Des règles aux contraintes avec le langage de modélisation Rules2CP

François Fages, Julien Martin

Projet Contraintes, INRIA Rocquencourt,
BP105, 78153 Le Chesnay Cedex, France.
http://contraintes.inria.fr
{Francois.Fages,Julien.Martin}@inria.fr

Résumé

Dans cet article, nous montrons que le paradigme de représentation des connaissances règles métier, largement utilisé dans l'industrie, peut être développé comme un langage de modélisation pour la programmation par contraintes. Nous présentons un langage de modélisation général à base de règles, nommé Rules2CP, et décrivons sa compilation vers des programmes de contraintes sur les domaines finis avec contraintes réifiées et contraintes globales, basée sur la réécriture de termes et l'évaluation partielle. Nous prouvons la confluence de ces transformations et fournissons une borne de complexité sur la taille des programmes de contraintes générés. L'expressivité de Rules2CP est illustrée avec une librairie complète de modélisation des problèmes de placement, appellée PKML, qui, en plus des problèmes purs Bin Packing et Bin Design, gère des règles de bon sens portant sur des contraintes de poids, de gravité et d'équilibre, ainsi que des règles métiers spécifiques compilées efficacement en des programmes de contraintes.

1 Introduction

D'un point de vue langage de programmation, un trait criant de la programmation par contraintes est sa déclarativité pour exprimer des problèmes combinatoires, décrivant seulement le "quoi" et non le "comment", ainsi que son efficacité à résoudre des instances de grandes tailles de tels types de problèmes dans de nombreux cas. Toutefois, du point de vue d'un utilisateur néophyte, un programme de contraintes n'est pas aussi déclaratif que l'on le voudrait, et les langages de programmation par contraintes sont en fait très difficiles à utiliser par les non-spécialistes, en dehors des problèmes déjà traités. Cette difficulté reconnue a été présentée comme un défi majeur pour la communauté de la programmation par contraintes, et a motivé des recherches sur des langages de modélisation plus déclaratifs, comme OPL [21] et Zinc [17, 8].

Dans l'industrie, l'approche règles métier pour la représentation des connaissances trouve un large public en vertu de la propriété d'indépendance des règles que l'on peut introduire, vérifier et modifier indépendamment les unes des autres et de toute interprétation procédurale par un moteur de règles [11]. Cela procure une méthode de représentation des connaissances séduisante pour faire évoluer rapidement des ensembles de règles et de contraintes, et maintenir facilement des systèmes d'information à jour.

Dans cet article, nous montrons que le paradigme règles métier pour la représentation des connaissances peut être développé pour constituer un langage de modélisation pour la programmation par contraintes. Nous présentons un langage de modélisation à base de règles général nommé Rules2CP.

A la différence des règles générales action-condition, aussi appellées règles de production dans la communauté des systèmes experts, les règles Rules2CP sont restreintes à des *règles logiques*, avec une tête et pas d'action impérative, et où des quantificateurs bornés sont utilisés pour représenter des conditions complexes. Elles sont conformes au manifeste sur les règles métier [11], et en particulier à l'indépendance vis-à-vis de tout système d'interprétation par un moteur de règles. Cela est concrètement démontré dans Rules2CP par leur compilation vers des programmes de contraintes utilisant une représentation complètement différente.

Dans la section suivante, nous présentons le langage Rules2CP et montrons comment les stratégies et les heuristiques peuvent être spécifiées déclarativement.

La Sec. 2 décrit la compilation des modèles Rules2CP vers des programmes de contraintes sur les domaines finis avec contraintes réifiées, basée sur la réécriture de termes et l'évaluation partielle. Nous prouvons la confluence de ces transformations, qui montre que le programme de contraintes généré ne dépend pas de l'ordre de l'application des réécritures. De plus, nous fournissons une borne de complexité de la taille des programmes générés.

La Sec. 4 illustre l'expressivité et l'efficacité de cette approche avec une librairie Rules2CP, appellée PKML, développée dans le projet européen Net-WMS[1] pour traiter les problèmes de Bin Packing non-purs et de taille réelle venant de l'industrie automobile.

Enfin, la dernière section compare Rules2CP aux travaux sur les règles métier, les langages de modélisation OPL et Zinc, la programmation logique avec contraintes et les systèmes de réécriture.

2 Le langage Rules2CP

2.1 Syntaxe

La syntaxe du langage Rules2CP est donnée par la grammaire formelle suivante :

statement	*: :=*	`import` *name*`.` | *head* `=` *expr*`.`
	|	*head* `-->` *fol*`.` | `?` *fol*`.`
head	*: :=*	*ident* | *ident(var,...,var)*
fol	*: :=*	*varbool* | *expr relop expr*
	|	*expr* `in` *expr* | *name*
	|	*name(expr,...,expr)*
	|	`not` *fol* | *fol logop fol*
	|	`forall`*(var,expr,fol)*
	|	`exists`*(var,expr,fol)*
	|	`let`*(var,expr,fol)*
	|	`aggregate`*(var,expr,logop,fol,fol)*
expr	*: :=*	*varint* | *fol* | *string* | `[` *enum* `]`
	|	`{`*ident* `=` *expr,...,ident*`=` *expr*`}`
	|	*name* | *name(expr,...,expr)*
	|	*expr op expr*
	|	`aggregate`*(var,expr,op,expr,expr)*
	|	`map`*(var,expr,expr)*
enum	*::=*	*enum* `,` *enum* | *expr* | *expr* `..` *expr*
varint	*: :=*	*var* | *integer*
varbool	*: :=*	*var* | `0` | `1`
op	*: :=*	`+` | `-` | `*` | `/` | `min` | `max`
relop	*: :=*	`<` | `=<` | `=` | `#` | `>=` | `>`
logop	*: :=*	`and` | `or`
	|	`implies` | `equiv` | `xor`
name	*: :=*	*ident* | *name* `:`*ident*

où *ident* est un mot commençant par une lettre minuscule ou n'importe quel mot mis entre apostrophes, *name* est un identificateur qui peut être préfixé par d'autres identificateurs comme noms de modules, et une *variable* est un mot commençant soit par une lettre majuscule, soit par un tiret bas (_). L'ensemble des *variables libres* dans une expression E, noté $V(E)$, est l'ensemble des variables apparaissant dans E et qui ne sont liées par aucun des opérateurs `forall, exists, let, map` ou `aggregate`. La *taille* d'une expression ou formule est le nombre de noeuds dans sa représentation sous forme d'arbre. Les seules expressions qui construisent des listes sont les expressions d'intervalle *expr .. expr*, l'opérateur `map`(var,expr,expr) et le prédicat `variables`(expr).

Parce qu'il est important de nommer les objets dans Rules2CP, le langage inclut un système simple de modules, qui préfixe les noms avec des noms de modules, similairement à [12]. Dans un fichier Rules2CP, l'ordre des déclarations n'importe pas. Les définitions récursives et les définitions surchargées sont interdites. Dans une règle, *L*`-->`*R*, nous supposons $V(R) \subseteq V(L)$, alors que dans une declaration, *H*`=`*E*, les variables introduites, dans $V(E) \setminus V(H)$, representent les inconnues du problème. Une expression *expr* peut être une formule du première ordre *fol*, alors considérée comme un entier 0/1. Cette coercition usuelle entre booléens et entiers, appellée *réification*, procure une expressivité remarquable [20]. L'opérateur d'agrégation ne peut être défini dans la logique du premier-ordre sans récursivité et constitue une primitive de Rules2CP. Cet opérateur applique itérativement un opérateur binaire à une liste d'arguments. Par exemple, le produit d'éléments d'une liste est défini par `product(L)=aggregate(X,L,*,1,X)`. Les quantificateurs sont éliminés à la compilation, et non pas traités à l'execution comme c'est le cas dans l'approche classique QCSPs [10].

Les seules structures de données de Rules2CP sont les entiers, les chaînes de caractères, les listes énumérées et les enregistrements. Les listes d'expressions peuvent être formées en énumérant leurs éléments, ou les intervalles de valeurs pour le cas des entiers. Par exemple, `[1,3..6,8]` représente la liste `[1,3,4,5,6,8]`. De telles listes sont utilisées pour représenter les domaines des variables dans les formules *(var* `in` *list)*, et dans les réponses renvoyées par les buts Rules2CP. Les expressions suivantes : `length`(*list*), `nth`(*integer,list*), `pos`(*element,list*) et *attribute*(*record*) sont prédéfinies pour accéder aux composantes des listes et des enregistrements. De plus, les enregistrements possèdent un attribut par défaut à valeur entière `uid` qui fournit un identificateur unique pour chaque enregistrement.

La fonction prédéfinie `variables`(*expr*) retourne la liste des variables contenues dans une expression.

Les prédicats prédéfinis *X* `in` *list* et

[1] `http ://net-wms.ercim.org`

`domain(`*expr,min,max*`)` contraignent la variable X (resp. la liste des variables apparaissant dans une expression *expr*) à prendre leur valeur dans une liste d'entiers (resp. entre *min* et *max*)

2.2 Prédicats pour la Recherche et les Heuristiques

Décrire la stratégie de recherche dans un langage de modélisation est une tâche ambitieuse puisque la recherche est habituellment considérée comme procédurale par nature, et donc contradictoire avec la modélisation déclarative. Ce n'est pourtant pas notre point de vue dans Rules2CP. Notre approche à cette question est de spécifier les *variables de décision* et les *formules d'exploration* du problème déclarativement, ainsi que de donner les heuristiques comme des *ordres de préférence* sur les variables et les valeurs.

Les variables de décision peuvent être déclarées grâce au prédicat prédéfini `labeling(`*expr*`)` qui énumère les valeurs possibles de toutes les variables *contenues* dans une expression, c'est-à-dire apparaissant comme attribut d'un enregistrement, ou récursivement dans un enregistrement référencé par des attributs, dans une liste ou dans une formule du premier ordre.

De plus, des *formules d'exploration* peuvent être déclarées avec le prédicat plus original `search(`*fol*`)` qui définit une recherche en "explorant" les disjonctions et les quantifications existentielles qui apparaissent dans une formule du premier ordre. On peut noter qu'une approche similaire pour la définition de la recherche a été proposée pour SAT dans [14]. Cependant, ici, la seule normalisation est l'élimination des négations dans la formule en les faisant descendre au niveau des contraintes. La structure de la formule est conservée sous forme d'arbre de recherche *et-ou*, où les disjonctions représentent les points de choix.

Dans Rules2CP, les *prédicats d'optimisation* `minimize(`*expr*`)` pour minimiser une expression et `maximize(`*expr*`)` déterminent des critères d'optimisation indépendants de la recherche, et sans restriction sur le nombre de leurs occurences dans une formule. Il est alors possible d'exprimer des problèmes d'optimisation multicritères et la recherche de solutions Pareto-optimales selon l'ordre lexicographique des critères lus de gauche à droite.

Doter Rules2Cp de la capacité d'exprimer des connaissances heuristiques est obligatoire eu égard aux questions d'efficacité. Cela est rendu possible grâce aux deux prédicats de définition des choix statiques et dynamiques de variables et de valeurs. Les critères dynamiques sont standards dans les systèmes de programmation par contraintes, *cf.* par exemple [2, 6]. La définition des critères statiques exploite l'expressivité de Rules2CP.

Le prédicat `variable_choice_heuristics` prend une liste de critères ordonnant les variables pour la recherche. Les variables sont triées selon le premier critère quand il s'applique, le second ensuite, *etc.* Les variables pour lesquelles aucun critère ne s'applique sont placées en dernière position pour l'énumération et dans un ordre quelconque. Chaque critère prend l'une des formes suivantes : `greatest(`*expr*`)`, `greatest(`*expr, option*`)`, avec potentiellement `smallest`, `any` ou `is` en lieu et place de `greatest`. L'expression *expr* dans un critère contient le symbole `^` qui indique, pour une variable donnée, la tête de la règle qui a introduit la variable. Si l'expression ne peut être évaluée sur une variable, le critère est alors ignoré pour cette variable. Une forme `any` sélectionne une variable pour laquelle l'expression s'applique indépendamment de sa valeur. Une forme `is` sélectionne une variable si elle est identique au résultat d'une expression. *option* est un *critère de choix dynamique* utilisant les mots-clefs suivants : `leftmost`, plus petite borne inférieure `min`, plus grande borne supérieure `max`, plus petit domaine `ff`, ou variable la plus contrainte `ffc`. Par exemple, dans un problème Bin Packing, le prédicat

```
variable_choice_heuristics([greatest(volume(^)),
                            smallest(uid(^), ff)])
```

définit un ordre lexicographique statique sur les variables par volume décroissant des objets dans lesquels elles ont été déclarées, par *uid* croissant, et pour celles qui apparaissent dans le même objet (*i.e.*, qui ont le même uid), un ordonnancement dynamique par taille de domaine croissant (`ff`).

Le prédicat `variable_choice_heuristics` prend de façon similaire une liste de critères de la forme : `up`, `up(`*expr*`)`, pour énumérer les valeurs dans un ordre croissant ou `down`, `step` pour des choix binaires, `enum` pour des choix multiples, `bisect` pour des choix dichotomiques. Une critère s'applique à une variable s'il "match" l'expression. Par exemple, dans un problème Bin Packing avec des coordonnées x, y, z, le prédicat

```
value_choice_heuristics([up(z(^)), bisect(x(^)),
                         bisect(y(^))])
```

spécifie l'énumération dans un ordre croissant pour les coordonnées en z, et par dichotomie pour les x et les y.

La dissociation des heuristiques de variables et de valeurs et l'usage de critères statiques sur les objets dans lesquels apparaissent les variables constituent des traits puissants. Il est intéressant de noter que ce pouvoir expressif pour les heuristiques crée toutefois certaines difficultés pour leur compilation vers des systèmes de contraintes qui mélangent les deux types de stratégies dans une seule liste, et pour lequels on ne peut pas exprimer différentes heuristiques de choix de valeur pour différentes variables [6].

2.3 Exemples Simples

Exemple 1 *Le problème des N-reines peut être modélisé en Rules2CP à l'aide de déclarations pour créer une liste d'enregistrements représentant la position de chaque reine sur le damier, et avec une règle pour désigner quand aucune reine d'une liste n'attaque une autre. Une règle déclare les contraintes d'un problème de taille N, et un but indique la taille du problème à résoudre :*

```
q(I) = {row=_, column=I}.
board(N) = map(I, [1..N], q(I)).
safe(L) --> forall(Q, L, forall(R, L,
            let(I, column(Q), let(J, column(R),
             I<J implies row(Q) # row(R) and
                         row(Q) # J-I+row(R) and
                         row(Q) # I-J+row(R))))).
solve(N) --> let(B, board(N),
                    domain(B,1,N) and
                    safe(B) and labeling(B)).
? solve(4).
```

□

Exemple 2 *Un problème d'ordonnancement disjonctif peut être modélisé comme suit :*

```
t1 = {start=_, dur=1}. t2 = {start=_, dur=2}.
t3 = {start=_, dur=3}. t4 = {start=_, dur=4}.
t5 = {start=_, dur=2}. t6 = {start=_, dur=0}.
precedences --> prec(t1,t2) and prec(t2,t3) and
                prec(t3,t6) and prec(t1,t4) and
                prec(t4,t5) and prec(t5,t6).
disjunctives --> disj(t2,t5) and disj(t4,t3).
prec(T1,T2) --> start(T1)+dur(T1) =< start(T2).
disj(T1,T2) --> prec(T1,T2) or prec(T2,T1).
? start(t1)>=0 and start(t6)<20 and precedences and
  search(disjunctives) and minimize(start(t6)).
```

Le but pose les contraintes de précédence, et développe un arbre de recherche pour les contraintes disjonctives sans variables d'énumération. Pour les systèmes où l'instanciation du coût est nécessaire dans les prédicats d'optimisation, l'énumération des valeurs de la variable de coût est ajoutée automatiquement par le compilateur Rules2CP. □

3 Compilation vers des Programmes de Contraintes sur Domaines Finis avec Réification

Les modèles Rules2CP se compilent en des problèmes de satisfaction de contraintes sur les domaines finis avec contraintes réifiées par une interprétation des déclarations basée sur un système de réécriture de termes, *i.e.* avec un procédé qui réécrit les sous-termes des termes selon des règles de réécriture générales. Il est à noter que pour des considérations d'interaction utilisateur et de déboguage à l'execution, les informations de *bookkeeping* demandent à être implantées dans cette transformation afin d'entretenir les (rétro)dépendances des variables CP vers les déclarations Rules2CP [9]. Appellons $\rightarrow_{csp}$ la relation de réécriture de termes de la compilation.

3.1 Règles de Réécriture Génériques

Les règles de réécritures suivantes sont associées aux règles et déclarations Rules2CP :

- $L \rightarrow_{csp} R$ pour chaque règle de la forme L `-->` R,
- $L \rightarrow_{csp} R$ pour chaque déclaration de la forme L `=` R avec $V(R) \subseteq V(L)$;
- $L\sigma \rightarrow_{csp} R\sigma\theta$ pour chaque déclaration de la forme L `=` R avec $V(R) \not\subseteq V(L)$ et pour chaque substitution σ des variables dans $V(L)$, où θ est une substitution de renommage qui donne des noms uniques indexés par $L\sigma$ aux variables dans $V(R) \setminus V(L)$.

Dans une règle Rules2CP, toutes les variables libres de la partie droite doivent apparaître dans la partie gauche. Dans une déclaration Rules2CP, des variables libres peuvent être introduites dans la partie droite et leur portée est globale. Ces variables se voient donc attribuer un nom unique (grâce à la substitution θ) qui sera le même à chaque invocation de l'objet. Ces noms sont indexés par la partie gauche de la déclaration qui doit dans ce cas être instanciée (substitution σ). Par exemple, la variable `row` dans les enregistrements déclarés par `q(N)` dans l'exemple 1 se voient attribuer un nom unique indexé par les instances de la tête. Ces conventions fournissent un mécanisme de *bookkeeping* basique qui permet de retrouver les variables Rules2CP introduites dans une déclaration, à partir de leur nom. Ce qui est essentiel eu égard au déboguage et à l'interaction utilisateur [9]. Les expressions arithmétiques sont réécrites par la règle :

- $expr \rightarrow_{csp} v$ si $expr$ est une expression instanciée et v sa valeur.

Cette règle fournit un mécanisme d'*évaluation partielle* simplifiant les expressions arithmétiques et booléennes. Ceci est déterminant pour limiter la taille des programmes générés et éliminer à la compilation une surcharge potentielle due aux structures de données utilisées dans Rules2CP. Dans l'exemple 1, l'inégalité `I<J` contenue dans le corps de la règle de tête `safe(L)` sera complètement instanciée par les opérateurs `let` et évaluée à `true` ou `false`. Ensuite interviendra l'évaluation partielle de la formule implicative dont elle fait partie. Par exemple, pour `I` et `J` valuées à 1, `I<J` est évaluée à `false`. La condition de l'implication valant alors `false` elle est évaluée à `true`, qui sera éliminé ensuite au niveau de la quantification universelle.

Les accesseurs aux structures de données sont réécrits selon le schéma de règles suivant qui impose que

les arguments de type liste soient développés d'abord :

- `[`i `..` j`]` $\rightarrow_{csp}$ `[`i`,` i+1`,...,`j`]` si i et j sont entiers et $i \leq j$
- `length([`e_1`,...,`e_N`])` $\rightarrow_{csp}$ N
- `nth(i,[`e_1`,...,`e_N`])` $\rightarrow_{csp}$ e_i
- `pos(`e`,[`e_1`,...,`e_N`])` $\rightarrow_{csp}$ i où e_i est la *première* occurrence de e dans la liste après réécriture,
- *attribute(R)* $\rightarrow_{csp}$ V si R est un enregistrement de valeur V pour *attribute*.

Les quantificateurs, opérateurs `aggregate`, `map` et `let` sont des lieurs qui utilisent une variable factice X pour indiquer les paramètres fictifs dans une expression. Ils sont réécrits sous la condition que leur premier argument X soit une *variable* et le second une liste développée :

- `aggregate(`X`,[`e_1`,`$\cdots$`,`e_N`],`op`,`e`,`ϕ`)` $\rightarrow_{csp}$ $\phi[X/e_1]$ *op...op* $\phi[X/e_N]$ (e si $N = 0$)
- `forall(`X`,[`e_1`,`$\cdots$`,`e_N`],`ϕ`)` $\rightarrow_{csp}$ $\phi[X/e_1]$ `and` ... `and` $\phi[X/e_N]$ (`1` si $N = 0$)
- `exists(`X`,[`e_1`,`$\cdots$`,`e_N`],`ϕ`)` $\rightarrow_{csp}$ $\phi[X/e_1]$ `or` ... `or` $\phi[X/e_N]$ (`0` si $N = 0$)
- `map(`X`,[`e_1`,`$\cdots$`,`e_N`],`ϕ`)` $\rightarrow_{csp}$ `[`$\phi[X/e_1], ..., \phi[X/e_N]$`]`
- `let(`X`,`e`,`ϕ`)` $\rightarrow_{csp}$ $\phi[X/e]$

où $\phi[X/e]$ indique la formule ϕ où chaque occurence libre de la variale X est remplacée par l'expression e (après les renommages usuels des variables dans ϕ afin d'éviter les captures de variables). Les négations sont éliminées en les faisant descendre au niveau des relations de comparaison, avec les règles de dualité évidentes pour les connecteurs logiques, comme par exemple la réécriture de la négation de `equiv` en `xor`. A noter que ces transformations n'augmentent pas la taille de la formule.

3.2 Règles de Réécriture spécifiques pour primitives PPC Cibles

Les contraintes primitives du langage cible (contraintes globales inclues) sont données avec des *règles d'inlining* spécifiques. De telles règles sont obligatoires pour les termes qui ne sont pas définis par des déclarations Rules2CP, ainsi que pour les expressions arithmétiques et logiques qui ne peuvent pas être développées au moyen des règles de réécritures génériques décrites dans la section précédente. Le résultat d'une règle d'*inlining* est appelé un *terme terminal*. Les variables libres des déclarations sont traduites en variables à domaine fini du langage cible, avec le *bookkeeping* fournit par les conventions de nommage. Les exemples de règles d'*inlining* donnés dans cette section concernent la compilation de Rules2CP vers SICStus-Prolog [6]. Les contraintes primitives sont donc réécrites par les règles de réécriture suivantes :

- `domain(`E, M, N`)` $\rightarrow_{csp}$ `"domain(`L, M, N`)"` si M et N sont des entiers et où L est la liste des variables restantes dans E après réécriture
- A `>` B $\rightarrow_{csp}$ `"‘`A `#> ‘`B`"`
- A `and` B $\rightarrow_{csp}$ `"‘`A `#/\ ‘`B`"`
- `lexicographic(`L`)` $\rightarrow_{csp}$ `"lex_chain(‘`L`)"`

où les accents graves dans les chaînes de caractères indiquent les sous-expressions à réécrire. Evidemment, ces règles de réécritures génèrent des programmes de taille linéaire. Les règles d'*inlining* pour les prédicats de recherche Rules2CP sont plus complexes car elles demandent de créer la liste des variables contenues dans une expression, et de trier les contraintes, les prédicats de recherche et les critères d'optimisation dans les conjonctions. Par exemple, le *schéma de règles* d'*inlining* pour le critère d'optimisation est :

- A `and minimize(`C`)` $\rightarrow_{csp}$ `"‘`B`,minimize((‘`D`,labeling([up],‘`L`)),‘`C`)"`

où L est la liste des variables apparaissant dans l'expression de coût C, D est le but associé aux expressions d'énumération et de recherche apparaissant dans A avec les disjonctions remplacées par des points de choix, et B est la traduction de la formule A sans ses expressions d'énumération et de recherche. Le code généré par ces règles d'*inlining* est ici encore de taille linéaire.

Exemple 3 *La compilation du problème des N-reines dans l'exemple 1 génère le but SICStus Prolog suivant :*

```
? domain([Q_1_,Q_2_,Q_3_,Q_4_],1,4),
  Q_1_#\=Q_2_, Q_1_#\=1+Q_2_, Q_1_#\= -1+Q_2_,
  Q_1_#\=Q_3_, Q_1_#\=2+Q_3_, Q_1_#\= -2+Q_3_,
  ...
  labeling([],[Q_1_,Q_2_,Q_3_,Q_4_]).
```

Notons que les contraintes d'inégalité sont posées uniquement sur les couples de reines et que les autres paires de reines générées par la quantification universelle ont été éliminées à la compilation par évaluation partielle. □

Exemple 4 *Le résultat de la compilation du problème d'ordonnancement disjonctif dans l'exemple 2 se présente ainsi :*

```
? T1_ #>= 0, T6_#< 20, T1_+1 #=< T2_, T2_+2 #=< T3_,
  T3_+3 #=< T6_, T1_+1 #=< T4_,
  T4_+4 #=< T5_, T5_+2 #=< T6_,
  minimize((((T2_+2 #=< T5_;T5_+2 #=< T2_),
             (T4_+4 #=< T3_;T3_+3 #=< T4_)),
            labeling([up],[T6_])), T6_).
```

Le prédicat de recherche appliqué à une formule du premier ordre a été transformé en un arbre de recherche et-ou, conservant l'imbrication des disjonctions sans normalisation. Ce qui est essentiel pour assurer à la transformation une complexité linéaire. □

3.3 Confluence, Terminaison et Complexité

En interdisant les définitions multiples (ou surcharges de fonctions et prédicats), et en restreignant les têtes à ne contenir comme arguments que des variables distinctes, on peut montrer que :

Proposition 1 *Pour tout modèle* `Rules2CP`*, la compilation par le système de réécriture de termes* $\rightarrow_{csp}$ *est confluent.*

Cela veut dire que les règles de réécriture peuvent être appliquées dans un ordre arbitraire, le programme de contraintes généré sera le même pour un même modèle d'entrée. La preuve dans [16] montre que le système de réécriture $\rightarrow_{csp}$ est orthogonal, *i.e.* linéaire à gauche et *non-overlapping*, ce qui implique la confluence [19], sans hypothèse de terminaison. En interdisant la récursion, on obtient :

Proposition 2 *Pour tout modèle* `Rules2CP`*, le système de réécriture* $\rightarrow_{csp}$ *est noethérien.*

La preuve de terminaison [16] fait intervenir un ordre sur un multi-ensemble de chemins qui implique les longueurs des dérivations récursives primitives [13]. Cependant, sans récursion, une meilleure borne de complexité sur la taille du programme généré peut être obtenue :

Définition 1 *Etant donné un modèle Rule2CP* M*, soit* $\alpha(s)$ *le* rang d'agrégat *d'un symbole* s *définit inductivement par :*

- $\alpha(s) = 0$ *si* s *n'est pas la tête d'une déclaration ou d'un règle dans* M*,*
- $\alpha(s) = max\{n + \alpha(s') \mid L = R \in M$*,* s *est le symbole de tête de* L *et* R *contient une imbrication de* n *opérateurs d'agrégation ou de quantification sur une expression contenant le symbole* $s'\}$*.*

Le rang d'agrégat de M *est le rang d'agrégat maximum parmi les symboles de* M*.*

Théorème 1 *Pour tout modèle* `Rules2CP` M*, la taille du programme généré est en* $O(l^a * b^r)$*, où* l *est la longueur maximale des listes développées dans* M *(ou au moins 1),* a *est le rang d'agrégat de* M*,* b *est la taille maximale des corps de règle et de déclaration dans* M*, et* r *est le rang de définition de* M*.*

Preuve 1 *La preuve se fait par induction sur* a*. Dans le cas de base,* $a = 0$*, il n'y a pas d'opérateur d'agrégation dans* M*, et la taille du programme généré est linéairement bornée par* r *duplications de corps de règles, i.e. est en* $O(b^r)$*. Dans le cas inductif,* $a > 0$*, considérons d'abord la taille du programme généré sans réécriture des occurences superficielles (ultrapériphériques) des opérateurs d'agrégat et de quantification. Par induction, cette taille est en* $O(l^{a-1} * b^r)$*. Maintenant, ce programme généré peut être dupliqué* l *fois par les opérateurs d'agrégation superficiels, donc la taille totale est en* $O(l^a * b^r)$ *selon cette stratégie. Puisque selon la Prop. 1 de confluence, le programme généré est indépendant de la stratégie, la taille du programme généré est donc en* $O(l^a * b^r)$ *quelle que soit la stratégié.* □

Dans l'exemple 1 du problème des N-reines, la rang d'agrégat est 2. Le théorème nous dit alors que la taille du programme généré pour un damier de taille l est effectivement en $O(l^2)$.

4 La Librairie de Modélisation des Connaissances de Placement PKML

Dans cette section, nous illustrons le pouvoir expressif de Rules2CP avec la définition d'une librairie de modélisation des connaissances de placement (PKML) qui est développée au sein du projet Net-WMS afin de traiter des problèmes Bin Packing non-purs et de taille réelle venant des industries automobile et logistique.

4.1 Formes et Objets

PKML porte sur des formes à coordonnées entières dans $\mathbb{Z}^K$ vivant dans un espace K-dimensionnel. Un *point* dans cet espace est représenté par la liste de ses K coordonnées entières `[i1,...,iK]`. Ces coordonnées peuvent être des variables ou des valeurs entières fixées. En PKML, une *forme* (*shape*) est un *assemblage rigide de boîtes*. Une *boîte* (*box*) est un orthotope dans $\mathbb{Z}^K$, et est représenté par un enregistrement contenant un attribut *taille* (*size*) qui donne la liste des longueurs de la boîte dans toutes les dimensions. Une forme est représentée par un enregistrement contenant un attribut *boxes* qui fournit la liste des boîtes composant la forme, et un attribut *positions* qui indique la liste de leur position dans l'assemblage (*i.e.* une liste de listes de coordonnées). D'autres attributs pourraient être ajoutés pour la représentation en réalité virtuelle, le poids, *etc.*

Les déclarations suivantes définissent respectivement le volume d'une boîte, une forme composée d'une seule boîte, la taille d'une forme (*i.e.*, un assemblage de boîtes) dans une dimension donnée (supposant aucun enchevêtrement au sein de l'assemblage) :

```
volume_box(B) = product(size(B)).
box(L) = { boxes = [ {size = L} ],
           positions = [ map(_,L,0) ] }.
size(S, D) =
  aggregate(I, [1..length(boxes(S))], max, 0,
            nth(D,nth(I,positions(S))) +
```

```
            nth(D,size(nth(I,boxes(S)))).
volume_assembly(S, Dims) =
  aggregate(B, boxes(S), +, 0,
            volume_box(B)).
```

Notons que si les tailles des boîtes constituant les formes sont connues, les expressions de taille et de volume s'évaluent en des valeurs entières, alors que si les tailles sont inconnues, les expressions s'évaluent en des termes contenant des variables. Un *objet* (*object*) tel qu'un container ou une boîte particulière, est un enregistrement contenant un attribut `shapes` qui donne la liste des *alternatives de formes* pour l'objet, un point *origine*, et des attributs optionnels tels que le poids (weight), représentation en réalité virtuelle ou autres. Les alternatives de formes d'un objet peuvent être utilisées pour représenter les différentes formes obtenues par rotation de la forme fondamentale autour des axes des différentes dimensions, ou pour exprimer le choix entre différentes formes d'objet dans un problème de configuration. Nous ne distinguons pas entre les traits des containers et ceux des autres boîtes, puisqu'un container à un niveau donné peut devenir une boîte à placer dans un autre, comme par exemple dans le problème de Bin Packing multiniveaux qui place des pièces (boîtes) dans des cartons, les cartons dans des palettes, et les palettes dans des camions. L'origine et la fin d'un objet dans une dimension ainsi que son volume sont prédéfinis sur les alternatives de formes (grâce à la réification) comme suit :

```
origin(O, D) = nth(D, origin(O)).
end(O, D) = origin(O, D) +
  aggregate(S, shapes(O), +, 0,
    (shape(O)=pos(S,shapes(O)))*size(S, D)).
volume(O, Dims) =
  aggregate(S, shapes(O), +, 0,
    (shape(O)=pos(S,shapes(O)))*
    volume_assembly(S,Dims)).
```

4.2 Relations de Placement

Pour exprimer les contraintes de placement, PKML se fonde sur les relations d'intervalles d'Allen [1] pour le cas unidimensionnel et sur les relations topologiques du Calcul des Connexions de Regions (RCC8) [18] en dimensions supérieures. Ces relations sont prédéfinies dans les librairies [16]. Elles sont utilisées en PKML pour définir les règles de placement pour les problèmes Bin Packing et Bin Design purs ; les stratégies de cassage de symétries ; ainsi que pour les règles métier spécifiques impliquées dans les problèmes non-purs, en considérant d'autres règles de bon sens et des besoins et expertises industriels.

La partie de la librairie PKML qui traite des *problèmes Bin Packing* est définie comme ci-dessous :

```
non_overlapping(Items, Dims) -->
   forall(O1, Items, forall(O2, Items,
       uid(O1) < uid(O2) implies
       not overlap(O1, O2, Dims))).
containmentAE(Items, Bins, Dims) -->
   forall(I, Items, exists(B, Bins,
       contains_touch_rcc(B,I,Dims))).
bin_packing(Items, Bins, Dims) -->
   containmentAE(Items, Bins, Dims) and
   non_overlapping(Items, Dims) and
   labeling(Items).
```

Les règles définissent respectivement le *non-overlapping* (non-intersection, non-recouvrement) d'une liste de boîtes (items) dans une liste de dimensions ; le confinement de toutes les boîtes dans un container ; et le Bin Packing pur. Les *problèmes Bin-Design* sont définis de façon semblable avec une déclaration pour le volume d'un container, et une règle de confinement de toutes les boîtes dans un container :

```
containmentEA(Items, Bins, Dims) -->
   exists(B, Bins,
     forall(I, Items,
       contains_touch_rcc(B,I,Dims))).

bin_design(Bin, Items, Dims) -->
   containmentEA(Items, [Bin], Dims) and
   labeling(Items) and minimize(volume(Bin)).
```

Exemple 5 *Considérons le problème simple de Bin Packing pur suivant :*

```
s1 = box([5,4,4]).
s2 = box([5,4,2]). s3 = box([4,4,2]).
o1 = object(s1, [0,0,0]).
o2 = object(s2, [_,_,_]). o3 = object(s3, [_,_,_]).
dimensions = [1,2,3].
bins = [o1]. items = [o2,o3].
? bin_packing([o2,o3], [o1], [1,2,3]) and
  variable_choice_heuristics([greatest(volume(^)),
                              is(z(^))]).
```

Le compilateur génère à partir de cet exemple le but SICStus-Prolog suivant :

```
? 0#=<O2_, O2_+4#=<5, 0#=<O2_2, O2_2+4#=<4,
  0#=<O2_3, O2_3+2#=<4, 0#=<O3_, O3_+5#=<5,
  0#=<O3_2, O3_2+4#=<4, 0#=<O3_3, O3_3+2#=<4,
  O3_+5#=<O2_#\/O2_+4#=<O3_#\/(O3_2+4#=<O2_2#\/
  O2_2+4#=<O3_2#\/(O3_3+2#=<O2_3#\/O2_3+2#=<O3_)),
  labeling([],[O3_3,O3_,O3_2,O2_3,O2_,O2_2]).
```

□

4.3 Règles Métier de Placement et Motifs

Les règles métier de placement sont définies en Rules2CP afin de prendre en compte le bon sens ainsi que des besoins industriels qui dépassent la portée du Bin Packing pur [7]. Par exemple, les règles suivantes qui traitent du poids

```
gravity(Items) -->
  forall(O1, Items, origin(O1, 3) = 0 or
    exists(O2, Items,
      uid(O1) # uid(O2) and on_top(O1, O2))).
weight_stacking(Items) -->
  forall(O1,  Items, forall(O2, Items,
    (uid(O1) # uid(O2) and on_top(O1, O2))
    implies lighter(O1,O2))).
weight_balancing(Items, Bin, D, Ratio) -->
  let(L, sum( map(Il, Items,
    weight(Il)*(end(Il,D) =< (end(Bin,D)/2)))),
  let(R, sum( map(Ir, Items,
    weight(Ir)*(end(Ir,D) >= (end(Bin,D)/2)))),
    100*max(L,R) =< (100+Ratio)*min(L,R))).
```

expriment des contraintes de poids particulières pour un placement admissible. La libraire PKML complète, qui inclut des règles de bon sens prenant en considération le poids des objets et la surface de contact de boîtes empilées, est présentée dans [16]. Avec ces règles, le théorème 1 montre que les modèles PKML contenant des listes d'au plus l éléments génèrent des programmes de contraintes de taille $O(l^4)$ en présence à la fois de formes alternatives et d'assemblages d'objets ; de taille $O(l^3)$ en présence d'un des deux ; de taille $O(l^2)$ en présence de formes composées d'une boîte unique.

Les *motifs métiers* peuvent être utilisés dans PKML pour exprimer des connaissances sur des solutions partielles de problèmes de placement. De tels motifs sont utilisés dans l'industrie, par exemple pour remplir des pallettes ou des camions avec une stabilité optimale. Des conditions de stabilité peuvent être exprimées grâce à des contraintes de non-guillotine et de non-visibilité [7], mais les motifs de placement constituent une approche pragmatique et complémentaire à ces besoins.

4.4 Compilation avec la Contrainte Globale `geost`

La contrainte `geost` [5] est une contrainte globale générique pour traiter les problèmes de placement multi-dimensionnels et qui est maintenant paramétrée par un langage de règles [4]. Un sous-ensemble des règles PKML peuvent être transformées directement vers les règles `geost`, procurant un élaguage à très haut niveau et remarquablement efficace. Le sous-ensemble des règles PKML concerné est restreint aux enregistrements d'objets et de formes, et aux expressions arithmétiques *linéaires*, *i.e.* combinaisons linéaires de variables de domaines. Grâce à ces restrictions, les règles `geost` peuvent être compilées en k-indexicaux, *i.e.* des fonctions qui calculent des *ensembles de points interdits* pour les objets, représentés par des boîtes k-dimensionnelles, et composés par unions et intersections [5]. La compilation des modèles PKML en problème de satisfaction de contraintes utilisant `geost`, consiste essentiellement à effectuer les étapes suivantes :

1. extraire les définitions des objets et des formes dans les déclarations PKML afin de les fournir à la contrainte `geost`,
2. extraire les déclarations et les règles qui font référence à des objets et des formes et qui satisfont la condition de linéarité,
3. compiler le but PKML en une contrainte `geost` et le code restant des règles et prédicats, qui ne sont pas acceptés par le langage de règles intégré à `geost`, comme décrit dans la section précédente.

Ce procédé de compilation peut être rafiné en ajoutant des dimensions supplémentaires, par exemple pour traiter les problèmes de placement multi-containers, chaque dimension supplémentaire représentant alors une dimension d'affectation des objets à un container ; ou bien pour gérer les aspects d'ordonnancement en ajoutant une dimension pour le temps, *etc.* [4].

5 Comparaisons aux travaux reliés

5.1 Règles Métier

Rules2CP tente d'intégrer à la programmation par contraintes le paradigme de représentation des connaissances par règles métier. Les règles métier sont très populaires dans l'industrie parce qu'elles fournissent un moyen simple d'exprimer l'expertise. Les règles métier devraient décrire des parcelles de connaissance indépendamment, et devraient être indépendantes de l'interprétation procédurale d'un moteur de règles particulier [11]. Rules2CP atteint cet objectif dans le contexte des problèmes d'optimisation combinatoires, en transformant les règles métier en des programmes efficaces utilisant des représentation complètement diférentes. Les règles Rules2CP ne sont pas des règles condition-action générales, aussi appellées règles de production dans la communauté systèmes experts, mais des *règles logiques* avec seulement une tête et pas d'action impérative. Les quantificateurs bornés servent à représenter des conditions complexes. De telles conditions peuvent aussi être exprimées dans beaucoup de systèmes de règles de production, mais elles sont exploitées ici à la compilation pour construire un problème de satisfaction de contraintes, plutôt qu'à l'exécution pour filtrer des faits dans une base de données. En tant que langage de modélisation à base de règles, Rules2CP est conforme aux principes énoncés dans le manifeste des règles métier [11].

5.2 OPL et Zinc

Rules2CP se distingue des langages de modélisation OPL [21] et Zinc [17, 8] par plusieurs aspects, parmi lesquels : la restriction à des structures de données simples d'enregistrements et de listes énumérées, l'absence de récursion, la spécification déclarative des heuristiques comme des ordres de préférences, et l'absence d'annotations. Ce compromis pour faciliter l'utilisation a été motivé par une investigation sur un langage de modélisation sans construction de programmation sophistiquée. Nous avons montré que les déclarations et les règles Rules2CP permettent à l'utilisateur de donner des noms aux données et aux règles de connaissance sans avoir à considérer des portées complexes de variables. Un système de module simple permet d'éviter les *conflits* de noms. La simplicité de ces choix de conception est reflétée par l'obtention de bornes de complexité sur la taille du programme de contraintes généré à partir des modèles Rules2CP (théorème 1). De plus, le mécanisme d'évaluation partielle pendant la réécriture élimine les surplus à la compilation introduits par la simplicité des structures de données et de contrôle.

Nous avons montré que des stratégies de recherche complexes peuvent être exprimées *déclarativement* dans Rules2CP, en déterminant les *variables de décision* et les *formules d'exploration*, ainsi que les heuristiques de choix statiques et dynamiques comme *ordres de préférences* sur les variables et les valeurs. Ces spécifications sont plus déclaratives que celles proposées par OPL pour programmer la recherche, et utilisent toute la puissance du langage pour définir les critères des heuristiques. D'un autre côté, nous ne nous sommes pas intéressé à la compilation de Rules2CP vers d'autres types de solveurs tels que la recherche locale ou la programmation linéaire mixte en nombres entiers, comme c'est le cas des systèmes OPL et Zinc.

5.3 Programmation Logique avec Contraintes

En tant que langage de modélisation, Rules2CP est un langage de programmation logique avec contraintes, mais pas au sens formel du système PLC de Jaffar et Lassez [15]. Les modèles Rules2CP peuvent être compilés vers des programme CLP(FD) simplement en traduisant les règles Rules2CP en clauses Prolog, et en gardant la réécriture $\rightarrow_{csp}$ pour le reste des expressions. A noter que la traduction inverse, de programmes Prolog vers des modèles Rules2CP, n'est pas possible (à part depuis un encodage arithmétique) à cause de l'absence de récursion et de constructeurs de listes généraux dans Rules2CP.

En outre, les variables libres ne sont pas autorisées dans une partie droite de règle Rules2CP. Là où l'on utilise un mécanisme de portée locale pour les variables libres dans les règles PLC, on fait usage d'un mécanisme de portée globale dans Rules2CP. Ce mécanisme de portée globale n'a pas d'homologue dans le système PLC qui oblige à passer la liste de toutes les variables comme arguments des prédicats PLC.

5.4 Outils de Réécriture de Termes

La compilation des modèles Rules2CP vers des programmes de contraintes est définie et implantée par un système de réécriture de termes. Les propriétés de confluence et de terminaison de ce procédé ont été montrées en utilisant la théorie de la réécriture de termes.

Il existe plusieurs outils de réécriture de termes disponibles qui pourraient directement être utilisés pour implanter le compilateur Rules2CP. Par exemple, dans le contexte du ciblage de solveurs Java, tels que choco par exemple, et pour des environements de programmation Java dans lesquels les structures de données Rules2CP devraient être définies par des objets Java, le système de réécriture TOM [3], offre un compilateur basé sur le *pattern matching* pour programmer les transformations définies par les règles. Cela placerait TOM comme système idéal pour implanter un compilateur Rules2CP vers Java, à travers une traduction directe des règles $\rightarrow_{csp}$ en règles TOM.

6 Conclusion

Le langage Rules2CP est un langage de modélisation à base de règles pour la programmation par contraintes. Il a été conçu pour permettre aux non-programmeurs d'exprimer des règles de bon sens et des besoins industriels à propos de problèmes d'optimisation combinatoires, grâce à des règles métier. En conformité avec le manifeste des règles métier [11], les règles Rules2CP sont déclaratives, indépendantes les unes des autres, et pas nécessairement exécutées par un moteur de règles.

Nous avons montré que les modèles Rules2CP peuvent être compilés vers des programmes de contraintes utilisant la réécriture de termes et l'évaluation partielle. Nous avons montré la confluence de ces transformations et fourni une borne sur la taille des programmes générés. L'obtention de tels résultats de complexité reflètent la simplicité des choix de conception pour Rules2CP, tels que l'absence de récursion et de constructeur général de listes.

L'expressivité de Rules2CP a été illustrée avec une librairie complète pour les problèmes de placement, appellée PKML, qui, en plus des problèmes Bin Packing et Bin Design purs, peut traiter des contraintes additionnelles portant sur le poids, l'équilibre, ou

d'autres règles métier spécifiques. De plus, une part substantielle des règles PKML peut être efficacement compilée vers la contrainte globale géométrique `geost` [4].

Les stratégies de recherche peuvent aussi être spécifiées déclarativement dans Rules2CP, ainsi que des heuristiques statiques et dynamiques définies comme des ordres de préférences sur les variables et les valeurs. Cette méthode pour définir les heuristiques est très expressive, et révèle une faiblesse dans les systèmes de programmation par contraintes où l'on ne peut exprimer, avec les prédicats consacrés, des heuristiques de choix de valeurs pour différents ordres de variables déterminés par une heuristique de choix de variables.

Remerciements.

Ce travail a été réalisé dans le cadre du projet Européen Strep FP7 Net-WMS.

Références

[1] J. Allen. Time and time again : The many ways to represent time. *International Journal of Intelligent System*, 6(4), 1991.

[2] Krzysztof Apt and Mark Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, 2006.

[3] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom : Piggybacking rewriting on java. In *Proceedings of th 18th International Conference on Rewriting Techniques and Applications, RTA'07*, number 4533 in Lecture Notes in Computer Science. Springer-Verlag, 2007.

[4] N. Beldiceanu, M. Carlsson, and J. Martin. A geometric constraint over k-dimensional objects and shapes subject to business rules. SICS Technical Report T2008 :04, Swedish Institute of Computer Science, 2008.

[5] N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In C. Bessière, editor, *Proc. CP'2007*, volume 4741 of *LNCS*, pages 180–194. Springer, 2007. Also available as SICS Technical Report T2007 :08, `http ://www.sics.se/libindex.html`.

[6] M. Carlsson et al. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, release 4 edition, 2007. ISBN 91-630-3648-7.

[7] H. Carpenter and W. Dowsland. Practical consideration of the pallet loading problem. *Journal of the Operations Research Society*, 36 :489–497, 1985.

[8] Maria Garcia de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. The modelling language Zinc. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming* CP *'06)*, pages 700–705. Springer-Verlag, 2006.

[9] François Fages, Sylvain Soliman, and Rémi Coolen. CLPGUI : a generic graphical user interface for constraint logic programming. *Journal of Constraints, Special Issue on User-Interaction in Constraint Satisfaction*, 9(4) :241–262, October 2004.

[10] Ian P. Gent, Peter Nightingale, and Kostas Stergiou. Qcsp-solve : A solver for quantified constraint satisfaction problems. In *IJCAI*, pages 138–143, 2005.

[11] Business Rules Group. The business rules manifesto, 2003. Business Rules Group `http ://www.businessrulesgroup.org/brmanifesto.htm`.

[12] Rémy Haemmerlé and François Fages. Modules for Prolog revisited. In *Proceedings of International Conference on Logic Programming ICLP 2006*, number 4079 in Lecture Notes in Computer Science, pages 41–55. Springer-Verlag, 2006.

[13] Dieter Hofbauer. Termination proofs by multiset path orderings imply primitive recursive derivation lengths. *Theoretical Computer Science*, 105(1) :129–140, 1992.

[14] Jinbo Huang and Adnan Darwiche. The language of search. *Journal of Artificial Intelligence Research*, 29 :191–219, 2007.

[15] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[16] François FagesJulien Martin. From rules to constraint programs with the Rules2CP modelling language. INRIA Research Report RR-6495, Institut National de Recherche en Informatique, April 2008.

[17] Reza Rafeh, Maria Garcia de la Banda, Kim Marriott, and Mark Wallace. From Zinc to design model. In *Proceedings of PADL'07*, pages 215–229. Springer-Verlag, 2007.

[18] D.A. Randell, Z. Cui, and A.G. Cohn. A spatial logic based on regions and connection. In B. Nebel, C. Rich, and W. R. Swartout, editors,

Proc. of 2nd International Conference on Knowledge Representation and reasoning KR'92, pages 165–176. Morgan Kaufmann, 1992.

[19] B.K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20 :160–187, 1973.

[20] Pascal Van Hentenryck. *Constraint satisfaction in Logic Programming*. MIT Press, 1989.

[21] Pascal Van Hentenryck. *The OPL Optimization programming Language*. MIT Press, 1999.