les colonnes avec ce propagateur, un autre propagateur sera également fourni : le propagateur de **permutation**, permettant simplement d'imposer qu'une relation est égale à une autre, mais dont les colonnes ont été permutées.

D'autres propagateurs sont également prévus dans un futur proche, mais nous ne les avons pas utilisés dans ce qui suit, nous ne les présentons donc pas ici. Cependant, leur définition est disponible dans le document [30]. Le fait que nous ne les ayons pas présentés ici ne signifie pas qu'ils ne pourraient pas un jour être utiles dans le domaine étudié.

Futur hypothétique Deux outils sont prévus pour le futur à long terme de cette librairie, bien qu'hypothétiques. Il serait intéressant de les posséder, raison pour laquelle nous les mentionnons ici :

- Programmation par contraintes au niveau des tuples. Autrement dit, les variables de décision sont des tuples (relation de cardinalité 1).
- Contraintes au niveau de la cardinalité des relations.

Deuxième partie

Programmation par contraintes et
musique

Avant-propos

« La musique, jusqu'au 17^{ième} siècle, était une des quatre disciplines mathématiques composant le *quadrivium*, à côté de l'arithmétique, la géométrie et l'astronomie. » [32] Et, comme pour ces autres disciplines, la musique possède ses modèles, propriétés, axiomes, théories, ...¹⁴ Nombreuses ont été et sont ces théories de la musique permettant d'exprimer comment la musique fonctionne et doit être composée. Aucune ne permet d'exprimer ou d'expliquer l'ensemble des oeuvres musicales faisant partie du patrimoine global.

Dans le cas de la musique occidentale telle que la plupart d'entre nous la conçoivent, la théorie a été construite au fil des siècles selon des facteurs qui ne se limitent pas à une théorie mathématique donnée, mais plutôt à plusieurs, ainsi qu'à d'autres facteurs sociaux, économiques, ou encore et surtout à l'influence liturgique. De plus, nous parlons ici d'**une** théorie, mais encore aujourd'hui, elle n'est pas unique.

Il faut donc d'emblée reconnaître que la composition et l'analyse de la musique (occidentale) ne se limitent pas à une théorie que l'on fixe et à laquelle n'importe quelle oeuvre peut se rapporter parfaitement. Par exemple, les notes *accidentelles* ne peuvent pas toujours être expliquées, spécialement dans le jazz. De plus, depuis le tempérament égal, l'accord des instruments n'est plus tout à fait conforme à l'acoustique¹⁵.

Si elle existe, rien ne porte à croire qu'on est sur le point de découvrir une théorie universelle et globalisante expliquant l'ensemble des principes de la musique¹⁶. Nous devons donc pour le moment accepter qu'il faut approximer celle-ci à l'aide des théories « partielles » actuelles et futures, et garder l'avis critique de l'être humain. Autrement dit, on ne pourra créer un morceau de musique selon une ou plusieurs théories uniquement.

On peut se demander pourquoi on devrait s'imposer une théorie. La réponse paraît assez

14. Par exemple, « il existe différentes approches pour expliquer les phénomènes harmoniques [...]. Une approche est basée sur la supposition que les fondamentales des accords indiquent un des sept degrés de la gamme. [...] Une autre approche (souvent appelée harmonie fonctionnelle) accepte seulement trois accords principaux (tonique, dominante et sous-dominante) et explique tous les accords comme des variantes de ces fonctions principales. » [23]

15. Le tempérament égal ne respecte en effet pas le rapport strict des fréquences temporelles dans la manière de définir les rapports entre les notes. La quinte tempérée par exemple, est une quinte dont on soustrait un douzième de comma. [9]

16. Bien que Iannis Xenakis ait tenté de faire table rase et se soit approché d'une théorie globale [39], sa théorie ne possède pas l'accord de tous.

évidente : dans le cas contraire, on ne ferait **que** de la musique totalement aléatoire¹⁷. Or, le passé nous montre qu'il existe des lignes de conduite, des règles, qui plaisent et d'autres moins. Lesquelles devrions-nous utiliser, comment les classer, etc ... sont d'autres questions, que nous ne soulevons pas ici.

Nous voulons seulement dire que se permettre d'utiliser des théories dans la musique nous donne l'avantage de pouvoir utiliser des lignes de conduites et d'en créer de nouvelles, sans avoir à systématiquement repartir de 0. Nous conservons ainsi les connaissances acquises en terme de sonorité et de cognition. Notre compréhension de la musique se développe alors à l'aide d'« expériences » musicales, nous aidant à développer l'une ou l'autre théorie à son sujet. Ces théories peuvent alors être réutilisées pour mener d'autres « expériences » musicales.

S'aider d'une théorie justifie l'utilisation d'un ordinateur dans l'aide à la composition/analyse musicale. Il est clair qu'on peut tout à fait composer sans cet outil (on l'a fait pendant des siècles). Mais, comme pour d'autres domaines, il peut servir de support à l'être humain. Par exemple, prouver un théorème dans la logique des propositions est faisable par l'être humain seul (en théorie). Mais à l'aide d'un ordinateur, ce travail peut être automatisé¹⁸.

Le réalisateur en informatique musicale Serge Lemouton corrobore ce qui vient d'être dit dans [34] : « OpenMusic accélère la génération automatique de matériel musical, une redoutable opération faite « à la main ». » Générer du matériel musical répondant à certains critères désirés par le compositeur est une tâche qui peut en effet être bien plus aisément réalisée de manière semi-automatique plutôt qu'à la main. Il paraît cependant clair qu'il faudra faire un traitement a posteriori du résultat obtenu, la théorie formelle permettant d'obtenir celui-ci ne suffisant pas.

Le compositeur Johannes Kretz nous fait bien prendre conscience que l'ordinateur aide l'être humain mais ne va pas plus loin. Il s'agit bien d'un processus interactif et non automatique : « Ceci montre l'interactivité du processus de création : le compositeur exprime formellement ses idées par des règles et des désirs. La machine génère une ou plusieurs solutions. Y étant confronté, le compositeur prend conscience des problèmes esthétiques et affine son système de règles. Le résultat musical est optimisé en répétant cette boucle d'interaction entre humain et machine. » [33]

Finalement, on pourrait ajouter qu'admettre que la machine puisse composer automatiquement (seule) rejoint l'idée que la machine possède la faculté de penser.

17. Et quelque part, imposer qu'il n'y a pas de théorie, c'est déjà une théorie. Auto-contradiction.

18. Maude [10] et Prover9 [11] sont deux exemples de langages permettant de le faire.

Choix de la programmation par contraintes

Plusieurs outils informatiques existent pour le travail musical : des séquenceurs¹⁹, des éditeurs de partitions²⁰ ou encore des langages de programmation²¹.

Dans le cadre de la composition et de l'analyse de la musique occidentale, la programmation par contraintes semble tout à fait appropriée. En effet, le caractère combinatoire de la représentation et de l'écoute de cette musique justifie son utilisation. Le compositeur Fabien Lévy décrit cette propriété générale de la musique occidentale : « La pensée analytique est caractérisée dans la musique occidentale en particulier par une séparation des paramètres musicaux en rythme, hauteur, et durée, par une réduction de phénomènes continus et complexes vers un alphabet fini et discontinu, par une pensée fonctionnelle, et en conséquence, en considérant le signe d'une manière combinatoire. » [35]. Conjointement à cela, il est en accord avec le fait que l'ordinateur est une aide pour le compositeur : « Dans certains cas, lorsque le processus musical est suffisamment verbalisé et formalisé, l'ordinateur permet au compositeur de se concentrer sur la musique et de passer moins de temps sur le calcul. » [35]. Remarquons cependant que « la délégation de sous-tâches de composition à l'ordinateur résulte souvent en une production musicale différente de la musique composée purement manuellement ». [23]

La programmation par contraintes permet à l'utilisateur de se focaliser sur ce qu'il désire de la musique qu'il compose, sans réellement se soucier de la manière dont le résultat est calculé par l'ordinateur. Ceci constitue un avantage certain : le seul travail nécessaire est la modélisation du problème musical. Le compositeur se concentre sur ce qui l'intéresse, la musique, et non sur la manière d'obtenir le résultat. La restriction principale imposée au compositeur est de travailler avec des nombres entiers, quoi que ces nombres représentent dans son esprit.

Les relations : une nouveauté dans le domaine d'étude

Beaucoup de systèmes utilisant la programmation par contraintes ont été développés afin de se placer dans un environnement de recherche combinatoire. Certains d'entre eux sont décrits dans l'annexe A.5 pour le lecteur intéressé.

Comme dit dans le chapitre 2, seuls les entiers, les booléens et les ensembles d'entiers sont utilisables en tant que variables de décision dans la programmation par contraintes pour le moment. Nous le verrons, ceci donne une limitation sur la façon de représenter la musique, ainsi que sur la manière dont on peut contraindre le matériel musical.

Johannes Kretz met d'ailleurs ce problème en avant : « Une des difficultés [...] était le besoin d'une représentation plus structurée des données musicales. Traiter les paramètres

19. Ex : Cubase, Ableton Live

20. Ex : Guitar Pro, TuxGuitar

21. Comme Nyquist, qui permet de synthétiser, analyser et enregistrer des sons.

indépendamment (comme des listes de valeurs « midicent²² » pour la hauteur, qui ignorent l'existence des restes, des listes de fractions pour les durées, où les restes sont représentés par des nombres négatifs, etc.) ont rendu difficile l'interaction de ces paramètres. Dans le futur, une structure de données riche, hiérarchiquement organisée, pour les notes (un paquet d'information contenant de nombreux paramètres comme la hauteur, la durée, l'intensité, le nombre de voix, la position dans la mesure, des marques d'expression, un contexte harmonique, etc.) devrait aider à créer des règles d'autant plus « intelligentes ». » [33]

Ce que propose le compositeur dans la fin de sa remarque semble pouvoir être en partie abordé par l'utilisation de relations. Le « paquet d'information » dont il parle pourrait être représenté à l'aide d'un tuple : celui-ci lie en effet plusieurs entiers entre eux par la relation dont il est un élément. En étendant GeLisp avec un nouveau type de variable et en l'utilisant conjointement avec OpenMusic, on peut donc s'attendre à un gain au niveau de la modélisation de problèmes musicaux.

Dans le chapitre 3, nous décrivons ce qui existe aujourd'hui dans le domaine étudié. Nous décrivons ensuite dans le chapitre 4 ce que les relations ont à apporter dans celui-ci.

22. Centième d'une unité MIDI (Musical Instrumental Digital Interface).

Chapitre 3

Etat de l'art

Dans ce chapitre, nous décrivons de manière générale comment la composition à l'aide de la programmation par contraintes peut être réalisée aujourd'hui. Pour le lecteur intéressé, deux systèmes existants sont globalement décrits dans l'annexe A.5.

3.1 Généralités

Nous l'avons dit, beaucoup de théories de la musique existent. Dans beaucoup de cas, ces théories sont basées sur des **règles de composition**¹. Par exemple, à la fin du Moyen Age, le *triton* (intervalle de trois tons), ou encore le « Diabolus in Musica », était évité, voire interdit, au vu de sa sonorité. Etant donné que les règles de composition sont un concept établi depuis longtemps, les approches de programmation se basant sur des règles ont généralement plus attiré l'attention des compositeurs. De plus, elles permettent à ceux-ci de travailler dans un domaine de haut niveau, sans nécessairement posséder de grandes connaissances dans le domaine de la programmation.

La programmation par contraintes a montré qu'elle était un paradigme assez concluant pour réaliser des systèmes basés sur les règles. Dans le cadre de la composition musicale à l'aide des règles de composition, le paradigme semble tout à fait adéquat ; une tâche de composition est déclarée par :

- une représentation de la musique dans laquelle **certains aspects musicaux** sont *inconnus*. Ils seront dès lors représentés par des **variables de décision**.
- des règles de compositions. Elles imposeront des **contraintes** sur ces variables.

Dans sa thèse, Torsten Anders introduit la notion de **problème de satisfaction de contraintes musical** : celui-ci implémente un modèle de théorie musicale comme un programme. Lorsque ce programme est exécuté, il génère de la musique qui

1. Bien que nous préférons le terme « propriété musicale », nous continuerons à employer ce terme, à l'instar de Torsten Anders dans sa thèse [23], car celle-ci a grandement aidé le développement de ce chapitre.

se soumet à la théorie modélisée.

Un modèle de théorie musicale implémenté par un problème de satisfaction de contraintes musical doit être totalement formalisé mathématiquement. Par contre, ce modèle ne doit pas forcément toujours être consistant avec de la musique existante, ce qui est en soi un point de vue plus général. Par exemple, un compositeur peut résoudre un **MCSP**² dont la solution n'est pas destinée à être utilisée dans une pièce, mais qui servira à l'élaboration d'une nouvelle composition plus générale.

Torsten Anders définit la notion de **système de contraintes musical générique**. Il s'agit d'un système de contraintes qui définit au préalable pour l'utilisateur un ensemble de connaissances et bases musicales, partagées par beaucoup de MCSP. Il permet alors de simplifier la définition des problèmes auxquels on désire s'attaquer. Deux exemples sont donnés dans l'annexe A.5.

L'utilisation d'OpenMusic couplée avec GeLisp ne rentre pas dans cette définition, puisqu'il faut tout redéfinir pour chaque MSCP. Cependant, GeLisp peut tout à fait servir de base à la construction d'un tel système.

Un système de contraintes musical est plus générique qu'un autre s'il permet d'implémenter plus de MSCP. De là découle la définition de **système de contraintes musical le plus générique** ; il permettrait à l'utilisateur d'implémenter n'importe quel modèle de théorie musicale concevable. En plaçant de côté les performances, un tel système serait idéal. Il ne restreint en effet en rien les possibilités de composition. Autrement dit, l'utilisateur n'est restreint à aucun type de MCSP. GeLisp, muni des relations, pourrait éventuellement aller dans ce sens.

Dans les systèmes existants, la représentation de la musique limite l'information explicite des partitions pouvant être enregistrée et le type d'information dérivée à laquelle on a accès. Beaucoup de systèmes ne permettent par exemple que de faire de la musique séquentielle. L'accès à de l'information dérivée telle que savoir si une note est sur un temps faible ou non, est également restreinte.

De plus, les stratégies de recherche sont généralement optimisées pour des classes spécifiques de MCSP. De là, certaines classes de MCSP, même si elles sont représentables dans le système, ne sont pas solubles de manière suffisamment efficace. La stratégie de recherche de Situation [24] par exemple est optimisée pour son format de représentation de la musique.

GeLisp, étant actuellement un système de contraintes général, ne possède pas ce type d'inconvénients, puisqu'il n'existe pas encore de couche d'un plus haut niveau. La limitation au niveau de la représentation musicale est celle d'OpenMusic, puisque c'est dans cet environnement qu'on utilise GeLisp.

Strasheela, le système de contraintes musical développé par Torsten Anders, se veut plus générique en étant plus programmable que les systèmes précédents sa réalisation. Il s'est pour cela appliqué à rendre la représentation de la musique, les règles appliquées aux partitions et

2. Pour raccourcir le discours, nous utiliserons un acronyme de « Musical Constraints Satisfaction Problem » à la place de problème de satisfaction de contraintes musical.

les stratégies de recherches plus programmables. Le système qu'on pourrait créer en se basant sur GeLisp, pourrait suivre cette idée, en se voulant encore plus générique que les systèmes précédents.

3.2 Approche déclarative de la composition

Une formalisation **déclarative** est généralement préférable à une formalisation procédurale. Souvent, les programmes déclaratifs :

- peuvent être facilement composés entre eux.
- sont plus simples pour raisonner.

L'approche déclarative est souvent utilisée dans le domaine de la musique, sans même entrer dans le secteur de la musique assistée par ordinateur. Certains théoriciens donnent des explications sur la composition en ne décrivant que les propriétés à respecter sans pour autant décrire la procédure pour arriver à un résultat satisfaisant ces propriétés.

L'idée de règle de composition découle de cette approche déclarative pour expliquer la connaissance musicale. Ces règles régissent le produit de la composition, et plus précisément les *objets d'une partition* - comme par exemple des ensembles de notes - ainsi que leurs paramètres.

L'avantage des règles de composition, est qu'elles permettent de décrire la nature multi-dimensionnelle de la musique de façon **modulaire**. Dans la musique, différents aspects sont perceptibles de manière simultanées : le rythme, l'harmonie, le timbre, Ces différentes dimensions de la musique sont inter-connectées et nécessitent d'être prises en compte de manière globale ainsi que selon différents points de vue lors de la composition. La formalisation de la composition se trouve grandement aisée lorsqu'on peut la réaliser de manière modulaire. Ainsi, si l'on désire modifier cette formalisation en ajoutant ou en retirant une règle de composition, l'effort pour cette tâche se trouve diminué. Ce n'est généralement pas le cas si l'on formalise un ensemble d'éléments (musicaux) inter-connectés en suivant une approche procédurale.

La programmation par contraintes étant un paradigme efficace pour réaliser un système basé sur les règles, elle a donc tout à fait sa place dans le domaine de la composition assistée par ordinateur. Seul le résultat est exprimé de manière déclarative selon des règles. Le compositeur ne s'attarde donc qu'à exprimer ce qu'il désire en termes de musique, sans avoir à se préoccuper de la manière dont le résultat est obtenu. Remarquons cependant que si on en a la possibilité, il est toujours mieux de comprendre ce qui se passe de manière interne. Cela dit, les utilisateurs de tels systèmes ne sont pas tous intéressés par ceci et ne se préoccupent effectivement que du résultat obtenu.

Remarque sur les contraintes faibles. Nous devons préciser que, comme beaucoup de théoriciens

l'ont écrit, les règles ne sont que des lignes de conduite [23]. Elles ne sont pas absolues en soi. Certains systèmes permettent donc d'avoir des **contraintes faibles**. Celles-ci sont des contraintes qui ne sont pas obligatoirement tout à fait respectées, mais qui sont respectées autant que possible.

Une contrainte faible possède un coût. En fonction de sa nature et de la manière dont elle est violée, ce coût est plus ou moins grand pour une solution donnée. Le but est alors de minimiser les coûts autant que possible.

Ceci est réalisable avec GeLisp en utilisant de la réification de contraintes. On utilise alors une somme des booléens permettant de réifier les contraintes, et on essaye de la minimiser³.

Cependant, l'élagage des domaines n'est alors pas aussi efficace (nettement moins dans certains cas) que pour des contraintes fortes. En effet, on risque parfois de descendre très profondément dans l'arbre de recherche avant de se rendre compte qu'une valeur est trop mauvaise pour une variable donnée (elle viole trop une contrainte pour cette variable), alors qu'avec une contrainte forte on aurait directement enlevé la valeur du domaine. Il faut en effet attendre d'être sûr que la somme des coûts soit suffisamment mauvaise pour retirer la valeur du domaine.

3.3 Classes de problèmes solubles

Nous finissons cette section en donnant un aperçu de tout ce qu'il est possible de contraindre dans la musique. Nous donnons donc ici des **classes de problèmes** pouvant être indépendants, ou liés. « Parfois des points communs apparaissent dans les structures musicales utilisées, et donc dans les modélisations des problèmes, parfois, plusieurs problèmes, semblant de prime abord apparentés, ne peuvent pas du tout se formuler de la même manière. » [38]

Premièrement, il est possible de résoudre des problèmes de **contrepoint**. Le contrepoint, discipline d'écriture musicale, vise à superposer entre elles plusieurs lignes mélodiques. Il s'agit donc d'une technique axée sur la polyphonie. Beaucoup de règles et de théories différentes existent pour cette discipline. Certaines prennent peu en compte la dimension harmonique, alors que d'autres montrent comment composer en fonction d'elle.

La théorie de Johann Joseph Fux est définie selon un certain nombre de règles strictes. Elle peut donc être aisément modélisée à l'aide de contraintes fortes. Ceci montre en quoi le contrepoint a pu être intéressant pour la composition assistée par ordinateur basée sur des règles.

Voici un exemple de règle dans ce domaine :

$$\bigwedge_{i=0}^{\#bars-1} \text{distinct}(\text{localMaximaInBarsFromTo}(i, i + 2))$$

Elle exprime que les pics mélodiques dans trois mesures consécutives sont tous distincts.

3. On suppose ici que la valeur du booléen associé à une contrainte est réifiée vaut 0 si la contrainte est respectée.

Les problèmes traitant de l'**harmonie** sont sans doute ceux qui ont suscité le plus grand intérêt des compositeurs dans le domaine de la composition assistée par ordinateur. Et bien que l'évolution de l'harmonie soit plus continue historiquement, différentes théories existent à son sujet. Les règles employées peuvent donc différer en fonction de l'utilisation de la théorie employée. De plus, le développement de l'harmonie est à ce jour encore en progression, et beaucoup de recherches ont été engagées au niveau de l'harmonisation basée sur des contraintes.

Charlotte Truchet nous décrit le problème d'harmonisation automatique comme suit : « Le problème de contraintes musicales le plus classique est celui de l'harmonisation automatique. Il s'agit de réaliser un exercice d'harmonisation à quatre voix, à chant donné ou à basse donnée par exemple. Rappelons qu'une harmonie doit respecter de nombreuses règles, consignées dans de multiples traités, de sorte que l'ensemble sonne correctement (selon les critères classiques). Ces règles imposent des structures verticales précises, les accords, dont la forme est codifiée, et restreignent en plus certaines combinaisons de notes. [38] »

Dans les problèmes d'harmonisation, les variables sont généralement des entiers représentant des hauteurs de notes ou des intervalles entre ces notes. On peut également supposer qu'un ensemble d'entiers peut représenter un accord (si les entiers sont des hauteurs de notes), ou un type d'accords (si les entiers représentent des intervalles). Ce qui est représenté par une variable dans la musique reste ici très **local** dans la partition.

Un exemple de problème harmonique présenté par Charlotte Truchet dans sa thèse est le problème du **classement d'accords**, défini comme suit :

« On part d'une séquence de n accords, $A_1 \dots A_n$, par exemple tirés aléatoirement dans un ambitus donné. Le but est de trouver une permutation des A_i maximisant le nombre de notes communes entre deux accords successifs. »

Une autre classe de problèmes est celle qui traite de **mélodie**, ou encore des mouvements horizontaux de la musique. Il faut avant tout admettre que l'écriture de mélodies est fortement dépendante d'un style de musique et que ce sujet n'est pas enseigné de manière traditionnelle. Certains pensent d'ailleurs que la mélodie n'est qu'une conséquence de l'harmonie sous-jacente (alors que d'autres pensent le contraire).

Ceci explique sans doute pourquoi peu de systèmes existent au niveau de la composition de mélodies. Un exemple général de ce type de problème est la génération d'une mélodie en fonction de l'harmonie. Celle-ci est alors une donnée du problème à résoudre.

On peut spécialement contraindre les intervalles horizontaux, c'est-à-dire les intervalles entre deux notes successives. Ceci est toujours réalisé à l'aide de nombres entiers, et on peut remarquer que les contraintes sont alors toujours très locales sur la partition. Un exemple de ce type de problème est la série tous-intervalles, décrite dans la sous-section 1.2.3.

Jusqu'au 20^{ième} siècle, il existe peu de théories dans la musique occidentale au niveau du **rythme**. Il existe donc peu de règles de composition dans cette dimension jusqu'alors. S'en est alors suivi un siècle de développements considérables, notamment grâce à Igor Stravinsky ou à

Olivier Messiaen.

Dans le cadre de MCSP, les contraintes peuvent porter sur les signatures permises, le nombre distinct de celles-ci, le nombre de changements permis, etc ... Le problème peut également se baser sur des motifs ayant un rôle structurel de l'oeuvre. Un exemple de problème purement rythmique donné par Charlotte Truchet dans son travail est celui des « Durées dans Fibonacci » : « On suppose donnés une unité de temps et un entier D . Le but est de trouver n durées qui sont des valeurs de la suite⁴, telles que deux durées voisines soient voisines dans la suite (en avant ou en arrière), et que la durée totale soit égale à D . » [38]

Certains problèmes tentent d'approcher l'**analyse musicale** par la programmation par contraintes. On pourrait définir de manière informelle et restreinte l'analyse musicale comme l'étude d'une oeuvre et des paramètres musicaux présents dans celle-ci, ainsi que des paramètres (ou de l'environnement) ayant permis de la créer. La programmation par contraintes ne permet évidemment pas de travailler dans tous les domaines de l'analyse musicale (l'analyse biographique de l'auteur est un exemple évident), mais peut aborder, au moins en théorie, un bon nombre d'entre eux. Un aspect intéressant peut être la capture des paramètres présents implicitement dans une partition, sans les connaître a priori. Par exemple, tenter de dégager le tempo, la signature, la tonalité etc ... d'une oeuvre. Charlotte Truchet présente également des problèmes visant à reproduire le plus exactement possible des techniques utilisées par des musiciens. Une mesure de qualité d'un modèle pour un problème donné est alors le nombre de solutions. Au moins il y en a, au plus elles se rapprocheront de l'aspect étudié. Le cas idéal survient lorsque la solution est unique, et qu'elle représente parfaitement la partition étudiée (si elle existe).

D'autres problèmes s'axant autour de contraintes physiques existent également. Ce sont des MCSP abordant l'**instrumentation**. Par exemple, pour un MCSP donné, on peut ajouter des contraintes permettant à la composition d'être jouée par un guitariste. Celles-ci découlent du fait que la guitare ne possède que 6 cordes, que la taille de la main humaine est limitée, que le temps nécessaire pour déplacer sa main sur le manche possède une borne inférieure, etc ... Remarquons que pour le cas des doigtés pour la guitare, « Au premier abord l'énoncé est simple : il s'agit de trouver des doigtés pour jouer à la guitare une harmonie donnée. Mais en fonction du but recherché, les modélisations possibles sont nombreuses. » [38] On peut le voir dans la thèse Charlotte Truchet, comme pour les CSP généraux, les MCSP peuvent parfois être modélisés de manières différentes.

D'autres types de problèmes, qui ne sont pas directement musicaux mais peuvent être liés à la musique, existent également. Par exemple, la correction d'erreurs lors de reconnaissance spatiale, durant une tâche de spatialisation du son.

Pour résumer, beaucoup de paramètres connexes de la musique peuvent être contraints. Parmi ces paramètres, on peut trouver : la durée des notes, la texture musicale (monodique, polyphonique, ...), la hauteur des notes, l'instrumentation, le timbre (par synthèse du son), les règles contrapuntiques en vigueur, etc ...

4. de Fibonacci

Chapitre 4

Apport des relations

« *Mathematical relations [...] are the bread-and-butter of compositional rules [...]* »
Torsten Anders

Après avoir donné une description générale de ce qui existe aujourd’hui dans le domaine de la programmation par contraintes appliquée à la musique, nous décrivons ce que les variables relationnelles peuvent apporter. Tout d’abord, nous décrivons les nouvelles possibilités théoriques. Ensuite, nous parlons du système en pratique, en donnant deux cas concrets et quelques courts exemples. Finalement, nous traitons un problème mis en avant par Torsten Anders dans sa thèse, le « problème de contexte de partition inaccessible » [23].

4.1 En théorie

Nous décrivons dans ce chapitre ce que les relations permettraient en théorie de réaliser dans le domaine musical. Bien que l’arité maximale des relations pouvant être utilisées aujourd’hui est de 8¹, nous traiterons ici de relations n -aires, dans un but de généralité.

4.1.1 Représentation d’une partition complète par une relation

Une note de musique (ou même plus généralement un son) peut être décrite par un ensemble de paramètres. Lorsqu’on la représente par un entier, on perd inévitablement un ensemble d’informations importantes. On se limite en effet à un seul de ses paramètres, et donc à une seule dimension de la musique. Par exemple, si l’entier représente la hauteur de la note, on perd inévitablement sa durée, et vice-versa.

L’idée suggérée par Johannes Kretz² devient en partie réalisable à l’aide de relations : la représentation des notes ne se limite plus à un seul entier. En effet, un tuple peut contenir un ensemble **ordonné** d’entiers. Puisqu’ils sont ordonnés, on peut assigner à chaque composante

1. Implémentation de Gustavo Gutiérrez

2. Voir la section à la page 39.

de ce tuple un paramètre de la note.

Formellement :

$$tuple = \langle param_1, \dots, param_n \rangle$$

qui dans notre cas, peut par exemple être :

$$tuple = \langle hauteur, duree, attaque, nuance, instrument, \dots \rangle$$

Si un seul tuple permet de représenter une note sur une partition, un ensemble non-ordonné de tuples permet de représenter un ensemble de notes, et donc une partition complète. Une seule relation suffit donc à représenter une partition complète.

Un exemple permettra de se convaincre de la conséquence de ceci au niveau de la modélisation de problèmes musicaux.

Soit une relation binaire, dont les composantes représentent respectivement la hauteur et le moment d'attaque³, et dont les domaines sont respectivement $60 \dots 72$ et $0 \dots 7$. Si la relation est totale, on a sur la partition toutes les notes allant de do à do, et ce sur les 8 premiers temps (on suppose ici une signature 4/4 et la noire comme durée pour les notes. Les autres paramètres ne sont pas pris en compte pour l'exemple.) . Ceci est illustré à la figure 4.1.



FIGURE 4.1 – Relation totale représentant une partition

En élaguant le domaine de cette relation, on peut alors tenter de conserver les notes qui respectent les règles de composition que l'on désire, autrement dit, les contraintes. On s'approche d'une certaine manière de l'esprit de composition à la main : on a tout un ensemble de notes à disposition, de « matériel musical », mais on ne désire en utiliser qu'une certaine partie, celles qui respectent les règles de composition.

Par exemple, on peut appliquer une contrainte exprimée comme suit :

$$\bigwedge_{i=0}^7 \#(r_{score} \bowtie \langle i \rangle) \leq 3$$

où $(r_{score}$ est la relation représentant la partition, \bowtie est l'opération de jointure dans l'algèbre relationnelle, et où $\langle i \rangle$ est un tuple unaire ne contenant que la valeur i .

Concrètement, cette contrainte exprime le fait qu'on accepte 3 notes simultanées au maximum.

3. Dans ce document, lorsque nous parlons de moment d'attaque, nous parlons de l'instant à laquelle une note débute (en référence à l'attaque, dans l'enveloppe d'un son). Le terme anglais est *onset*, et nous ne connaissons pas de meilleur équivalent en français.

Remarque à propos de l'idée développée et de l'utilisation des fichiers MIDI avec OpenMusic. Nous remarquons ici que l'idée décrite dans cette sous-section se prête très bien à l'utilisation des fichiers MIDI dans OpenMusic.

OpenMusic permet l'importation de fichier MIDI via l'objet *MIDIFILE*. Une fois importé, il est possible de récupérer l'information contenue dans le fichier sous la forme d'une liste de *piste*. Chaque piste est une liste de notes. Et chacune de ces notes est une liste de paramètres de la forme :

$$(midiPitch, onset, duration, velocity, channel)$$

où

- *midiPitch* est une valeur entière représentant la hauteur de la note, selon le format MIDI.
- *onset* est le moment d'attaque de la note, c'est-à-dire le moment où la note commence à être entendue. Cette valeur est donnée en milliseconde.
- *duration* est la durée de la note, en milliseconde.
- *velocity* permet de représenter numériquement l'intensité d'une note.
- *channel* est un des 16 canaux MIDI disponibles. Un canal permet de définir l'instrument alloué à une note.

Chaque piste peut donc être vue comme une relation ground 5-aire. Ceci permet donc de travailler avec des pièces existantes, codées dans un fichier MIDI.

4.1.2 Représentation d'une transformation d'une partition par une relation

Une autre manière d'utiliser les relations dans le domaine de la musique est de les voir comme une transformation d'objets musicaux⁵ en d'autres objets musicaux. Nous développons ici cette idée de manière théorique.

Définition et application d'une transformation

Par transformation d'un objet musical, nous entendons la création d'un objet musical à partir d'un autre, c'est-à-dire l'application d'une **fonction** sur un objet musical. Cette fonction retourne des valeurs entières en fonction d'autres.

Afin de démarrer simplement, nous allons tout d'abord transformer une seule note : à partir d'une note dont la hauteur est $pitch_i$ et le moment d'attaque est $onset_i$, on peut obtenir une nouvelle note dont la hauteur est $pitch_j$ et le moment d'attaque est $onset_j$. Ceci est donc réalisé en appliquant une fonction sur les paramètres de cette note pour en obtenir d'autres.

5. Voir la définition dans la sous-section 4.1.1. Par contre, dans la sous-section présente, nous parlons d'objet musical essentiellement en tant qu'ensemble de notes, représentée par une relation. Il est cependant tout de même possible d'utiliser ce qui est décrit ici de manière plus générale, raison pour laquelle nous gardons le terme objet musical.

Or, une fonction peut être appliquée à toutes les notes faisant partie du « domaine » de cette fonction. Elle peut donc être appliquée à un ensemble de notes et donc à un objet musical. On peut donc transformer un objet musical pour en obtenir un autre grâce à une seule fonction.

La représentation de cette fonction/transformation peut se faire à l'aide d'une relation. L'application de cette transformation peut quant à elle être réalisée à l'aide de l'opération \bowtie de l'algèbre relationnelle. Le résultat s'obtient ensuite en utilisant l'opérateur π de l'algèbre relationnelle. Afin d'illustrer ceci, nous allons prendre un exemple simple. Prenons un accord, représenté par une relation binaire :

$$chord = \{\langle pitch, onset \rangle\}$$

Une relation représentant une transformation pourrait être définie comme suit :

$$T = \{\langle pitch_{start}, onset_{start}, pitch_{end}, onset_{end} \rangle\}$$

Afin d'obtenir le résultat de l'application de la fonction/transformation sur cet accord, nous utilisons les opérations \bowtie et π ⁶ :

$$transformedChord = \{\pi_{pitch_{end}, onset_{end}}(chord \bowtie T)\}$$

transformedChord est donc une relation binaire, représentant l'accord *chord* transformé par la relation *T*.

Il est à noter que nous pouvons être plus général que cela. En effet, les fonctions ne sont qu'un cas particulier des relations. Ceci a deux implications importantes :

- un objet musical peut être transformé en plusieurs objets musicaux.
- si un objet musical est transformé en un deuxième objet musical par une transformation, celle-ci transforme également le second vers le premier. Dit de manière plus générale, ils sont **liés** par cette relation.

Différentes possibilités

Nous venons de voir que les relations permettent de représenter des transformations de manière très générale. Nous montrons ici quels sont les différents types de transformation qu'on pourrait appliquer.

Tout d'abord, si l'on représente les objets musicaux par des relations *n*-aires, une transformation doit être représentée par une relation *2.n*-aire afin d'encapsuler totalement toute

6. A strictement parler, on suppose ici qu'on a d'abord renommé les composantes *pitch* et *onset* de la relation *chord* respectivement en *pitch_{start}* et *onset_{start}*.

l'information contenue dans cette transformation⁷.

Schématiquement, la transformation de l'objet o_1 en l'objet o_2 , qui est également une transformation de l'objet o_2 en l'objet o_1 peut être représentée par la relation :

$$transformation = \{\langle param_1^{o_1}, \dots, param_n^{o_1}, param_1^{o_2}, \dots, param_n^{o_2} \rangle\}$$

qui est bien une relation $2.n$ -aire.

Durant la recherche de solutions, modifier le domaine des paramètres de o_1 par l'exécution d'un propagateurs, a des répercussions sur le domaine des paramètres de o_2 , et **vice-versa**. Ceci montre l'intérêt d'utiliser des relations pour représenter des transformations d'objets musicaux.

Une transformation peut également ne transformer qu'un sous-ensemble des paramètres de l'objet musical. Cette transformation ne s'applique alors qu'à un sous-ensemble des composantes de la relation représentant l'objet musical. La relation représentant la transformation est alors d'arité $2 * k$, k étant le nombre de composantes transformées ($k < n$).

Un exemple trivial serait de transformer la hauteur d'une note de musique uniquement pour créer une autre hauteur de note. Quel que soit le nombre de paramètres permettant de représenter la note, la transformation de cette note sera alors binaire :

$$transformation = \{\langle hauteur_{depart}, hauteur_{arrivee} \rangle\}$$

Il est également possible d'utiliser des relations qui ont des arités qui ne sont pas forcément multiples de 2 : si on utilise certaines composantes d'un objet musical pour modifier certaines autres composantes d'un autre objet musical. Par exemple, utiliser la hauteur de note et la durée d'un objet musical pour définir la nuance d'un autre objet musical.

Nous devons remarquer ici que de l'information au sujet d'un objet musical peut soit être encapsulée dans la relation, soit être comprise dans le modèle. Si l'on prend l'exemple simple d'une suite de 3 accords ayant lieu aux premiers, deuxième et troisième temps d'une mesure (voir figure 4.3), on peut utiliser différentes modélisations.

Premièrement, on peut utiliser une relation binaire dont les composantes représenteront la hauteur de note et le moment d'attaque d'une note. Cette relation seule suffit pour définir la suite d'accords : il suffit de poser une contrainte obligeant les notes à avoir lieu aux premier, deuxième et troisième temps de la mesure.

Deuxièmement, on pourrait représenter le problème comme suit : le premier accord est un ensemble d'entiers, le second est obtenu par la transformation du premier via une première relation binaire, et le troisième est une transformation du second via une seconde relation binaire.

7. à moins de coder différents sous-ensembles des composantes de cette relation par des entiers (à l'aide de fonctions bijectives) et de créer une nouvelle relation possédant pour composantes ces codes. L'arité de la relation est alors diminuée mais le domaine pour chaque nouvelle composante est alors de cardinalité au moins aussi grande que le produit des cardinalités des domaines des composantes codées dans ces nouvelles composantes.

Dans le premier cas, les moments d'attaque des notes seront décidés au moment de la recherche. Dans le deuxième cas, on décide dans le modèle, quelles hauteurs de notes (non-décidées au moment de la modélisation) sont associées aux différents temps de la mesure.

La manière dont on utilise une relation pour modéliser un problème devrait en pratique être décidée en fonction de ce qui est le plus efficace au moment de la recherche de la solution.

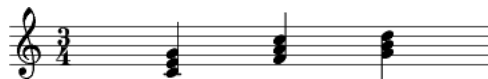


FIGURE 4.3 – Suite de 3 accords. Une première représentation de ceux-ci peut être une relation binaire $\{pitch, onset\}$, dans laquelle chaque tuple est une note. Une seconde représentation est la suivante : le premier accord est un ensemble d'entiers, chaque entier étant la hauteur d'une des notes de l'accord. Le deuxième accord est construit en transformant le premier à l'aide d'une relation binaire $\{pitch_{init}, pitch_{end}\}$. Le troisième est construit à partir du second à l'aide d'une autre relation similaire.

Nous avons dit précédemment que *lier* était peut-être un terme plus approprié pour la transformation entre deux objets musicaux. On peut pousser cette idée plus loin : une relation pourrait, au lieu de lier (certains paramètres de) deux objets musicaux uniquement, lier (certains paramètres de) k objets musicaux. On lie alors plusieurs objets musicaux par une « transformation multi-directionnelle ». Grâce à cela, on peut exprimer des choses qui deviennent de plus en plus générales. Par exemple, on pourrait représenter le lien entre les différentes voix d'une fugue à quatre voix par une seule relation.

Cependant, il faut remarquer que l'arité de ces relations devient de plus en plus grande. En effet, pour représenter totalement le lien de k objets représentés par une relation n -aire, il est nécessaire d'utiliser une relation d'arité $n * k$.

Récupération d'une transformation à partir d'une partition existante : un apprentissage par la machine

Jusqu'ici, nous avons décrit comment **utiliser** une relation lorsqu'elle représente une transformation d'objets musicaux. Cependant, une question subsiste : comment, à partir d'une partition donnée, récupérer cette transformation. Ceci peut sembler a priori assez simple mais ne l'est en fait pas du tout.

Pour le montrer, prenons deux mesures d'une partition, et représentons les par deux relations ground ternaires bar_1 et bar_2 , dont les composantes sont la hauteur de note, le moment d'attaque et la durée. Coder l'information contenue par ces deux mesures par les relations bar_1 et bar_2 n'a rien de compliqué. Il suffit, pour chaque note de créer un tuple contenant l'information requise et de l'insérer dans la relation correspondante.

Tentons maintenant de définir une relation d'arité 6 contenant la transformation totale de bar_1 en bar_2 (qui rappelons-le est également la transformation de bar_2 en bar_1). Nommons cette

transformation T .

Un tuple t_t (arité 6) de T est donc formé d'un tuple t_1 (ternaire) de bar_1 concaténé à un tuple t_2 (ternaire) de bar_2 . Ainsi, une des notes de bar_1 est liée à une note de bar_2 par T . Mais comment choisir ces deux notes ? En fonction de quels critères ? Comment affirmer que c'est bien la note t_1 qui est liée à t_2 et pas une autre note t_1' ? Et pourquoi pas les deux ? Les notes de même moment d'attaque dans les deux partitions sont-elles liées entre elles où uniquement les notes de même moment d'attaque **et** de même durée ?

Même si nous tenions tout de même à mettre ceci en avant, ces questions ne trouveront pas de réponse ici. Ce n'est en effet pas le but premier de ce travail. Cependant, nous allons tout de même donner un moyen général que nous utiliserions pour réaliser cette tâche. Ceci n'est qu'une idée et n'assure en rien de bons résultats.

Trouver les notes qui sont liées entre elles pourrait se faire à l'aide d'une **fonction de similarité**. Celle-ci serait appliquée sur les paramètres utilisés pour représenter ces différentes notes dans leur relation respective. Cette fonction donnerait alors une mesure permettant de décider si oui ou non deux notes sont liées. Autrement dit, si deux tuples sont suffisamment liés pour être concaténés et placés dans la relation T .

L'intuition de ceci est qu'une note n'est « transformée » en une autre que si elle est suffisamment similaire à celle-ci. Par exemple, si la hauteur et le moment d'attaque d'une note a sont fort différents de ceux d'une autre note b , les paramètres de ces notes ne sont peut-être pas les meilleurs à utiliser pour encapsuler l'information de transformation dans la relation T .

Deux meta (ou hyper)-paramètres sont donc évoqués ici pour l'apprentissage :

- la fonction de similarité f
- le seuil de similarité θ , au-dessus duquel on estime qu'une note n'est plus similaire à une autre

De là, la manière la plus générale de procéder pour construire la relation T est de permettre **toutes** les transformations possibles à partir de ce qu'on vient d'apprendre. C'est-à-dire, si pour des tuples $t_a \in bar_1$ et $t_b \in bar_2$, on a :

$$f(t_a, t_b) \leq \theta$$

alors t_a et t_b pourront être utilisés pour la transformation. Ils pourront donc être concaténés et inclus dans T .

Puisque nous voulons uniquement rendre ces tuples (concaténés) **possibles** pour la relation, nous les plaçons dans $lub(T)$. Ainsi, c'est la recherche qui déterminera si oui ou non ces « tuples de transformations » seront effectivement utilisés pour la transformation. Dans ce cas, $glb(T) = \emptyset$.

Pour finir, on pourrait ajouter un troisième meta-paramètre afin d'être un peu plus spécifique. Il s'agirait d'un autre seuil de similarité, en-dessous duquel on estime que les tuples sont suffisamment similaires pour être **obligatoirement** utilisés dans la transformation. Leur concaténation sera donc placée dans $glb(T)$, ce qui assurera leur utilisation. Nommons ce paramètre γ .

Pour résumer, exprimons tout ceci mathématiquement :

$$\forall t_a \in bar_1, \forall t_b \in bar_2 : f(t_a, t_b) \leq \theta \implies concat(t_a, t_b) \in lub(T)$$

$$\forall t_a \in bar_1, \forall t_b \in bar_2 : f(t_a, t_b) \leq \gamma \implies concat(t_a, t_b) \in glb(T)$$

Rappelons que tout tuple présent dans $lub(T)$ doit être présent dans $glb(T)$.

Bien que nous parlons ici de transformation d'objets musicaux, nous pourrions utiliser tout ce qui a été dit précédemment pour **lier** les objets musicaux, sans que ceci ait la moindre connotation de transformation. Dans ce cas, on pourrait procéder tout autrement.

4.1.3 Relations utilisées de manière générale

Dans cette courte sous-section, nous rappelons qu'il est également possible d'utiliser les variables relationnelles en se basant sur la définition générale de relation : un ensemble non-ordonné de tuples (d'ensembles ordonnés, dont la cardinalité est fixée par l'arité de la relation). Le rôle qu'on peut donner à une relation ne s'arrête pas à ce qu'on décrit dans ce document, beaucoup d'autres significations peuvent sans doute leur être données. Les relations peuvent donc être utilisées pour ce qu'elles sont, un ensemble non-ordonné d'ensemble ordonnés de valeurs liées entre elles.

Pour donner un exemple, une relation ternaire peut représenter un ensemble non-ordonné d'accords majeurs non-renversés : le premier élément de chaque tuple est la tonique, le second la tierce et le troisième la quinte.

4.1.4 Utilisation des relations ground

Au-delà de l'utilisation de variables de décision, il est également possible d'utiliser les relations ground de manière intéressante. L'information contenue dans le domaine d'une relation ground peut être utilisée directement puisqu'elle est connue avant la recherche. On peut alors se servir d'information pré-calculée pour imposer des contraintes.

Un exemple simple est la relation ground que l'on pourrait nommer *plus*, définie comme suit :

$$\langle t_1, t_2, t_3 \rangle \in plus \iff t_1 + t_2 = t_3$$

On possède dès lors toute l'information à l'avance sur l'ensemble de toutes les sommes de deux éléments. Si on veut représenter trois entiers dont le troisième est la somme des deux

premiers, au lieu d'imposer une contrainte sur des variables entières comme on le ferait normalement, on crée une relation d'arité 3 et on impose qu'elle soit incluse dans *plus*. L'élagage sera directement appliqué et ne sera plus nécessaire durant la recherche comme c'est le cas pour les variables entières. En effet, il suffit de retirer des tuples inconnus de la relation ceux qui ne font pas partie de *plus*. Autrement dit, la contrainte elle-même est pré-calculée.

Ce type d'utilisation peut évidemment prendre plus de mémoire mais peut également permettre de réduire le temps de recherche.

Il est possible d'appliquer cette même idée dans le domaine musical. Par exemple, en utilisant une relation *ground* qui contient l'ensemble de tous les accords majeurs non renversés (en valeur MIDI). Appelons-la *major*. Il s'agit donc d'une relation *ground* d'arité 3, définie comme suit :

$$\langle t_1, t_2, t_3 \rangle \in major \iff t_2 - t_1 = 4 \wedge t_3 - t_1 = 7$$

On possède alors cette information de manière pré-calculée. Si une variable relationnelle devait représenter un accord majeur non renversé ou un ensemble d'accords majeurs non renversés, il suffirait d'imposer qu'elle est incluse dans la relation *major*. Exprimé mathématiquement :

$$lub(D_X) \subseteq major$$

où D_X est le domaine relationnel de la variable relationnelle.

Avant même de commencer la recherche, on est déjà certain que la propriété est satisfaite. Le domaine est alors directement élagué selon cette contrainte. En utilisant des tableaux d'entiers, on aurait dû faire une recherche afin de s'assurer que les valeurs dans les domaines satisfont bien les contraintes définissant un accord majeur non renversé.

Il faut remarquer que cette information doit bien évidemment être connue à l'avance, ce qui n'est pas immédiat pour toutes les relations (sinon les variables relationnelles ne seraient même plus nécessaires, ce qui n'est évidemment pas le cas).

Par contre, pour trouver l'ensemble total des tuples vérifiant une propriété donnée, on peut résoudre un sous-problème utilisant des variables entières, à l'avance, et donc à ne calculer qu'une seule fois. Pour chacune des composantes de la relation, on crée une variable entière. On pose des contraintes sur ces variables afin que la solution du sous-problème retourne un ensemble de valeurs ordonnées (un tuple). On cherche alors toutes les solutions à ce problème. Une fois trouvées, on possède alors un ensemble fixe de tuple, c'est-à-dire une relation *ground*. On peut alors utiliser celle-ci comme expliqué précédemment. Il est clair qu'on ne peut faire ceci pour toutes les propriétés. Trouver toutes les solutions pour n'importe quel problème utilisant des variables entières n'est pas du tout immédiat.

Remarquons pour finir que l'avantage de pouvoir élaguer directement le domaine de la relation amène un désavantage : les relations *ground* prennent de la place mémoire. On est donc

face à un compromis temps-mémoire. La représentation du domaine de la relation doit donc être efficace afin de comprimer suffisamment l'information pour que l'avantage apporté soit utile.

4.1.5 Nouvelles classes de problèmes abordées et contraintes associées

Dans cette sous-section, nous citons rapidement les différents types de problèmes pouvant à présent être abordés, en utilisant les relations comme nous l'avons décrit précédemment. Ce qui est présenté est juste un ensemble d'idées qui n'est certainement pas exhaustif.

Premièrement, en utilisant des relations comme transformations d'objets musicaux, il est possible de modéliser des problèmes de **thème et variations**. Dans ce type de problème, une mélodie est connue (le thème) et on recherche des variations de celle-ci ou de son accompagnement. On peut alors s'attaquer à différents types de problèmes :

- appliquer des transformations connues (ground ou non) à d'autres mélodies afin d'obtenir de nouvelles variations.
- transformer un accompagnement afin d'en obtenir un nouveau, en fonction ou non du thème et/ou de variations.
- combiner des relations obtenues précédemment, avec certains opérateurs, afin d'obtenir de nouvelles transformations, et donc de nouvelles variations.

Rappelons que toutes ces tâches peuvent également être réalisées au moment de la recherche, afin de servir un problème plus global.

Deuxièmement, en utilisant des relations pour représenter une partition, il est possible d'imposer des contraintes globales sur une partition complète. Pour cela, on peut utiliser des contraintes réalisant des opérations simples de l'algèbre relationnelle. Par exemple, l'*inclusion* permet d'imposer des **leitmotivs** (pouvant être eux-mêmes transformés par une relation). L'*union* permet de combiner différentes partitions, en les ayant transformées par d'autres relations ou non. Un exemple simple en est le canon : en transformant la première voix dans le temps et en faisant l'union de la première voix et de la deuxième transformée, on obtient un canon à deux voix.

En utilisant l'*intersection*, il est possible de récupérer et d'utiliser l'information commune de plusieurs partitions. On peut alors utiliser cette information au moment de la recherche pour différentes choses : **extraire des patterns** et les appliquer à d'autres partitions, créer des **medleys**... Il peut également être possible, via d'autres contraintes, d'extraire la signature ou le tempo d'une partition. De manière limitée, on possède donc une sorte d'**apprentissage** au moment de la recherche, même si cet apprentissage est plutôt un ensemble de requêtes au sens de l'algèbre relationnelle.

Ensuite, à l'aide de liens entre différents objets musicaux, on peut créer des voix supplémentaires par-dessus une mélodie en fonction d'elle et/ou de règles harmoniques. On pourrait également harmoniser une mélodie, en liant les accords sous-jacents à celle-ci par des relations.

Certaines théories musicales décrivent la composition comme un processus générateur d'une partition à partir de notes de base [31]. En généralisant, on peut voir la composition comme un processus générateur à partir de certains objets musicaux. En utilisant les relations comme des transformations, on peut appliquer ce principe. Par exemple, un problème facilement exprimable peut consister à n'avoir qu'une ligne de basse et à générer le reste de la partition (selon des critères harmoniques, rythmiques, mélodiques, cadentiels, ...). Un exemple est donné à la figure 4.4. Dans celui-ci, chacune des notes d'un accord serait générée à partir de la note de la basse ayant le même moment d'attaque. Remarquons que, puisque la transformation est modélisée par une relation, si la ligne de basse n'est pas fixée a priori, elle est tout autant générée par la mélodie.



FIGURE 4.4 – Mélodie générée à partir d'une ligne de basse.

De plus, nous pouvons remarquer qu'il est possible d'appliquer ces transformations plusieurs fois consécutives, ce qui rappelle un peu l'idée de musique fractale (cas où une partie de l'objet musical génère l'objet musical). [32] [31]

Les idées décrites précédemment peuvent évidemment être combinées entre elles, ou encore utilisées en conjonction avec l'utilisation des autres types de variables.

4.1.6 Remarque sur le gain apporté par les relations

Dans cette sous-section, nous nous concentrons un peu plus sur le gain que procurent les relations dans la programmation par contraintes dans le domaine musical.

Tout d'abord, elles permettent de modéliser de manière claire et simple un problème avec une approche **globale**. En effet, il est vrai qu'on peut parfois réaliser ce qui est fait à la sous-section 4.1.1 à l'aide d'entiers ou d'ensembles d'entiers, mais pas aussi aisément.

Prenons l'exemple d'un objet musical dont les paramètres sont la hauteur de note et le moment d'attaque : en utilisant uniquement des entiers, il faudrait un tableau d'entiers par temps, et une valeur spécifique pour le silence. Il faudrait également que les valeurs présentes dans le tableau soient distinctes, sauf pour la valeur du silence. Si on possède les relations, une seule variable relationnelle suffit.

De plus, alors que pour les entiers, le nombre maximum⁸ d'objets musicaux est fixé par le

8. Maximum car on peut utiliser une valeur négative pour le silence. Le nombre de notes peut donc être de zéro si toutes les variables sont assignées à cette valeur. Le nombre de notes ne pourra par contre pas dépasser le nombre de variables entières.

nombre de variables lors de la modélisation, ce n'est pas le cas pour les relations. En effet, c'est alors le **domaine** qui fixe cette borne. Dans le cas des ensembles d'entiers, si on représente un accord (ici vu comme un ensemble de notes simultanées) par un ensemble, le nombre de variables étant fixé, le nombre maximal d'accords l'est également.

Ensuite, nous devons parler du **gain d'expressivité** apporté par les relations. Tout d'abord, par rapport aux systèmes précédents, l'expressivité **théorique** est identique. En effet, en théorie, il est toujours possible de représenter un ensemble de tuples par un ensemble d'entiers, tel qu'un tuple est associé à un et un seul entier. Il suffit pour cela d'appliquer une fonction bijective⁹ de \mathbb{N}^n dans \mathbb{N} , où n est l'arité de la relation en question. En d'autres mots, on code les tuples par des entiers, ce qui permet en théorie d'utiliser les relations via des ensembles d'entiers.

Cependant, en pratique, utiliser des ensembles d'entiers est dans certains cas nettement moins efficace que des relations. En effet, en fonction de la représentation interne des domaines des relations, l'information contenue dans des relations ground peut être comprimée. Elles permettent donc **en pratique** l'utilisation de relations dont le domaine est assez large. Par contre, avec un ensemble d'entiers, chacun de ces entiers doit effectivement être présent dans le domaine de l'ensemble d'entiers si le tuple correspondant fait partie de la relation. Ceci peut donc prendre plus de mémoire, la représentation interne ne pouvant pas forcément utiliser le même type de compression.

On peut ajouter que l'encodage et le décodage des tuples peuvent sans aucun doute réduire les performances.

En résumé, au niveau de l'expressivité théorique, les relations n'apportent rien, mais en pratique, elles apportent deux avantages fondamentaux :

- Une expressivité allégée, plus claire et plus globale.
- Une expressivité de problèmes de taille plus grande car les domaines des relations peuvent avoir une taille plus grande, en fonction de la manière dont ils sont représentés de façon interne.

Bien que le second avantage permette de réduire cet effet, le premier s'accompagne inévitablement d'un désavantage bien connu : le **fléau de la dimension**. En effet, au plus l'arité d'une relation augmente, au plus son domaine augmente, et ce, de manière exponentielle. En pratique, la recherche devra donc utiliser des heuristiques de recherche des plus intelligentes.

Finalement, au niveau du gain par rapport à la propagation, c'est assez difficile à exprimer sans faire d'expérimentations pratiques. Ce gain dépend du problème posé, de la manière dont on modélise celui-ci, des contraintes utilisées, ...

9. Si on utilise des variables représentant des entiers, ceci peut être réalisé à l'aide de la contrainte **table**.

4.2 Cas pratiques

Dans cette section, nous donnons deux cas pratiques rendus possibles par GeLisp, muni des relations. Ensuite, nous donnons une courte liste d'utilisation simple des relations. Celles-ci n'ont pas été implémentées, mais permettent de se rendre compte facilement de la manière de les utiliser.

Mais avant tout, nous devons décrire comment nous représentons la musique à l'aide de GeLisp afin de pouvoir l'utiliser dans OpenMusic.

Pour ce qui est des entiers et des ensembles d'entiers, GeLisp permet en toute généralité de les utiliser comme dans les systèmes de contraintes précédents. Ainsi, un entier peut représenter une hauteur de note ou encore une durée. Un ensemble d'entiers peut par exemple représenter un accord ou une structure rythmique. Nous n'utiliserons pas ces types de variables dans les cas présentés. En effet, le **channelling** entre les variables relationnelles et les autres types de variable n'est pas fourni dans l'implémentation actuelle.

Bien que ceci soit laissé libre à l'utilisateur, nous représentons une partition à l'aide d'une relation ternaire. La première composante représente la hauteur de note, la seconde le moment d'attaque et la troisième la durée. Ce sont en effet trois paramètres principaux qu'on peut passer à un objet de type Chord-seq. On peut alors passer trois listes de valeurs, chacune de ces listes étant dédiée à un paramètre (voir figure 4.5).

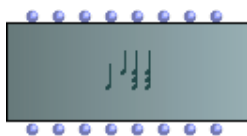


FIGURE 4.5 – Objet **Chord-seq** du langage OpenMusic. Le deuxième, le troisième et le quatrième paramètres sont respectivement utilisés pour la hauteur de note, le moment d'attaque et la durée des notes.

La hauteur de note peut être utilisée aisément, ainsi que la durée. Les valeurs se suffisent à elles-mêmes. Par contre, afin de pouvoir représenter le moment d'attaque, ou plutôt tous les moments d'attaque, la valeur ne suffit pas. Il faut en effet donner une valeur proportionnelle à la plus petite subdivision du temps que l'on permet pour la représentation donnée. Ceci est dû au fait qu'on travaille dans le domaine des entiers.

Par exemple, si on divise une mesure en 4 parties (et en supposant que la signature est de 4/4), on ne pourra pas représenter des moments d'attaque situés entre deux temps. L'utilisation de croches est donc dans ce cas proscrite. En fonction de la « précision » qu'on veut permettre dans la représentation, la division du temps change. Le domaine de la composante du moment d'attaque doit donc être adapté en fonction. Remarquons que ce domaine augmente de manière exponentielle à chaque division du temps.

4.2.1 Recherche de parties communes de deux partitions

En tant que premier cas pratique, nous montrons comment il est possible de trouver des parties communes de deux partitions. Ce problème n'a pu être résolu en pratique, les propagateurs \bowtie et π n'étant actuellement pas encore disponibles. Cependant, il permet d'illustrer la manière dont on peut utiliser des variables relationnelles pour modéliser et résoudre un problème concret.

Pour ce problème, nous représentons une partition par une relation (ici *ground*). Appelons les deux relations utilisées $score_1$ et $score_2$. Trouver les parties communes des partitions revient à trouver quels sont les tuples communs de ces deux relations. Nous utilisons donc le propagateur d'**intersection** pour imposer qu'une troisième relation (non-*ground*), nommée $score_3$, est l'intersection des deux autres.

$$score_3 = score_1 \cap score_2$$

Cependant, on peut facilement s'imaginer que cette intersection est généralement vide (ou au mieux contient quelques notes), les morceaux de musique étant facilement différents, surtout si l'arité des relations utilisées est grande (autrement dit, si la représentation de la partition est plus précise). En effet, une pièce musicale n'est généralement pas représentée par une relation très dense¹⁰. Ceci est une conséquence du fléau de la dimension mentionné dans la sous-section 4.1.6. Au plus on utilise d'informations pour représenter la musique, au plus l'information sera disparate dans l'espace de représentation.

Par exemple, rien qu'en faisant l'intersection d'une partition avec elle-même, mais décalée d'un temps, on a peu de chance de contenir beaucoup d'information (à moins qu'il s'agisse d'un canon).

Cette contrainte est donc trop forte. On peut alors la relâcher, en permettant un décalage pour chacune des composantes. La contrainte devient alors :

$$score_3 = score_{1_{offset}} \cap score_2$$

où $score_{1_{offset}}$ est la relation $score_1$ dont les composantes de tous les tuples ont été décalées d'une certaine valeur, propre à chaque composante.

Ce décalage peut être exprimé à l'aide d'une relation de cardinalité une, qu'on peut encore voir comme un tuple. Soit Θ ce décalage :

$$\Theta = \langle offset_0, \dots, offset_{n-1} \rangle$$

où n est l'arité des différentes relations.

10. Voir le dernier paragraphe de 4.1.1.

Afin de ne pas tomber dans le même problème de contrainte trop forte mais décalé, on permet à Θ de varier.

Afin de pouvoir exprimer la contrainte avec les propagateurs que nous possédons, nous devons encore définir une relation *ground*, basée sur la relation *plus* déjà définie dans la sous-section 4.1.4. Il s'agit de la relation $plus^{3.n}$, qu'on peut définir comme suit :

$$t \in plus^{3.n} \iff \forall i \in [0, \dots, n-1] : t(i) + t(i+n) = t(i+2n)$$

Cette relation peut être créée à partir de la relation *plus* comme suit :

$$plus^{3.n} = move_{i \rightarrow (i \% n) * 3 + i / n} \underbrace{(plus \times \dots \times plus)}_n$$

où $move_{i \rightarrow j}$ est un ré-agencement des composantes de la relation où la composante i devient la composante j .

Dès lors, on peut définir $score_{1_{offset}}$ comme ceci :

$$score_{1_{offset}} = \pi_{2.n, \dots, 3.n-1} (score_1 \times \Theta \bowtie plus^{3.n})$$

où, rappelons-le, Θ est variable, ce qui fait que $score_{1_{offset}}$ est également variable (π est ici l'opérateur de projection de l'algèbre relationnelle). Les 3 variables de ce problème sont donc $score_3$, $score_{1_{offset}}$ et Θ .

On peut se demander quel est l'avantage de pouvoir faire ceci par rapport à avant. En effet, trouver l'intersection de listes de tuples peut être réalisé sans trop de difficulté (hormis le temps) à l'aide de boucles. Cependant, au-delà de ça, ceci permet d'utiliser cette information au moment de la recherche. On peut alors utiliser cette information générale pour créer des heuristiques de recherche. La composition assistée par la programmation par contraintes prend donc une dimension plus globale qu'avant.

Finalement, à partir de ce problème de satisfaction, on pourrait également imaginer un problème d'optimisation, pour lequel on voudrait trouver :

$$\Theta_{max} = \underset{\Theta}{\operatorname{argmax}} (\#score_3)$$

4.2.2 Répartition d'une pièce musicale entre différents instruments

Un deuxième cas pratique, assez trivial s'il est utilisé seul, mais qui peut s'avérer utile s'il est utilisé en conjonction avec d'autres exigences (contraintes), est la répartition d'une pièce entre différents instruments. Ce problème est assez simple à exprimer lorsqu'on représente la pièce par une relation. De plus, il nécessite uniquement deux propagateurs très simple : l'**union** et la

disjonction **disjoint**.

Supposons qu'on veuille répartir une pièce musicale, représentée par une relation ternaire :

$$pieceScore = \{\langle pitch, onset, duration \rangle\}$$

entre deux instruments i_1 et i_2 .

Les partitions de ces instruments peuvent être également représentées par deux relations ternaires $score_{i_1}$ et $score_{i_2}$, dont les composantes ont la même sémantique que celles de $pieceScore$.

Imposer que $score_{i_1}$ et $score_{i_2}$ forment ensemble $pieceScore$ revient à dire qu'elles en sont ensemble une **partition** (au sens de la théorie ensembliste). Ceci peut s'exprimer simplement par :

$$pieceScore = score_{i_1} \cup score_{i_2}$$

et

$$score_{i_1} \cap score_{i_2} = \emptyset$$

Les propagateurs d'union et de disjonction suffisent donc.

Remarquons que, bien que le problème puisse être résolu assez facilement dans un autre paradigme de programmation, cette résolution possède l'avantage de ne devoir exprimer le problème que de manière déclarative (voir la section 3.2).

La figure 4.6 donne la résolution du problème dans OpenMusic. Pour résumer, la partie gauche du programme crée la partition (en l'occurrence, deux mesures des violons 1 et 2 du canon de Pachelbel). La partie droite du programme récupère cette partition sous la forme d'une relation. Elle impose ensuite les contraintes données ci-avant sur deux variables relationnelles. La recherche trouve alors une solution pour ces deux variables.

L'implémentation¹¹ de ceci est réalisée en Common Lisp dans la *lambda function* nommée *createarrangementoftwovoice*. Dans notre cas, beaucoup de solutions existent car le domaine relationnel des deux variables relationnelles leur permet de contenir toutes les notes. Une des solutions est donnée dans la figure 4.7.

Nous pouvons donner une extension un peu plus concrète et précise à ce problème : supposons qu'on ait une partition chantée par une chorale et que l'ambitus de la partition est très large. On veut adapter la pièce pour des instruments à cordes, en leur imposant une tessiture restreinte à chacun. Utiliser l'exemple précédent est alors utile : on crée une partition pour chaque instrument, telle que chaque instrument ne joue que dans la tessiture qui lui est accordée ; de

11. donnée en annexe C

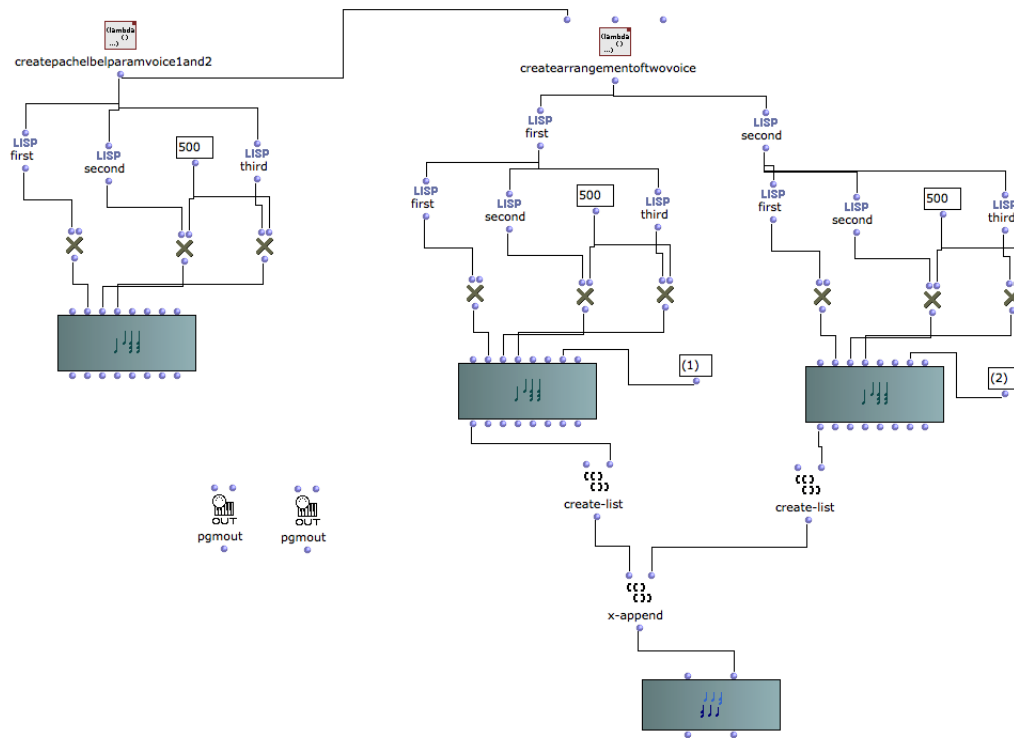


FIGURE 4.6 – Résolution du problème de répartition avec GeLisp.



FIGURE 4.7 – Solution du problème de répartition. La partition initiale contenait toutes les notes des deux voix.

plus, on impose que toutes les partitions (musicales) forment une partition (mathématique) de la partition jouée.

La seconde contrainte est celle du problème précédent. La première peut être imposée au moment de la déclaration du domaine ou à l'aide de l'intersection avec une relation ground qui ne contient que des tuples dont la composante *pitch* est dans la tessiture de l'instrument.

4.2.3 Exemples d'applications simples

Nous donnons ici des applications simples et directes utilisant des relations et expliquons comment les réaliser de manière informelle.

Si une relation représente une partition :

- augmenter la composante *pitch* de tous les tuples permet de transposer.
- multiplier la composante *onset* de tous les tuples par une constante k permet de ralentir ou accélérer le morceau.
- si $0 \dots onset_t$ est une suite de moments d'attaque consécutifs, en leur appliquant une fonction quadratique, on pourra créer un **accelerando** ou un **ritardando**. (il ne faut cependant pas oublier qu'on ne travaille ici qu'avec des entiers.)
- au niveau des moments d'attaque, on peut également décaler certaines mesures pour que la première note soit systématiquement située sur un temps faible.
- ...

Si une relation représente une transformation (liaison) :

- En appliquant une transformation binaire au niveau des hauteur de note, on peut réaliser des **changements de mode**. Par exemple, transformer une mélodie du mode majeur en une autre mélodie dans le mode mineur.
- En appliquant des transformations au niveau des moments d'attaque, on peut créer une mélodie rétrograde. Plus généralement, on peut modifier l'ordre d'apparition des notes.
- On peut lier le moment d'attaque et la hauteur de note d'une partition à la nuance d'une autre partition. Cette liaison pourrait par exemple servir dans le cas suivant : deux instruments jouent ensemble (une partition par instrument donc) et on veut imposer que si un instrument joue une note constitutive sur un temps fort, l'autre instrument doit jouer modérément.

Une autre application pourrait être de l'empêcher de jouer une note avec une altération accidentelle. On voit dans cet exemple simple en quoi les relations permettent des liaisons entre les éléments musicaux. L'avantage est que ces liaisons peuvent être utilisées pour lier n'importe quels types d'élément, à condition qu'il puisse être représenté par un entier (ou un tuple).

Ces applications simples ne sont que des pistes. Elles permettent déjà de voir ce que les relations ont à apporter en pratique. Cette liste semble loin d'être exhaustive.

4.3 Le problème du contexte de partition inaccessible revisité

Dans cette dernière section, nous reparlons d'un problème mis en avant par Torsten Anders dans sa thèse : le **problème du contexte de partition inaccessible**. [23]

Avant tout, nous donnons la définition générale de ce qu'il entend par contexte de partition :

Contexte de partition. Un contexte de partition est un ensemble arbitraire d'objets de partition. [23]

Ce qu'il entend par « objet de partition » est un cas particulier de ce que nous avons défini comme étant un objet musical (nous ne nous arrêtons pas à un objet musical pouvant être défini au sein d'une partition. Voir la sous-section 4.1.1). Nous pouvons ici faire l'amalgame, le problème énoncé étant applicable aux deux définitions.

Le problème peut être défini comme suit :

Problème du contexte de partition inaccessible. La représentation utilisée de la musique ne contient pas assez d'informations pour permettre l'accès (direct) à un contexte de partition donné.

Dans certains cas, il n'est pas possible de contraindre certains contextes de partition. Ceci est dû au fait qu'il existe un manque d'information dans la définition du problème, rendant ces contextes inaccessibles. Leur accès n'est fourni qu'au moment de la recherche. Plus précisément, un propagateur **non-réifié** ne peut pas être appliqué au début de la recherche, car l'ensemble des variables à contraindre par ce propagateur n'est pas encore connu.

Un exemple donné par Torsten Anders est le fait de permettre uniquement certains intervalles entre des notes simultanées. Il est clair que si on ne connaît pas au moment de la modélisation du problème la structure rythmique de la partition, on ne peut savoir quelles sont les notes simultanées et donc quelles sont les notes auxquelles on doit appliquer la règle.

Plusieurs techniques proposées permettent de pallier à ce problème :

- Premièrement, la solution naïve est de générer une solution du problème modélisé et de vérifier a posteriori si la contrainte est respectée. Ceci revient à faire de la recherche sans aucune propagation pour la contrainte donnée. La contrainte n'est vérifiée qu'au moment où toutes les variables sont assignées. Ceci est évidemment des plus inefficaces.
- Deuxièmement, il parle d'application retardée de règle. La règle est alors appliquée lorsqu'assez d'informations est disponible. Dans l'exemple donné, si à un moment de la recherche, certaines notes deviennent simultanées, on impose une nouvelle contrainte sur ces notes. Le problème est que le MCSP change au cours de la recherche, ce qui rend difficile sa compréhension.
- Ensuite, on peut également reformuler la modélisation afin de retirer le problème d'inaccessibilité. C'est souvent une bonne manière de procéder, mais ce n'est pas toujours possible.
- Finalement, on peut contourner le problème en utilisant des connecteurs logiques. On peut alors utiliser des contraintes réifiées mais l'élagage des domaines peut être très pauvre dans ce cas.

Ces solutions présentées peuvent parfois être inefficaces. Les relations permettent d'aborder le problème de contexte de partition inaccessible d'une manière différente. Ce qui suit explique en quoi.

Avant tout, rappelons que chaque objet musical peut être représenté par une relation. Il suffit qu'à chaque composante de la partition soit assignée un paramètre ou une information à propos de l'objet musical. Chaque tuple représente alors un élément de l'objet musical. Par exemple, si on représente une mesure par une relation, chacune des notes est représentée par un tuple (voir la sous-section 4.1.1).

Dès lors, on peut dire que, sans aucune contrainte sur la relation, celle-ci peut en soi représenter n'importe quel objet musical, la limitation de ce qu'elle modélise étant fonction des paramètres choisis comme étant représentés par une composante. En tenant compte de cette limitation, il est possible en théorie d'« accéder » à tous les objets musicaux représentables grâce à un sous-ensemble de l'univers (voir section 2.2.1).

En effet, ces éléments font partie du **domaine** de la variable relationnelle. Avant cela, les paramètres de l'élément étaient représentés par des variables séparées. Dans notre cas, si on veut avoir accès à tous les éléments (tous les tuples)¹² partageant une même propriété, il suffit de récupérer ces tuples (ou sous-tuples) au sein d'une variable relationnelle auxiliaire R_{aux} . Pour ceci, on peut utiliser les opérations π (projection) et σ ¹³ (sélection) de l'algèbre relationnelle. Une fois les tuples voulus « récupérés » au sein de R_{aux} , on peut appliquer la contrainte qu'on désire sur cette variable.

Deux choses sont importantes à comprendre ici :

- Les tuples récupérés sont dans le domaine de R_{aux} , car ils sont liés aux tuples présents dans le domaine de la variable d'origine par les propagateurs π et \bowtie . Dès lors, appliquer une contrainte sur les tuples présents dans R_{aux} revient à appliquer cette contrainte sur les tuples de la variable d'origine qui partagent une même propriété (celle qu'on a utilisée pour les récupérer). La propagation dans le domaine de R_{aux} sera donc également appliquée au domaine de la variable d'origine.
- Même si on ignore encore au début de la recherche si les tuples de R_{aux} seront présents ou non dans la solution, on peut déjà appliquer la contrainte désirée car on est certain qu'ils partagent une même propriété.

Ce cas idéal (au niveau du problème de l'inaccessibilité) possède cependant une condition. Il faut être capable d'exprimer la récupération des tuples partageant la propriété désirée à l'aide de propagateurs. Il n'est pas sûr que ce soit toujours le cas. De plus, on est pas sûr que cette manière de procéder soit plus efficace en pratique. Peut-être que la réification de propagateurs sur des anciennes variables est plus efficace pour certains problèmes. Ceci est à vérifier en pra-

12. ou tous les sous-éléments, c'est-à-dire les sous-tuples, si on ne veut récupérer que certains des paramètres et pas tous.

13. réalisée à l'aide du propagateur \bowtie

tique. On peut penser que dans certains problèmes on sera plus efficace, et pour d'autres non.

Par contre, dans certains cas, on peut au moins être sûr d'avoir un modèle plus clair et plus court, ce qui est déjà un avantage. De plus, il peut être plus modulaire, dans le sens qu'on peut le modifier plus aisément pour aborder un autre problème.

Tentons de modéliser un problème repris de Torsten Anders à l'aide d'entiers et ensuite à l'aide de relations. Nous verrons qu'avec les entiers, le problème doit être résolu avec l'une des techniques présentées précédemment, et avec les relations, nous pouvons le résoudre comme nous venons de le décrire. Le problème est défini de manière très courte : on cherche un ensemble de notes d'une partition telles que les notes simultanées sont consonantes.

Dans le cas des entiers, on utilise un tableau d'entiers pour les hauteurs des notes, nommé *pitch*, et un tableau d'entiers pour les moments d'attaque des notes, nommé *onset*. *pitch[i]* est donc la hauteur de la note *i* et *onset[i]* est son moment d'attaque. On suppose également que les notes sont toutes de même durée et qu'elles s'arrêtent de sonner juste avant le prochain moment d'attaque. On utilise également la valeur spéciale -1 pour le pitch afin qu'une note puisse représenter un silence.

On cherche à poser la contrainte disant que toutes les notes ayant lieu au même moment sont consonantes. Autrement dit, on a que

$$\forall i, j : onset[i] = onset[j] \implies consonant[i, j] \text{ où } i \neq j$$

Mais, afin de pouvoir poser la contrainte *consonant[i, j]*, on doit connaître ces *i, j* au moment de la modélisation, et ce n'est évidemment pas le cas, puisqu'on cherche à les déterminer. Le contexte est donc bien inaccessible. Rappelons qu'on peut gérer ce problème à l'aide des techniques présentées ci-avant. Imaginons qu'on le fasse avec la réification, qui est sans doute la technique la plus simple.

Il faudra donc ajouter les contraintes :

$$equal(onset[i], onset[j], b_{i,j})$$

qui expriment que le booléen $b_{i,j} = 1$ si $onset[i] = onset[j]$, 0 sinon. On possède donc déjà dans le modèle une variable et une contrainte supplémentaires par paire de moments d'attaque.

En plus de cela on impose les contraintes :

$$areConsonant(pitch[i], pitch[j], b_{i,j})$$

qui imposent que si $b_{i,j} = 1$, alors $areConsonant(pitch[i], pitch[j])$.

$areConsonant(pitch[i], pitch[j])$ peut être dans ce cas exprimée comme :

$$areConsonant(pitch[i], pitch[j]) \iff (pitch[i] - pitch[j]) \bmod 12 \in setOfConsonantIntervals$$

où *setOfConsonantIntervals* est un ensemble fixe de valeurs.

On peut donc voir que cette contrainte simple n'est pas si simple à exprimer. De plus, modifier le problème nécessite de modifier fortement le modèle. Imaginons qu'on ait un troisième tableau d'entiers *duration* destiné à contenir les durées de chacune des notes. Et, supposons qu'on veuille imposer une contrainte sur toutes les notes de même durée. Il faut alors changer le modèle en ajoutant d'autres variables booléennes et d'autres contraintes réifiées, avant même de pouvoir imposer la contrainte. Ceci est un travail supplémentaire.

Maintenant, supposons qu'on représente la partition à l'aide de la relation binaire *score*, dont la première composante est la hauteur de note et le second est le moment d'attaque. Supposons également que la relation soit totale. Toutes les notes de cette partition sont accessibles puisqu'elles font partie du domaine. On peut alors poser la contrainte comme suit :

$$\forall i \in Onsets : areConsonant(\pi_{pitch}(score \bowtie \{\langle i \rangle\}))$$

où *Onsets* est l'ensemble de toutes les valeurs de moments d'attaque possibles.

Le contexte est devenu accessible directement car l'information à propos des notes n'est plus découplée comme elle l'est dans le cas d'entiers. Les notes qui sonnent au même moment sont connues, au sein du domaine, même si elles ne font pas partie de la solution finale.

Le propagateur *areConsonant(pitchSet)* peut ici être exprimé très simplement :

$$areConsonant(pitchSet) \iff tuple(pitchSet) \in setOfConsonantChords$$

où :

- *tuple(S)* est une fonction qui permet d'obtenir un tuple à partir d'un ensemble.
- *setOfConsonantChords* est une relation ground (donc pré-calculée) . Chacun de ces tuples contient un ensemble de hauteurs de notes qu'on estime consonantes.

Dès lors, si

$$tuple(pitchSet) \notin setOfConsonantChords$$

ou

$$arity(tuple(pitchSet)) \neq arity(setOfConsonantChords)$$

la contrainte ne tiendra pas. Dans le cas contraire, la contrainte est respectée. Remarquons qu'ici on pose une contrainte supplémentaire sur le nombre de notes consonantes. Mais on peut très facilement être plus général en utilisant une disjonction de conditions et plusieurs relations ground. Ceci n'est pas forcément plus efficace, mais la modélisation est en tout cas plus claire, plus courte et plus déclarative qu'avec des entiers.

On peut remarquer qu'on aurait pu résoudre ce problème à l'aide d'ensembles d'entiers. Mais rappelons que les ensembles d'entiers ne sont qu'un cas particulier des relations.

Prenons maintenant le cas d'une relation ternaire, la troisième composante étant la durée. Imaginons qu'on veuille imposer une contrainte sur le moment d'attaque et la hauteur de toutes les notes de même durée. Accéder à tous les tuples de même durée peut se faire aisément et de manière très similaire au modèle précédent :

$$\pi_{pitch,onset}(score \bowtie \{\langle i \rangle\})$$

où i est une valeur de durée possible.

On peut alors poser la contrainte désirée sur ces tuples, sans même savoir s'ils feront partie ou non du résultat final. Faire ceci à l'aide d'entiers et d'ensembles d'entiers n'est pas du tout immédiat. En plus des avantages cités précédemment, on peut donc voir que la modélisation est plus modulaire.

Rappelons également que dans certains cas, on peut directement écarter toutes les notes du domaine non désirées avant même d'avoir débuté la recherche (voir la sous-section 4.1.4). Par exemple, si on veut imposer la contrainte

$$duration[i] = onset[i] + pitch[i]$$

il suffit de retirer de la borne supérieure du domaine de la relation tous les tuples ne respectant pas cette condition.

Finalement, nous devons remarquer qu'on peut gérer le problème d'inaccessibilité dans une mesure limitée : l'arité de la relation pose une borne supérieure sur le nombre d'informations qu'on peut encapsuler au sein d'une même note, d'un même tuple. De plus, on a en contrepartie une taille de domaine plus grande. Une représentation efficace de celui-ci est donc nécessaire afin d'éviter une utilisation excessive de mémoire.

Troisième partie

Evaluation et perspectives

Evaluation

Dans ce court chapitre, nous résumons ce qu’apporte ce travail dans le domaine étudié. Nous critiquons tout d’abord GeLisp, pour ensuite récapituler notre avis sur l’utilisation des relations dans le domaine de la programmation par contraintes appliquée à la musique.

Evaluation de GeLisp

GeLisp est une interface qui permet d’utiliser la librairie Gecode en utilisant le langage Common Lisp. Elle permet donc l’utilisation du paradigme de programmation par contraintes dans ce langage. La librairie *Screamer* [12] est une librairie Common Lisp qui permet également la programmation par contraintes. Nous ne connaissons pas ce système et ne pouvons donc pas faire de véritable comparaison.

Cependant, nous pouvons donner une critique générale de GeLisp :

- GeLisp possède toutes les fonctionnalités de Gecode, dont la programmation par contraintes relationnelles. Screamer n’en est évidemment pas à ce point.
- GeLisp peut aisément être étendue et maintenue à jour.
- GeLisp est générique. On peut modéliser tous les problèmes que Gecode permet de modéliser.
- L’expressivité n’est jusqu’ici pas la plus simple. Cependant, elle peut facilement être améliorée en développant une couche Common Lisp par-dessus. Ceci permettra alors d’utiliser Gecode à partir d’une librairie Common Lisp dont l’utilisation pour l’utilisateur est simplifiée.

GeLisp dans le domaine musical

Avant de parler des relations, nous critiquons brièvement l’utilisation de GeLisp pour la programmation par contraintes dans OpenMusic. Nous critiquons donc GeLisp pour son utilisation dans le domaine musical.

Concrètement, GeLisp possède une propriété qui est sous un certain point de vue un avantage et sous un autre un inconvénient : la programmation se fait de manière **textuelle**. Il est donc moins facile pour les compositeurs de l’utiliser car ils ne sont pas a priori programmeurs. Cependant, elle permet une utilisation plus générique car elle est plus programmable [23].

Il est donc possible de résoudre plus de MCSP grâce à elle¹⁴. En plus de cela, bien que les langages visuels sont plus simples à utiliser a priori, il devient de plus en plus difficile de programmer lorsque les programmes prennent une taille plus grande.

Certaines expressions peuvent être exprimées simplement avec des langages textuels, alors qu'il est moins direct de le faire dans un langage graphique. Le filtrage par motif (« pattern matching ») en est un exemple [23].

Nous disions que GeLisp est générique car elle est textuelle et interface l'ensemble de Gecode. Mais un autre point négatif est présent par rapport à cela : elle ne fournit pas d'outils prédéfinis pour le domaine musical, comme le font par exemple Situation et OMRC¹⁵.

Deux problèmes sont donc présents : GeLisp utilise un langage textuel et ne fournit pas d'outils intégrés pour l'application musicale. Mais ces deux problèmes peuvent être résolus. Il est en effet tout à fait possible de construire un ou plusieurs niveaux supérieurs afin de fournir une librairie plus simple à utiliser par les compositeurs. Le langage de cette librairie peut alors être OpenMusic, langage visuel plus facilement utilisable par des compositeurs non-programmeurs. Cette librairie peut également définir un ensemble d'éléments utilisables directement, sans qu'on ait à les redéfinir pour chaque nouveau MCSP.

Finalement, nous pouvons rappeler que GeLisp, de part sa généralité, permet de définir un bon nombre des MCSP solubles par les autres systèmes. En plus de cela, elle permet l'utilisation de la programmation par contraintes relationnelles. La limite de potentiel de modélisation imposée à GeLisp est celle de Gecode.

Evaluation de l'utilisation de relations pour les MCSP

Nous divisons cette section en deux parties, l'une traitant de l'apport théorique et l'autre de l'apport pratique.

Apport théorique

Comme nous l'avons dit, les relations apportent une dimension globale à la programmation par contraintes. Dans le cadre de ce travail, les deux idées générales d'utilisation de variables relationnelles sont :

- la représentation d'un « objet musical »(le cas le plus général étant une pièce musicale complète) par une seule variable relationnelle.
- la représentation de liens entre des objets musicaux, par une seule variable relationnelle. Un cas particulier pour ceci est la transformation (bidirectionnelle) d'un objet musical vers un autre.

14. Voir le chapitre 3

15. Pour plus d'informations à ce sujet, le lecteur peut consulter l'annexe A.5

Cette vision globale ouvre des portes au niveau des contraintes utilisables et des problèmes à résoudre. Cependant, il n'est pas forcément immédiat de savoir comment les utiliser : par exemple, une réflexion préalable à propos des contraintes qu'on désire imposer sur une pièce complète est nécessaire. Quelle contrainte veut-on imposer à une pièce globalement ?

De plus, la musique n'est pas uniquement globale. La qualité esthétique d'une pièce dépend tout autant du détail.

Les propagateurs pour la jointure (\bowtie) et la projection (π) permettent d'atteindre en détail les parties désirées d'une relation, et donc de les contraindre. La précision peut aller jusqu'à la précision d'un tuple unaire, qu'on peut voir comme un entier. Cependant, les contraintes qu'on peut appliquer sur cette variable seront les contraintes relationnelles, qui restent des contraintes globales. On ne peut donc pas travailler de manière locale comme on le ferait avec les autres types de variable.

La solution à ce problème est le « channelling » entre les variables relationnelles et les anciens types de variables. On peut alors travailler de manière globale **et** de manière locale, détaillée. Pour le moment, on ne peut travailler qu'avec une seule des deux approches à la fois.

Nous pouvons également ajouter que pour savoir comment utiliser les variables et les contraintes relationnelles dans le domaine musicale, il faut d'autant plus posséder des capacités de composition et d'analyse dans ce domaine. Autrement dit, savoir à quels problèmes on peut s'attaquer grâce aux relations n'est pas immédiat si on ne possède pas un bagage dans cette spécialité. Cependant, nous estimons avoir proposé quelques pistes intéressantes, qui pourront sans doute donner suite à des résultats et/ou à d'autres idées.

Finalement, nous mettons en avant que de manière générale, nous pensons qu'une relation représentant une partition complète devrait être assez peu dense afin d'obtenir des résultats esthétiques satisfaisants. Ou en tout cas, la relation devrait conserver une structure assez « stable », afin que l'oreille puissent percevoir une structure dans la musique. Si la relation est trop « chaotique », l'oreille percevra la musique représentée par celle-ci comme « fausse ».

Trouver cette structure fait entre autre partie de l'art du compositeur.

Apport pratique

Nous l'avons vu, le système est pour le moment très minimal, ce mémoire ayant été réalisé durant l'élaboration du système sur lequel il se base. Il possède donc peu de contraintes. Un seul des deux cas pratiques présentés a pu donc être réalisé effectivement. Ce que nous avons pu découvrir de manière pratique est donc essentiellement l'ensemble des manques et nécessités afin de travailler dans le domaine musical.

Premièrement, les propagateurs \bowtie et π sont indispensables afin de pouvoir travailler de manière détaillée sur un objet global. En effet, pouvoir représenter un objet musical de manière globale est un avantage, mais pour pouvoir imposer des contraintes intéressantes, il faut pouvoir accéder à des parties de celui-ci. Sans ces propagateurs, nous ne pouvons que donner des contraintes sur l'ensemble de l'objet musical. Ceci est sans aucun doute intéressant, mais trop

restrictif.

Ensuite, les contraintes de **cardinalités** sur les relations seront utiles dans le futur. Par exemple, il est pour le moment impossible d'imposer qu'une relation ne soit pas vide. Si on représente une partition par une relation, ceci a évidemment tout son sens. De plus, la contrainte imposant une borne sur le nombre de note à un moment d'attaque donné ne peut pas non plus être exprimé (voir la sous-section 4.1.1).

Dû à ceci, le travail au niveau des tuples plutôt qu'au niveau d'une relation n'est pas possible non plus. Le cas pratique décrit dans la sous-section 4.2.1 ne peut donc pas non plus être exprimé pour le moment.

De plus, le **channelling** fait également défaut. Pouvoir lier les variables relationnelles avec les autres permettrait d'avoir accès aux fonctionnalités des deux approches pour un même problème. La modification du domaine de variables entières pourrait alors avoir un effet direct sur le domaine d'une relation, et inversement. Dès lors, tous les problèmes et techniques utilisées dans les deux approches peuvent être combinées entre elles.

Bien qu'on puisse penser que les relations fournissent une nouvelle dimension à l'utilisation de la programmation par contraintes appliquée à la musique, le système est encore aujourd'hui assez léger. Des conclusions au niveau pratique ne peuvent donc pas encore être plus détaillées que celles données ici. Cependant, ne fut-ce qu'avec les fonctionnalités manquantes décrites ici, il nous semble qu'on pourra déjà tirer parti du potentiel au niveau de l'application musicale.

Ajoutons finalement que les paramètres qu'on peut utiliser au niveau des partitions sont ceux d'OpenMusic. Cependant, les relations permettent potentiellement de travailler avec un nombre supérieur de composantes, et donc de paramètres.

Problèmes rencontrés

Travailler au niveau de l'état de l'art d'un domaine est bien sûr très intéressant, mais on rencontre malheureusement un certain nombre de problèmes d'ordre technique. Nous terminons ce chapitre en mentionnant les différents problèmes rencontrés durant ce mémoire.

Premièrement, bien qu'une interface entre Gecode et Common Lisp existait déjà précédemment, nous avons dû la recréer dans son ensemble. Gecode a en effet été mise à jour vers sa version 3, qui n'était plus compatible avec l'ancienne version¹⁶. La version sur laquelle le module des relations a été ajouté étant la version 3, nous devons obligatoirement travailler avec celle-là.

Ensuite, nous avons tenté d'intégrer l'implémentation précédente des relations, qui était un peu plus complète que celle utilisée actuellement. Cependant, CFFI et cette librairie étant des

16. Refactoriser le code aurait sans doute pris beaucoup plus de temps.

projets qui ne sont pas dans une version stable¹⁷, l'association des deux n'a pu être réalisée. Nous avons donc dû travailler avec la nouvelle implémentation des relations. Cependant, nous avons eu accès à celle-ci très tard dans l'élaboration de ce mémoire. De plus, elle est encore dans une version peu complète, fournissant peu de possibilités pratiques.

Bien que tout ceci ait pu retarder l'avancée du travail ainsi que la démonstration de cas pratiques, nous avons tout de même pu mettre en avant le potentiel de la programmation par contraintes relationnelles dans le domaine musical.

17. Version actuelle de CFFI : 0.10.6. La programmation par contraintes relationnelles est le fruit d'une thèse réalisée actuellement.

Perspectives

Dans le futur, une tâche à réaliser sera de résoudre de véritables cas concrets à l'aide de la programmation par contraintes relationnelles. En se basant sur le potentiel développé par ce travail, compositeurs et informaticiens devraient travailler collectivement pour découvrir quelles sont les réelles applications possibles.

Les relations sont prometteuses : pouvoir créer des liens entre tout ce qui existe de manière presque inconditionnelle semble ouvrir des portes vers un champ d'application beaucoup plus général qu'avant. Les structures qu'on pourra désormais construire et réutiliser, tant a priori que durant la recherche de solution d'un problème, seront bien plus complexes et globales que précédemment. Une autre manière de le voir est que, jusqu'ici on ne pouvait travailler que dans une seule dimension à la fois. Les valeurs entières étaient indépendantes les unes des autres parce que séparées dans des variables différentes.

De manière plus pratique, il faudrait également concevoir des outils concrets et pratiques pour les compositeurs. Ils pourront ainsi travailler directement, sans avoir à se soucier des notions techniques sous-jacentes. De plus, pour chaque nouveau problème abordé, ils auraient accès à des outils de base, sans avoir à les recréer systématiquement.

Ensuite, une autre opportunité qu'offre la programmation par contraintes relationnelles est le travail dans le domaine du son. Iannis Xenakis propose dans une certaine mesure cette idée lorsqu'il propose ses différentes représentations du son à l'aide de deux natures physiques, la fréquence et l'intensité [39].

On pourrait également travailler avec les paramètres de la transformée de Fourier, ou encore avec d'autres transformations.

Cependant, nous devons garder à l'esprit que dans la programmation par contraintes, on ne peut travailler qu'avec des entiers. On peut bien sûr supposer que les nombres sont multipliés par une puissance négative du nombre 10, mais on réduit alors les bornes maximales des composantes. Une application pour ceci est par exemple la synthèse de son.

Combiner la programmation par contraintes relationnelles avec de la recherche locale basée sur les contraintes pourrait également être intéressant. On pourrait obtenir des résultats plus satisfaisants esthétiquement. Par exemple, pour un problème donné, on pourrait chercher une solution avec la programmation par contraintes. Ensuite, on pourrait essayer d'améliorer cette solution en faisant de la recherche locale à partir d'elle.

Finalemment, développer l'idée proposée dans la courte partie 4.1.2 pourrait aboutir à des transformations d'objets musicaux et à d'autres applications intéressantes. En effet, on pourrait alors travailler à partir d'oeuvres musicales et apprendre d'elles. L'information acquise pourrait alors être utilisée dans différents domaines, dont ceux développés dans ce mémoire.

Conclusion

Dans ce travail, nous avons pu créer une interface vers la librairie Gecode pour le langage Common Lisp : GeLisp. Elle permet la programmation par contraintes de manière générique et est extensible. GeLisp s'intègre parfaitement dans l'environnement musical OpenMusic, et fournit donc un nouvel outil pour les compositeurs. De plus, elle fournit la programmation par contraintes relationnelles, qui n'est pour le moment présente nulle part ailleurs.

Ensuite, nous nous sommes penchés sur l'apport que fournissait la programmation par contraintes relationnelles, utilisée dans le champ musical. Les résultats sont essentiellement théoriques. Premièrement, il est possible de représenter à peu près tout ce qu'on veut à l'aide d'une seule relation : mélodie, suite d'accords, motif rythmique, partition complète n'en sont que des exemples. Il est également possible d'appliquer des transformations de partitions à l'aide de relations. La construction de ces transformations n'est cependant pas triviale, et relève sans doute plus de l'apprentissage par la machine (« machine learning »). Finalement, les relations permettent d'aborder le problème de contexte inaccessible de partition d'une nouvelle manière. Bien que théorique, cet apport forme une bonne base pour le futur.

De manière pratique, nous avons pu représenter des partitions existantes dans le format MIDI à l'aide de variables relationnelles. Les paramètres MIDI sont en effet directement utilisables avec ces variables, spécifiées comme nous l'avions fait théoriquement. Un problème concret a également pu être résolu. De plus, nous pouvons également résoudre d'autres problèmes utilisant les anciens types de variables. Un exemple est le problème de la série tous-intervalles. En pratique, nous permettons donc plus que la programmation par contraintes relationnelles.

Le travail présenté permet donc de travailler avec la librairie Gecode munie des variables relationnelles dans OpenMusic. Les fonctionnalités offertes sont multiples, et ne s'arrêtent pas aux propositions données. C'est maintenant aux compositeurs, avec l'aide des informaticiens, d'évaluer comment tirer parti de GeLisp.

Dans le futur, plusieurs ouvertures sont possibles. Premièrement, continuer à faire évoluer GeLisp en fonction de l'évolution de Gecode. Ceci permettra de continuer à pouvoir utiliser l'ensemble des fonctionnalités de Gecode à partir d'OpenMusic. La mise à jour la plus importante est celle pouvant être réalisée une fois le module de programmation par contraintes relationnelles terminé.

Celui-ci permet déjà de lier des ensembles d'entiers de grande taille avec d'autres, et ainsi de

créer des structures plus complexes et plus globales qu'avant. Par exemple, une partition peut être liée avec une autre, ou même avec plusieurs autres. Cette liaison dont nous parlons peut elle-même être liée avec d'autres liaisons du même type pour en créer une nouvelle plus générale. Alors que les structures se complexifient, l'expressivité augmente ...

Simultanément, les modèles gagnent parfois en concision et en simplicité, avec la possibilité de travailler sur des problèmes plus globaux et sans doute plus difficiles. Plus difficiles car, ces regroupement de liens, au-delà de pouvoir les représenter, nous pouvons les contraindre.

Par ailleurs, il serait intéressant de combiner ce que nous avons présenté avec d'autres domaines de l'intelligence artificielle, comme la recherche locale basée sur les contraintes et l'apprentissage par la machine.

Utiliser les relations dans le domaine du son pourrait également être une piste à suivre.

Pour terminer, nous tenons à rappeler que d'autres stratégies de composition existent [23] et que l'être humain ne peut être écarté du processus global de composition, à moins d'obtenir des résultats esthétiquement insatisfaisants. La meilleure optique est sans doute de combiner les différentes techniques de composition entre elles.

Bibliographie

- [1] http://fr.wikipedia.org/wiki/Langage_graphique.
- [2] <http://repmus.ircam.fr/openmusic/home>.
- [3] <http://www.emn.fr/z-info/choco-solver/>.
- [4] http://fr.wikipedia.org/wiki/Fichier:YB0256_Intervalle_renverse.png.
- [5] <http://www.swig.org/>.
- [6] <http://common-lisp.net/project/cffi/>.
- [7] <http://www.comet-online.org/>.
- [8] <http://jacop.osolpro.com/>.
- [9] http://fr.wikipedia.org/wiki/Temperament_egal.
- [10] <http://maude.cs.uiuc.edu/>.
- [11] <http://www.cs.unm.edu/~mccune/prover9/>.
- [12] <http://www.cl-user.net/asp/libs/screamer>.
- [13] <http://support.ircam.fr/forum-ol-doc/om/om6-manual/co/Lisp.html>.
- [14] <http://support.ircam.fr/forum-ol-doc/om/om6-manual/co/OM-Documentation.html>.
- [15] <http://common-lisp.net/project/babel/>.
- [16] <http://common-lisp.net/project/alexandria/>.
- [17] <http://www.cliki.net/trivial-features>.
- [18] <http://www.cliki.net/ASDF-Install>.
- [19] <http://www.quicklisp.org/>.
- [20] <http://www.gecode.org/>.
- [21] Carlos AGON, Gérard ASSAYAG, Jaccobo BABONI, Karim HADDAD, Matthew LIMA, and Mikhail MALT. *OpenMusic Reference Manual*, May 2003.
- [22] Carlos AGON, Gérard ASSAYAG, Jaccobo BABONI, Karim HADDAD, Matthew LIMA, and Mikhail MALT. *OpenMusic Tutorial*, May 2003.
- [23] Torsten Anders. *Composing Music by Composing Rules : Design and Usage of a Generic Music Constraint System*. PhD thesis, Queen's University Belfast, February 2007.
- [24] Antoine Bonnet and Camilo Rueda. *OpenMusic Situation version 3*, Avril 1999.

- [25] Jean Bresson Carlos Agon, Gérard Assayag, editor. *THE OM COMPOSER'S BOOK.1*. Musique/Sciences. DELATOUR FRANCE, 2006.
- [26] Jean Bresson Carlos Agon, Gérard Assayag, editor. *THE OM COMPOSER'S BOOK.2*. Musique/Sciences. DELATOUR FRANCE, 2008.
- [27] Guido Tack Christian Schulte and Mikael Z. Lagerkvist. Case studies. In Guido Tack Christian Schulte and Mikael Z. Lagerkvist, editors, *Modeling and Programming with Gecode*. 2011. Corresponds to Gecode 3.5.0.
- [28] Guido Tack Christian Schulte and Mikael Z. Lagerkvist. Modeling. In Guido Tack Christian Schulte and Mikael Z. Lagerkvist, editors, *Modeling and Programming with Gecode*. 2011. Corresponds to Gecode 3.5.0.
- [29] Yves Deville. Programmation par contraintes (cours) - université catholique de louvain.
- [30] Gustavo Gutiérrez. Constraint programming on relations : Theory, practice and implementation. PhD. confirmation document, January 2011.
- [31] Brian Henderson-Sellers and David Cornish Cooper. Has classical music a fractal nature? a reanalysis. *Computers and the Humanities.*, 27(4) :277–284, 1993.
- [32] K.J. Hsü and A.J. Hsü. Fractal geometry of music. *Proceedings of the National Academy of Sciences of the United States of America*, 87(3) :938, 1990.
- [33] Johannes Kretz. Navigation of structured material in second horizon for piano and orchestra. In Jean Bresson Carlos Agon, Gérard Assayag, editor, *THE OM COMPOSER'S BOOK.1*, pages 107, 113. 2006.
- [34] Serge Lemouton. Generating melodic, harmonic and rythmic processes in k,... an opera by philippe manoury. In Jean Bresson Carlos Agon, Gérard Assayag, editor, *THE OM COMPOSER'S BOOK.1*, page 115. 2006.
- [35] Fabien Lévy. When the computer enables freedom from the machine (on an outline of the work hérédo-ribotes). In Jean Bresson Carlos Agon, Gérard Assayag, editor, *THE OM COMPOSER'S BOOK.1*, pages 125, 126. 2006.
- [36] Orjan Sandred. *OpenMusic RC library Tutorial*, March 2000.
- [37] Christian Schulte. Programming search engines. In Guido Tack Christian Schulte and Mikael Z. Lagerkvist, editors, *Modeling and Programming with Gecode*. 2011. Corresponds to Gecode 3.5.0.
- [38] Charlotte Truchet. *Contraintes, recherche locale et composition assistée par ordinateur*. PhD thesis, Ircam, Décembre 2003.
- [39] Iannis Xenakis. Musique formelles. *La revue musicale*, 1963.

Annexes

A

Informations complémentaires

A.1 Utilisation de code Common Lisp dans OpenMusic

Nous décrivons ici la manière dont il est possible d'exécuter du code Common Lisp externe à OpenMusic à partir de l'environnement. Plusieurs approches sont possibles, mais nous ne décrivons que celle utilisée dans ce mémoire. Pour une information complète, nous renvoyons le lecteur au chapitre traitant de cela dans le manuel de l'utilisateur d'OpenMusic [13].

Deux questions sont posées ici : comment exécuter du code Common Lisp à partir de l'environnement de travail, et comment permettre l'utilisation de code déjà implémenté, sous la forme d'une librairie externe pour OpenMusic. Pour ce qui est de l'exécution de code Common Lisp, deux possibilités ont été utilisées dans ce travail : la première est l'utilisation du listener fourni par OpenMusic, et la seconde est la définition d'une lambda-fonction. Ceci permet de définir des fonctions pouvant être intégrées de manière visuelle dans l'environnement d'OpenMusic.

Ensuite, à propos de l'intégration de code Common Lisp comme librairie externe, l'option utilisée est très simple : il suffit de placer un fichier contenant tout le code destiné à être chargé au moment du lancement d'OpenMusic dans un dossier dédié nommé *patches*. Il nous faut indiquer ici que ce fichier de chargement a dû être modifié en raison du fait qu'il y avait un conflit entre celui-ci et le code chargé pour OpenMusic : tous deux chargeaient CFFI¹. Afin de ne pas modifier le code d'OpenMusic, le chargement de CFFI a été retiré du fichier de chargement de la librairie externe. Afin d'appeler des fonctions Common Lisp définies dans cette librairie externe, un package a été défini par les développeurs d'OpenMusic : *cffl-user*. A chaque appel d'une fonction de la librairie externe, il faut donc spécifier ce package.

1. CFFI et la raison de son utilisation sont décrits dans l'annexe B.

A.2 Description de la résolution du problème des n reines avec Gecode

Le but ici n'est pas de décrire Gecode dans sa totalité mais plutôt d'expliquer la manière générale dont on modélise un problème à l'aide de cette librairie. Nous décrivons donc dans ce qui suit la manière de résoudre un problème bien connu de la programmation par contraintes : le problème des n reines.

Pour rappel, le but est ici de placer n reines sur un échiquier de $n * n$ cases, sans qu'aucune ne puisse attaquer les autres. Il s'agit là d'un problème combinatoire a priori assez simple, mais qui est np-complet. Il est cependant résolu de manière très efficace par la programmation par contraintes.

Voici donc la manière dont ce problème est modélisé avec Gecode² :

Afin de modéliser un problème, il faut créer une classe fille de la classe *Space* dans laquelle on définira le problème à résoudre. En l'occurrence, on définit une classe *Queens* qui étend la classe *Script*³ qui étend elle-même la classe *Space* :

```
class Queens : public Script
```

Une variable d'instance est ensuite définie. Son type est une classe qui définit un tableau de variables de décision, variables représentant un nombre entier. C'est cette variable qui contiendra les positions des différentes reines sur le plateau⁴ une fois la solution au problème trouvée.

```
IntVarArray q;
```

Le constructeur est ensuite défini et c'est précisément dans celui-ci qu'on impose les contraintes sur les variables de décision pour un problème donné. Dans ce cas, il s'agit d'interdire qu'une reine ne soit sur la même colonne, la même ligne ou la même diagonale qu'une autre reine. Ceci peut être exprimé à l'aide de différents propagateurs. C'est une des raisons pour lesquelles le constructeur prend en paramètre un objet contenant un ensemble d'options. En fonction des options passées en arguments au constructeur, celui-ci construira l'objet selon une modélisation du problème plutôt qu'une autre. Nous donnons ici le code d'une modélisation utilisant des propagateurs binaires et un propagateur implémentant la contrainte globale⁵ très connue *AllDiff* :

```
for (int i = 0; i < n; i++)
  for (int j = i + 1; j < n; j++) {
```

2. L'ensemble du code est disponible sur le site web de Gecode [20]

3. Cette classe fournit des outils d'aide supplémentaires pour l'exécution d'un programme dans le terminal. Elle contient par exemple en plus de la classe *Space* une méthode *print*.

4. Techniquement parlant, la valeur contenue dans une variable ne donne que le numéro de colonne, l'index de la variable dans le tableau imposant le numéro de ligne, ou vice-versa.

5. Différentes définitions de *globalité* existent dans ce cadre [29], mais de manière générale, une contrainte globale est une contrainte pouvant s'appliquer à un nombre quelconque (sauf 1) de variables.

```

    rel(*this, q[i]+i != q[j]+j);
    rel(*this, q[i]-i != q[j]-j);
}
distinct(*this, q, opt.icl());

```

Après avoir imposé les contraintes, il faut définir les heuristiques de recherche dans l'arbre de recherche. Ceci est réalisé à l'aide de la méthode *branch*. Dans ce cas, la recherche choisira d'abord les variables dont le domaine est le plus petit et assignera d'abord les valeurs les plus petites du domaine :

```
branch(*this, q, INT_VAR_SIZE_MIN, INT_VAL_MIN);
```

Finalement, il reste à définir le constructeur de copie, c'est-à-dire un constructeur qui prend en paramètre un objet de la classe elle-même. Ceci est indispensable car Gecode fait de la recherche « copy-based ». Ce constructeur a entre autre pour tâche de mettre à jour les domaines des variables de décision. On peut également remarquer que le constructeur de copie prend un second paramètre booléen, afin de définir si l'espace sera partagé ou non. Gecode permet en effet de partager un espace entre différentes recherches.

```
Queens(bool share, Queens& s) : Script(share, s) {
    q.update(*this, share, s.q);
}

```

Une dernière méthode est également à implémenter : la méthode *copy* qui a pour tâche d'appeler le constructeur de copie.

```
virtual Space*
copy(bool share) {
    return new Queens(share, *this);
}

```

En C++, les constructeurs et constructeurs de copie ne sont pas hérités d'une classe parente. C'est la raison pour laquelle il faut nécessairement les définir lorsqu'on écrit une classe héritant de la classe *Space*. La méthode *copy* est quant à elle déclarée virtuelle et sans corps dans la classe *Space*. Il faut donc absolument également l'implémenter lors de l'écriture d'une classe héritant de cette classe. La raison de son existence est que les objets de la classe *SearchEngine* ont accès à un modèle uniquement par le biais de la classe *Space*. Ils ne peuvent donc pas accéder aux constructeurs d'une classe fille. La méthode *copy* étant virtuelle et implémentée dans la classe fille permet quant à elle d'accéder indirectement au constructeur de copie de la classe fille. C'est donc via cette méthode que les instances de la classe *SearchEngine* accéderont au constructeur de copie.

Maintenant que cette classe est définie, il faut, avant de pouvoir trouver une solution, définir les options de l'exécution (comme le nombre d'itérations, ...); ceci est fait en créant un objet de type *Options*. Cet objet est alors passé en argument à la fonction *run*, qui permettra de

lancer effectivement la recherche. Cette fonction prend de plus en argument `template` la classe qui modélise le problème, une classe définissant le moteur de recherche à utiliser (en l'occurrence, la recherche en profondeur) ainsi qu'une classe fille de la classe `Options`. Ce dernier `template` argument permet de définir certaines options à passer au script exécutable obtenu après compilation.

Remarquons que, de manière interne, le moteur de recherche a le rôle d'appeler les brancheurs et de définir à quel moment il le fait. Dans la classe fille de la classe `Space`, on définit les brancheurs à utiliser, bien qu'ils ne fassent pas à proprement parler partie du modèle. Dans Gecode, il est plus correct de dire qu'ils sont l'intermédiaire entre l'espace et le moteur de recherche (voir la figure 1.9).

```
Script :: run<Queens,DFS,SizeOptions>(opt);
```

A.3 Utilisation des entiers, des booléens et des ensembles d'entiers en tant que variables de décision dans Gecode

Nous expliquons ici comment représenter les entiers, les booléens et finalement les ensembles d'entiers par des variables dans Gecode. Pour chacun des cas, le concept est encapsulé par une classe qui étend la classe `VarImpVar`.

Les entiers

Afin d'utiliser une variable représentant un entier dans Gecode, il faut créer une instance de la classe `IntVar`. Cette classe possède plusieurs constructeurs :

```
IntVar (void);
IntVar (const IntVar &y);
IntVar (const Int::IntView &y);
IntVar (Space &home, int min, int max);
IntVar (Space &home, const IntSet &d);
```

Les trois premiers constructeurs n'ajoutent pas de variables de décision dans le modèle. Les deux premiers sont respectivement le constructeur par défaut et le constructeur de copie. En C++, le compilateur crée toujours automatiquement ces constructeurs si on ne les déclare pas. Cependant, il est toujours recommandé d'au moins les déclarer, pour que le compilateur ne fournisse pas une implémentation dont on n'est pas sûr du comportement.⁶

Le troisième constructeur quant à lui est utilisé de manière interne par Gecode et n'est donc pas à utiliser lors de la modélisation d'un problème (Les objets de type `View` sont quant à eux utilisés pour l'implémentation de propagateurs et de brancheurs).

6. Par exemple, un constructeur de copie peut soit créer un nouvel identificateur à partir de l'ancien, soit recréer un objet en mémoire, selon l'implémentation du constructeur. Mieux vaut donc définir soi-même ce que fait un constructeur afin d'éviter des erreurs inattendues.

Les deux derniers permettent de véritablement créer une nouvelle variable de décision. Ils prennent en effet en paramètre une référence vers un objet de type *Space*. Le constructeur rattachera donc cette variable au modèle, encapsulé par un objet de type *Space*. Pour ces deux constructeurs, il faut comme toujours en programmation par contraintes définir le domaine de la variable. On peut soit le faire à l'aide d'un intervalle borné par deux valeurs, soit à l'aide d'un ensemble d'entiers (à ne pas confondre avec les variables de décision représentant un ensemble d'entiers).

Remarquons ici que Gecode représente de manière interne le domaine d'une variable par une liste d'intervalles et ne permet pas de représentations différentes. En effet, de manière interne, le constructeur prenant deux bornes en paramètres utilisera celui qui prend un ensemble d'entiers. Ce dernier représente alors le domaine par une liste d'intervalles. D'autres systèmes comme JaCoP [8] permettent quant à eux de représenter les domaines des variables de plusieurs manières.

```
IntVar(sp, 1, 5)
```

FIGURE A.1 – Définition d'une variable représentant un entier dont la valeur est 1, 2, 3, 4 ou 5.

Tableaux d'entiers

Gecode permet également de créer des tableaux d'entiers. Il existe deux classes permettant de représenter des tableaux d'entiers dans la librairie : *IntVarArray* et *IntVarArgs*. La première contient un ensemble d'instances de la classe *IntVar* alors que la seconde contient un ensemble de **pointeurs** vers des instances de cette classe.

Les propagateurs prennent normalement en paramètre des objets de la classe *IntVarArgs*. Ceci permet de travailler avec des tableaux de variables qui sont déjà présentes en mémoire. Cependant, le casting dynamique d'un objet de la classe *IntVarArray* vers le type *IntVarArgs* est possible. Il est donc également possible d'utiliser une instance de la classe *IntVarArray* en tant que paramètre d'un propagateur. Nous le verrons plus loin, dans notre cas, nous ne travaillerons qu'avec la classe *IntVarArgs*, raison pour laquelle nous ne présentons que les constructeurs de celle-ci :

```
IntVarArgs (void);
IntVarArgs (int n);
IntVarArgs (const IntVarArgs &a);
IntVarArgs (const VarArray< IntVar > &a);
IntVarArgs (Space &home, int n, int min, int max);
IntVarArgs (Space &home, int n, const IntSet &s);
```

A nouveau, seuls les deux derniers permettent d'ajouter des variables de décision au modèle d'un problème. Ils ont le même effet que les constructeurs de la classe *IntVar* ajoutant des variables de décision hormis qu'au lieu de créer une seule variable, ils créent un tableau de n

variables. Le premier et le troisième sont respectivement le constructeur par défaut et le constructeur de copie. Finalement, le quatrième constructeur permet entre autres le casting dynamique mentionné ci-dessus. Il est toujours possible d'ajouter une variable dans ces tableaux à l'aide de l'opérateur <<.

```
IntVarArgs (sp, 3, 1, 5)
```

FIGURE A.2 – Définition d'un tableau de 3 variables représentant un entier dont la valeur est 1, 2, 3, 4 ou 5.

Les booléens

Les booléens sont gérés de manière assez similaire aux entiers dans Gecode. Il faut cette fois instancier la classe *BoolVar*, à l'aide de l'un des constructeurs suivants :

```
BoolVar (void);
BoolVar (const BoolVar &y);
BoolVar (const Int::BoolView &y);
BoolVar (Space &home, int min, int max);
```

Les constructeurs sont très semblables aux constructeurs de la classe *IntVar*. La seule véritable différence est qu'il n'y a qu'un seul moyen de définir le domaine de la variable. En effet, dans le cas des booléens les deux manières de définir le domaine précédemment sont maintenant équivalentes puisque seules les valeurs 0 et 1 peuvent en faire partie.

Tableaux de booléens

Comme attendu, les tableaux de booléens sont gérés de manière similaire aux tableaux d'entiers. Il existe encore une fois deux classes différentes, *BoolVarArray* et *BoolVarArgs*. La première représente un tableau d'objets de la classe *BoolVar* et la seconde un tableau de pointeurs vers ce type d'objet. Dans ce travail, nous n'utilisons que la deuxième classe. Nous présentons donc les constructeurs de la deuxième classe :

```
BoolVarArgs (void);
BoolVarArgs (int n);
BoolVarArgs (const BoolVarArgs &a);
BoolVarArgs (const VarArray< BoolVar > &a);
BoolVarArgs (Space &home, int n, int min, int max);
```

A nouveau, les constructeurs sont fort similaires aux constructeurs de la classe *IntVarArgs* et les domaines de nouvelles variables de décision ne sont représentables que d'une seule manière.

Les ensembles d'entiers

Nous présentons encore une fois les différents constructeurs pour ce type de variables :

```

SetVar(void);
SetVar(const SetVar& y);
SetVar(const Set::SetView& y);
SetVar(Space& home);
SetVar(Space& home, int glbMin, int glbMax, int lubMin, int lubMax, unsigned int
    cardMin = 0, unsigned int cardMax = Set::Limits::card);
SetVar(Space& home, const IntSet& glbD, int lubMin, int lubMax, unsigned int cardMin
    = 0, unsigned int cardMax = Set::Limits::card);
SetVar(Space& home, int glbMin, int glbMax, const IntSet& lubD, unsigned int cardMin
    = 0, unsigned int cardMax = Set::Limits::card);
SetVar(Space& home, const IntSet& glbD, const IntSet& lubD, unsigned int cardMin =
    0, unsigned int cardMax = Set::Limits::card);

```

Une fois encore, les deux premiers sont le constructeur par défaut et le constructeur de copie, et le troisième est utilisé de manière interne par Gecode. Les constructeurs restants permettent d'ajouter une variable de décision au modèle. Le quatrième permet d'ajouter une variable représentant un ensemble d'entiers dont le domaine possède pour plus grande borne inférieure l'ensemble vide et pour plus petite borne supérieure l'ensemble de tous les entiers⁷. Le cinquième permet de définir le domaine via deux bornes (entières) pour la borne inférieure et deux bornes (entières) pour la borne supérieure. Comme pour tous les constructeurs suivants, deux paramètres optionnels peuvent être passés pour définir la cardinalité minimale et maximale de l'ensemble créé. Le sixième constructeur permet d'utiliser une instance de la classe *IntSet* pour définir la borne inférieure du domaine, à la place de deux bornes entières. Les deux derniers constructeurs sont aisément compréhensibles à l'aide des descriptions des constructeurs précédents.

```
SetVar(sp, 1, 1, 1, 2, 1, 2);
```

FIGURE A.3 – Définition d'une variable représentant un ensemble d'entiers. L'ensemble doit contenir l'élément 1 et peut contenir l'élément 2. Sa cardinalité est soit 1, soit 2. Les ensembles possibles sont donc $\{1\}$ et $\{1,2\}$. La représentation interne du domaine est $[\{1\} \dots \{1,2\}]$, $\#[1 \dots 2]$.

Tableaux d'ensembles d'entiers

Comme pour les deux autres types de variables, il est possible de faire des tableaux de ces variables. Deux classes différentes existent, *SetVarArgs* et *SetVarArray*, et c'est une fois encore la première qui nous permet de modéliser un problème. La liste des différents constructeurs de

7. La cardinalité d'un ensemble devant pouvoir être contenue dans une instance de la classe *IntVar*, elle est limitée par l'élément maximal qu'une telle variable peut contenir.

cette classe est la suivante :

```

SetVarArgs(void);
SetVarArgs(int n);
SetVarArgs(const SetVarArgs& a);
SetVarArgs(const VarArray<SetVar>& a);
SetVarArgs(Space& home, int n, int glbMin, int glbMax, int lubMin, int lubMax,
    unsigned int minCard = 0, unsigned int maxCard = Set::Limits::card);
SetVarArgs(Space& home, int n, const IntSet& glb, int lubMin, int lubMax, unsigned
    int minCard = 0, unsigned int maxCard = Set::Limits::card);
SetVarArgs(Space& home, int n, int glbMin, int glbMax, const IntSet& lub, unsigned
    int minCard = 0, unsigned int maxCard = Set::Limits::card);
SetVarArgs(Space& home, int n, const IntSet& glb, const IntSet& lub, unsigned int
    minCard = 0, unsigned int maxCard = Set::Limits::card);

```

Ces constructeurs sont aux constructeurs de la classe *SetVar* ce que les constructeurs de la classe *IntVarArgs* sont aux constructeurs de la classe *IntVar*. Le paramètre n requis pour certains d'entre eux définit donc la taille du tableau d'ensemble d'entiers.

```
SetVarArgs(sp, 3, 1, 1, 1, 2, 1, 2);
```

FIGURE A.4 – Définition d'une variable représentant un tableau de 3 ensembles d'entiers. Ceux-ci possèdent les mêmes domaines que l'ensemble d'entiers de l'exemple précédent.

A.4 Utilisation pratique des relations dans Gecode

Nous expliquons d'abord les variables, puis les propagateurs, et finalement les brancheurs. La manière dont on exprime la recherche ne change pas, car elle est orthogonale à la modélisation de manière générale⁸.

Variables

La classe destinée à représenter les relations est la classe *CPRelVar*.

Le seul constructeur que l'utilisateur doit connaître (et disponible pour le moment) est le suivant :

```
CPRelVar(Space& home, const CPreL::GRelation& l, const CPreL::GRelation& u);
```

Ce constructeur prend en paramètre l'espace, ainsi que deux relations ground, permettant d'approximer le domaine relationnel de la relation. Il est clair que les deux relations ground

8. Pour être exact, seul le paramètre template de la classe qui modélise le problème change.

doivent être de même arité. Ces relations ground peuvent être construites à l'aide du constructeur suivant (pour l'utilisateur) :

```
GRelation(int a);
```

Ce constructeur définit une relation ground vide, d'arité a . Afin d'ajouter des tuples à cette relation, il faudra utiliser la méthode *add*. Deux fonctions permettent également de créer une relation ground :

```
GRelation create(const std::vector<Tuple>& dom);
GRelation create_full(int a);
```

La première prend un vecteur de tuple et sera donc une relation ground comprenant cet ensemble de tuples. Tous les tuples du vecteur doivent bien sûr être de même arité. La seconde crée une relation ground qui comprend tous les tuples de l'univers (représentables par la machine) d'arité a .

Il reste à présenter comment construire un tuple. Il existe pour cela trois constructeurs pour l'utilisateur :

```
Tuple(int k);
Tuple(int a, int b);
Tuple(int a, int b, int c);
```

Le premier construit un tuple d'arité k mais vide. Il faudra donc le remplir par la suite. Le second crée un tuple binaire qui contiendra les entiers a et b . Le dernier fait de même pour un tuple ternaire avec les valeurs a , b et c .

```
Tuple t(0,0);
GRelation glb(2);
GRelation lub(2);
lub.add(t);
CPRelVar(sp, glb, lub);
```

FIGURE A.5 – Définition d'une variable représentant une relation binaire dont les valeurs possibles sont $\{\}$ et $\{(0,0)\}$.

Comme pour les autres types de variable, il est également possible de créer des tableaux de variables. Ces objets sont représentés par la classe *CPRelVarArgs*, et fonctionnent de manière similaire aux autres.

Propagateurs

Etant donné que le module de cette librairie est toujours en construction à ce jour, il n'existe que quelques propagateurs. Voici la liste de ceux-ci :

```
equal (Gecode::Space& home, CPreVar A, CPreVar B);
```

Sa sémantique est :

$$A = B$$

```
complement (Gecode::Space& home, CPreVar A, CPreVar B);
```

Sa sémantique est :

$$A = \overline{B}$$

```
intersect (Gecode::Space& home, CPreVar A, CPreVar B, CPreVar C);
```

Sa sémantique est :

$$A \cap B = C$$

```
Union (Gecode::Space &home, CPreVar A, CPreVar B, CPreVar C);
```

Sa sémantique est :

$$A \cup B = C$$

```
subset (Gecode::Space &home, CPreVar A, CPreVar B);
```

Sa sémantique est :

$$A \subseteq B$$

```
disjoint (Gecode::Space &home, CPreVar A, CPreVar B);
```

Sa sémantique est :

$$A \cap B = \emptyset$$

```
implies (Gecode::Space &home, CPreVar A, CPreVar B, CPreVar C);
```

Sa sémantique est :

$$\forall t : t \in A \implies t \in B \iff t \in C$$

Brancheurs

Le seul brancheur existant à ce jour est le suivant :

```
branch (Home home, CPreIVar R);
```

Il s'agit d'un brancheur « naïf » : il choisit un tuple dans l'ensemble des tuples inconnus de la relation et crée deux noeuds fils dans l'arbre de recherche, l'un incluant le tuple et l'autre l'excluant.

A.5 Deux systèmes existants : Situation et OMRC

Nous décrivons à présent de manière générale deux systèmes de contraintes pour des problèmes musicaux : **Situation** et la librairie **RC** pour OpenMusic⁹. Apprendre à propos de ces systèmes nous a permis de mieux connaître ce qui l'était possible de réaliser aujourd'hui, et quels types de MCSP étaient abordables. Les deux systèmes sont des systèmes de contraintes génériques. Les types de problèmes et la manière dont ils les abordent sont cependant différents :

- Situation était à la base plutôt adapté à des problèmes harmoniques. Il a cependant été étendu afin de permettre de résoudre d'autres problèmes, musicaux (rythmiques) ou non.
- OMRC est spécifique pour les problèmes rythmiques.

Avant de présenter ces deux systèmes, nous devons préciser qu'ils ont été créés pour être utilisés dans l'environnement OpenMusic. OpenMusic ne permet qu'une seule représentation des paramètres musicaux des notes, utilisant des entiers et des valeurs MIDI¹⁰. La représentation est donc bien uniforme [23]. Puisque GeLisp est destiné à être utilisé dans OpenMusic, nous serons dans le même cas.

Situation

Situation est une librairie pour le langage OpenMusic, permettant de résoudre des MCSP, à la base spécifiquement harmoniques. Elle est spécialement adaptée pour la génération d'objets musicaux lorsque leurs éléments internes doivent satisfaire certaines contraintes et qu'ils sont liés par un ensemble de propriétés. Tout comme OpenMusic, Situation a été pensé pour que des utilisateurs n'étant pas informaticiens puissent l'utiliser.

Trois notions sont définies dans Situation : les *objets*, les *points* et les *distances*. Un objet est simplement un ensemble de points. Les distances sont quant à elle mesurées entre des points. On différencie également les *distances internes*, qui sont mesurées entre deux points d'un même objet, *des distances externes*, qui sont mesurées entre des points d'objets distincts.

9. raccourci ici par « OMRC ».

10. Pour d'autres choses, comme le traitement du son, OpenMusic fournit bien d'autres outils.

Les points et les distances constituent les variables de décision d'un CSP défini à l'aide de Situation. Leur domaine se limite aux entiers.

Remarquons que Situation ne donne pas de limitations quant à ce que les variables représentent. C'est pour cette raison que depuis sa version 3 elle permet de résoudre d'autres problèmes que des problèmes harmoniques. Ceci fait donc d'elle un système plus générique. Cependant, Situation reste surtout bien adapté pour des problèmes de suites d'accords. Une autre raison pour laquelle on peut dire qu'elle est générique est que le lien entre points et distances peut être modifié, selon ce qu'on veut que les points représentent. De plus, elle fournit par défaut des contraintes permettant d'exprimer des règles de composition généralisées. Elle permet également de définir ses propres contraintes.

Le langage défini par cette librairie fournit également des mécanismes pour exprimer quelles seront les variables qui doivent être contraintes par une contrainte donnée. Ces mécanismes sont basés sur un indexage des éléments. Ceci possède l'inconvénient d'imposer une représentation séquentielle de la musique. Par exemple, une représentation hiérarchique de plusieurs partitions n'est pas possible. De plus, le langage pour les mécanismes de choix ne peut être étendu par l'utilisateur, ce qui limite les choix d'application des règles.

La structure qu'utilise Situation pour représenter la musique ne permet que de traiter des relations en fonction de la position des objets et de distances. Ceci ne convient pas forcément pour appliquer des règles complexes, comme par exemple pour les **note de passage**. On peut les définir comme des notes non-harmoniques jouées sur un temps faible, et qui dépendent d'autres notes du contexte.

Il n'est pas simple d'exprimer de contraintes sur les notes de passages uniquement en fonction de positions et de distances. En effet, savoir si une note est sur un temps faible ou non ne fait pas immédiatement partie de l'information dont on dispose. Utiliser l'harmonie sous-jacente en tant qu'information n'est pas non plus aisé.

L'algorithme de recherche exécuté dans Situation est nommé *first-found forward checking*. Il est basé sur le *minimal forward checking*, qui fait du forward checking mais est basé sur des exécutions « paresseuses », c'est-à-dire qu'il retarde l'élagage du domaine des variables au moment où c'est nécessaire. Les domaines larges (comme ceux des séquences de distance) sont structurés hiérarchiquement, et l'algorithme first-found forward checking tire parti de cette structure pour savoir comment et quand exécuter les différents propagateurs.

Un avantage que permet Situation est la possibilité d'utiliser des contraintes faibles. Une valeur entière de préférence doit alors être passée en argument au propagateur en question. Par contre, les contraintes globales ne sont apparemment pas bien gérées. [38]

Une autre limitation que possède Situation est le fait de ne pouvoir utiliser qu'un seul espace de recherche. Dans certains cas, il est adéquat de pouvoir exécuter plusieurs recherches en même temps et de les combiner. Gecode permettant ce mécanisme, GeLisp le permet également.

A titre de comparaison, nous donnons ici un exemple de problème simple¹¹ [22] résolu par GeLisp et par Situation. Il s'agit de trouver une suite de 8 accords de densité 3, en utilisant les valeurs MIDI comprises entre 60 et 72. Les intervalles verticaux autorisés sont la seconde majeure (2) et la quinte juste (7). Les intervalles horizontaux admis pour la voix supérieure sont la seconde mineure (1) et la tierce mineure (3).

Les deux patches permettant d'implémenter ce problème, respectivement avec GeLisp et Situation, sont illustrés dans la figure A.6. La différence majeure est dans la fonction lambda, qui est implémentée en Common Lisp en utilisant GeLisp¹². Remarquons la simplicité de l'implémentation avec Situation : seuls quelques paramètres et quelques icônes suffisent. Par contre, la liberté de modélisation et de recherche n'est pas aussi grande (on ne peut utiliser que les propagateurs fournis et un seul moteur de recherche). On peut ajouter que l'implémentation à l'aide de GeLisp a utilisé des contraintes dans ce cas, alors Situation ne pose ici pas de contraintes, et ne fait que restreindre les valeurs des domaines lors de la modélisation.

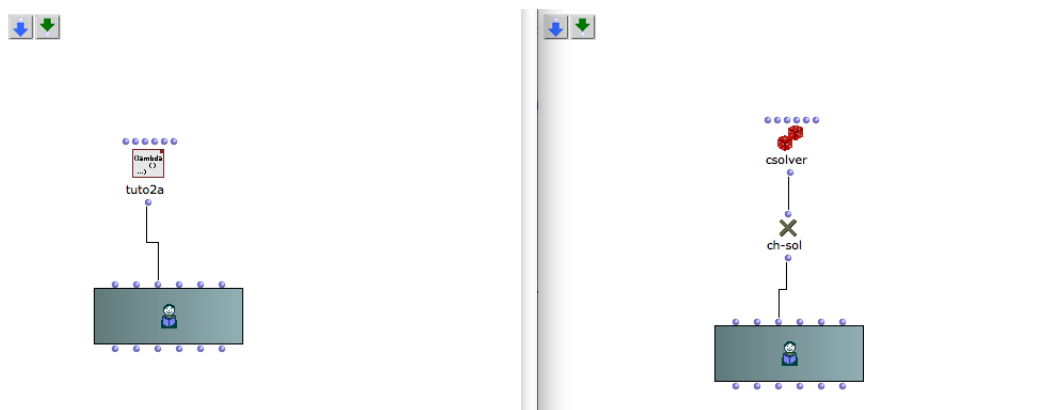


FIGURE A.6 – Patches implémentant le même problème, à gauche à l'aide de GeLisp, à droite à l'aide de Situation.

Deux résultats obtenus, l'un avec GeLisp, l'autre avec Situation, sont respectivement donnés aux figures A.7 et A.8.

OMRC

OMRC est une librairie OpenMusic totalement vouée à la résolution de MCSP **rythmiques**. Elle a été conçue par le compositeur Örjan Sandred. Le but de (OM)RC n'est pas de générer rapidement des résultats satisfaisants, mais de fournir un cadre pour que des compositeurs puissent travailler sur des règles rythmiques.

Elle se base en réalité sur deux moteurs de recherche déjà existants : celui de Situation (nommé **Csolver**), et celui d'*OMCS* (nommé **pmc**), déjà récupéré de la librairie *PWConstraints*. Les deux ont des fonctionnements et utilisent des formats différents. Le moteur **pmc**

11. Tiré d'un tutoriel écrit pour Situation.

12. code fournit à l'annexe C

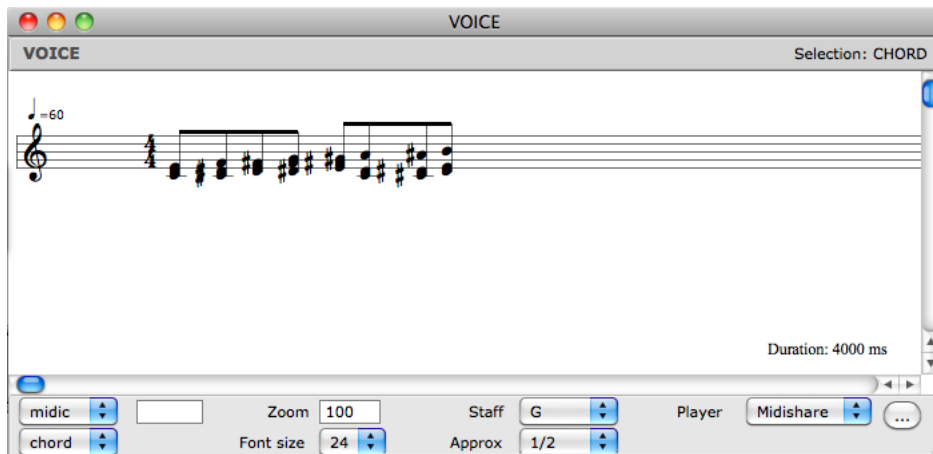


FIGURE A.7 – Résultat obtenu à l'aide de GeLisp utilisé dans OpenMusic.

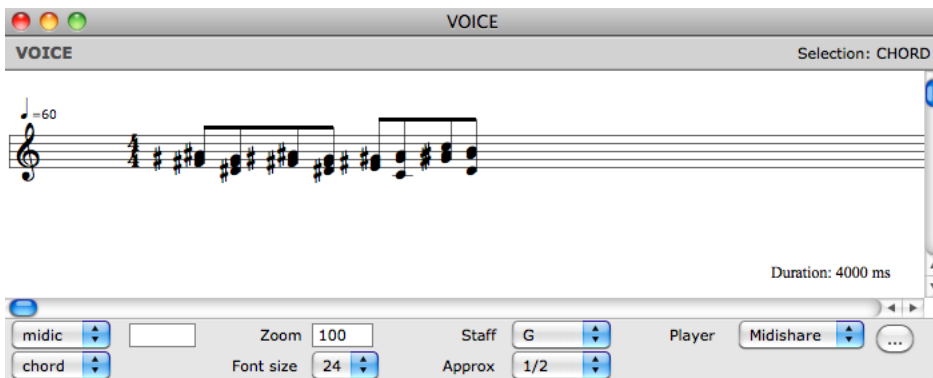


FIGURE A.8 – Résultat obtenu à l'aide de Situation utilisé dans OpenMusic.

s'est apparemment montré plus efficace pour RC. Nous ne décrivons donc ici que l'utilisation de RC avec *pmc*, bien que les deux moteurs soient utilisables.

Afin de définir le domaine de variables de décision, il existe une fonction nommée *voice-domain*. Les trois types d'éléments pouvant être présents dans le domaine de variables de décision sont : une **durée de note**, un **motif rythmique** ou une **signature**. La première entrée de *voice-domain* est toujours réservée pour les signatures. La seconde ou les autres optionnelles sont utilisées pour les durées de notes et/ou les motifs rythmiques. Les entrées optionnelles servent en fait à définir des niveaux supplémentaires pour la partition rythmique créée. Il peut y en avoir jusqu'à 4, ce qui fait donc 5 entrées maximum pour la fonction *voice-domain*. Le moteur de recherche retient quels éléments appartiennent à quels niveaux. Par contre, ces niveaux peuvent avoir de l'influence les uns sur les autres. La manière dont les niveaux sont interprétés est laissée libre à l'utilisateur. Il peut par exemple s'agir de voix complémentaires, ou plutôt de niveaux hiérarchiques.

Une fonction *voice-domain* permet de créer une voix. La librairie RC permet d'aller jusqu'à 7 voix. Autrement dit, il est possible de créer jusqu'à 35 domaines différents, 7 domaines étant alors réservés pour les signatures des 7 voix différentes.

Afin d'être compatible avec *pmc*, RC fournit des fonctions de conversions. Dans ce cas, il s'agit de la fonction *domain->pmc*. La sortie de la fonction *voice-domain* est utilisée comme entrée à cette fonction. L'autre paramètre est le nombre de variables pour cette voix. Le nombre de variables détermine la longueur de la séquence d'éléments. Ces données converties sont alors passées au moteur de recherche. Après avoir résolu un problème, il faut encore décoder la solution afin de pouvoir l'utiliser dans le langage OpenMusic.

Au niveau des règles, c'est-à-dire des contraintes, il existe également une fonction pour les convertir afin qu'elles soient compréhensibles par *pmc* : *rules->pmc*. Il existe des règles prédéfinies dans RC exprimant des règles de composition. D'autres « règles », influencent le comportement du moteur de recherche. Par exemple, la règle *r-eqlength* conserve des séquences de même longueur au moment de la recherche. Ceci permet d'éviter d'avoir des séquences de signatures très longues et une séquence de motifs très courte, ce qui n'est en général pas désiré. La règle *r-canon* permet quant à elle de construire un canon rythmique entre deux niveaux d'une même voix. D'autres règles contrôlant les relations entre des événements rythmiques, des signatures et des pulsations existent dans RC. Il est par exemple possible de contraindre les types de syncopes ainsi que les temps sur lesquels elles peuvent avoir lieu.

De manière générale, deux types de règles existent, les règles régulières, qui s'appliquent au sein d'une même voix, et les règles globales, utilisées entre deux voix différentes.

Il est également possible de définir de nouvelles règles, selon ses besoins. La manière dont elles sont définies doit suivre une ligne de conduite imposée par RC. Des contraintes faibles existent également, les problèmes abordés pouvant parfois être sur-contraints.

Une commodité que permet RC est de pouvoir verrouiller certaines parties de solution, afin de conserver une partie de résultat satisfaisant et travailler sur le reste de la pièce.

GeLisp pourrait permettre de résoudre des problèmes résolus par RC. GeLisp est en effet générique puisqu'elle permet la programmation par contraintes de manière générale (sa limitation étant bien sûr le potentiel de Gecode). Cependant, une couche supplémentaire est nécessaire afin de pouvoir permettre à un utilisateur de modéliser un problème aussi simplement qu'avec RC. Ceci est spécialement vrai pour des compositeurs n'étant pas informaticiens. En effet, RC étant une librairie OpenMusic, elle utilise le paradigme de programmation visuelle. GeLisp ne fournit qu'un ensemble de fonctions Common Lisp et nécessite donc de programmer avec du code et non des fonctions et des objets visuels.

B

Description détaillée de GeLisp

Nous décrivons ici en détail la manière dont l'interface GeLisp a été créée et comment elle peut être utilisée. Nous supposons ici que le lecteur connaît Gecode, ou en tout cas les classes permettant de définir les différentes variables de décision.

B.1 Génération de l'interface

Dans cette section, nous expliquons premièrement l'architecture générale de l'interface. Ensuite, nous expliquons de manière globale comment l'interface a été réalisée.

B.1.1 Architecture de GeLisp

Dans cette sous-section, nous décrivons donc l'architecture générale de GeLisp.

Avant toute chose, nous avons créé une seule classe C++ destinée à permettre la modélisation de tous les problèmes que l'on désire à partir de LispWorks (un graphe UML simplifié est donné à la figure B.1). En effet, nous l'avons vu dans la section 1.2, si l'on désire modéliser un nouveau problème à l'aide de Gecode, il faut créer une classe qui étend la classe *Space*. Or, à partir de LispWorks, nous ne pouvons évidemment pas implémenter une telle classe, la compiler et ensuite l'instancier (ou en tout cas, si cela est possible, l'approche présentée ici paraît bien plus simple). La classe en question est la classe *GLSpace*.

Afin de permettre une telle généralité au niveau de la modélisation, la manière dont nous¹ permettons à un utilisateur de modéliser son problème diffère de la manière dont il doit le faire normalement dans Gecode. Nous expliquons en quoi dans ce qui suit.

Tout d'abord, la classe possède pour variables d'instance 4 objets de type *vector* :

1. Nous nous sommes inspirés à ce niveau de l'architecture que Mauricio Toro Bermudez et Camilo Rueda avaient utilisé pour leur version de GeLisp.

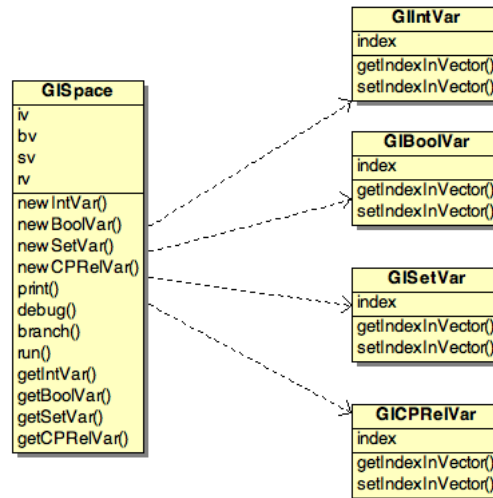


FIGURE B.1 – Graphe UML de la classe *GISpace* et de ses dépendances. Nous ne mettons ici en avant que les parties importantes des différentes classes.

- **iv** : vecteur destiné à contenir des instances de la classe *GIIntVar*².
- **bv** : vecteur destiné à contenir des instances de la classe *GIBoolVar*².
- **sv** : vecteur destiné à contenir des instances de la classe *GISetVar*².
- **rv** : vecteur destiné à contenir des instances de la classe *GICPRelVar*².

Les vecteurs n'ayant pas une taille fixée a priori, ceci permet à l'utilisateur de modéliser n'importe quel problème, quel que soit le nombre de variables requis. Il n'est en effet pas possible d'ajouter une variable au modèle sans créer une variable d'instance dans la classe lorsqu'on utilise Gecode. Le moyen le plus simple de pallier à ce problème était donc de définir un vecteur de variables dont la taille n'était pas connue a priori.

Ensuite, la classe *GISpace* possède différentes méthodes permettant d'ajouter des variables à ces vecteurs. Ces méthodes prennent exactement les mêmes paramètres que ceux pris par les constructeurs des variables de décision de Gecode, hormis l'instance de la classe *Space*. En effet, puisque ce sont des méthodes de la classe *GISpace* qui étend la classe *Space*, ce paramètre est connu dans l'environnement dans lequel on est grâce à l'identificateur *this*. Ces différentes méthodes ont donc pour rôle d'appeler le constructeur relatif et de placer la référence de l'objet dans le vecteur correspondant au type de variable construit.

Le constructeur de cette classe possède pour seul rôle d'initialiser les 4 vecteurs avec une taille nulle. Le constructeur de copie, après avoir cloné l'instance de la classe *GISpace*, se chargera de mettre à jour toutes les variables contenues dans chacun des 4 vecteurs, sans que l'utilisateur

2. Ces classes seront décrites un peu plus loin dans cette sous-section. Pour le moment, le lecteur peut simplement supposer qu'il s'agit respectivement des classes *IntVar*, *BoolVar*, *SetVar* et *CPRelVar*.

ait à s'en soucier. La méthode *copy* ne fait qu'appeler le constructeur de copie, comme c'est le cas normalement dans Gecode.

Une méthode *branch* a également été implémentée, afin de permettre d'utiliser des branchements par défaut sur l'ensemble des variables. Ceci permet moins de possibilités au niveau des heuristiques de recherche, mais aussi d'assurer que toutes les variables seront assignées à la fin de la recherche³. Cette méthode a surtout été utile lors du développement de l'interface. Il est cependant bien entendu également possible de brancher chaque variable indépendamment selon ses besoins.

La méthode permettant de lancer la recherche est nommée *run*. Elle utilise pour le moment uniquement le moteur de recherche en profondeur. Elle pourrait cependant être aisément mise à jour afin de rendre le moteur de recherche paramétrable. Le rôle de cette méthode est de remplir un vecteur d'espace-solution en fonction du nombre demandé par l'utilisateur. Gecode fonctionnant par copie, toutes les solutions trouvées par le système sont des instances de la classe *GlSpace*. Ceci permet donc de récupérer les différentes solutions dans un vecteur de solution. Nous pouvons alors récupérer les variables de décision contenues dans les vecteurs des espace-solution, pour finalement récupérer les valeurs qui leur ont été assignées pour une solution donnée. Nous pouvons dès lors utiliser ces valeurs dans un contexte autre que celui de Gecode, dans notre cas, LispWorks ou OpenMusic.

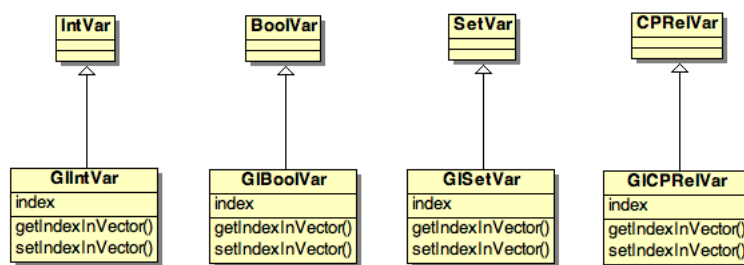


FIGURE B.2 – Graphe UML des classes *GlIntVar*, *GlBoolVar*, *GlSetVar* et *GlCPRelVar*.

Nous venons de dire qu'il était possible de récupérer les variables dans un espace-solution en la récupérant dans le vecteur correspondant. Il est nécessaire pour cela de connaître l'index de cette variable dans le vecteur. Il faut donc, au moment où l'on crée une variable de décision, retenir en mémoire à quel index du vecteur cette variable est sauvegardée. Nous aurions pu laisser ce travail à l'utilisateur, mais nous avons préféré encapsuler cette information dans l'objet variable de décision. Ceci est exactement la raison d'être des classes *GlIntVar*, *GlBoolVar*, *GlSetVar* et *GlCPRelVar*, qui étendent respectivement les classes *IntVar*, *BoolVar*, *SetVar* et *CPRelVar*, en leur ajoutant une variable d'instance destinée à contenir l'index de cette variable

3. Ceci n'est pas forcément le cas si on ne pose pas de branchements sur toutes les variables.

dans le vecteur correspondant (un graphe UML simplifié est donné à la figure B.2).

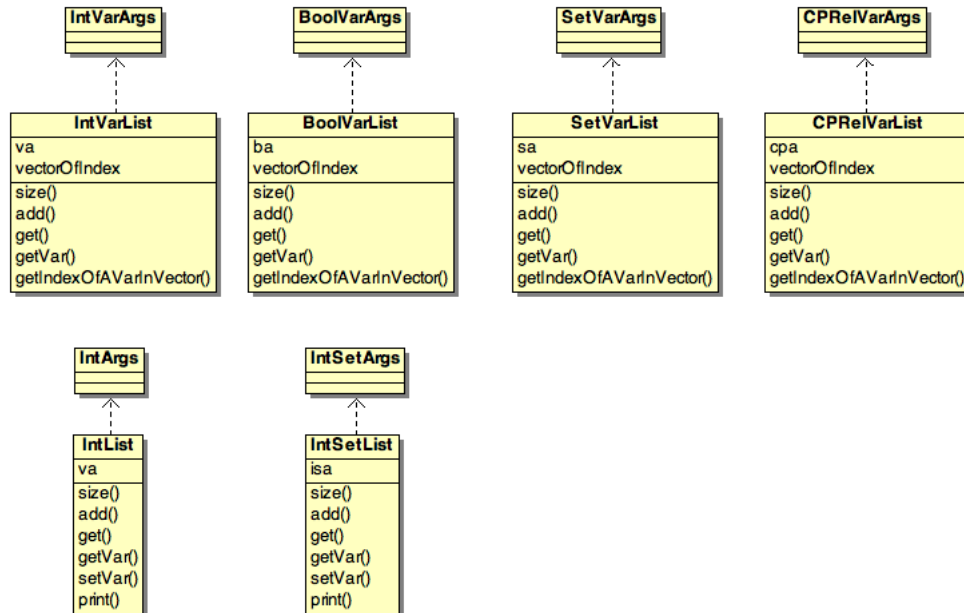


FIGURE B.3 – Graphe UML des classes `IntVarList`, `BoolVarList`, `SetVarList`, `CPreVarList`, `IntList` et `IntSetList`. L'opérateur `new` des classes `IntVarArgs`, `BoolVarArgs`, `SetVarArgs`, `CPreVarArgs`, `IntArgs` et `SetVarArgs` étant privé, la relation d'héritage a été remplacée par une relation d'utilisation.

Après avoir étendu ces classes, nous avons voulu étendre les classes `IntVarArgs`, `BoolVarArgs`, `SetVarArgs`, `CPreVarArgs` - qui sont des tableaux de variables de décision -, `IntArgs` et `IntSetArgs` - qui permettent de créer des variables de décision ou de poser des contraintes spécifiques -. Cependant, il s'est avéré que l'opérateur `new` de ces classes avaient été défini **privé** par les développeurs de Gecode. Il n'est donc possible que de construire ces objets, mais pas de créer un pointeur vers eux via l'utilisation de l'opérateur `new`. Or, SWIG utilise cet opérateur dans un des fichiers qu'il génère et nous ne pouvions modifier ce comportement.

Nous avons alors dû contourner le problème. Nous l'avons fait en encapsulant chacun de ces objets, respectivement dans les classes `IntVarList`, `BoolVarList`, `SetVarList`, `CPreVarList`, `IntList` et `IntSetList`. Les instances de ces classes ne sont donc destinées qu'à contenir l'objet qui nous intéresse. (un graphe UML simplifié est donné à la figure B.3)

Les quatre premières classes contiennent également un vecteur d'index, afin de connaître l'index des objets du tableau dans le vecteur d'objets correspondant de l'espace (instance de la classe `GlSpace`). Lorsqu'un tableau de variables de décision doit être créé, le constructeur appelle les méthodes de construction des variables de décision correspondantes de la classe `GlSpace`, et remplit la variable d'instance (le tableau) avec ces variables construites. Ce processus est illustré dans la figure B.4, dans le cas de tableaux d'entiers (le mécanisme est similaire pour les autres

types de variables).

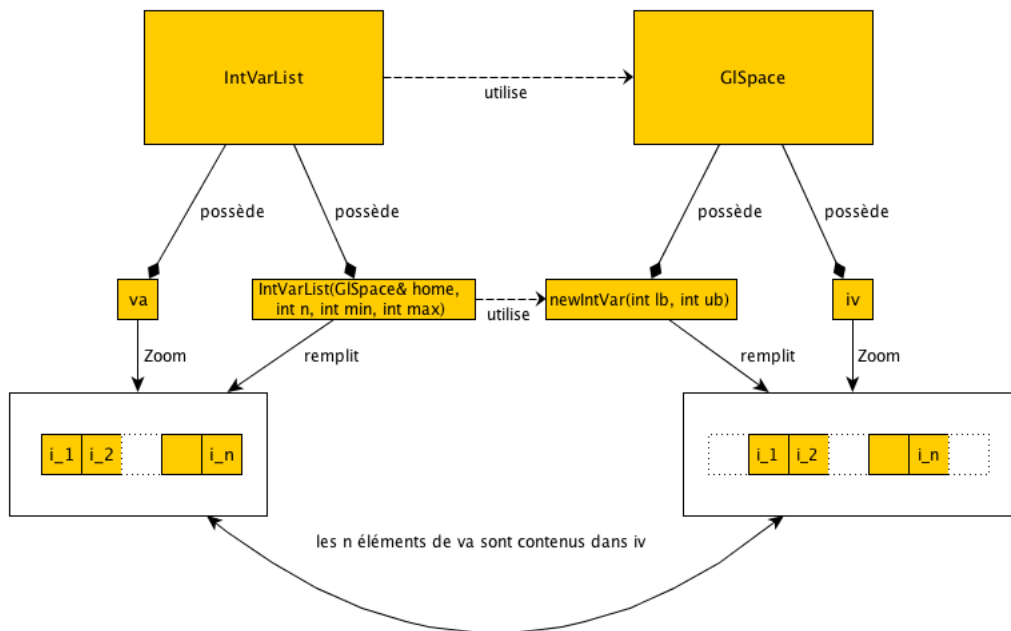


FIGURE B.4 – Ajout d'un tableau de variables de décision (ce qui est présenté ici pour les entiers vaut également pour les autres types de variables). Le constructeur de la classe *IntVarList* appelle *n* fois la méthode *newIntVar* de la classe *GISpace*. A chaque itération, la nouvelle variable de décision est donc bien placée dans le vecteur de la classe *GISpace* et dans la variable d'instance *va* de la classe *IntVarList*.

Nous avons tenté de créer une architecture aussi générique que possible pour cette interface. Par exemple, si l'on désire ajouter un nouveau type de variables de décision, c'est tout à fait possible : il suffit d'ajouter un vecteur à la classe *GISpace*, ainsi qu'une classe étendant la classe de ce nouveau type d'objet afin que l'objet puisse contenir son index dans le vecteur. C'est d'ailleurs ce qui a été fait pour la classe *CPRelVar* lors du développement de l'interface : seules les trois premières sortes de variables de décision avaient été rendues utilisables au départ.

Remarque importante. Nous devons mettre l'accent sur le fait qu'à partir de *LispWorks*, **seules** les classes suivantes doivent être utilisées :

- **GISpace** : afin de créer un objet de type *Space*, de créer des variables de décision et de lancer la recherche.
- **IntVarList** : afin de créer un objet de type *IntVarArgs*.
- **BoolVarList** : afin de créer un objet de type *BoolVarArgs*.
- **SetVarList** : afin de créer un objet de type *SetVarArgs*.
- **CPRelVarList** : afin de créer un objet de type *CPRelVarArgs*.

- **IntList** : afin de créer un objet de type *IntArgs*.
- **IntSetList** : afin de créer un objet de type *IntSetArgs*.

A partir de LispWorks, il ne faut **en aucun cas** créer d'instances des classes *GIIntVar*, *GIBoolVar*, *GISetVar* et *GLCPRelVar*, en utilisant leur constructeur. Celles-ci ne seront alors en effet présentes dans aucun des vecteurs de l'espace et ne seront donc pas mises à jour lors d'une recherche.

Normalement, une bonne utilisation du paradigme de la programmation orientée objet aurait été de définir ces classes privées.

B.1.2 Implémentation de GeLisp à l'aide de SWIG

Dans cette sous-section, nous expliquons globalement comment nous avons pu utiliser SWIG afin de rendre cette interface utilisable à partir de LispWorks.

Fonctionnement général de SWIG

Concrètement, SWIG n'a pour rôle que de générer deux fichiers principaux : *gelispCFFI_wrap.cxx* et *gelisp.lisp*. D'autres fichiers sont également générés mais il n'y a pas grand intérêt à les décrire ici.

Le fichier *gelispCFFI_wrap.cxx* contient un ensemble de fonctions C++ générées automatiquement par SWIG. Chacune de ces fonctions a pour rôle de réaliser ce qu'un constructeur, un destructeur, une méthode ou une fonction du code à interfacier exécute. Concrètement, elles les appellent et retournent leur résultat. Ainsi, tout ce qui était possible d'accomplir à l'aide du code à interfacier est réalisable grâce à l'ensemble des fonctions contenues dans ce fichier.

Le fichier *gelisp.lisp* définit quant à lui un ensemble de **fonctions étrangères** pour le langage Common Lisp. A chacune des fonctions étrangères de ce fichier est associée une et une seule fonction contenue dans le fichier *gelispCFFI_wrap.cxx*. Appeler une fonction étrangère définie dans ce fichier à partir du langage Common Lisp revient à appeler la fonction C++ associée. Puisque les fonctions du fichier *gelispCFFI_wrap.cxx* permettent de réaliser en C++ tout ce que le code à interfacier accomplit, les fonctions étrangères le permettent également. Autrement dit, le fichier *gelisp* contient toutes les fonctions Common Lisp que l'utilisateur peut employer afin d'utiliser Gecode à partir de LispWorks.

Compilation de l'interface

Nous ne donnons pas ici les détails de la manière dont la compilation a été réalisée. Cependant, nous attirons l'attention sur le fait que cette interface a été compilée pour un adressage en 32 bits. En effet, bien que Gecode permette de travailler en 64 bits, la version 5 de LispWorks ne le permet pas. Nous avons donc fait le choix de nous restreindre à 32 bits pour notre interface et pour la compilation de Gecode. Aujourd'hui, LispWorks permet de travailler en 64 bits depuis la

version 6, mais uniquement dans sa version payante. Nous sommes donc restés dans un adressage de 32 bits.

De plus, nous devons préciser que le code contenu dans le fichier *gelispCFFI_wrap.cxx* doit avant tout être compilé dans une librairie afin d'être rendu utilisable par Common Lisp via des fonctions étrangères. Dans notre cas, cette librairie est contenue dans le fichier *libgelisp.so*. Précisons que c'est bien le compilateur *gcc* qui compile cette librairie et non SWIG.

Description des fichiers implémentant l'interface GeLisp

Nous décrivons ici dans les grandes lignes les différents fichiers implémentant l'interface GeLisp.

Tout d'abord, les fichiers **gelisp.hpp**⁴ et **gelisp.cpp**⁵ implémentent l'architecture décrite dans la sous-section B.1.1.

Le premier définit la classe *GlSpace* et déclare les en-têtes des méthodes et constructeurs de cette classe. Il implémente également les autres classes. Le second implémente les méthodes et constructeurs de la classe *GlSpace*.

L'architecture générale de GeLisp ayant déjà été décrite, il n'y a plus grand intérêt à décrire ces deux fichiers, qui ne font qu'implémenter ce qui a été expliqué. Il y a cependant deux points sur lesquels nous voudrions attirer l'attention :

- Premièrement, dans le constructeur de la classe *GlSpace*, nous redirigeons la sortie standard et la sortie d'erreurs standard, respectivement vers les fichiers **/tmp/mystdout.txt** et **/tmp/mystderr.txt**. C'est en effet le seul moyen nous permettant d'obtenir les messages écrits sur ces sorties, SWIG ne nous permettant pas de les rediriger vers le Listener de LispWorks (ou du moins, c' est le seul moyen fonctionnel que nous avons réussi à créer).
- Ensuite, la méthode *runEngine* (permettant de résoudre le problème) ne prend pas en paramètre un objet de type *SizeOptions* ni la classe du problème en argument template comme dans Gecode. La raison pour le premier est que dans Gecode, le constructeur de la classe définissant le modèle construit dans son corps les variables d'instances. Pour ce faire, le constructeur doit connaître la taille du problème abordé (par exemple, dans le problème des *n*-reines, la taille du plateau est nécessaire afin de connaître la longueur du tableau *IntVarArray*). Dans GeLisp, le constructeur de la classe *GlSpace* n'a pas ce rôle, et c'est l'utilisateur qui doit appeler des méthodes de cette classe afin de créer des variables et les ajouter à leur vecteur respectif.

Pour le second pour lequel la méthode *runEngine* ne prend pas la classe en paramètre template est que celui-ci est toujours la classe *GlSpace*, puisque cette classe permet de résoudre tous les problèmes à partir de LispWorks. Le seul argument commun par rapport à Gecode est le moteur de recherche, que l'on veut laisser paramétrable. La méthode *run* appelle la

4. code source à l'annexe C.

5. code source à l'annexe C.

méthode *runEngine* en lui passant la recherche en profondeur comme moteur de recherche.

Ensuite, nous avons utilisé un autre fichier, nommé **gecheader.hpp**, destiné à contenir l'ensemble des classes et des en-têtes des fonctions faisant partie des fichiers d'en-tête de Gecode, et que nous voulions mettre à disposition de l'utilisateur Common Lisp. Donc, si l'on désire ajouter une fonction ou une classe de Gecode à GeLisp, il suffit d'ajouter leur déclaration à ce fichier. À l'aide du fichier d'interfaçage que nous décrivons juste après, SWIG fournira alors des fonctions Common Lisp permettant de les utiliser.

gelisp.i : le fichier d'interfaçage

Nous décrivons à présent le fichier principal requis par SWIG afin de générer l'interface entre du code C++ et du code Common Lisp. Ce fichier se nomme **gelisp.i**. Son contenu est en soi assez simple et se divise en plusieurs parties⁶ :

- **Définition des fichiers d'en-tête devant être inclus dans le fichier *gelispCFFI.wrap.cxx*.** Les instructions d'inclusion à insérer doivent simplement être encadrées par les symboles `%{` et `%}`.
- **Définition de la gestion des exceptions dans le fichier *gelispCFFI.wrap.cxx*.** À l'aide du mot réservé `%exception`, il est possible de définir comment les erreurs doivent être gérées. Dans notre cas, nous avons juste exprimé le fait que quel que soit le type d'erreur, il faut l'attraper et l'imprimer sur la sortie d'erreurs.
- **Explicitation des fichiers dont on veut rendre les fonctionnalités disponibles à l'utilisateur Common Lisp.** Ceci est fait à l'aide du mot réservé `%include`. Dans notre cas :

```
%include <gelisp.hpp>
%include <gecheader.hpp>
```

Toutes les fonctions, méthodes, constructeurs, destructeurs, énumérations, ... de ces deux fichiers seront dès lors disponibles à l'utilisateur Common Lisp.

- **Don de nouveaux noms.** Grâce au mot réservé `%rename`, nous avons décrit à SWIG quels étaient les méthodes, fonctions et constructeurs, ... qu'il devait renommer lors de l'interfaçage et comment il devait le faire. Ceci est nécessaire car par défaut, SWIG récupère simplement le nom d'une méthode, fonction, ... dans le nom des fonctions générées dans le fichier *gelispCFFI.wrap.cxx* ainsi que pour les fonctions étrangères générées dans le fichier *gelisp.lisp*. Or, en C++ il est possible de faire de la surcharge de fonctions, mais pas en Common Lisp. En effet, Common Lisp est un langage typé dynamiquement, il n'est dès lors

6. d'autres fonctionnalités sont fournies par SWIG, mais nous ne les avons pas toutes utilisées.

pas possible de différencier une fonction d'une autre en fonction du type des paramètres et/ou de leur nombre. Seul le nom de la fonction le permet. Il a donc fallu renommer « à la main » chacun des constructeurs, fonctions et méthodes surchargés. Afin d'aider au mieux l'utilisateur Common Lisp, nous avons suivi un modèle autant que possible :

Une fonction nommée *nom* et prenant en paramètre deux instances de la classe *classeA* et une instance de la classe *classeB* sera renommée comme suit :

```
%rename(nom_on2classeAAndclasseB) nom(Space& home, classeA param1, classeA
param2, classeB param3);
```

De plus, si un paramètre optionnel est possible, le suffixe *_default* est ajouté à la fonction utilisant la valeur par défaut du paramètre optionnel et *_option* pour celui demandant de spécifier le spécifier (si plusieurs paramètres optionnels sont possibles, les suffixes *_1option* et *_2option* sont utilisés).

Dans certains cas cependant, nous n'avons pas suivi ce modèle, lorsque nous avons la possibilité de donner un nom explicitant un peu le comportement d'une fonction ou le rôle des paramètres. Ceci est une bonne pratique pour ne pas avoir à vérifier dans la documentation de Gecode et dans le fichier `gelistp.i` quelle fonction on désire utiliser.

B.2 Utilisation de l'interface

Dans cette section, nous décrivons comment il est possible d'utiliser l'interface GeLisp. Nous expliquons tout d'abord quelles sont les dépendances nécessaires à l'utilisation de GeLisp et nous expliquons ensuite son utilisation. Finalement, nous donnons un bref récapitulatif sur la manière d'utiliser la librairie Gecode à partir d'OpenMusic.

B.2.1 Dépendances nécessaires

La librairie CFFI possède elle-même des dépendances :

- Babel [15] : une librairie d'encodage/décodage de caractères. Elle fournit des convertisseurs entre chaîne de caractères et des vecteurs d'octets.
- Alexandria [16] : une collection d'utilitaires portables du domaine public pour le langage Common Lisp.
- trivial-features [17] : un niveau de portabilité qui assure des caractéristiques consistantes entre différentes implémentations de Common Lisp.
- ASDF-Install [18] : programme permettant de télécharger et d'installer des paquets Common Lisp.⁷

7. Nous devons remarquer ici qu'ASDF-Install est obsolète et qu'une alternative était possible et proposée par la communauté de CFFI : `clbuild`. ASDF-Install a été choisi arbitrairement mais se voit toujours être fonctionnel. Le but de ce travail étant avant tout de travailler sur la programmation par contraintes plutôt que sur la technologie Common Lisp, ce choix ne pose pas de réels problèmes.

Fort heureusement, un gestionnaire de bibliothèques nommé *quicklisp* [19] permet de charger la bibliothèque CFFI ainsi que toutes ses dépendances automatiquement (dont ASDF-Install). Un avantage de *quicklisp* est qu'il est portable sur différentes architectures.

B.2.2 Emploi effectif de l'interface

Nous pouvons à présent expliquer la manière d'utiliser l'interface. Nous décrirons tout d'abord dans cette sous-section comment utiliser GeLisp à partir de LispWorks. Concrètement, ceci permet de travailler dans le paradigme de la programmation par contraintes à partir de Common Lisp, en possédant tous les outils de Gecode. Après cela, nous donnerons la résolution du problème des n reines à l'aide de GeLisp, à titre de tutoriel.

Emploi dans LispWorks

La seule chose à faire afin de pouvoir utiliser Gecode dans LispWorks est de charger les dépendances et l'interface afin d'avoir accès à toutes les fonctions interfacées par SWIG. Il faut pour cela charger le fichier **setup.lisp** en évaluant l'instruction suivante :

```
(load "chemin/setup.lisp")
```

Une fois ceci fait, l'utilisateur a accès à toutes les fonctions Common Lisp recouvrant l'ensemble des fonctionnalités de Gecode. Nous devons remarquer ici que l'interface donne accès à la même expressivité, bien que toutes les commodités ne soient pas fournies (un certain nombre de sucres syntaxiques n'ont pas été interfacés). Donc, bien que l'ensemble des fonctionnalités de Gecode ne sont pas mises à disposition, nous fournissons le même potentiel de modélisation que Gecode. La manière de formuler ces problèmes sera juste moins évidente dans certains cas. Nous devons également remarquer que, mises à part les relations, aucun des modules supplémentaires de Gecode n'a été interfacé. Par exemple, les contraintes sur les graphes n'ont pas été interfacées (mais rappelons-le, elles pourraient l'être sans trop de problèmes).

Comme pour Gecode, nous ne décrirons pas l'ensemble de GeLisp. Afin de connaître l'ensemble des fonctions de GeLisp, il suffit de consulter le fichier `gelistp.lisp`. Nous expliquons uniquement dans ce qui suit la manière dont nous avons pu résoudre le problème des n reines à l'aide de GeLisp.

Résolution du problème des n reines avec GeLisp

Nous décrivons ici notre résolution du problème des n reines avec GeLisp. Le code source est donné dans l'annexe C.

Avant toute chose, nous devons créer un **espace**, destiné à contenir l'ensemble des variables de décision, les propagateurs et les brancheurs :

```
(setq sp (new-GISpace))
```

La fonction *new_GlSpace* est le constructeur de la classe *GlSpace*. Nous enregistrons la référence de cet objet dans une variable *sp*. Ensuite, nous créons un objet de la classe *IntVarList*, qui contient un objet de la classe *IntVarArgs* :

```
(setq queen (new_IntVarList_minmax sp n 0 (- n 1)))
```

La fonction *new_IntVarList_minmax* équivaut au constructeur *IntVarList(GlSpace& home, int n, int min, int max)*. Une variable d'instance de l'objet créé est une instance de la classe *IntVarArgs*, dont nous avons besoin pour modéliser le problème. Sa taille est *n* et le domaine de chacune des variables contenues dans le tableau est $[0, \dots, n-1]$. Après l'appel de la fonction, les variables de décision créées sont rattachées à l'espace *sp*. De manière interne, le vecteur d'objets *IntVar* de l'espace *sp* est rempli de *n* nouvelles variables de décision.

Ensuite, nous avons besoin de créer deux instances de la classe *IntArgs*. Nous le faisons en créant deux objets de la classe *IntList*, qui vont contenir ces objets en tant que variables d'instance. Ceci est fait avec les deux instructions suivantes :

```
(setq intlist1 (new_IntList_sizeAndStartAndInc n 0 1))
(setq intlist2 (new_IntList_sizeAndStartAndInc n 0 -1))
```

La fonction *new_IntList_sizeAndStartAndInc* est la fonction servant d'interface au constructeur *IntList(int n, int start, int inc)*. Ce constructeur initialise sa variable d'instance (un objet de la classe *IntArgs*) avec les mêmes paramètres qui lui sont passés. Intuitivement, les deux tableaux d'entiers créés sont les suivants :

$$[0, 1, \dots, n-1]$$

et

$$[0, -1, \dots, 1-n]$$

Nous devons à présent imposer les contraintes désirées. La première impose que chaque reine soit sur une colonne (symétriquement ligne) différente :

```
(distinct_default sp (IntVarList_get queen))
```

Rappelons tout de suite que l'instance de la classe *IntVarArgs* est une variable d'instance de *queen* (défini plus haut). Puisque c'est avec elle que nous devons travailler, nous devons avant tout la récupérer à l'aide de la fonction *IntVarList_get*. Ensuite, nous utilisons la fonction *distinct_default* qui permet d'imposer la contrainte implémentée en C++ par le propagateur *distinct(Space& home, const IntVarArgs& x)*.

De manière similaire, nous imposons qu'aucune reine n'est sur une diagonale commune :

```
(distinct_constant_added_to_var_default sp (IntList_get intlist1) (IntVarList_get
  queen))
(distinct_constant_added_to_var_default sp (IntList_get intlist2) (IntVarList_get
  queen))
```

Maintenant que le problème est modélisé, nous devons définir quels brancheurs nous voulons utiliser durant la recherche. Pour cet exemple, nous avons utilisé les brancheurs par défaut de GeLisp :

```
(G1Space_branch sp)
```

Avant de lancer la recherche, nous créons un vecteur d’espaces, destiné à contenir le nombre désiré de solutions de l’utilisateur :

```
(setq v (G1Space_createG1SpaceVector sp))
```

Finalement, nous pouvons démarrer la recherche. Ici, nous ne demandons qu’une seule solution pour le problème :

```
(G1Space_run sp 1 v)
```

L’espace solution sera enregistré dans le vecteur v , et nous pourrions récupérer les valeurs assignées aux variables de décision pour un travail ultérieur en Common Lisp. Les solutions sont toujours imprimées dans le listener de LispWorks après exécution. Voici celle que nous avons obtenue pour ce problème, où $n = 10$:

```
"Integer variables:
   iv [0]: 0
   iv [1]: 2
   iv [2]: 5
   iv [3]: 8
   iv [4]: 6
   iv [5]: 9
   iv [6]: 3
   iv [7]: 1
   iv [8]: 4
   iv [9]: 7
Bool variables:
Set variables:
CPRel variables:
```

Pour conclure la manière dont il a été rendu possible d’utiliser Gecode à partir de LispWorks, nous donnons dans la figure B.5 un récapitulatif global.

B.2.3 Utilisation de Gecode à partir d’OpenMusic : Récapitulatif

- Premièrement, il faut compiler l’interface GeLisp⁸.
- Après cela, il faut placer un fichier⁹ destiné à charger la librairie étrangère ainsi que toutes

8. Précisons que cette interface n’a pas été construite de manière portable. Il n’est donc pas possible au lecteur de l’installer sur sa machine. Nous fournissons cependant les différents fichiers l’implémentant dans l’annexe D.

9. Dans notre cas, *setup.lisp*.

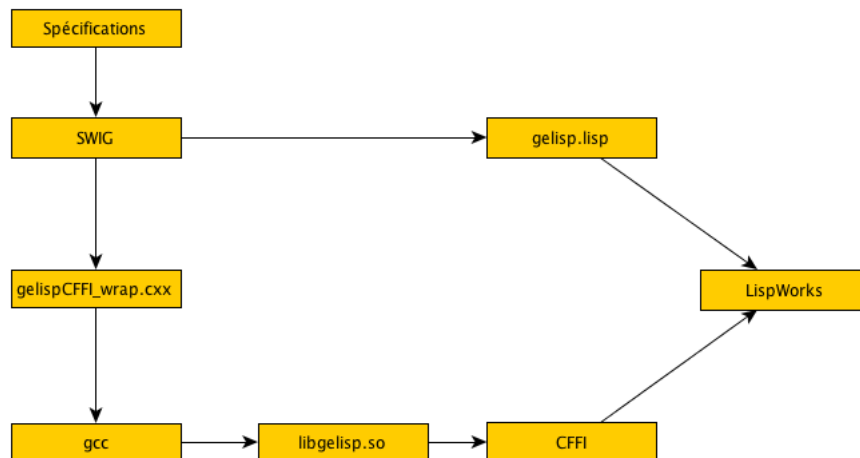


FIGURE B.5 – Récapitulatif général de l'intégration de Gecode dans LispWorks.

les fonctions étrangères, dans le dossier *patches* d'OpenMusic, prévu à cet effet. Rappelons que CFFI ne doit pas être chargé, car il est chargé par défaut par OpenMusic.

- Dans OpenMusic, quelle que soit la manière dont on utilise le langage Common Lisp, il faut appeler les fonctions étrangères en utilisant le package *cl-user* : :

C

Codes source

Problème de la série tous-intervalles

Gecode

```
#include <gecode/driver.hh>
#include <gecode/int.hh>
#include <gecode/minimodel.hh>

using namespace Gecode;

class AllIntervalsSerie : public Script {
public:
  IntVarArray pitchClasses;
  IntVarArray intervals;
  IntVarArray diffpitch;
  IntVarArray diffpitchshifted;

  AllIntervalsSerie(const Options& opt) : pitchClasses(*this,12,0,11), intervals(*
    this,11,0,11),
    diffpitchshifted(*this,11, 1, 23)
  {
    IntVar divisor(*this, 12, 12); //only used to express constraints

    for(int i = 0 ; i < 11 ; i++)
    {
      //establish needed coefficients
      int coeffTable[12] = {0};
      coeffTable[i]=-1;
      coeffTable[i+1]=1;

      //diffpitchshifted(i) = pitchClasses(i+1) - pitchClasses(i) + 12
      //gecode doesn't manage negative modulo dividend
      rel(*this, pitchClasses[i+1] - pitchClasses[i] + 12 ==
        diffpitchshifted[i]);

      //intervals(i) = (pitchClasses(i+1) - pitchClasses(i) + 12) mod 12
    }
  }
};
```

```

        mod(*this, diffpitchshifted[i], divisor, intervals[i]);
    }

    //dodecaphonic serie
    distinct(*this, pitchClasses);

    //by definition
    distinct(*this, intervals);

    //symmetry break (avoid transposition) : pitchClasses(0) = 0
    dom(*this, pitchClasses[0], 0);

    //redundant constraint (allows to prune more) ; pitchClasses(11) = 6
    dom(*this, pitchClasses[11], 6);

    //we only need the values of those variables
    branch(*this, pitchClasses, INT_VAR_SIZE_MIN, INT_VAL_MIN);
    branch(*this, intervals, INT_VAR_SIZE_MIN, INT_VAL_MIN);
}
/// Print solution
virtual void
print(std::ostream& os) const {
    os << "\t" << pitchClasses << std::endl;
    os << "\t" << intervals << std::endl;
}

/// Constructor for cloning \a s
AllIntervalsSerie(bool share, AllIntervalsSerie& s) : Script(share, s) {
    pitchClasses.update(*this, share, s.pitchClasses);
    intervals.update(*this, share, s.intervals);
    diffpitchshifted.update(*this, share, s.diffpitchshifted);
}
/// Copy during cloning
virtual Space*
copy(bool share) {
    return new AllIntervalsSerie(share, *this);
}
};

int
main(int argc, char* argv[]) {
    Options opt("All intervals serie");
    Script::run<AllIntervalsSerie, DFS, Options>(opt);
    return 0;
}

```

GeLisp

```
;load GeLisp
```

```

(load "/Users/saschavancauwelaert/Documents/EPL/Memoire/gelisp/setup.lisp")

(defun allint ()

  ; GlSpace creation
  (setq sp (new_GlSpace))

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; decision variables ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  (setq pitchClasses (new_IntVarList_minmax sp 12 0 11))
  (setq intervals (new_IntVarList_minmax sp 11 1 11))

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; auxiliary variables ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  ; only used for mod divisor
  (setq divisor (GlSpace_newIntVar_minmax sp 12 12))
  ; only used to express the mod constraint. It will contains the true pitch
  ; differences.
  (setq diffpitch (new_IntVarList_minmax sp 11 -11 11))
  ; used because domain must be shifted because Gecode doesn't manage modulo with
  ; negative dividend
  (setq diffpitchshifted (new_IntVarList_minmax sp 11 1 23))

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; constraints ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  ; coeff used to shift pitch differences
  (setq coeffListForShift (new_IntList_sizeAndDefaultValue 2 1))

  ; inversionalEquivalentInterval
  (loop for x from 0 to 10 do

    ; establish needed coefficients
    (setq coeffList (new_IntList_sizeAndDefaultValue 12 0))
    (IntList_setVar coeffList x -1)
    (IntList_setVar coeffList (+ x 1) 1)

    ; diffpitch(i) = pitchClasses(i+1) - pitchClasses(i)
    (linear_onIntArgsAndIVAAndIntVar_default sp (IntList_get coeffList) (
      IntVarList_get pitchClasses) :IRT_EQ (IntVarList_getVar diffpitch x)
    )

    ; diffpitchshifted(i) = diffpitch(i) + 12
    (setq tempVarList (new_IntVarList))
    (IntVarList_add tempVarList (IntVarList_getVar diffpitch x))
    (IntVarList_add tempVarList divisor)
    (linear_onIntArgsAndIVAAndIntVar_default sp (IntList_get
      coeffListForShift) (IntVarList_get tempVarList) :IRT_EQ (
      IntVarList_getVar diffpitchshifted x))

    ; intervals(i) = diffpitchshifted(i) mod 12
    (mod_default sp (IntVarList_getVar diffpitchshifted x) divisor (
      IntVarList_getVar intervals x))
  )
)

```

```

;additional constraint to have a symmetric serie : intervals(i) +
;intervals(11 - 1 - i) = 12
(setq tempVarList (new_IntVarList))
(IntVarList_add tempVarList (IntVarList_getVar intervals x))
(IntVarList_add tempVarList (IntVarList_getVar intervals (- 10 x)))
(linear_onIntArgsAndIVAAndIntVar_default sp (IntList_get
coeffListForShift) (IntVarList_get tempVarList) :IRT.EQ divisor)

)

;distinctPitchClasses
(distinct_default sp (IntVarList_get pitchClasses))

;distinctIntervals
(distinct_default sp (IntVarList_get intervals))

;symmetry break (avoid transposition) : pitchClasses(0) = 0
(dom_onIntVar_default sp (IntVarList_getVar pitchClasses 0) 0)

;redundant constraint (allows to prune more) : pitchClasses(11) = 6
(dom_onIntVar_default sp (IntVarList_getVar pitchClasses 11) 6)

;;;;;;;;;;;;;branching and search
;;;;;;;;;;;;;

(GlSpace_branch sp)
(setq solVec (GlSpace_createGlSpaceVector sp))
(GlSpace_run sp 1 solVec)

)

```

Coeur de GeLisp

gelisp.hpp

```

#ifndef __GELISP_HPP__
#define __GELISP_HPP__

#include <iostream>
#include <vector>
#include <gecode/kernel.hh>
#include <gecode/int.hh>
#include <gecode/set.hh>
#include <cprel/cprel.hh>

namespace GeLisp {

class GlIntVar;
class GlBoolVar;

```

```

class G1SetVar;
class G1CPRelVar;

class G1Space : public Gecode::Space {
  G1Space(bool share , G1Space&);
public:
  /// The integer variables
  std::vector<G1IntVar> iv;
  /// The Boolean variables
  std::vector<G1BoolVar> bv;
  /// The set variables
  std::vector<G1SetVar> sv;
  /// The relation variables
  //std::vector<G1CPRelVar> rv;
  std::vector<G1CPRelVar> rv;
  /// Construct an empty space
  G1Space(void);
  /// Destructor
  ~G1Space(void);

  /// Create a new integer variable with domain [lb,ub]
  G1IntVar& newIntVar(int lb , int ub);
  /// Create a new integer variable from IntSet is
  G1IntVar& newIntVar(const Gecode::IntSet &d);

  /// Create a new boolean variable with domain [lb,ub]
  G1BoolVar& newBoolVar(int lb , int ub);

  /// Create a new set variable from home
  G1SetVar& newSetVar();

  /// Create a new set variable with bounds and cardinality
  G1SetVar& newSetVar(int glbMin,int glbMax,int lubMin,int lubMax ,
                    unsigned int cardMin = 0, unsigned int cardMax = Gecode::
                    Set::Limits::card);

  /// Create a new set variable with bounds(lower bound is an IntSet) and
  cardinality
  G1SetVar& newSetVar(const Gecode::IntSet& glbD,int lubMin,int lubMax ,
                    unsigned int cardMin = 0,
                    unsigned int cardMax = Gecode::Set::Limits::card);

  /// Create a new set variable with bounds(upper bound is an IntSet) and
  cardinality
  G1SetVar&
  newSetVar(int glbMin,int glbMax,const Gecode::IntSet& lubD ,
          unsigned int cardMin = 0,
          unsigned int cardMax = Gecode::Set::Limits::card);

  /// Create a new set variable with bounds(lower and upper bound is an IntSet)
  and cardinality
  G1SetVar&

```

```

newSetVar(const Gecode::IntSet& glbD, const Gecode::IntSet& lubD,
          unsigned int cardMin = 0,
          unsigned int cardMax = Gecode::Set::Limits::card);

    GICPRelVar&
newCPRelVar(const MPG::CPRel::GRelation& l, const MPG::CPRel::GRelation& u);

    /// Perform propagation
    const char* status(void);
    /// Print space information
    void print(std::ostream& os) const;
    /// Debug
    const char* debug(void) const;
    /// Post the branchings
    void branch(void);
    /// Run the search for nbSolutions, returns the number of solutions. if
    nbSolutions = 0, search on all the space.
    template<template<class> class Engine>
    unsigned int runEngine(std::ostream& os, int nbSolutions, std::vector<GISpace
        *>& solutionsVector);
    /// Returns the number of solutions to the problem
    const char* run(int nbSolutions, std::vector<GISpace*>& solutionsVector);

    /// Copy function
    virtual Gecode::Space* copy(bool share);

    ///get the variable reference from the index of iv
    GIIntVar& getIntVar(int indexInVector);

    ///get the variable reference from the index of bv
    GIBoolVar& getBoolVar(int indexInVector);

    ///get the variable reference from the index of sv
    GISetVar& getSetVar(int indexInVector);

    ///get the variable reference from the index of rv
    GICPRelVar& getCPRelVar(int indexInVector);

    ///create a vector of GISpace and return a reference on it
    std::vector<GISpace*> createGISpaceVector(void);

    ///gives access to an element of a vector of GISpace at a given index
    GISpace* getGISpaceFromVector(std::vector<GISpace*>& vec, int index);
};

///classe utilisee a la place de IntVar car on a besoin de connaitre l'indice de
l'objet dans le vecteur iv
class GIIntVar: public Gecode::IntVar
{

```

```

    int index; //index of the variable in iv vector (used to get the values
        solution after search)
public:
    G1IntVar(void):Gecode::IntVar()
    {
        index = -1;
    }

    G1IntVar(G1Space& home, int lb, int ub):Gecode::IntVar(home, lb, ub)
    {
        index = -1;
    }
    G1IntVar(G1Space& home, const Gecode::IntSet &d):Gecode::IntVar(home, d)
    {
        index = -1;
    }
    int getIndexInVector(void)
    {
        return index;
    }
    void setIndexInVector(int newIndex)
    {
        index = newIndex;
    }
};

//classe utilisee a la place de BoolVar car on a besoin de connaitre l'indice de
    l'objet dans le vecteur bv
class G1BoolVar: public Gecode::BoolVar
{
    int index; //index of the variable in bv vector (used to get the values
        solution after search)
public:
    G1BoolVar(void):Gecode::BoolVar()
    {
        index = -1;
    }

    G1BoolVar(G1Space& home, int lb, int ub):Gecode::BoolVar(home, lb, ub)
    {
        index = -1;
    }

    int getIndexInVector(void)
    {
        return index;
    }
    void setIndexInVector(int newIndex)
    {
        index = newIndex;
    }
}

```

```

};

//classe utilisee a la place de SetVar car on a besoin de connaitre l'indice de
//l'objet dans le vecteur sv
class G1SetVar: public Gecode::SetVar
{
    int index; //index of the variable in sv vector (used to get the values
               //solution after search)
public:

    G1SetVar(void):Gecode::SetVar()
    {
        index = -1;
    }

    G1SetVar(G1Space& home):Gecode::SetVar(home)
    {
        index = -1;
    }

    G1SetVar(G1Space& home, int glbMin,int glbMax,int lubMin,int lubMax, unsigned
             int cardMin = 0, unsigned int cardMax = Gecode::Set::Limits::card):Gecode
             ::SetVar(home, glbMin, glbMax, lubMin, lubMax, cardMin, cardMax)
    {
        index = -1;
    }

    G1SetVar(G1Space& home, const Gecode::IntSet& glbD,int lubMin,int lubMax,
             unsigned int cardMin = 0,
             unsigned int cardMax = Gecode::Set::Limits::card):Gecode::SetVar(home
             , glbD, lubMin, lubMax, cardMin, cardMax)
    {
        index = -1;
    }

    G1SetVar(G1Space& home, int glbMin,int glbMax,const Gecode::IntSet& lubD,
             unsigned int cardMin = 0,
             unsigned int cardMax = Gecode::Set::Limits::card):Gecode::SetVar(home
             , glbMin, glbMax, lubD, cardMin, cardMax)
    {
        index = -1 ;
    }

    G1SetVar(G1Space& home, const Gecode::IntSet& glbD,const Gecode::IntSet& lubD,
             unsigned int cardMin = 0,
             unsigned int cardMax = Gecode::Set::Limits::card):Gecode::SetVar(home
             , glbD, lubD, cardMin, cardMax)
    {
        index = -1;
    }
}

```

```

int getIndexInVector(void)
{
    return index;
}
void setIndexInVector(int newIndex)
{
    index = newIndex;
}

};

class GICPRelVar: public MPG::CPRelVar
{
    int index;

public:
    GICPRelVar(void):MPG::CPRelVar()
    {
        index = -1;
    }

    GICPRelVar(GISpace& home, const MPG::CPRel::GRelation& l, const
        MPG::CPRel::GRelation& u):MPG::CPRelVar(home,l,u)
    {
        index = -1;
    }

    int getIndexInVector(void)
    {
        return index;
    }
    void setIndexInVector(int newIndex)
    {
        index = newIndex;
    }

};

class IntVarList {
    Gecode::IntVarArgs va;
    std::vector<int> vectorOfIndex; //will contain the indexes of the IntVar
    objects in the iv vector
public:
    IntVarList(void)
    {

    }

    IntVarList(GISpace& home, int n, int min, int max) {

```

```

//va = new Gecode::IntVarArgs(0);
GIIntVar* tempVar;
for (int i = 0; i < n; i++) {
    tempVar = &(home.newIntVar(min,max));
    va << *tempVar;
    vectorOfIndex.push_back(tempVar->getIndexInVector());
}
}
IntVarList(GISpace& home, int n, const Gecode::IntSet &d){
    for (int i = 0; i < n; i++) {
        va << home.newIntVar(d);
    }
}

unsigned int size(void) {
    return va.size();
}

void add(/*const*/ GIIntVar& x) {
    va << x;
    vectorOfIndex.push_back(x.getIndexInVector());
}

const Gecode::IntVarArgs& get(void) {
    return va;
}

const GIIntVar& getVar(int i) {
    assert(i >= 0 && i < va.size());
    GIIntVar& toReturn = ((GIIntVar&) va[i]) ;
    return toReturn;
}

//return the index in iv of an IntVar in va
const int getIndexOfAVarInVector(int i)
{
    return vectorOfIndex[i];
}
};

class BoolVarList {
    Gecode::BoolVarArgs ba;
    std::vector<int> vectorOfIndex; //will contain the indexes of the BoolVar
    objects in the bv vector
public:
    //BoolVarList(void) {}
    BoolVarList(GISpace& home, int n, int min, int max) {
        GIBoolVar* tempVar;
        for (int i = 0; i < n; i++) {
            tempVar = &(home.newBoolVar(min,max));
            ba << *tempVar;
            vectorOfIndex.push_back(tempVar->getIndexInVector());
        }
    }
}

```

```

}
unsigned int size(void) {
    return ba.size();
}
void add(GlBoolVar& x) {
    ba << x;
    vectorOfIndex.push_back(x.getIndexInVector());
}
const Gecode::BoolVarArgs& get(void) {
    return ba;
}

const GlBoolVar& getVar(int i) {
    assert(i >= 0 && i < ba.size());
    GlBoolVar& toReturn = ((GlBoolVar&) ba[i]) ;
    return toReturn;
}

//return the index in bv of an IntVar in ba
const int getIndexOfAVarInVector(int i)
{
    return vectorOfIndex[i];
}
};

class SetVarList {
    Gecode::SetVarArgs sa;
    std::vector<int> vectorOfIndex; //will contain the indexes of the SetVar
    objects in the sv vector
public:

    //CONSTRUCTORS

    SetVarList(GlSpace& home, int n)
    {
        GlSetVar* tempVar;
        for (int i = 0; i < n; i++)
        {
            tempVar = &(home.newSetVar());
            sa << *tempVar;
            vectorOfIndex.push_back(tempVar->getIndexInVector());
        }
    }

    SetVarList(GlSpace& home, int n, int glbMin, int glbMax, int lubMin, int lubMax,
        unsigned int cardMin = 0, unsigned int cardMax = Gecode::Set::
        Limits::card)
    {
        GlSetVar* tempVar;

```

```

    for (int i = 0; i < n; i++)
    {
        tempVar = &(home.newSetVar(glbMin, glbMax, lubMin, lubMax, cardMin,
            cardMax));
        sa << *tempVar;
        vectorOfIndex.push_back(tempVar->getIndexInVector());
    }
}

SetVarList(GlSpace& home, int n, const Gecode::IntSet& glbD, int lubMin, int
lubMax,
    unsigned int cardMin = 0, unsigned int cardMax = Gecode::Set::
Limits::card)
{
    GlSetVar* tempVar;
    for (int i = 0; i < n; i++)
    {
        tempVar = &(home.newSetVar(glbD, lubMin, lubMax, cardMin, cardMax));
        sa << *tempVar;
        vectorOfIndex.push_back(tempVar->getIndexInVector());
    }
}

SetVarList(GlSpace& home, int n, int glbMin, int glbMax, const Gecode::IntSet&
lubD,
    unsigned int cardMin = 0, unsigned int cardMax = Gecode::Set::
Limits::card)
{
    GlSetVar* tempVar;
    for (int i = 0; i < n; i++)
    {
        tempVar = &(home.newSetVar(glbMin, glbMax, lubD, cardMin, cardMax));
        sa << *tempVar;
        vectorOfIndex.push_back(tempVar->getIndexInVector());
    }
}

SetVarList(GlSpace& home, int n, const Gecode::IntSet& glbD, const Gecode::
IntSet& lubD,
    unsigned int cardMin = 0, unsigned int cardMax = Gecode::Set::
Limits::card)
{
    GlSetVar* tempVar;
    for (int i = 0; i < n; i++)
    {
        tempVar = &(home.newSetVar(glbD, lubD, cardMin, cardMax));
        sa << *tempVar;
        vectorOfIndex.push_back(tempVar->getIndexInVector());
    }
}

//METHODS
unsigned int size(void) {

```

```

    return sa.size();
}
void add(GlSetVar& x) {
    sa << x;
    vectorOfIndex.push_back(x.getIndexInVector());
}
const Gecode::SetVarArgs& get(void) {
    return sa;
}

const Gecode::SetVar& getVar(int i) {
    assert(i >= 0 && i < sa.size());
    GlSetVar& toReturn = ((GlSetVar&) sa[i]);
    return toReturn;
}

//return the index in sv of an IntVar in sa
const int getIndexOfAVarInVector(int i)
{
    return vectorOfIndex[i];
}
};

class CPreRelVarList {
    MPG::CPreRelVarArgs cpa;
    std::vector<int> vectorOfIndex; //will contain the indexes of the
    IntVar objects in the iv vector

    CPreRelVarList(GlSpace& home, int n, const MPG::CPreRel::GRelation& l,
        const MPG::CPreRel::GRelation& u)
    {
        GlCPreRelVar* tempVar;
        for (int i = 0; i < n; i++) {
            tempVar = &(home.newCPreRelVar(l,u));
            cpa << *tempVar;
            vectorOfIndex.push_back(tempVar->getIndexInVector());
        }
    }

    unsigned int size(void) {
        return cpa.size();
    }
    void add(GlCPreRelVar& x) {
        cpa << x;
        vectorOfIndex.push_back(x.getIndexInVector());
    }
    const MPG::CPreRelVarArgs& get(void) {
        return cpa;
    }
}

```

```

    const GICPRelVar& getVar(int i) {
        assert(i >= 0 && i < cpa.size());
        GICPRelVar& toReturn = ((GICPRelVar&) cpa[i]) ;
        return toReturn;
    }

    //return the index in rv of an CPRelVar in cpa
    const int getIndexOfAVarInVector(int i)
    {
        return vectorOfIndex[i];
    }
};

```

```

class IntList {

private:
    Gecode::IntArgs va;
public:

    IntList(int n):va(n){}

    IntList(int n, int defaultValue):va(n)
    {
        for(int i = 0; i < n; i++)
            {
                va[i] = defaultValue;
            }
    }

    IntList(int n, int start, int inc):va(n)
    {
        for(int i = 0; i < n; i++, start+=inc)
            {
                va[i] = start;
            }
    }
    const Gecode::IntArgs& get(void) {
        return va;
    }

    unsigned int size(void) {
        return va.size();
    }

    const int getVar(int i) {
        assert(i >= 0 && i < va.size());

```

```

    return va[i];
}
const void setVar(int i, int value) {
    assert(i >= 0 && i < va.size());
    va[i] = value ;
}

const char* print(){

    std::stringstream out;
    out << "[ " ;
    for(int i = 0; i < va.size() - 1; i++)
        {
            out << va[i] << ", " ;
        }
    out << va[va.size() - 1] << "]" << std::endl;
    return out.str().c_str();
}

};

class IntSetList {

private:
    Gecode::IntSetArgs isa;
public:
    IntSetList(int n, int min, int max):isa(n)
    {
        for(int i = 0; i < n; i++)
            {
                isa[i] = Gecode::IntSet(min,max);
            }
    }
    const Gecode::IntSetArgs& get(void) {
        return isa;
    }

    const Gecode::IntSet& getElement(int i) {
        return isa[i];
    }
};

}

#endif

```

gelisp.cpp

```

#include <sstream>
#include <gecode/search.hh>
#include <gecode/gist.hh>
#include <gelisp.hpp>

namespace GeLisp {

using namespace Gecode;
using namespace MPG;

GISpace::GISpace(void)
{
    freopen ("/tmp/mystdout.txt", "w", stdout);
    freopen ("/tmp/mystderr.txt", "w", stderr);
    std::cout << "Redirected stdout." << std::endl;
    std::cerr << "Redirected stderr." << std::endl;
    rv.reserve(1000);
}

GISpace::~GISpace(void)
{
    fclose(stdout);
    fclose(stderr);
}

GISpace::GISpace(bool share, GISpace& s)
: Space(share, s), iv(s.iv), bv(s.bv), sv(s.sv), rv(s.rv) {
    for (int i = 0; i<iv.size(); i++)
    {
        iv[i].update(*this, share, s.iv[i]);
        iv[i].setIndexInVector(s.iv[i].getIndexInVector());
    }
    for (int i = 0; i<bv.size(); i++)
    {
        bv[i].update(*this, share, s.bv[i]);
        bv[i].setIndexInVector(s.bv[i].getIndexInVector());
    }

    for (int i = 0; i<sv.size(); i++)
    {
        sv[i].update(*this, share, s.sv[i]);
        sv[i].setIndexInVector(s.sv[i].getIndexInVector());
    }

    for (int i = 0; i<rv.size(); i++)
    {
        rv[i].update(*this, share, s.rv[i]);
        rv[i].setIndexInVector(s.rv[i].getIndexInVector());
    }
}
}

```

```

Space*
GISpace::copy(bool share) {
    return new GISpace(share, *this);
}

GIntVar&
GISpace::newIntVar(int lb, int ub) {
    iv.push_back(GeLisp::GIntVar(*this, lb, ub));
    iv.back().setIndexInVector(iv.size() - 1);
    return iv.back();
}

GIntVar&
GISpace::newIntVar(const IntSet &d){
    iv.push_back(GeLisp::GIntVar(*this, d));
    iv.back().setIndexInVector(iv.size() - 1);
    return iv.back();
}

GBoolVar&
GISpace::newBoolVar(int lb, int ub){
    bv.push_back(GeLisp::GBoolVar(*this, lb, ub));
    bv.back().setIndexInVector(bv.size() - 1);
    return bv.back();
}

GSetVar&
GISpace::newSetVar() {
    sv.push_back(GeLisp::GSetVar(*this));
    sv.back().setIndexInVector(sv.size() - 1);
    return sv.back();
}

GSetVar&
GISpace::newSetVar(int glbMin, int glbMax, int lubMin, int lubMax,
    unsigned int cardMin,
    unsigned int cardMax){
    sv.push_back(GeLisp::GSetVar(*this, glbMin, glbMax, lubMin, lubMax,
        cardMin, cardMax));
    sv.back().setIndexInVector(sv.size() - 1);
    return sv.back();
}

GSetVar&
GISpace::newSetVar(const IntSet& glbD, int lubMin, int lubMax,
    unsigned int cardMin,
    unsigned int cardMax){
    sv.push_back(GeLisp::GSetVar(*this, glbD, lubMin, lubMax, cardMin, cardMax
    ));
    sv.back().setIndexInVector(sv.size() - 1);
}

```

```

        return sv.back();
    }

    G1SetVar&
    G1Space::newSetVar(int glbMin,int glbMax,const IntSet& lubD,
        unsigned int cardMin,
        unsigned int cardMax){
        sv.push_back(GeLisp::G1SetVar(*this,glbMin,glbMax,lubD, cardMin,cardMax
            ));
        sv.back().setIndexInVector(sv.size() - 1);
        return sv.back();
    }

    G1SetVar&
    G1Space::newSetVar(const IntSet& glbD,const IntSet& lubD,
        unsigned int cardMin,
        unsigned int cardMax){
        sv.push_back(GeLisp::G1SetVar(*this,glbD,lubD, cardMin,cardMax));
        sv.back().setIndexInVector(sv.size() - 1);
        return sv.back();
    }

    G1CPRelVar&
    G1Space::newCPRelVar(const MPG::CPRel::GRelation& l, const MPG::CPRel::
        GRelation& u)
    {
        rv.push_back(GeLisp::G1CPRelVar(*this,l,u));
        rv.back().setIndexInVector(rv.size() - 1);
        std::cerr << "Glb of new CPRelVar : " << rv.back().glb().arity()
            << std::endl;
        std::cerr << "Lub of new CPRelVar : " << rv.back().lub().arity()
            << std::endl;
        return rv.back();
    }

    const char*
    G1Space::status(void) {
        Space& sp = static_cast<Space&>(*this);
        switch (sp.status()) {
            case SS_FAILED:
                return "SS_FAILED";
            case SS_SOLVED:
                return "SS_SOLVED";
            case SS_BRANCH:
                return "SS_BRANCH";
        }
        return "UNKNOWN";
    }

    void
    G1Space::print(std::ostream& os) const {
        os << "Integer variables:" << std::endl;

```

```

for (int i = 0; i < iv.size(); i++)
    os << "\tiv[" << i << "]: " << iv[i] << std::endl;
os << "Bool variables:" << std::endl;
for (int i = 0; i < bv.size(); i++)
    os << "\tbv[" << i << "]: " << bv[i] << std::endl;
os << "Set variables:" << std::endl;
for (int i = 0; i < sv.size(); i++)
    os << "\tsv[" << i << "]: " << sv[i] << std::endl;
os << "CPRel variables:" << std::endl;
for (int i = 0; i < rv.size(); i++)
    {
        std::cout << rv[i] << std::endl ;
        os << "\trv[" << i << "]: " << rv[i] << std::endl;
    }
}

const char*
G1Space::debug(void) const {
    std::stringstream os;
    print(os);
    return os.str().c_str();
}

void
G1Space::branch(void) {
    std::cout << " will branch...." << std::endl;
    if (iv.size() > 0) {
        IntVarArgs ivArgs;
        for (int i =0; i < iv.size(); i++) {
            ivArgs<<iv[i];
        }
        Gecode::branch(*this, ivArgs, INT_VAR_SIZE_MIN, INT_VAL_MIN);
    }

    if (bv.size() > 0) {
        BoolVarArgs bvArgs;
        for (int i =0; i < bv.size(); i++) {
            bvArgs<<bv[i];
        }
        Gecode::branch(*this, bvArgs, INT_VAR_SIZE_MIN, INT_VAL_MIN);
    }

    if (sv.size() > 0) {
        SetVarArgs svArgs;
        for (int i =0; i < sv.size(); i++) {
            svArgs<<sv[i];
        }
        Gecode::branch(*this, svArgs, SET_VAR_NONE, SET_VAL_MIN_INC);
    }

    if (rv.size() > 0) {
        for (int i =0; i < rv.size(); i++) {
            MPG::branch(*this, rv[i]);
        }
    }
}

```

```

    }
}

template<template<class> class Engine>
unsigned int
G1Space::runEngine(std::ostream& out, int nbSolutions, std::vector<G1Space*>&
    solutionsVector) {
    std::cout << " will search ...." << std::endl;
    using namespace std;
    Search::Options o;
    o.threads = 1;
    Engine<G1Space> se(this,o);
    int noOfSolutions = nbSolutions;
    int findSol = noOfSolutions;
    G1Space* sol = NULL;
    while (G1Space* next_sol = se.next()) {
        sol = next_sol;
        solutionsVector.push_back(sol);
        sol->print(out);
        out << "—————" << std::endl;
        if (--findSol==0)
            goto stopped;
    }
    stopped:
        Gcode::Search::Statistics stat = se.statistics();
        out << std::endl
            << "%% solutions:      "
            << std::abs(noOfSolutions - findSol) << std::endl
            << std::endl;
        out << std::endl << "No more solutions" << std::endl;
        std::cout << "search finished ..." << std::endl;
        return std::abs(noOfSolutions - findSol);
}

const char* G1Space::run(int nbSolutions, std::vector<G1Space*>& solutionsVector
) {
    std::stringstream os;
    runEngine<DFS>(os,nbSolutions,solutionsVector);
    return os.str().c_str();
}

G1IntVar& G1Space::getIntVar(int indexInVector)
{
    return iv[indexInVector];
}

G1BoolVar& G1Space::getBoolVar(int indexInVector)
{
    return bv[indexInVector];
}

```

```

    G1SetVar& G1Space::getSetVar(int indexInVector)
    {
        return sv[indexInVector];
    }

    G1CPRelVar& G1Space::getCPRelVar(int indexInVector)
    {
        return rv[indexInVector];
    }

std::vector<G1Space*> G1Space::createG1SpaceVector(void)
{
    std::vector<G1Space*> vec;
    return vec;
}

G1Space* G1Space::getG1SpaceFromVector(std::vector<G1Space*>& vec, int index)
{
    if(index >= 0 && index < vec.size())
        return vec[index];
    else
        return NULL;
}
}

```

Problème des n reines résolu avec GeLisp

```

;load GeLisp
(load "/Users/saschavancauwelaert/Documents/EPL/Memoire/gelisp/setup.lisp")

(defun n-queens (n)

  ;G1Space creation
  (setq sp (new_G1Space))

  ; declare the array of decision variables
  (setq queen (new_IntVarList_minmax sp n 0 (- n 1)))

  ;create IntArgs objects, useful for the linear constraint (linear expressions
  haven't been interfaced)
  (setq intlist1 (new_IntList_sizeAndStartAndInc n 0 1))
  (setq intlist2 (new_IntList_sizeAndStartAndInc n 0 -1))

  ; post all different constraint on queen
  (distinct_default sp (IntVarList_get queen))
  (distinct_constant_added_to_var_default sp (IntList_get intlist1) (
    IntVarList_get queen))
  (distinct_constant_added_to_var_default sp (IntList_get intlist2) (
    IntVarList_get queen))

```

```

;branch on all variables with default branchers of GeLisp
(GlSpace_branch sp)
;(GlSpace_run sp 1) deprecated

;create a vector of GlSpaces to save the desired number of solutions (allows to
  use the values assigned to the different variables afterwards)
(setq v (GlSpace_createGlSpaceVector sp))

;do the search for one solution and save it in the vector
(GlSpace_run sp 1 v)

)

```

Résolution d'un problème soluble par la librairie Situation avec GeLisp

```

(lambda (nobj ppts npts idst cnstr xdst)

  (progn

    ;cree l'espace de recherche
    (setq sp (cl-user::new_GlSpace))

    ;boucle pour creer les accords .
    (setq chords (list (cl-user::new_IntVarList_minmax sp npts (first ppts) (first
      (rest ppts))))))
    (loop for x from 1 to (- nobj 1) do
      (setq chords (append chords (list (cl-user::new_IntVarList_minmax sp npts (
        first ppts) (first (rest ppts)))))))
    )

    ;cree une liste d'intervalles verticaux interdit (les intervalles redoubles ne
      sont pas pris en compte par situation 3)
    (setq forbiddenVerticalIntervals nil)
    (setq tempRestOfIdst (rest idst))
    (loop for x from (+ (first idst) 1) to (first (last idst)) do
      (if (= x (first tempRestOfIdst))
        (setq tempRestOfIdst (rest tempRestOfIdst))
        (setq forbiddenVerticalIntervals (cons x forbiddenVerticalIntervals))))
    )

    ;cree une liste d'intervalles horizontaux interdit (les intervalles redoubles
      ne sont pas pris en compte par situation 3)
    (setq forbiddenHorizontalIntervals nil)
    (setq tempRestOfXdst (rest xdst))
    (loop for x from (+ (first xdst) 1) to (first (last xdst)) do
      (if (= x (first tempRestOfXdst))
        (setq tempRestOfXdst (rest tempRestOfXdst))
        (setq forbiddenHorizontalIntervals (cons x forbiddenHorizontalIntervals))))
    )
  )
)

```

```

    (setq forbiddenHorizontalIntervals (cons x forbiddenHorizontalIntervals
      )))
  )

;tableau destine a contenir les notes de la voix superieure
(setq notesUpperVoice (cl-user::new_IntVarList))

;post des contraintes
(dolist (tempChord chords nil)

  ;enregistrement de la note superieur de l'accord
  (cl-user::IntVarList_add notesUpperVoice (cl-user::IntVarList_getVar
    tempChord (- npts 1)))

  ;contrainte d'ordonnancement des notes d'un accord (casse la symmetrie, et
    de plus les renversements d'accords ne sont pas pris en compte car les
    intervalles redoubles ne sont pas pris en compte)

  (cl-user::rel_onIVA_default sp (cl-user::IntVarList_get tempChord) :IRT_LE)

  ;contraintes intervalles verticaux

  ;retrait des intervalles interdits et imposition de borne superieur et
    inferieur pour les intervalles
  (loop for x from 0 to (- npts 2) do

    ;liste des coefficients de l'expression reguliere exprimee a l'aide de la
      contrainte linear
    (setq coeffList (cl-user::new_IntList_sizeAndDefaultValue npts 0))
    ;les deux notes qu'on veut contraindre ont pour coefficient -1 et 1, les
      autres 0
    (cl-user::IntList_setVar coeffList x -1)
    (cl-user::IntList_setVar coeffList (+ x 1) 1)

    ;retrait des intervalles interdits
    (dolist (tempForbidInter forbiddenVerticalIntervals nil)
      (cl-user::linear_onIntArgsAndIVAAndInt_default sp (cl-user::
        IntList_get coeffList) (cl-user::IntVarList_get tempChord) :IRT_NQ
        tempForbidInter)
    )

    ;imposition de bornes superieur et inferieur pour les intervalles
    (cl-user::linear_onIntArgsAndIVAAndInt_default sp (cl-user::
      IntList_get coeffList) (cl-user::IntVarList_get tempChord) :
      IRT_GQ (first idst))
    (cl-user::linear_onIntArgsAndIVAAndInt_default sp (cl-user::
      IntList_get coeffList) (cl-user::IntVarList_get tempChord) :
      IRT_LQ (first (last idst)))

  )

)

```

```

; contraintes intervalles horizontaux
(loop for x from 0 to (- nobj 2) do

  ;liste des coefficients de l'expression reguliere exprimee a l'aide de la
  contrainte linear
  (setq coeffList (cl-user::new_IntList_sizeAndDefaultValue nobj 0)
  )
  ;les deux notes qu'on veut contraindre ont pour coefficient -1 et
  1, les autres 0
  (cl-user::IntList_setVar coeffList x -1)
  (cl-user::IntList_setVar coeffList (+ x 1) 1)

  ;retrait des intervalles interdits
  (dolist (tempForbidInter forbiddenHorizontalIntervals nil)
    (cl-user::linear_onIntArgsAndIVAAndInt_default sp (cl-user::
      IntList_get coeffList) (cl-user::IntVarList_get
      notesUpperVoice) :IRT_NQ tempForbidInter)
  )

  ;imposition de bornes superieur et inferieur pour les intervalles
  (cl-user::linear_onIntArgsAndIVAAndInt_default sp (cl-user::
    IntList_get coeffList) (cl-user::IntVarList_get notesUpperVoice)
    :IRT_GQ (first xdst))
  (cl-user::linear_onIntArgsAndIVAAndInt_default sp (cl-user::
    IntList_get coeffList) (cl-user::IntVarList_get notesUpperVoice)
    :IRT_LQ (first (last xdst)))

  )

;resolution du MCSP

(setq solVec (cl-user::G1Space_createG1SpaceVector sp))

(cl-user::G1Space_branch sp)
(cl-user::G1Space_run sp 1 solVec)

;recupere le premier espace solution
(setq solSpace (cl-user::G1Space_getG1SpaceFromVector sp solVec 0))

;enregistrement de la solution dans un format qu'OpenMusic peut utiliser.
(setq sol nil)
(dolist (tempChord chords nil)

  (setq tempSolChord nil)
  (loop for x from 0 to (- npts 1) do

    (setq tempSolChord (cons (* (cl-user::IntVar_val (cl-user::
      G1Space_getIntVar solSpace (cl-user::

```

```

                IntVarList_getIndexOfAVarInVector tempChord x)) ) 100)
            tempSolChord ))
        )
    (setq sol (cons (reverse tempSolChord) sol))
    )
    (setq sol (reverse sol) )
}
)

```

Résolution du problème de répartition d'instruments

```

(lambda (groundRelparam nbSol numSol)
  (progn
    ;create a ground relation from the paraml list passed as arguments
    ;(setq tupleList (cl-user::getTuplesFromScoreParam (car groundRelparam) (car (
      cdr groundRelparam)) (car (last groundRelparam))))
    (setq tupleList (cl-user::getTuplesFromScoreParam groundRelparam))
    (setq lubC (cl-user::createGroundRelFromTupleList tupleList))
    (setq glbC (cl-user::createGroundRelFromTupleList tupleList))

    ;space
    (setq sp (cl-user::new_GlSpace))

    ;ground relations
    ;(setq lubA (cl-user::new_GRelation 3))
    (setq glbA (cl-user::new_GRelation 3))
    (setq lubA (cl-user::createBoundedFullGroundRelation '(64 78 0 7 1 1)))

    ;(setq lubB (cl-user::new_GRelation 3))
    (setq glbB (cl-user::new_GRelation 3))
    (setq lubB (cl-user::createBoundedFullGroundRelation '(64 78 0 7 1 1)))

    ;create the decision variable
    (setq relA (cl-user::GlSpace_newCPRelVar sp glbA lubA))
    (setq relB (cl-user::GlSpace_newCPRelVar sp glbB lubB))
    (setq relC (cl-user::GlSpace_newCPRelVar sp glbC lubC))

    (cl-user::union_rel sp relA relB relC)

    (cl-user::disjoint sp relA relB)

    (cl-user::glSpace_debug sp)
  )
)

```

```
(cl-user::G1Space-branch sp)
(setq v (cl-user::G1Space-createG1SpaceVector sp))
(cl-user::G1Space-run sp nbSol v)
(setq sol (cl-user::G1Space-getG1SpaceFromVector sp v numSol))
;(cl-user::G1Space-debug sol)
(list (cl-user::getScoreParamFromSol 3 sol relA) (cl-user::
      getScoreParamFromSol 3 sol relB))
)
)
```

D

Fichiers contenus sur le CD-ROM annexe

Au-delà des autres annexes, un CD-ROM a également été fourni au terme de ce travail, le code source contenu dans certains fichiers étant trop long pour être fourni en version papier.