Intégration d'un système de contraintes temporelles dans le lociciel de composition musicale *OpenMusic*

Bruno Valèze

5 septembre 2006

Table des matières

Résumé

Il existe de nos jours sur le marché de nombreux logiciels pour la création musicale assistée par ordinateur, souvent polyvalents et très riches en fonctionnalités mais finalement assez semblables, chacun trouvant ses adeptes en fonction de critères d'ergonomie qui diffèrent en fonction du public visé. *Boxes* et *OpenMusic* sont deux logiciels dédiés à la composition musicale, chacun avec des spécificités différentes, et offrant une approche se démarquant de la majorité. Boxes présente la particularité de pouvoir organiser les objets musicaux (sons) entre eux, au moyen de contraintes temporelles. Le but de ce rapport est d'analyser de quelle manière ce système a été mis en place, puis d'élaborer une stratégie visant à intégrer un système similaire au sein d'OpenMusic.

Chapitre 1

Contexte du stage

1.1 Le SCRIME

Le *Studio de Création et de Recherche en Informatique et Musique Électroacoustique*[1] est une cellule d'activité rassemblant artistes et scientifiques. Son objectif est de permettre aux premiers de bénéficier d'un transfert de connaissances scientifiques et aux seconds d'une expertise musicale. Le *Scrime* résulte d'une convention de coopération entre le Conservatoire National de Région de Bordeaux, l'ENSEIRB, et l'université Bordeaux 1. Les membres du Scrime sont des chercheurs en informatique musicale du *LaBRI* (*Laboratoire Bordelais de Recherche en Informatique*) et des compositeurs issus du Conservatoire National de Région de Bordeaux et de l'association Octandre, pour la plupart.

La recherche fondamentale menée au Scrime concerne principalement la modélisation informatique du son (analyse, transformation et synthèse), et l'aide à la composition (calcul symbolique) ; tandis que la recherche appliquée concerne davantage les interfaces musicales Homme-Machine (logiciels pédagogiques, instruments virtuels).

[1] http ://scrime.labri.fr/

1.2 L'IRCAM

L'*Institut de Recherche et de Coordination Acoustique/Musique*[2] est crée en 1969 par le président Georges Pompidou. La direction est alors confiée au compositeur et chef d'orchestre Pierre Boulez. L'IRCAM dispose d'un statut d'association à but non lucratif reconnue d'utilité publique, et est associé au Centre Pompidou et placé sous la tutelle du ministère de la culture et de la communication. Depuis 1995, l'Institut accueille une unité mixte avec le CNRS.

La vocation première de l'IRCAM est d'établir une coordination entre recherche scientifique et création musicale, ce qui se traduit au travers des nombreuses activités de recherche, création/diffusion et enseignement mises en place au sein de l'Institut.

1.2.1 Les activités de recherche

L'IRCAM accueille actuellement près de 90 scientifiques pour mener à bien les activités de recherche, ce qui en fait le plus grand centre de recherche scientifique au monde entièrement dédié aux technologies pour la création musicale. Ces activités regroupent à la fois la recherche fondamentale - en mathématiques, acoustique, informatique et physique - et appliquée avec notamment l'acoustique instrumentale ou spatiale, l'analyse le traitement et la synthèse des sons, la représentation numérique des structures musicales ou encore la musicologie. Depuis 2002 ont lieu chaque année à l'IRCAM les rencontres *Résonances* accueillant des chercheurs du monde entier, et qui sont l'occasion d'un échange et d'une réflexion à la fois scientifique et artistique.

1.2.2 Création et diffusion

Depuis 1977, les résultats de la recherche ont été utilisés par 240 compositeurs pour la production de leurs oeuvres. Ceux-ci sont accueillis dans les studios de l'IRCAM tout au long de l'année, les projets étant sélectionnés en fonction de leur adéquation avec les priorités de l'Institut, notamment en matière de recherche.

Ces œuvres sont diffusées régulièrement au cours de rendez-vous qui ont lieu aussi bien à l'IRCAM que dans d'autres lieux propices à la diffusion, et qui sont l'occasion de présenter les jeunes compositeurs issus du cursus que dispense l'Institut.

1.2.3 Activités pédagogiques

L'Ircam dispense deux formations universitaires qui le rapprochent du monde de la recherche scientifique : un Mastère Sciences et Technologies (Paris VI/Ircam) - Parcours

[2]http ://www.ircam.fr

ATIAM - et une Formation Interdisciplinaire de Recherche (ENS/Ircam/CNSMDP) - Firmus.

Un cursus d'un an, destiné à dix jeunes compositeurs de niveau international, traite de composition et d'informatique musicale. Un dispositif post-cursus permet d'inscrire dans la durée, sous la forme de résidence "jeunes compositeurs", des collaborations avec ces jeunes musiciens qui renouvellent les problématiques musicales.

1.3 Musiques et informatique

1.3.1 La musique électroacoustique

Le terme *électroacoustique*[3] ne désigne pas plus une esthétique musicale qu'une manière particulière d'aborder la composition. Il concerne la technologie utilisée, et non la manière de la mettre en oeuvre. Pour tenter d'éviter cette confusion, de nombreux compositeurs de musique électroacoustique sérieuse ont décidé d'utiliser les termes plus spécifiques de musique électronique, concrète, acousmatique, mixte, etc., toutes réunies sous le vocable global d'électroacoustique.

Au sens large, les moyens électroacoustiques sont tous les appareils et instruments qui utilisent l'électronique et/ou l'informatique pour enregistrer, reproduire, synthétiser, analyser ou transformer le son. Ils comprennent entre autres les microphones, les enregistreurs, les amplificateurs et les haut-parleurs, mais aussi les filtres, les modules d'effets, de transposition et de transformation ainsi que les générateurs de fréquences, les synthétiseurs de son, les systèmes informatiques généraux ou dédicacés, etc.

1.3.2 L'informatique musicale

Avides de découvrir de nouveaux moyens de formaliser la pensée musicale ainsi que de nouveaux outils de synthèse et de transformation des sons, les compositeurs ont été rapidement attirés par les possibilités virtuellement illimitées de l'ordinateur. On peut identifier deux voies principales de recherches : d'une part la manipulation de symboles abstraits tels que les notes d'une partition traditionnelle, d'autre part la manipulation des sons proprement dits. Au premier âge de l'informatique, la puissance disponible rendait la deuxième option impraticable. C'est donc dans le domaine de la composition de partitions destinées à être jouées par des instrumentistes que les premiers pionniers ont travaillé. Selon les cas, les compositeurs écrivent des programmes suffisamment complexes pour générer l'oeuvre complète ou, au contraire, utilisent les résultats des programmes comme matériaux bruts retravaillés ensuite de manière traditionnelle.

Actuellement, c'est surtout la deuxième approche qui prévaut. On parle alors de CMAO (Composition Musicale Assistée par Ordinateur) : l'ordinateur propose une bibliothèque d'outils de traitement des abstractions qui seront utilisés tour à tour par le compositeur en fonction de ses besoins, mais ce sera toujours à ce dernier qu'incombera le choix final parmi l'ensemble des solutions proposées par l'ordinateur. La deuxième voie de recherche, qui consiste à créer des outils de synthèse, d'analyse et de transformation des sons est plus proche de l'esprit de la musique concrète ou acousmatique. Mais il

[3] *Musique Electroacoustique et Acousmatique* http ://users.pandora.be/alver.bvba/beffeb/febeme/eamus.htm

est bien sûr possible de combiner les deux approches en recourant à une procédure algorithmique pour générer les valeurs des paramètres qui seront ensuite utilisées par le processus de réalisation sonore.

Depuis les prémices de l'utilisation des ordinateurs dans la musique, de nombreux chercheurs et compositeurs ont suivi les pas des pionniers et une nouvelle branche scientifique est née : l'informatique musicale. Elle est, par définition, pluridisciplinaire puisque l'enjeu consiste à marier harmonieusement différentes disciplines scientifiques au service de la musique. La collaboration entre scientifiques d'une part, et les compositeurs d'autre part est donc une nécessité pour qui désire progresser rapidement et efficacement dans ce domaine. C'est pourquoi les structures de recherche performantes, comme l'IRCAM, réunissent généralement ingénieurs, informaticiens, acousticiens, psychologues, musicologues, enseignants et compositeurs. Ceci dans un effort commun de formalisation, de réalisation et d'évaluation d'outils originaux qui enrichissent le langage musical contemporain en proposant de nouvelles techniques d'écriture et en offrant aux compositeurs des possibilités expressives accrues.

Chapitre 2

Cahier des charges

2.1 Sujet initial

Le sujet tel qu'il était établi au départ consistait à "intégrer le logiciel de composition musicale Boxes au sein de l'environnement OpenMusic développé par l'Ircam". En effet, Boxes comporte un modèle de contraintes temporelles entre les évènements musicaux qui est absent dans OpenMusic. Nous allons donc en premier lieu étudier les besoins, présenter brièvement ces deux logiciels puis nous pourrons ensuite préciser le travail à effectuer et voir comment le sujet à évolué au cours de la phase d'analyse et de conception par laquelle a débuté le stage.

2.2 Besoins

2.2.1 Contraintes temporelles entre des évènements musicaux

Le principe d'établir des relations temporelles entre des objets graphiques (boîtes) correspondant à des objets musicaux (partition MIDI, fichier audio...) permet à un compositeur d'organiser temporellement ses différentes parties ou sons de manière durable, car lorsque une relation est établie entre deux éléments d'une pièce, on garantit l'intégrité de cette contrainte à tout moment. Ceci est particulièrement intéressant lorsqu'on considère que le compositeur va élaborer une pièce selon différents mouvements, travaillant chacun d'eux d'une manière relativement indépendante.

Avec les séquenceurs ou logiciels multipistes disponibles dans le commerce - qui ne permettent pas d'établir ce type de contraintes - il est aisé d'imaginer que dans le cas d'une pièce complexe une simple modification d'un évènement musical peut mettre en cause l'intégrité de la pièce. Ceci puisque les relations entre les évènements, établies mentalement par le compositeur mais ignorées par le système, ont des risques de se retrouver non vérifiées. Il peut être ensuite délicat et fastidieux de remettre à jour manuellement la pièce afin qu'elle re-vérifie les contraintes décidées au départ.

Pour formaliser cette idée, on va s'appuyer sur les relations de Allen (illustrées par la figure 2.1) qui sont les relations binaires suivantes :
- *A before B* : A est joué avant B ;
- *A meets B* : B commence lorsque B se termine ;
- *A overlaps B* : A commence avant, et se termine pendant, B ;
- *A starts B* : A est joué pendant, et commence en même temps, que B ;
- *A equals B* : A commence et finit en même temps que B ;
- *A during B* : A est joué pendant B ;
- *A finishes B* : A est joué pendant, et se termine en même temps, que B ;
- *A after B* : A est joué après B.

2.2.2 Interaction avec le compositeur

Tout comme dans Boxes, les actions permises dans le cadre des contraintes au compositeur utilisant OpenMusic seront les suivantes :
- ajout de contraintes entre deux boîtes ;
- suppression de contraintes devenues obsolètes ;
- déplacement et modification des longueurs des boîtes avec pour résultat une mise à jour de la configuration pour que le système soit toujours valide.

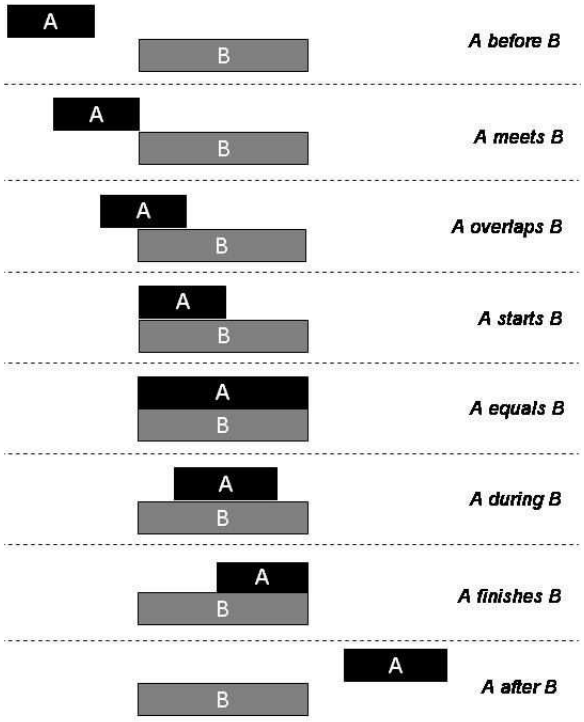

FIG. 2.1 – Relations de Allen

Afin que le système de contraintes soit exploitable dans le cadre d'une utilisation par les compositeurs, il est indispensable de respecter certains impératifs :

- intuitivité : le comportement du logiciel, en particulier la distinction entre date de début et longueur des boîtes, doit correspondre à l'intuition qu'en a le compositeur ;
- temps de calcul raisonnables : la mise à jour de la configuration des boîtes doit se faire quasi-instantanément à la suite d'un ajout de contrainte ou d'une édition.

2.3 Présentation de *Boxes*

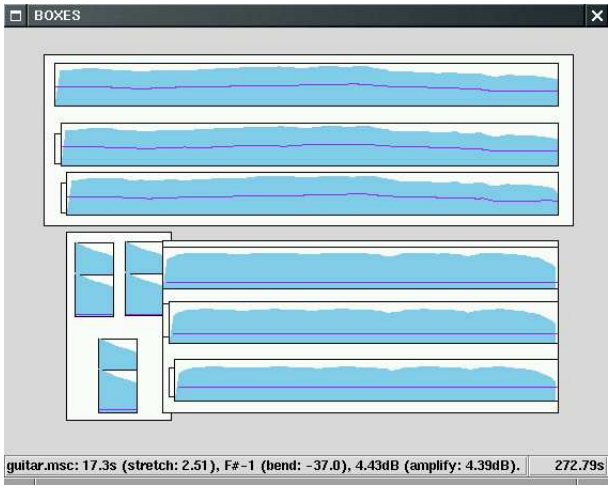

FIG. 2.2 – L'interface graphique de *Boxes*

Boxes (cf. [1] et [2]) est un logiciel de composition musicale, développé en 1999 par Anthony Beurivé au Scrime. Il est écrit dans le langage *STklos* (cf. [3]) qui est une variante de *Scheme* comprenant une couche supplémentaire pour la programmation orientée objet. Il comporte une interface graphique très simple (cf figure 2.2) qui permet au compositeur de placer sur le plan de travail des "boîtes", des objets graphiques représentant des sons. Par rapport aux produits disponibles sur le marché, Boxes présente trois particularités :

- il intègre un module de traitement spectral permettant aisément et très efficacement l'étirement ou la contraction des sons (*time stretching*) ainsi que la modification des hauteurs (*pitch shifting*) ;
- il fournit des structures de données hiérarchiques qui permettent de diviser une pièce en parties distinctes ou regrouper plusieurs sons en une même entité ;
- il permet la composition de pièces basées sur des contraintes temporelles entre les boîtes.

Les fonctionnalités de Boxes sont par ailleurs très réduites comparées aux séquenceurs et logiciels multipistes actuels mais ce dépouillement d'une part et ses trois particularités d'autre part en font un logiciel très intéressant pour la composition de pièces électro-acoustiques.

Une étude approfondie de Boxes a été menée au cours de ce stage, en particulier pour comprendre comment est abordé le problème des contraintes temporelles, et elle est présentée au chapitre 3.

2.4 Présentation d'*OpenMusic*

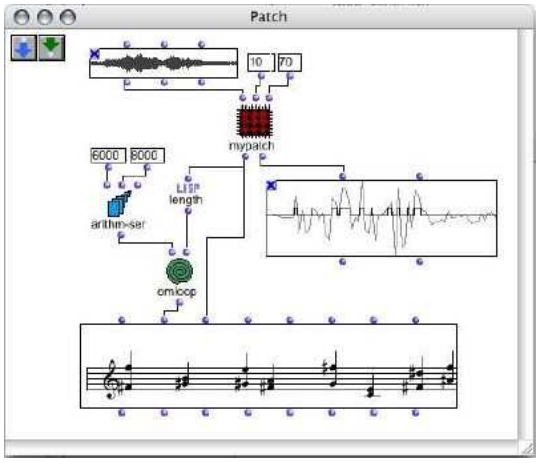

FIG. 2.3 – Exemple de patch dans *OpenMusic*

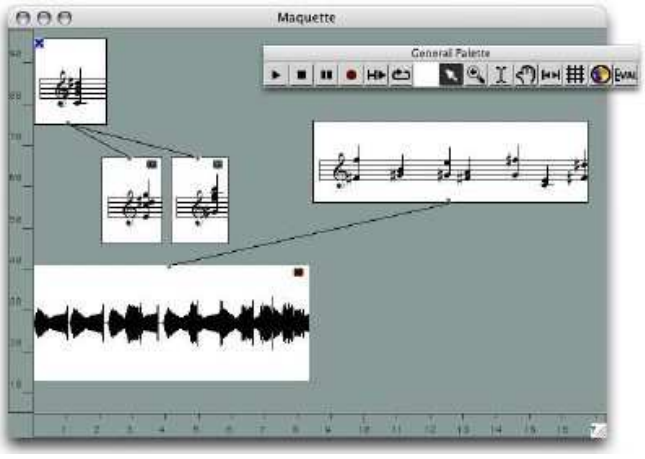

FIG. 2.4 – Exemple de maquette dans *OpenMusic*

OpenMusic (cf. [4]) est un logiciel très complexe dédié à la composition musicale. Il a la particularité de proposer une approche très éloignée des autres logiciels existant dans le domaine en s'appuyant notamment sur un point de vue très rigoureux et mathématique de la composition musicale. Par exemple, OpenMusic permet de créer des patches (voir figure 2.3) qui sont en fait des combinaisons de fonctions, de processus, appliqués sur des entités musicales. Un patch peut servir par exemple à générer automatiquement une mélodie à partie d'un accord, ou inversement. Au final, OpenMusic est tout simplement un langage de programmation visuel, orienté objet, et qui n'est d'ailleurs pas utilisé seulement par des compositeurs. Il s'appuie sur *MCL* (*Macintosh Common Lisp*, voir [5]) qui est un environnement de développement complet en Common Lisp.

Malgré tout, on trouve un grand nombre de fonctionnalités dédiées à la composition comme par exemple l'éditeur de partitions, des modules d'analyse et de synthèse sonore, et un objet maquette (voir figure 2.4) qui s'approche de l'interface de Boxes et permet d'organiser des entités musicales variées dans le domaine temporel. OpenMusic est donc un environnement très puissant pour le compositeur, paramétrable à l'infini mais dont l'utilisation requiert un apprentissage conséquent et des notions poussées en programmation. Les compositeurs sont d'ailleurs en général assistés par des ingénieurs lorsqu'ils travaillent sous OpenMusic.

2.5 Redéfinition du sujet

Pour des raisons que nous verrons dans le chapitre 3 consacré à l'étude de Boxes, l'utilisation de la bibliothèque *Cassowary* qui était utilisée pour la résolution des contraintes a du être exclue. Les développeurs de l'Ircam travaillant sur OpenMusic ont alors décidé d'explorer plusieurs pistes pour la résolution des contraintes. L'objectif était alors en parallèle d'implémenter des solutions basées sur différents algorithmes pour comparer leurs performances, et également d'essayer d'utiliser une autre bibliothèque réputée très complète et performante, à savoir *Gecode*. Mon travail a été dès lors d'explorer cette éventualité, sachant que le fonctionnement de Gecode est très différent de celui de Cassowary. On final on peut le résumer de la manière suivante :

- utiliser Gecode pour développer un solveur permettant d'ajouter dynamiquement des variables et des contraintes et d'obtenir les solutions en temps réel ;
- faire la liaison avec OpenMusic ;
- intégrer à la maquette d'OpenMusic les fonctionnalités liées aux contraintes de Allen.

Chapitre 3

Etude de Boxes

La première partie de mon stage a été d'analyser la conception de Boxes afin de voir de quelle manière il pourrait s'intégrer à OpenMusic. L'absence de documentation technique a entraîné la nécessité d'une étude complète de l'architecture du logiciel, en particulier des mécanismes liés à la gestion des contraintes.

3.1 Les contraintes temporelles dans Boxes

Pour établir des contraintes temporelles entre les boîtes, Boxes utilise les relations de Allen illustrées figure 2.1 et énumérées dans la section 2.2.1. Le compositeur peut déclarer ces contraintes très facilement au moyen de raccourcis clavier. Dès qu'une nouvelle contrainte est ajoutée, la configuration des boîtes doit éventuellement être mise à jour de manière à ce que le système de contraintes soit toujours satisfait. Pour effectuer ce travail, Boxes doit avoir recours à un solveur de contraintes, qui est en l'occurence une bibliothèque externe nommée *Cassowary* (cf. [6]). Celle-ci permet d'enregistrer des variables, des contraintes sur ces variables, et de récupérer une solution au système. Il est également possible de passer au solveur une *fonction objective* à minimiser ou maximiser, construite à partir des variables, qui lui permet d'orienter sa résolution pour obtenir la "bonne" solution parmi l'ensemble des solutions au système. Cette technique sera détaillée dans la section 4.1.3.

3.2 Architecture du logiciel

Nous allons présenter ici l'architecture du logiciel Boxes. Bien que tous les aspects n'aient pas été étudiés avec la même profondeur, une énumération exhaustive des classes a été dressée. En effet, la documentation technique de Boxes étant inexistante au début du stage, il était intéressant de s'y atteler dans le but de faciliter la tâche d'un futur programmeur qui aurait à travailler sur Boxes. Ce travail était d'autant plus utile que l'architecture de Boxes utilise beaucoup l'héritage multiple, ce qui permet une grande modularité mais rend la lecture du code assez éprouvante, d'où le côté appréciable de pouvoir profiter d'une documentation même minimaliste.

3.2.1 Les bibliothèques externes

ProSpect

Prospect[1] est une bibliothèque dédiée au traitement spectral du son. Elle a été développée par Sylvain Marchand, chercheur au LaBRI, et est écrite en C ce qui rend assez aisée son intégration par des programmes écrits dans d'autres langages.Dans Boxes, Prospect est utilisée pour les fonctions de transposition des sons et d'étirement temporel, et elle s'avère d'une grande efficacité puisque tout fonctionne quasiment en temps réel.

Cassowary

Cassowary[2] est une bibliothèque d'outils pour la résolution de contraintes développée par une équipe de chercheurs de l'université de Washington. Cette bibliothèque possède un solveur qui accepte l'ajout et la suppression dynamiques de variables et de contraintes, et calcule la solution actuelle automatiquement. Le solveur est également paramétré par une fonction objective, c'est à dire une expression linéaire en fonction des variables du système que le solveur doit minimiser ou maximiser pour atteindre la bonne solution parmi toutes celles qui vérifient le système. Cassowary est écrite en C++.

Utilisation des bibliothèques externes par Boxes

Comme nous l'avons vu, Boxes est écrit en STKlos ce qui n'est pas le cas des deux bibliothèques que nous venons de présenter. Néanmoins, il existe des solutions répandues - dont il n'y a pas d'intérêt à parler ici - pour utiliser dans des programmes en STK du code C++. Sachant que STKlos est une surcouche objet de STK, et qu'il est aisé d'écrire une interface C++ pour du code en C, on comprend bien que grâce à un certain nombre d'interfaces écrites dans les langages appropriés on arrive assez vite à utiliser Prospect et Cassowary depuis du code STKlos.

[1] http ://scrime.labri.fr/ProSpect

[2] http ://www.cs.washington.edu/research/constraints/cassowary

3.2.2 Classes dédiées à la gestion et au traitement des sons

Cette partie n'étant pas primordiale dans le cadre du projet, l'architecture est présentée en annexe, figures A.1 et A.2.

3.2.3 Représentation des objets graphiques

Elle est visible sur les deux figures en annexe A.3 et A.4, pour des raisons de lisibilité. De plus, seules les relations d'héritage apparaissent.

Première remarque : les classes dont le nom apparaissent en italique sont des classes abstraites. On remarque en un coup d'œil qu'elles sont nombreuses (*Boxes-Canvas*, *Ced-HBox*, *Ced-Box*, etc.), ceci par souci d'évolutivité du code. Nous ne passerons pas en revue les objets graphiques - pour les boîtes les *GBox* - car il n'y a pas lieu de rentrer dans ces détails dans le cadre du projet, une interface équivalente étant déjà implémentée dans OpenMusic sous la forme de l'objet *maquette*.

Les classes qui nous intéressent plus sont celles dérivées de *Ced-Box* et *Ced-HBox*. Elles correspondent à des représentations abstraites des boîtes dans Boxes. L'aspect gestion des contraintes apparaît dans l'architecture avec les classes dont le nom est préfixé par *Ced* (*constrained*), et la distinction hiérarchique entre les boîtes avec la classe *Ced-HBox* pour *constrained hierarchic box*. Si on regarde un peu plus en détail les attributs et méthodes, on constate que les objets représentant les boîtes sont organisés sous forme d'arbre, où la relation de descendance traduit une notion de contenance.

3.2.4 Interaction avec le solveur *Cassowary*

Les structures de données de Boxes plus spécifiques à la gestion des contraintes sont présentées en annexe figure A.5. Notons que comme pour le reste de l'architecture de Boxes, toutes les classes dérivent d'une classe abstraite (interface) mais elles n'apparaîssent pas sur le diagramme pour une meilleure lisibilité. Il suffit de savoir qu'en réalité, les noms apparaissant sur le schéma sont ceux des interfaces et que celles-ci sont chacune implémentées par la classe *InterfaceName-Implementation* où *InterfaceName* est le nom de l'interface.

Pour résumer, les objets du type *Ced-Objects* représentent des éléments qui peuvent être soumis à des contraintes. Ils sont associés dans Boxes aux objets graphiques représentant des boîtes (*GBox*) et lorsque les propriétés de l'un sont modifiées - par l'utilisateur pour les boîtes et par Cassowary pour les *Ced-Object* - celles de l'autre le sont automatiquement. Ainsi en simplifiant on a :

– des structures représentant des sons ;

– une collection d'objets graphiques pour visualiser ces sons ;
– des objets soumis à des contraintes et liés aux boîtes graphiques.

Nous ne rentrerons pas plus dans les détails car les mécanismes sont beaucoup plus complexes mais dans le cadre du projet, l'important est de bien visualiser les différentes couches de Boxes.

3.3 Conclusion de l'étude de Boxes

L'étude de Boxes met en avant la façon dont les objets graphiques représentant des sons et les relations de Allen établies entre ces objets sont traduits sous formes de variables et de contraintes soumises au solveur. On comprend également comment interagissent le logiciel de composition musicale et le solveur. Mais ce dernier, en l'occurence Cassowary, présente un problème de taille : son développement a été stoppé en 2000 car elle utilise une bibliothèque nommée *GTL* pour *Graph Template Library* (cf. [7]). Celle-ci, à l'époque du développement de Cassowary, était disponible gratuitement en version 0.3.1 mais depuis le passage à la première version stable (1.0.0), l'achat de la licence est impératif et s'élève à 500 euros. Après avoir contacté les développeurs, il s'avère que Cassowary utilisait GTL sans autorisation, ce qui explique peut-être en partie l'abandon du projet.

Chapitre 4

Définition d'une interface pour la résolution des contraintes temporelles et implémentation d'un solveur avec *Gecode*

4.1 Définition de l'interface du solveur

4.1.1 Stratégie : conception d'un solveur spécialisé

Aucun solveur adapté n'existant en Common Lisp, nous allons devoir utiliser une bibliothèque externe qui sera forcément appelée par OpenMusic via une interface en C. En effet, *Macintosh Common Lisp (MCL)*, l'interpréteur choisi par l'Ircam, dispose de macros permettant d'utiliser des fonctions C, et de surcroît le code en C est parmi tous les langages de programmation celui qui semble le plus simple à utiliser depuis d'autres langages.

Sachant qu'il faudrait utiliser une bibliothèque externe dédiée à la satisfaction de contraintes, deux solutions étaient alors envisageables :

- proposer une interface très riche, constituée de fonctions C appelant celles de la bibliothèque en question, dans le but de permettre à son utilisateur la plus grande liberté d'utilisation possible ;
- développer une couche réalisant une abstraction de la bibliothèque et constituant un solveur, et offrir ainsi une interface simple permettant à l'utilisateur d'utiliser le solveur de contraintes ainsi développé de manière efficace et sans avoir à se confronter aux difficultés inhérentes à l'utilisation de la bibliothèque.

L'approche retenue a été la seconde solution (voir figure 4.1), dans l'optique d'une

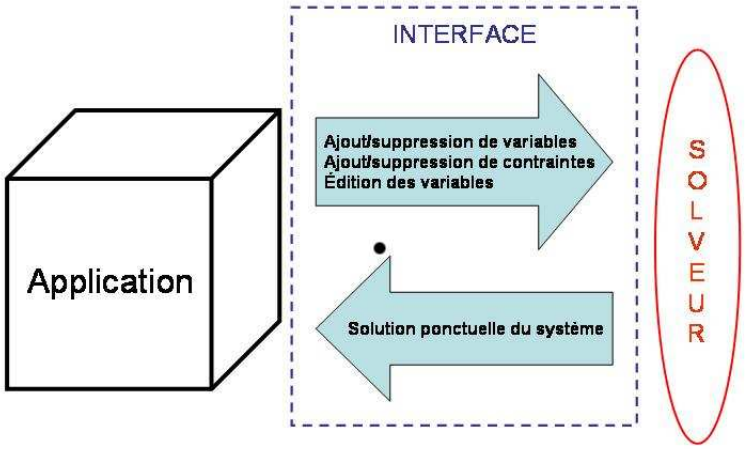

FIG. 4.1 – Interaction application/solveur

réutilisation possible par le Scrime du solveur de contraintes. En effet, intégrer la couche de dialogue avec la bibliothèque externe au sein même d'*OpenMusic* signifierait que dans l'éventualité d'une réutilisation de ce code pour une autre application, le programmeur serait contraint d'effectuer le portage vers le nouveau langage, voire de reprendre à zéro. D'autre part les outils spécialisés dans la satisfaction de contraintes sont en général très génériques ce qui entraîne des mécanismes très complexes, alors que notre problème peut se résumer à :

- déclarer un ensemble de variables définies sur un domaine entier ;
- déclarer des contraintes entre ces variables ;
- récupérer la meilleure solution (i.e. les nouvelles valeurs des variables qui sont les plus proches de ce que voudrait obtenir le compositeur) satisfaisant ce système de contraintes.

Si avec cette approche on restreint évidemment les fonctionnalités de la bibliothèque qui sera utilisée, cette manière de voir le problème peut convenir pour de nombreuses applications. Nous allons maintenant voir la manière dont a été conçue l'interface de notre solveur, en s'inspirant notamment de l'utilisation de la bibliothèque *Cassowary* dans *Boxes*.

4.1.2 Besoins au niveau du programme client

Dans la maquette *OpenMusic*, le compositeur dispose des "boîtes" qui matérialisent graphiquement des objets sonores. Celles-ci peuvent-être définies au niveau temporel par deux variables qui sont au choix :

- la date de début et la date de fin ;
- la date de début et la durée.

Nous verrons dans la section 4.1.4 quel choix a été adopté parmi ces deux possibilités, celui-ci n'ayant pas d'incidence sur la conception du solveur. En effet, on doit simplement retenir que chaque boîte est représentée par un ensemble de variables qui possèdent chacune un domaine de définition et une valeur initiale. Lorsqu'on ajoute des relations de Allen entre des boîtes, on les traduit en relations sur leurs variables. Voyons ceci sur un exemple avec deux boîtes A et B et la contrainte *A meets B*, $minX$ et $maxX$ étant respectivement la date minimale et la date maximale sur une maquette :

	dates de début et de fin	date de début et durée
variables	$beg\_A, beg\_B[minX, maxX - 1]$ $end\_A, end\_B[minX + 1, maxX]$	$beg\_A, beg\_B[minX, maxX]$ $length\_A, length\_B[1, maxX - minX]$
contraintes d'intégrité	$end\_A > beg\_A$ $end\_B > beg\_B$	$beg\_A + length\_A <= maxX$ $beg\_B + length\_B <= maxX$
A meets B	$end\_A = beg\_B$	$beg\_A + length\_A = beg\_B$

Tab. 4.1 – Exemple de relations entre variables

On a bien les mêmes choses exprimées de part et d'autre, mais de manière différente - à noter que les domaines de définitions des variables sont essentiels puisqu'ils expriment eux-mêmes déjà des contraintes sur les variables.

Assez rapidement, on peut alors préciser les fonctions de base que devra proposer le framework :

- ajouter une variable avec son domaine de définition et sa valeur initiale ;
- ajouter une contrainte linéaire sur un certain nombre de variables ;
- mettre à jour la valeur de variables après une édition ;
- accéder aux valeurs des variables.

auxquelles s'ajoutent les fonctions permettant d'enlever des variables (suite à la disparition d'une boîte) et des contraintes (sur décision de l'utilisateur d'enlever une contrainte de Allen).

4.1.3 Minimisation de l'écart entre le résultat souhaité et la solution calculée par le solveur

Lorsqu'une opération d'édition va être effectuée par le compositeur sur une ou plusieurs boîtes - un déplacement temporel par exemple - on ne pourra pas forcément l'autoriser telle quelle en raison des contraintes établies auparavant. Dans ce cas, on va essayer de minimiser les écarts entre les valeurs souhaitées - celles après édition pour les boîtes concernées et les anciennes pour les autres boîtes.

La méthode utilisée pour effectuer cette minimisation est similaire à celle mise en place dans *Boxes*. Chaque variable (date de début ou durée) va en fait être associée à quatre *IntVar* de Gecode :

- une pour la valeur qu'on voudrait dans l'idéal ;
- une pour la valeur optimale ;
- une pour la différence positive entre la première et la seconde ;
- une pour la différence négative entre la première et la seconde.

Les deux dernières sont nécessaires car on ne peut pas donner de valeur négative à une variable. Appelons respectivement ces variables *idealValue*, *optValue*, *posDelta* et *negDelta*. On devra alors déclarer la relation (contrainte) suivante :

$$idealValue = optValue + posDelta - negDelta \tag{4.1}$$

soit

$$optValue = idealValue - posDelta + negDelta$$

Les contraintes sont alors posées, pour chaque variable déclarée par *OpenMusic*, pour les composantes *optValue*. Pour fixer la valeur des *idealValue*, on fixe leur domaine réduit à cette seule valeur. Enfin, les *delta* permettent de construire la fonction objective suivante :

$$f = \sum_{i=O}^{n} (posDelta_i + negDelta_i) \tag{4.2}$$

On minimise donc tout simplement la somme des écarts et logiquement on se rapproche le plus possible de la solution "idéale" (cf équation 4.1). La manière dont on utilise la fonction objective avec Gecode est détaillée dans la section 4.2.4.

4.1.4 Affectation de poids aux variables

Prenons l'exemple d'une maquette composée de 5 boîtes, illustré à la figure 4.2. Dans cet exemple les boîtes sont liées de gauche à droite successivement par la relation "before".

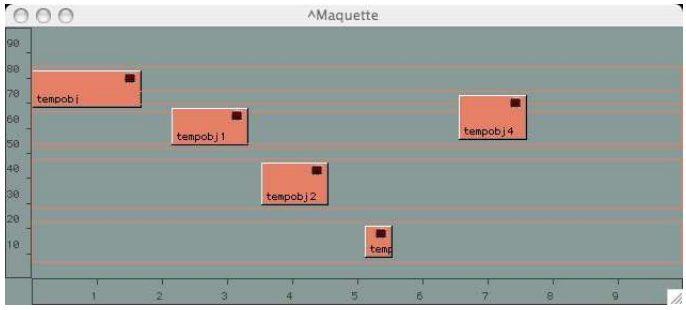

FIG. 4.2 – Exemple de maquette : 5 boîtes enchaînées

On va ensuite pousser la boîte de droite le plus possible vers la gauche. On obtiendra alors logiquement la configuration de la figure 4.3.

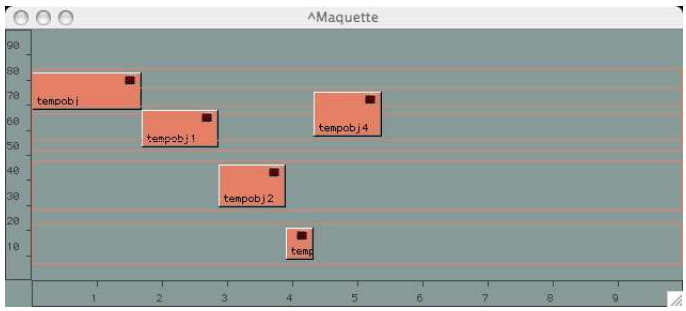

FIG. 4.3 – Tassement des boîtes sur la gauche

La question qui se pose à partir de là est la suivante : si l'utilisateur continue de "pousser" vers la gauche, doit on considérer que le déplacement est interdit ou qu'il faut réduire les durées de certaines boîtes (cf. figure 4.4) ?

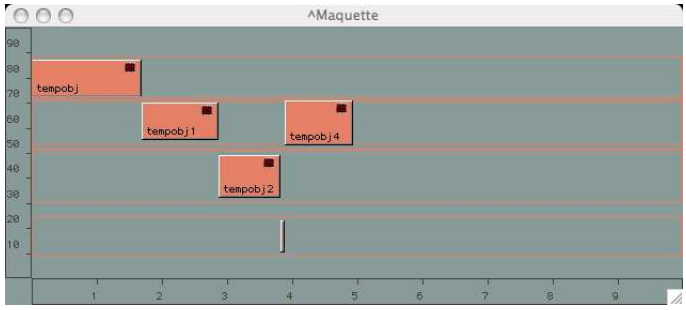

FIG. 4.4 – Compression de la durée

Étant donné que toutes les contraintes doivent toujours être respectées et comme l'on a vu que la recherche de la solution optimale était basée sur la minimisation des écarts sur les variables, la réponse à cette question consiste à affecter des poids aux variables dans la fonction objective. Cela signifie que l'on va établir une hiérarchie entre les variables consistant à privilégier la conservation des valeurs de certaines variables par rapport à d'autres :

- la conservation des durées est considérée plus importante que celle des dates ;
- les variables en cours d'éditions sont prioritaires.

L'affectation d'un poids *p(i)* à une variable i aura alors pour effet de modifier la fonction objective (cf. formule 4.2) de la manière suivante :

$$objFunction = \sum_{i=O}^{n} [p_i \times (posDelta_i + negDelta_i)] \tag{4.3}$$

Il est évident que dans ces conditions, il est prioritaire pour minimiser la fonction de réduire les *delta* des variables ayant un poids important.

Pour terminer, ce besoin d'affecter des poids aux variables en fonction de leur type règle le dilemme auquel nous avions été confrontés à là section 4.1.2. On doit forcément prendre comme variables pour les boîtes la date de début et la durée, car on peut hiérarchiser les variables grâce au poids alors que si on prend la date de fin au lieu de la durée, cette dernière se retrouve exprimée au moyen d'une contrainte (cf tableau 4.1). Cela impliquerait d'exprimer également les dates de début par des contraintes, et ensuite de hiérarchiser les contraintes ce qui complexifie énormément le problème.

4.1.5 Conception de l'interface en C

Le solveur sera identifié dans le programme client par un pointeur. Pour représenter les variables et les contraintes, il existe deux possibilités :

- travailler avec les adresses physiques sous la forme de pointeurs *void** dans les listes de paramètres ou les valeurs de retour des fonctions ;
- repérer les instances par des identifiants entiers.

La solution choisie a été la seconde, les identifiants étant liés aux instances par des tables de hachage. Ainsi, seul l'objet matérialisant le solveur est connu dans *OpenMusic* par son adresse physique. Les prototypes des fonctions nécessaires dans l'interface sont présentés dans la table 4.2.

```
// initialisation du solveur
void *instanciate_solver();

// ajoute une variable entière définie sur [min, max], de poids w,
// et retourne l'identifiant
int add_int_var( void *solver, int min, int max, int val, VarWeight w );

// enlève une variable
int remove_int_var( void *solver, int varID );

// ajoute la contrainte "c[0].x[0] + ... + c[nbVar-1].x[nbVar-1] <relType> val"
// et retourne son identifiant ou -1 si la contrainte ne peut pas être ajoutée
int add_linear_constraint( void *solver, int *x, int *c, int nbVars, int relType, int val);

// enlève une contrainte
int remove_constraint( void *solver, int constID );

// suggère de nouvelles valeurs pour certaines variables
int suggest_values( void *solver, int *x, int *vals, int nbVars );

// desalloue le solveur
void delete_solver( void *solver );

// accesseur pour la valeur de la variable <varID>
int get_val( void *solver, int varID );
```

TAB. 4.2 – Fonctions de l'interface C du framework

4.1.6 Un exemple d'utilisation de l'interface

Sur un exemple très simple, nous allons voir comment dialoguer avec le solveur. On simulera deux boîtes - donc quatre variables - que l'on va lier par la relation "meets". La configuration d'origine est :

– la boîte A commence à la date 2000 et sa durée vaut 1500 ;
– la boîte B commence à la date 3000 et sa durée vaut 2000.

Les dates sont comprises entre 0 et 10000. Les poids des variables de début et de durée sont des constantes déjà définies respectivement par *BEGIN_WEIGHT* et *LENGTH_WEIGHT*.

Initialisation :

```
int beg_A=2000, beg_B=3000, length_A=1500, length_B=2000;
int *vars, *coeffs;
void *solver = instanciate_solver();
```

Déclaration des variables :

```
int id_beg_A = add_int_var(solver, 0, 9999, beg_A, BEGIN_WEIGHT);
int id_beg_B = add_int_var(solver, 0, 9999, beg_B, BEGIN_WEIGHT);
int id_length_A = add_int_var(solver, 1, 10000, length_A, LENGTH_WEIGHT);
int id_length_B = add_int_var(solver, 1, 10000, length_B, LENGTH_WEIGHT);
```

Contraintes d'intégrité des boîtes :

```
*vars = (int*)(malloc(2*sizeof(int)));
*coeffs = (int*)(malloc(2*sizeof(int)));

// beg_A + length_A <= 10000
vars[0] = id_beg_A;
vars[1] = id_length_A;
coeffs[0] = 1;
coeffs[1] = 1;
add_linear_constraint(solver, vars, coeffs, 2, REL_LQ, 10000);

// beg_B + length_B <= 10000
vars[0] = id_beg_B;
vars[1] = id_length_B;
coeffs[0] = 1;
coeffs[1] = 1;
add_linear_constraint(solver, vars, coeffs, 2, REL_LQ, 10000);
```

Ajout de la contrainte de Allen :

```
vars = (int*)(malloc(3*sizeof(int)));
coeffs = (int*)(malloc(3*sizeof(int)));

// A meets B : beg_A + length_A = beg_B
```

```
vars[0] = id_beg_A;
vars[1] = id_length_A;
vars[2] = id_beg_B;
coeffs[0] = 1;
coeffs[1] = 1;
coeffs[2] = -1;
add_linear_constraint(solver, vars, coeffs, 3, REL_EQ, 0);
```

Récupération des nouvelles valeurs :

```
beg_A = get_val(solver, id_beg_A);
beg_B = get_val(solver, id_beg_B);
length_A = get_val(solver, id_length_A);
length_B = get_val(solver, id_length_B);
```

Cette manière de fonctionner, en déclarant des variables et des contraintes, et avec une résolution automatique et en temps réel du solveur, ressemble beaucoup à l'utilisation de *Cassowary* dans *Boxes*, mis à part le fait que le programme utilisateur du solveur travaille avec des identifiants et non pas des instances de variables ou de contraintes. Cette façon de voir rend son utilisation très aisée et fait complètement abstraction de la bibliothèque externe utilisée. Nous allons maintenant voir quelle bibliothèque a été choisie dans ce projet et comment l'implémentation du solveur sous forme de framework a été réalisée.

4.2 Introduction à Gecode

Comme énoncé dans la section 2.5, plusieurs pistes étaient à explorer et ma contribution consistait à implémenter un solveur basé sur *Gecode*. *GECODE* (cf. [8]) est un acronyme pour *GEneric COnstraint Development Environment*, il s'agit donc d'un ensemble de bibliothèques dédiées à la programmation par contraintes. La première version stable de Gecode date de Décembre 2005 et a bénéficié malgré son jeune âge de nombreuses corrections et évolutions puisque trois versions sont sorties depuis, dont la plus récente (1.2.0) en Juin 2006. De plus comme le code source est ouvert il est utilisé notamment par de nombreux chercheurs, qui participent eux-mêmes à l'évolution du produit. Les atouts de Gecode sont :

- sa généricité ;
- son efficacité ;
- la réactivité de ses programmeurs.

Il faut savoir qu'en contrepartie Gecode est un outil complexe dont la prise en main s'avère difficile, étant donnée que la documentation est pour le moment très sommaire et que les personnes familiarisées avec cet environnement sont encore peu nombreuses.

La figure 4.5 résume le travail désormais défini. Nous allons voir maintenant quelques points primordiaux dans la compréhension des mécanismes de Gecode qui nous intéressent dans le cadre du projet.

4.2.1 Notions d'*espace* et de *moteur de recherche* dans Gecode

Une des structures de données les plus importantes dans Gecode est la classe *Space* (classe abstraite d'où la nécessité pour le programmeur de concevoir une classe fille) qui représente un espace de travail englobant l'ensemble des variables et contraintes du système. Les différents *moteurs de recherche* (*search engines*) implémentent pour leur part les méthodes de résolution du système avec de nombreux paramètres qui permettront de s'adapter à tous types de problèmes. En pratique, un moteur de recherche permet à partir d'un espace initial de progresser par étapes successives vers la solution recherchée (réduction des domaines des variables, solution particulière parmi l'ensemble des solutions au système, etc.) en construisant à chaque fois de nouveaux espaces où les valeurs s'approchent de plus en plus du but à atteindre.

On trouve dans Gecode plusieurs moteurs de recherche qui correspondent à différents algorithmes de réduction de domaines ou de propagation de valeurs dans un graphe de contraintes, mais nous ne rentrerons pas dans le détail de ces algorithmes. Pour information, on peut tout de même citer les principaux moteurs qui sont DFS (*deep-first search*), LDS (*limited discrepancy search*) et BAB (*branch-and-bound*). Nous nous intéresserons de plus près au fonctionnement de cette dernière dans la section 4.2.4.

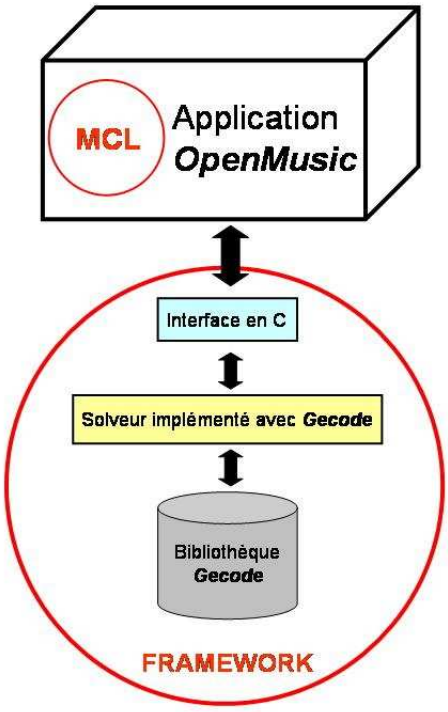

FIG. 4.5 – Un solveur basé sur *Gecode* pour *OpenMusic*

4.2.2 Variables et contraintes

Par soucis de généricité, Gecode permet de définir n'importe quel type de variables, auxquelles seront associés des *vues* et des *propagateurs*. Ces notions sont expliquées dans l'article [9] et sont fondamentales dans la conception de Gecode. Elles font appel à des mécanismes très pointus dont heureusement nous n'avons pas besoin de rentrer dans les détails dans le cadre de ce stage, puisque l'on s'intéresse uniquement à des variables entières, qui disposent d'une implémentation complète via la classe *IntVar*.

En ce qui concerne les contraintes, nous avons uniquement besoin de savoir qu'elles doivent porter sur l'ensemble des variables que l'utilisateur de Gecode aura déclaré, et qu'on les enregistre dans l'espace par une méthode *linear(...)* dont les paramètres principaux sont l'espace en question, les variables et leurs coefficients qui forment une expression linéaire, ainsi que le type de relation binaire et la valeur numérique en relation avec cette expression linéaire. On trouve également les méthodes *rel(...)* et *post(...)* dont le propos est le même mais qui offrent des commodités ou une manière de voir différente dans l'expression de la relation.

4.2.3 Réification de contraintes

La réification consiste à transposer une abstraction en un objet concret. Dans le cas d'une contrainte dans Gecode, on peut lui associer une variable booléenne qui vaut :

- 1 si la contrainte est satisfaite ;
- 0 si la contrainte n'est pas satisfaite.

On peut alors grâce à ce mécanisme établir des contraintes sur la satisfaction d'autres contraintes. Ceci se fait grâce à une surcharge de la méthode *post(...)* qui renvoie une variable booléenne. A ce moment là, la contrainte passée en paramètre ne doit plus forcément être satisfaite pour que le système soit valide. Voyons ceci sur un exemple, où l'on déclare qu'une seule parmi deux contraintes doit être vérifiée :

```
BoolExpr constaint1, constraint2;

// Initialisation des contraintes
...
Ici on initialise les contraintes qui sont des expressions booléennes du type :
<expression linéaire> <opérateur binaire> <valeur>
ce qui revient à une contrainte similaire à celles que l'on peut poster avec
la méthode linear(...)
...

// Réification des contraintes
BoolVar var_cst1 = post(my_space, constraint1);
BoolVar var_cst2 = post(my_space, constraint2);

// Construction d'un tableau avec les deux variables
BoolVarArgs expr(2);
expr[0] = var_cst1;
expr[1] = var_cst2;

// Contrainte : 1 seule des deux contraintes est satisfaite
linear(my_space, expr, IRT_EQ, 1);
```

Précisons ici que *my_space* est la variable qui représente l'espace de Gecode et *IRT_EQ* la constante correspondant à l'opérateur binaire d'égalité.

Pour terminer avec ce qui concerne la réification, il est important de savoir que l'on peut également réifier une expression linéaire avec une variable entière. Ici on n'a non pas une valeur binaire - une contrainte est soit satisfaite soit pas - mais une valeur entière qui correspond à la valeur de l'expression linéaire à l'itération courante - avec les valeurs des variables dans l'espace courant (cf. 4.2.1).

4.2.4 Le moteur BAB

Nous allons maintenant revenir sur le moteur de recherche BAB (*branch-and-bound*) car c'est celui que nous allons utiliser pour notre problème. En effet, il permet de spécifier

une *fonction objective*, c'est à dire une expression linéaire que le moteur devra au choix minimiser ou maximiser, ce qui permet d'orienter la recherche de la solution qu'on veut privilégier parmi toutes celles qui sont possibles.

Si on reprend l'exemple précédent de réification des contraintes par des variables booléennes et que l'on imagine un problème pour lequel on a un grand nombre de contraintes, pas forcément compatibles entre elles, on peut vouloir maximiser le nombre de contraintes satisfaites. Pour faire ceci, il suffit d'associer une variable entière à la somme des booléens liés aux contraintes et de dire au système de maximiser cette variable. Ici intervient la particularité du moteur BAB qui est l'appel à la méthode *Space : :constrain(Space *previous_space)* à chaque nouvelle itération du moteur. C'est dans cette méthode que le développeur utilise la notion de fonction objective, en ajoutant une nouvelle contrainte. Dans notre exemple de maximisation du nombre de contraintes satisfaites, celà donne :

```
rel(this, my_objectiveFunction, IRT_GR, (previous_space)->my_objectiveFunction);
```

où *my_objectiveFunction* est la variable représentant la fonction objective dans l'objet-espace. Grâce à cette ligne, on impose que dans le nouvel espace la valeur de la fonction objective soit supérieure à celle résultant de l'itération précédente. Lorsqu'on ne peut plus maximiser, le moteur s'arrête et on obtient l'espace "final" dans lequel les valeurs des variables correspondent à la solution optimale.

Maintenant que nous avons fait le tour des choses importantes à savoir sur Gecode, nous allons passer à la présentation du travail effectué pour aboutir à un solveur gérant les contraintes temporelles, se présentant sous la forme d'un framework pour MacOS X.

4.3 Réalisation du framework

4.3.1 Vue globale de l'architecture

Contrairement à la bibliothèque Cassowary, Gecode ne permet pas d'ajouter et d'enlever des variables ou des contraintes au système, et de récupérer la solution, pendant toute la durée de vie de la partition. Pour permettre cela, la couche d'abstraction de Gecode (cf. figure 4.5) va s'occuper de gérer les structures de données propres à la bibliothèque afin que le programme utilisant le framework puisse le faire d'une manière proche de celle dont on utilise Cassowary dans Boxes.

L'idée pour arriver à ce résultat est de mettre en place des structures de données (classes C++) afin de pouvoir retrouver l'intégralité du contexte à chaque fois que l'on doit faire appel à Gecode pour trouver une solution. Ces structures sont les suivantes :

- variables entières (classe *IntegerVariable*)
- contraintes linéaires (classe *LinearConstraint*)

Comme on l'a vu dans la section 4.2.1, il est également nécessaire de créer une classe fille de la classe abstraite *Space* de Gecode. Pour le moteur de recherche on pourra également créer une nouvelle classe. Enfin il est intéressant de définir une entité *Solver* qui comme dans Cassowary représente concrètement un solveur, et servira d'interface pour le programme utilisant le framework. Voyons maintenant précisément l'architecture de ce framework avec le diagramme de classes figure 4.6.

4.3.2 Présentation détaillée des classes

Cette partie n'est pas indispensable pour simplement utiliser le framework. Nous allons voir ici comment ont été conçues les différentes classes et comment elles interagissent avec Gecode. Comme on l'a vu dans la section 4.2, la résolution d'un système nécessite :

- une instance d'une classe dérivée de la classe *Space* pour y brancher les variables et enregistrer les contraintes ;
- un moteur de recherche configuré en fonction des besoins.

Après avoir évoqué ces deux points précis, nous examinerons comment sont conçues les structures liées aux variables et aux contraintes, celles de Gecode étant trop simplifiées pour les besoins du projet. Enfin, nous verrons comment la classe *Solver* encapsule tout celà pour une simplicité d'utilisation et une intuitivité optimale.

La classe *CustomSpace* : cette classe qui est donc dérivée de *Space* a pour rôle de contenir le tableau des variables de Gecode ainsi qu'une variable qui représente la fonction objective (cf. section 4.2.4). Elle dispose d'une méthode *addVariable(int min, int max)* qui permet simplement de créer une variable Gecode à partir d'un domaine de définition. L'appel à cette fonction renvoie un entier qui permet d'identifier la variable ainsi créée.

Plus tard, on peut accéder à celle-ci à partir de cet identifiant, en utilisant la méthode *getIntVar(int id)*. En plus de ces services qui résultent de nos choix d'implémentation, il a fallu implémenter la méthode *constrain(CustomSpace *old_space)* comme expliqué en section 4.2.4, ainsi que le constructeur par copie et la méthode de clonage de l'espace, qui sont appelées lors du processus de recherche de la solution.

La classe *SearchEngine* est celle qui symbolise le moteur de recherche de Gecode. Dans notre cas, il s'agit de la machine *branch-and-bound*.

La classe *IntegerVariable* est plus complexe car elle se détache des variables dans le sens où elles sont considérées dans Gecode. En effet, comme expliqué en section 4.1.3, on s'intéresse dans notre framework au problème précis où les variables ont une valeur initiale, et où suite à une perturbation du système on veut calculer la solution pour laquelle les écarts avec les valeurs initiales sont minimisés. C'est pourquoi une variable telle que nous la considérons dans notre framework correspond en fait à 4 variables de Gecode. Les variables possèdent également un poids et un domaine de définition, celui-ci devant être le plus précis possible pour limiter les calculs effectués par Gecode.

La classe *LinearConstraint* permet quant à elle de gérer les contraintes pendant toute la durée de vie d'une maquette car Gecode en lui-même ne gère pas les ajouts et suppressions dynamiques de contraintes, contrairement à ce que faisait Cassowary. On a donc besoin d'une structure qui garde en mémoire la relation linéaire représentant la contrainte, et permet d'ajouter celle-ci dans l'espace de Gecode lorsque celà est nécessaire.

La classe *Solver* encapsule toutes les précédentes. Elle a pour rôle d'offrir la meilleure abstraction possible de ce que l'on considère comme un *solveur*, à savoir une entité à laquelle on soumet ponctuellement des variables, des contraintes, et qui est capable à tout moment de donner les bonnes valeurs pour l'ensemble des variables. Grâce à cette classe l'utilisation du framework est très simple, l'interface C que nous avions décrit à la table 4.2 correspondant simplement aux méthodes publiques de la classe *Solver*.

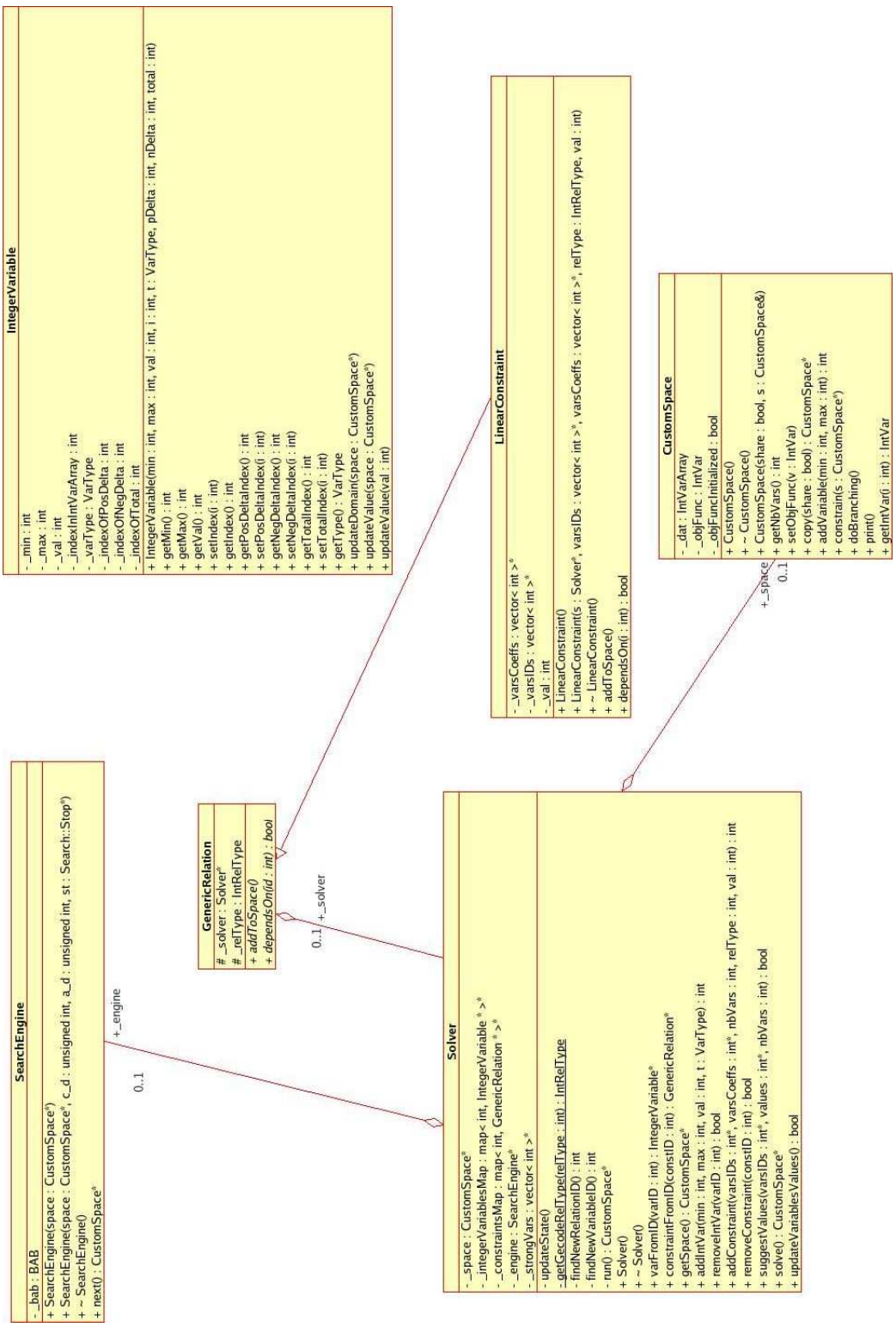

FIG. 4.6 – Diagramme de classes du framework

Chapitre 5

Utilisation du framework de gestion des contraintes et intégration dans OpenMusic

5.1 Les besoins

Les parties de code que l'on doit ajouter dans OpenMusic vont avoir deux objectifs :

- assurer le dialogue avec le framework C++ pour s'assurer que le système de contraintes est respecté à tout moment ;
- gérer l'interaction avec l'utilisateur par le biais des objets graphiques.

Nous allons voir maintenant en détail l'architecture mise en place dans OpenMusic pour notre projet, et particulièrement comment sont traduites les contraintes de Allen entre boîtes par des variables et des relation linéaires entre ces variables. Enfin, nous évoquerons rapidement la liaison des nouvelles fonctionnalités avec les structures déjà mises en place dans OpenMusic.

5.2 Les structures de données

Les structures de données créées afin de répondre aux besoins décrits dans la section précédente (5.1) sont visibles sur le schéma 5.1. Comme on peut le voir, elles sont regroupées en 3 paquetages.

Le paquetage *Gecode-Interface* qui regroupe des services bas-niveau pour le dialogue avec le framework C++ utilisant Gecode. On utilise une macro implémentée dans MCL qui permet d'utiliser des fonctions C, et il faut également pour certains cas - lorsqu'on doit passer en paramètres des tableaux de variables - allouer/désallouer des zones mémoires, chose très inhabituelle en Common Lisp qui est un langage de programmation très abstrait.

Le paquetage *CSP* pour *Constraint Satisfaction Problem*, qui constitue la couche au dessus, au niveau de l'abstraction, de celle du paquetage précédent. On y trouve les structures qui seront décrites dans la section 5.2.1 et qui servent de représentation abstraite de la configuration de la maquette.

Le paquetage *OpenMusic* est quant à lui celui qui regroupe la plupart du code du logiciel. On y redéfinit les classes déjà existantes et qui ont dû être modifiées (la maquette, les boîtes, etc.) et vraisemblablement c'est dans ce paquetage que sera inclus le code relatif à la gestion des contraintes dans la prochaine version stable d'OpenMusic.

5.2.1 Gestion des contraintes

Cet aspect consiste principalement à lier les boîtes de la maquette à leurs variables et traduire les relations de Allen entre boîtes par des relations linéaires entre variables. Pour ceci, on retrouve une certaine symétrie par rapport au solveur basé sur Gecode au niveau de l'architecture globale. Ceci s'explique par le fait qu'on doit manipuler les mêmes concepts cette fois-ci au niveau d'OpenMusic, avec cependant un point de vue différent ou plutôt complémentaire. En effet, on s'intéresse à présent à la *modélisation* d'un problème - organisation des différentes composantes d'une pièce musicale entre elles - sous forme de variables et de contraintes, et non plus à la résolution du système, c'est à dire le contraire du solveur qui lui faisait abstraction du problème concret pour se concentrer sur l'aspect mathématique de la satisfaction de systèmes de contraintes.

La classe *Solver* contient tout simplement un pointeur vers le solveur, indispensable lorsqu'on voudra utiliser n'importe quelle fonction de celui-ci (cf. 4.1.5).

La classe *ConstrainedVariable* permet de modéliser les variables qui sont de deux types : date de début ou durée. On trouve bien sûr des champs pour le domaine de définition et la valeur, ainsi que l'identifiant (cf. 4.1.5) de la variable correspondante pour le solveur. On mémorise également l'identité de la boîte de la maquette à laquelle est associée la variable.

La classe *LinearConstraint* modélise quant à elle une contrainte et ressemble beaucoup à la classe du même nom dans le solveur, avec une liste des variables - type *ConstrainedVariable* - et une des coefficients dans l'expression linéaire, le type de relation et la constante entière. Ici aussi on a également l'identifiant pour la contrainte linéaire correspondante dans le solveur.

La classe *Allen* est un niveau au dessus puisqu'elle encapsule un certain nombre de contraintes lineaires, qui ensemble représentent une contrainte de Allen entre deux boîtes. Par exemple, pour la contrainte *A before B*, on aura deux contraintes linéaires qui sont

$$debut(B) > debut(A)$$
et
$$debut(B) \geq debut(A) + duree(A)$$

Cette décomposition se fait à l'instanciation d'un objet de type *Allen*, en fonction de la nature de la relation, et c'est alors que sont créés les objets de type *LinearConstraint*. Dans les attributs de la classe *Allen* on trouve également les deux boîtes liées par la relation.

La classe *CSP* constitue une étape supplémentaire puisqu'elle encapsule l'ensemble des relations de Allen, les boîtes impliquées et le solveur. Un objet de ce type est créé pour chaque maquette et parmi ces méthodes on trouve notamment celle qui sera appelée à chaque édition et fera appel au solveur, afin de connaître la nouvelle configuration et mettre à jour l'affichage. Dans chaque objet *OMMaquette* d'OpenMusic, on a une instance de la classe *CSP*.

5.2.2 Intégration dans la maquette d'OpenMusic

Dans la maquette, on a défini un nouveau type de boîtes nommé *MagicBox*, héritant de la classe d'OpenMusic *OMTemporalBox*, pour lesquelles sont implémentées les fonctionnalités liées aux contraintes de Allen. Pour chaque boîte, on a deux objets du type *ConstrainedVariable* pour les variables de début et de durée.

Ensuite, la plupart des nouvelles méthodes sont implémentées (ou réimplémentées) dans la classe *MaquettePanel* qui gère l'objet graphique lié à une maquette d'OpenMusic. On trouve parmi les nouvelles fonctions :

– l'affichage d'une boîte de dialogue pour l'ajout d'une contrainte de Allen, déclenché lorsque le compositeur utilise le raccourci clavier affecté à cette opération ;
– une méthode *make-move-after* qui est appelée à chaque opération de *drag and drop* (déplacement d'une ou plusieurs boîtes) et déclenche la mise à jour de la configuration par l'intermédiaire du *CSP* lié à la maquette ;
– diverses méthodes liées à l'affichage qu'il n'est pas intéressant de détailler ici.

La liaison entre la couche de gestion des contraintes (cf. section 5.2.1) et l'interface graphique ayant été assurée en grande partie par Jean Bresson, ingénieur de développement à l'Ircam, nous ne rentrerons pas ici plus dans les détails.

FIG. 5.1 – Les nouvelles structures de données dans *OpenMusic*

Chapitre 6

Bilan

6.1 Avancement du projet

6.1.1 Le solveur de contraintes

Actuellement, on dispose d'un solveur opérationnel basé sur Gecode, qui permet d'enregister :

- des variables dont les propriétés sont une valeur, un domaine de définition et un poids ;
- des contraintes quelconques sur ces variables.

Par ailleurs, on peut à tout moment modifier les valeurs des variables et récupérer la solution du système de variables telle que les écarts avec les valeurs souhaitées sont minimisées, en prenant en compte les poids des variables.

Les performances du solveur dépendent du nombre de contraintes et des domaines de définition des variables. Les moteurs de recherche travaillant sur des graphes, on comprend que la complexité est exponentielle par rapport au nombre de valeurs à tester pour chaque variable, c'est à dire leur domaine de définition.

6.1.2 OpenMusic

En ce qui concerne le logiciel OpenMusic, les objectifs atteints sont les suivants :

- possibilité d'ajouter une contrainte de Allen pour chaque couple de boîtes de la maquette ;
- suppression de contraintes ;
- interdiction d'ajout de contraintes conflictuelles avec celles déjà en place ;
- mise à jour de la configuration de la maquette lors de chaque opération d'édition effectuée par la compositeur.

On retrouve les fonctionnalités qui étaient présentes dans Boxes.

6.2 Perspectives

6.2.1 Concernant OpenMusic

L'une des améliorations à apporter en priorité, au niveau de la maquette d'OpenMusic, consisterait à permettre au compositeur de sélectionner les boîtes qui peuvent être modifiées en priorité lors d'une opération d'édition. En effet, on considère que dans la plupart des cas, la conservation de la durée doit être prioritaire par rapport à une opération d'édition qui viserait à déplacer temporellement des boîtes. Prenons par exemple l'exemple de la figure 6.1, où les boîtes de gauche à droite sont chaînées successivement par la relation *before*.

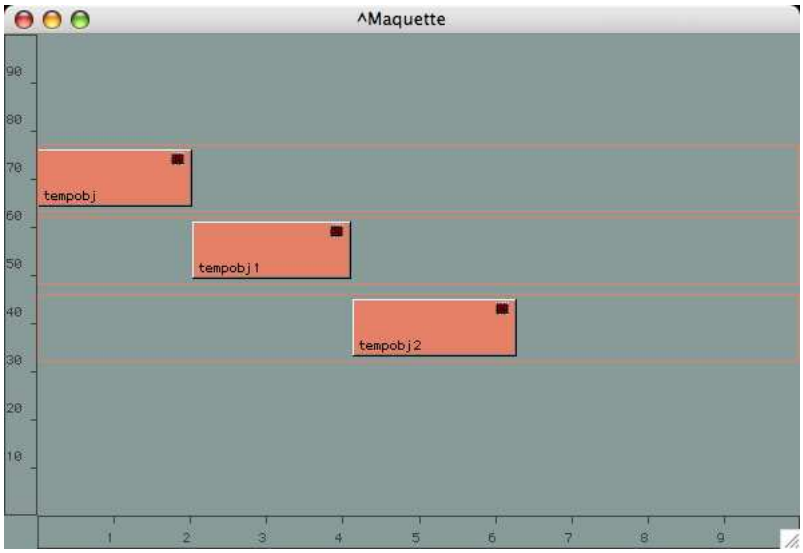

FIG. 6.1 – Un exemple de maquette

On se rend bien compte que comme la première est au tout début de la maquette et que les autres s'enchaînent sans laisser de silence, aucun déplacement de la troisième boîte ne peut être effectué vers la gauche si on veut préserver les durées. L'idée serait par exemple de pouvoir sélectionner la boîte du milieu, signifiant ainsi qu'on autorise la contraction de celle-ci lors de l'opération d'édition qui va suivre.

Une des idées à explorer rapidement serait d'essayer de mettre en place une sorte de "prévisualisation" lors des opérations d'édition, c'est à dire à chaque instant montrer à l'utilisateur sous la forme de "fantômes" des boîtes déplacées quelle serait la configuration si l'opération d'édition s'arrêtait là. On éviterait ainsi d'avoir des résultats inatendus à la fin d'un déplacement, par exemple si l'utilisateur à sélectionné par mégarde ou au contraire omis de sélectionner un boîte de la maquette. Ceci implique néanmoins de faire

travailler le solveur continuellement ce qui pourrait s'avérer trop lourd pour le système, il faut donc procéder à des essais pour juger si un tel système pourrait être viable.

6.2.2 Portage de Boxes sous *MacOS X*

Boxes est actuellement un logiciel qui tourne sous Linux, et dont la redistribution est désormais délicate à cause des problèmes liés à la bibliothèque Cassowary. Un des objectifs prioritaires du Scrime est à présent de développer une version de Boxes qui fonctionne sous MacOs X, système très utilisé chez les compositeurs férus d'informatique musicale. Un tel projet pourrait être envisagé selon les grands axes suivants :

- portage du code de Boxes dans un langage plus actuel ;
- réutilisation du solveur basé sur Gecode développé durant ce stage ;
- ajout de nouvelles fonctionnalités en s'inspirant par exemple d'OpenMusic (cf. section 6.2.1).

Le premier point est peut-être celui qui est le plus problématique car si on veut effectuer le portage vers un langage proche de STKlos - celui utilisé pour Boxes - sous Macintosh, le choix s'orientera forcément vers Common Lisp ce qui implique d'utiliser :

- soit MCL comme pour OpenMusic, ou une autre implémentation commerciale du langage ;
- soit *OpenMCL*[1] si on veut pouvoir distribuer Boxes librement, mais dans ce cas on devra développer l'interface avec un outil tel que *McCLIM*[2], l'équivalent libre de *CLIM*[3] (*Common Lisp Interface Manager* qui va de pair avec MCL, mais qui n'est pas encore arrivé en version stable.

Toutes ces considérations devront être prises en compte pour définir plus précisément les conditions dans lesquelles pourra être réalisé le projet de portage de Boxes.

[1] http ://openmcl.clozure.com
[2] http ://common-lisp.net/project/mcclim
[3] http ://www.digitool.com

Annexe A

Architecture de Boxes

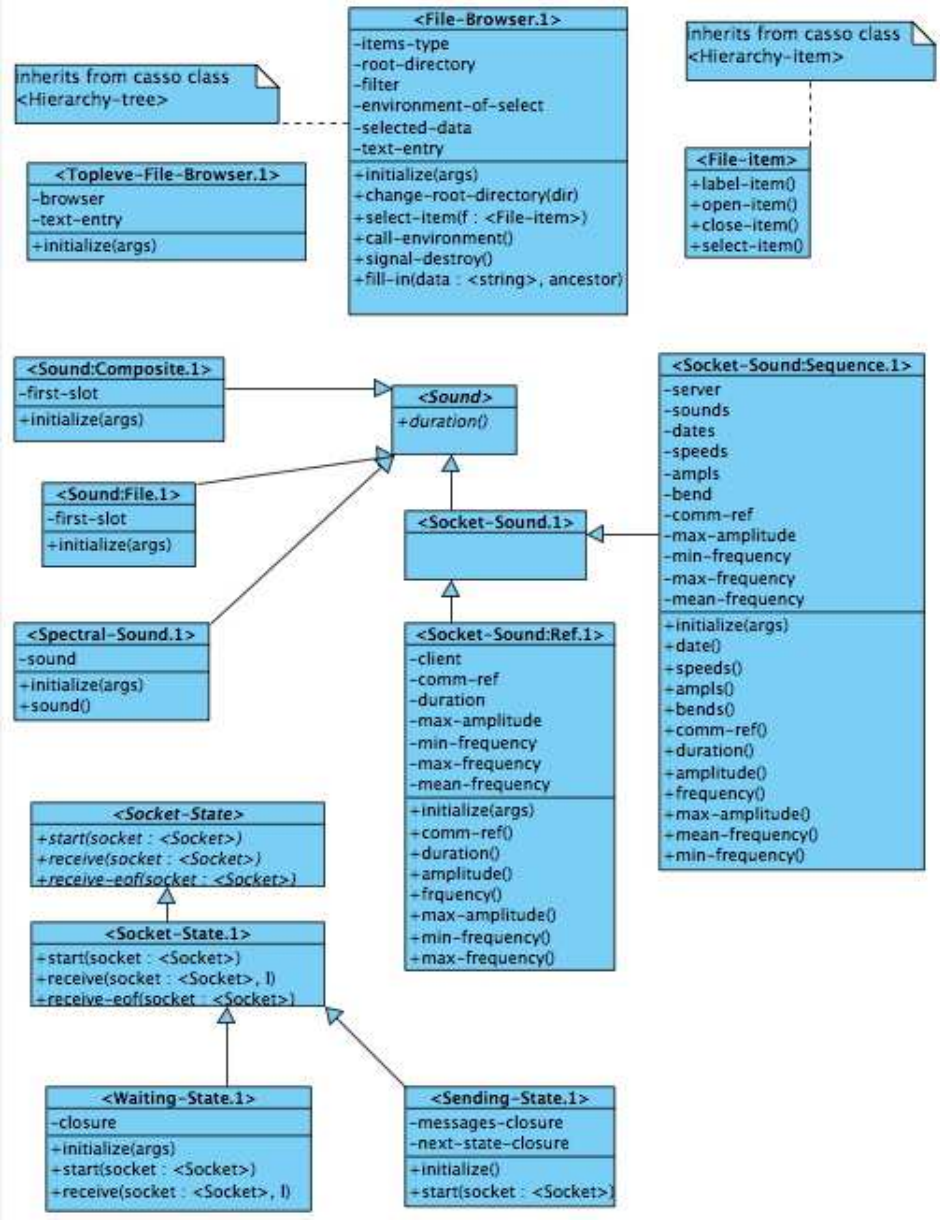

FIG. A.1 – Gestion des sons dans *Boxes*, première partie

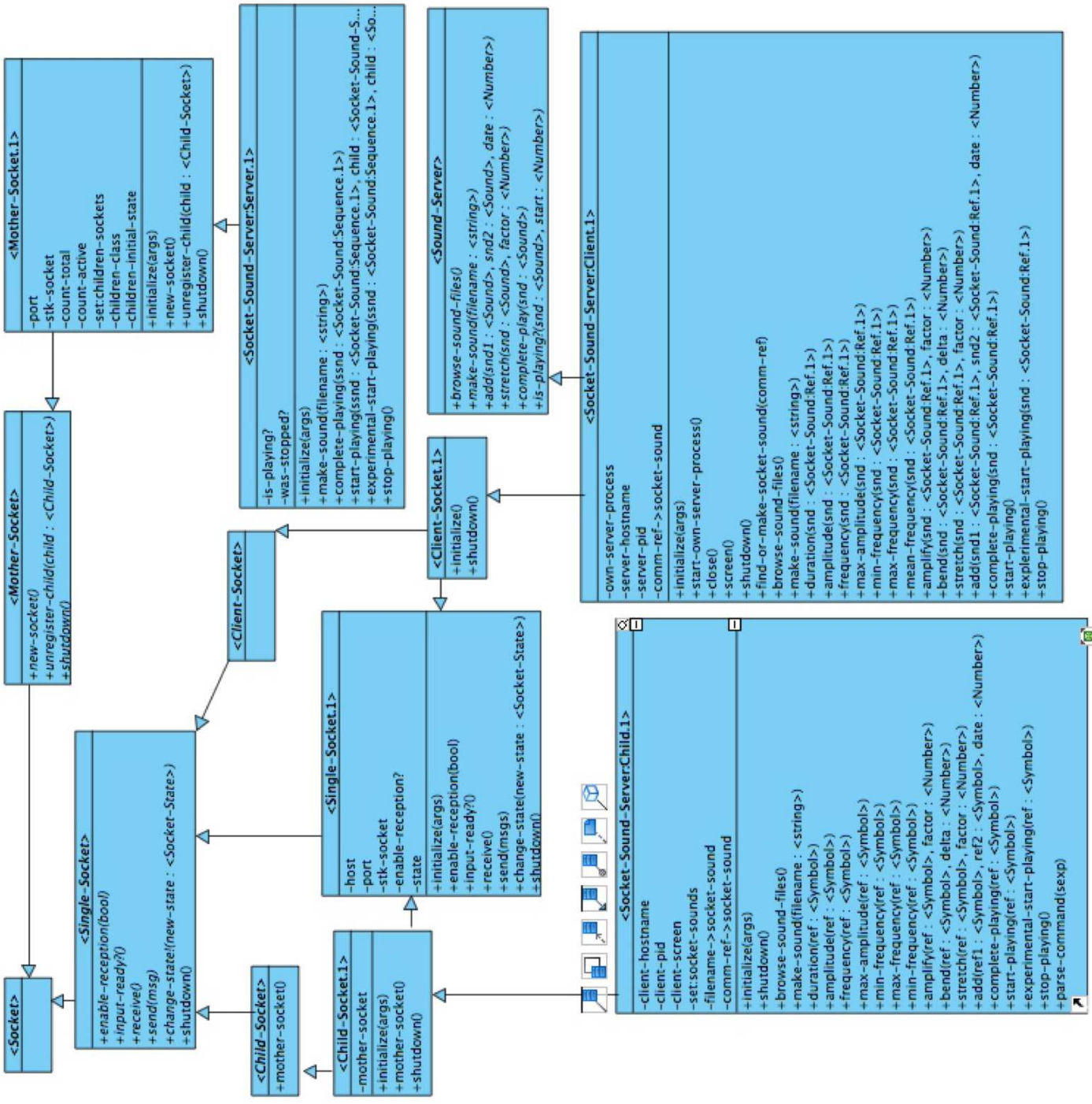

FIG. A.2 – Gestion des sons dans *Boxes*, deuxième partie

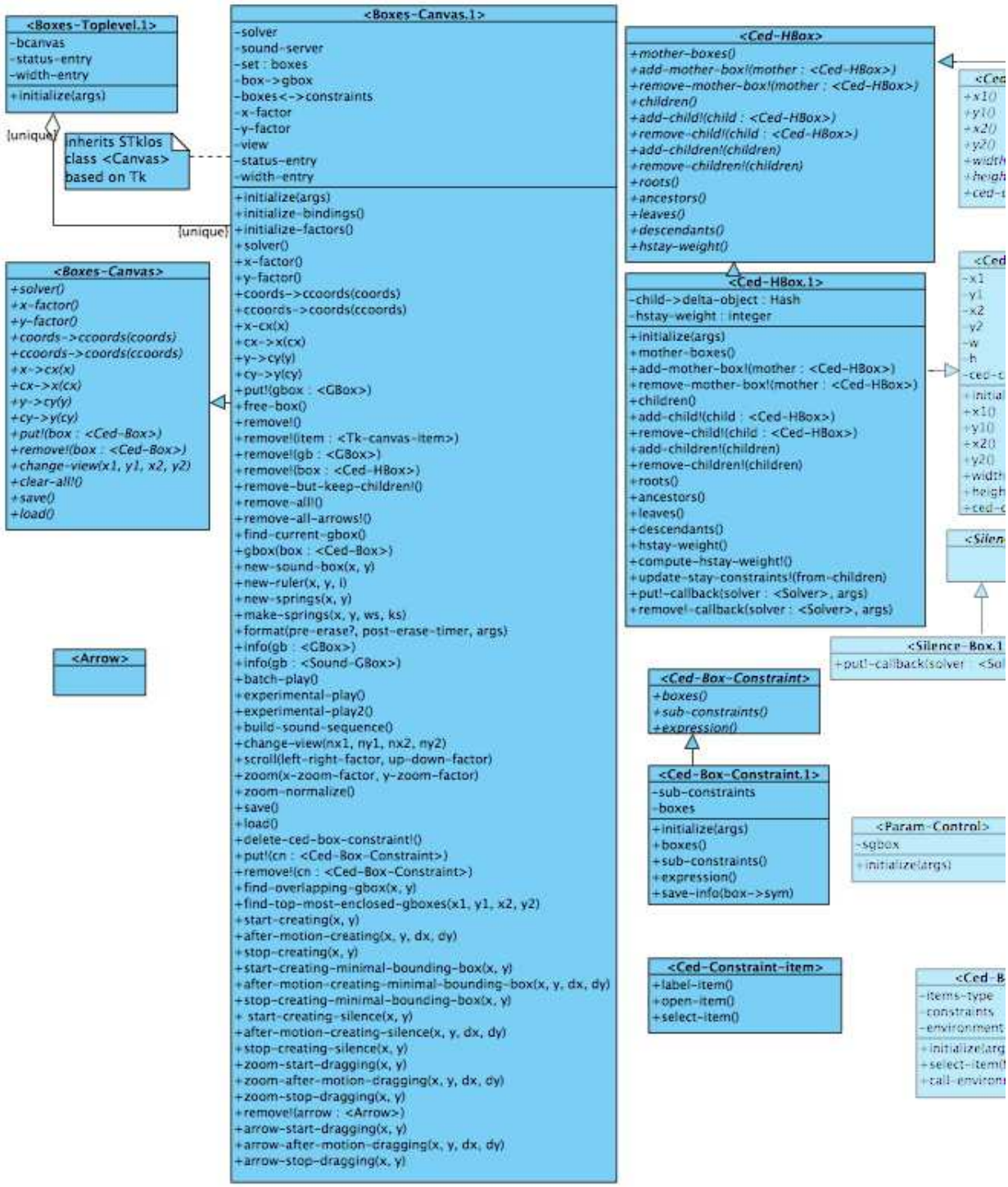

FIG. A.3 – Diagramme de classes de *Boxes*, première partie

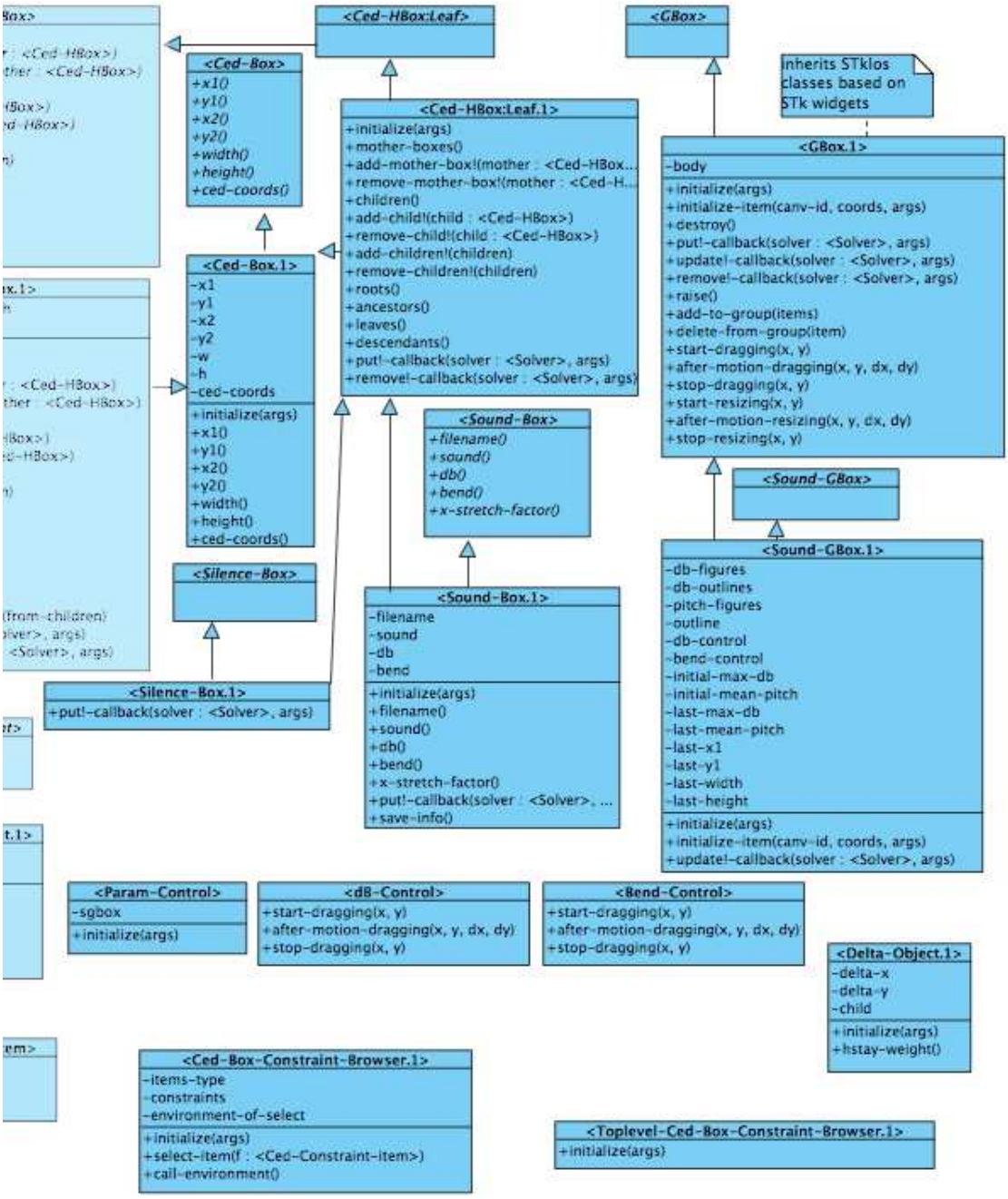

FIG. A.4 – Diagramme de classes de *Boxes*, deuxième partie

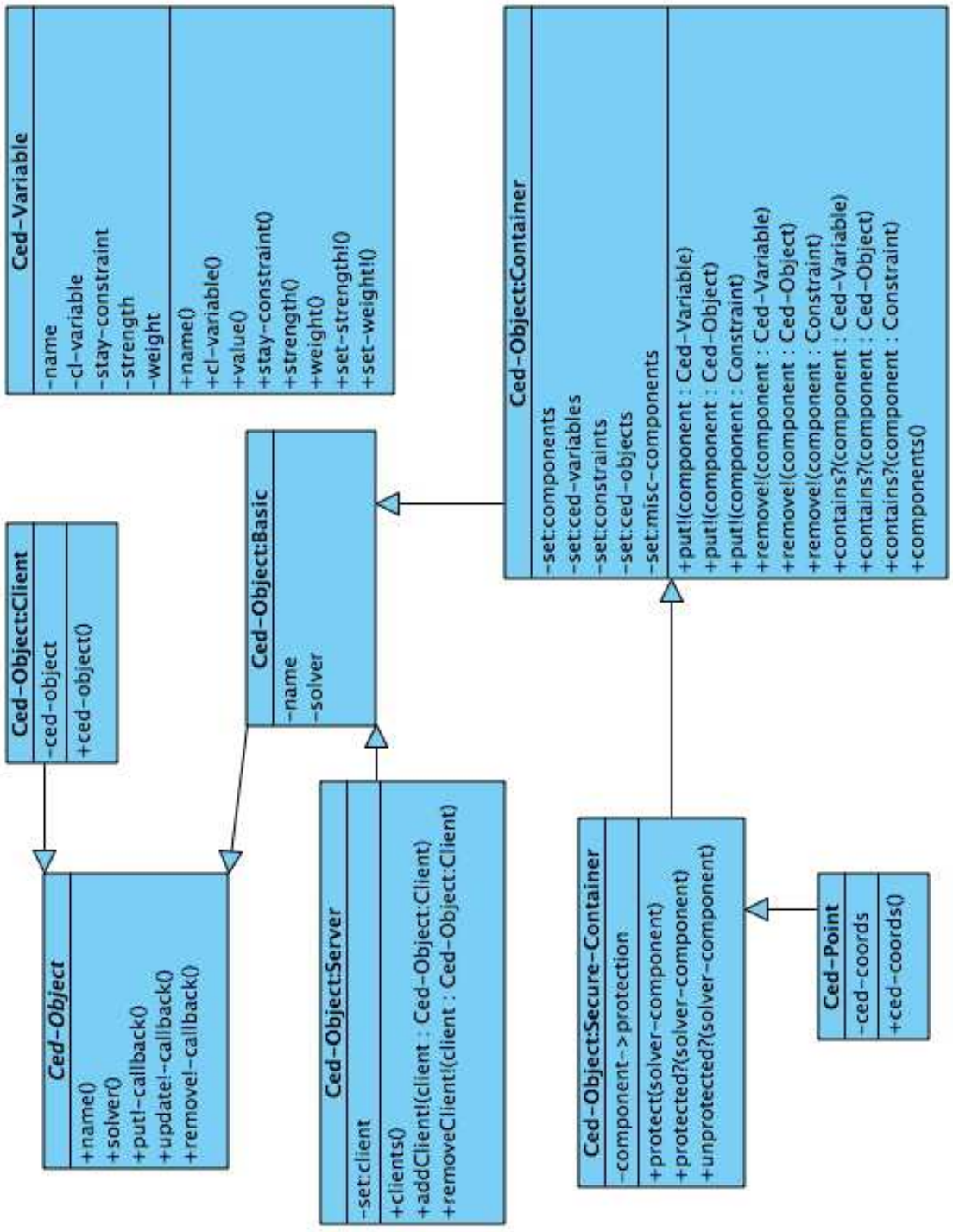

FIG. A.5 – Diagramme de classes de *Boxes*, gestion des contraintes

Bibliographie

[1] Site Boxes. http ://scrime.labri.fr/logiciels/BOXES/.

[2] Anthony Beurivé. Un logiciel de composition musicale combinant un modèle spectral, des structures hiérarchiques et des contraintes. 2000.

[3] Site STklos. http ://www.stklos.org.

[4] Site OpenMusic. http ://recherche.ircam.fr/equipes/repmus/OpenMusic/.

[5] Site Macintsh Common Lisp. http ://www.digitool.com/home.html.

[6] Site Cassowary. http ://www.cs.washington.edu/research/constraints/cassowary/.

[7] Site GTL. http ://www.infosun.fmi.uni-passau.de/GTL/.

[8] Site Gecode. http ://www.gecode.org.

[9] Christian Schulte and Guido Tack. Views and iterators for generic constraint implementations. In *Recent Advances in Constraints (2005)*, volume 3978 of *Lecture Notes in Computer Science*, pages 118–132. Springer-Verlag, 2006.

[10] Myriam Desainte-Catherine and Antoine Allombert. Specification of temporal relations between interactive events. 2005.