Programmation Logique avec Contraintes et Ordonnancement

Patrick ESQUIROL* — Pierre LOPEZ

Laboratoire d'Analyse et d'Architecture des Systèmes du C.N.R.S.
7, avenue du Colonel Roche
31077 Toulouse Cedex

** I.N.S.A. de Toulouse*
Complexe Scientifique de Rangueil
31077 Toulouse Cedex

RÉSUMÉ. *Les problèmes combinatoires tels que les problèmes d'ordonnancement ont fait l'objet de nombreux travaux en Recherche Opérationnelle, et ont donné lieu à l'utilisation de méthodes de recherche arborescente, dont l'efficacité dépend notamment de l'exploitation "intelligente" des contraintes définissant les solutions admissibles. Malgré la puissance et l'élégance qu'offre la programmation logique au niveau de la représentation, les langages impératifs lui ont souvent été préférés pour des raisons d'efficacité. A l'heure actuelle, on constate un intérêt grandissant pour la programmation logique avec contraintes, dotant la programmation logique de mécanismes puissants d'interprétation de contraintes arithmétiques et symboliques. Après avoir rappelé les fondements théoriques de ces mécanismes, dont certains sont issus de travaux en Intelligence Artificielle, nous essayons de cerner les avantages réels de l'application de cette nouvelle technologie aux problèmes d'ordonnancement.*

ABSTRACT. *Combinatorial problems such as scheduling have been the target of many works in the field of Operations Research, and led to develop tree-search methods, whose efficiency strongly depends on an "intelligent" processing of the constraints that define the feasibility of the solutions. Although logic programming is a powerfull and elegant support for the representation of such problems, classic imperative languages have been preferred for efficiency considerations. Today constraint logic programming seems more attractive since it extends logic programming with efficient mechanisms, managing symbolic and numeric constraints. Once presented the theoretical bases of such mechanisms, this paper attempts to delimit the real advantages of applying this new technology for the solving of scheduling problems.*

MOTS-CLÉS : *problèmes de satisfaction de contraintes, programmation logique avec contraintes, ordonnancement.*
KEY-WORDS : *Constraint Satisfaction Problems, Constraint Logic Programming, Scheduling.*

1. Introduction

Un grand nombre de problèmes pratiques abordés d'abord en Recherche Opérationnelle puis en Intelligence Artificielle [LAU 76, DIN 90], (*e.g.*, ordonnancement, affectation de ressources, découpe de surfaces, vérification de circuits logiques...) sont des problèmes combinatoires dont la résolution revient à explorer un espace de recherche discret pour trouver un point satisfaisant un ensemble de contraintes. Pour la plupart de ces problèmes, il n'existe pas d'algorithme à la fois général et efficace pour les résoudre. Il sont donc souvent abordés à l'aide de techniques de recherche arborescente qui reposent sur une *stratégie par tentatives et retour arrière*.

La principale difficulté provient de la sémantique des contraintes. Celles-ci expriment des connaissances de *nature déclarative* énonçant davantage des *relations* que des dépendances fonctionnelles, ce qui ne facilite pas la conception d'algorithmes en langage impératif. L'instanciation des variables intervenant dans une contrainte n'obéit donc pas à un ordre connu *a priori*. De plus, les contraintes ne peuvent être considérées indépendamment, de manière séquentielle de par le fort couplage entre contraintes à travers les variables qu'elles partagent. Enfin la nature *disjonctive* de certaines, notamment en ordonnancement [GOT 93], engendre des discontinuités de l'espace des solutions rendant difficile sa formulation synthétique.

L'énumération exhaustive des valeurs permet de tester toutes les contraintes, mais ceci n'est envisageable que pour des problèmes de très petite taille et dans le cas de domaines finis. Face à cette combinatoire, la "décomposabilité" d'un problème peut être une propriété à rechercher car elle permet de résoudre un problème de grande taille à travers une suite de résolutions de problèmes de taille réduite [LEV 94]. En outre, il peut exister plusieurs solutions admissibles qui ne sont généralement pas totalement équivalentes. Les critères utilisés dans les approches d'*optimisation* jouent alors un rôle de guide pour un choix ordonné des variables et/ou de leurs valeurs.

On peut aussi avoir recours à des méthodes plus expérimentales, utilisant des connaissances spécifiques au domaine considéré. On trouve souvent des stratégies très efficaces dans les *méthodes heuristiques* [CAR 93], mais cette spécificité rend les modèles peu réutilisables, notamment dans la perspective de la conception d'un langage de programmation.

Sur le plan pratique, on trouve deux classes d'outils. Les outils généraux, *e.g.*, la programmation linéaire en nombres entiers, proposent une énonciation standard, moyennant une réécriture qui a tendance à augmenter substantiellement la taille de l'espace de recherche (augmentation du nombre de variables et/ou du nombre de contraintes...) et qui abandonne certaines caractéristiques importantes du domaine (existence d'heuristiques, de symétries...). Les outils dédiés, écrits à l'aide de langages procéduraux, procurent une efficacité réelle mais posent essentiellement le problème de leur réutilisabilité.

Il existe donc un besoin réel mais difficile à satisfaire : disposer d'un langage suffisamment riche pour énoncer sans les déformer une classe large de problèmes combinatoires, et suffisamment ouvert de manière à y intégrer des connaissances spécifiques pouvant améliorer considérablement l'efficacité.

2. La programmation logique standard (PL)

En PL [CLO 81, COL 83, ACM 92], l'exécution d'un programme correspond au déroulement d'un raisonnement logique contrôlé par un démonstrateur automatique. En PL, contrairement à la programmation impérative, une distinction est faite entre représentation (le programme) et traitement (l'exécution).

Le programmeur modélise le problème à l'aide d'un ensemble de relations de *la logique des prédicats du premier ordre*, puis construit un ensemble d'énoncés sous la forme de *clauses de Horn*. Dans le cas d'un graphe orienté, une relation *arc(x,y)* symbolise l'arc orienté qui lie le sommet x au sommet y et une relation *chemin(x,y,c)* permet de relier deux nœuds x (origine) et y (extrémité) par un chemin c (liste des nœuds traversés pour aller de x en y, x et y compris). La définition récursive d'un chemin donne lieu à deux clauses :

```
chemin(x,y,[x,y]) :- arc(x,y).
chemin(x,z,[x|L]) :- arc(x,y), chemin(y,z,L).
```

Le même programme peut servir à : (1) déterminer tous les chemins entre deux nœuds donnés ; (2) construire tous les chemins partant d'un nœud donné ou (3) arrivant en un nœud donné ; (4) générer l'ensemble des chemins du graphe ; (5) donner tous les chemins de longueur donnée passant par un nœud particulier. On qualifie de *réversibles* les programmes définissant des relations telles que la relation chemin(x,y,c), pour lesquels le rôle d'entrée/sortie des arguments n'est pas figé.

Pour répondre à une question donnée, le langage applique le *principe de résolution* (PR) [ROB 65] qui permet, à travers une démonstration par réfutation, d'obtenir les valeurs des variables éventuellement présentes dans la question posée. L'exécution d'un programme logique correspond à l'exploration d'un arbre de dérivation dont la racine constitue la négation de la clause à démontrer et dont les feuilles sont des clauses vides (*cf.* figure 1). Chaque branche est régie par un algorithme d'*unification* qui génère l'ensemble minimal des substitutions nécessaires à l'application du PR. Le choix des clauses devant être mises en relation lors des différentes inférences est contrôlé par une *stratégie d'exploration en profondeur d'abord avec retour arrière chronologique*. Par défaut, tous les points de choix sont mémorisés ainsi que leur contexte (listes des variables locales et des substitutions) pour assurer l'exhaustivité de la recherche.

Pour aborder la résolution de problèmes avec contraintes, la PL est un outil intéressant.

Le caractère *non-déterministe* de la PL et sa *sémantique déclarative* permettent de séparer la représentation d'un problème de sa résolution. De plus lorsqu'ils sont réversibles, les programmes gagnent en généralité et en réutilisabilité. La proximité conceptuelle de la stratégie de résolution et des techniques de recherche arborescente dans les problèmes combinatoires facilitent également la conception des programmes.

Toutefois, ces caractéristiques générales ne sauraient reléguer au second plan le souci de l'efficacité des programmes. Dans la suite, nous expliquons pourquoi il est nécessaire d'étendre la PL lorsqu'on envisage de traiter des problèmes dont la

résolution est soumise à un grand nombre de contraintes numériques et symboliques.

3. La programmation logique avec contraintes (PLC)

Lors d'une exploration arborescente, le nombre d'échecs et la profondeur à laquelle ils sont détectés déterminent fortement l'efficacité globale.

A titre d'exemple, le programme et l'arbre ci-dessous correspondent au problème de la recherche de tous les couples de chiffres (x,y) parmi {1,2,3} tels que x<y. Les branches menant aux solutions figurent en gras sur l'arbre.

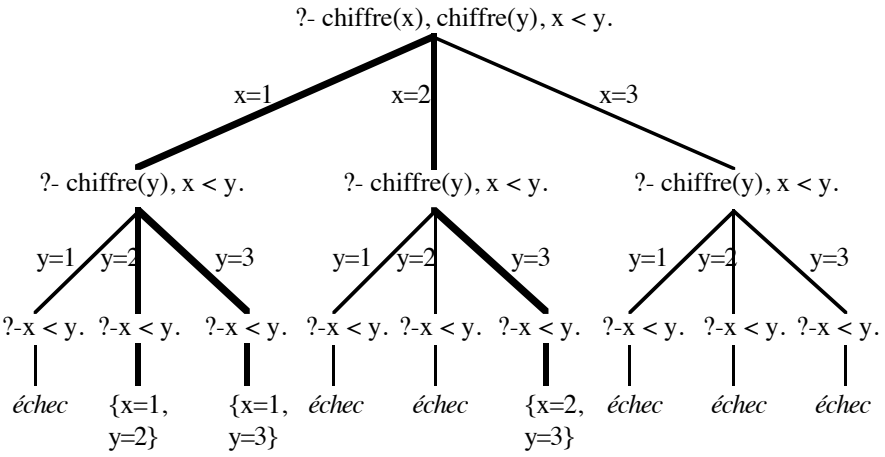

Figure 1. *Arbre de dérivation Prolog*

Sur 9 branches terminales, l'arbre de recherche comporte 6 échecs, alors que ceux-ci auraient pu être anticipés, évitant ainsi une exploration inutile. En effet, les domaines initiaux respectifs de x et de y comportent certaines valeurs incompatibles avec l'inégalité stricte x<y, par exemple les valeurs x=3 ou y=1.

La mise en place d'un tel raisonnement au sein même du langage (et non par programmation) implique qu'il puisse ***implicitement*** *exploiter la contrainte x<y avant toute instanciation des variables*. Le problème vient donc du fait que la relation "x<y" n'est pas interprétée comme une contrainte numérique mais comme un simple test sur des variables nécessairement instanciées au préalable. Les tests numériques ne sont d'ailleurs pas des primitives logiques et leur existence a permis de ne pas cantonner Prolog au seul calcul symbolique mais d'en faire un véritable langage de programmation.

Si la réduction initiale des domaines présente un caractère utile et nécessaire, elle n'est pas suffisante : une contrainte peut ne devenir active qu'à partir d'une certaine étape d'instanciation ; c'est le cas de la branche x=2. Dès que x est fixé à 2, la contrainte x<y peut réduire le domaine de y à la valeur 3, ce qui permet d'éviter l'exploration inutile des branches y=1 et y=2.

Ce type d'inférence n'est pas produit en PL car il exigerait de faire la différence entre les variables non typées (prenant comme valeur un terme) et les variables numériques (entières, rationnelles ou réelles). Le manque d'efficacité de la PL standard pour la résolution des problèmes comportant des contraintes numériques provient donc d'une *utilisation passive des contraintes* : les valeurs incompatibles avec les contraintes ne sont éliminées que lors du retour arrière consécutif à un échec (*génère-et-teste*), et non *a priori* (*teste-et-génère*).

Dans la suite, nous nous intéressons aux mécanismes d'inférence qui ont été intégrés à la PL pour aboutir à la PLC. Nous présenterons un certain nombre de définitions et de résultats théoriques importants obtenus dans le domaine des *problèmes de satisfaction de contraintes* (PSC) et leur intégration dans la PLC à travers le traitement des contraintes sur les variables à *domaines finis*. Des résultats spécifiques sont également utilisables pour le traitement des contraintes sur les booléens, pseudo-booléens, rationnels, réels, listes, arbres ou chaînes de caractères, que nous ne pourrons que résumer dans cet article. Nous renvoyons le lecteur à [BEN 93, JAF 94] pour un tour d'horizon des recherches actuelles en PLC et à [TSA 93] pour une synthèse détaillée des PSC.

3.1. *Les PSC sur les domaines finis*

Un PSC peut être défini [WAL 72, MON 74, MAC 77] par un 3-uplet (V,D,C) tel que :

• V est un ensemble de *variables* ; $V = \{V_1, V_2,..., V_n\}$;
• D est un ensemble de *domaines finis* ; $D = \{D_1, D_2,..., D_n\}$ où chaque domaine D_i est l'ensemble des valeurs de V_i ;
• C est un ensemble conjonctif de *contraintes* ; $C = \{C_1, C_2,..., C_m\}$ où chaque contrainte C_i est définie par :

- un sous-ensemble de variables $var(C_i) = \{V_{i_1}, V_{i_2},..., V_{i_{n_i}}\}$;
- une relation $rel(C_i) = rel(V_{i_1}, V_{i_2},..., V_{i_{n_i}}) \subseteq D_{i_1} \times D_{i_2} \times ... \times D_{i_{n_i}}$.

Une solution est un n-uplet $\{v_1, v_2,..., v_n\}$ de valeurs (une par variable) tel que pour chaque contrainte C_k , les valeurs associées aux variables $var(C_k)$ assurent le respect de C_k.

Sur ce modèle, on peut décliner plusieurs types de problèmes, de complexité équivalente (NP-complets [GAR 79]), le problème de la satisfaction d'un ensemble de contraintes sur des variables à domaine fini pouvant se ramener à celui de la coloration d'un graphe :

• démontrer l'existence de solutions, trouver une (toutes les) solution(s) ;
• démontrer qu'une instanciation partielle des variables caractérise au moins une (toutes les) solution(s) ;
• déterminer toutes les valeurs possibles d'une variable sur l'ensemble des solutions ;
• trouver une (toutes les) solution(s) optimale(s).

3.2. *Notion de consistance dans les PSC et propagation de contraintes*

La *consistance* [FRE 78, FRE 82, DEC 92, TSA 93] est une propriété établie par comparaison entre les valeurs d'une ou plusieurs variables et les valeurs autorisées par les contraintes. Le retrait (filtrage) de valeurs ou de n-uplets de valeurs inconsistants constitue un *renforcement de cohérence* ou *propagation de contraintes*. Un filtrage complet permet en théorie d'obtenir une représentation explicite de l'ensemble des solutions. L'absence de solution, ou inconsistance globale, est détectée à l'issue d'un filtrage, s'il existe une variable V_i telle que $D_i = \varnothing$.

Les algorithmes de filtrage [NAD 89] ont pour objet de transformer un CSP en un nouveau CSP' tel qu'un certain type de consistance soit vérifié. Ils se distinguent selon l'arité des contraintes considérées :

- contraintes unaires

Un CSP est *domaine-consistant* si le domaine de chaque variable est cohérent avec l'ensemble des contraintes unaires qui pèsent sur elle.

- contraintes binaires

• Un CSP est *arc-consistant* si le domaine de chaque variable est cohérent avec l'ensemble des contraintes binaires qui pèsent sur elle. L'algorithme AC4 proposé dans [MOH 86] est de complexité $O(m \times d^2)$ avec m = nombre de contraintes et d = taille maximale des domaines.

• Un CSP est *chemin-consistant* si tout couple de valeurs autorisé par une contrainte liant 2 variables l'est aussi par tous les chemins de contraintes liant ces variables. L'algorithme PC4 est en $O(n^3d^3)$ avec n = nombre de variables [HAN 88].

La chemin-consistance est une condition plus forte que l'arc-consistance mais ne constitue pas une condition suffisante de consistance globale, comme le montre le problème de coloration de graphes de la figure 2, où il existe bien des triplets de valeurs consistants, alors que le problème de coloration d'un graphe complet de 4 sommets en 3 couleurs n'admet aucune solution.

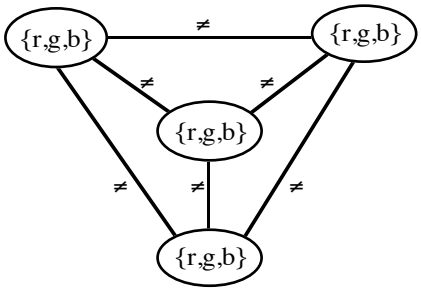

Figure 2. *Un PSC arc-consistant mais non chemin-consistant*

- contraintes n-aires

Un CSP est *k-consistant* si toute instanciation localement consistante de k-1 variables, peut être étendue à toute instanciation localement consistante de k variables.

En l'absence d'hypothèses particulières (*e.g.*, contraintes numériques binaires conjonctives), il n'existe pas d'algorithme polynomial pour vérifier la consistance globale d'un réseau de contraintes quelconques. Il n'est donc pas possible d'implanter un mécanisme général efficace dans un langage de PLC.

3.3. *Les algorithmes d'instanciation progressive et de propagation*

Lorsqu'aucune déduction ne permet de restreindre un domaine, la stratégie d'instanciation consiste en une simple énumération des valeurs des variables, celles-ci étant considérées dans un ordre arbitraire. L'application des règles de filtrage données ci-dessous doit être envisagée dans un contexte dynamique, à partir d'un état courant où certaines variables du problème ont déjà été instanciées, d'autres pas.

Le *forward checking* (FC) est une technique de renforcement de cohérence par domaine-consistance. L'idée est d'exploiter plus efficacement les contraintes, en particulier lorsqu'en cours de génération de solution, il ne subsiste dans une contrainte qu'une seule variable non instanciée : toute valeur du domaine non consistante avec les valeurs des variables déjà instanciées est alors retirée du domaine.

Deux cas particuliers sont importants à l'issue d'un filtrage de ce type :

- si le domaine devient vide, la résolution nécessite un retour arrière ;
- si le domaine se réduit à une seule valeur, la variable prend automatiquement cette valeur (variable *figée*).

Le *look ahead* (LA) est à la k-consistance ce que le FC est à la domaine-consistance. L'idée est de vérifier que toute valeur du domaine d'une variable encore libre demeure consistante avec au moins une instanciation possible des variables encore libres. Toute valeur telle qu'il est impossible de trouver une instanciation consistante des variables encore libres, est écartée du domaine. Comme précédemment, les mêmes actions sont réalisées dans le cas d'un domaine devenant vide ou se réduisant à une seule valeur.

L'intégration des mécanismes type FC ou LA peut améliorer considérablement l'efficacité globale d'une résolution par exploration et retour arrière chronologique, en limitant *le nombre d'échecs* et la *profondeur* à laquelle ils sont découverts.

Cependant, un compromis doit être trouvé entre le gain obtenu par la diminution de l'espace de recherche et l'effort (en temps et en mémoire) consenti à l'analyse. Par exemple l'utilisation systématique du LA, meilleur que le FC en ce qui concerne l'élagage de l'espace de recherche, n'est pas réaliste ; au contraire, elle doit être contrôlée en évaluant la complexité à partir du nombre moyen de variables mises en jeu, la taille moyenne des domaines et l'interdépendance des contraintes.

3.4. *Les contraintes booléennes*

De nombreux problèmes combinatoires donnent lieu à des modèles à variables booléennes : analyse/synthèse de circuits logiques, problèmes de séquencement,

d'affectation ... Si le problème général de la résolution d'équations booléennes est lui aussi NP-complet, il existe néanmoins différentes techniques de réécriture et d'interprétation, exploitables pour une meilleure gestion des contraintes booléennes en PLC.

Le traitement de contraintes booléennes pose d'abord le problème de l'unification d'expressions booléennes [RUD 74, COL 86, DIN 87]. Il faut définir le langage des expressions (constantes, variables, opérateurs) et l'interprétation de l'égalité entre 2 expressions booléennes.

En PrologIII, l'énoncé de contraintes booléennes est défini par un langage utilisant l'ensemble des opérateurs classiques. Les réponses sont fournies sous la forme d'un système d'équations canonique, dont la forme peut dépendre de l'ordre dans lequel les contraintes ont été considérées. La stratégie de résolution est la *SL-résolution*, saine et complète.

Le langage CHIP est basé sur une stratégie différente : la réécriture des expressions d'une algèbre de Boole dans l'anneau booléen muni des opérateurs logiques *ou exclusif* et *et*. L'intérêt est d'assurer l'existence d'un plus grand unificateur unique, dont la détermination peut être réalisée par un algorithme d'unification booléenne efficace [DIN 87, BUT 87].

Contrairement à PrologIII, CHIP ne retourne pas un système d'équations booléennes lorsque le problème admet plusieurs solutions. Par contre, il donne bien toutes les variables figées par la prise en compte des contraintes. Nous donnerons au §5.2 un exemple d'application en ordonnancement.

3.5. *Les contraintes sur les rationnels*

Pour la satisfaction d'un ensemble de contraintes linéaires sur des variables continues, l'algorithme repris en PLC est celui du *Simplexe*. Initialement orienté sur la recherche d'une solution optimale, cet algorithme a subi certaines adaptations autorisant le caractère incrémental de la résolution et la caractérisation *symbolique* d'un ensemble de solutions.

La résolution donne lieu à deux types de réponses, soit l'échec, lorsque le système de contraintes n'est pas satisfiable, soit une représentation symbolique de l'ensemble des solutions (*e.g.*, {y = 6x - 4z + 8, z >= 0}). Dans le cas d'une solution unique (variables figées), la solution est une liste d'équations du type {Variable1= valeur1, Variable2= valeur2, ...}.

PrologIII et CHIP utilisent une arithmétique sur les *rationnels*, qui peuvent être représentés de manière exacte en machine (précision infinie), alors que la représentation des réels en nombres flottants entraîne des erreurs d'arrondi.

CLP(*R*) utilise quant à lui une arithmétique sur les *réels* en notation flottante. Un ensemble très complet de primitives permettant de poser des contraintes non linéaires (*e.g.*, Y = X*Z, Y= log(X)), de constantes et de fonctions mathématiques a permis à ce langage de se distinguer sur des problèmes du type analyse/synthèse de circuits électriques analogiques, dans lesquels interviennent effectivement de nombreuses contraintes linéaires et non linéaires. Il présente d'autre part des fonctionnalités de "méta-programmation" ; il permet en particulier de traiter

symboliquement les termes et les contraintes arithmétiques grâce à une forme codée, traitement qu'il n'est pas possible de réaliser élégamment en CHIP ou PrologIII.

4. Les langages de PLC

Pour effectuer un bilan sur l'ensemble des possibilités offertes par ces langages, nous nous restreindrons aux trois langages qui nous semblent les plus représentatifs de la PLC : CHIP [DIN 88], PrologIII [COL 90] et CLP(*R*) [JAF 87].

4.1. *Le fonctionnement de base des machines Prolog avec contraintes*

L'état de la machine abstraite PrologIII peut être représenté par un triplet (V,B,C) où :

- V est l'ensemble des variables auxquelles on s'intéresse ;
- B est la liste des buts à résoudre ; $B = b_0\ b_1 ... b_n$;
- C est le système de contraintes courant.

L'utilisation d'une règle du type :

$s_0 \text{ -> } s_1\ s_2 \ldots s_m, c$

où c désigne les contraintes attachées à l'utilisation de la règle, engendre le nouvel état (V,B',C') tel que :

$B' = s_1\ s_2 \ldots s_m\ b_1 \ldots b_n$ et $C' = C \cup c \cup \{b_0 = s_0\}$.

Les contraintes propres à une clause sont placées à la suite des littéraux logiques. Le langage se charge de l'ordre dans lequel elles doivent être interprétées compte tenu des contraintes d'unification. En CHIP et en CLP(*R*), au contraire, les contraintes ne sont pas différenciées des littéraux classiques.

4.2. *Génération de solutions*

4.2.1. *Schéma général de résolution*

La résolution d'un problème dont les variables sont soumises à des contraintes obéit au schéma suivant :

1. Création des variables du problème et/ou pose des contraintes de domaine.
2. Pose des contraintes binaires ou n-aires.
3. Instanciation des variables.
4. Arrêt sur la première solution ou retour sur les points de choix laissés en 3 (éventuellement en 2 dans le cas de points de choix sur la pose de contraintes).

L'ordre des étapes 2 et 3 est primordial dans l'optique d'une utilisation *active* des contraintes. Bien que l'instanciation des variables soit non-déterministe, on peut

néanmoins agir sur l'ordre d'instanciation à deux niveaux : (1) stratégie de choix des variables à instancier ; (2) stratégie de choix des valeurs pour une variable donnée.

Les primitives de choix des variables permettent d'appliquer le *first-fail-principle* [HAR 80]. Ce principe dit qu'il vaut mieux échouer le plus tôt possible dans l'exploration de l'espace de recherche. En pratique, il faut sélectionner en priorité les variables ayant le plus petit domaine et présentes dans le plus grand nombre de contraintes. Cette sélection peut être assurée par le langage lui-même moyennant des primitives adéquates.

Exemple (CHIP) : boucle d'instanciation d'une liste de variables à domaine fini.

```
labeling1([]).
labeling1([X|Y]) :-
        delete(Var, [X|Y], Rest, 0, most_constrained),
        indomain(Var),
        labeling1(Rest).
```

Le prédicat “indomain(X)” de CHIP est un générateur de valeurs qui évite au programmeur de spécifier explicitement le domaine de la variable, domaine qu'il n'est pas certain de connaître au moment de l'instanciation, étant donné les restrictions qu'ont pu opérer les contraintes. De même, le prédicat “enum(X)” de PrologIII énumère toutes les constantes entières n telles que le système courant de contraintes augmenté de la contrainte X=n soit globalement satisfait.

4.2.2. *Optimisation*

Les langages de PLC incluent des primitives permettant de trouver une solution optimale à un problème de satisfaction de contraintes.

Dans le cas de problèmes à variables rationnelles (ou réelles), l'algorithme du Simplexe permet de répondre au problème d'optimisation (à la condition que le critère soit linéaire) grâce aux primitives du type “minimize(X)” (PrologIII) ou “rmin(X)” (CHIP).

Pour les variables à domaines finis, la primitive “min_max” de CHIP permet de rechercher toutes les solutions et conserve celle de plus petit coût. Lorsqu'il existe un très grand nombre de solutions, la recherche peut être réduite selon quatre critères dont le “réglage” ne peut être qu'expérimental : (1) une limite de temps machine, (2) une borne supérieure et (3) inférieure du coût, (4) un pourcentage d'amélioration.

4.2.3. *Résolution incrémentale*

Afin de conserver la flexibilité et la souplesse de la PL, on ne peut faire l'hypothèse que toutes les contraintes sont connues à l'avance et posées en bloc. Au contraire, la possibilité de déclarer les contraintes de manière modulaire, de les intégrer progressivement, voire de les retirer (grâce à la pose de points de choix), permet de mettre au point des stratégies de résolution plus évoluées.

Exemple. Le programme partiel suivant (en CHIP) correspond à une stratégie de résolution d'un problème dans lequel il existe 2 niveaux de contraintes C1, C2, et 2 critères différents f1, f2. On désire utiliser le critère f2 si l'ensemble C1∪C2 est satisfiable, et le critère f1 si seulement C1 l'est.

```
resoudre(V) :-
        creer_variables(V),
        poser_C1(V),
        suite_resoudre(V).

suite_resoudre(V) :- poser_C2(V), !, minimiser_f2(V).
suite_resoudre(V) :- minimiser_f1(V).
```

Le caractère incrémental de la résolution et la mémorisation automatique du contexte résultant de la prise en compte de C1 évite une reprise de la résolution depuis le début lorsque C2 n'est pas satisfiable.

On peut imaginer d'autres types de stratégies, comme par exemple la pose conditionnelle de contraintes de plus en plus restrictives à partir d'une observation de l'effet de ces contraintes sur le domaine des variables, au fur et à mesure de leur pose. De telles stratégies sont intéressantes lorsque les contraintes peuvent être hiérarchisées. En effet, la pose de contraintes s'arrête dès que l'ensemble courant n'est plus satisfiable, et le système restaure le dernier état précédant l'échec. La solution représente alors un état dans lequel les contraintes les plus "importantes" ont été prises en compte, les contraintes "responsables" de l'échec, moins importantes, ont le plus de raisons d'être remises en cause, notamment dans l'optique d'une *résolution interactive* des problèmes [ESQ 93a, HUG 94].

4.3. *Types de contraintes gérées par les langages de PLC*

Les types de contraintes sont résumés dans le tableau suivant.

	Arbres	Chaînes	Domaines Finis	Booléens	Rationnels	Flottants	Entiers
PrologIII	+++ (infinis)	+	-	+++	+++	+	+
CHIP	+ (finis)	+	+++	+++	+++	+	+++
CLP(*R*)	+ (finis)	+	-	-	-	+++	+

+++ : gestion active des contraintes sur ces types de variables
\+ : types de variables autorisés
\- : types de variables non autorisés

Figure 3. *Contraintes couvertes par les langages de PLC*

Le langage CHIP apparaît comme celui qui couvre le plus de types de contraintes. Il est capable de propager activement des contraintes sur des variables à

domaine fini (notamment les entiers), avec la possibilité de gérer des ensembles discontinus (*e.g.*, X :: [1,2,3, 7,8,9]). Cette propagation est néanmoins incomplète pour des contraintes n-aires ($n \geq 3$) du fait du caractère local du FC (*cf.* §3.3). Dans le cas de contraintes linéaires, la complétude passe par une modélisation en variables rationnelles (dans ce domaine CHIP et PrologIII sont équivalents).

Mais ce jugement doit être relativisé. En effet, la modélisation d'un problème concret fait souvent intervenir des contraintes liées à des variables de types différents (modèles à variables mixtes). S'il existe plusieurs paradigmes de résolution de contraintes au sein d'un même langage, la propagation de contraintes entre différents modèles est relativement limitée. Certaines faiblesses actuelles seront données en conclusion (*cf.* §4.5).

4.4. *Retardement de contraintes*

Toutes les contraintes n'ont pas un effet immédiat dès leur pose ; certaines nécessitent une instanciation partielle des variables pour devenir actives. Pour pouvoir poser des contraintes sans qu'elles soient immédiatement évaluées, on peut utiliser un mécanisme de retardement de buts. Ceci correspond à une *programmation dirigée par les données* et ne remet pas en cause pour autant le schéma général présenté au §4.2.1. Les littéraux retardés sont réveillés et placés dans la liste des buts courants dès que les conditions d'interprétation sont réunies.

Exemple 1 (CLP(*R*)) :

• Z = X*Y est retardée jusqu'à ce que l'une au moins des 2 variables à droite de l'égalité soit instanciée.

• Z = pow(X,Y) est retardée jusqu'à ce que :
  X et Y soient instanciées (=> Z est évaluable), ou
  X et Z soient instanciées (=> Y est évaluable), ou
  X = 1 (=> Z = 1), ou Y = 0 (=> Z = 1), ou Y = 1 (=> Z = X).

Le principe est mis en œuvre par un mécanisme de *co-routining* [DIN 88, HEN 89]. De ce fait, il peut y avoir une incohérence dans le système de contraintes à une étape donnée, celle-ci n'étant détectée que lors du réveil des contraintes retardées, ce qui présente un inconvénient certain sur le plan de l'efficacité (utilisation passive des contraintes).

L'exemple suivant montre que le retardement de contraintes est un mécanisme délicat, et qu'il est très important de maîtriser les spécificités du langage dans ce domaine.

Exemple 2 (PrologIII) :

```
>{U::N, 0<N<1};
```

La définition incohérente d'un arbre U de taille N comprise entre 0 et 1 (!) n'est pas détectée car la variable N est libre au moment de la pose de ces contraintes.

Il existe deux modes de retardement de contraintes, implicite et explicite. Le premier correspond au cas où le langage prend en charge le retardement, après avoir détecté l'impossibilité d'exploiter immédiatement la contrainte (*cf.* exemple 1). Le deuxième met en œuvre des primitives dédiées comme :

• freeze(x, but(A1,A2,...,An)) (PrologIII)
but(...) n'est évalué que lorsque x est instancié.

• delay but(A1,A2,...,An) (CHIP)
but(...) n'est évalué que lorsque les arguments (A1, A2,..., An) vérifient certaines conditions d'instanciation (complète ou partielle).

• touched(but,X,Info,Type_ev) (CHIP)
L'apparition d'un événement modifiant le domaine fini de X (augmentation/diminution de la borne minimum/maximum, retrait d'une valeur interne) engendre l'exécution immédiate de but(X,Info).

• une primitive de propagation conditionnelle (CHIP) :

if <condition> then <but1> else <but2>

dont le fonctionnement est le suivant :

- si <condition> est satisfiable pour toutes les valeurs des variables, alors <but1> est posé ;
- si <condition> n'est pas satisfiable pour toutes les valeurs des variables, alors <but2> est posé ;
- sinon <condition> est systématiquement réévalué à chaque modification du domaine des variables impliquées dans <condition>.

4.5. *Conclusion*

Les langages de PLC ne permettent pas à l'heure actuelle de modéliser directement un problème faisant intervenir des contraintes mélangeant plusieurs types de variables, ce qui crée un certain cloisonnement entre les différents systèmes de traitement de contraintes. Par exemple, une expression hétérogène comme "B=(x<y)", qui associe un booléen B à l'évaluation d'une inégalité arithmétique, n'est pas autorisée. En règle générale, l'intégration de telles expressions oblige à programmer explicitement des relations de passage entre les différents systèmes de variables et de contraintes. Mais les contraintes qu'elles représentent sont alors utilisées de manière passive. Signalons des travaux récents [BOC 93] sur le traitement de *contraintes pseudo-booléennes*.

D'autre part, la relation qui lie un rationnel à son entier immédiatement inférieur ne peut pas être une contrainte. Une telle contrainte permettrait de propager implicitement toute actualisation des bornes d'un domaine d'une variable sur le domaine de l'autre. Pour résoudre ce problème, le co-routining n'est même pas utilisable ; en CHIP, on peut en effet propager l'actualisation du domaine d'un entier vers un rationnel ("touched/4"), mais pas le contraire.

On ne peut donc pas faire de propagation de contraintes d'un système de contraintes à un autre, ce qui diminue les possibilités d'une représentation naturelle.

Ainsi, pour un problème donné, il est préférable de s'orienter vers le langage le plus adapté au domaine de variables du problème considéré, même s'il couvre moins de domaines que le langage concurrent.

5. Application aux problèmes d'ordonnancement

L'organisation du travail dans un système de production ou dans un grand projet nécessite une décomposition du travail relativement aux ressources disponibles pour son exécution. Le produit de cette décomposition conduit à définir un ensemble de tâches, chaque tâche représentant une unité de travail élémentaire, caractérisée par une durée et un ensemble de ressources nécessaires à son exécution. Le problème d'ordonnancement naît de cette décomposition : étant donné un ensemble de tâches élémentaires et un ensemble de ressources dont la disponibilité est limitée, il est nécessaire d'organiser l'exécution des tâches dans le temps en respectant des contraintes variées telles que :

• les contraintes de cohérence technologique (gammes, contraintes logiques d'enchaînement...) ;

• les contraintes de ressources : elles expriment une disponibilité limitée des ressources, en nombre, en intensité ou en *énergie* [LOP 92] ;

• les contraintes de localisation temporelle : elles résultent de l'existence d'objectifs globaux de réalisation, tels que des dates limites de début au plus tôt ou de fin au plus tard, ou tels qu'une durée totale limitée.

Compte tenu des différents paradigmes de résolution utilisés dans les langages de PLC, nous nous sommes limités à deux grandes classes de problèmes pour lesquels une approche par la PLC peut apporter une aide :

- les problèmes pour lesquels il s'agit de déterminer des *dates d'exécution*, généralement les dates de début ;

- les problèmes pour lesquels il s'agit de trouver un *séquencement* des tâches sur chaque ressource.

Remarque. Un autre classe de problème concerne la *planification de quantités* à produire sur un horizon discrétisé en périodes de durée connue, en respectant les délais et quantités commandées, et la disponibilité limitée des ressources sur chaque période. Une modélisation par flots est souvent proposée pour résoudre ce type de problèmes et les équations de conservation de la matière sont linéaires [THI 93]. D'autres exemples d'application de la PLC à certains problèmes de gestion du temps et des ressources peuvent être avantageusement consultés dans [BAP 92, BOI 95, LEG 92, WAL 94].

Glossaire.

	S_i	date de début de i
	F_i	date de fin de i
	D_i	durée de i
Pour une tâche i :	B_i	borne temporelle de i (S_i ou F_i)
	B_i^-	borne inférieure de B_i

B_i^+	borne supérieure de B_i
X_{ij}	variable booléenne (vraie si i précède j)

5.1. *Problèmes de dates*

C'est le cas des problèmes de gestion de projets, ou d'ordonnancement d'atelier. La décomposition du travail en tâches s'y exprime à travers des *contraintes de localisation temporelle relative* entre deux tâches.

D'autre part, la définition de dates limites d'exécution (*e.g.*, fin d'un projet, délais d'approvisionnement, de livraison...) impose des contraintes de localisation *temporelle absolue* des tâches.

5.1.1. *Contraintes de localisation temporelle relative et absolue*

La majorité des contraintes de localisation temporelle relative peut s'exprimer à travers une ou plusieurs inégalités entre des dates, du type : $B_j - B_i \geq a_{ij}$, où B_i et B_j représentent des bornes temporelles caractéristiques des tâches i et j (dates de début et/ou de fin), et a_{ij} une durée égale à la distance temporelle minimale (positive ou négative) entre ces bornes. Ces contraintes temporelles peuvent se représenter sous la forme d'un graphe *potentiels-tâches* ou *potentiels-étapes* [DIB 70, ROY 70, ELM 77, ELM 92] ou d'un graphe *potentiels-bornes* [ESQ 93b], permettant de prendre en compte des durées variables et des contraintes sur ces durées.

Les contraintes de localisation absolue peuvent également se traduire par des inégalités du même type à condition d'instancier une des deux bornes par un instant de référence :

$$B_i \geq B_i^- \quad \Leftrightarrow \quad B_i - 0 \geq B_i^-$$
$$B_i \leq B_i^+ \quad \Leftrightarrow \quad 0 - B_i \geq -B_i^+$$

Traditionnellement, l'application d'algorithmes de recherche des plus longs chemins entre deux sommets permet de caractériser de manière nécessaire et suffisante l'ensemble des solutions. Cette caractérisation est de complexité acceptable ($O(n^3)$ si n est le nombre de tâches) [PAP 82, DEC 91]. Elle est directement réalisable par un langage de PLC tel que CHIP, et l'exemple du traitement d'un problème PERT est souvent utilisé par les concepteurs [DIN 90]. Le squelette d'un programme CHIP permettant de calculer les fenêtres temporelles d'exécution d'un ensemble de tâches avec contraintes de localisation absolue ou relative des tâches est le suivant :

```
trouver_fenetres(Liste_des_dates_de_debut_Si) :-
        % pose des contraintes de localisation absolue
        Si :: Smin..Smax,
        ...
        % pose des contraintes de localisation relative
        Sj #>= Si + Di,
        ...
        % éventuellement minimisation de la date de fin de l'ensemble des tâches
        minval(Sprojet).

minval(S) :- domain_info(S, Min, _,_,_,_), S = Min.
```

Pour des problèmes dont les tâches ont des durées connues, le programme précédent permet de trouver les fenêtres temporelles associées aux dates de début des tâches sur l'ensemble des solutions admissibles.

En revanche, lorsque les durées ne sont pas connues, cette caractérisation est incomplète car les contraintes font intervenir 3 variables (début, durée, fin) et la consistance globale n'est plus assurée par les mécanismes de propagation sur les domaines finis. La seule alternative est d'adopter une modélisation en nombres rationnels.

L'exemple ci-dessous permet de modéliser un graphe de contraintes temporelles dont les sommets sont les dates de début et les dates de fin d'un ensemble de tâches, et dont les arcs correspondent à une contrainte de distance entre deux dates. Un tel graphe, muni d'un sommet origine des temps permet de déterminer de manière complète les valeurs minimales des distances entre tout couple de dates.

Exemple (PrologIII) :

/* Le graphe initial, déclaré par la clause graphe0(G0), est modélisé par une liste d'arcs du type : <origine, extrémité, valuation>. Chaque noeud (origine ou extrémité) est de l'un des trois types :

(1) <org, 0> ;
(2) <deb(i), x> où x est la variable date de début d'une tâche i ;
(3) <fin(i), y> où y est la variable date de fin de i.

On pose la contrainte : y-x ≥ d, d étant une constante positive ou négative.

Le graphe courant G reprend la même structure que G0 à la différence que chaque valuation est une variable dont on cherche la valeur minimale après avoir posé et propagé les contraintes de G0.

Le programme principal (top) consiste à lire le graphe initial, lancer la propagation, et retourner les valeurs minimales des arcs. */

```
top ->
        graphe0(G0)
        propa_graphe(G0, G)
        affiche(G).

graphe0([arc1, arc2, ...]) ->;

propa_graphe([], []) ->;
propa_graphe([<<n_i, i>, <n_j, j>, d0> | G0], [<<n_i, i>, <n_j, j>, d> | G]) ->
        propa_graphe(G0, G)
        minimum(j - i, d),
        {j - i >= d0};

affiche([]) ->;
affiche([A_rc | Liste_d_arcs]) ->
        outl(A_rc)
        affiche(Liste_d_arcs);
```

L'exemple suivant permet de poser les contraintes de gamme dans un problème de *job-shop*. Un ensemble de travaux doivent être exécutés sur un ensemble de

machines. Chaque travail est constitué d'un ensemble de tâches ordonnées et réalisées sur des machines différentes. On considère chaque travail successivement. Pour chaque tâche d'un travail, on crée une variable domaine de début et on pose les contraintes qui constituent la séquence de tâches au sein du travail.

Exemple (CHIP) :

```
travail([S1|S],[D1|D],Max) :-
        S1 :: 0..Max,
        pose_precede([S1|S],[D1|D],Max).

pose_precede([Sn],[Dn],Max) :-
        Sn + Dn #<= Max.
pose_precede([S1,S2|S],[D1,D2|D],Max) :-
        S2 :: 0..Max,
        S1 + D1 #<= S2,
        pose_precede([S2|S],[D2|D],Max).
```

5.1.2. *Contraintes de disjonction*

Les problèmes d'ordonnancement à ressources non partageables définissent les problèmes d'ordonnancement *disjonctifs*. Une modélisation par graphe non-conjonctif [ERS 79, BAR 88] est alors possible mais il n'existe pas d'algorithmes de complexité acceptable permettant comme dans le cas précédent de caractériser de manière nécessaire et suffisante l'ensemble des solutions. On ne peut donc attendre de la PLC qu'une gestion passive de ces contraintes, au sens où les domaines des variables ne peuvent être réduits *a priori* par les contraintes.

Une contrainte disjonctive entre deux tâches t1 et t2 peut s'exprimer sous la forme d'une disjonction d'inégalités du type :

$$(S_2 - F_1 \geq 0) \vee (S_1 - F_2 \geq 0)$$

où "$\vee$" représente la disjonction logique. Elle traduit le fait que l'une des deux tâches doit précéder l'autre. Dans le cas où les durées sont connues, la contrainte s'exprime :

$$(S_2 - S_1 \geq D_1) \vee (S_1 - S_2 \geq D_2).$$

5.1.2.1. Modélisation par un point de choix

C'est la solution la plus directe (mais aussi la plus inefficace) pour traduire une telle contrainte, la disjonction en Prolog étant naturellement représentée par l'existence de 2 clauses définissant la même relation.

Exemple (CHIP) :

```
disjonction1(S1,S2,D1,D2) :- S2 #>= S1 + D1.
disjonction1(S1,S2,D1,D2) :- S1 #>= S2 + D2.
```

L'inconvénient majeur de cette solution est de créer, pour n tâches en conflit, un espace de recherche de taille exponentielle en n. Les feuilles de l'arbre généré par la

pose de l'ensemble des contraintes disjonctives représentent une solution séquentielle du problème et il est possible d'obtenir les marges temporelles associées à une séquence donnée avant toute instanciation des dates de début. La figure 4 représente les réductions éventuelles de domaine temporel d'une tâche (fenêtre) après passage par un point de choix.

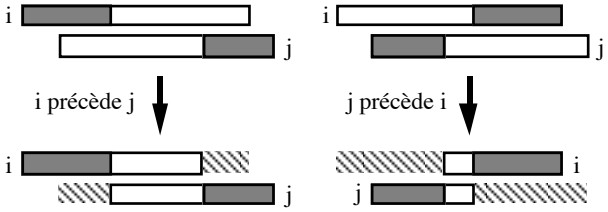

Figure 4. *Réduction des fenêtres après un point de choix lié à une disjonction*

L'ordre de pose des contraintes proposé dans les exemples est en général statique et imposé par une boucle d'énumération des couples possibles. L'efficacité peut être augmentée en remplaçant cet ordre statique par un ordre plus dynamique qui forme en priorité les couples de tâches dont les fenêtres temporelles se chevauchent le moins (principe du *first-fail*). La complexité d'une telle procédure ne garantit pas au bout du compte une amélioration très nette de l'efficacité étant donnée les nombreux tests de comparaison qu'exige le reclassement dynamique des tâches avant chaque pose de contrainte disjonctive.

5.1.2.2. Modélisation par propagation conditionnelle

L'utilisation de la primitive de propagation conditionnelle de CHIP, présentée au §4.4, permet de s'affranchir de la pose d'un point de choix, en retardant l'activation des contraintes disjonctives tant que les domaines des variables ne permettent pas de trancher pour l'une des deux configurations. La modélisation est la suivante :

```
disjonction2(S1,S2,D1,D2) :-
        if S1 #< S2+D2 then S2 #>= S1 + D1,
        if S2 #< S1+D1 then S1 #>= S2 + D2.
```

5.1.2.3. Autre type d'inférence

La prise en compte de contraintes disjonctives, que ce soit par pose de points de choix ou par propagation conditionnelle ne réduit que les bornes extrêmes des domaines des variables (*2B-consistance* [LHO 93, FAR 94]). Il existe cependant des cas où les contraintes de disjonction pourraient opérer des réductions de domaine *a priori*, sans pose de point de choix, et sans attendre qu'une des deux alternatives de séquencement soit impossible, comme l'illustre la figure 5.

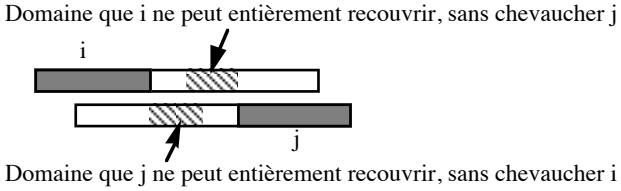

Figure 5. *Réduction interne de domaines*

L'interdiction de valeurs au sein d'un domaine, amène à créer des “trous” dans les domaines, information plus riche que la seule actualisation des bornes. En CHIP, ce type d'inférence peut être réalisée grâce à l'utilisation de la primitive “cumulative”, que nous détaillons plus loin (§5.1.4), ou directement par une proposition présentée et démontrée dans [LOP 95] dont la programmation est donnée ci-dessous. Celle-ci utilise le prédicat “notin/3” qui permet de supprimer des valeurs au sein du domaine d’une variable. D’autre part, ce programme n’engendre qu’une manipulation passive des contraintes ; il sera donc intéressant de poser un mécanisme de “démon” afin de le rendre dynamique.

```
arc_consistance(S1,S2,D1,D2) :-
        D is D1 + D2,
        if S2 #< S2 + D
            then trou(S1,S2,D1,D2),
        if S1 #< S1 + D
             then trou(S2,S1,D2,D1).

trou(S1,S2,D1,D2) :-
        domain_info(S2,S2min,S2max,_,_,_),
        Min is S2max - D1 + 1,
        Max is S2min + D2 - 1,
        notin(S1,Min,Max).
```

5.1.3. *Ensembles non postérieurs / non antérieurs*

La contrainte de disjonction peut déduire des conclusions fortes pour le séquencement global. En pratique cependant, cette règle peut ne pas être très active lorsque les durées opératoires sont petites par rapport aux fenêtres temporelles. Il peut alors être plus intéressant d’étudier les positions extrêmes d’une tâche par rapport à un groupe de tâches [ESQ 87, CAR 88]. On examine ainsi si une tâche peut être exécutée avant ou après un sous-ensemble donné de tâches. Par exemple, si A ne peut pas être placée avant B et C, elle doit être placée au moins après B ou C. Le sous-ensemble {B,C} est un *ensemble non-postérieur* à A.

Le programme ci-dessous, implémenté en CHIP, permet de déterminer les ensembles non-postérieurs et non-antérieurs maximaux d’une tâche et d’actualiser le domaine de la date de début de cette tâche. Il utilise le mécanisme de propagation conditionnelle et les prédicats “minimum/2” et “maximum/2” qui permettent de calculer les extrémités temporelles d’un ensemble de tâches.

```
non_posterieur(Si,Di,T) :-
        duree_T(T,Sigma),
        liste_fins(T,L_fins),
        minimum(Fin_min,L_fins),
        maximum(Fin_max,L_fins),
        D is Di + Sigma,
        if Fin_max #< Si + D
            then Si #>= Fin_min.
```

```
non_anterieur(Si,Di,T) :-
        duree_T(T,Sigma),
        liste_debuts(T,L_debuts),
        minimum(Debut_min,L_debuts),
        maximum(Debut_max,L_debuts),
        if Si #< Debut_min + Sigma
            then Si + Di #<= Debut_max.
```

5.1.4. *Contraintes cumulatives*

Les limitations sur la disponibilité des ressources induit des *contraintes cumulatives*. Ces contraintes interdisent l'exécution simultanée d'un nombre de tâches tel que l'intensité totale d'utilisation de la ressource dépasse l'intensité maximale de la ressource. Il s'en suit un séquencement partiel des tâches, moins fort que le séquencement total imposé par les contraintes disjonctives. A titre d'exemple, un problème de 4 tâches nécessitant toutes une ressource disponible en deux exemplaires est un problème à contraintes cumulatives : tout sous-ensemble de trois tâches doit contenir au moins deux tâches ordonnées. Les problèmes de découpe de pièces sur un format de taille limitée représentent également une version “spatiale” de problèmes d'ordonnancement à contraintes cumulatives [ESQ 93a, CHA 94].

Le langage CHIP est le premier langage de PLC ayant été doté d'une primitive dédiée au traitement des contraintes cumulatives. Cette primitive réalise une interprétation active mais limitée des contraintes de ce type. L'utilisation de cette contrainte n'est pas facile étant donné le nombre impressionnant d'options qui sont proposées (“cumulative/8”). Toutefois, en dehors de toute approche d'optimisation, elle permet de resserrer les domaines des dates de début, en créant éventuellement des trous dans les domaines.

Les différents arguments qui composent la contrainte cumulative sont respectivement, des listes : (S) de dates de début des tâches, (D) de durées des tâches, (R) de quantités de ressource consommée, (F) de dates de fin des tâches, (E) de surfaces des tâches ; (Q) une capacité de ressource ; (Fin) une durée totale d’ordonnancement ; une valeur intermédiaire. La plupart des arguments peuvent prendre des valeurs entières ou s’inscrire dans des domaines finis. Une caractéristique originale de cette primitive concerne l’argument représentant les surfaces des tâches. Cette surface, ou *énergie*, est une grandeur agrégée qui permet de raisonner simultanément sur le temps et les ressources. Son calcul est issu du produit de la quantité de ressource requise par une tâche par sa durée. Elle permet d’asseoir certains raisonnements à un niveau plus élevé d’abstraction, lorsqu’on ne connaît pas de manière précise les caractéristiques de réalisation, *e.g.*, la durée des tâches. Cette notion sert de base à une méthode de propagation de contraintes mettant en jeu des *bilans énergétiques* [LOP 92].

Les exemples suivants [COS 93] illustrent l’utilisation de la contrainte cumulative. Dans le premier exemple, les durées des tâches sont connues et on cherche à minimiser la durée totale de l’ordonnancement. Dans le second, les durées

ne sont pas connues et on cherche à minimiser la surface au dessus d'une valeur moyenne d'utilisation de ressource fixée à 2.

Prédicats communs aux 2 exemples.

```
definition(S,F,R,Fin,Q,SurfInt) :-
        S=[S1,S2,S3], F=[F1,F2,F3], R=[1,2,2],
        S :: 1..10, F :: 1..10, Q :: 1..3,
        [Fin,SurfInt] :: 1..10.

labeling2([]).
labeling2([X|L]) :-
        indomain(X),
        labeling2(L).
```

Exemple 1 : durées connues, minimisation de la durée de l'ordonnancement.

```
top(S,F,Fin,Q,SurfInt) :-
        definition(S,F,R,Fin,Q,SurfInt),
        D=[4,2,3],
        cumulative(S,D,R,F,unused,Q,Fin,[2,SurfInt]),
        min_max(labeling2(S),Fin).
```

```
?- top(S,F,Fin,Q,SurfInt).

                S=[1,1,3]       F=[5,3,6]
                Fin=6           Q=3             SurfInt=4
```

Exemple 2 : durées non connues, minimisation de la surface intermédiaire.

```
top(S,F,Fin,Q,SurfInt) :-
        definition(S,F,R,Fin,Q,SurfInt),
        D=[D1,D2,D3],
        E=[4,4,6],
        D :: 1..10,
        cumulative(S,D,R,F,E,Q,Fin,[2,SurfInt]),
        min_max(labeling2(S),SurfInt).
```

```
?- top(S,F,Fin,Q,SurfInt).

                S=[1,4,6]       F=[5,6,9]
                Fin=9           Q=3             SurfInt=1
```

Dans [AGG 92], les auteurs présentent certaines applications de la contrainte cumulative à des problèmes de placement, d'ordonnancement de projet ou encore de job-shop. Pour ce dernier cas, ils utilisent une procédure du type "labeling1", un prédicat semblable à "pose_precede", et la contrainte cumulative dans laquelle la consommation de ressource des tâches est équivalente à une liste de 1. Des résultats expérimentaux, en particulier sur le fameux problème 10×10×10, montrent qu'il est

possible d'obtenir des résultats intéressants (une solution à 1088 en 1 s, la solution optimale en 25 mn sur SPARC-IPC 12MB) et ceci par une programmation simple mettant en jeu la contrainte cumulative.

On constate toutefois (voir exemple 3) qu'en l'absence de toute génération, les déductions engendrées par la pose de la contrainte cumulative de CHIP dépendent de l'ordre dans lequel les variables sont passées en arguments, sans qu'aucune indication visant à tirer parti de cet ordre ne soit fournie dans le manuel d'utilisation.

Exemple 3 :

/* Trois tâches requièrent chacune une unité d'une ressource disponible en deux exemplaires. La troisième tâche est bloquée dans sa fenêtre temporelle ; le problème devrait donc se réduire à un problème disjonctif sur les deux premières tâches. Le résultat fournit bien toute l'information sur le début de la deuxième tâche, alors que des dates interdites sont conservées dans le domaine de la première. */

```
incompletude_cumulative([T1,T2,T3]) :-
        T1 :: 0..3, T2 :: 0..6, T3 = 0,
        cumulative([T1,T2,T3],[6,3,11],[1,1,1],unused,unused,2,unused,unused).

?- incompletude_cumulative(L).

                L = [T1 in {0..3}, T2 in {0,6}, 0]
```

/* On s'aperçoit même que le domaine de T1 peut être modifié si la durée de T3 change, ce qui ne devrait avoir aucune sorte d'influence sur les autres tâches... */

5.2. *Problèmes de séquencement*

Dans ce qui précède, le séquencement de tâches est traduit implicitement par des contraintes d'inégalités entre dates de début des tâches. Un autre modèle, utilisant une *variable booléenne* par couple de tâches en disjonction, permet de traiter symboliquement les problèmes à contraintes disjonctives.

Soit O un ensemble de tâches tels que toute paire de tâche soit nécessairement ordonnée. On peut modéliser le problème du séquencement absolu des tâches par un ensemble de n(n-1)/2 variables booléennes (une par couple de tâches ordonnées lexicographiquement) :

$\forall (i, j) \in O^2$ / i<j, $X_{ij} = 1$ (vrai) si i précède j, $X_{ij} = 0$ (faux) si j précède i.

L'intérêt d'un tel modèle est d'abord la représentation de contraintes basées sur la relation symbolique de succession, du type :

1. ensemble non antérieur : *e.g.*, l'expression $X_{12} \vee X_{13} \vee X_{14}$ impose à la tâche T1 d'être située avant T2 ou T3 ou T4 ;

2. ensemble non postérieur ;

3. ensemble *non intercalable* : entre deux tâches i et j on ne peut pas insérer simultanément toutes les tâches d'un ensemble donné ;

4. couplage entre 2 relations de succession ; la contrainte “i précède j ou k précède l” peut se réécrire : $X_{ij} \vee X_{kl}$.

Exemple.

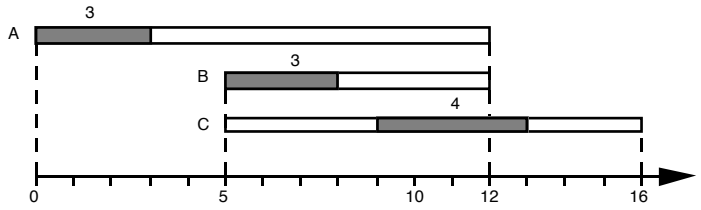

Figure 6. *Trois tâches en disjonction*

Sur cet exemple, une recherche d'ensembles non antérieurs et non postérieurs fournit les résultats suivants :
{A,B} est non postérieur de C (on ne peut pas placer C avant A et B) ;
{B,C} est non antérieur de A (on ne peut pas placer A après B et C).

Exprimées à l'aide de variables booléennes ces relations conduisent à :

$$(X_A^C \vee X_B^C) \wedge (X_A^B \vee X_A^C)$$
$$= X_A^C \wedge (X_A^B \vee X_B^C)$$

soit en CHIP (en rajoutant la relation de transitivité) :

```
?- (Xac!Xbc) & (Xab!Xac) & ((Xab & Xbc) => Xac) &= 1.

        Xac = 1
```

Ce traitement révèle une relation de précédence entre A et C, nouvelle contrainte qui peut améliorer l'efficacité d'une procédure de résolution.
Pour mettre en place ce type d'inférence symbolique il faut associer à chaque couple de tâches une variable booléenne, *e.g.*, en modifiant disjonction2 (*cf.*5.1.2.2).

```
disjonction3(S1,D1,S2,D2,X12) :-
      if S1 #< S2 + D2
          then precede(S1,S2,D1,X12),
      if S2 #< S1 + D1
          then precede(S2,S1,D2,X21),
      X21 &= not(X12).

precede(Si,Sj,Di,1) :-  Sj #>= Si + Di.
```

Les contraintes ci-dessus sont des cas particuliers de contraintes plus générales exprimables sous forme d'expressions booléennes bâties sur la relation de précédence entre deux tâches. On peut trouver de telles contraintes dans la formulation initiale des problèmes ; on peut également rechercher certaines formes de contraintes à partir d'une *analyse sous contraintes*, en particulier celles qui

autorisent une actualisation des fenêtres d'exécution des tâches dans le but de caractériser les solutions admissibles [ERS 76]. Une précaution doit cependant être prise lors du traitement symbolique de ces contraintes ; la transitivité de la relation de précédence doit être posée explicitement pour tout triplet de variables booléennes, ce qui alourdit de manière non négligeable le système de contraintes.

5.3. *Discussion*

Dans le cas disjonctif, la caractérisation des solutions admissibles d'un problème d'ordonnancement peut faire intervenir à la fois des contraintes numériques (temporelles) et symboliques (de séquencement). Pour chacun de ces cas, on a montré l'adéquation d'un modèle à variables entières ou à variables booléennes.

A l'heure actuelle cependant, il n'existe pas de moyen élégant de lier les inférences réalisées sur le domaine des variables entières et les variables booléennes, du fait de l'impossibilité de construire des expressions mélangeant ces deux types de variables dans les langages de PLC (*cf.* §4.5). Si la propagation de contraintes peut s'effectuer depuis le système de dates vers le système booléen grâce à une propagation conditionnelle, on ne peut en revanche, associer de "démons" aux variables booléennes dans le but d'effectuer la propagation dans l'autre sens.

6. Conclusion

La PL offre un cadre à la fois rigoureux et souple pour la représentation des problèmes combinatoires dont la résolution nécessite le parcours non-déterministe d'un espace de recherche. Par sa sémantique déclarative et la puissance de la logique du premier ordre, elle procure lisibilité, concision et flexibilité aux programmes. Par contre la stratégie de résolution s'avère très inefficace étant donné la faiblesse des contraintes d'unification comme seul mécanisme de propagation de contraintes, comparés aux règles spécifiques existant sur les variables typées, comme les domaines finis, entiers, réels ou encore rationnels. D'autre part le retour arrière chronologique en cas d'échec reflète une utilisation passive des contraintes.

Augmentée de règles d'inférence spécifiques au traitement de contraintes sur des variables typées, la PLC devient beaucoup plus intéressante pour la représentation et la résolution de ces mêmes problèmes, notamment les problèmes d'ordonnancement. Cependant les différents paradigmes de résolution sont encore relativement cloisonnés et il n'existe pas encore de langage complet capable de faire interagir les mécanismes de propagation de contraintes sur des domaines différents, ce qui permettrait d'aborder des problèmes dont la modélisation nécessite plusieurs types de variables.

Au contraire, certains langages se spécialisent dans la résolution de problèmes très spécifiques, privilégiant le point de vue de l'optimisation au détriment de la caractérisation de la consistance des solutions et de la standardisation de la PLC, standardisation dont l'absence a longtemps nuit (et nuit encore) au langage Prolog lui-même. Le but est clair : concurrencer les outils spécialisés, ce qui entre en contradiction avec les buts originaux de la PLC qui étaient la formalisation et l'intégration de mécanismes généraux de propagation de contraintes. De plus, le fonctionnement des primitives les plus évoluées n'est pas clairement expliqué

(concurrence oblige) et leur utilisation passe par une syntaxe relativement contraignante, ce qui ne facilite pas leur appropriation par les programmeurs.

7. Bibliographie

[ACM 92] *Communications of ACM*, Special section on Logic Programming, 35(3), 1992.

[AGG 92] AGGOUN A., BELDICEANU N., Extending CHIP in order to solve complex scheduling and placement problems, *Actes des Journées Francophones de Programmation Logique,* pp.51-66, 1992.

[BAR 88] BARTUSCH M., MÖHRING R.H., RADERMACHER F.J., Scheduling project networks with resource constraints and time windows, *Annals of Operations Research*, 16, pp.201-240, 1988.

[BAP 92] P. Baptiste, B. Legeard, C. Varnier, Hoist scheduling problem: An approach based on Constraint Logic Programming, *Proc. IEEE International Conference on Robotics and Automation*, pp.1139-1144, Nice, 1992.

[BEN 93] BENHAMOU F., COLMERAUER A., *Constraint Logic Programming : selected research*, Logic Programming Series, E. Shapiro (ed.), MIT Press, 1993.

[BOC 93] BOCKMAYR A., Logic Programming with Pseudo-Boolean Constraints, *Constraint Logic Programming : selected research*, Logic Programming Series, F. Benhamou & A. Colmerauer (eds.), MIT Press, 1993.

[BOI 94] BOIZUMAULT P., DELON Y., PERIDY L., Constraint Logic Programming for Examination Timetabling, *Journal of Logic Programming*, 1995, to appear.

[BUT 87] BUTTNER W., SIMONIS H., Embedding Boolean expressions into Logic Programming, *Journal of Symbolic Computation,* 4, 1987.

[CAR 88] CARLIER J., PINSON E., An algorithm for solving the Job-shop problem, *Management Science*, 35(2), pp.164-176, 1988.

[CAR 93] Mc CARTHY B.L., LIU J., Addressing the gap in scheduling research : A review of optimization and heuristic methods in production scheduling, *IJPR*, 31, pp.59-79, 1993

[CHA 94] CHAMARD A., FISHLER A., MADE : a workshop scheduler system written in CHIP, *Proc. 2nd International Conference on the Practical Application of Prolog (PAP'94)*, London (UK), 1994.

[CLO 81] CLOCKSIN W.F., MELLISH C.S., *Programming in Prolog*, Springer Verlag, New York, 1981.

[COL 83] COLMERAUER A., KANOUI H., Van CANEGHEM M., Prolog, Bases théoriques et développements actuels, *Techniques et Science Informatiques*, 2(4), 1983.

[COL 90] COLMERAUER A., An introduction to PrologIII, *Communications of ACM*, 33(7), pp.69-90, 1990.

[COS 93] COSYTEC, *Reference Manual,* COSY/REF/001, 1993.

[DEC 91] DECHTER R., MEIRI I., PEARL J., Temporal Constraints Networks, *Artificial Intelligence*, 49, pp.61-95, 1991.

[DEC 92] DECHTER R., From local to global consistency, *Artificial Intelligence,* 55, 1992.

[DIB 70] DIBON M., *Ordonnancement et potentiels / Méthode M.P.M.*, Hermann, 1970.

[DIN 87] DINCBAS M., SIMONIS H., Van HENTENRYCK P., Extending equation solving and constraint handling in logic programs, *Proc. Colloquium on Resolution of Equations in Algebraic Structures*, MCC, Austin (Texas, USA), 1987.

[DIN 88] DINCBAS M., Van HENTENRYCK P., SIMONIS H., AGGOUN A., GRAF T., BERTHIER F., The Constraint Logic Programming Language CHIP, *Proc. International Conference on Fifth Generation Computer Systems (FGCS'88)*, pp.693-702, Tokyo (Japan), 1988.

[DIN 90] DINCBAS M., SIMONIS H., Van HENTENRYCK P., Solving large combinatorial problems in logic programming, *Journal of Logic Programming*, 8, pp.75-93, 1990.

[ELM 77] ELMAGHRABY S.E., *Activity networks*, John Wiley & Sons, 1977.

[ELM 92] ELMAGHRABY S.E., KAMBUROWSKI J., The analysis of activity networks under generalized precedence relations (GPRs), *Management Science*, 38, pp.1245-1263, 1992.

[ERS 76] ERSCHLER J., *Analyse sous contraintes et aide à la décision pour certains problèmes d'ordonnancement,* Thèse de Doctorat d'état, Univ. Paul Sabatier, Toulouse, 1976.

[ERS 79] ERSCHLER J., FONTAN G., ROUBELLAT F., Potentiels sur un graphe non-conjonctif et analyse d'un problème d'ordonnancement à moyens limités, *RAIRO-RO*, 13, 1979.

[ESQ 87] ESQUIROL P., *Règles et processus d'inférence pour l'aide à l'ordonnancement de tâches en présence de contraintes*, Thèse de Doctorat, Univ. Paul Sabatier, 1987.

[ESQ 93a] ESQUIROL P., LOPEZ P., BOY G., BRADSHAW J., HAUDOT L., SICARD M., *SCOOP : Système COopératif pour l'Ordonnancement de Production,* rapport LAAS 94171, 1994.

[ESQ 93b] ESQUIROL P., LOPEZ P., HUGUET M-J., Modelling and managing disjunctions in scheduling problems, *Journal of Intelligent Manufacturing*, 6, 1995, to appear.

[FRE 78] FREUDER E.C., Synthesizing constraint expressions, *Communications of ACM*, 21(11), 1978.

[FRE 82] FREUDER E.C., A sufficient condition for backtrack-free search, *Journal of ACM*, 29(1), 1982.

[GAR 79] GAREY M.R., JOHNSON D.S., *Computers and Intractability*. W.H. Freeman and Company, New York, 1979.

[GOT 93] GOTHA, Les problèmes d'ordonnancement, *RAIRO-RO*, 27(1), pp.77-150, 1993.

[HAN 88] HAN C., LEE C., Comments on Mohr and Henderson's path consistency algorithm, *Artificial Intelligence*, 36, 1988.

[HAR 80] HARALICK R.M., ELLIOT G.L., Increasing search efficiency for Constraint Satisfaction Problems, *Artificial Intelligence*, 14, 1980.

[HEN 89] Van HENTENRYCK P., *Constraint satisfaction in logic programming*, Logic Programming Series, E. Shapiro (ed.), MIT Press, 1989.

[HUG 94] HUGUET M-J., *Approche par contraintes pour l'aide à la décision et à la coopération en gestion de production*, Thèse de Doctorat, INSA Toulouse, 1994.

[JAF 87] JAFFAR J., MICHAYLOV S., Methodology and Implementation of a CLP System, *Proc. 4th International Conference on Logic Programming*, Melbourne (Australia), 1987.

[JAF 94] JAFFAR J., MAHER M.J., Constraint Logic Programming : A survey, *Journal of Logic Programming*, 19-20, pp.503-581, 1994.

[LAU 76] LAURIERE J-L., *Un langage et un programme pour énoncer et résoudre des problèmes combinatoires*, Thèse de Doctorat, Univ. Pierre et Marie Curie, Paris, 1976.

[LEG 92] LEGEARD B., BAPTISTE P., Solving some problems in the CIM area : perspectives of the constraint logic programming approach, *Proc. International Conference on Data and Knowledge Systems for Manufacturing and Engineering (DKSM'92),* Lyon, 1992.

[LEV 94] LEVY M-L, LOPEZ P., PRADIN B., Characterization of feasible schedules for the Flow-shop problem : A decomposition approach, *Proc. European Workshop on Integrated Manufacturing Systems Engineering (IMSE'94)*, pp.307-315, Grenoble, 1994.

[LHO 93] LHOMME O., Consistency techniques for numeric CSPs, *Proc. 13th Joint Conference on Artificial Intelligence (IJCAI'93)*, Chambéry, 1993.

[LOP 92] LOPEZ P., ERSCHLER J., ESQUIROL P., Ordonnancement de tâches sous contraintes : une approche énergétique, *RAIRO-APII*, 26(5-6), 1992.

[LOP 95] LOPEZ P., HAUDOT L., SICARD M., ESQUIROL P., Constraint-based approach to design a DSS for scheduling, *Proc. 3rd International Conference on the Practical Application of Prolog (PAP'95)*, Paris, 1995.

[MAC 77] MACKWORTH A.K., Consistency in networks of relations, *Artificial Intelligence*, 8(1), 1977.

[MOH 86] MOHR R., HENDERSON T.C., Arc and path consistency revisited, *Artificial Intelligence*, 28, 1986.

[MON 74] MONTANARI U., Networks of Constraints : fundamental properties and applications to picture processing, *Information Sciences*, 7, 1974.

[NAD 89] NADEL B.A., Constraint Satisfaction Algorithms, *Journal of Computers Intelligence*, 5, 1989.

[PAP 82] PAPADIMITRIOU C.H., STEIGLITZ K., *Combinatorial optimization : Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[ROB 65] ROBINSON J.A., A machine-oriented logic based on the resolution principle, *Journal of ACM*, 12(1), 1965.

[ROY 70] ROY B., *Algèbre moderne et théorie des graphes*, tome II, Dunod, 1970.

[RUD 74] RUDEANU S., *Boolean Functions and Equations*, North Holland, Amsterdam, London, 1974.

[THI 93] THIERRY C., BEL G., ESQUIROL P., A Constraint Based Model For MultiSite Scheduling, *Proc. IFAC'93*, Sidney (Australia), 1993.

[TSA 93] TSANG E., *Foundations of constraint satisfaction*, Computation in Cognitive Science Series, Academic Press, 1993.

[WAL 72] WALTZ D.L., *Generating semantic descriptions from drawings of scenes with shadows*, MAC-AI-TR-271, MIT, 1972.

[WAL 94] WALLACE M., Applying constraints for scheduling, *Constraint Programming*, B. Mayoh, E. Tyugu and J. Penjaam (eds.), NATO Advanced Science Institute Series, Springer-Verlag, 1994.

Biographies

Patrick ESQUIROL a obtenu sa thèse de Doctorat de l'Université Paul Sabatier en 1987. Il a ensuite rejoint une société de services en informatique industrielle pour développer plusieurs projets dans les domaines de l'ordonnancement et des systèmes à base de connaissances. Depuis 1989, il est maître de conférences au Département de Génie Electrique de l'Institut National des Sciences Appliquées de Toulouse, où il enseigne l'algorithmique, la programmation et l'intelligence artificielle. Il effectue sa recherche dans le groupe Systèmes de Production du Laboratoire d'Analyse et d'Architecture des Systèmes du C.N.R.S. à Toulouse. Ses travaux portent sur l'étude des mécanismes d'analyse et de propagation de contraintes et la conception de logiciels coopératifs pour les problèmes de gestion du temps et des ressources.
Email : esquirol@laas.fr

Pierre LOPEZ a obtenu sa thèse de Doctorat de l'Université Paul Sabatier en 1991. Depuis 1992, il est chargé de recherche au CNRS dans le groupe Systèmes de Production du Laboratoire d'Analyse et d'Architecture des Systèmes du C.N.R.S. à Toulouse. Chargé de cours au Département de Génie Electrique de l'Institut National des Sciences Appliquées de Toulouse, il enseigne la théorie des graphes, l'ordonnancement et la simulation des systèmes à événements discrets. Ses travaux de recherche concernent l'analyse et la propagation de contraintes, en particulier les contraintes énergétiques, les techniques de décomposition des problèmes d'ordonnancement et la conception de systèmes coopératifs pour les problèmes de gestion du temps et des ressources.
Email : lopez@laas.fr