Institut National de la Recherche Agronomique

Laboratoire de Biométrie et d'Intelligence Artificielle
BP 27, 31326 Castanet Tolosan cedex, France
email : conflex@toulouse.inra.fr

CON'*FLEX*

Manuel de l'utilisateur

version 1.2
16 janvier 1998

Les modifications apportées par rapport à la précédente version sont repérées par une barre dans la marge.

CON'*FLEX* est un logiciel conçu, développé et mis à disposition par l'Institut National de la Recherche Agronomique (INRA). Les membres du groupe de projet, basé au Laboratoire de Biométrie et d'Intelligence Artificielle de Toulouse, sont les suivants :

– Jean-Pierre Rellier, coordinateur du projet et co-rédacteur de ce manuel

– Frédéric Vardon, concepteur de la version 1.0 du logiciel, développeur du code et co-rédacteur de ce manuel

– Christine Gaspin, Roger Martin-Clouaire et Thomas Schiex, conseillers scientifiques

– Martin Bouchez, Patrick Chabrier, Marie-Françoise Jourjon, conseillers techniques

Table des matières

Chapitre 1

Introduction

Dans les tâches de gestion de ressource (par exemple, l'organisation d'un atelier de fabrication), ou de conception (par exemple, l'assemblage de pièces mécaniques), il s'agit souvent de rechercher un plan, une structure, un ordonnancement qui respecte un ensemble de contraintes. Quand plusieurs de ces objets existent (ce sont les solutions du problème), on peut vouloir en choisir un particulier, sur la base de préférences exprimées par le concepteur ou l'utilisateur de l'objet.

Prenons l'exemple de la réalisation de l'emploi du temps d'un collège. Il s'agit d'affecter un cours à chaque salle du collège, pour chaque tranche horaire de la semaine. Un cours est défini comme une matière enseignée à une classe par un professeur. *A priori* tous les cours peuvent être placés dans une classe pour une tranche donnée. En fait, beaucoup de ces choix ne sont pas possibles, du fait qu'une classe ne peut suivre qu'un cours à la fois, que les professeurs ne travaillent qu'un nombre d'heures limité, qu'il ne faut pas enchaîner trois heures de math pour une même classe, que les salles peuvent être dédiées à certaines matières, etc... Toutes ces connaissances peuvent être exprimées en termes de contraintes pesant sur la mise au point de l'emploi du temps. Seuls les emplois du temps satisfaisant l'ensemble des contraintes peuvent être effectivement mis en place. S'il y en a plusieurs, il faut en choisir un, par exemple celui qui minimise les déplacements des élèves entre les salles. S'il n'y en a aucun, il faut enlever du problème la ou les contraintes (*relaxation* du problème) qui empêchent de trouver un emploi du temps réalisable.

De façon générale, les problèmes relevant de la satisfaction de contraintes sont difficiles lorsqu'ils présentent un grand nombre de combinaisons candidates parmi lesquelles seulement un petit nombre sont réalisables ou vraisemblables, et que la nature, la forme et la diversité des contraintes empêche la mise en œuvre d'un algorithme numérique efficace (par exemple le simplexe, lorsque les contraintes sont toutes linéaires sur des variables entières ou réelles).

CON'*FLEX* est un outil de résolution de problèmes de satisfaction de contraintes, fondé sur le formalisme CSP (pour *Constraint Satisfaction Problems*). Il propose un langage d'expression du problème en termes de variables, de contraintes et de requêtes, et fournit un ensemble d'algorithmes de résolution. Ces algorithmes visent d'une part à réduire l'espace dans lequel il faut rechercher des solutions (cette opération s'appelle le *filtrage*), et d'autre part à parcourir cet espace (de manière *arborescente*) pour y trouver des solutions. On parle de résolution pour désigner la procédure qui met en œuvre tout à la fois le filtrage et la recherche arborescente. Voir [Tsa93] pour une introduction générale au domaine des CSP.

CON'*FLEX* permet de résoudre des CSP *mixtes*, c'est-à-dire contenant à la fois des *variables symboliques* (avec un domaine constitué d'une liste de symboles), des *variables entières* (avec un domaine constitué d'une liste d'entiers) et des *variables réelles* (avec un domaine représenté par une liste d'intervalles disjoints de flottants). Les contraintes peuvent être exprimées soit en *extension*, en décrivant les combinaisons possibles de valeurs pour les variables entrant dans la contrainte, soit en *intension*, par une équation liant les variables. Il est aussi possible d'exprimer des contraintes *conditionnelles* (c'est-à-dire qui n'entrent en jeu dans le problème que si une certaine condition est satisfaite), des contraintes en intension *définies par morceaux*, et enfin des *disjonctions* et des *conjonctions* de contraintes.

L'originalité de CON'*FLEX* réside surtout dans sa capacité à traiter les problèmes dits flexibles, c'est-à-dire ceux où les variables ont des domaines flous, où les contraintes sont plus ou moins importantes à satisfaire et, par ailleurs, plus ou moins satisfaites par les différentes combinaisons de valeurs des variables, et enfin où on peut admettre une solution qui ne satisfait que partiellement l'ensemble des contraintes. Noter cependant que CON'*FLEX* n'offre pas de fonctionnalité de *relaxation automatique* du problème en cas d'absence de solutions.

Pour utiliser CON'*FLEX* , aucune connaissance d'un langage de programmation n'est requise. Il suffit en effet de déclarer l'existence de variables et de contraintes, dans un ordre relativement libre, à partir d'un vocabulaire limité et d'une grammaire simple, spécifiques à CON'*FLEX* .

Cette documentation décrit la structure des fichiers de données (c'est-à-dire le codage du problème) et la commande d'appel de CON'*FLEX* dans le chapitre 2, avec l'exemple d'un petit problème. Le chapitre 3 précise la façon de poser la requête de résolution et de donner une valeur à certains paramètres du calcul. Les chapitres 4 et 5 décrivent respectivement la syntaxe d'expression des variables et des contraintes. Des articles concernant les algorithmes utilisés par CON'*FLEX* sont mentionnés dans la bibliographie. Le lecteur est invité à les consulter pour une compréhension approfondie des mécanismes de résolution. Toutefois, un certain nombre d'explications informelles sur ces mécanismes seront fournies en notes de pas de page.

Chapitre 2

Généralités

2.1 Le langage d'expression du problème

Le problème à résoudre est exprimé par l'utilisateur dans un langage propre à CON'*FLEX* . Ce langage possède un *vocabulaire* simple, constitué des *mots-clés* réservés (tableau 2.1) qui ont un sens imposé par le système, et des *identificateurs* et valeurs fournis par l'utilisateur et spécifiques au problème à résoudre.

Il y a deux sortes de mots-clés. Les uns désignent une commande (requête de résolution, déclaration d'une variable ou d'une contrainte, définition de la valeur d'un paramètre) ou un opérateur logique ou de contrôle à l'intérieur d'une commande (`and`, `or`, `if/then`,`...`) : ils sont toujours précédés du caractère *backslash* "\". Les autres mots-clés (non précédés du caractère "\") sont utilisés pour désigner certaines valeurs prédéfinies pour les paramètres des commandes (comme par exemple le type d'algorithme ou d'heuristique à utiliser). Le sens des mots-clés et la syntaxe des commandes seront complètement précisés dans les chapitres suivants.

Tab. 2.1 - Les mots réservés de CON'*FLEX*

Commandes	`\acut`	`\allbut`	`\alors`
	`\alpha`	`\and`	`\conjunction_of_constraints`
	`\const_int_multiple`	`\constraint_conditionnal`	`\constraint_extension`
	`\constraint_intension`	`\constraint_piecewise`	`\disjunction_of_constraints`
	`\do`	`\dynamic_labeling_order`	`\faire`
	`\filtering`	`\for`	`\fuzzy_cut_step`
	`\if`	`\or`	`\outsol`
	`\pour`	`\save`	`\search`
	`\si`	`\static_labeling_order`	`\then`
	`\variable_integer`	`\variable_real`	`\variable_symbolic`
	`\verbose`		
Paramètres des commandes	`all`	`all_solutions`	`alphabetic_order`
	`best_solutions`	`bt`	`display_all`
	`display_csp`	`display_filtering`	`display_intervals`
	`display_search`	`display_search_tree`	`display_solutions`
	`display_step`	`display_ultra_filtering`	`f`
	`fc`	`first_solution`	`first_solutions`
	`greatest_degree`	`no_alphabetic_order`	`no_display_step`
	`rfla`	`smallest_domain`	`smallest_domain_by_degree`
	`uf`		
Opérateurs	`abs`	`arccos`	`arcsin`
	`cos`	`exp`	`known`
	`ln`	`max`	`min`
	`nthroot`	`sin`	`sqrt`
Constantes	`INFINITY`	`2PI`	`PI`
	`PI/2`	`PI/4`	

Les identificateurs (noms) de variables ou de contraintes[1] peuvent être n'importe quelle *chaîne de caractères*. Une chaîne de caractère commence par un caractère alphabétique (a-z, A-Z) suivi d'un nombre quelconque de caractères alphanumériques (a-z, A-Z, 0-9) ainsi que le point "." et le souligné "_"[2].

D'une manière générale, une déclaration (d'une variable, d'une contrainte, ...) se termine par le caractére ";" qui sert donc de *caractère séparateur*.

Enfin, la syntaxe autorise l'incorporation de lignes de *commentaires* commençant par le caractère "#". Une telle ligne étant ignorée lors de la lecture du fichier, elle peut aussi servir à inhiber une requête, une option ou une déclaration déjà présente dans le fichier.

2.2 Conventions typographiques

Un certain nombre de conventions typographiques ont été adoptées dans ce manuel. Le style `courrier` est utilisé pour les exemples de code CON'*FLEX* . Les mots en *italique* sont placés dans l'index, avec la mention de la page où ils sont définis ou utilisés pour la première fois. Chaque élément de la syntaxe des commandes est décrit dans un cadre, avec les conventions suivantes[3] :

- le style `courrier` est utilisé pour les mots-clés réservés de CON'*FLEX* ;
- le style *italique* entre les caractères "<" et ">" est utilisé pour désigner une expression en général définie juste après son utilisation.
- le caractère "|" indique le "`ou` *inclusif*" dans une liste de choix possibles ;
- le caractère "/" indique le "`ou` *exclusif*" dans une liste de choix possibles ;
- les caractères "[ ]" entourent les valeurs optionnelles ;
- les points de suspension "..." ou "⋮" avant et après une expression régulière indiquent la répétition possible de l'expression.

Par exemple, la description suivante :

`\toto` <*terme1*> <*terme2*> `;`

avec :

<*terme1*> = `x` / `y` / `z`
<*terme2*> = ... a [(<*n*>)] | `b` ...
<*n*> : un nombre entier

autorise, entre autres, les combinaisons suivantes :

```
\toto x a a (0) b a;
\toto y b b b b a;
\toto z a (2) b a a;
```

mais pas celles-ci :

1. ... ainsi que les valeurs des variables symboliques

2. Dans la notation classique des expressions régulières, une chaîne de caractères correspond à : [a-zA-Z][a-zA-Z0-9._]*.

3. ... non conformes à la notation BNF, car la simplicité du langage de CON'*FLEX* ne nécessite pas une telle notation.

`\toto x y ;`	car `x` et `y` ne peuvent pas apparaître ensemble (*ou exclusif*)
`\toto y b (2);`	car la valeur `(2)` n'est optionnelle qu'à la suite de la valeur `a` et non de `b`

2.3 Utilisation de CON'*FLEX*

2.3.1 Commande de résolution

L'utilisation de CON'*FLEX* consiste simplement en l'appel du programme exécutable `conflex` suivi d'un ou plusieurs nom(s) de fichiers (appelés *fichiers d'entrée*) qui contiennent la description du CSP.

Syntaxe :

`conflex` <*filename1*> [<*filename2*> ...]

avec :

<*filename1*> : nom du premier fichier d'entrée
<*filename2*> : nom du fichier d'entrée suivant (optionnel)

Les fichiers d'entrée déclarés comme paramètres doivent exister. Dans le cas contraire, l'utilisateur reçoit le message "<*filename*> `: fichier inexistant`". Ils peuvent être désignés par leur chemin absolu ou par leur chemin relatif par rapport au répertoire d'où la commande `conflex` est lancée. Leur noms sont librement choisis par l'utilisateur. L'extension `.csp` est conseillée mais non obligatoire.

Le logiciel écrit des informations dans la *sortie standard* qui, par défaut, est l'écran. Il est possible de garder la trace de ces informations en *redirigeant* la sortie standard vers un fichier. Chaque système d'exploitation possède en principe une telle commande de redirection.

Exemple, sous Unix, de redirection dans le fichier `toto.out` :

```
conflex toto1.csp toto2.csp > toto.out
```

Dans cet exemple, le fichier de sortie redirigée ne doit pas être déjà présent dans le répertoire. S'il est présent, l'utilisateur reçoit le message "`toto.out : File exists`". Trois options sont alors possibles :

– changer le nom du fichier dans la commande ;

– supprimer le fichier, avant de le réutiliser pour la redirection des sorties ;

– utiliser la commande ">!" à la place de ">".

2.3.2 Ordonnancement des déclarations dans les fichiers de données

Les fichiers d'entrée contiennent la requête de résolution exprimée par l'utilisateur puis la description du CSP (les variables puis les contraintes). Dans le cas où la description du CSP est fractionnée en plusieurs fichiers (par exemple trois fichiers pour, respectivement, la requête, les variables et les contraintes), ils seront lus et interprétés par le système dans l'ordre de leur apparition dans la ligne de commande. Dans tous les cas, l'ordre de lecture des trois ensembles suivants doit être respecté, mais l'ordre des déclarations à l'intérieur de chacun des trois ensembles est indifférent pour la résolution.

1. Déclaration des **requêtes**, des **options** et des **paramètres** :

   – les *requêtes* : il s'agit des actions que l'utilisateur désire appliquer au CSP (filtrage, ultra-filtrage, sauvegarde, recherche arborescente des solutions, trace) ;

- les *options* : heuristiques de choix (statique ou dynamique) des variables à *instancier*[4] et des valeurs à examiner, intensité de la trace de la résolution ;
- les *paramètres* : précision de la représentation des variables réelles, degré de satisfaction minimum pour qu'une solution soit acceptée.

2. Déclaration des **variables**, parmi 3 types possibles :
   - variable dite *symbolique*, dont le domaine est une enumération de valeurs symboliques ;
   - variable dite *entière*, dont le domaine est une enumération de nombres entiers ;
   - variable dite *réelle*, dont le domaine est une disjonction d'intervalles.
3. Déclaration des **contraintes**, liant les variables précédemment déclarées, parmi 6 types de contraintes possibles :
   - contrainte dite *en extension*, dont la *relation* est la liste exhaustive des combinaisons possibles des valeurs des variables liées ;
   - contrainte dite *en intension*, dont la *relation* est une équation entre les variables liées ;
   - contrainte dite *conditionnelle*, où les contraintes de la conclusion participent au CSP dès que les contraintes de la prémisse sont satisfaites ;
   - contrainte dite *par morceaux*, c'est-à-dire exprimant une relation différente selon différents domaines de définition des variables ;
   - *conjonction* de contraintes, pour exprimer des "*sous-CSP*" (et les placer dans une disjonction de contraintes par exemple);
   - *disjonction* de contraintes, pour pouvoir exprimer le "*OU*" entre des contraintes.

2.4 Exemple de problème

Il s'agit d'établir, à l'intention d'une cantatrice, un menu diététique, agréable, suffisamment énergétique mais sans trop de calories apportées. On pose que le menu sera à base de toasts, d'œufs, de lait et de thé. Il faut trouver leurs quantités respectives, telles que l'apport calorique total soit compris entre 950 et 1050. Cependant, faute d'y parvenir, on se contentera, avec un *degré de satisfaction* de moins en moins élevé, d'un apport jusqu'à 900, mais pas en deçà, ou jusqu'à 1100, mais pas au-delà. On souhaite au moins deux toasts par œuf, au moins deux toasts (mais pas plus de 5) et au moins un œuf (mais pas plus de 5). Il ne faut pas boire plus de 5 décilitres de thé ou de lait. Toutes les proportions entre les deux liquides sont bonnes, sauf l'absence de l'un associée à une très petite quantité de l'autre.

On désire trouver et afficher toutes les solutions à ce problème dont le degré de satisfaction soit au moins de 0.5[5], en utilisant l'algorithme du *forward checking*, précédé d'une opération de filtrage des domaines des variables, avec un ordre particulier d'instanciation des variables. Après la phase de filtrage initial, on souhaite sauvegarder les domaines filtrés, avant qu'ils continuent à être modifiés au cours de la résolution.

---

4. La résolution consiste à parcourir l'espace des variables, à la recherche d'une combinaison de valeurs admissible. Pour une variable donnée, il essaie chaque valeur du domaine (on parle d'instanciatiation) et teste si elle respecte les contraintes. Ce test consiste à essayer chaque valeur d'une autre variable, choisie de manière heuristique, et ainsi de suite en profondeur. Lorsqu'un test est négatif à un niveau, il est inutile d'aller plus en profondeur, et on peut "remonter" pour essayer une autre valeur. L'instanciation est donc une action réversible. La représentation graphique de ce parcours est un arbre.

5. Par définition, une solution est une instanciation de toutes les variables qui satisfait toutes les contraintes. Dans le cadre flexible, une solution est une instanciation qui satisfait toutes les contraintes, au moins à un certain degré.

2.4.1 Expression du problème

```
#   CSP flou du menu de la cantatrice
#   ---------------------------------

#  REQUETES, PARAMETRES ET OPTIONS

\fuzzy_cut_step         : 0.10;
\alpha                  = 0.5;
\filtering              : f;
\search                 : fc all_solutions;
\dynamic_labeling_order : smallest_domain_by_degree;
\verbose                : display_solutions;
#\save                  : cantatrice.csp.save;

#    VARIABLES

\vi : THE,LAIT,TOAST,OEUF 0...5 ;

\vr : X 10 [500, 1500] ;

\outsol : all  alphabetic_order  display_step;

#    CONTRAINTES

\ce CE1 (1)
        THE      LAIT ,
\allbut
        0        0       (0)
        0        1       (0.6)
        1        0       (0.5)
;

\ci CI1 (1) , X = THE*75 + LAIT*100 + TOAST*300 + OEUF*310;
\ci CI2 (1) , TOAST >= OEUF*2 ;
\ci CI3 (1) , TOAST >=2 ;
\ci CI4 (1) , OEUF >= 1 ;
\ce : CI5  X, ([900,950,1050,1100]) (1);
```

2.4.2 Fichier de résultats

```
Lecture du fichier... "../pb/cantatrice.csp"
  ... OK

###### Filtering "superficiel" (AC pour les variables entieres) ...

###### Fin filtering du CSP  ######

############  Resolution par Forward-Checking  ############

  - Recherche de toutes les solutions satisfaisantes au moins à 0.500
  - Tri dynamique des variables : rapport 'taille domaine/degré' croissant.
  - Ordre d'examen des valeurs numériques : de la plus petite à la plus grande.
----------------------------------------------------
```

```
SOLUTION No 1
   LAIT = 0   OEUF = 1   THE = 1   TOAST = 2   X = 985.000 (+- 5.000)  sat = 0.500 .
  (trouvee apres 5 instanciations et 8 tests de contraintes)

SOLUTION No 2
   LAIT = 0   OEUF = 1   THE = 2   TOAST = 2   X = 1060.000 (+- 5.000)  sat = 0.800 .
  (trouvee apres 7 instanciations et 13 tests de contraintes)

SOLUTION No 3
   LAIT = 1   OEUF = 1   THE = 0   TOAST = 2   X = 1010.000 (+- 5.000)  sat = 0.600 .
  (trouvee apres 10 instanciations et 18 tests de contraintes)

############   Fin du Forward-Checking ############
   Nombre de solution(s) trouvee(s) : 3
   Nb d'instanciations : 10, Nb de tests de contraintes : 18

durée de la résolution : 0'00''24
```

Noter dans les trois solutions la valeur de la variable X, qui représente l'apport calorique total. Bien que, dans cet exemple, la valeur trouvée soit précise, une mention du type "`(+- 5.000)`" signifie, en général dans CON'*FLEX* , que la valeur solution est un intervalle centré sur la valeur affichée et dont la largeur est le *pas* de la variable (cf. section 4.3.2). La valeur suivant "`sat =`" est le *degré de satisfaction* de la solution. Les indications "`Nb d'instanciations`" et "`Nb de tests de contraintes`" [6] sont une mesure du chemin (dans la recherche arborescente) qu'il a fallu parcourir pour trouver les solutions.

6. Chaque fois que, dans la phase de recherche arborescente, toutes les variables d'une contrainte sont complètement instanciées (c'est-à-dire avec une valeur unique) le système teste si cette combinaison respecte la contrainte (voir section 5.1.2).

Chapitre 3

Déclaration des requêtes, options et paramètres

Cet ensemble de déclarations doit figurer en tête dans le ou les fichiers de description du problème car il contient des informations necéssaires pour la déclaration ultérieure des variables. A l'intérieur de cet ensemble, l'ordre des déclarations est cependant sans influence sur la résolution.

3.1 Déclaration des requêtes

Il existe deux types de requêtes : l'une pour le *filtrage initial* des variables du CSP et l'autre pour le choix de l'algorithme de *recherche arborescente*. Pour une connaissance approfondie des algorithmes dans le cadre classique des CSP, le lecteur est invité à consulter [Kum92], [Nad89], [Pro93]. Les extensions liées à l'aspect flexible des CSP et exploitées dans CON'*FLEX* sont décrites dans [FMCS92].

3.1.1 Type de filtrage

Les domaines des variables peuvent être filtrés[1] ou non, avant le début de la phase de recherche arborescente. Deux algorithmes de filtrage sont disponibles.

Syntaxe :

`\filtering` [: / =] <*algorithme*> ;

avec :

<*algorithme*> = `f` / `uf` [<*p*>]

avec les significations suivantes :

- `f` : pour un filtrage simple (*arc-consistance*[2] pour les variables symboliques et entières et *arc-consistance de bornes*[3] pour les variables réelles) ;

1. De façon générale, le filtrage consiste à enlever des domaines des variables les valeurs qui n'ont plus de chance de figurer dans des solutions, mais à l'inverse il ne doit pas éliminer de valeurs pouvant encore entrer dans une solution.

2. Un CSP est arc-consistant quand, pour chacune de ses variables, le domaine ne contient que des valeurs compatibles, selon le sens de la contrainte *binaire* en jeu, avec au moins une valeur du domaine de toute autre variable qui lui est liée. La recherche de solutions sur un CSP arc-consistant est plus rapide que sur un CSP qui ne l'est pas, mais l'établissement de l'arc-consistance peut être coûteux.

3. L'arc-consistance de bornes ([Lho94]) est une propriété d'arc-consistance vérifiée, non pas pour tous les éléments des domaines des variables réelles, mais seulement pour leurs bornes.

- `uf [<`p`>]` : pour un *ultra-filtrage*, c'est-à-dire un filtrage simple (`f`) suivi d'un filtrage renforcé par *domain-splitting* pour les variables réelles jusqu'à une profondeur p (la valeur p est optionnelle et la profondeur par defaut vaut 8). Le *domain-splitting* à une profondeur p signifie que le domaine réel (intervalle) sera decoupé en 2^p morceaux, sur lesquels on recherche l'arc-consistance de bornes. L'ultra-filtrage retire du domaine les morceaux dont aucun point ne peut figurer dans une solution (voir exemple page 33). Il ne peut pas détecter un morceaux à retirer dont la taille est inférieure à $\frac{taille\ du\ domaine}{2^p}$. Un ultra-filtrage à profondeur faible est peu coûteux, mais peut ne pas être efficace.

Exemples :

`\filtering: f ;`	(filtrage simple)
`\filtering uf 10 ;`	(filtrage simple suivi d'un ultra-filtrage à la profondeur 10)

3.1.2 Type de résolution

Trois algorithmes de résolution sont disponibles. Ils sont tous fondés sur une *recherche arborescente*, mais deux d'entre eux sont des algorithmes hybrides, en ce sens qu'ils effectuent un certain niveau de filtrage à chaque instanciation de variable dans le développement de l'arbre.

Syntaxe :

`\search` [: / =] <*algorithme*> [<*solution*>] ;

avec :

<*algorithme*> = `bt / fc / rfla`
<*solution*> = `first_solution / first_solutions [<`n`>] / all_solutions / best_solutions`

avec les significations suivantes :

- `bt` : recherche de solution par l'algorithme *backtrack* ;
- `fc` : recherche de solution par l'algorithme *forward-checking* ;
- `rfla` : recherche de solution par l'algorithme *real full look-ahead*[4].

et

- `first_solution` : recherche d'une seule solution (option retenue par défaut);
- `first_solutions [<`n`>]` : recherche des n premières solutions[5] ;
- `all_solutions` : recherche de toutes les solutions;
- `best_solutions` : recherche de la meilleure solution ; plusieurs solutions peuvent être affichées, mais toujours dans un ordre croissant du degré de satisfaction (on dit aussi degré de cohérence)).

4. Il n'y a aucun filtrage pour `bt` au cours de la recherche arborescente; dans le `forward-checking`, les variables non encore instanciées sont filtrées une fois et une seule; avec le `real full look-ahead`, le filtrage d'une variable non encore instanciée peut à son tour entraîner le filtrage d'une autre, jusqu'à stabilisation des domaines. Le filtrage d'une variable réelle n'entraîne le filtrage des variables qui lui sont liées que si son domaine a été suffisamment modifié, en l'occurrence si une borne a été déplacée d'une valeur au moins égale au dixième du pas de la variable (voir définition page 24.

5. Pour un ordre d'instanciation donné (voir sections 3.2.2 et 3.2.3), l'ordre de découverte des solutions est déterminé, mais il peut être différent pour deux ordres d'instanciation différents. Les solutions affichées avec les options `first_solution` et `first_solutions [<`n`>]` ne sont pas nécessairement celles de plus forts degrés de satisfaction.

Exemples :

`\search = bt first_solutions 4 ;` (recherche des 4 premières solutions par l'algorithme *backtrack*)
`\search : fc best_solution ;` (recherche de la meilleure solution par l'algorithme *forward-checking*)

3.2 Déclaration des options

Les options portent sur la sauvegarde sur fichier de l'état des variables (domaines) et des contraintes après un filtrage initial, sur l'ordre d'instanciation des variables lors de la recherche arborescente de solutions, et sur le suivi des étapes de la résolution.

3.2.1 Sauvegarde dans un fichier

Après un filtrage initial, il peut être intéressant de sauvegarder l'état du CSP. Cela permet en effet de lancer plusieurs recherches de solutions, par exemple avec des algorithmes différents, en économisant (sauf la première fois) cette phase de filtrage. La procédure de sauvegarde écrit dans un fichier (dans cet ordre) les paramètres, les requêtes, les options (sauf l'option de sauvegarde), les variables et enfin les contraintes. Une fois la sauvegarde effectuée, il suffit de lancer la résolution avec le nom du fichier de sauvegarde comme argument de la commande `conflex` (voir section 2.3).

Syntaxe :

`\save` [: / =] *<nom du fichier>* ;
avec :
<nom du fichier> : une chaîne de caractères (cf. section 2.1)

Exemple :

`\save : toto.csp ;` (sauvegarde le CSP dans le fichier `toto.csp`, après le filtrage demandé)

Remarque : la sauvegarde du CSP après un filtrage est une sorte de “photographie” de celui-ci. Le CSP de ce fichier de sauvegarde est donc dépendant des paramètres de départ tels que le pas de précision des α-coupes `\fuzzy_cut_step`, le seuil d'acceptation `\alpha` (cf. section 3.3) et les *pas* de précision des variables (cf. section 4.3.2). Il ne faut donc pas les modifier après la sauvegarde.

3.2.2 Heuristique de choix statique de la variable à instancier

Il s'agit d'établir l'ordre d'instanciation des variables avant le lancement de la procédure de recherche. Cet ordre, qualifié d'heuristique, est celui qui, du point de vue de l'utilisateur, va entraîner la plus forte diminution de la taille de l'espace de recherche. Pour un type de problème donné, il existe en général un ordre meilleur que les autres. Il est difficile de le connaître avec certitude et *a priori*, seulement en fonction des caractéristiques du problème. L'utilisateur peut donc essayer différentes heuristiques sur une instance particulière de son problème, puis adopter la meilleure (en termes de temps d'exécution, de nombre d'instanciations ou de nombre de tests de contraintes) sur les autres [6].

6. Il n'est pas donné à l'utilisateur la possibilité de définir lui-même une heuristique qui lui paraît bien adaptée à son type de problème, car cet ajout nécessite une modification du code définissant le vocabulaire et une nouvelle compilation de l'ensemble du logiciel.

Syntaxe :

\static_labeling_order [: / =] <*type d'ordre*> ;
avec :
<*type d'ordre*> = smallest_domain / greatest_degree / smallest_domain_by_degree

avec les significations suivantes :

- `smallest_domain` : place d'abord les variables dont la *taille* du domaine est la plus petite (nombre d'éléments dans les domaines de variables symboliques et entières ; nombre d'intervalles lors du découpage du domaine des variables réelles avec leur pas ; voir la section 4.3.2) ;
- `greatest_degree` : place d'abord les variables dont le *degré* est le plus grand (c'est-à-dire celles qui sont impliquées dans le plus grand nombre de contraintes) ;
- `smallest_domain_by_degree` : place d'abord les variables dont la valeur $\frac{taille\ du\ domaine}{degre}$ est la plus petite.

Exemple :

`\static_labeling_order = smallest_domain_by_degree ;`

Si une heuristique d'ordre statique n'est pas fournie, l'ordre d'instanciation sera celui de la déclaration des variables dans le fichier d'entrée.

3.2.3 Heuristique de choix dynamique de la variable à instancier

Ce type d'heuristique modifie l'ordre d'instanciation des variables pendant la procédure de recherche. Elle se justifie par le fait que la procédure de recherche (en particulier le filtrage suivant éventuellement l'instanciation d'une variable) réduit le domaine des variables, diminuant ainsi les valeurs de la taille du domaine et du rapport $\frac{taille\ du\ domaine}{degre}$.

Syntaxe :

\dynamic_labeling_order [: / =] <*type d'ordre*> ;
avec :
<*type d'ordre*> = smallest_domain / smallest_domain_by_degree

avec les mêmes significations que pour l'heuristique de choix statique.

Exemple :

`\dynamic_labeling_order = smallest_domain ;`

Si une heuristique de choix dynamique n'est pas fournie, l'ordre statique sera utilisé tout au long de la résolution, quels que soient les changements apportés aux domaines des variables.

Remarques :

- Lorsqu'on utilise à la fois une heuristique de choix statique et une heuristique de choix dynamique, l'effet de la première sera annulée dès la première instanciation : il est donc presque inutile d'utiliser les deux à la fois.

– Une heuristique de choix dynamique est sans objet si on utilise l'algorithme `bt`, puisque celui-ci ne modifie pas le domaine des variables.

3.2.4 Heuristique d'ordre d'examen des valeurs

Une fois choisie la variable à instancier, à une certaine profondeur dans l'arbre de recherche, et à supposer qu'elle soit numérique et non symbolique, les valeurs de cette variable peuvent être examinées (c'est-à-dire donner lieu à un développement de l'arbre) selon quatre ordres différents.

Syntaxe :

\\`value_order` [: / =] <*type d'ordre*> ;

avec :

<*type d'ordre*> = `bottom_first / top_first / edges_first / mid_first`

avec les significations suivantes :

– `bottom_first` : examen des valeurs par ordre croissant (option par défaut) ;

– `top_first` : examen des valeurs par ordre décroissant ;

– `edges_first` : itérativement, examen de la valeur la plus faible non encore examinée, puis de la valeur la plus forte non encore examinée ;

– `mid_first` : examen de la valeur centrale (ou d'une des deux valeurs centrales) puis, itérativement examen de la plus forte valeur à gauche non encore examinée, puis de la plus faible valeur à droite non encore examinée.

Exemple :

Si le domaine d'une variable est la liste d'entiers $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, et qu'on spécifie :

\\`value_order = mid_first ;`

l'ordre d'examen des valeurs sera le suivant : 5, puis 4, 6, 3, 7, 2, 8, 1, 9 et enfin 0.

Remarques :

1. Cas où la variable à instancier est réelle.

   – lorsque le domaine de la variable est compact, il est découpé en une liste de sous-intervalles dont la largeur est le *pas* de la variable (voir définition page 24); les valeurs considérées pour l'ordonnancement sont les bornes inférieures des sous-intervalles.

   – lorsque le domaine de la variable est un ensemble d'intervalles compacts disjoints, il y a deux niveaux de découpage ; les valeurs considérées au premier niveau de l'ordonnancement sont les bornes inférieures des intervalles compacts ; au deuxième niveau, chaque intervalle compact est à son tour découpé en sous-intervalles, ordonnés selon le même critère. Les options `bottom_first` et `top_first` génèrent au total une séquence de sous-intervalles qui respecte complètement le critère définissant l'option. On voit facilement qu'il n'en est pas de même pour les options `edges_first` et `mid_first`, si la largeur d'un des intervalles compacts est supérieure au *pas* de la variable.

2. Cas où la variable à instancier est entière.
   Le domaine de la variable est constitué de toutes ses valeurs possibles au début de la résolution. Lorsqu'au cours de la résolution une valeur devient impossible, l'arbre de recherche ne sera pas développé à partir d'elle, mais elle reste dans le domaine avec un degré 0. Ainsi, dans l'exemple ci-dessus, si les valeurs 0, 1 et 2 sont devenues impossibles, l'ordre d'examen des valeurs sera le suivant : 5, puis 4, 6, 3, 7, 8, et enfin 9.

3.2.5 Suivi (trace) de la résolution

Il est parfois utile de connaître divers résultats intermédiaires au cours d'un filtrage, ou bien la séquence des étapes d'une résolution. L'option suivante permet l'écriture de telles informations dans la sortie standard (l'écran par défaut, ou le fichier spécifié dans la commande `conflex`).

Syntaxe :

`\verbose` [: / =] <*type de suivi*> ;

avec :

<*type de suivi*> = `display_solutions` | `display_csp`
| `display_filtering` | `display_ultra_filtering`
| `display_search_tree` | `display_search`
| `display_intervals` | `display_all`

avec les significations suivantes :

- `display_solutions` : affiche les solutions [7] ;
- `display_csp` : affiche le CSP (variables et contraintes) avant et après le filtrage initial ;
- `display_filtering` : suivi du filtrage ;
- `display_ultra_filtering` : suivi du l'ultra-filtrage ;
- `display_search_tree` : trace de l'arbre des instanciations, avec indication de la profondeur ;
- `display_search` : suivi détaillé de la recherche de solution ;
- `display_intervals` : suivi du calcul d'intervalles ;
- `display_all` : cumul des effets de toutes les options précédentes.

Exemple :

`\verbose : display_solutions display_filtering;` (suivi du filtrage et affichage des solutions)

7. Cette option est généralement présente, sauf dans les cas où on ne souhaite réaliser que le filtrage initial et, éventuellement, une sauvegarde.

3.2.6 Affichage des solutions

Les solutions du problème sont affichées, sur l'écran ou dans un fichier, au fur et à mesure de leur découverte par CON'*FLEX* . Il est possible de n'afficher, pour chaque solution, qu'un sous-ensemble des variables du problème. De plus, cet affichage peut se réaliser de différentes manières.

Syntaxe :

\outsol [: / =] <*liste d'affichage*> [<*mode d'affichage*>] ;

avec :

<*liste d'affichage*> =	`all` / <*liste de noms de variables*>
<*liste de noms de variables*> =	. . . , <*nom*$_j$> , . . .
<*mode d'affichage*> =	<*mode de tri*> | <*affichage du pas*>
<*mode de tri*> =	`alphabetic_order` / `no_alphabetic_order`
<*affichage du pas*> =	`display_step` / `no_display_step`

avec les significations suivantes :

- `all` : affichage de toutes les variables ;
- *liste de noms de variables* : affichage des variables listées ; si cette option est utilisée, la commande `\outsol` doit être placée après la déclaration des variables ;
- `alphabetic_order` : les variables sont affichées par ordre alphabétique ;
- `no_alphabetic_order` : les variables sont affichées dans un ordre quelconque, mais constant pour toutes les solutions ;
- `display_step` : pour les variables réelles, la valeur est donnée sous la forme d'un intervalle, avec la valeur du demi-*pas* de la variable ;
- `no_display_step` : pour les variables réelles, la valeur affichée est seulement le centre de l'intervalle obtenu avec l'option ci-dessus ;

Exemples :

`\outsol : THE, LAIT alphabetic_order;` (affichage de la variable `LAIT` puis de la variable `THE`)

`\outsol : THE, LAIT;` (affichage des deux variables dans un ordre quelconque)

`\outsol : all alphabetic_order;` (affichage de toutes les variables par ordre alphabétique)

En l'absence de toute indication, les variables sont toutes affichées, dans un ordre quelconque, et les variables réelles sont affichées sans mention du *pas*.

3.3 Déclaration des paramètres

Deux caractéristiques de CON'*FLEX* sont paramétrables : la précision de la représentation des domaines flous des variables réelles et le degré minimum d'acceptabilité d'une solution.

Syntaxe :

– La précision de la représentation en *α-coupes*[8] (elle vaut 0.1 par défaut) :

fuzzy_cut_step [: / =] <*distance*> ;
avec :
<*distance*> entre 0 et 1

La précision est exprimée par la distance qui sépare deux α-coupes successives dans la représentation d'un ensemble flou sur les réels. Si la distance vaut 0.1, il y a au plus 11 α-coupes (de celle de degré 0 à celle de degré 1). Si la distance vaut 0.2, il y a au plus 6 α-coupes. Plus la distance est petite, plus la précision des calculs sera grande, mais plus de temps prendront les calculs sur les variables réelles.

– Le *degré minimum d'acceptabilité* (il vaut 0.5 par défaut) :

alpha [: / =] <*degré minimum*> ;
avec :
<*degré minimum*> entre 0 et 1

C'est le degré de satisfaction des contraintes par une solution, en dessous duquel l'utilisateur ne souhaite pas connaître cette solution. Plus élevée est cette valeur, plus rapide est la résolution (car les domaines des variables sont plus rapidement réduits), mais plus grand est le risque de n'exhiber aucune solution, fût-ce de degré relativement faible.

Exemples :

`\fuzzy_cut_step = 0.05 ;`	(pour affiner le pas de précision des α-coupes)
`\alpha : 0.85 ;`	(pour augmenter le degré minimum d'acceptabilité à 0.85)

8. Ce paramètre fait référence au choix, fait dans CON'*FLEX* , de représenter un ensemble flou sur les réels par une superposition de segments (voir section 4.3). La position d'un segment dans l'empilement est conventionnellement appelée α et prend ses valeurs entre 0 et 1. Les segments sont ainsi appelés α-coupes.

Chapitre 4

Déclaration d'une variable

La déclaration des variables se place après celle des requêtes (car le système doit connaître le pas de précision des α-coupes au moment d'interpréter les déclarations de variables) et avant celle des contraintes (car les contraintes font référence aux variables).

4.1 Déclaration d'une variable symbolique

Une variable symbolique possède un domaine (éventuellement flou) constitué d'une énumération de symboles, représentés par des chaînes de caractères.

Chaque symbole possède *un degré d'appartenance* au domaine[1]. Ce degré vaut 1 par défaut, mais il est possible d'expliciter sa valeur (entre 0 et 1) entre parenthèses après chaque symbole.

4.1.1 Syntaxe

`\variable_symbolic` [: / =] $<\mathit{nom}>$ $\ldots$ $<e_i>$ [($<d_i>$)] $\ldots$;

avec :

$<\mathit{nom}>$: le nom de la variable ;
$<e_i>$ le i^{ème} élément symbolique du domaine et la valeur optionnelle $<d_i>$ (avec $d_i \in [0, 1]$) le degré d'appartenance (1 par défaut).

Il est possible de déclarer conjointement des variables symboliques ayant le même domaine. Les variables sont alors séparées par des virgules.

`\variable_symbolic` [: / =] $\ldots$, $<\mathit{nom}_j>$, $\ldots$ $\ldots$ $<e_i>$ [($<d_i>$)] $\ldots$;

avec :

$<\mathit{nom}_j>$: le nom de la j^{ème} variable déclarée ;
les autres symboles ont la même signification que ci-dessus.

1. Dans la théorie classique, étant donné un ensemble, un élément lui appartient complètement ou ne lui appartient pas du tout. Quand il s'agit d'un ensemble flou, un élément lui appartient à un certain degré, conventionnellement compris entre 0 et 1.

Le mot-clé pour indiquer la déclaration d'une variable symbolique est `\variable_symbolic`, mais deux abréviations peuvent être utilisées pour faciliter la saisie : `\var_symb` et `\vs`.

4.1.2 Exemple

```
\vs :   Taille   tres_petit (0.75)  petit  moyen  grand  tres_grand (0.7) ;
```

Dans cet exemple, l'utilisateur exprime que la variable `Taille` peut prendre une ou plusieurs valeurs parmi cinq, et que les valeurs `petit`, `moyen` et `grand` sont complètement possibles, alors que la valeur `tres_petit` n'est possible (ou vraisemblable, ou préférée) qu'au degré 0.75 [2].

Dans l'exemple suivant, on déclare deux variables symboliques ayant le même domaine.

```
\vs :   maillot,short    adidas  nike  umbro ;
```

2. Une solution où la variable aura la valeur `petit` (ou `moyen` ou `grand`) pourra être totalement satisfaisante, alors qu'une autre où la variable `Taille` aura la valeur `tres_grand` ne le sera au mieux qu'au degré 0.7. Ce degré pourra être plus faible, par exemple si la solution comporte, pour d'autres variables, des valeurs dont le degré d'appartenance au domaine est plus petit que 0.7.

4.2 Déclaration d'une variable entière

Une variable entiere possède un domaine (éventuellement flou) constitué d'un ensemble de nombres entiers. La déclaration s'effectue par la simple énumération des nombres un à un (éventuellement suivis de leur *degré d'appartenance* entre parenthèses) ou par des intervalles (éventuellement suivis du degré d'appartenance entre parenthèses valable pour tous les entiers de l'intervalle).

4.2.1 Syntaxe

`\variable_integer` [: / =] <*nom*> ... <*e*> [(<d_1>)] | <*inf*>`..`<*sup*> [(<d_2>)] ... ;

avec :

<*nom*> : le nom de la variable ;
<*e*> : un nombre entier du domaine et la valeur optionnelle ($d_1 \in [0, 1]$) qui est son degré d'appartenance (1 par défaut).
<*inf*>`..`<*sup*> : respectivement les bornes inférieure et supérieure de l'intervalle d'entiers inclus dans le domaine et la valeur optionnelle <d_2> ($d_2 \in [0, 1]$) qui est le degré d'appartenance de chacun de ces entiers au domaine (1 par défaut)

On peut déclarer conjointement des variables entières ayant le même domaine. La declaration des variables se fera alors en déclarant les différents noms de variables, separés par des virgules, juste avant leur domaine.

`\variable_integer` [: / =] ..., <*nom*$_j$> , ... <*e*> [(<d_1>)]
| <*inf*>`..`<*sup*> [(<d_2>)] ... ;

avec :

<*nom*$_j$> : le nom de la j^{ème} variable déclarée ;
Les autres symboles ont la même signification que ci-dessus.

Les deux abréviations du mot-clé `\variable_integer` sont `\var_int` et `\vi`.

4.2.2 Exemple

```
\vi =   I    2 (0.9)  -1  4..6 (0.85)  -3  ;
```

Cette déclaration représente la variable entière `I` de domaine $D_I = \{-3_{1.0}, -1_{1.0}, 2_{0.9}, 4_{0.85}, 5_{0.85}, 6_{0.85},\}$). L'interprétation de ces degrés et leur prise en compte dans la résolution suivent les mêmes principes que pour les variables symboliques (cf. section 4.1.2).

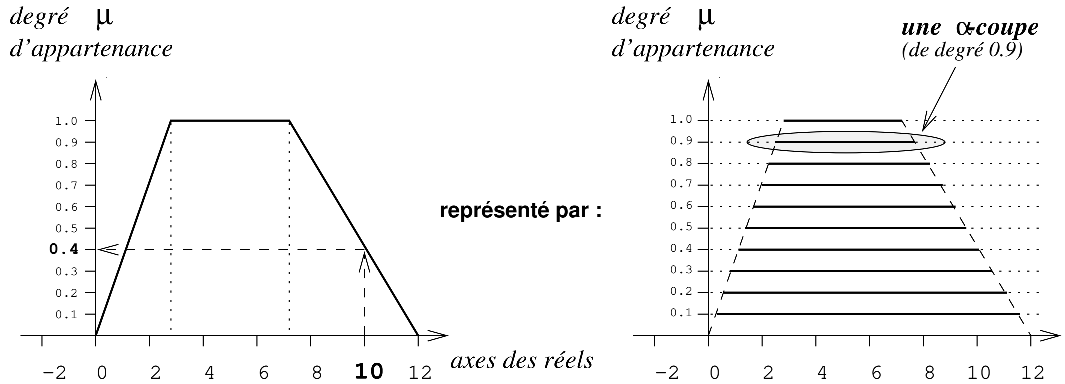
La déclaration suivante est une déclaration de deux variables entières ayant le même domaine.

```
\vi =   V1,V2    1  2..6 (0.8)  ;
```

4.3 Déclaration d'une variable réelle

Une variable réelle possède un domaine constitué d'un ou de plusieurs intervalles (éventuellement flous) sur les réels. Le domaine est représenté par un ensemble d'α-coupes. Une α-coupe est une disjonction ordonnée de segments disjoints. Chaque segment est représenté par deux nombres flottants : ses bornes inférieure et supérieure.

Il existe deux manières de déclarer le domaine d'une variable réelle : une suite d'intervalles flous résumés par des rectangles ou des trapèzes (s'élevant jusqu'à une certaine hauteur), ou une liste explicite d'α-coupes.



Dans cet exemple, le domaine est déclaré comme un trapèze de *noyau*[3] [2.8, 7.2] et de *support* [0, 12]. Comme conséquence de sa représentation interne par une liste d'α-coupes, le degré d'appartenance de la valeur 10 est $\mu = 0.4$, alors que celui de la valeur $10 + \epsilon$ est $\mu = 0.3$. Cette discontinuité est une cause d'imprécision dans le calcul sur les variables réelles[4].

3. Le noyau d'un intervalle flou est l'ensemble des valeurs qui appartiennent complètement à l'ensemble (cette notion n'est utilisée que pour les intervalles normalisés, c'est-à-dire de hauteur 1) ; le support est l'ensemble des valeurs qui appartiennent à l'intervalle avec un degré strictement supérieur à 0.

4. Supposons dans l'exemple ci-dessus que le seuil `\alpha` soit fixé à 0.95. Le filtrage va chercher à éliminer les valeurs de la variable (sur l'axe des réels) telles que leur degré d'appartenance soit inférieur à ce seuil. Le domaine résultant sera celui qui correspond à l'α-coupe de niveau $\mu = 1.0$. Si le paramètre `\fuzzy_cut_step` avait été de 0.05, le système aurait effectué un filtrage strictement au niveau `\alpha`.

4.3.1 Syntaxe de la déclaration par intervalles

`\variable_real` [: / =] $<nom>$ $<pas>$

$\vdots$

$<domaine_i>$

$\vdots$

;

avec :

$<nom>$: le nom de la variable ;
$<pas>$: le pas de la variable (0.1 par défaut) ;
$<domaine_i>$: le i^{ème} morceau du domaine de la variable, qui sera dans ce cas un intervalle (flou ou non) d'une des formes suivantes :

- `[`$<a>$`, `$<b>$`]` : un intervalle $[a, b]$[5] représentant le noyau et le support de l'intervalle flou de hauteur 1 (par défaut) ;
- `[`$<a>$`, `$<b>$`, (`$<d>$`)]` : un intervalle $[a, b]$ et la hauteur d (entre 0 et 1) de la dernière α-coupe ;
- `[`$<s_1>$`, `$<n_1>$`, `$<n_2>$`, `$<s_2>$`]` : quatre nombres représentant le support $[s_1, s_2]$ et le noyau de l'intervalle flou $[n_1, n_2]$ de hauteur 1 (par défaut) ;
- `[`$<s_1>$`, `$<n_1>$`, `$<n_2>$`, `$<s_2>$`, (`$<d>$`)]` : le support $[s_1, s_2]$, la plus haute α-coupe $[n_1, n_2]$ et la hauteur d (entre 0 et 1) de celle-ci.

Pour chacune des quatre formes, l'intervalle ne comprendra pas les bornes si on remplace le symbole "`[`" par le symbole "`]`" (à gauche) et le symbole "`]`" par le symbole "`[`" (à droite)[6].

On peut déclarer conjointement des variables réelles ayant le même domaine. La déclaration des variables se fera alors en énumérant les différents noms de variables, separés par des virgules, juste avant leur domaine. La déclaration du pas de précision peut alors être spécifique à chaque variable, et dans ce cas il accompagne la variable sur laquelle il porte. Sinon, il est commun à toutes les variables et est déclaré en fin de la liste des variables. On a donc deux autres syntaxes possibles :

`\variable_real` [: / =] $\ldots, <nom_j> <pas_j>, \ldots$

$\vdots$

$<domaine_i>$

$\vdots$

;

avec :

$<nom_j>$: le nom de la j^{ème} variable déclarée ;
$<pas_j>$: le pas de la j^{ème} variable déclarée ;
$<domaine_i>$: le i^{ème} morceau du domaine de la variable, de l'une des formes citées dans le premier tableau.

5. Si la borne exprimée est un nombre réel possèdant une représentation exacte sur la machine, ce nombre sera effectivement la borne de l'intervalle. Sinon, la borne inférieure (resp. supérieure) de l'intervalle sera le plus grand (resp. petit) nombre représentable inférieur (resp. supérieur) à la valeur exprimée par l'utilisateur.

6. Le plus petit (resp. le plus grand) nombre inclus dans l'intervalle sera le plus petit (resp. le plus grand) nombre représentable strictement supérieur (resp. inférieur) à la valeur exprimée par l'utilisateur.

`\variable_real` [: / =] ..., $< nom_j >$, ... <*pas*>

$\vdots$

$< domaine_i >$

$\vdots$

;

avec :

$<nom_j>$: le nom de la j^{ème} variable déclarée ;
<*pas*> : le pas commun à toutes les variables ;
$<domaine_i>$: le i^{ème} morceau du domaine de la variable, de l'une des formes citées dans le premier tableau.

Les deux abréviations du mot-clé `\variable_real` sont `\var_real` et `\vr`.

Le *pas* de la variable[7] est une valeur de même dimension que la variable. Il exprime la différence minimale qui doit exister entre deux valeurs de la variable pour que l'utilisateur considère ces deux valeurs comme différentes. Cette caractéristique est utilisée par CON'*FLEX* lors du découpage du domaine en sous-intervalles (dans la recherche arborescente). Les intervalles générés ont une largeur égale au pas de la variable[8].

7. ...à ne pas confondre avec le pas de précision du découpage d'un intervalle flou en α-coupes (voir section 3.3 et ci-dessous)

8. Plus l'utilisateur est exigeant quant à la précision des valeurs réelles dans les solutions, plus large sera l'arbre de recherche et, par conséquent, plus long sera son parcours.

4.3.2 Syntaxe de la déclaration par α-coupes

Les quatre possibilités de déclaration ci-dessus sont simples à utiliser mais contraignent l'utilisateur à n'exprimer des domaines réels que par des intervalles de forme simple (rectangulaire ou trapézoïdale), de hauteur quelconque. Pour lever cette limitation, il existe le moyen (certes fastidieux) de déclarer un domaine réel flou en exprimant une à une toutes ses α-coupes (en commençant par celle de degré 1) :

`\variable_real` [: / =] <*nom*> <*pas*>
⋮
<*alpha-coupe*$_i$ >
⋮
;

avec :

<*nom*> : le nom de la variable ;
<*pas*> : le pas de précision de la variable (0.1 par défaut) ;
<*alpha-coupe*$_i$ > : l'α-coupe de niveau α=i, déclarée comme suit :
`\acut` suivi d'une série d'intervalles `[`<$a_{i,k}$>`,` <$b_{i,k}$>`]` : déclaration du $k^{ème}$ segment de l'α-coupe ; les segments doivent être disjoints et ordonnés ; cette alpha-coupe peut éventuellement être vide, auquel cas on notera l'intervalle vide : `[]`.

Le nombre d'α-coupes générées par le système dépend de la précision demandée (`\fuzzy_cut_step`). La première α-coupe correspond au degré 1 du domaine réel flou et les suivantes sont celles immédiatement inférieures selon la discrétisation définie par la précision `\fuzzy_cut_step`. Si la dernière α-coupe dans la liste des `\acut` correspond à un degré strictement supérieur à 0, le système génère les α-coupes suivantes à l'identique (voir le dernier parmi les exemples ci-dessous).

Il est possible d'utiliser une syntaxe permettant de déclarer en même temps des variables ayant le même domaine. On a également deux formes :

`\variable_real` [: / =] ..., <*nom*$_j$ > `,` ...<*pas*>
| ..., <*nom*$_j$ > <*pas*$_j$ > `,` ...
⋮
<*domaine*$_i$ >
⋮
;

avec :

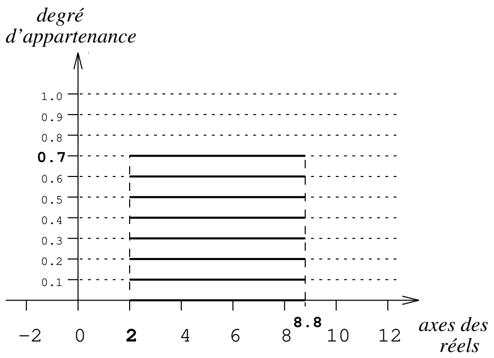
<*nom*$_j$ > : le nom de la j^{ème} variable déclarée ;
<*pas*> : le pas de précision commun à toutes les variables (0.1 par défaut) ;
<*pas*> : le pas de précision de la j^{ème} variable (0.1 par défaut) ;
<*domaine*$_i$ > : a la même signification que ci-dessus.

<u>Remarque</u> : L'α-coupe d'un niveau donné doit être incluse (non strictement) dans l'α-coupe du niveau immédiatement inférieur. Le non respect de cette règle peut entraîner des résultats inatendus lors de la résolution. Cette propriété est assurée par le système lorsque le domaine est déclaré comme un rectangle ou un trapèze.

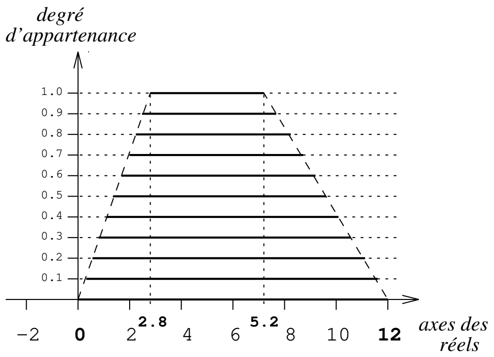
4.3.3 Exemples

Dans tous les exemples suivants, la précision du découpage en α-coupes est de 0.1.

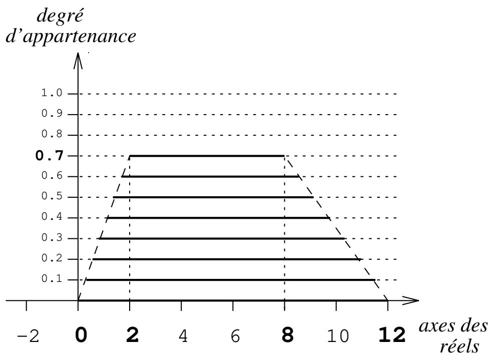
```
\vr :  X,Y   0.1   [2, 8.8, (0.7)] ;
```



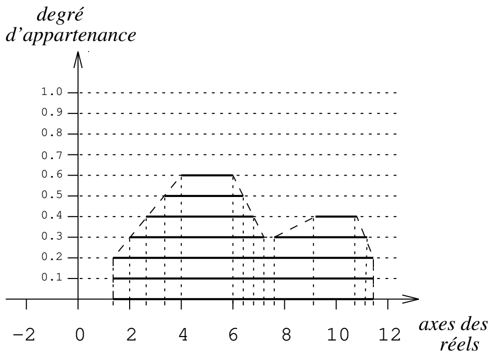

```
\vr :  X   0.1   [0, 2.8, 5.2, 12] ;
```




```
\vr :  X   0.1   [0, 2, 8, 12, (0.7)] ;
```




```
\vr :   X   0.1
        \acut  []
        \acut  []
        \acut  []
        \acut  []
        \acut  [4, 6]
        \acut  [3.3, 6.4]
        \acut  [2.6, 6.8]  [9.2, 10.7]
        \acut  [2, 7.2]  [7.6, 11]
        \acut  [1.2, 11.5] ;
```



Chapitre 5

Déclaration d'une contrainte

De façon générale, une contrainte est caractérisée par l'ensemble des variables sur lesquelles elle porte et par l'expression des combinaisons de valeurs que cette contrainte autorise. Cette expression est appelée *relation*. Il existe six types de contraintes, différenciés par la forme de la relation entre les variables. Les contraintes sont déclarées après les variables, car au moment d'interpréter la déclaration d'une contrainte, le système doit connaître certains descripteurs des variables liées par cette contrainte.

Les contraintes en extension et en intension possèdent un *degré de priorité* (compris entre 0 et 1) permettant d'exprimer son importance par rapport aux autres :

- un degré de priorité de 1 signifie que toute violation de la contrainte, même très faible, aura une répercution sur le degré de satisfaction des solutions

- un degré de priorité de 0 signifie qu'on accepte que la contrainte soit complètement violée sans que le degré de satisfaction des solutions en soit affecté (ce qui équivaut à l'absence de la contrainte)

- entre les deux, un degré de priorité de 0.3, par exemple, signifie qu'aucune violation de la contrainte ne pourra affecter la satisfaction d'une solution au-delà de la valeur 0.7 ; en d'autres termes, toute combinaison de valeurs sera au moins satisfaisante, vis-à-vis de cette contrainte, au degré 0.7.

Les contraintes par morceaux pourront être affectées d'un degré de priorité dans une prochaine version du logiciel. Ce degré n'est pas défini pour les contraintes conditionnelles et pour les conjonctions et les disjonctions de contraintes : ce sont les contraintes en conclusion ou celles qui les composent qui sont chacune affectées d'un degré de priorité.

5.1 Déclaration d'une contrainte en extension

La relation d'une contrainte en extension est la liste explicite et exhaustive des combinaisons de valeurs possibles pour les variables sur lesquelles cette contrainte pèse.

La déclaration débute avec la liste des n variables ($V_1, \ldots, V_n$ de types quelconques) sur lesquelles pèse la contrainte (c'est donc une contrainte d'*arité n*) puis continue avec la liste des n-uplets qui constituent la relation (c'est un sous-ensemble du produit cartésien $D_1 \times \ldots \times D_n$ où D_i est le domaine de la variable V_i).

Enfin, à chaque n-uplet, on associe un *degré de compatibilité* avec le sens véhiculé par la contrainte, c'est-à-dire le niveau dans l'échelle [0,1] auquel le n-uplet de valeur $v_1, \ldots, v_n$ est possible (ou vraisemblable, ou préféré) en tant qu'instanciation des variables $V_1, \ldots, V_n$.

5.1.1 Syntaxe

`\constraint_extension` [: / =] <*nom*> [(<*priorité*>)] ... <*var*$_i$> ... ,

$\vdots$

... <*élément*$_{i,j}$> ... (<c_j>)

$\vdots$

;

avec :

<*nom*> : le nom de la contrainte ;
<*priorité*> : degré de priorité (importance) de la contrainte (optionnel et valant 1 par défaut) ;
<*var*$_i$> : la i^{ème} variable de l'ensemble des variables sur lesquelles pèse la contrainte ;
<*élément*$_{i,j}$> : la valeur de la i^{ème} variable dans le j^{ème} n-uplet de la relation ;
<c_j> : le degré de compatibilité du j^{ème} n-uplet de la relation (entre 0 et 1).

Afin d'éviter l'énumération de toutes les valeurs de n-uplets possibles pour les variables d'une contrainte, il est possible de donner implicitement la valeur de satisfaction maximale (1) pour les n-uplets non mentionnés. Dans ce cas, les variables réelles ont un traitement particulier qui sera expliqué plus loin (page 30). La syntaxe correspondante est la suivante :

`\constraint_extension` [: / =] <*nom*> [(<*priorité*>)] ... <*var*$_i$> ... ,
`\allbut`

$\vdots$

... <*élément*$_{i,j}$> ... (<c_j>)

$\vdots$

;

Toutes les déclarations ont le même sens que dans le tableau ci-dessus.

Le mot-clé pour indiquer la déclaration d'une contrainte en extension est `\constraint_extension`, mais deux abréviations peuvent être utilisées pour faciliter la saisie : `\const_ext` et `\ce`.

5.1.2 Exemple

Exemple d'une contrainte en extension de degré de priorité 0.9 et pesant sur 3 variables (c'est donc une contrainte ternaire ou d'arité 3) de 3 types différents : variable symbolique `Espece`, variable entière `Date` et variable réelle `Azote`.

```
\ce : C1 (0.9)  Espece  Date  Azote  ,
                colza   120   38                         (1)
                colza   120   34.0                       (0.8)
                ble     130   ([30, 40] [50, 60])        (1)
                ble     132   ([30, 40, 50, 60])         (1)
                orge    135   ([37, 50, (0.9)])          (1)
                orge    150   ([37, 50, (0.9)])          (0.8)
                orge    150   ([30, 40, 40, 50, (0.8)])  (1)  ;
```

Pour chaque n-uplet, les éléments correspondant à une variable de type énuméré (symbolique ou entier) sont simplement des valeurs du domaine de ces variables (comme `colza` ou `120`).

Dans le cas des variables réelles, les éléments de n-uplets sont soit des valeurs réelles précises (comme `34.0`), soit des disjonctions d'intervalles de forme rectangulaire (comme `([30, 40] [50, 60])` et `([37, 50, (0.9)])`), ou trapézoïdale (comme `([30, 40, 50, 60])` ou `([33, 44, 44, 50, (0.8)])`).

Chaque n-uplet exprime qu'un ensemble de combinaisons de valeurs sont possibles, avec un dégré déterminé de compatibilité avec la contrainte. Plus précisément :

- `colza 120 38 (1)` : si, au cours de la procédure de résolution, les variables `Espece`, `Date` et `Azote` prennent les valeurs `colza`, `120` et `38`, la contrainte sera complètement satisfaite (au degré 1).
- `colza 120 34.0 (0.8)` : pour les valeurs `colza`, `120` et `34`, la contrainte sera satisfaite au degré 0.8 [1].
- `ble 130 ([30, 40] [50, 60]) (1)` : pour les valeurs `ble`, `130` et `35` ou `55`, la contrainte sera complètement satisfaite.
- `ble 132 ([30, 40, 50, 60]) (1)` : pour les valeurs `ble`, `132` et `35`, la contrainte sera satisfaite au degré 0.5 (degré d'appartenance de la valeur 35 à l'ensemble flou trapézoïdal [30, 40, 50, 60]).
- `orge 135 ([37, 50, (0.9)]) (1)` : pour les valeurs `orge`, `135` et `40`, la contrainte sera satisfaite au degré 0.9 (degré d'appartenance de la valeur 40 à l'ensemble flou rectangulaire [37, 50, (0.9)]).
- `orge 150 ([37, 50, (0.9)]) (0.8)` : pour les valeurs `orge`, `150` et `40`, la contrainte sera satisfaite au degré 0.8 (le minimum entre le degré d'appartenance de la valeur 40 à l'ensemble flou rectangulaire [37, 50, (0.9)] et le degré de compatibilité 0.8).
- `orge 150 ([30, 40, 40, 50, (0.8)]) (1)` : pour les valeurs `orge`, `150` et `35`, la contrainte sera satisfaite au degré 0.4 (le minimum entre le degré d'appartenance de la valeur 35 à l'ensemble flou rectangulaire [30, 40, 40, 50, (0.8)] et le degré de compatibilité 1) [2]

Remarques :

- Ne pas oublier la virgule "`,`" entre la déclaration des variables et la déclaration de la relation (liste de n-uplets). Dans cet exemple, où le premier élément du premier n-uplet est une chaîne de

1. On considère dans ce qui suit que les valeurs prises par les variables appartiennent complètement à leur domaine de définition. Dans le cas d'une appartenance partielle, le degré de satisfaction de la contrainte par le n-uplet ne peut être supérieur au degré d'appartenance.

2. Le degré de satisfaction de la contrainte par une combinaison particulière de valeurs sera le plus grand des degrés calculés sur les différents n-uplets. Dans notre exemple, le n-uplet (`orge 150 35`) satisfait la contrainte au degré 0.4.
Ce degré est modulé par le degré de priorité (importance) de la contrainte. La priorité étant ici 0.9, le degré de satisfaction est conservé à 0.4. Si le degré de priorité avait été de 0.3, le degré de satisfaction aurait été porté à 0.7 ($1 - 0.3$).
La satisfaction du problème par une combinaison de valeurs est le plus petit des degrés calculés sur toutes les contraintes. Autrement dit, le degré de satisfaction d'une solution est celui de la moins satisfaite des contraintes.
Les principes de calcul énoncés ici pour les contraintes en extension restent valables pour les autres types de contraintes.

caractères (comme les noms de variable), l'élément `colza` pourrait en effet être interprété comme un nom de variable au même titre que `Azote`).

– Pour les intervalles, l'utilisation du symbole "`]`" (à gauche) et du symbole "`[`" (à droite) permet d'indiquer que la borne exprimée n'est pas incluse dans l'intervalle.

– Dans le cas d'une utilisation du mot clé `\allbut` , il faudra faire attention à la déclaration des valeurs des variables réelles. Les domaines de ces variables ne peuvent pas, dans ce cas, être découpés, c'est-à-dire qu'une combinaison de variables énumérées (entières ou symboliques), dans une contrainte comprenant une ou plusieurs variables réelles, ne pourra pas être associée à différentes valeur de domaines pour la ou les variables réelles. Il est donc nécessaire d'adapter le domaine réel pour qu'il corresponde dans sa totalité à la combinaison de variables énumérées données. Voyons ceci sur un exemple.

  Les variables du problème sont :

```
\vr : X  0.1  [2.1, 4.2 ] ;
\vi : A  0 ...5;
\vi : B  0 ...5;
```

  ... et considérons la contrainte suivante :

  "Pour le couple de variables `(A,B)`, le couple de valeurs `(0,0)` est exclu et le couple de valeurs `(0,1)` est compatible avec le segment `[2.1, 2.9 ]` au degré 0.7, avec le segment `[2.1, 2.9 ]` à un degré 0.8, et à un degré 1 avec le reste du domaine de `X`. Tous les autres couples sont complètement possibles."

  Cette contrainte pourra s'écrire, en utilisant le mot clé `\allbut`, de cette manière :

```
 \ce:  C1 (0.9)  A  B  X  ,
       \allbut
                 0  0  [2.1, 4.2 ]                                        (0)
                 0  1  ([2.1, 2.9,(0.7)][2.901, 3.499,(1)][3.5, 4.2,(0.8)])  (1)
;
```

  La déclaration suivante est erronée :

```
 \ce:  C1 (0.9)  A  B  X  ,
       \allbut
                 0  0  [2.1, 4.2 ]  (0)
                 0  1  [2.1, 2.9 ]  (0.7)
                 0  1  [3.5, 4.2 ]  (0.8)
;
```

  En effet, une même combinaison de variables énumérées est répétée : `(0,1)`. Il faut regrouper les deux triplets correspondants en un seul.

5.2 Déclaration d'une contrainte en intension

La relation d'une contrainte en intension est une expression fonctionnelle dans laquelle interviennent des variables déclarées précédemment dans le ou les fichiers d'entrée.

On distingue une contrainte en intension (`\constraint_intension`), dont l'équation qui la caractérise est composée de deux membres et d'un opérateur binaire, d'une contrainte en intension multiple (`\constraint_intension_multiple`), dont l'équation est une relation composée d'un opérateur binaire et de plusieurs membres.

5.2.1 Syntaxe d'une contrainte en intension simple

`\constraint_intension` [: / =] *<nom>* [(*<priorité>*)] , *<équation>* ;

avec :

<nom> : le nom de la contrainte ;
<priorité> : degré de priorité (importance) de la contrainte (valant 1 par défaut) ;
<équation> : la relation fonctionnelle de la contrainte, de la forme suivante :

<expression> <binop> <expression>

avec :
<binop> : un des opérateurs binaires suivants : $=$, $\leq$, $<$, $\geq$, $>$, $! =$ (ou <> : "*différent de*")[3] ;
<expression> : expression incluant des variables (entières ou réelles), des constantes entieres ou réelles (en particulier le nombre PI tel que $sin(PI) = 0$, ainsi que PI/2, PI/4 et 2PI, et la valeur infinie positive INFINITY),
des intervalles ou des intervalles flous (cf. exemples ci-après),
et les opérateurs suivants, par ordre croissant de priorité :
- opérateurs arithmétiques : $+$, $-$,
- opérateurs arithmétiques : $*$, $/$,
- opérateurs arithmétiques : ^ (puissance entière), ** (puissance réelle, à valeur dans $\mathbb{R}+$), la fonction unaire *sqrt* (racine carrée) et la fonction binaire *nthroot* ($\sqrt[n]{}$)[4] ;
- les fonctions binaires : *min*, *max*, et les fonctions unaires : $-$, *sin*, *arcsin*, *cos*, *arccos*, *ln*, *exp*, *abs* (valeur absolue).[5]

Les deux abréviations du mot-clé `\constraint_intension` sont `\const_int` et `\ci`.

3. Lors de la lecture du fichier, a ! = b (ou a <> b) est remplacé par la représentation interne |a-b| > 0 ; c'est donc cette notation que l'on retrouvera dans un fichier de sauvegarde (voir section 3.2.1).

4. $\sqrt[5]{X}$ s'écrit `nthroot(5,X)`

5. Les techniques mises en jeu dans le filtrage à travers des contraintes en intension fait appel à l'*arithmétique d'intervalle* (cf. [Moo66], [Dav87] et [Hyv92]) dont voici un aperçu :

$[a, b] + [c, d] = [a + c, b + d]$

$[a, b] - [c, d] = [a - d, b - c]$

$[a, b] \times [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$

$[a, b] \div [c, d] = [min(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}), max(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d})]$ si $0 \notin [c, d]$

5.2.2 Exemple

- Exemple d'une contrainte en intension de degré de priorité 0.8 exprimant l'équation $\frac{X}{Y} < \ln Z$:

```
\ci :   CI1   (0.8)   ,  X/Y < ln(z) ;
```

- La contrainte en intension de degré de priorité 1 (valeur par défaut) exprimant le polynôme $X^2 - 3X + 2 \in$ [-0.5, 0.5] peut s'exprimer de deux manières :
  - par une seule contrainte contenant un intervalle comme opérande :

    ```
    \ci :   CI2   ,  X^2-3*X+2 = [-0.5, 0.5] ;
    ```
  - ou par les deux contraintes

    ```
    \ci :   CI2   ,  X^2-3*X+2 >= -0.5 ;
    \ci :   CI3   ,  X^2-3*X+2 <= 0.5 ;
    ```

5.2.3 Remarques

1. L'opérateur " ^ " (puissance entière) ne doit être utilisé que si le paramètre est une constante entière explicite ou bien une variable entière, ou bien une expression à valeur entière.

2. Pour élever une expression *expr* à la puissance $1/N$, où N est entier, utiliser la fonction *nthroot* : `nthroot(N, expr)`, ou bien la fonction *sqrt* si N = 2.

3. L'utilisation de l'opérateur "**" pour élever à une puissance, même entière, limite le domaine de valeurs à $\mathbb{R}+$.

4. Les **expressions à valeur constante** sont réduites dès la lecture du fichier. Par exemple :
   (a) la contrainte `y = sin(PI/2) ;` est réduite à `y = 1 ;` ,
   (b) la contrainte `y = 2*PI*k*T ;` est réduite à `y = 6.28...*k*T ;` ,
   (c) la contrainte `y = k*2*PI*T ;` n'est pas réduite, parce que les opérateurs "*" ne portent pas sur des constantes (`k`, et `2` pour le premier ; l'expression `k*2` et `PI` pour le second),
   (d) la contrainte `y = k*(2*PI)*T ;` est réduite à `y = k*6.28...*T ;`.

5. Un problème important se pose lorsque l'équation d'une contrainte en intension possède **plusieurs occurences** de la même variable, car la dépendance entre variables n'est pas prise en compte dans CON'*FLEX* . Un exemple simple peut illustrer ce problème :

   Soit les variables

   X de domaine $D_X = [0, 1]$

   Y de domaine $D_Y = [-\infty, \infty]$

   et la contrainte

   $Y = X - X.$

   Le calcul d'intervalle assure :

   $Y = [0, 1] - [0, 1] = [-1, 1],$

ce qui est un encadrement trop large de Y, car le calcul symbolique assure évidemment que $Y = 0$.

Ce problème est gênant dans les CSP possédant des contraintes en intension de type polynôme de degré supérieur ou égal à 2 (cf. exemple en section 5.2.2) car le filtrage n'aura en général aucun effet. L'"*ultra-filtrage*" pourra alors être utilisé, mais consommera d'autant plus de temps de calcul que la discrétisation demandée sera fine (voir section 3.1.1).

Soit par exemple le CSP suivant, avec les variables

X de domaine $D_X = [-10, 10]$ et de pas 0.1,
Y de domaine $D_Y = [-10, 10]$ et de pas 0.1,

et la contrainte

$Y = X^3 - 3 * X$.

Un filtrage initial simple ne réduit le domaine de X qu'à $D_X = [-2.17; 2.21]$, alors qu'un ultra-filtrage de profondeur 8 le réduit à $D_X = [-1.83; -1.56] \cup [-0.19; 0.02] \cup [1.66; 1.87]$.

6. Une attention particulière doit être portée aux **contraintes mixtes**, liant des variables entières et réelles :

```
\vi =   I    0..9;
\vr  :  X    1    [0, 10]  ;
\ci :   ci1   ,   I = 2*X  ;
\ci :   ci2   ,   X = 2.01  ;
```

Ce premier CSP n'a pas de solution. En effet il n'existe pas de valeur de `I`, telle que `ci2` soit respectée. L'instanciation d'une variable réelle avec une valeur trop précise peut empêcher le système de proposer une solution "proche" telle que : `I = 4    X = 2`.

```
\vi =   I    0..9;
\vr  :  X    1    [0, 10]  ;
\ci :   ci1   ,   I = 2*X  ;
\ci :   ci2   ,   X = [1.6, 2.4]  ;
```

Ce second CSP a une solution : `I = 4    X = 2.000 (+- 0.500)    sat = 1.000`.

7. **Les opérateurs trigonométriques** sont définis sur les domaines suivants :

sin	$\mathbb{R}$	$\longrightarrow$	$[-1, 1]$
cos	$\mathbb{R}$	$\longrightarrow$	$[-1, 1]$
arcsin	$[-1, 1]$	$\longrightarrow$	$[-\pi, \pi]$
arccos	$[-1, 1]$	$\longrightarrow$	$[-\pi, \pi]$

Ainsi, l'évaluation de l'expression `arcos(0)` renvoie les deux nombres $-\pi/2$ et $\pi/2$ (contrairement à l'appel C++ `acos(0)`, qui renvoie $\pi/2$).

Soit un CSP avec, entre autres :

une variable X définie sur $[-10, 10]$,
une variable Y définie sur $[-1, 1]$,
une contrainte $Y = sin(X)$.

Sans précaution particulière, les valeurs de X dans les solutions de ce CSP ne pourront appartenir qu'à l'intervalle $[-\pi, \pi]$, parce que le filtrage du domaine de X par la contrainte $Y = sin(X)$ met en jeu l'opérateur *arcsin*.

Pour trouver d'éventuelles solutions avec des valeurs de X en dehors de $[-\pi, \pi]$, on peut écrire le CSP avec, entre autres, les éléménts suivants :

une variable X définie sur $[-10, 10]$,
une variable Y définie sur $[-1, 1]$,
une variable T définie sur $[-\pi, \pi]$,
une variable entière k avec le domaine $\{0, 1, \ldots, 10\}$,
une contrainte $Y = sin(T)$,
une contrainte $X = T + 2*k*\pi$.

5.2.4 Syntaxe d'une contrainte en intension multiple

`\constraint_intension_multiple` [: / =] <*nom*> [(<*priorité*>)] **,** <*équation*> ;

avec :

<*nom*> : le nom de la contrainte ;
<*priorité*> : degré de priorité (importance) de la contrainte (valant 1 par défaut) ;
<*équation*> : la relation fonctionnelle de la contrainte, de la forme suivante :

<*binop*> (... ,<$expression_i$ >**,** ...)

avec

<*binop*> : un des opérateurs binaires suivants : =, $\leq$, <, $\geq$, >, ! = (ou <> : "*différent de*") ;
Les différentes expressions s'écrivent de la même manière que dans le tableau ci-dessus.

Les deux abréviations du mot-clé `\constraint_intension_multiple` sont `\const_int_mult` et `\cim`.

Exemples

- Exemple d'une contrainte en intension multiple exprimant le fait que les trois expressions T, $U+V$ et $5*W$ doivent prendre des valeurs différentes :

  ```
  \cim :  CI2  ,  <>(T, U+V, 5*W ) ;
  ```

  Cette contrainte est décomposée en trois contraintes en intension simples. Il est équivalent d'écrire :

  ```
  \ci :  CI21  ,  T <> U+V ;
  \ci :  CI22  ,  T <> 5*W ;
  \ci :  CI23  ,  U+V <> 5*W ;
  ```

- Exemple d'une contrainte en intension multiple exprimant le fait que les trois expressions T, $U+V$ et $5*W$ doivent prendre des valeurs égales :

  ```
  \cim :  CI2  ,  =(T, U+V, 5*W ) ;
  ```

  Cette contrainte est décomposée en deux contraintes en intension simples. Il est équivalent d'écrire :

  ```
  \ci :  CI21  ,  T = U+V ;
  \ci :  CI22  ,  U+V = 5*W ;
  ```

5.3 Déclaration d'une contrainte conditionnelle

Une contrainte conditionnelle possède deux ensembles de contraintes (en extension ou en intension exclusivement [6]) : la *prémisse* et la *conclusion*.

Lors d'une action sur le CSP (filtrage ou résolution), les contraintes contenues dans la conclusion (et seulement celles-là) entrent en jeu dès que toutes les contraintes de la prémisse sont satisfaites [7].

5.3.1 Syntaxe

```
\constraint_conditionnal  [: / =]  <nom>
      \if  /  \si
            ⋮
         <CP_i>
            ⋮
      \then  /  \alors
            ⋮
         <CC_j>
            ⋮
      ;
```

avec :

<*nom*> : le nom de la contrainte ;
<CP_i> : la i^{ème} contrainte de la prémisse ;
(contrainte en extension ou en intension exclusivement)
<CC_j> : la j^{ème} contrainte de la conclusion
(contrainte en extension ou en intension exclusivement).

Les deux abréviations du mot-clé `\constraint_conditionnal` sont `\const_cond` et `\cc`.

Remarques :

- Dans la prémisse, il est fortement conseillé de ne déclarer que des contraintes unaires et "simples" (comme l'appartenance à un domaine), car plus une contrainte est complexe, plus la notion de *nécessaire satisfaction* est difficile à apprécier (en particulier à cause des limites du calcul d'intervalles pour les contraintes en intension).

  D'autre part, il est interdit d'exprimer l'appartenance d'une variable à un intervalle à l'aide d'une contrainte en intension (comme `\ci : cp1 , X = [0, 1] ;` pour exprimer la contrainte $X \in [0, 1]$), car le calcul de la nécessaire satisfaction des contraintes de la prémisse interprète différemment une contrainte de ce type selon que c'est une prémisse ou non.

- Un opérateur unaire particulier est mis à la disposition de l'utilisateur : *known(<nom de variable>)*. Il retourne une valeur booléenne : *vrai* si la variable indiquée en argument est complèmenent instanciée, *faux* sinon.

---

6. ...c'est-à-dire à l'exclusion d'une autre contrainte conditionnelle, d'une contrainte par morceaux, d'une conjonction ou une disjonction de contraintes. La possibilité d'exprimer une contrainte par morceaux en conclusion sera introduite dans une prochaine version du logiciel.

7. Il faut même qu'elles le soient *nécessairement*. Une contrainte est *nécessairement satisfaite* lorsqu'elle est satisfaite (au moins au degré α) pour toutes les combinaisons de valeurs encore présentes dans les domaines de ses variables.

Il permet, par exemple, de n'introduire des contraintes dans le problème qu'après une première phase de recherche ayant abouti à l'instanciation (par une valeur quelconque) d'une variable déterminée.

5.3.2 Exemples

1) La contrainte conditionnelle

si
$X \in [0, 1]$
alors
$Y = X^2 - 1$

peut s'exprimer de 2 manières différentes (différence au niveau de l'expression de la prémisse) :

```
\cc  :  CC1
        \if
                \ce : cp1 X , ([0, 1]) ;
        ou
                \ci : cp1 , X >= 0 ;
                \ci : cp2 , X <= 1 ;
        \then
                \ci : cc1 , Y = X^2-1 ;
        ;
```

Dès que, au cours de la résolution, le domaine des valeurs de X de degré supérieur à α sera réduit à un sous-ensemble de [0, 1], alors la contrainte `cc1` sera introduite dans le CSP.

2) La contrainte conditionnelle

si
X est complètement instanciée
alors
contraindre Y à être le double de X

s'exprime par :

```
\cc  :  CC1
        \if
                \ci : cp1 , known(X) ;
        \then
                \ci : cc1 , Y = X*2 ;
        ;
```

Dès que, au cours de la résolution, le domaine des valeurs de X de degré supérieur à α sera réduit à une valeur réelle unique, alors la contrainte `cc1` sera introduite dans le CSP.

5.4 Déclaration d'une contrainte par morceaux

Une contrainte par morceaux est définie en plusieurs blocs ou *morceaux*. Chaque morceau possède deux ensembles de contraintes : sa *prémisse* et sa *conclusion*.

Les prémisses des différents morceaux décrivent autant de blocs de définition disjoints entre eux (en deux dimensions, on réalise un "pavage" de l'espace).

5.4.1 Syntaxe

```
\constraint_piecewise  [: / =]  <nom>
      ⋮
      \for  /  \pour
            ⋮
         <CP_{i,j}>
            ⋮
      \do  /  \faire
            ⋮
         <CC_{i,j}>
            ⋮
      ⋮
      ;
```

avec :

<*nom*> : le nom de la contrainte ;
$<CP_{i,j}>$: la j^{ème} contrainte de la prémisse du i^{ème} morceau ;
(contrainte en extension ou en intension exclusivement)
$<CC_{i,j}>$: la j^{ème} contrainte de la conclusion du i^{ème} morceau
(contrainte en extension ou en intension exclusivement).

Les deux abréviations du mot-clé `\constraint_piecewise` sont `\const_piece` et `\cp`.

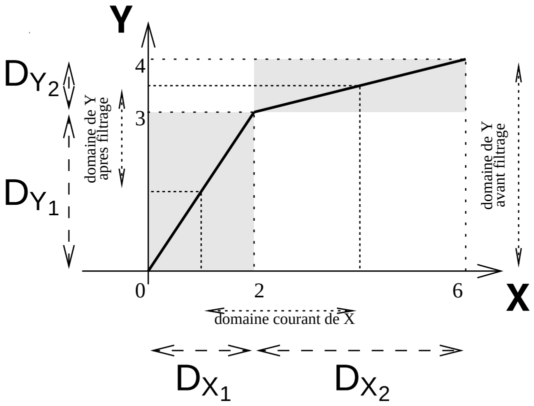
Une contrainte par morceaux permet d'introduire une discontinuité dans la représentation d'une relation. Une contrainte par morceaux n'est cependant pas scindable en autant de contraintes que de morceaux (conjonctivement). En effet, il suffit qu'un morceau voie sa relation vérifiée pour que la contrainte par morceaux soit satisfaite, alors que toutes les contraintes doivent être satisfaites dans une conjonction. La possibilité de scinder une contrainte par morceaux en une disjonction de contraintes est commentée dans la section 5.6.2.

Dans la présente version du logiciel, le degré de priorité d'une contrainte par morceaux est toujours de 1.

5.4.2 Exemple

La contrainte par morceaux

pour
$X \in D_{X_1}$
$Y \in D_{Y_1}$
faire
$Y = \frac{3}{2}X$
pour
$X \in D_{X_2}$
$Y \in D_{Y_2}$
faire
$Y = \frac{X}{4} + 2.5$



peut s'exprimer de la manière suivante :

```
\cp  :  CP1
        \for
              \ce : cp11 X , ([0, 2]) (1) ;
              \ce : cp12 Y , ([0, 3]) (1) ;
        \do
              \ci : cc11 , Y = 3*X/2 ;
        \for
              \ce : cp21 X , ([2, 6]) (1) ;
              \ce : cp22 Y , ([3, 4]) (1) ;
        \do
              \ci : cc21 , Y = X/4+2.5 ;
        ;
```

Remarque : il est conseillé de ne déclarer dans les prémisses que des contraintes unaires et "simples" (permettant de représenter l'appartenance à un domaine). Cependant, il est possible d'exprimer des prémisses avec des contraintes en intension d'arité supérieure à 1.

Exemple :

pour $X < \mathit{SeuilX}$ **et** $Y < \mathit{SeuilY}$ **faire** $Y = \frac{3}{2}X$

Pour ce morceau, le filtrage mutuel de X et de Y sera effectué en réduisant leurs domaines (de façon temporaire, pour pouvoir être exploités à nouveau dans leur forme initiale pour d'autres morceaux) aux valeurs inférieures à la valeur de SeuilX et SeuilY, respectivement. Pour que cette réduction soit efficace, il faut que les deux variables SeuilX et SeuilY soient à ce moment-là complètement instanciées (par exemple : $\mathit{SeuilX} = 2$ et $\mathit{SeuilY} = 3$). Dans le cas contraire, le filtrage mutuel de X et de Y ne parviendra pas à réduire leurs domaines.

5.5 Déclaration d'une conjonction de contraintes

Poser une conjonction de contraintes directement dans le CSP semble *a priori* ne pas avoir d'intérêt puisqu'un CSP est déjà une conjonction de contraintes. Ce type de contrainte est en fait utilisé dans des disjonctions, pour exprimer des contraintes un peu complexes du type : (C_1 **et** C_2) **ou** (C_3 **et** C_4)[8].

Les contraintes d'une conjonction sont des contraintes en extension, en intension ou conditionnelles exclusivement)[9][10].

5.5.1 Syntaxe

`\conjonction_of_constraints` `[: / =]` <*nom*>

<C_i>
`\and`
⋮
;

avec :

<*nom*> : le nom de la contrainte ;
<C_i> : la i^{ème} contrainte (en extension, en intension ou conditionnelle) de la conjonction .

Les deux abréviations du mot-clé `\conjonction_of_constraints` sont `\conj_of_const` et `\coc`.

5.5.2 Exemple

$$Y = \ln(X+1)$$

et

$$(S_1, S_2) \in \{(a,a), (a,b), (b,a)\}$$

et

$$\textbf{si} \quad X \in [-2,-1] \cup [1,2] \quad \textbf{alors} \quad Y = \mathrm{abs}(X)$$

est une conjonction de contraintes qui s'exprime de la manière suivante :

```
\coc  :  DOC1
   \ci  :  C1   ,  Y = ln(X+1);
   \and
   \ce  :  C2   S1 S2 ,       a  a       a  b       b  a;
   \and
   \cc  :  C3
        \if                \ce  :  C31  X   , ([-2,-1] [1,2]);
        \then              \ci  :  C32   ,  Y = abs(X);
        ;
   ;
```

8. Le résultat du filtrage d'une variable dans une conjonction de contraintes sera l'*intersection* des résultats des filtrages effectués dans chacune des contraintes de la conjonction. Le degré de satisfaction d'une conjonction de contraintes sera le *minimum* des degrés de satisfaction de chacune des contraintes de la conjonction, comme pour le CSP dans son ensemble.

9. Les contraintes par morceaux seront autorisées dans une prochaine version du logiciel.

10. L'expression : C_1 **et** (C_3 **ou** C_4) se simplifie en (C_3 **ou** C_4), puisque le CSP est déjà une conjonction de contraintes.

5.6 Déclaration d'une disjonction de contraintes

Un CSP est, par définition, une conjonction de contraintes (C_1 **et** C_2 **et** ...). Cependant, il est parfois nécessaire d'exprimer une disjonction de contrainte (C_1 **ou** C_2 **ou** ...). Ce type de contrainte est donc un ensemble d'autres contraintes (en extension, en intension, conditionnelles, ou conjonction de contraintes) [11] reliées par des "*OU*" [12].

5.6.1 Syntaxe

```
\disjonction_of_constraints  [: / =]  <nom>
        <Ci>
        \or
          ⋮
        ;
```

avec :

<*nom*> : le nom de la contrainte ;
<C_i> : la i^{ème} contrainte de la disjonction
(en extension, en intension, conditionnelle ou conjonction de contraintes exclusivement).

Les deux abréviations du mot-clé `\disjonction_of_constraints` sont `\disj_of_const` et `\doc`.

5.6.2 Exemple

$$Y = \frac{1}{sin(X)}$$

ou

$$(S_1, S_2) \in \{(a,a), (a,b), (b,a)\}$$

ou

$$Y = X^2 + 1$$

et

$$Y = \exp(X) - 2$$

est une disjonction de contraintes qui s'exprime de la manière suivante :

```
\doc  :  DOC1
      \ci  :  C1  ,  Y = 1/sin(X);
      \or
      \ce  :  C2  S1 S2 ,    a  a        a  b        b  a;
      \or
      \coc  :  C3
           \ci  :  C31  ,  Y = X^2+1;
           \and
           \ci  :  C32  ,  Y = exp(X)-2;
           ;
      ;
```

11. Les contraintes par morceaux seront autorisées dans une prochaine version du logiciel.

12. Le résultat du filtrage d'une variable dans une disjonction de contraintes sera la *réunion* des résultats des filtrages effectués dans chacune des contraintes de la disjonction. Le degré de satisfaction d'une disjonction de contraintes sera le *maximum* des degrés de satisfaction de chacune des contraintes de la disjonction.

Remarque : l'exemple de contrainte par morceaux donné en section 5.4.2 ne peut pas s'exprimer comme une disjonction de contraintes conditionnelles du type :

```
\doc  :  DOCfalse
     \cc  :  CC1
             \if
                    \ci : cp11 X , ([0, 2]) (1) ;
                    \ci : cp12 Y , ([0, 3]) (1) ;
             \then
                    \ci : cc11 , Y = 3*X/2 ;
            ;
     \or
     \cc  :  CC2
             \if
                    \ci : cp21 X , ([2, 6]) (1) ;
                    \ci : cp22 Y , ([3, 4]) (1) ;
             \then
                    \ci : cc21 , Y = X/4+2.5 ;
            ;

;
```

La raison est que dans le cas d'un domaine de X réduit, par exemple, à l'intervalle [1 4], la contrainte par morceaux filtre le domaine de Y en l'intervalle $[\frac{3}{2}\,\frac{5}{2}]$, alors que la disjonction ci-dessus ne filtre pas du tout le domaine de Y. En effet, aucune des premisses des deux contraintes conditionnelles `CC1` et `CC2` n'est nécessairement satisfaite[13], ce qui empêche leurs conclusions de rentrer dans le CSP.

13. Parce qu'il existe des éléments du domaine courant [1, 4], e.g. 3, qui n'appartiennent pas à l'ensemble [0, 2], et d'autres, e.g. 1, qui n'appartiennent pas à l'ensemble [2, 6].

Bibliographie

[Dav87] Davis (E.). – Constraint propagation with interval labels. *Artificial Intelligence*, vol. 32, n° 3, juillet 1987, pp. 281–331.

[FMCS92] Fargier (H.), Martin-Clouaire (R.) et Schiex (T.). – Satisfaction de contraintes souples. *In : Journées nationales : Les applications des ensembles flous*, pp. 183–191. – Nîmes, France, octobre 1992.

[Hyv92] Hyvönen (E.). – Constraint reasoning based on interval arithmetic: the tolerance propagation approach. *Artificial Intelligence*, vol. 58, décembre 1992, pp. 71–112.

[Kum92] Kumar (V.). – Algorithms for constraint-satisfaction problems : A survey. *AI Magazine*, vol. 13, n° 1, 1992, pp. 32–44.

[Lho94] Lhomme (O.). – *Contribution à la résolution de contraintes sur les réels par propagation d'intervalles.* – Sophia Antipolis, Thèse de doctorat, Université de Nice Sophia Antipolis, juin 1994.

[Moo66] Moore. – *Interval analysis.* – New Jersey, Prentice Hall, 1966.

[Nad89] Nadel (B. A.). – Constraint satisfaction algorithms. *Comput. Intell.*, vol. 5, n° 4, novembre 1989, pp. 188–224.

[Pro93] Prosser (Patrick). – Hybrid algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, vol. 9, n° 3, août 1993, pp. 268–299.

[Tsa93] Tsang (E. P. K.). – *Foundations of Constraint Satisfaction.* – London, Academic Press Ltd., 1993.

Index

Symboles

E

F

G

H

I

M

N

O