Reformulation de problèmes de satisfaction de contraintes basée sur des métamodèles

Raphaël Chenouard, Laurent Granvilliers, Ricardo Soto

Reformulation de problèmes de satisfaction de contraintes basée sur des métamodèles

Raphaël Chenouard[1], **Laurent Granvilliers**[1] **and Ricardo Soto**[1,2]

[1]LINA, CNRS, Université de Nantes, France

[2]Escuela de Ingeniería Informática, Pontificia Universidad Católica de Valparaíso, Chile

{raphael.chenouard,laurent.granvilliers,ricardo.soto}@univ-nantes.fr

Abstract

Un des challenges importants en programmation par contraintes est la reformulation de modèles déclaratifs en programmes exécutables permettant de calculer leurs solutions. Cette phase peut nécessiter de faire des traductions entre les langages de la programmation par contraintes, de changer la représentation des contraintes ou d'optimiser les modèles et de paramétrer les stratégies de recherche. Dans cet article, nous présentons un métamodèle pivot qui prend en compte les structures communément rencontrées dans les modèles en programmation par contraintes. Ce métamodèle comprend ainsi plusieurs types de contraintes, des structures conditionnelles ou des boucles, mais aussi les concepts de classe et de prédicat. Ce métamodèle est suffisamment général pour s'adapter aux structures de la plupart des langages, comme les langages orientés-objet ou les langages basés sur la logique, tout en en restant indépendant. Les opérations de reformulation traitent des instances du métamodèle pivot indépendamment des langages contraintes. Ainsi, les opérations définies sont génériques et s'appliquent quelques soient les langages choisis. L'espace des langages est relié à l'espace des métamodèles à l'aide d'analyseurs lexicaux et grammaticaux. Les outils de l'ingénierie logicielle peuvent alors être utiliser pour implémenter un tel cadre de transformation pour les modèles contraintes.

1 Introduction

En programmation par contraintes (PPC), l'utilisateur décrit les caractéristiques d'un problème (CSP) comme des contraintes s'appliquant à des variables. Il utilise, ensuite, un solveur pour calculer les solutions du problème qu'il a décrit. La correspondance automatique entre un modèle et un programme exécutable par un solveur est un des points clés abordés dans ce papier. L'objectif est de développer des outils logiciels intermédiaires qui sont capables de reformuler les modèles en tenant comptes des caractéristiques du solveur.

La modélisation de problèmes réels nécessite l'utilisation de langages de modélisation de haut-niveau avec des structures telles que la définition de contraintes, d'instructions de programmation et des possibilités de modularité. Récemment, plusieurs langages de modélisation sont apparus pour répondre aux besoins variés d'utilisateurs et de catégories de problèmes. D'un côté, il y a de nombreux langages de modélisation pour les problèmes combinatoires tel qu'OPL [19], Essence [5] et MiniZinc [14] ou pour les problèmes numériques comme Numerica [18] et Realpaver [9]. D'un autre côté, des librairies de résolution par contraintes ont été intégrés dans des langages de programmation informatique, comme pour ILOG Solver [15], Gecode [16] et ECL^{i}PS^e [1]. Par la suite, nous allons principalement considérer les langages de modélisation pour définir les modèles CSP. Cependant, les langages de programmation peuvent être choisis comme cible du processus de traduction. Notre objectif est de fournir un outil de traduction *many-to-many* qui peut prendre en compte des langages variés.

La plupart des langages partagent les mêmes structures, comme par exemple la définition des contraintes. D'autres structures sont plus spécifiques, comme les classes dans les langages orientés-objet ou les prédicats dans les langages logiques. Nous avons intégré l'ensemble de ces concepts dans un métamodèle, c'est-à-dire un modèle des modèles CSP. Ce métamodèle sert de pivot dans notre approche et décrit les relations existantes entre les concepts rencontrés en PPC. Il décrit, de manière abstraite, les règles de la modélisation en PPC, ce qui représente en soi une contribution importante par rapport aux travaux précédents [2] qui étaient restreints à des correspondances *one-to-many*

en partant d'un seul langage : s-COMMA.

Le processus de reformulation que nous proposons se décompose en trois étapes. Durant la première, le modèle définit par l'utilisateur est analysé grammaticalement et un modèle conforme à un métamodèle est généré. Durant la dernière étape, le programme résultant est obtenu à l'aide du processus inverse en reformulant l'information présente dans un modèle en respectant la grammaire du langage visé. Ces deux étapes établissent un pont entre l'espace des grammaires et l'espace des modèles. L'étape intermédiaire implémente des opérations de reformulation sur des modèles conformes au métamodèle pivot pour, par exemple, reformuler un modèle basé sur des entiers en un modèle booléen. L'un des points clés de notre approche est de manipuler les concepts CSP avec leur sémantique et non pas des éléments syntaxiques. Ainsi, les opérations de reformulation sont définies sans être restreintes à un langage donné.

Le travail sur la plateforme Cadmium [4] est le plus proche de ce que nous présentons. Cette plateforme utilise un langage de programmation à base de règles combinant les *Constraint Handling Rules* [7] et les opérations de réécriture de termes pour transformer des modèles contraintes formuler en Zinc ou MiniZinc. L'algorithme de réécriture fait la correspondance entre des règles et des termes pour générer de nouveaux termes, jusqu'à obtenir un ensemble de termes sur lequel plus aucune règle n'agit. Cette approche fournit une sémantique claire sur la procédure de transformation, tout en adressant des problèmes de confluence et de terminaison. La définition de métamodèles nous permet d'utiliser les outils de l'ingénierie des modèles comme ATL [12], qui est un langage général de transformation à base de règles mixant des règles déclaratives s'appliquant à des éléments typés et des structures de programmation impératives. Kermeta [13] est une autre plateforme de transformation, qui se base plus sur les concepts de la programmation orientée-objet. Un des bénéfices de l'approche dirigée par les modèles est de directement manipuler des éléments typés conformément à des concepts, qui sont reliés et hiérachisés dans des métamodèles.

La section suivante de l'article présente notre plateforme générale de transformation de modèles appliquée à la PPC. Dans la section 3, un exemple illustrant l'intérêt de l'approche est présenté en se basant sur des langages connus en PPC. La section 4 intègre une présentation du métamodèle pivot ainsi qu'une description des opérations de reformulation. Des expérimentations sur des problèmes couramment rencontrés en PPC sont présentées en section 5. Enfin, la section 6 fait le bilan de la contribution proposée dans cet article et détaille de futurs pistes de recherche.

2 Une plateforme dirigée par les modèles

Un modèle CSP est une représentation d'un problème, écrite dans un langage et ayant une structure. Notre objectif est de transformer des modèles indépendants des solveurs vers des modèles dépendants des solveurs. Cela implique de :

- changer la formulation des constraintes pour améliorer la résolution.
- traduire des modèles écrit dans des langages à haut-niveau niveau d'abstraction vers des langages bas-niveau acceptés par les solveurs ou vers des langages de programmation.
- modifier la structure des modèles par rapport aux caractéristiques des solveurs. Par exemple, il peut être nécessaire de passer d'un modèle orienté-objet à un modèle logique basé sur des prédicats.

La gestion des représentations des contraintes nécessite de spécifier des règles de transformation permettant d'obtenir une formulation équivalente ou des relaxations. La traduction des langages requiert la définition de correspondances entre les syntaxes concrètes et les concepts des métamodèles. La manipulation des structures concerne, en particulier, les concepts de modélisation abstraits comme les objects ou les prédicats. Une motivation importante de ce travaille est de séparer ces différentes tâches. En effet, l'équivalence de formulations des contraintes est indépendante d'un langage spécifique. Le processus de traduction fait alors appel à deux espaces techniques : (1) celui des grammaires (Grammar TS) qui se rapporte aux problèmes liées aux langages et leur syntaxe, (2) celui de l'ingénirie des modèles (MDE TS) qui permet la définition des concepts de modélisation et des règles de transformation à appliquer sur les modèles (cf. figure 1).

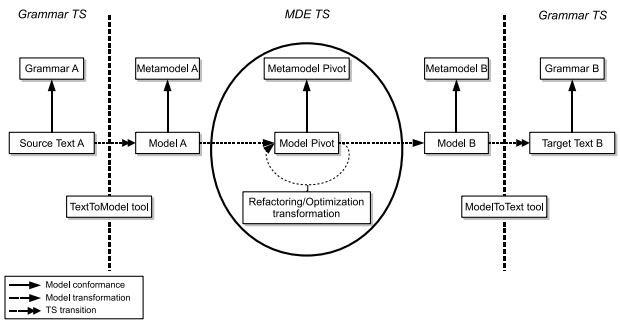

FIGURE 1 – Le processus de transformation des modèles contraintes.

Dans l'espace des grammaires, des termes sont considérés et sont organisés en fonction des règles définies dans une grammaire. Dans l'espace des modèles, les éléments considérés sont typés par rapport à un concept défini dans un métamodèle et correspondent à une instance de ce concept. Les éléments sont alors reliés entre eux à l'aide de simples références ou d'un lien de composition. Par exemple, une contrainte $x + y = z$ est issue du concept de contrainte algébrique composé d'une expression algébrique. Ainsi, il est possible de définir des éléments complexes comme des systèmes de contraintes composés d'ensemble de contraintes.

Le passage des langages aux modèles peut être implémenté à l'aide de techniques d'analyse lexical et grammaticale. Un modèle A est créé à partir d'un fichier source définit par l'utilisateur. Ce modèle doit être conforme au métamodèle du langage choisi par l'utilisateur, comme requis par l'espace des modèles. En conséquence, les métamodèles des langages (les langages de modélisation, les langages de PPC et les langages des solveurs) doivent être définis pour pouvoir établir le type des modèles. La sortie B est générée à partir d'un modèle B. Dans notre approche, ce modèle se conforme à un métamodèle de langage de solveur.

Les transformations de modèles sont définies dans l'espace des modèles. L'objectif est de transformer un modèle A correspondant au modèle écrit par l'utilisateur vers un modèle B associé à un solveur. Comme mentionné précédemment, il est nécessaire de changer la représentation des modèles et leur structure. Ce processus peut être mis en oeuvre à l'aide d'opérations transformant les concepts de A vers ceux de B. Afin d'exploiter les similarités qui peuvent exister entre les langages de modélisation PPC, nous proposons d'introduire un métamodèle intermédiaire servant de pivot. La chaine de transformation est alors composée de trois étapes : (1) le passage d'un modèle A vers le pivot, (2) l'application d'opérations de reformulation sur le modèle pivot et (3) le passage d'un modèle pivot à un modèle B. Ainsi, les étapes de reformulation nécessaire pour un langage donné peuvent être utilisées pour reformuler un autre langage.

Par la suite (section 4), nous allons présenter le métamodèle pivot et les opérations de transformation de modèles. Cependant, nous présenterons d'abord un exemple illustratif et nous discuterons des conditions préalables à la manipulation de modèles contraintes.

2.1 Un exemple illustratif

Nous avons choisi l'exemple des golfeurs sociables pour illustrer le processus de transformation. Le modèle initial est écrit à partir du langage de modélisation orienté-objet s-COMMA. Le modèle cible est un programme écrit dans le langage de programmation logique ECL^{i}PS^e. Dans ce problème, un ensemble de $n = g \times s$ joueurs veulent jouer ensemble au golf chaque semaine. Ils sont répartis dans g groupes de s golfeurs. L'objectif est de trouver un planning permettant aux joueurs de jouer pendant w semaines sans jamais rencontrer deux fois le même joueur. Les figures 2 et 3 présentent le problème écrit en s-COMMA et en ECL^{i}PS^e.

```
//Data file
1. enum Name := {a,b,c,d,e,f,g,h,i};
2. int s := 3; //size of groups
3. int w := 4; //number of weeks
4. int g := 3; //groups per week

//Model file
1. main class SocialGolfers {
2.  Week weekSched[w];
3.  constraint differentGroups {
4.   forall(w1 in 1..w)
5.    forall(w2 in w1+1..w)
6.     forall(g1 in 1..g)
7.      forall(g2 in 1..g) {
8.        card(weekSched[w1].groupSched[g1].players
          intersect
          weekSched[w2].groupSched[g2].players) <= 1;
9.      }
10.   }
11. }
12. class Group {
13.  Name set players;
14.  constraint groupSize {
15.    card(players) = s;
16.  }
17. }
18.
19. class Week {
20.  Group groupSched[g];
21.   constraint playOncePerWeek {
22.    forall(g1 in 1..g)
23.     forall(g2 in g1+1..g) {
24.      card(groupSched[g1].players
          intersect groupSched[g2].players) = 0;
25.     }
26.   }
27. }
```

FIGURE 2 – Un modèle s-COMMA pour le problème des golfeurs sociables.

Le modèle s-COMMA est divisé en deux parties : un fichier de données et un fichier définissant le modèle. Le fichier de données comporte la définition de l'énumération représentant les noms des golfeurs et trois constantes fixant les dimensions du problème (la taille des groupes, le nombre de semaines et le nombre de groupes par semaine). Le fichier du modèle est divisé en trois classes : une pour définir les groupes, une pour les semaines et la classe principale pour définir le problème global des golfeurs sociables. La classe `Group` contient l'attribut `players` qui correspond à l'ensemble des golfeurs qui doivent jouer ensemble. Chaque golfeur est identifié par un nom qui est défini dans l'énumération du fichier de données. Dans cette classe, le bloc appelé `groupSize` (lignes 14 à 16) est un bloc de contraintes, qui permet de regrouper des contraintes s'appliquant aux attributs de la classe. Ces contraintes peuvent être formulées à l'aide de boucles ou de struc-

tures conditionnelles. Ainsi, le bloc mentionné restreint la taille du groupe de golfeurs. La classe `Week` comporte un tableau d'objets de type `Group` et un bloc de contraintes `playOncePerWeek` qui empêche les joueurs de faire parti de plusieurs groupes pour une même semaine. Enfin, la classe `SocialGolfers` contient un tableau d'objets de type `Week` et un bloc de contraintes `differentGroups` qui définit le fait que chaque golfeur ne peut rencontrer qu'une fois le même golfeur tout au long des semaines considérées.

```
1. socialGolfers(L):-
2.  S $= 3,
3.  W $= 4,
4.  G $= 3,
5.
6.  intsets(WEEKSCHED_GROUPSCHED_PLAYERS,12,1,9),
7.  L = WEEKSCHED_GROUPSCHED_PLAYERS,
8.
9.  (for(W1,1,W),param(L,W,G) do
10.  (for(W2,W1+1,W),param(L,G,W1) do
11.   (for(G1,1,G),param(L,G,W1,W2) do
12.    (for(G2,1,G),param(L,G,W1,W2,G1) do
13.      V1 is G*(W1-1)+G1,nth(V2,V1,L),
14.      V3 is G*(W2-1)+G2,nth(V4,V3,L),
15.      #(V2 /\ V4, V5),V5 $=< 1
16.    )
17.   )
18.  )
19. ),
20.
21. (for(I1,1,W),param(L,S,W,G) do
22.  (for(I2,1,G),param(L,S,W,G,I1) do
23.    V6 is G*(I1-1)+I2,nth(V7,V6,L),
24.    #(V7, V8), V8 $= S
25.  )
26. ),
27.
28. (for(I1,1,W),param(L,G) do
29.  (for(G1,1,G),param(L,G,I1) do
30.   (for(G2,G1+1,G),param(L,G,I1,G1) do
31.     V9 is G*(I1-1)+G1,nth(V10,V9,L),
32.     V11 is G*(I1-1)+G2,nth(V12,V11,L),
33.     #(V10 /\ V12, 0)
34.   )
35.  )
36. ),
37.
38. label_sets(L).
```

FIGURE 3 – Le problème des golfeurs sociables exprimés dans le langage ECL^{i}PS^e.

Le modèle ECL^{i}PS^e de la figure 3 est généré après avoir appliqué le processus de transformation présenté dans la section précédente. Un seul prédicat est déclaré et contient l'ensemble du problème. Les dimensions du problème sont d'abord déclarées (lignes 2 à 4), suivies de la liste des variables (des ensembles d'entiers) `L` (lines 6 à 7) correspondant aux joueurs de golf. Enfin, trois blocs de boucles sont définis suite à la transformation des contraintes des trois classes (lignes 9 à 36). Certaines parties de ces deux modèles sont clairement similaires, ce qui se répercute au niveau des métamodèles de ces langages qui partagent un grand nombre de concepts, comme la notion de boucles ou de contraintes. Cependant, les syntaxes pour exprimer ces modèles sont très différentes. Ainsi, la déclaration de boucle `for` en ECL^{i}PS^e nécessite l'utilisation du mot-clé `param` pour définir les variables à importer dans le contexte interne à la boucle.

Le traitement des objets est plus subtile, puisque ce concept n'existe pas en ECL^{i}PS^e. Plusieurs stratégies de transformation sont possibles, comme par exemple, la définition d'un prédicat par classe d'objets [17]. Nous avons choisi une autre approche en enlevant la structure objet des problèmes. L'applatissement d'un problème nécessite d'explorer les hiérarchies d'héritage et les liens de compositions. Plusieurs problèmes peuvent être rencontrés et sont détaillés par la suite. Les attributs peuvent être modifiés de manière conséquente. Par exemple, le tableau `weekSched`, contenant des objets de type `Week` défini à la ligne 2 du fichier modèle de la figure 2, est transformé en une liste plate d'entiers : `WEEKSCHED_GROUPSCHED_PLAYERS` (ligne 6 de la figure 3). Il a aussi été nécessaire d'introduire de nouvelles boucles pour aplatir les tableaux d'objets et ainsi considérer les contraintes de tous les objets déclarés. Le dernier bloc de boucles dans le modèle ECL^{i}PS^e (lignes 27 à 35) a ainsi été généré à partir du bloc de contraintes `playOncePerWeek` du modèle s-COMMA. Il y a une boucle supplémentaire (ligne 27), puisque des instances de `Week` sont contenues dans le tableau `weekSched`. Un autre point à mentionner est la différence d'accès aux éléments des listes entre s-COMMA et ECL^{i}PS^e. Des variables locales ont été introduites et le prédicat Prolog `nth` (équivalent de la contrainte globale `element`) est utilisée dans le modèle ECL^{i}PS^e.

3 Modèle pivot

Nous avons défini un métamodèle pour définir des servant de pivot dans le processus de transformation. Ce métamodèle capture la majeur partie des concepts rencontrés dans les langages contraintes. Les modèles conforment à ce métamodèles sont manipulés à l'aide de transformation de modèles raffinant leur structure et leur formulation. Chaque transformation implémente une unique opération de raffinage ou d'optimisation des modèles.

3.1 Le métamodèle du pivot

La figure 4 présente un extrait de la structure du métamodèle pivot en s'appuyant sur un formalisme de diagramme de classe UML simplifié. Le concept racine est le concept de `Model` qui contient l'ensemble des entités. Trois concepts spécialisent la classe abstraite `ModelElement` :

- `Classifier` représente tous les types qui peuvent être utilisés pour définir des variables ou des constantes :
  - `DataType` correspond aux types de données courants qui sont utilisés en PPC : booléen, entier et réel.
  - `Enumeration` sert à définir des types symboliques, c'est-à-dire basés sur des ensembles de

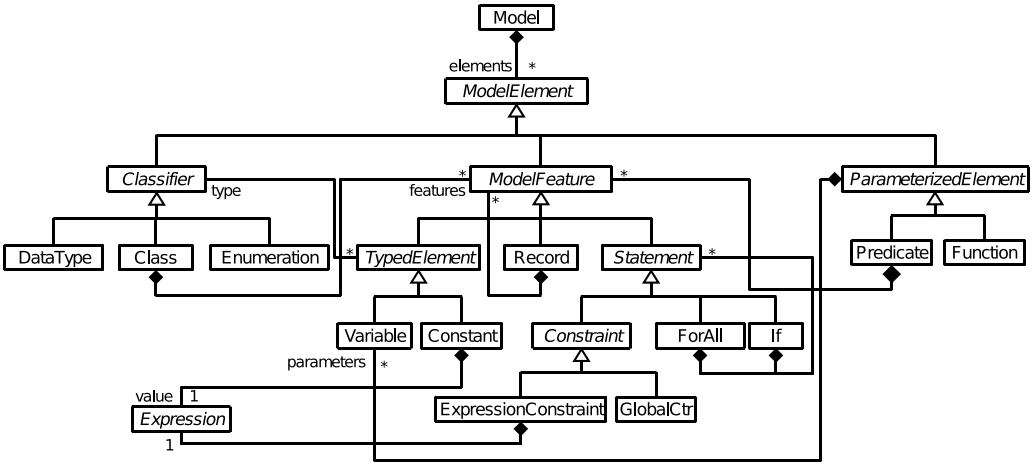

FIGURE 4 – Structures des problèmes et représentation des variables dans le métamodèle pivot.

valeurs symboliques (non détaillées ici, cf. `enum Name :={a,b,...}`, ligne 1 du fichier de données de la figure 2).

- `Class` est similaire au concept de classe défini dans les langages de programmation objet, mais dans le contexte de la PPC [17]. Ainsi une classe comporte des déclarations d'attributs (variables ou constantes), mais aussi des contraintes ou d'autres instructions encapsulant des contraintes (boucle ou conditionnelle). Ces éléments sont regroupés sous le concept abstrait de `ModelFeature`.

– `ModelFeature` correspond à l'ensemble des instances quil est possible de déclarer dans un modèle ou une classe :
  - `Record` se rapporte à des instances non-typées contenant une collection d'éléments à la manière des tuples. Pour étendre la portée des `Record`, nous les avons définis comme une composition d'instances de `ModelFeature`.
  - `TypedElement` est abstrait et regroupe l'ensemble des éléments auxquels nous associons un type (cf. l'association avec le concept `Classifier` sur la figure 4). Le concept de tableau n'est pas dissocié du concept de variable. Un tableau est uniquement défini par des tailles qui correspondant à chacune de ses dimensions. Pour plus de flexibilité, ces tailles sont définies comme des instances d'`Expression`.
    - `Variable` comporte un domaine optionnel (non détaillé ici) qui retreint les valeurs associées à son type. Trois sous-types de `Domain` sont pris en compte : les intervalles, les ensembles et les domaines définis comme des expressions.
    - `Constant` est composé d'une valeur définie comme une expression constante.
  - `Statement` est utilisé pour représenter toutes les autres caractéristiques qui peuvent apparaître dans un modèle ou une classe :
    - `Constraint` est un concept abstrait qui est spécialisé de deux manières : `ExpressionConstraint` correspond aux contraintes définies à l'aide de termes et de relations. `GlobalCtr` correspond aux contraintes globales identifiées par un nom et une liste de paramètres.
    - `ForAll` définit le concept de boucle sur des instances de `Statement`. Une variable locale est alors nécessaire pour définir les itérations de la boucle.
    - `If` permet de définir des conditions pour la prise en compte d'instances de `Statement`. Une expression définit le test booléen. Deux blocs d'éléments sont possibles, le premier étant obligatoire contrairement au deuxième.
- `ParameterizedElement` correspond aux concepts ayant une liste de paramètres et ne correpondant pas à une définition de type ni une `ModelFeature` :
  - `Predicate` permet de déclarer des prédicats logiques comme dans les modèles ECL^{i}PS^e. Les prédicats ont des paramètres et sont composés d'une séquence de `ModelFeature` (par exemple, des variables ou des contraintes).
  - `Function` permet à un utilisateur de définir des fonctions utilisées ensuite dans le modèle. Contrairement aux prédicats, une fonction ne contient qu'une seule instruction.

La notion d'expression est omniprésente en PPC. Les concepts correspondant dans le métamodèle pivot sont détaillées dans la figure 5. Ils représentent les entités apparaissant dans les expressions du premier ordre basées sur des variables, des termes, des relations et des opérateurs.

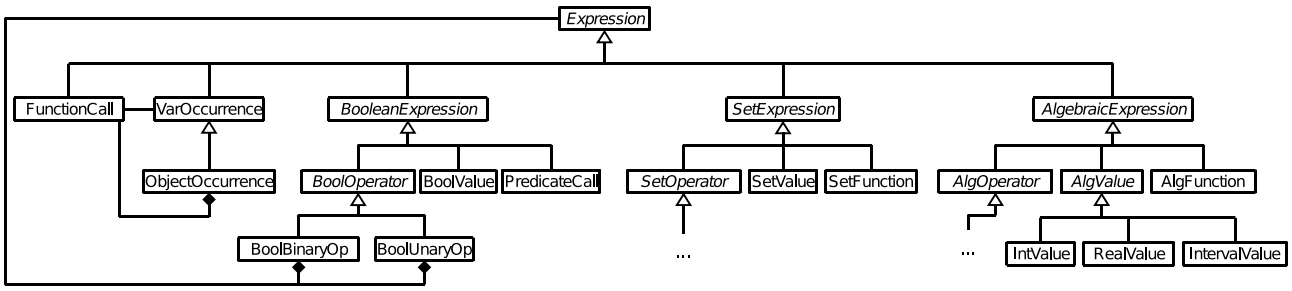

FIGURE 5 – Représentation des expressions utilisées pour définir des contraintes dans le métamodèle pivot.

Le concept `Expression` est abstrait et est spécialisé en plusieurs sortes d'expressions :

- `FunctionCall` fait référence aux fonctions définies. Il comporte une liste de paramètres définis à leur tour comme des expressions.
- `VarOccurrence` sert à représenter l'occurence des variables précédemment déclarées. Tous les éléments ayant un nom sont concernés par ce concept. Lorsque l'élément référencé est un tableau, alors la référence doit avoir un ou plusieurs attributs supplémentaires correspondant aux indices d'accès au tableau. Le concept d'`ObjectOccurrence` spécialise `VarOccurrence` en définissant l'occurence d'un attribut d'objet (par exemple `groupSched[g1].players`, ligne 8 dans la figure 2). Les occurences de variable ne sont pas classifiées par rapport à leur type pour éviter trop de redondances de concept.
- `BooleanExpression` représente globalement les expressions booléennes :
  - `BoolValue` représente les valeurs booléennes `true` et `false`.
  - `PredicateCall` est similaire à `FunctionCall` mais fait référence à des prédicats.
  - `BoolOperator` est abstrait et représente les opérateurs communéments utilisés dans les expressions booléennes : $\neg$, $\Leftrightarrow$, $\rightarrow$, and, or, $=$, $\neq$, $\leq$, $\geq$, $<$, $>$.
- `SetExpression` permet de représenter des expressions ensemblistes en regroupant les valeurs ensemblistes (par exemple $\{1, 2, 3\}$), les fonctions associées (par exemple la fonction de cardinalité) et les opérateurs usuellement utilisés (intersection, union et différence).
- `AlgebraicExpression` représente les expressions numériques sur les entiers ou les réels en utilisant aussi les fonctions et opérateurs classiques ($+$, $-$, $*$, $/$ and $\hat{}$, etc.).

Nous avons définit le métamodèle pivot pour satisfaire la plupart des besoins de modélisation rencontrés en PPC, mais aussi pour correspondre avec les métamodèles des langages PPC. De cette manière des simplifications ont été faites sur ce métamodèle, comme pour le cas des occurences de variables qui ne sont pas définies pour chaque sorte d'expression.

3.2 Reformulation du pivot

Les transformations de modèles sont implémentées commes des opérations de reformulation sur les modèles pivots. Pour plus de clareté, nous ne présentons que quelques opérations principales en utilisant un style de pseudo-code impératif, même si en pratique nous utilisons un langage dédié aux transformations de modèles. L'intérêt principal qui est apporté par la hiérarchie de concepts est le mécanisme de navigation dans les modèles. Par exemple, il est aisé de parcourir l'ensemble des variables d'une contrainte, puisque cette information est décrite dans le concept abstrait de contrainte.

3.2.1 Aplatissement des objets

Cette étape de reformulation remplace les intances correspondant à des objets (les variables dont le type est une classe) par tous les éléments définis dans la classe (variables, constantes, contraintes et autres déclarations). Pour éviter tout conflit de nom, les éléments nommés sont préfixés avec le nom de l'objet.

L'algorithme 1 présente cette étape de transformation écrite en pseudo-code. Ainsi, la fonction `ObjectRemoval` traite un modèle source en itérant sur tous ses éléments (ligne 2). Si des objets sont détectés (ligne 3), alors la fonction `flatten` est appelée et son résultat est ajouté au modèle de sortie (ligne 4). Les éléments qui ne sont pas des objets sont simplement recopiés dans le modèle de sortie (ligne 5 et 6), alors que les classes ne sont pas traitées et donc éliminées. Dans la fonction `flatten` chaque élément de l'ensemble de `ModelFeature` passé en paramètre est dupliqué et ajouté à l'ensemble résultat. Dans le cas des variables (et des constantes), leur nom est redéfini en le préfixant du nom de l'objet passé en premier paramètre (ligne 6). La figure 6 montre le résultat de cette transformation sur l'exemple des golfeurs sociables.

La reformulation des tableaux d'objets et des expressions n'est pas présenté ici pour garder un algorithme

simple. Dans le cas des tableaux d'objets (cf. fin de la section 3), nous devons transférer la définition des tailles du tableau aux attributs des objets et nous ajoutons une déclaration de boucle pour itérer sur l'ensemble des éléments de type `Statement` qui sont encapsuler dans les objets. Au sein des expressions, les occurences de variables doivent simplement être mise à jour pour référencer les nouvelles variables aplaties.

Algorithm 1 Transformation et élimination des variables objets et des classes.

objectRemoval(m : `Model`)
: `Model`

1: **let** res : `Model`
2: **for all** o **in** m.elements **do**
3: **if** is_var(o) **and** is_class(o.type) **then**
4: res.insert(**flatten**(o,o.type.features))
5: **else if not** $is\_class(o)$ **then**
6: res.elements.insert(o)
7: **end if**
8: **end for**
9: **return** res

flatten(o : `Variable`, features : **Set** of `ModelFeature`)
: **Set** of `ModelFeature`

1: **let** res : **Set** of `ModelFeature` = $\emptyset$
2: **for all** f **in** features **do**
3: **if** is_var(f) and **not** is_class(f.type) **then**
4: **let** v : `Variable`
5: v ← duplicate(f)
6: v.name = o.name + '_' + v.name
7: res.insert(v)
8: **else**
9: ...
10: **end if**
11: **end for**
12: **return** res

```
class SocialGolfers { Week weekSched[w];...}
class Week { Group groupSched[g];...}
class Group { Name set players;...}
⇒ Name set weekSched_groupSched_players[g*w];
```

FIGURE 6 – Résultat de l'application de l'aplatissement des objets sur l'exemple des golfeurs sociables présenté en s-COMMA.

3.2.2 Elimination de la contrainte Alldifferent

Les contraintes globales ne sont pas gérées par tous les solveurs, ce qui nous a poussés à définir des opérations de reformulation ou de relaxation pour ces contraintes. Nous présentons ici plusieurs opérations distinctes pour la contrainte *alldifferent*($x_1, ..., x_n$). Nous supposons que le domaine de chaque variable x_i est compris entre 1 et n pour simplifier la définition des deux dernières transformations présentées. Nous présentons ainsi trois possibilités de reformulation pour cette contrainte globale, qui est alors remplacée par :

– Un ensemble d'inégalités (voir Algorithme 2). Pour chaque combinaison de variables (lignes 2 et 3), une contrainte est générée et ajoutée au résultat (ligne 6).

Algorithm 2 Transformation de alldifferent en un ensemble d'inégalités

AllDiffToDisequalities(c : `GlobalConstraint`)
: **Set** of `Constraint`

1: **let** res : **Set** of `Constraint` = $\emptyset$
2: **for all** i **in** 1..c.parameters.size() **do**
3: **for all** j **in** $i+1$..c.parameters.size() **do**
4: let x : `Variable` = c.parameter[i]
5: let y : `Variable` = c.parameter[j]
6: res.insert(**new** `Constraint`(x $\neq$ y))
7: **end for**
8: **end for**
9: **return** res

– Une relaxation (voir Algorithme 3. Seulement une contrainte est générée (ligne 3). Elle calcule la somme des valeurs des variables, qui doit alors être égale à $n(n+1)/2$.

Algorithm 3 Génération d'une relaxation de alldifferent

AllDiffToRelaxation(c : `GlobalConstraint`)
: `Constraint`

1: **let** n : Integer = c.parameters.size()
2: **let** sum : `Expression` = $\sum_{i=1}^{n}$ c.parameters[i]
3: **return new** `Constraint`(sum = n(n+1)/2)

– Une version booléenne (voir Algorithme 4). Dans ce cas, nous définissons une matrice de variables booléennes (lignes 2 à 4), où $b[i, j]$ est vraie lorsque x_i a pour valeur j. La ligne 7 vérifie qu'une seule valeur est définie pour chaque variable. La ligne 10 assure ensuite que deux variables ont une valeur différente.

4 Expérimentations

L'architecture présentée a été implémentée avec trois outils et langages issus de l'ingénierie des modèles : (1) KM3 [10] est un langage permettant de définir des métamodèles, (2) ATL [12] est une langage déclaratif à base de règles pour décrire des transformations de modèles et (3) TCS [11] est un langage déclaratif permettant de lier une grammaire et un métamodèle. Ces outils nous permettent de choisir les opérations de reformulation à appliquer sur

Algorithm 4 Reformulation de alldifferent en modèle booléen

AllDiffToBoolean(c : `GlobalConstraint`)
: **Set** of `ModelFeature`

1: **let** res : **Set** of `Constraint` = $\emptyset$
2: **let** n : Integer = c.parameters.size()
3: **let** m : Integer = card(c.parameters.domain)
4: **let** b[n,m] : Boolean
5: res.insert(b)
6: **for** i **in** $1..n$ **do**
7: res.insert(**new** `Constraint`($\sum_{j=1}^{m}$ b[i,j] = 1))
8: **end for**
9: **for** j **in** $1..m$ **do**
10: res.insert(**new** `Constraint`($\sum_{i=1}^{n}$ b[i,j] = 1))
11: **end for**

les modèles du pivot. Nous pouvons ainsi choisir de garder ou de supprimer la structure suivant le métamodèle cible.

Nous avons mené une série de tests pour analyser les performances de notre approche. Nous avons utilisé cinq problèmes connus en PPC : les golfeurs sociables (Golfers), la conception d'un moteur (Engine), Send+More=Money, les mariages stables (Marriage) et les 10-reines (10-Q). La première série d'expérimentation présente les performances en termes de temps de traduction, alors que la deuxième montre que la génération automatique de modèle n'impacte pas négativement les temps de résolution. Les test ont été effectués sur un ordinateur à 2.66 Ghz avec 2 Go de mémoire ram sous Ubuntu.

Problems	sC Lines	s-to-P (s)	Object (s)	Enum (s)	P-to-E (s)	Total (s)	Ecl Lines
Golfers	31	0.276	0.340	0.080	0.075	0.771	37
Engine	112	0.292	0.641	0.146	0.087	1.166	78
Send	16	0.289	0.273	-	0.089	0.651	21
Marriage	46	0.330	0.469	0.085	0.067	0.951	26
10-Q	14	0.279	0.252	-	0.033	0.564	12

TABLE 1 – Temps obtenus pour une chaine de transformation complète sur plusieurs exemples classiques.

Dans ce premier test, nous présentons une chaine de transformation de s-COMMA (sC) vers ECL^{i}PS^e (Ecl). La table 1 présente les résultats obtenus, où la première colonne correspond aux noms des problèmes et la deuxième le nombre de lignes des fichiers sources. Les colonnes suivantes présentent les temps mesurés pour appliquer les étapes atomiques de la chaine de transformation en secondes : de s-COMMA vers le Pivot (s-to-P), l'aplatissement d'objets (Object), l'élimination des énumérations (Enum) et la transformation du pivot vers ECL^{i}PS^e (P-to-E). La colonne suivante montre le temps total de la chaine de transformation et la dernière colonne représente le nombre de lignes des fichiers ECL^{i}PS^e générés.

Les résultats obtenus montrent que les phases de traitement depuis ou vers les langages PPC (s-to-P et P-to-E) sont rapides, même si on peut constater que les problèmes considérés sont assez courts (au maximum 112 lignes). La transformation de s-COMMA vers le pivot est plus lente que la transformation du pivot vers ECL^{i}PS^e. Cela s'explique par les phases de reformulation sur le pivot qui réduisent le nombre d'éléments à traiter pour cette dernière opération. La phase d'aplatissement des objets est la plus coûteuse. C'est l'exemple du moteur qui obtient ainsi les temps de transformation les plus long, car il comporte plusieurs objets encapsulés les uns dans les autres. Sur l'ensemble des phases de transformation, nous pensons que les temps obtenus sont raisonnables, ce qui permet de traiter aisément des problèmes de plus grande taille.

Problems	Native		Generated		Generated (Flat)	
	solve(s)	Lines	solve(s)	Lines	solve(s)	Lines
Golfers	0.21	28	0.21	31	0.22	276
Marriage	0.01	42	0.01	46	0.01	226
20-Q	4.63	11	4.65	12	5.02	1162
28-Q	80.73	11	80.78	12	87.73	2284

TABLE 2 – Temps de résolution et taille des modèles pour des fichiers écrits à la main ou générés automatiquement.

Dans la deuxième série de tests, nous comparons les fichiers ECL^{i}PS^e générés automatiquement par notre approche avec les fichiers ECL^{i}PS^e des mêmes problèmes, mais écrits à la main (cf. table 2). Nous considérons le temps de résolution et le nombre de lignes de chaque fichier. Les résultats des modèles écrits à la main sont d'abord présentés. Ensuite, les fichiers générés en conservant la structure des boucles. Enfin, nous considérons les fichiers générés pour lesquels les boucles ont été déroulées (Flat). Pour les modèles avec des boucles le nombre de lignes et les temps de calculs sont comparables aux modèles écrits à la mains. En ce qui concerne les modèles déroulés, la taille des modèles augmente évidemment très significativement. Cependant, le temps de résolution est peu différent, sauf pour les modèles de plus grande taille (20-Q et 28-Q de 0,4 et 7 secondes). Cet impact négatif peut être attribué à l'algorithme de propagation incrémentale qui est généralement utilisé dans les systèmes CLP. Nous supposons ainsi qu'une propagation est lancée à chaque fois qu'une contrainte est ajoutée au store. Par contre, si une boucle est déclarée, l'ensemble de contraintes induites sont ajoutées d'un bloc et ainsi il n'y a qu'une seule étape de propagation. Lorsque les boucles sont déroulées, alors une phase de propagation est lancée après chaque ajout de contrainte, ce qui explique les pertes de temps entre les deux versions générées des mêmes problèmes. Cet impact négatif sur le temps de résolution montre qu'il peut être préférable de garder la structure des modèles lorsqu'elle est gérée par le solveur visé, plutot que de générer des modèles complètement aplatis.

5 Travaux connexes

La transformation de modèles est un sujet de recherche récent en PPC. Peu d'approches de transformation des modèles PPC ont été proposées. Les structures indépendantes des solveurs disponibles sont les plus proches de notre travail, comme par exemple MiniZinc (and Zinc), Essence and s-COMMA.

MiniZinc est un langage de modélisation haut-niveau qui permet d'obtenir des programmes ECL^{i}PS^e et Gecode. Les transformations sont implémentées dans Cadmium, qui utilise des techniques basées sur la réécriture de termes. Le processus de traduction implique un modèle intermédiaire, où les structures de MiniZinc sont remplacées par celles supportées par tous les solveurs. Cela facilite ensuite la traduction vers les solveurs, tout en optimisant parfois le modèle obtenu.

Essence est une autre langage impliquant des transformations indépendantes d'un solveur. Des modèles pour ECL^{i}PS^e et Minion [8] peuvent être générés. Le processus de transformation est implémenté au sein du système Conjure [6], qui prend en entrée des spécificaiton Essence pour les raffiner dans un modèle intermédiaire Essence'. La transformation de Essence' vers les solveurs se fait alors à l'aide de traducteurs écrits à la main.

s-COMMA est un langage orienté-objet basée sur une plateforme de modélisation indépendante de tout solveur. Des programmes pour ECL^{i}PS^e, Gecode/J, RealPaver et GNU Prolog [3] peuvent être générés. Un langage intermédiaire (Flat s-COMMA) est aussi utilisé pour faciliter les traductions vers les solveurs. Des traducteurs écrits à la main et utilisant des outils de l'ingénierie des modèles sont disponibles.

Notre approche peut être vue comme une évolution naturelle dela plateforme s-COMMA, tout en apportant deux avantages principaux :

- Dans les approches mentionnées précédemment, seul un langage de modélisation peut être utilisé comme source du processus de transformation, ce qui n'est pas le cas dans notre approche. Nous pensons que cela apporte plus de flexibilité et de liberté pour l'utilisateur.
- Dans les processus de transformation de s-COMMA, MiniZinc, et Essence, toutes les phases de reformulation sont toujours appliquées. Il en résulte des modèles exécutables généralement très différents des modèles initiaux. De notre côté, nous cherchons à générer des modèles optimisés tout en maintenant autant que possible la structure initiale des modèles sources. Nous pensons, que transférer les caractéristiques des modèles sources aux modèles cibles, améliore ainsi la lisibilité et la compréhensibilité des modèles générés.

6 Conclusion et travaux futurs

Dans cet article, nous avons présenté un nouveau cadre pour la transformation des modèles en PPC. Ce cadre est basé sur une approche dirigée par les modèles avec un métamodèle pivot qui nous permet d'être indépendant et flexible tout en s'adaptant aux différents langages de la PPC. La chaine de transformation implique trois principales étapes : (1) du langage source vers le pivot, (2) des opérations de reformulation sur le pivot et (3) du pivot vers le langage cible. Contrairement aux autres approches, les chaines de transformation sont modulaires et le travail de reformulation/optimisation est effectué de manière générique sur le pivot. Les transformations sont plus simples et, par conséquent, l'intégration de nouveaux langages et de nouvelles étapes de reformulation nécessite moins d'efforts.

Nous comptons prochainement intégrer de nouveaux langages à notre approche. La définition de nouvelles opérations de reformulation et d'optimisation des modèles est aussi une de nos priorités. Une autre piste de recherche que nous voulons développer concerne la gestion de chaines complexes de transformation. L'ordre d'application des transformations et leur choix peuvent être automatisés, mais cela nécessite d'analyser statiquement les transformations et les modèles à considérer.

Références

[1] Krzysztof R. Apt and Mark Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.

[2] R. Chenouard, L. Granvilliers, and R. Soto. Model-Driven Constraint Programming. In *ACM SIGPLAN PPDP*, pages 236–246, 2008.

[3] D. Diaz and P. Codognet. The GNU Prolog System and its Implementation. In *SAC 2000*, pages 728–732, 2000.

[4] Gregory J. Duck, Peter J. Stuckey, and Sebastian Brand. ACD Term Rewriting. In *ICLP*, pages 117–131, 2006.

[5] A. M. Frisch, M. Grum, C. Jefferson, B. Martínez Hernández, and I. Miguel. The Design of ESSENCE : A Constraint Language for Specifying Combinatorial Problems. In *IJCAI*, pages 80–87, 2007.

[6] A.M. Frisch, C. Jefferson, B. Martinez-Hernandez, and I. Miguel. The Rules of Constraint Modelling. In *IJCAI*, pages 109–116, 2005.

[7] T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, June 2009. to appear.

[8] I. P. Gent, C. Jefferson, and I. Miguel. Minion : A Fast Scalable Constraint Solver. In *ECAI*, pages 98–102, 2006.

[9] L. Granvilliers and F. Benhamou. Algorithm 852 : RealPaver : an Interval Solver Using Constraint Satisfaction Techniques. *ACM Trans. Math. Softw.*, 32(1) :138–156, 2006.

[10] F. Jouault and J. Bézivin. KM3 : A DSL for Metamodel Specification. In *FMOODS*, LNCS 4037, pages 171–185, 2006.

[11] F. Jouault, J. Bézivin, and I. Kurtev. TCS : a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *ACM GPCE*, pages 249–254, 2006.

[12] Ivan Kurtev, Klaas van den Berg, and Frédéric Jouault. Rule-based Modularization in Model Transformation Languages Illustrated with ATL. *Science of Computer Programming,*, 68(3) :138–154, 2007.

[13] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML*, LNCS 3713, pages 264–278, Montego Bay, Jamaica, October 2005.

[14] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc : Towards A Standard CP Modelling Language. In *CP*, LNCS 4741, pages 529–543, 2007.

[15] J.F. Puget. A C++ Implementation of CLP. In *SPICIS*, 1994.

[16] C. Schulte and G. Tack. Views and Iterators for Generic Constraint Implementations. In *Recent Advances in Constraints (2005)*, LNCS 3978, pages 118–132, 2006.

[17] R. Soto and L. Granvilliers. The Design of COMMA : An Extensible Framework for Mapping Constrained Objects to Native Solver Models. In *IEEE ICTAI*, pages 243–250, 2007.

[18] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica : a Modeling Language for Global Optimization*. MIT Press, 1997.

[19] P. Van Hentenryck, L. Michel, L. Perron, and J.-C. Régin. Constraint Programming in OPL. In *PPDP*, LNCS 1702, pages 98–116, 1999.