Un modèle d´exécution réactif et interactif pour la programmation logique avec contraintes

J. Fowler, T. Sola
LCR Thomson-CSF
Domaine de Corbeville
91404 Orsay cedex, France.
{julian,sola}@thomson-lcr.fr

F. Fages
LIENS, CNRS
Ecole Normale Supérieure
45, rue d'Ulm
75005 Paris, France
fages@dmi.ens.fr

Abstract

Nous définissons un modèle d'exécution réactif et interactif pour la programmation logique avec contraintes (PLC). Ce modèle d'exécution s'appuie d'une part sur un système de transformation des dérivations CSLD pour l'ajout et le retrait de contraintes et d'atomes dans la requête, et d'autre part sur un résolveur de contraintes dynamiques supportant l'ajout et le retrait incrémental de contraintes. La complétude du modèle d'exécution réactif de la PLC se démontre en prouvant séparément la correction d'un ensemble d'opérations sur les dérivations, et la complétude d'une procédure de recherche réactive qui permet la prise en compte asynchrone et préemptive des commandes de modification de la requête. L'implantation réalisée est évaluée sur deux applications réelles d'optimisation combinatoire.

1 Introduction

La programmation logique avec contraintes (PLC) repose sur une correspondance forte entre logique et calcul : un programme est identifié à une théorie logique sur un domaine, une exécution à la recherche d'une preuve. Le paradigme de base est celui du calcul à base de modèles relationnels : le système est modélisé par une théorie du premier ordre sur une structure algébrique qui fixe le "domaine du discours", i.e. le domaine d'interprétation. L'exécution d'un programme PLC peut être vue comme une simulation des spécifications, qui sert à vérifier leur cohérence, analyser leurs propriétés et calculer des solutions pour différentes tâches. Cette approche s'est avérée particulièrement fructueuse en modélisation de systèmes complexes et en résolution de problèmes combinatoires [8].

La pénétration de la PLC dans des applications industrielles a motivé très tôt la recherche de modèles d'exécution interactifs incrémentaux offrant une plus grande souplesse d'utilisation et une plus grande efficacité pour non seulement calculer une solution à un problème mais également maintenir une solution au cours de plusieurs interactions. Les deux domaines où le besoin d'interactivité avec l'environnement est nécessaire que nous avons plus particulièrement étudiées concernent des systèmes d'aide à la décision et d'ordonnancement en ligne ou ré-ordonnancement. Dans les systèmes d'aide à la décision l'interaction entre l'homme et la machine est une propriété fondamentale. La possibilité d'interaction permet un véritable discours entre l'homme et la machine, les résultats intermédiaires fournis par le système servent de points de référence dans l'élaboration d'un résultat final au travers des interactions choisies. Ceci est particulièrement pertinent pour des problèmes d'optimisation multi-critères où la connaissance de la combinaison de critères définissant une bonne solution est de nature partielle et dépend du contexte. Dans le domaine des systèmes

d'ordonnancement en ligne, le problème n'est pas seulement de trouver une allocation des tâches, une fois pour toutes, mais de maintenir un ordonnancement au cours des interactions avec l'environnement : par exemple prendre en compte la panne d'une machine, le changement de priorité des tâches à accomplir, etc.

La réalisation de tels systèmes réactifs et interactifs en PLC nécessite de doter le modèle d'exécution de mécanismes de prise en compte d'événements extérieurs. Un choix de réalisation consiste à considérer uniquement des événements de modification de la requête principale, les communications s'effectuent alors au travers de ces modifications. Maher et Stuckey [10] ont défini des commandes de manipulation de la requête et un modèle d'exécution incrémental qui procède seulement par ajout de contraintes et de buts. Les méthodes de ré-exécution par oracles pour l'ajout et le retrait de contraintes à un but ont été décrites dans [12]. Ces méthodes ne permettent toutefois pas de combiner toutes les possibilités d'ajout et de retrait incrémental de contraintes et de buts à la requête.

Dans cet article nous étudions un modèle d'exécution réactif et interactif pour la PLC qui autorise toutes les commandes de manipulation de la requête. Ce modèle d'exécution repose d'une part sur un résolveur générique de contraintes supportant l'ajout et le retrait incrémental de contraintes [6] [11], et d'autre part sur un système de transformation de dérivations CSLD supportant l'ajout et le retrait d'atomes et de contraintes.

Dans la troisième section nous présentons le système de transformation des dérivations pour chaque type de modification de la requête, ainsi que la procédure de recherche réactive d'exploration de l'arbre de dérivation CSLD de la requête courante, qui permet la prise en compte asynchrone et préemptive des commandes de modification de la requête.

Dans la quatrième section nous dérivons du modèle d'exécution réactif une présentation des méthodes de Maher et Stuckey [10] et de Van Hentenryck [12] et montrons que le modèle d'exécution réactif autorise des interactions non réalisables avec ces méthodes.

La dernière partie illustre la mise en œuvre de ce modèle d'exécution dans deux applications et fournit des éléments d'évaluation des performances de l'implémentation réalisée.

2 Préliminaires

Les notions de base de la programmation logique avec contraintes sont supposées connues, le lecteur pourra se reporter à [8] par exemple pour plus de détails. On considère des programmes logiques avec contraintes sur une structure $\mathcal{D}$ qui fixe le domaine d'interprétation. Une contrainte c est une conjonction de contraintes de base, les contraintes seront manipulées aussi comme des *multi-ensembles* de contraintes de base, l'union est notée c, d et la différence multi-ensemble $c \setminus d$ (le multi-ensemble vide représente la contrainte *true*). Une clause de programme PLC, notée $A \leftarrow c|\alpha$, est formée d'un atome de tête A, et d'un corps formé d'une contrainte c et d'un multi-ensemble α d'atomes (le multi-ensemble vide est noté $\Box$). Un but est une clause de la forme $\leftarrow c|\alpha$. Cependant dans la suite nous utilisons la notation simplifiée $c|\alpha$ à la place de $\leftarrow c|\alpha$. Pour simplifier les notations nous supposerons de plus que dans les clauses de programme et les buts, les atomes sont formés uniquement avec un symbole de prédicat et des variables distinctes, les symboles de constantes et de fonctions apparaissent uniquement dans les contraintes.

Les programmes PLC sont interprétés opérationnellement par un système de transitions simple sur les buts défini par la règle de résolution CSLD. Pour notre propos il est utile de représenter explicitement les buts en échec contenant une contrainte insatisfiable. Le test de satisfiabilité des contraintes dans la règle de résolution CSLD porte donc délibérement sur le but à résoudre, pas sur le but résolu qui peut être incohérent.

$$CSLD \frac{(A \leftarrow d|\beta) \in P \quad \mathcal{D} \models \exists c}{c|A, \alpha \overset{A \leftarrow d|\beta}{\rightarrow} c, d|\beta, \alpha}$$

L'atome A dans la transition est appelé l'*atome sélectionné*. Une *dérivation CSLD* pour un but G_0 est une suite de transitions CSLD à partir de G_0, notée $G_0 \overset{A_1 \leftarrow c_1|\alpha_1}{\rightarrow} G_1 \overset{A_2 \leftarrow c_2|\alpha_2}{\rightarrow} \ldots$. Par la suite, afin de simplifier les

notations nous écrirons $G_0 \rightarrow G_1 \rightarrow ...$ lorsqu'il n'est pas nécessaire de spécifier la règle $A \leftarrow d|\beta$ appliquée. Une dérivation pour G est un *succès avec contrainte réponse* c si elle est finie et termine avec un but de la forme $c|\Box$ avec c satisfiable. Une dérivation est un *échec fini* si elle est finie et termine avec un but contenant une contrainte insatisfiable. Un arbre de dérivation CSLD pour G est l'arbre de toutes les dérivations obtenu en fixant un atome sélectionné en chaque nœud.

3 Modèle d'exécution réactif pour la programmation logique avec contraintes

Le modèle d'exécution réactif est basé sur un ensemble de commandes de modification de la requête, pour l'ajout et le retrait de contraintes et de buts, et sur un système de transformations des dérivations.

3.1 Opérations sur les dérivations

Deux opérations sur les dérivations CSLD sont définies : l'élagage d'une dérivation CSLD par une contrainte (i-e l'ajout d'une contrainte) et le relèvement d'une dérivation CSLD par une contrainte (i-e le retrait d'une contrainte). De même nous définissons l'ajout d'un multi-ensemble d'atomes au but initial d'un arbre CSLD, et sa contrepartie quelque peu plus complexe : le retrait d'un multi-ensemble d'atomes du but initial de la dérivation.

Définition 1 *L'***élagage** *d'une dérivation CSLD, Δ, par une contrainte c, noté $\Delta \otimes c$, est la dérivation obtenue par l'addition de la contrainte c aux buts de Δ. L'opération d'élagage est définie récursivement par :*

$$\begin{array}{lcl} (d|\beta) \otimes c & = & (c, d|\beta), \\ (d|\beta \rightarrow \Delta') \otimes c & = & (c, d|\beta \rightarrow (\Delta' \otimes c)) \textit{ si } c \wedge d \textit{ est satisfiable,} \\ & & (c, d|\beta) \textit{ sinon.} \end{array}$$

Le **relèvement** *d'une dérivation CSLD, Δ, par une contrainte c élément du but, noté $\Delta \oslash c$, est la dérivation obtenue par le retrait de la contrainte c aux buts de Δ :*

$$\begin{array}{lcl} (c, d|\alpha) \oslash c & = & (d|\alpha), \\ (c, d|\alpha \rightarrow \Delta') \oslash c & = & (d|\alpha \rightarrow (\Delta' \oslash c)). \end{array}$$

*L'***ajout des atomes** *α à une dérivation CSLD, Δ, noté $\Delta \oplus \alpha$, est obtenu par l'addition des atomes α aux buts de Δ :*

$$\begin{array}{lcl} (c|\beta) \oplus \alpha & = & (c|\alpha, \beta), \\ (c|\beta \rightarrow \Delta') \oplus \alpha & = & (c|\alpha, \beta \rightarrow (\Delta' \oplus \alpha)). \end{array}$$

Le retrait d'un multi-ensemble d'atomes demande plus d'attention. En effet le retrait d'un atome dans une dérivation demande que les atomes et la contrainte introduits par le pas de résolution CSLD de l'atome soient également retirés.

Définition 2 *Le* **retrait des atomes** *α à une dérivation CSLD, Δ, dont le but initial contient les atomes α, noté $\Delta \ominus \alpha$, est défini récursivement par :*

$$\begin{array}{lcl} (c|\alpha, \beta) \ominus \alpha & = & (c|\beta), \\ (c|\alpha, \beta \overset{A \leftarrow d|\gamma}{\rightarrow} \Delta') \ominus \alpha & = & (c|\beta \overset{A \leftarrow d|\gamma}{\rightarrow} (\Delta' \ominus \alpha)) \textit{ if } A \notin \alpha, \\ & = & \Delta' \oslash d \ominus \alpha' \textit{ if } A \in \alpha \textit{ and } \alpha' = (\alpha \setminus \{A\}) \cup \gamma. \end{array}$$

L'élagage d'une dérivation ne change pas l'ordre des atomes sélectionnés dans une dérivation. Pour conserver un nombre maximum de pas de résolution CSLD, il est possible de retarder la sélection de l'atome responsable de l'insatisfiabilité avec la contrainte ajoutée, au lieu de couper la dérivation à la première insatisfiabilité rencontrée. L'opération suivante formalise le retard des étapes de résolution qui introduisent une contrainte donnée.

Définition 3 *Le* **retard** *des pas de résolution qui introduisent une contrainte c dans une dérivation CSLD Δ, noté $\Delta \odot c$, est défini recursivement par :*

$$\begin{array}{lcl}(d|\alpha) \odot c & = & (d|\alpha), \\ (d|\alpha \overset{A \leftarrow e|\beta}{\rightarrow} \Delta') \odot c & = & (d|\alpha \overset{A \leftarrow e|\beta}{\rightarrow} (\Delta' \odot c)) \textit{ si } e \neq c, \\ & = & (\Delta' \oslash c \ominus \beta \oplus \{A\}) \odot c \textit{ si } e = c.\end{array}$$

3.2 Transformations des dérivations

Il convient d'examiner les propriétés souhaitables d'une transformation pour la commande d'ajout d'une contrainte à la requête. Un aspect important dans la recherche d'une nouvelle solution est d'utiliser activement la nouvelle contrainte dans cette recherche [10, 12], un deuxième est de minimiser la ré-exécution à effectuer [2].

La solution que nous avons retenue pour l'ajout d'une contrainte intègre ces deux propriétés. La transformation de dérivation associée à l'ajout d'une contrainte au but n'est pas simplement la dérivation obtenue par élagage (opération $\otimes$), mais une dérivation préservant le plus grand nombre d'étapes de résolution, obtenue en retardant la sélection des atomes causant une insatisfiabilité avec la contrainte ajoutée (opération $\odot$). Une transformation pour l'ajout d'une contrainte est illustrée dans l'exemple 1.

Exemple 1 *La transformation pour l'ajout d'une contrainte est illustrée sur un exemple typique d'ordonnancement disjonctif sur les entiers naturels.*

```
disj(X,Y,DX,DY):-Y >= X + DX.
disj(X,Y,DX,DY):-X >= Y + DY.
```

Le but $G = (m \geq x, m \geq y, m \geq z \mid disj(y,z,2,1), disj(x,z,1,1), disj(x,y,1,2))$ *admet une dérivation réussie* Δ*:*

$\Delta : m \geq x, m \geq y, m \geq z \mid \underline{disj(y,z,2,1)}, disj(x,z,1,1), disj(x,y,1,2)$
$\rightarrow m \geq x, m \geq y, m \geq z, z \geq y+2 \mid \underline{disj(x,z,1,1)}, disj(x,y,1,2)$
$\rightarrow m \geq x, m \geq y, m \geq z, z \geq y+2, z \geq x+1 \mid \underline{disj(x,y,1,2)}$
$\rightarrow m \geq x, m \geq y, m \geq z, z \geq y+2, z \geq x+1, y \geq x+1 \mid \square$

L'ajout de la contrainte $2 \geq m$ *au but* G *est effectué par la commande* add$-$constraint$(2 \geq m)$*, qui transforme la dérivation* Δ *en la dérivation* Δ'*, ou* $\Delta' = \Delta \odot z \geq y+2 \otimes 2 \geq m$*.*

$\Delta' : 2 \geq m, m \geq x, m \geq y, m \geq z \mid disj(y,z,2,1), \underline{disj(x,z,1,1)}, disj(x,y,1,2)$
$\rightarrow m \geq x, m \geq y, m \geq z, z \geq x+1 \mid disj(y,z,2,1), \underline{disj(x,y,1,2)}$
$\rightarrow 2 \geq m, m \geq x, m \geq y, m \geq z, z \geq x+1, y \geq x+1 \mid disj(y,z,2,1)$

La recherche continue à partir de la dérivation transformée Δ'*, et une dérivation réussie est trouvée en un pas de résolution.*

La transformation pour l'ajout de contrainte est donc définie formellement à l'aide des opérations $\otimes$ et $\odot$. Les transformations associées aux autres commandes de retrait de contraintes et d'ajout-retrait d'atomes sont obtenues par les opérations correspondantes (i.e. $\oslash$, $\oplus$ et $\ominus$).

Définition 4 *Soient G un but et Δ une dérivation CSLD pour G, les commandes de modification du but sont définies par un système de transformation du couple $< G, \Delta >$*[1].

$< d|\alpha; \Delta >$ add−constraint(c) $< c, d|\alpha;\ \Delta \otimes c >$ *si* $e \wedge c$ *est satisfiable*
où $e|\gamma$ *est le but final dans* Δ
ou si $c \wedge d$ *est insatisfiable.*
$< c, d|\alpha;\ \Delta \odot c_1 ... \odot c_n \otimes c >$
si $\{c_1, ..., c_n\}$ *est un ensemble de contraintes*
introduit dans les étapes de résolution dans Δ
tel que $(c, e \setminus c_1 ... \setminus c_n)$ *est satisfiable*

$< c, d|\alpha;\ \Delta >$ del−constraint(c) $< d|\alpha;\ \Delta \oslash c >$

$< c|\beta;\ \Delta >$ add−atoms(α) $< c|\alpha, \beta;\ \Delta \oplus \alpha >$

$< c|\alpha, \beta;\ \Delta >$ del−atoms(α) $< c|\beta;\ \Delta \ominus \alpha >$

On vérifie aisément que les transformations des dérivations associées à la modification d'un but définissent des dérivations pour le but modifié.

Proposition 5 (Correction des manipulations de requête) *[7] Soient Δ une dérivation CSLD pour un but G et $< \Delta', G' >$ le but transformé pour la dérivation obtenue par la manipulation d'une requête. La dérivation Δ' est une dérivation CSLD pour le but G'.*

Le modèle d'exécution réactif consiste à développer un arbre CSLD pour le but modifié contenant la dérivation transformée. Comme pour toute dérivation pour G il existe un arbre de dérivation CSLD pour G contenant cette dérivation, la complétude du modèle d'exécution réactif s'ensuit sans préciser une stratégie de sélection particulière.

Théorème 6 (Complétude du modèle d'exécution) *[7] Soient Δ une dérivation CSLD pour un but G et $< \Delta', G' >$ le but transformé pour la dérivation obtenu par la manipulation d'une requête. Alors le modèle d'exécution réactif développe un arbre CSLD pour le but modifié, G'.*

3.3 Recherche réactive

Nous devons considérer les stratégies de recherche adéquates pour le programmation logique avec contraintes réactive. Notons que la stratégie de recherche doit permettre de parcourir l'arbre de dérivation à partir d'un noeud interne, correspondant à la dérivation transformée en premier lieu, sinon cela réduit l'intérêt de garder des informations sur les dérivations. La figure 1 illustre cette idée de parcours par élargissement progressif à partir d'un noeud interne.

Un parcours réactif permet d'explorer l'arbre par élargissement progressif autour de la dérivation transformée, en prenant en compte de façon asynchrone et préemptive les commandes de modification du but. Une procédure de recherche réactive est présentée dans la figure 2 sous la forme d'un meta-interpréteur dans lequel on suppose que les prédicats sont définis par deux clauses au plus.

La procédure de réfutation réactive est appelée par le prédicat `react`. Le premier argument de `react` représente le but de la dérivation courante. Chaque atome de ce but est représenté par les deux règles le

[1] La commande d'ajout d'une contrainte peut produire des transformations différentes selon le choix d'un sous-ensemble satisfiable de contraintes. Le choix d'un sous-ensemble satisfiable peut être basé sur les informations de dépendance utilisées par un solveur incrémental présenté dans la partie 3.4. On remarque que la taille et la structure des arbres peuvent être différentes, selon le sous-ensemble satisfiable choisi. Dans le contexte de la PLC la notion de sous-ensemble satisfiable *minimal* doit rendre compte des dépendances entre des étapes de résolution dans une dérivation. Toutefois si la contrainte ajoutée est satisfiable avec les contraintes dans la dérivation, alors aucun choix n'est effectué. Dans le cas extrême où la contrainte ajoutée est en conflit avec la contrainte dans le but, alors la dérivation est élaguée jusqu'à la racine.

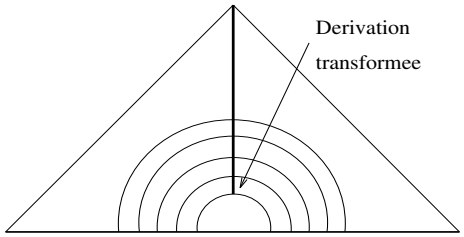

Figure 1 : Stratégie de recherche réactive

définissant (`A<-C\LC, A<-D\LD`). Le deuxième argument est la suite des règles du programme utilisé dans la dérivation. Cette information est gardée sous la forme d'une liste de couples de règles, où la première règle du couple est la règle utilisée dans l'étape de résolution. Les arguments du prédicat `parcours` et `retro-parcours` sont du même type. La présence d'une opération explicite de retrait de contrainte `del` offre la possibilité d'écrire une procédure de parcours *déterministe*. L'opération d'ajout de contrainte est effectuée par le prédicat `add`, le prédicat `rules` associe les règles définissant les atomes.

```
react(G,D):-
  (modification(M)->
      transformation(M,G,D,G1,D1),
      parcours(G1,D1),
      retro-parcours(G1,D1)
  ;
      parcours(G,D)).

parcours([],Der):-reussi.
parcours([(A<-C\LC,A<-D\LD)|G],Der):-
        rules(LC,GC),
        rules(LD,GD),
        conc(GC,G,G1),
        conc(GD,G,G2),
        (add(C)->react(G1,[(A<-C\LC,A<-D\LD)|Der]),del(C);true),
        (add(D)->react(G2,[(A<-D\LD,A<-C\LC)|Der]),del(D);true).

retro-parcours(G,[]):-echec.
retro-parcours(G,[(A<-C\LC,A<-D\LD)|Der]):-
         del(C),
         rules(LD,GD),
         conc(GD,G,G2),
         (add(D)->parcours(G2,[(A<-D\LD,A<-C\LC)|Der]),del(D);true),
         retro-parcours([(A<-C\LC,A<-D\LD)|G],Der).
```

Figure 2 : Meta-interpréteur réactif

Lorsqu'un évènement de modification de la requête est saisi par le prédicat `modification`, le prédicat `transformation` transforme le but et la dérivation. Ce prédicat encapsule les tranformations de dérivation définies formellement et applique la transformation appropriée. Ensuite la recherche pour une dérivation réussie commence à partir de la dérivation courante. Le parcours est par élargissement progressif autour de la dérivation transformée. Il est appelé de la façon suivante `<-parcours(G,Der), retro-parcours(G,Der)`. Les dérivations sous le point de départ sont explorées en priorité en profon-

deur, de gauche à droite. Ensuite le prédicat `retro-parcours` remonte la dérivation progressivement en explorant les autres parties de l'arbre de dérivation. Lorsque une dérivation succès est trouvée, le prédicat `réussi` est appelé, à la fin du parcours le prédicat `echec` est appelé.

3.4 Solveur de contraintes incrémental avec ajout et retrait de contraintes

Le modèle d'exécution réactif suppose que le résolveur de contrainte est capable de tester la satisfiabilité d'un ensemble de contraintes qui évolue par ajout et retrait de contraintes `del(c)`. Traditionnellement, le retrait d'une contrainte est effectuée à travers le mécanisme de backtracking, ce qui entraîne la perte de toutes les inférences suivant la pose de la contrainte à relâcher. Les solveurs de contraintes dynamiques permettent de traiter le relâchement de contraintes plus efficacement (cf. [4] [1] et [13]). Dans [11] et [6], un algorithme de résolution de contrainte dynamique est présenté dans un cadre abstrait où les contraintes sont identifiées à des opérateurs de fermeture. L'implantation réalisée s'appuie sur une gestion du graphe de dépendance dynamique des contraintes. Nous renvoyons à [11] et [6] pour l'étude du solveur de contraintes dynamiques et de son intégration dans le système de PLC réalisé.

4 Comparaison

Le modèle d'exécution réactif de la PLC est un modèle d'exécution très général qu'il est possible de comparer en particulier aux modèles de ré-exécution basés sur des oracles et au modèle de Maher et Stuckey pour les commandes d'ajout de contraintes et de buts à la requête. Dans cette section nous présentons ces méthodes avec les formalismes des opérations sur les dérivations, et comparons ces méthodes.

4.1 Ré-exécution avec oracles

Van Hentenryck [12] propose un modèle d'exécution pour la manipulation de requêtes d'ajout et de retrait de contrainte. Cette méthode implémente uniquement la transformation pour le retrait de contrainte et la transformation simplifiée pour l'ajout de contrainte sans retarder des pas de résolution insatisfiables avec la contrainte ajoutée.

Un oracle est une dérivation réussie pour un but G. L'oracle est utilisé pour développer un arbre CSLD pour le but modifié en explorant en priorité la branche correspondant à cet oracle. Nous pouvons analyser ce modèle dans le formalisme des opérations sur les dérivations. La transformation utilisée pour l'ajout d'une contrainte à la requête s'exprime en terme d'opération d'élagage sur la dérivation de l'oracle.

Définition 7 *Soit Δ un oracle réussi pour un but $d \mid \alpha$ et un programme P. La transformation pour l'ajout d'une contrainte c est*

$$< d \mid \alpha, \Delta > oracle - add - constraint < c, d \mid \alpha, \Delta \otimes c > .$$

En conséquence toutes les informations sur la dérivation après la première inconsistance pour guider la recherche vers une dérivation réussie sont perdues. Dans notre modèle l'opération d'ajout de contraintes permet d'utiliser l'oracle pour un plus grand nombre d'étape de résolution grâce à l'opération de retard.

Dans [12] deux traitements sont proposés pour la transformation suite au retrait d'une contrainte. Le premier repose sur le même principe que celui utilisé pour l'ajout d'une contrainte, i-e l'élagage d'une dérivation, en utilisant un oracle pour un but moins contraint supposé stocké dans le système.

Définition 8 *Soient un but $c, d \mid \alpha$, et Δ un oracle réussi pour un but $e \mid \alpha$ et un programme P, et $e, f = d$. La transformation pour le retrait d'une contrainte c est*

$$< e \mid \alpha, \Delta > oracle - del - constraint < d \mid \alpha, \Delta \otimes f > .$$

La deuxième transformation est basée sur l'utilisation d'un oracle pour le but avant le retrait de la contrainte. Cette transformation se formalise par l'opération de relèvement.

Définition 9 *Soit Δ un oracle réussi pour un but $c, d \mid \alpha$ et un programme P. La transformation pour le retrait d'une contrainte avec heuristique c est*

$$< c, d \mid \alpha, \Delta > oracle - del - constraint - heuristic < d \mid \alpha, \Delta \oslash c > .$$

La procédure de recherche pour la méthode de ré-exécution par oracle est basée sur une exploration d'un arbre CSLD qui contient la dérivation transformée. Van Hentenryck propose une optimisation de la procédure de recherche pour une classe restreinte des arbres CSLD, qui permet de ne pas re-parcourir les parties de l'arbre déjà explorées avant l'ajout d'une contrainte.

4.2 Les transformations Maher-Stuckey

Maher et Stuckey proposent un modéle d'exécution pour la manipulation de requête dans [10]. Cependant à la différence de notre approche le modèle est défini uniquement pour les opérations d'*ajout* de contraintes ou d'atomes à la requête, ainsi que pour la recherche de solutions optimales.

La méthode de Maher-Stuckey s'applique à des arbres CSLD formés avec la stratégie de sélection gauche-droite de Prolog. L'ajout d'un atome A à une requête $c|\alpha, \beta$ peut se faire au milieu entre α et β. La transformation de l'arbre CSLD consiste à couper les dérivations en un point correspondant à la résolution du but $c|\alpha$, à insérer un arbre de dérivation pour A et à greffer l'arbre de dérivation pour β. La figure 3 illustre cette transformation.

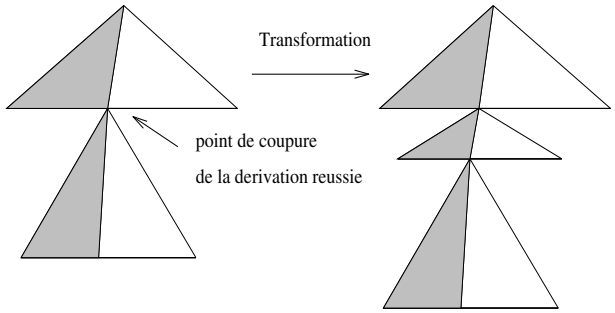

Figure 3 : Transformation et Procédure de recherche

La procédure de recherche pour la méthode est basée sur une exploration de l'arbre transformé. La procédure de recherche peut être optimisée pour éviter de parcourir les dérivations d'échec fini transformées et déjà parcourues, qui demeurent des dérivations d'échec fini. La partie grise de l'arbre CSLD de la figure 3 correspond à la partie de l'arbre CSLD qu'il n'est pas nécessaire de reparcourir.

Notre modèle d'exécution ne préjuge pas d'une stratégie de sélection particulière, l'opération d'ajout d'atome au milieu du but n'influence pas la transformation qui ajoute le but toujours à la fin de la dérivation. L'opération d'ajout d'atome dans la méthode Maher-Stuckey correspond donc à une autre transformation des dérivations qui peut être formalisée de la façon suivante :

Définition 10 *Soit un but $G = c|\alpha, \beta$ et A un atome à ajouter au but entre α et β. Soit Δ une dérivation finie pour G de la forme, $\Delta = \Delta' \rightarrow d(|\beta) \rightarrow \Delta''$. Soit Δ_1 une dérivation pour $d|A$. La dérivation transformée pour le but $c|\alpha, A, \beta$ est la dérivation:*

$\Delta' \rightarrow (\Delta_1 \oplus \beta) \rightarrow (\Delta'' \otimes e)$ *si Δ_1 est une dérivation succès avec réponse calculée e,*

$\Delta' \rightarrow (\Delta_1 \oplus \beta)$ *si* Δ_1 *est une dérivation en échec pour* A.

Notons que dans notre modèle l'opération d'ajout de contraintes avec retard peut avoir un effet similaire.

5 Applications

Dans cette section nous illustrons le modèle d'exécution réactif et interactif de la PLC sur deux applications d'optimisation combinatoire qui combinent toutes les commandes de manipulation de la requête.

5.1 Allocation multi-fréquences

La première application que nous avons développée porte sur les problèmes d'allocation de fréquences dans les réseaux de communications. Le problème central consiste à allouer les fréquences d'un spectre radio à un ensemble de réseaux. Le spectre radio comporte deux à trois mille fréquences distinctes. Le système doit allouer des bandes de fréquences pour des centaines de réseaux. Ces réseaux sont partionnés en sous-ensembles d'une vingtaine de réseaux fortement connexes. Les fréquences sont ensuite allouées à ces sous-ensembles.

Cette allocation doit respecter des contraintes obligatoires : certaines bandes de fréquences sont interdites (ex. bandes brouillées ou réservées) ; le nombre de fréquences allouées à un réseau est borné et est constitué de deux blocs de fréquences ; deux réseaux proches doivent utiliser des fréquences suffisament éloignées pour réduire les interférences, trois degrés de proximité sont considérés, la distance à respecter en fréquences est une fonction qui dépend à la fois du degré de proximité des réseaux et des fréquences allouées.

L'optimisation des allocations de fréquences représente la partie la plus difficile de cette application. Ceci résulte du fait qu'il y a plusieurs critères de préférence et que ceux-ci sont antagonistes. Nous avons considéré deux critères : la maximisation du nombre de fréquences allouées par réseau (résistance au brouillage et à l'interception) et la maximisation de la séparation des deux blocs de fréquences alloués à un même réseau.

En plus des techniques d'optimisation multi-critère décrites dans [5], la capacité du système à réagir aux interactions de l'utilisateur pour améliorer les solutions sur des critères donnés demeure d'une importance primordiale. L'opérateur peut interagir avec le système :

- en demandant d'augmenter le nombre de fréquences allouées à un réseau désigné,
- en demandant d'augmenter la séparation des blocs de fréquences sur un réseau désigné,
- en modifiant les bandes de fréquences interdites.

Ces interactions sont réalisées par les commandes de modification de la requête. Dans la suite nous décrivons l'effet du modèle d'exécution réactif pour ces interactions.

Pour augmenter le nombre de fréquences `NFj` d'un réseau choisi `j`, l'idée est d'ajouter une contrainte `add-constraint(NFj > cj)` au but, la contrainte impose que le nombre de fréquences pour la nouvelle allocation après l'interaction doit être supérieure au nombre de fréquences (`cj`) disponibles pour le réseau choisi avant l'interaction. La contrainte n'est pas compatible avec la dérivation courante et donc la transformation de la dérivation sollicite l'opération `del` pour retirer le sous-ensemble de contraintes insatisfiables avec l'introduction de cette contrainte. Le choix d'un tel sous-ensemble est restreint par l'addition de contraintes `add-constraint(NFi >= ci)` pour chaque réseau `i` compatible avec la dérivation courante. Ces contraintes sont ajoutées au but avant l'addition de la contrainte `add-constraint(NFj > cj)`, elles imposent que le nombre de fréquences allouées à chaque réseau aprés l'interaction soit supérieur

ou égal à celui avant l'interaction.

Des contraintes semblables sont ajoutées pour augmenter la séparation des blocs de fréquences. Ceci est réalisé par l'ajout de contraintes empêchant le système de trouver une allocation diminuant le nombre des fréquences ou l'écartement entre les blocs de fréquences pour les autres réseaux. La manipulation de requête `add-constraint` est appliquée pour contraindre le réseau choisi d'avoir une séparation de blocs de fréquences supérieur par rapport à sa séparation avant interaction.

L'ajout de fréquences interdites repose sur l'ajout de contraintes sous-jacentes à la présence d'une bande de fréquences interdites pour chaque réseau. L'interdiction de la bande de fréquences allant de `DFF` et finissant à `EFF` pour un bloc `i` commençant à `DBI` et de longueur `Li`, est réalisée par l'ajout d'une contrainte disjonctive `add-constraint(or(DBi + Li =< BFF,DBi >= EFF))` pour chaque bloc et réseau. Quand la contrainte n'est pas compatible avec la dérivation courante, la transformation de la dérivation peut avoir recours à l'opération `del` pour retirer le sous-ensemble de contraintes insatisfiables avec l'introduction de cette contrainte. Contrairement à l'interaction d'optimisation aucune contrainte n'est ajoutée pour limiter le choix du sous-ensemble à retirer, et l'allocation trouvée après l'interaction peut être moins bonne que l'allocation trouvée avant l'interaction.

Dans un scénario typique, une allocation initiale est améliorée progressivement par la sélection d'un réseau et l'amélioration du nombre de fréquences qui lui est alloué ou par l'augmentation de l'écart entre ses blocs de fréquences. Ce processus continue jusqu'à l'obtention d'une bonne allocation. Quand les fréquences interdites sont changées une nouvelle allocation est trouvée et le processus de raffinage de l'allocation peut continuer.

Le démonstrateur d'allocation de fréquences montre que le modèle d'exécution réactif de la PLC permet de résoudre efficacement des problèmes dynamiques et hautement interactifs. La puissance des manipulations de requêtes augmente considérablement la facilité de réaliser un démonstrateur interactif et permet d'élaborer des stratégies complexes d'optimisation multi-critères.

5.2 Séquencement des vols

La gestion du traffic aérien dans la zone terminale pose de nombreux problèmes d'optimisation combinatoire. Lors des périodes de pointes, la gestion des arrivées en zone terminale (vols à 30-40 minutes de l'aéroport) devient une tâche très complexe due au manque de ressources (les pistes). Le problème central pour le contrôleur aérien est de trouver une séquence d'atterrissage des vols entrant dans une zone terminale.

La solution au problème de séquencement retenue dans [9], présente l'avantage de combiner plusieurs modélisations du problème qui améliorent l'efficacité de la résolution en PLC. Pour simplifier nous considérons dans cette section une seule modélisation. Le problème de séquencement est vu comme un problème d'ordonnancement avec contraintes disjonctives. A chaque vol on associe une variable STAi représentant son heure d'arrivée, avec des contraintes de date au plus tôt et de date au plus tard. L'expression des contraintes de distance de sécurité entre appareils se traduit par des contraintes disjonctives `disj(STAi,STAj,Dij,Dji)` définies comme dans l'exemple 1, où `Dij`, resp. `Dji`, représente la séparation minimale entre les vols, lorsque j suit i, resp. i suit j. Les contraintes de position des vols dans les couloirs aériens définissent des contraintes de précédence du même type, qui interdisent certaines permutations. Le critère de minimisation est la date d'arrivée du dernier vol de la séquence ou sa combinaison linéaire avec les retards de certains vols prioritaires.

Le problème de ré-ordonnancement dynamique est résolu par le modèle d'exécution réactif en traitant les interactions suivantes:

• Addition d'un vol. Cette interaction crée une nouvelle variable STA associée à ce vol, et requiert l'ajout de contraintes de précédence avec les autres vols non-permutables, par le biais de l'opération

add_constraint. Il est également nécessaire d'ajouter une contrainte disjonctive par l'addition d'un atome `add_atom(disj(STAx,STAy,Dx,Dy))` au but pour chaque vol dans la zone terminale ayant la possibilité de permuter avec le vol ajouté.

• Le retrait d'un vol est le complément de l'ajout et consiste au retrait des contraintes et des atomes (**del_constraint** et **del_atom**) introduits par la présence de ce vol.

• Arrivée d'un vol. Cette interaction demande une gestion prenant en compte l'évolution temporelle du problème. Tous les plans de vols des avions doivent être mis à jour par rapport à la date d'arrivée du premier vol dans la séquence courante. Ceci se traduit par l'ajout de contraintes au but, car les distances de sécurité à respecter par les vols sont de plus en plus sévères à l'approche de la piste d'atterrissage.

• Optimisation d'une séquence de vols. Dans ce cadre, l'optimisation est modélisée comme un cas particulier d'interaction avec un agent qui contraint de plus en plus le temps final d'atterrissage.

Le tableau 1 présente les temps de calcul en ms pour trouver une solution d'un problème de séquencement au cours d'une suite d'interactions. Pb est l'identificateur du problème. Le problème de départ (1) est d'ordonner 20 vols dans deux couloirs aériens. Le problème évolue de la façon suivante : le premier avion atterrit (2), un nouvel avion rentre en zone terminale (3), un deuxième avion arrive dans la zone terminale (4), un avion est dérouté (5), un deuxième avion atterri (6), un troisième avion arrive dans la zone terminale (7), un troisième avion atterri (8), et finalement un quatrième avion arrive dans la zone terminale (9).

Les temps pour trouver une solution avec une approche statique (S) et réactive (R) sont donnés dans le tableau 1. Les temps sont présentés aussi pour trouver la séquence optimale après chaque évolution du problème avec une approche statique (S_{opt}) et réactive (R_{opt}). Ces temps ont été obtenu sur une sparc 1. La version compilée en C de Meta(F), initialement décrit dans [3], a été utilisé pour résoudre les problèmes statiques et l'extension réactive pour la résolution réactive.

pb	S	R	S_{opt}	R_{opt}	S/R	S_{opt}/R_{opt}
1	99	110	2909	3257	**0.90**	**0.89**
2	160	10	3448	40	**16.0**	**86.2**
3	149	20	3839	10	**7.45**	**384**
4	160	10	10050	88	**16.0**	**114**
5	140	1510	3949	4380	**0.09**	**0.90**
6	129	20	3907	60	**6.45**	**65.1**
7	99	20	9425	528	**4.95**	**17.9**
8	99	10	8050	60	**12.0**	**134**
9	99	10	5179	10	**14.0**	**518**

Tableau 1 : Temps de calcul pour le séquencement des vols

L'évaluation démontre que le modèle d'exécution que nous proposons pour la programmation logique est efficace par rapport à un modèle d'exécution normal. Les gains de temps réalisés par le traitement réactif sont substantiels, surtout quand il s'agit de trouver des séquences optimales (donc des solutions de qualité égale). Ceci est une double conséquence du traitement de l'ajout d'une contrainte au but pour l'optimisation réactive. Primo le processus de séparation et évaluation part d'une solution plus proche de l'optimal, et deuxio une nouvelle solution est trouvée plus rapidement. La contre performance du modèle pour la cinquième interaction de retrait d'un avion est due au traitement du retrait de contraintes dans le solveur. Le retrait d'un avion engendre le retrait des contraintes de cette avion avec tous les autres avions dans le zone terminale. Actuellement ces contraintes sont retirées une par une, c'est-à-dire une première contrainte est retirée, puis l'arbre est mis à jour, une nouvelle contrainte est retirée et ainsi de suite. Cependant

le retrait de ces contraintes pourrait être traité beaucoup plus efficacement en une seule opération.

6 Conclusion et perspectives

Ce travail a permis de concevoir et d'évaluer un schéma de programmation logique avec contraintes pour les systèmes réactifs. Dans ce schéma, l'interaction de l'environnement sur le système est décrite par quatre types de manipulations de requête. Le modèle d'exécution est basé sur l'utilisation des informations des preuves précédentes.

Un cadre formel pour étudier un tel modèle d'exécution a été développé. À la base de ce développement théorique se trouve la définition d'un ensemble d'opérations sur les dérivations CSLD. Ces opérations permettent d'ajouter, de retirer des contraintes et des atomes, et aussi de retarder la sélection d'un atome. Un nouveau modèle d'exécution a été proposé, celui-ci est basé sur les *transformations de dérivations* et le concept de *recherche réactive*. Ces tranformations sur les dérivations ont été définies formellement comme des combinaisons d'opérations sur ces dérivations. La correction de ces transformations est établie ainsi que la complétude du modèle d'exécution pour les procédures de recherche réactive. Le retrait incrémental des contraintes et la capacité d'analyser les dépendances entre ces contraintes sont au coeur de ce nouveau modèle.

L'implantation réalisée a permis de modéliser et résoudre des problèmes réels d'optimisation combinatoire dans un environnement dynamique. L'extension a été mise à l'épreuve sur deux applications : un systèmes d'aide à la décision pour un problème d'optimisation multi-critères, et un problème d'ordonnancement d'avions dans un environnement simulé. Le schéma a démontré sur ces applications sa puissance pour l'optimisation, la pertinence de ses commandes de manipulation de requêtes pour la construction des systèmes d'aide à la décision complexe et son efficacité par rapport à un modèle d'exécution statique.

Suite à ce travail nous pouvons envisager des développements dans les domaines de recherche suivants.

- Consolider les algorithmes d'ajout et retrait de contraintes développés pour le solveur, notamment pour les points suivants : les problèmes de gestion de la mémoire, d'extension à d'autres types de contraintes, et la gestion interne du graphe de dépendance des variables. Actuellement la gestion de la mémoire ne récupére pas l'espace laissé suite au retrait d'une contrainte.

- Développer une nouvelle machine abstraite pour la PLC réactive pour permettre un développement plus systématique des applications. Dans la version actuelle du solveur, la communication d'un processus avec le démonstrateur pour une manipulation de requête ou la réception d'un signal de succès ou d'échec est faite d'une façon *ad hoc* pour chaque application.

- Expérimenter d'autres algorithmes de recherche pour le parcours de l'arbre CSLD comme : le recuit simulé, les algorithmes génétiques, la réparation de solution, ou encore la méthode TABU. Le système de transformation des dérivations peut donner une présentation unifiée de ces procédures de recherche et donc permettre de les implanter en s'appuyant sur le modèle d'exécution présenté dans cet article.

Remerciements

Cette recherche a été en partie financée par le Ministère de la Défense, contrat DRET 91 34 402.

Références

[1] C. Bessière. Arc-consistency in dynamic constraint satisfaction problems. In *Proceedinds of the 9th AAAI*, pages 221–226, Anaheim, California, USA, 1991. MIT Press.

[2] P. Chatalic. *IMPRO: An environment for incremental execution in Prolog*. rapport technique TR-LP-42, European Computer-Industry Research Centre, European Computer-Industry Research Centre GmbH, Arabellastr. 17, D-8000 Muenchen 81, Germany, May 1989.

[3] P. Codognet, F. Fages, et T. Sola. A metalevel compiler of clp(fd) and its combination with intelligent backtracking. In F. Benhamou et A. Colmerauer, éditeurs, *Constraint Logic Programming Selected Research*, pages 437–456. MIT Press, 1993.

[4] R. Dechter et A. Dechter. Belief maintenance in dynamic constraint networks. In *AAAI*, pages 37–42, St Paul, Minnesota, USA, August 1988.

[5] F. Fages, J. Fowler, et T. Sola. Handling preferences in constraint logic programming with relational optimization. In M. Hermenegildo et J. Penjam, éditeurs, *6th International Symposium, PLILP'94*, pages 261–276, Madrid, Spain, September 1994. LNCS 844, Springer-Verlag.

[6] F. Fages, J. Fowler, et T. Sola. A reactive constraint logic programming scheme. In *12th International Conference on Logic Programming*, Tokyo, Japan, june 1995. MIT Press.

[7] J. Fowler. *à paraître : Extension Réactive de la Programmation Logique avec Contraintes et Applications en Optimisation Combinatoire*. Thèse de doctorat, Paris XI, 1995.

[8] J. Jaffar et M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19-20 : 503–581, May-July 1994.

[9] J. Jourdan. *Concurrence et Coopération des modèles multiples dans les langages de contraintes CLP et CC: vers une méthodologie de programmation par modélisation.* Thèse de doctorat, Paris VII, 1995.

[10] M. J. Maher et P. J. Stuckey. Expanding Query Power in Constraint Logic Programming Languages. In Ewing L. Lusk et Ross A. Overbeek, éditeurs, *Proceedings of the North American Conference on Logic Programming*, pages 20–36. MIT Press, Cleveland, Ohio, USA, 1989.

[11] T. Sola. *en préparation : Modèles d'exécution de la programmation logique basés sur le graphe de dépendances des liaisons : Contraintes, Backtracking Intelligent et Maintien des déductions*. Thèse de doctorat, Université de Paris XI, 1995.

[12] P. Van Hentenryck. Incremental constraint satisfaction in logic programming. In D. H. D. Warren et P. Szeredi, éditeurs, *Proceedings of the Seventh International Conference on Logic Programming*, pages 189–202. MIT Press, Jerusalem, 1990.

[13] G. Verfaillie et T. Schiex. Solution reuse in dynamic CSPS. In *Proceedings AAAI 1994*, pages 307–312, Seattle, WA, 1994.