Département d'informatique et recherche opérationnelle

UNIVERSITÉ DE MONTRÉAL

Projet IFT3051

IMPLANTATION DU PTIDEJSOLVER EN JAVA

PAR

Salim Bensemmane

Iyadh Sidhom

Fayçal Skhiri

24 JUILLET 2004

TABLE DES MATIERES :

I. Introduction:

Les programmes passent environ 90% de leurs vies en maintenance. La maintenance des programmes est responsable d'au moins 75% du coût total d'un logiciel. Pourtant la maintenance des programmes reste encore une activité difficile car elle nécessite de comprendre le fonctionnement des programmes, d'évaluer l'impact des changements, et de réaliser ces changements.

Ptidej est une application qui utilise un méta-modèle pour représenter les patrons de conceptions pour leur l'instanciation et leur détection. Une fois qu'un patron de conception (comme *Composit*, *Facade*, *Mediato*, *Singleton*, etc.) a été décrit en utilisant ce méta-modèle, il est possible de l'instancier sur un code source utilisateur (pour adapter ce code source utilisateur aux spécifications du patron de conception) et de détecter des micro-architecture de ce patron (avec un plus ou moins grand degré de fiabilité) dans un code source donné.

Notre tâche consiste à réimplanter le solver en Java avec la plateforme Eclipse de IBM. L'application PtidejSolver est l'outil qui permet de déterminer le patron de conception utilisé par un programme écrit en Java en le comparant avec un meta-modèle donné.

Cette application definit cinq solveurs : simpleInterractiveSolver, InterractiveSolver, simpleAutomaticSolver, automaticSolver et combinatorialAutomaticSolver ainsi que plusieurs contraintes unaires et binaires.

II. Aperçus sur la programmation par contraintes :

1. Définition de base

- Système de contraintes :

On appelle système de contraintes la donnée d'un domaine de calcul D, d'un ensemble V de variables, de l'ensemble D de leur domaine associe et d'un ensemble C de contraintes sur V.

- Solution :

Une solution pour un système de contraintes est la sélection d'un sous ensemble du domaine (instanciation) de chaque variable de telle sorte que toutes les contraintes soient vérifiées.

- Propriété :

Une propriété P représente un degré de consistance devant être respecte par le système de contraintes : il s'agit d'une condition nécessaire pour la consistance du système de contraintes considère. Ainsi, si celui-ci n'est pas consistant, la propriété P n'est pas vérifiée.

- P-satisfaisabilite d'un système de contraintes :

Un système de contrainte est dit P-satisfaisable si et seulement si la propriété P est vérifie pour les contraintes du problème et les domaines courants des variables concernées.

- Ajout/retrait de contraintes :

Soit un problème dont la configuration courante est (A, R).

(A, R) = (ensemble des contraintes ajoutées, ensemble des contraintes retirées)

- l'ajout d'une contrainte *c* consiste à passer de la configuration (A, R) à la configuration (A U {*c*}, R \ { *c* }) si $c \in$ R et à la configuration (A U {*c*} , R) sinon.

- le retrait d'une contrainte *c* consiste à passer de la configuration (A, R) à la configuration (A \ { *c* }, R U { *c* }).

- Problème dynamique :

Nous définissons un problème dynamique comme étant une suite de problèmes différant l'un de l'autre par l'ajout ou le retrait d'une contrainte.

- Problème sur-contraints :

Un problème est dit sur-contraint pour une propriété P si le système de contraintes le définissant ne vérifie pas la propriété P.

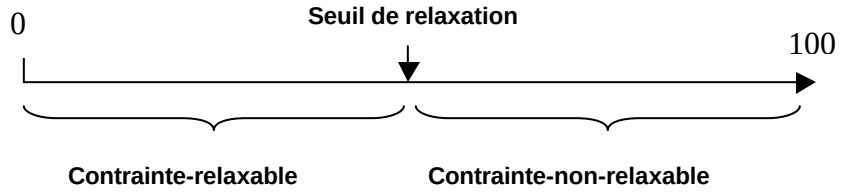
Lorsqu'il est confronté à un problème sur-contraint, un solveur classique répond à l'utilisateur qu'il n'existe pas de solution. Cette réponse est vraiment très pauvre et souvent préjudiciable. Par exemple, si dans le cadre d'un ordonnancement dynamique, une nouvelle tâche est à prendre en compte et qu'aucune solution n'existe, il n'est pas

acceptable que le systeme de resolution reponde simplement : il n'existe pas de solution. Il est plus intéressant d'être capable, par exemple, de dire quelle contrainte ne doit pas être prise en compte pour permettre l'existence d'une solution c'est le cas dans les solveurs offert par PtidejSolver et cela en utilisant la librairie *JChoco*.

Lorsque certaines contraintes d'un problème ne sont pas prises en compte dans une solution proposée à l'utilisateur, on parle de **relaxation** de contrainte. Il s'agit d'un retrait de la contrainte considérée au sens de la définition de retrait ci-dessus. D'un point de vue opérationnel, tous les effets passés de la contrainte relaxée doivent être supprimés.

Dans la plupart des cas, on ne peut pas simplement relaxer la dernière contrainte introduite (celle qui a provoqué la contradiction). Dans une application réelle, une telle contrainte ajoutée au dernier moment est souvent un impératif que l'on vient de découvrir et qui ne peut être écarté. C'est la raison pour la quelle on demande à l'utilisateur d'enlever une contrainte dans le cas des solveurs interactives ou on relaxe nous même une contrainte dans le cas des solveurs automatiques.

Dans notre application chaque contrainte est caractérisée par un poids qui varie entre 0 et 100, en plus chaque problème a son propre seuil de relaxation qui est définit par l'usager donc on peut relaxer que les contraintes de poids inférieur à ce seuil.



« Échelle des poids »

2. Exemple de programmation par contraintes

Nous avons choisi de donner un exemple en langage *CLAIRE* puisque il est plus facile à lire :

Example 1. Handling over-constrained problems

```
[over-constrained-problem()
 -> let pb := makePalmProblem("demo over-constrained system", 3, 5),
                                    // maximum relaxation level: 5
        a  := makeBoundIntVar(pb, "a", 1, 3),
        b  := makeBoundIntVar(pb, "b", 1, 3),
        c  := makeBoundIntVar(pb, "c", 1, 3),
       ct1 := (a > b),
       ct2 := (b > c),
       ct3 := (c > a)
    in (
        try ( // using claire exception handling mechanisms
            printf("Variables: ~S\n", list(a,b,c)),

            printf("posting ~S \n", ct1),
            post(pb, ct1, 2), propagate(pb), // ct1 is not that
important (weight 2)
            printf("Variables: ~S\n", list(a,b,c)),

            printf("posting ~S \n", ct2),
            post(pb, ct2, 5), propagate(pb), // ct2 is really
important (weight 5)
            printf("Variables: ~S\n", list(a,b,c)),

            printf("posting ~S \n", ct3),
            post(pb, ct3, 3), propagate(pb), // ct1 is quite
important (weight 3)
            printf("Variables: ~S\n", list(a,b,c))
        )
        catch PalmContradiction ( // if a contradiction occurs
            repair(pb),
            printf("Variables: ~S\n", list(a,b,c))
        )
    )]
```

Example 2. Handling over-constrained problems (result)

```
palm> over-constrained-problem()
eval[0]>
  Variables: (a:[1..3], b:[1..3], c:[1..3])
  posting a >= b + 1
  Variables: (a:[2..3], b:[1..2], c:[1..3])
  posting b >= c + 1
  Variables: (a:3, b:2, c:1)
  posting c >= a + 1
  // here a contradiction occurred : repair is called
  PALM: Removing constraint a >= b + 1 (w:2)
  // the least important relevant constraints has been chosen for
relaxation
  Variables: (a:1, b:3, c:2)
  nil
palm>
```

III. ENVIRONNEMENT DE TRAVAIL

Pour réaliser ce projet nous utilisons l'environnement de développement pour Java fourni avec la plate-forme Eclipse, qui est un outil gratuit (open source).
Ce logiciel est compatible avec la majorité des plateformes Linux, QNX, MAC OSx et Windows.
Nous utilisons aussi le système de travail collaboratif CVS qui est incorporé à Eclipse. Cet outil nous permet de partager nos travaux entre tous les membres de l'équipe.
Nous utilisons également pour ce projet les bibliothèques Choco et Palm développé en langage Claire ainsi que la bibliothèque JChoco qui est une re-implantation des bibliothèques Choco et Palm en Java.

IV. CONTRAINTES TECHNIQUES

- Comme notre projet se base sur la programmation par contraintes, il a donc fallu collecter la documentation sur ce domaine pour comprendre le mécanisme de ce type de programmation, puis étudier quelques exemples de programmation par contraintes comme les fameux problèmes des *n-reines* et *MajicSquare* fournis avec la bibliothèque JChoco.

- Aussi, nous avons démarré avec une version de PtidejSolver implanté en langage Claire, il a fallu nous familiariser avec ce langage qui était nouveau pour tous les membres de l'équipe en cherchant la documentation nécessaire. (difficile à trouver au départ car ce langage est peu utilisé.)

- Enfin la version originale de PtidejSolver utilise les bibliothèque Choco et Palm implantées en Claire, il nous a fallu faire la correspondance entre la bibliotheque JChoco en Java et celles en Claire. Aussi on a eu des problèmes à comprendre la librairie JChoco avec le peu de documentation existante et malgré sa complexité au niveau de son implantation (fort couplage).

V. Architecture de Ptidej (voir Diagramme d'architecture de JPtidejSolver)

❖ La classe *JPtidejBranching* :

Est une super classe qui permet de gérer la gestion des affectations de variable à l'aide de ces sous classes :

- *JPtidejAutomaticBranching*
- *JPtidejInteractiveBranching*
- *JPtidejSaveAllSolutions*

JPtidejAutomaticBranching est utilisé par *JPtidejAutomaticSolver* et *JPtidejCombinatorialAutomaticSolver*.

JPtidejInteractiveBranching est utilisé par *JPtidejInteractiveSolver* *JPtidejSimpleInteractiveSolver*.

JPtidejSaveAllSolutions est utilisé par *JPtidejSimpleAutomaticSolver*

❖ La classe *JPtidejSolveur* :

Contient l'algorithme générique de la méthode *repair*,

Voici l'algorithme :

- Récupérer la liste des contraintes à effacer
  - **Si** elle n'est pas vide faire :
    Pour chaque contrainte de cette liste **faire** :
      -**Si** poids_contrainte < poids_maximal **faire** :
        -**Si** poids_contrainte = 0 alors on maintient l'état actuelle de la recherche
        -**Sinon** on sauvegarde les contraintes relaxées
      -Effacer la contrainte du problème

-**Si** aucune contrainte n'a été effacée, lever une contradiction système et terminer le programme

-**Sinon** on propage de nouveau le problème et on appelle à *repair* en cas d'une contradiction

-On fait la négation des contraintes de décision afin de trouver d'autres solutions

-poster les contraintes à rajouter, puis on propage de nouveau le problème et on appelle à *repair* en cas de contradiction

Toutes les classes qui héritent de *JPtidejSolver* permettent de faire l'attachement du *branching* et du *repair* adéquat au solveur.

- La classe *JPtidejRepair*

  Il y'a deux classes principales qui héritent de *JPtidejRepair* :

  - *JPtidejMemoryRepair*

    Toutes les sous classes de *JPtidejMemoryRepair* permetent de mémoriser les contraintes retirées.

  - *JPtidejNoMemoryRepair*

    Les sous classes de *JPtidejNoMemoryRepair* n'ont pas besoin de mémorisation .

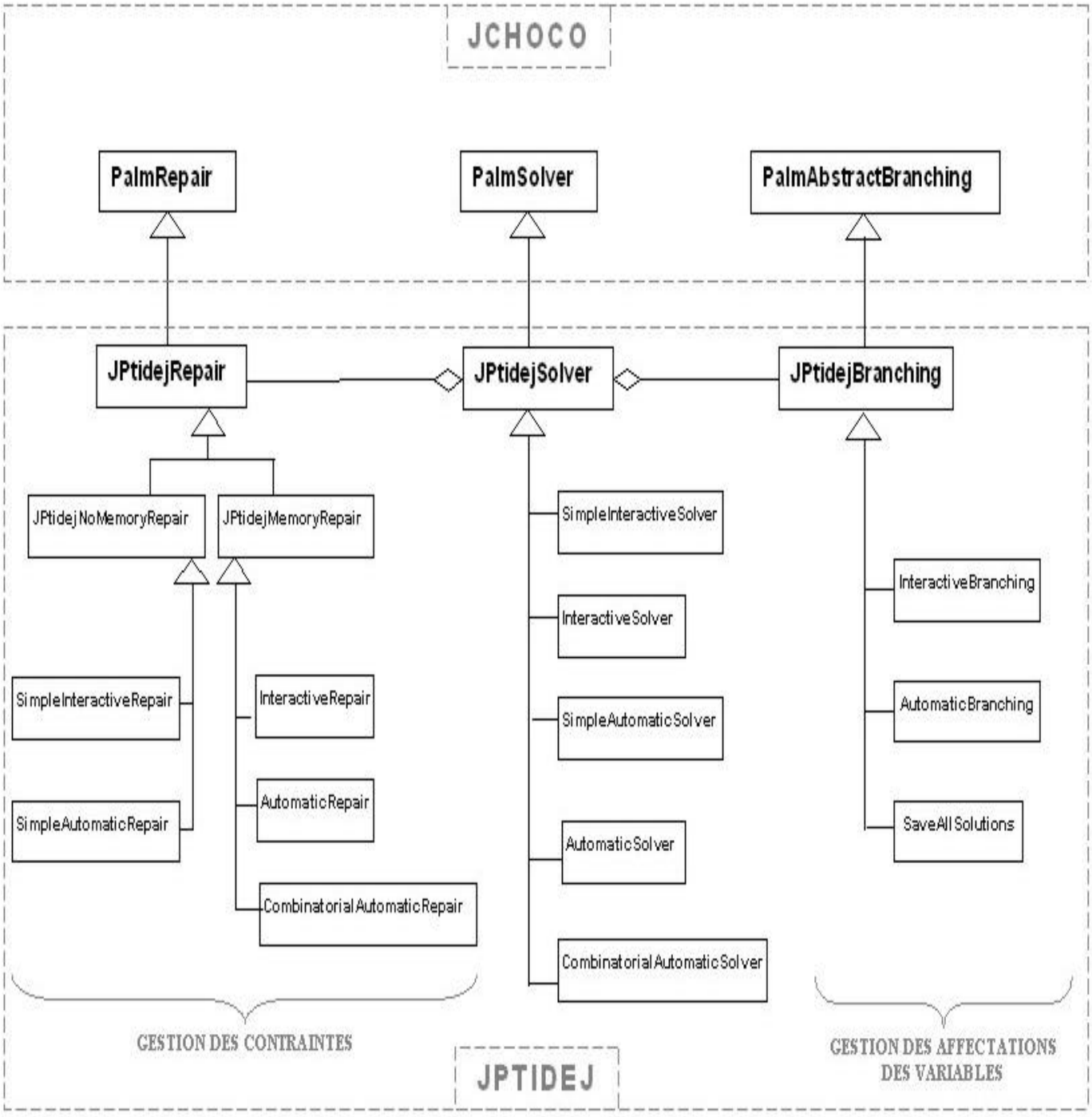


Diagramme d'architecture de JptidejSolver

VI. Les Contraintes

Il existe deux types de contraintes, les contraintes de décision (système) intégrant le solveur par défaut de la bibliothèque Jchoco, et les contraintes *JPtidejConstraint* définies dans le cadres de notre projet .

- **contrainte de décision (système) :**

Il s'agit de contrainte unaire qui sont générées dans le processus de résolution.

Il en existe deux types : contraintes de décision et contrainte de négation.

Soit la variable « V » avec le domaine « D ».

La contrainte de décision consiste a affecter une valeur « q » du domaine « D » a la variable « V » (V == q) .

Tandis que la contrainte de négation consiste a restreindre une valeur « q » du domaine « D »

(V ? q) .

- **contrainte *JPtidejConstraint* :**

Elles se décomposent en deux catégories JPtidejBinConstraint et JPtidejLargeConstraint. (comme illustrée dans le diagramme de l'architecture ci dessus) .

Le type JPtidejBinConstraint représente les contraintes binaires, tandis que le type JPtidejLargeConstraint représente un type où une $3^{ème}$ variable de type *PalmIntVar* est utilisée.

JPtidejBinConstraint :

Soient deux variables « V_1 » et « V_2 » respectivement de domaines D_1 et D_2.

Sachant qu'un domaine est un ensemble d'entités.

En terme d'implantation, une entité a les champs : *inheritanceEntities*, *composedEntities*, *associationEntities* …

Chaque contrainte fait appel au champ approprié, utilisé dans le processus de propagation à travers la méthode *propagate.*

On préfère expliquer les deux mécanismes (*propagate*, *awakeOnRemove*) de nettoyage de domaine utilisés dans le cadre de notre projet, avant d'entamer l'explication

sémantique de chaque contrainte.

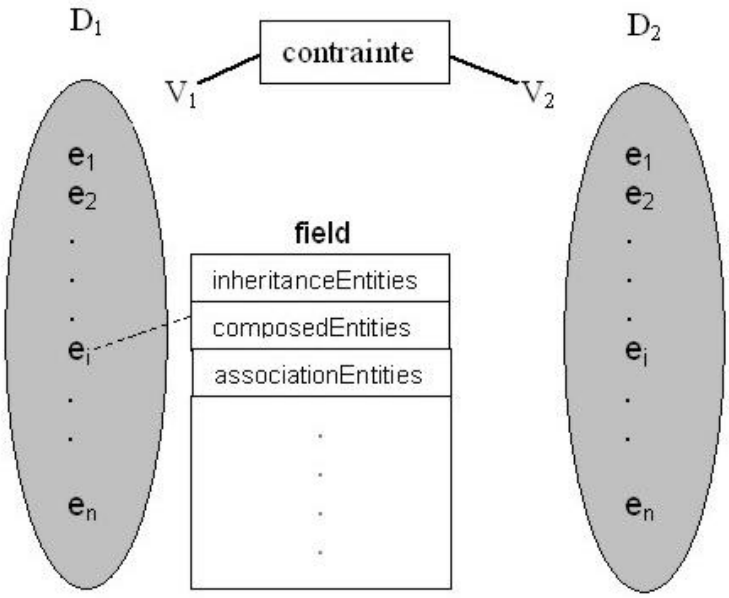


Diagramme de nettoyage de domaine

L'optimisation structurelle des contraintes s'est basée surtout dans la factorisation des méthodes de nettoyage.

- mécanisme de nettoyage de domaine :

a/- concept de *propagate* :

Cela consiste a faire le nettoyage séquentiel des domaines D_1 et D_2, selon l'algorithme ci-dessous:

- nettoyage de D_1 :

pour chaque entité e_i appartenant D_1 ,

si il existe $e_j \in D_2$ tel que la relation $R(e_j,e_i)$ existe

alors on garde e_i dans le domaine D_1

sinon on supprime e_i du Domaine D_1

puis

- nettoyage de D_2 :

pour chaque entité e_j appartenant D_2 ,

si il existe $e_i \in D_1$ tel que la relation $R(e_j,e_i)$ existe

alors on garde e_j dans le domaine D_2

sinon on supprime e_j du Domaine D_2

note : $R(e_j,e_i) = e_j \in e_i$.field ,

où « field » représente l'un des champs de l'entité, selon la contrainte .

b/- concept de *awakeOnRemove* :

Il s'agit d'une mise à jour du domaine D_1 ou D_2, suite à un retrait d'une entité de l'un des deux domaines.

L'algorithme est comme suit :

si une entité e_k est supprimé du domaine D_1

alors mise à jour de D_2 :

pour chaque entité e_j appartenant D_2 ,

si il existe $e_i \in D_1$ tel que la relation $R(e_j,e_i)$ existe

alors on garde e_j dans le domaine D_2

sinon on supprime e_j du Domaine D_2

si une entité e_k est supprimé du domaine D_2

alors mise à jour de D_1 :

tout d'abord on détermine le sous-ensemble SD1 définit comme suit :

$SD_1 = \{ e_i \in D_1 / e_k \in$ à e_i.field $\}$

puis on fait un nettoyage de SD_1 comme suit :

s'il existe $e_j \in$ à D_2 tel que $e_j \in$ à e_i.field $\setminus \{e_k\}$

alors on garde e_i dans SD_1 (donc dans D_1) .

sinon on supprime e_i de SD_1 (donc de D_1) .

note : à savoir que lorsqu'on fait appel à la méthode *awakeOnRemove* ,
une entité (soit e_k) a été auparavant supprimé du domaine D_1 ou D_2.

- sémantique des contraintes

a/ JPtidejBinConstraint :

- *StrictInheritance :*
Correspond a la relation R(ej,ei) , sachant que ej appartient à D_2 et ei appartient à D_1, où l'entité ei hérite de l'entité ej . (profondeur d'héritage = 1 et ei ? ej) .

- *Inheritance :*
Idem que la relation de *strictInheritance*, mais en plus on peut avoir ei = ej (profondeur d'heritage = 0 ou 1) .

- *StrictInheritancePath :*
Idem que la relation de *strictInheritance*, sauf que : profondeur d'héritage = 1.

- *InheritancePath :*
Idem que la relation de inheritance, sauf que : profondeur d'héritage = 1.

- *Agrégation :* relation d'agrégation similaire à UML.

- *Association :* relation d'association similaire à UML.

- *Composition :* relation de composition similaire à UML.

-C*ontainerAgrégation & containerComposition :*
Soient deux classes A et B, où la classe A contient un champ d'instance «b » de type B non instancié lors de sa déclaration.

Soit « f » une méthode de la classe A, qui a un paramètre « b' » de type B, et dont le corps contient l'affectation b = b'.

Si l'instance de b' est partagé alors on dit que la relation entre l'entité A et l'entité B est une relation de containerAgrégation.

Si l'instance de b' est non partagé alors on dira qu'il s'agit d'une relation de containerAgrégation .

- *Knowledge :*

Soient deux classes A et B.

Cela correspond à la relation où A prend connaissance de B, c'est-à-dire qu'il existe un paramètre de type B dans une méthode ou un constructeur de la classe A.

- *Creation :*

Soient deux classes A et B.

Cela correspond au fait qu'une variable de type B est instanciée dans une méthode ou Constructeur de la classe B. On dit que A crée B.

- *Ignorance :*

Il s'agit du complément de l'ensemble des relations.

Cela correspond au fait qu'aucune relation n'existe entre A et B, et on dit que A ignore B.

b/ JPtidejLargeConstraint :

comparée au type *JPtidejBinConstraint*, il y a en plus une 3ème variable de type *PalmIntVar* qui fait intervenir « compteur » ,c'est-à-dire le facteur distance entre les domaines de deux variables, et cela en lui appliquant une contrainte unaire de type *Equal*, *GreaterOrEqual* ou *Less*.

- AssociationDistance :

On dit la relation R(ej,ei) est une relation d'associationDistance,

s'il existe un n tuplet d'entités (e_1, ….., e_n) tel que les relations

R(ej, e_1) , R(e_1, e_2) , ... , R(e_{n-1}, e_n), R(e_n, e_i) sont des relations d'association .

De plus si on pose la contrainte compteur = 3, cela signifie que n = 3.

- InheritanceTreePath :

Par rapport a la contrainte associationDistance, les relations

R(ej, e_1), R(e_1, e_2), ... , R(e_{n-1}, e_n), R(e_n, e_i) sont des relations d'héritage .

- KnowledgeDistance :

Par rapport a la contrainte associationDistance, les relations

R(ej, e_1) , R(e_1, e_2) , ... , R(e_{n-1}, e_n), R(e_n, e_i) sont des relations knowledge.

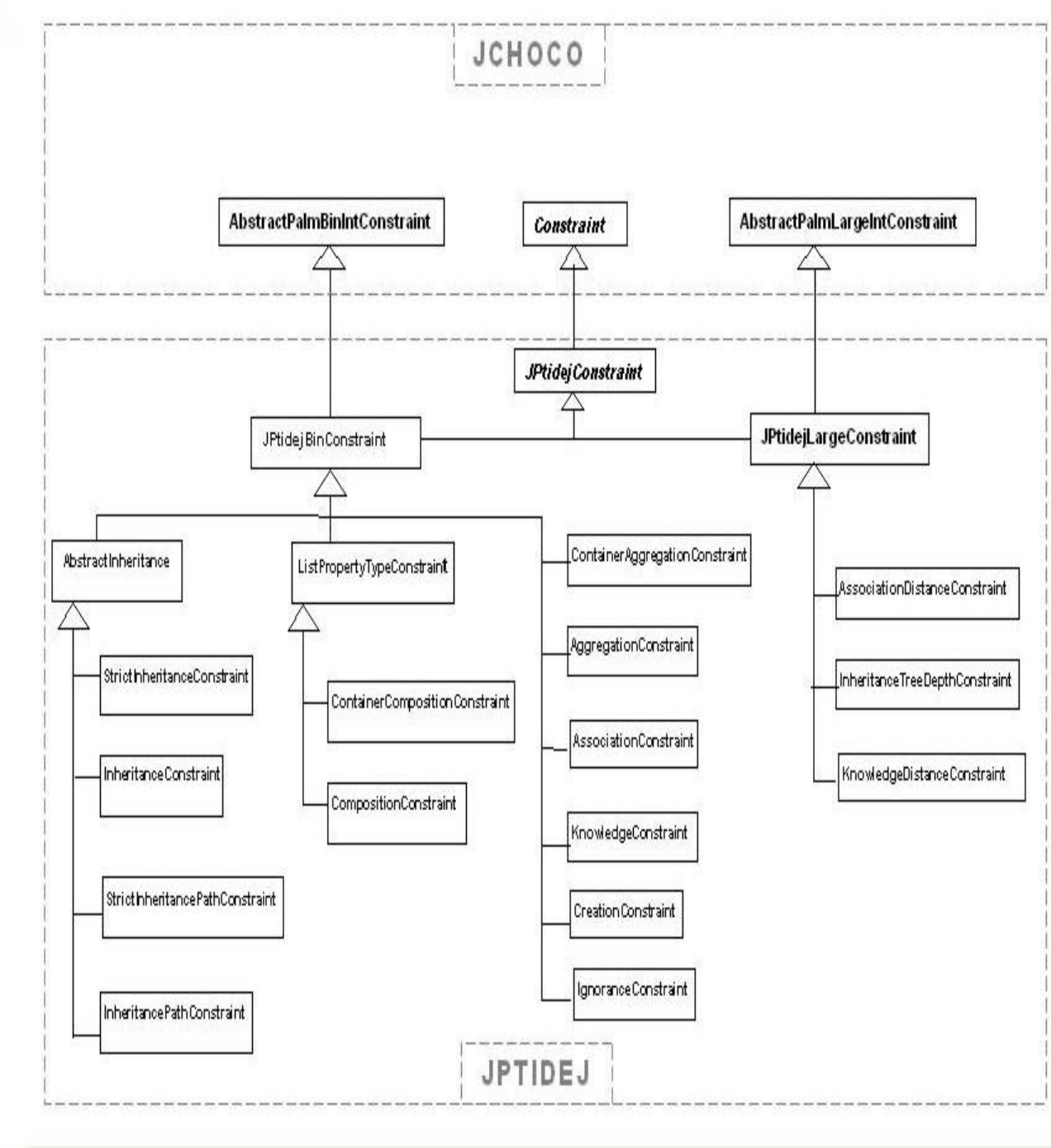


Diagramme UML de l'architecture des contraintes

VII. les solveurs

1.SimpleInteractiveSolver

Il s'agit d'un solveur qui fait appel au solveur de la librairie *JChoco*, et où l'usager interagit à travers la console en choisissant la contrainte à relaxer dans une liste de contraintes responsables d'une contradiction, avant que le processus de résolution ne soit lancé à nouveau.
Le processus de résolution se déroule selon l'algorithme ci-dessous :

1- On lance la résolution du problème avec le solveur de la librairie *JChoco* .
2- S'il y a une contradiction système aller en **(8) .**
3- Si aucune contradiction simple aller en **(7).**
4- On identifie et affiche dans la console l'ensemble E des contraintes causant une contradiction simple , et cela par ordre de priorité (ordre croissant de poids)
5- L'utilisateur sélectionne une contrainte
6- On propage le problème , puis aller en **(2).**
7- Le branching *InerractiveBranching* enregistre la solution (affectation de l'ensemble des variables) , et on affiche la solution , puis aller en **(1).**
8- FIN (« no more solution »).

2. InteractiveSolver

Le *InteractiveSolveur* utilise comme *branching* le *InteractiveBranching* son gestionnaire de contrainte (*repair*) est le *InteractiveRepair*. Cette classe définit la méthode *ptidejSelectDecisionToUndo* qui est responsable de déterminer la contrainte à retirer en cas de problème sur-contraint.
Ce solveur interagit avec l'usager de la manière que *SimpleInteractiveSolver*, la seul différence est lorsqu'on relaxe une contrainte on la remplace par la suivante (next-constraint), et dans l'étape suivante on propose a l'usager de remettre une parmi celle qui ont été relaxé auparavant.

1-On lance la résolution du problème.

2-S'il y a une contradiction système aller en **(10).**

3-Si aucune contradiction simple aller en **(9).**

4-On identifie et affiche dans la console l'ensemble E des contraintes causant une contradiction simple, et cela par ordre de priorité (ordre croissant de poids)
Notons qu'après chaque contrainte, on affiche celle qui va la remplacer en cas de retrait.

5- L'usager sélectionne une contrainte

6- On identifie et on affiche dans la console les contraintes qui ont été relaxées à l'étape précédente.

7- L'usager sélectionne une contrainte à remettre

8- On propage le problème, puis aller en **(2).**

9- le branching de InteractiveBranching enregistre la solution
(Affectation de l'ensemble des variables), et on affiche la solution, puis aller a **(1).**

10 – FIN (« no more solution »).

3. SimpleAutomaticSolver:

Ce solveur se distingue du *SimpleInteractiveSolver* de faite qu'il retire les contraintes automatiquement, il n'a pas besoin d'interraction avec l'utilisateur.
Il utilise comme *branching* le *saveAllSolutions* et son gestionnaire de contrainte (*repair*) est le *SimpleAutomaticRepair*. Cette classe définit la méthode *ptidejSelectDecisionToUndo* chargée d'identifier la contrainte à retirer en cas de problème sur-contraint et l'algorithme de cette méthode est ci-dessous :

Fonction *ptidejSelectDecisionToUndo()*

Début

Si la contrainte de plus faible poids est une contrainte de décision (poids = 0)
Retirer la contrainte du problème
Sinon si la poids du contrainte < MAX_RELAX_LEVEL
Déterminer la liste de toutes les contraintes

Trier ces contraintes en ordre croissant en fonction de leurs poids

Retirer la contrainte de plus faible poids

Retourner cette contrainte.

Fin.

4. AutomaticSolver

Ce solveur a le même principe que le *simpleAutomaticSolver* mais il se caractérise par sa mémorisation des contraintes à retirer, son affectation de variable se fait à travers le *AutomaticBranching* et son repair est le *AutomaticRepair* qui opère selon cet algorithme :

Fonction ptidejSelectDecisionToUndo()

Début

Déterminer la liste de toutes les contraintes

Trier ces contraintes en ordre croissant en fonction de leurs poids

Si la contrainte de plus faible poids est une contrainte de décision (poids = 0)

Retirer la contrainte du problème

Sinon si (! Toutes les contraintes relaxées)

Si (le poids de la contrainte < MAX_RELAX_LEVEL)

Retirer la contrainte de plus faible poids

Si (Toutes les contraintes relaxées)

Si (le nombre de contrainte retirée est > 0)

Trier les contraintes en ordre croissant en fonction de leurs poids

Ajouter la contrainte de plus faible poids

Supprimer cette contrainte de la liste de contrainte à ajouter (userRelaxedConstraint)

Si (le nombre de contrainte à relaxer = 0 et le nombre de contrainte à ajouter = 0)

Lancer une contradiction système pour terminer le programme

Retourner la contrainte à retirer.

Fin.

5. CombinatorialSolver

Il consiste dans la définition de tous les problèmes possibles à partir du problème de base. Il utilise comme gestionnaire des contraintes le *CombinatorialAutomaticRepair* et comme gestionnaire des affectation des variables le *AutomaticBranching .*

Soit un problème P caractérisé par l'ensemble de *n* removables-constraint soit RC.

On va générer différentes catégories de problèmes, comme suit :

***catégorie 0* :** où aucune removable-constraint est relaxée .

catégorie 1 : où 1 removable-constraint est relaxée .

catégorie 2 : où 2 removable-constraint sont relaxées .

.

.

catégorie p : où p removable-constraint sont relaxées .

.

.

.

catégorie n : où toutes les removable-constraint sont relaxées .

Et pour chaque catégorie, soit la « *catégorie p »* on détermine l'ensemble des problèmes possibles de cardinalite C^p_n :

***catégorie 0* :** a un ensemble de $C^0_n = 1$ problème .

***catégorie 1* :** a un ensemble de C^1_n problèmes .

.

.

catégorie n : a un ensemble de $C^n_n = 1$ problème .

Ainsi, respectivement pour chaque catégorie de problème, soit la « *catégorie p »*, et de façon indépendante on lance la résolution pour chaque problème P_i appartenant a la « *catégorie p »* (sachant que p = 0 … n).

Le processus de résolution se déroule selon l'algorithme sommaire ci-dessous :

1- On lance la résolution de P_i avec le solveur de la bibliothèque JChoco .

2- S'il y a une contradiction système aller en **(8).**

3- Si aucune contradiction simple aller en**(7).**

4- On identifie l'ensemble E des contraintes causant une contradiction simple.

5- On choisit la contrainte Cmin (de moindre poids) appartenant à E,

on la relaxe du problème P_i et on la remplace par son next-constraint (*weaker-constraint*)

si le next-constraint existe, sinon on élimine la contrainte de poids minimal du problème P_i .

6- On propage le problème, puis aller en **(2).**

7- L'automatic branching enregistre la solution (affectation de l'ensemble des variables)

dans un fichier résultat , et post des contraintes de négation au problème P_i (par rapport a la solution courante), puis aller en **(1) .**

8 – FIN (« no more solution »).

VIII. Discussion:

A- Critiques & observations

- Critiques par rapport à l'implantation de PtidejSolver (en Claire) :

1/- erreur de logique par rapport a la définition, dans la contrainte "*StrictInheritanceConstraint*, dans la methode « *awakeOnremove* ».

-avant :

Lors d'un retrait d'une valeur «X » du domaine de V2, on identifiait les entités Ei appartenant a D1 tel que x appartient à Ei.supEntities, et systématiquement l'entité Ei était supprimée de D1 **(erreur) .**

-à présent :

Pour chaque Ei, on teste est ce qu'il existe une entité X' appartenant à Ei.supEntities -{ X } et appartenant à D2 . Si oui on garde Ei, sinon on supprime Ei de D1.

2/ dans la méthode repair (générique) de JPtidejSolver :

```
. . .
if (weight(ct) > 0)
(
   nil
 )
else (

      removeDecision(state, ct)
     ),
```

`remove(repairer, ct),` ← problème , car une contrainte de poids null, (contrainte de décision) n'est pas sensée être mémorisée dans la list userRelaxedConstraint .

. . . . . .

- solution proposée :

```
if (plug.getWeight() <= 0)
state.removeDecision(ct);
else
    ((JPtidejRepair) this.getRepair()).remove(ct);
```

3/ On a changé de le nom de la méthode *findNewVariable* par *findDynamicVariable*, on trouve que c'est plus explicite, puisque son rôle est de retourner une variable , qui a été postée dynamiquement dans un problème dynamique par le combinatorialSolver.

4/ Dans la méthode repair du *CombinatorialSolve*, il y avait une erreur de logique, par rapport au filtrage de la contrainte causant une contradiction.

```
// . . . . .
 if (get(nextConstraint, ct) != unknown  &  get(nextConstraint,  ct)  !=
nil
                                    & ct.nextConstraint.isa = method)
 (
    . . . . . // remplacer la contrainte ct par son next

 )
```

```
 else (// If the constraint is of weight 0, I just relax it.
         if (weight(ct) = 0)
             add(re[1], ct)

       ),
//....
```

La contrainte « ct » peut être une contrainte de décision, a qui on teste l'existence de son next-constraint ce qui n'a pas de sens .

De plus, rien n'est fait dans le cas où la contrainte « ct » a son next-constraint égal à null ce qui implique un problème car en principe « ct » doit être relaxée.

On est sensé filtrer uniquement les contraintes relaxable.

On peut se retrouver dans une situation ou toute les contraintes relaxables ont été relaxées du coup, la contrainte « ct » peut être une contrainte non relaxable de plus petit poids .

solution proposée :

```
if (weight != 0 && weight < this.getProblem().getMaxRelaxLevel())
{
    String        nextConstraint        =        ((JPtidejBinConstraint)
  ct).getNextConstraint();
    if (nextConstraint != null)
    {
        ..... // remplacer la contrainte ct par son next


    }
    else //if nextConstraint == null
     {
        ..... // relaxer la  contrainte ct

       re[0].add(ct);
     }
}
```

```
else // If the constraint is of weight 0, I just relax it.
   {
       // s'il s'agit d'une contrainte de décision , on la relaxe .
       if (weight == 0)
            re[0].add(ct);

   }
```

- Critiques par rapport à la bibliothèque JCHOCO :

1- Très fort couplage de la bibliothèque.

2- le champ « `hook` » a été défini de type « `Object` **»** dans la classe « `choco.AbstractEntity` », puis redéfini en tant que **«** `ConstraintPlugin` **»** (interface) dans la sous-classe « `choco.AbstractConstraint` **».**

3- On a constate que des champs d'instance ont été déclaré public, tels que dans les classes :

PalmProblem et *AbstractEntity***.**

4- dans la classe «`PalmSolver` », dans le constructeur, on remarque des attachements par défaut, (des instances de types « `PalmLearn` » et « `PalmRepair` » de la librairie JCHOCO), et cela nuit à l'extensibilité.

```
//. . .
this.attachPalmLearn(new PalmLearn());
this.attachPalmRepair(new PalmRepair());
. . .//
```

- ***solution proposée*** : passer en argument le type de Learn et le Repair dans le constructeur.

6- Dans la classe *choco.integer.var.AbstractIntDomain* la méthode *getIterator* retourne null . (ceci a été corriger par le professeur Yann)

- Observations

1- A noter, que bien que structurellement la classe *CombinatorialAutomaticRepair* hérite de la classe *JPtidejMemoryRepair***,** ce qui implique que les contraintes relaxées sont mémorisées dans une liste,cette mémorisation n'est pas du tout exploitée .

Car dans avant l'affichage des résultats, c'est à travers le branching et la méthode *setMessage* qui permet d'afficher le cheminement (remplacement par le next-contraint) des contraintes responsables des contradiction et dans le cas échéant la suppression.

On penserait donc a mettre le « CombinatorialAutomaticRepair » comme sous-classe de « JPtidejNoMemoryRepair » .

Néanmoins, on n'a pas fait ce changement, car on pense que cette mémorisation pourrait servir dans l'amélioration de l'affichage des résultats, par exemple pour faire la distinction entre les contraintes supprimées lors de la définition du problème, et les contraintes relaxées dans le processus de résolution.

B – Amélioration et ajout possible/futures :

- Le programme PtidejSolver est une application flexible par son architecture où nous pouvons ajouter des solveurs et des contraintes autant qu'on veut. Il sera plus intéressant d'ajouter des solveurs plus optimisés au niveau du calcul des solutions qui nous permetterait d'obtenir des résultats plus pertinents.

- On peut aussi modifier l'algorithme de certains solveurs pour avoir un algorithme plus efficace au niveau de la recherche des solutions tel que dans le cas de *AutomaticSolver* où il n'est pas très intéressant d'ajouter des contraintes après la relaxation de toutes les contraintes.

C- Évolution personnelle

Ce projet était intéressant pour plusieurs raisons : d'abord il nous a permit de se familiariser avec l'environnement de développement Eclipse très utilisé dans l'industrie.
Ensuite nous avons acquit une expérience sur l'outil de travail en équipe CVS qui a été très utile pour rendre le code disponible à tous les membres de l'équipe.
En plus, ce projet nous a permit d'approfondir notre connaissance sur certains aspects en programmation orientée objets en Java tels que : la réflexion, le polymorphisme et l'héritage…
Enfin, ce projet nous a fournit l'occasion d'apprendre un nouveau langage de programmation (CLAIRE) ainsi qu'un nouveau type de programmation qui est la programmation par contraintes qu'on l'a trouvée très intéressante pour la résolution de beaucoup de problèmes.

D- Remerciement :

Nous remercions:

- nous tenons a remercier notre directeur de projet : Mr Yann-Gäel Guéhéneuc.
Premièrement pour sa disponibilité, son aide et son support lors de notre
réalisation qui ont été précieux pour l'avancement de notre projet, mais aussi pour sa bonne humeur.
-nous remercions le support technique du DIRO pour l'espace supplémentaire qui nous a alloué qui était indispensable pour la réalisation de ce projet.

E- Sites respectifs

http://choco.sourceforge.net/

-documentation sur JChoco en Java.

http://www.e-constraints.net

-documentation sur Palm et Choco en Claire.

http://www.choco-constraints.net/download/CHOCO%201.008.pdf

-documentation sur Choco en Claire.

http://claire3.free.fr/

-documentation sur le langage CLAIRE.

IX. Références

- Thèse de doctorat de Jussien Narendra.

[Relaxation des contraintes pour les problèmes dynamique -1997-]

- http://www.yann-gael.gueheneuc.net/Work/Research/Introduction/