Université d'Orléans

Explications de retraits de valeurs en programmation par contraintes et application au diagnostic déclaratif

Thèse

présentée pour l'obtention du titre de

Docteur de l'Université d'Orléans

spécialité Informatique

par

Willy Lesaint

soutenue le 24 Novembre 2003

Composition du jury présidé par Pierre Deransart

Rapporteurs:	Irène Guessarian	Professeur - Université de Paris VI
	Patrice Boizumault	Professeur - Université de Caen
Examinateurs:	Pierre Deransart	Directeur de recherche - INRIA Rocquencourt
	Jean-Philippe Anker	Professeur - Université d'Orléans
	Gérard Ferrand	Professeur - Université d'Orléans
	Alexandre Tessier	Maître de Conférence - Université d'Orléans

Thèse réalisée au LIFO sous la direction de Gérard Ferrand et Alexandre Tessier.

Université d'Orléans

Explications de retraits de valeurs en programmation par contraintes et application au diagnostic déclaratif

Thèse

présentée pour l'obtention du titre de

Docteur de l'Université d'Orléans

spécialité Informatique

par

Willy Lesaint

soutenue le 24 Novembre 2003

Composition du jury présidé par Pierre Deransart

Rapporteurs:	Irène Guessarian	Professeur - Université de Paris VI
	Patrice Boizumault	Professeur - Université de Caen
Examinateurs:	Pierre Deransart	Directeur de recherche - INRIA Rocquencourt
	Jean-Philippe Anker	Professeur - Université d'Orléans
	Gérard Ferrand	Professeur - Université d'Orléans
	Alexandre Tessier	Maître de Conférence - Université d'Orléans

Thèse réalisée au LIFO sous la direction de Gérard Ferrand et Alexandre Tessier.

Table des matières

Chapitre 1

Introduction

Depuis quelques années, la programmation par contraintes a montré son efficacité pour la résolution de problèmes difficiles, tant au point de vue de leur modélisation que de leur résolution. En effet, l'approche déclarative de ce paradigme de programmation permet de résoudre les Problèmes de Satisfaction de Contraintes (CSP) en faisant abstraction des calculs, laissant ainsi le programmeur se concentrer uniquement sur les contraintes modélisant son problème. Une contrainte est une relation entre des variables du problème. Pour les CSP, ces variables prennent leurs valeurs dans des domaines finis. Le nombre d'outils académiques et commerciaux ayant vu le jour, CHIP (Dincbas *et al.*, 1988), ILOG-SOLVER (Puget, 1994), GNU-PROLOG (Diaz & Codognet, 2000), ECL^{i}PS^e (Aggoun *et al.*, 2001), CHOCO (Laburthe & the OCRE project, 2000), SICSTUS PROLOG (Sicstus, 2003) pour ne citer qu'eux, est une preuve indiscutable de l'intérêt suscité par ce paradigme de programmation. Ils sont utilisés avec succès pour les problèmes d'ordonnancement, de planification, de gestion de ressources, de configuration, ...

Il existe plusieurs méthodes de recherche pour traiter de tels problèmes. Parmi elles, la réduction de domaine par des notions de consistance s'est révélée très performante pour les problèmes de grande taille. L'idée est de supprimer peu à peu des valeurs qui ne vérifient pas les relations posées par les contraintes. Cependant, cette méthode permet souvent de n'obtenir qu'une approximation des solutions. Elle est alors mêlée à des méthodes d'énumération basées principalement sur des techniques de backtrack. Le solveur peut alors fournir des instanciations (i.e. une valeur par variable) satisfaisant toutes les contraintes du problème. Le maintien de consistance qui mêle réduction de domaine et énumération, est une des méthodes les plus efficace en général. Elle est utilisée dans la plupart des solveurs.

La communauté contrainte s'intéresse depuis peu à la notion d'explication, sous le nom de justification, de nogood, d'ensemble conflit ou d'explication de contradiction. L'idée est de conserver des informations pendant la résolution d'un problème afin de pouvoir les utiliser dans certaines situations. Ces informations ont pour objectif d'expliquer le retrait de valeurs ou l'obtention d'échec (i.e. pas de solution). Leur principale utilisation concerne la relaxation de contrainte, c'est-à-dire quand une contrainte doit être retirée (que ce soit une contrainte du problème ou une contrainte posée par l'énumération).

Cela donne lieu principalement à trois applications. D'abord, lors d'une recherche systématique, il est alors possible d'éviter de tester des instanciations partielles qui ont déjà mené à un échec. Ensuite, face aux problèmes sur-contraints (i.e. les problèmes pour lesquels il n'existe pas de solution), elles offrent un choix de contraintes à retirer du problème pour lever la contradiction. Enfin, les explications sont utilisées pour gérer les problèmes dynamiques (i.e. les problèmes pour lesquels l'ensemble de contraintes varie au cours du temps).

Une lacune de la programmation par contrainte concerne le manque d'outils d'aide à la mise au point. Celle-ci comprend deux aspects : la mise au point de performance qui a pour but d'améliorer les performances de la résolution et la mise au point de correction pour corriger les éventuelles erreurs du programme. La principale cause du manque d'outils est la déclarativité de la programmation par contraintes. En effet, pour les langages algorithmiques, la mise au point se fait souvent par une analyse des calculs. Pour la programmation par contrainte, le solveur est une boîte noire. D'une part les calculs y sont cachés, l'utilisateur n'a pas besoin de les connaître et d'autre part ils sont souvent très complexes. Cette complexité rend impossible la décomposition du programme et donc de considérer un fragment de programme sans prendre en compte le reste du calcul. De plus, il est très difficile pour l'utilisateur de décider si un état de l'éxécution est correct ou non.

Les outils proposés jusque là, essentiellement pour l'aspect performance, reposent principalement sur une approche visuelle. Ils tentent de fournir des vues des domaines ou de l'arbre de recherche (pour l'énumération). Ils permettent ainsi de découvrir où le solveur perd du temps et comment une autre stratégie pourrait être utilisée plus efficacement. Ces outils sont par contre peu efficaces pour l'aspect correction qui reste un problème ouvert. Pour déboguer un problème avec contraintes il faut donc rester, tant que possible, à un niveau déclaratif plus compréhensible pour un utilisateur. Il serait de toute façon incohérent de lui demander de comprendre les calculs pour déboguer alors qu'il n'en a pas eu besoin pour programmer. Les traces du calcul sont peu exploitables, principalement à cause de leur taille, surtout si l'utilisateur n'a aucune idée sur l'emplacement de l'erreur. Il faut donc également restreindre le plus possible l'information à traiter.

Cette thèse propose principalement trois contributions au domaine de la programmation par contraintes : une formalisation de la réduction de domaine comme application de la théorie de point fixe, une définition d'explication sous une forme arborescente et enfin l'utilisation de ces explications pour le diagnostic déclaratif de réponse manquante.

La réduction de domaine a déjà été décrite dans différents travaux (Benhamou, 1996; Codognet & Diaz, 1996; Van Emden, 1995; Van Hentenryck, 1989). Nous proposons ici une reformulation de ce formalisme en termes ensemblistes. La réduction de domaine est vue comme un calcul de clôture par des opérateurs monotones et peut être considérée comme une application de la théorie du point fixe. Ce calcul de clôture se fait par l'utilisation d'itérations chaotiques (Fages *et al.*, 1995; Cousot & Cousot, 1977) des opérateurs. Le formalisme proposé possède l'avantage, hormis le fait qu'il est plus facile de manipuler des ensembles, d'être totalement indépendant des stratégies et par là même s'applique à tout type de solveur effectuant de la réduction de domaine. Les opérateurs utilisés sont associés aux contraintes du problème selon les notions de consistance utilisées et reflètent exactement les réduction effectuées par un solveur classique. Il est montré comment la réduction de domaine peut être mêlée à l'énumération pour obtenir exactement les solutions du CSP.

Nous proposons ensuite une vision duale de la réduction de domaine. A l'opposé de la réduction de domaine qui considère les valeurs restant dans le domaine, on s'intéresse ici aux valeurs retirées. Cette vision permet de définir les opérateurs par des règles exprimant le retrait d'une valeur comme conséquence d'autres retraits. Une telle règle explique donc exactement le retrait d'une valeur. Dès lors ces règles permettent la définition inductive d'arbres de preuves appelés arbres explicatifs. Certains peuvent être construits à partir d'un calcul et sont alors appelés arbres explicatifs calculés. Ces arbres contiennent toute l'information expliquant le retrait d'une valeur durant un calcul et peuvent être considérés comme l'essence de la réduction de domaine. Les notions existantes d'explication (Prosser, 1993; Schiex & Verfaillie, 1993; Ginsberg, 1993; Bessière, 1991; Jussien, 2001) peuvent facilement y être retrouvées.

Enfin, nous proposons une approche de la mise au point de réponses manquantes basée sur le diagnostic déclaratif grâce à la notion d'arbre explicatif. Le diagnostic déclaratif proposé repose sur la notion de symptôme de réponse manquante. Un tel symptôme est une valeur de variable qui a été retirée pendant le calcul du solveur alors que l'utilisateur l'attendait comme participant à une solution. Ces symptômes sont définis par rapport à une sémantique attendue qui formalise la connaissance de l'utilisateur. Dans un arbre explicatif ayant pour racine un tel symptôme se trouve une erreur (responsable du symptôme). Le but du diagnostic déclaratif est de localiser cette erreur en découvrant peu à peu d'autres symptômes dans l'arbre. L'utilisateur n'a donc pas à s'interroger sur le calcul mais sur les résultats de celui-ci. L'utilisation des explications semble donc une approche adéquate pour la mise au point de réponse manquante. Elles peuvent en effet être considérées comme une trace déclarative des calculs et permettent d'adapter les méthodes de débogage algorithmique introduites par Shapiro (Shapiro, 1982). Rechercher une erreur dans une telle trace semble plus facile, d'une part parce qu'elle contient peu d'information par rapport à une trace classique et d'autre part, parce que l'utilisateur n'a pas à se plonger dans les calculs du solveur.

La thèse est organisée de la façon suivante. Tout d'abord, quelques résultats sur les clôtures et les arbres de preuve sont rappelés.

La première partie est consacrée à la recherche de solutions dans les CSP. Le chapitre 3 pose les définitions de CSP et de solutions. Les principales méthodes pour obtenir ces solutions sont ensuite décrites. Le chapitre 4 propose une formalisation de la réduction de domaine en termes ensemblistes.

La seconde partie du document se concentre sur les explications. Le chapitre 5 fait le tour des notions d'explications existantes. Les méthodes d'énumération basées sur celles-ci sont rappelées. Puis les problèmes de satisfaction de contraintes dynamiques sont décrits ainsi que l'utilisation des explications pour rendre leur résolution plus efficace. Enfin, le système `PaLM` entièrement basé sur les explications est brièvement évoqué. Dans le chapitre 6, les ensembles explicatifs sont tout d'abord définis. Un ensemble explicatif est un ensemble d'opérateurs ayant causé le retrait d'une valeur du domaine d'une variable. Ces ensembles sont totalement indépendants du calcul du fait que toute itération chaotique de ces opérateurs retirera toujours la valeur en question. Puis la notion d'arbre explicatif est proposée. Il est montré comment les ensembles explicatifs peuvent être extraits de ces arbres et comment l'énumération peut être prise en compte dans la construction de ces arbres. Le chapitre 7 est une preuve de correction des algorithmes de retrait de contrainte grâce à la notion d'ensemble explicatif. Ces ensembles permettent en effet de connaître les valeurs qui ont été retirées par une contrainte qui doit être relâchée.

La troisième partie de la thèse porte sur la mise au point en programmation par contraintes. Le chapitre 8 est un rappel des techniques existantes de mise point en programmation par contraintes. Les travaux de Meier (Meier, 1995) sont d'abord décrits, ils posent clairement la problématique et proposent l'outil GRACE (Meier, 1996) pour y répondre. Les outils réalisés dans le projet DiSCiPl (Deransart *et al.*, 2000) sont ensuite évoqués ainsi que d'autres travaux indépendants. Le chapitre 9 propose une approche déclarative de la mise au point de réponse manquante. Les notions de sémantique attendue sont définies. Le diagnostic déclaratif de réponse manquante consiste alors à rechercher une erreur dans l'arbre explicatif d'un tel symptôme. Il est montré comment inclure l'énumération dans cette méthode. L'explication d'échec est également définie.

Finalement, le chapitre 10 conclut la thèse et propose quelques perspectives.

Chapitre 2

Préliminaires

Sommaire

On rappelle ici quelques notions sur les définitions inductives utilisées dans la thèse. Plus de détails sont disponibles dans (Aczel, 1977; Arnold & Niwinski, 2001). On va considérer des ensembles E, F tels que $F \subseteq E$. L'inclusion ensembliste $\subseteq$ est un ordre partiel sur $\mathcal{P}(E)$.

2.1 Clôture par un opérateur

On considère un opérateur monotone $r : \mathcal{P}(E) \to \mathcal{P}(E)$ (i.e. si $X \subseteq Y$ alors $r(X) \subseteq r(Y)$).

L'opérateur monotone $X \mapsto F \cap r(X)$ a un plus grand point fixe qui est aussi le plus grand X tel que $X \subseteq F \cap r(X)$. Dès lors, l'ensemble $\{X \subseteq E \mid X \subseteq F \text{ et } X \subseteq r(X)\}$ a un unique *maximum* (c'est-à-dire un plus grand élément par rapport à $\subseteq$).

Définition 2.1 *La* clôture descendante *de F par r, notée* $\mathrm{CL}{\downarrow}(F, r)$, *est définie par* $\mathrm{CL}{\downarrow}(F, r) = max\{X \subseteq E \mid X \subseteq F \ et\ X \subseteq r(X)\}$.

De la même manière, $\min\{X \subseteq E \mid F \subseteq X \text{ et } r(X) \subseteq X\}$ existe. C'est le plus petit point fixe de l'opérateur $X \mapsto F \cup r(X)$.

Définition 2.2 *La* clôture ascendante *de F par r, notée* $\mathrm{CL}{\uparrow}(F, r)$, *est définie par* $\mathrm{CL}{\uparrow}(F, r) = min\{X \subseteq E \mid F \subseteq X \ et\ r(X) \subseteq X\}$.

Le *dual* de r, noté $\widetilde{r}$, est l'opérateur monotone défini par $\widetilde{r}(X) = \overline{r(\overline{X})}$ où $\overline{X} = E \backslash X$. Le résultat classique (Arnold & Niwinski, 2001) suivant peut alors être rappelé.

Lemme 2.1 $\mathrm{CL}{\uparrow}(\overline{F}, \widetilde{r}) = \overline{\mathrm{CL}{\downarrow}(F, r)}$

Preuve. $\mathrm{CL}{\uparrow}(\overline{F}, \widetilde{r}) = \min\{X \subseteq E \mid \overline{F} \subseteq X \text{ et } \widetilde{r}(X) \subseteq X\}$
$\mathrm{CL}{\uparrow}(\overline{F}, \widetilde{r}) = \min\{X \subseteq E \mid \overline{X} \subseteq F \text{ et } \overline{X} \subseteq r(\overline{X})\}$
$\mathrm{CL}{\uparrow}(\overline{F}, \widetilde{r}) = \overline{\max}\{\overline{X} \subseteq E \mid \overline{X} \subseteq F \text{ et } \overline{X} \subseteq r(\overline{X})\}$
$\mathrm{CL}{\uparrow}(\overline{F}, \widetilde{r}) = \overline{\mathrm{CL}{\downarrow}(F, r)}$ □

Le plus grand point fixe d'un opérateur monotone $r : \mathcal{P}(E) \to \mathcal{P}(E)$ est la limite d'une itération (qui peut être transfinie) de r donc $\mathrm{CL}{\downarrow}(F, r)$ est la limite d'une itération (qui peut être transfinie) de l'opérateur monotone $X \mapsto F \cap r(X)$. Il est facile de vérifier que $\mathrm{CL}{\downarrow}(F, r)$ est aussi la limite de la séquence définie par : $F_0 = F,\ F_{\alpha+1} = F_\alpha \cap r(F_\alpha), F_\alpha = \bigcap_{\beta<\alpha} F_\beta$ si l'ordinal α est *limite*.

De la même façon $\mathrm{CL}{\uparrow}(F, r)$ est la limite de la séquence définie par : $F_0 = F, F_{\alpha+1} = F_\alpha \cup r(F_\alpha), F_\alpha = \bigcup_{\beta<\alpha} F_\beta$ si l'ordinal α est *limite*.

2.2 Clôture par un ensemble d'opérateurs

Soit R un ensemble d'opérateurs monotones $\mathcal{P}(E) \to \mathcal{P}(E)$ et $F \subseteq E$.

En considérant $\rho : \mathcal{P}(E) \to \mathcal{P}(E)$, l'opérateur défini par $\rho(X) = \bigcap_{r \in R} r(X)$ et $\mathrm{CL}\!\downarrow\!(F, \rho)$, il est facile de voir que $\max\{X \subseteq E \mid X \subseteq F \text{ et } \forall r \in R, X \subseteq r(X)\}$ existe.

Définition 2.3 *La* clôture descendante *de F par R, notée* $\mathrm{CL}\!\downarrow\!(F, R)$*, est définie par* $\mathrm{CL}\!\downarrow\!(F, R) = max\{X \subseteq E \mid X \subseteq F \text{ et } \forall r \in R, X \subseteq r(X)\}$.

De la même façon en considérant l'opérateur $\rho : \mathcal{P}(E) \to \mathcal{P}(E)$, défini par $\rho(X) = \bigcup_{r \in R} r(X)$ et $\mathrm{CL}\!\uparrow\!(F, \rho)$, $\min\{X \subseteq E \mid F \subseteq X \text{ et } \forall r \in R, r(X) \subseteq X\}$ existe.

Définition 2.4 *La* clôture ascendante *de F par R, notée* $\mathrm{CL}\!\uparrow\!(F, R)$*, est définie par* $\mathrm{CL}\!\uparrow\!(F, R) = min\{X \subseteq E \mid F \subseteq X \text{ et } \forall r \in R, r(X) \subseteq X\}$.

Soit $\widetilde{R} = \{\widetilde{r} \mid r \in R\}$, l'ensemble des opérateurs duaux de R. Le lemme 2.1 s'étend alors aux ensembles d'opérateurs (Arnold & Niwinski, 2001).

Théorème 2.2 $\mathrm{CL}\!\uparrow\!(\overline{F}, \widetilde{R}) = \overline{\mathrm{CL}\!\downarrow\!(F, R)}$.

Preuve. Par le lemme 2.1. □

Les remarques sur l'égalité entre une clôture et la limite d'une itération peuvent s'appliquer pour un ensemble d'opérateurs monotones. Si R est un ensemble d'opérateurs monotones et $\rho(X) = \bigcap_{r \in R} r(X)$ alors $\mathrm{CL}\!\downarrow\!(F, R)$ est la limite de la séquence définie par :

- $F_0 = F$,
- $F_{\alpha+1} = F_\alpha \cap \rho(F_\alpha), F_\alpha = \bigcap_{\beta < \alpha} F_\beta$ si l'ordinal α est *limite*.

$\mathrm{CL}\!\uparrow\!(F, R)$ est la limite de la même séquence avec cette fois avec $\rho(X) = \bigcup_{r \in R} r(X)$.

2.3 Règles

Définition 2.5 *Une* règle *est une paire $h \leftarrow B$ avec $B \subseteq E$, appelé ensemble des* prémisses*, et $h \in E$ appelé la* conclusion*.*

On considère un ensemble de règles noté $\mathcal{R}$.

Définition 2.6 *L'opérateur r est* défini par $\mathcal{R}$ *si $r(X) = \{h \mid \exists B \subseteq X, h \leftarrow B \in \mathcal{R}\}$.*

Un tel opérateur est évidemment monotone.

Le *plus petit point fixe de* r est dit *inductivement défini par* $\mathcal{R}$.

Lemme 2.3 *Soit* $r : \mathcal{P}(E) \to \mathcal{P}(E)$ *un* opérateur monotone. *Il existe au moins un ensemble de règles* $\mathcal{R}$ *tel que* r *est* défini par $\mathcal{R}$.

> *Preuve.* Soit $\mathcal{R}$ l'ensemble de règles $h \leftarrow B$ tel que $B \subseteq E$ et $h \in r(B)$ (donc $h \in E$). Soit r' l'opérateur défini par $\mathcal{R}$. On vérifie que $r = r'$:
>
> $r(X) \subseteq r'(X)$ **:** si $h \in r(X)$ alors $h \leftarrow X \in \mathcal{R}$ donc $h \in r'(X)$.
>
> $r'(X) \subseteq r(X)$ **:** si $h \in r'(X)$ alors $\exists B \subseteq X$, $h \leftarrow B \in \mathcal{R}$ donc $h \in r(B)$ donc $h \in r(X)$.
>
> □

En général l'ensemble $\mathcal{R}$ utilisé dans la preuve n'est pas le plus intéressant. En effet, c'est le plus grand ensemble de règles définissant r. Il existe souvent d'autres ensembles de règles qui sont plus naturels par rapport à la signification de r.

2.4 Arbres

Soit E un ensemble. La concaténation est notée $\cdot$. Les arbres sur E sont définis à partir des notions suivantes :

- Un *nœud* est une séquence finie non vide $[l_1, \ldots, l_n]$, $(n \geq 1)$ avec $l_i \in E$. C'est un nœud *racine* si $n = 1$.
- L'*étiquette* d'un nœud $\Lambda \cdot [l]$ est définie par $\mathrm{label}(\Lambda \cdot [l]) = l$.
- Le *parent* d'un nœud non racine est défini par $\mathrm{parent}(\Lambda \cdot [l]) = \Lambda$.

Définition 2.7 *Un* arbre t *est un ensemble de nœuds non vide tel que :*

- *le parent de chaque nœud non racine de* t *est aussi un nœud de* t.
- *tous les nœuds de* t *ont le même premier élément. Ce premier élément est appelé la* racine *de* t *et est noté* $\mathrm{root}(t)$.

Soit T un ensemble d'arbres sur E et $l \in E$. L'ensemble $\{[l] \cdot \Lambda \mid \exists t \in T, \Lambda \in t\} \cup \{[l]\}$ est un arbre, que l'on note $\mathrm{cons}(l, T)$. Remarquons que le cas de base $\mathrm{cons}(l, \emptyset) = \{[l]\}$ a seulement un nœud étiqueté par l.

On considère les règles $h \leftarrow B$ comme précédemment (c'est-à-dire avec $h \in E$ et $B \subseteq E$).

Soit $\mathcal{R}$ un ensemble de règles. Les *arbres de preuve par rapport à* $\mathcal{R}$ sont définis inductivement comme suit.

Définition 2.8 $\text{cons}(h, T)$ *est un* arbre de preuve par rapport à $\mathcal{R}$ *si* T *est un ensemble d'*arbres de preuve par rapport à $\mathcal{R}$ *et* $h \leftarrow \{\text{root}(t) \mid t \in T\}$ *est une règle de* $\mathcal{R}$.

Remarquons le cas de base $T = \emptyset$ quand la règle est $h \leftarrow \emptyset$. Une telle règle est appelée un *fait*.

Pour tout opérateur r, on note $\mathcal{R}_r$ l'ensemble de règles le définissant. Le *plus petit point fixe* de r est $\{\text{root}(t) \mid t$ arbre de preuve par rapport à $\mathcal{R}_r\}$.

Soit R un ensemble d'opérateurs et $\mathcal{R} = \bigcup_{r \in R} \mathcal{R}_r$. Le *plus petit point fixe commun* des opérateurs de R vérifie le théorème suivant.

Théorème 2.4 *Soit* $F \subseteq E$ *et* $\mathcal{R}_F = \mathcal{R} \cup \{h \leftarrow \emptyset \mid h \in F\}$.
$\text{CL}\!\uparrow\!(F, R) = \{\text{root}(t) \mid t$ *arbre de preuve prà* $\mathcal{R}_F\}$.

Preuve. $\{\text{root}(t) \mid t$ arbre de preuve prà $\mathcal{R}_F\} \subseteq \min\{X \subseteq E \mid F \subseteq X$ et $\forall r \in R, r(X) \subseteq X\}$: par induction sur les arbres de preuve.
$\{\text{root}(t) \mid t$ arbre de preuve prà $\mathcal{R}_F\} \supseteq \min\{X \subseteq E \mid F \subseteq X$ et $\forall r \in R, r(X) \subseteq X\}$: Soit $Y = \{\text{root}(t) \mid t$ arbre de preuve prà $\mathcal{R}_F\}$. On vérifie facilement que $F \subseteq Y$ et $\forall r \in R, r(Y) \subseteq Y$. Dès lors, $Y \supseteq \min\{X \subseteq E \mid F \subseteq X$ et $\forall r \in R, r(X) \subseteq X\}$. □

Par les théorèmes 2.2 et 2.4, il a donc été montré que :

$\text{CL}\!\uparrow\!(F, R) = \{\text{root}(t) \mid t$ arbre de preuve prà $\mathcal{R}_F\} = \overline{\text{CL}\!\downarrow\!(\overline{F}, \widetilde{R})}$.

Autrement dit, le plus petit point fixe commun des opérateurs de R est l'ensemble des racines d'arbres de preuve par $\mathcal{R}$ qui est donc aussi le complémentaire du plus grand point fixe commun des opérateurs duaux de R.

Première partie

Résolution des CSP

Chapitre 3

CSP et Recherche de solutions

Sommaire

Dans ce chapitre sont présentées les notions de base de la programmation par contrainte sur domaines finis. Nous introduisons tout d'abord les notations utilisées dans cette thèse. Nous donnons les définitions de *Problème de Satisfaction de Contraintes* (CSP) et de solution d'un CSP. Nous survolons ensuite les principales méthodes de résolution utilisées pour obtenir ces solutions.

3.1 Notations et définitions

Quelques notations et définitions utilisées dans cette thèse sont introduites. Ces notations ensemblistes permettront une formalisation de la réduction de domaine en terme de point fixe et de clôture dans le chapitre 4, ce qui rendra possible une définition inductive des explications dans le chapitre 6.

On considère fixé un ensemble fini de symboles de *variables* V. On s'intéresse aux problèmes sur les domaines finis. Dans ce cadre chaque variable ne peut avoir qu'un ensemble fini de *valeurs* possibles. On considère fixé une famille $(D_x)_{x\in V}$ dont chaque D_x est un ensemble *fini* non vide, D_x est appelé le *domaine de la variable* x.

Nous aurons à considérer différentes *familles* $f = (f_i)_{i\in I}$. Une telle famille peut être identifiée à la *fonction* $i \mapsto f_i$, elle même identifiable à l'*ensemble* $\{(i, f_i) \mid i \in I\}$. Afin d'avoir des définitions simples et uniformes d'opérateurs monotones sur des ensembles quand on s'intéressera à la réduction de domaine, on utilise un ensemble qui est similaire à une base de Herbrand en programmation logique.

Définition 3.1 *On appelle* domaine *l'ensemble défini par* $\mathbb{D} = \bigcup_{x\in V}(\{x\} \times D_x)$.

> **Exemple 1** *On considère l'ensemble de variables* $V = \{X, Y, Z\}$ *avec les domaines de variables* $D_X = D_Y = D_Z = \{1, 2, 3\}$. *Le domaine est alors :* $\mathbb{D} = \{(X, 1), (X, 2), (X, 3), (Y, 1), (Y, 2), (Y, 3), (Z, 1), (Z, 2), (Z, 3)\}$.

La réduction de domaine a pour but de réduire l'ensemble des valeurs que peut prendre une variable. Par la suite, il sera alors nécessaire de considérer des sous-ensembles de $\mathbb{D}$ appelés *environnements*. Le domaine $\mathbb{D}$ est donc le plus grand environnement (et l'ensemble vide le plus petit).

On note $d|_W$ la *restriction* d'un environnement d à un ensemble de variables $W \subseteq V$, c'est-à-dire, $d|_W = \{(x, v) \in d \mid x \in W\}$. On dira que $d|_W$ est l'*environnement des variables* W dans d.

Les remarques suivantes peuvent alors être faites :

- pour $d \subseteq \mathbb{D}$, $d = d|_V = \bigcup_{x\in V} d|_{\{x\}}$;
- pour $d, d' \subseteq \mathbb{D}$, on a $d \subseteq d' \Leftrightarrow \forall x \in V, d|_{\{x\}} \subseteq d'|_{\{x\}}$.

Un problème de satisfaction de contraintes consiste à poser des contraintes sur les variables. Le résoudre (i.e. trouver une solution) revient à instancier les variables (c'est-à-dire à choisir une valeur dans le domaine de chaque variable) de telle sorte que les contraintes soient satisfaites. Nous aurons donc à considérer des environnements particuliers dans lesquels chaque variable ne possède qu'une valeur.

Une *instanciation* (aussi appelée *valuation* ou *tuple*) t est un environnement particulier pour lequel chaque variable apparaît une et une seule fois, c'est-à-dire $t \subseteq \mathbb{D}$ et $\forall x \in V, \exists v \in D_x, t|_{\{x\}} = \{(x,v)\}$. Un *tuple t sur* un ensemble de variables $W \subseteq V$ est défini par $t \subseteq \mathbb{D}|_W$ et $\forall x \in W, \exists v \in D_x, t|_{\{x\}} = \{(x,v)\}$. Une instanciation est donc un tuple sur V et un tuple sur un ensemble de variables peut être vu comme une instanciation restreinte à cet ensemble de variables. On parlera aussi d'*instanciation partielle* sur W pour un tuple sur W.

Dans le reste de la thèse, nous considérerons l'ensemble de variables V et le domaine $\mathbb{D}$ fixés.

3.2 Problème de Satisfaction de Contraintes

3.2.1 Définition

Les problèmes de satisfaction de contraintes ont été introduits dans (Montanari, 1974) sous le nom de *réseaux de contraintes*. On rappelle ici leur définition à la manière de (Tsang, 1993). Les notations introduites précédemment sont naturelles pour exprimer des notions basiques de contraintes impliquant seulement un sous-ensemble de variables.

Définition 3.2 *Un* Problème de Satisfaction de Contraintes *(CSP) sur* $(V, \mathbb{D})$ *est composé de :*

- *un ensemble fini de symboles de* contraintes C *;*
- *une fonction* var : $C \to \mathcal{P}(V)$, *qui associe à chaque symbole de contraintes l'ensemble des variables de la contrainte ;*
- *une famille* $(T_c)_{c \in C}$ *telle que pour chaque* $c \in C$, T_c *est un ensemble de tuples sur* $\mathrm{var}(c)$. T_c *est l'ensemble des* solutions de c.

Notons qu'un tuple $t \in T_c$ est équivalent à la famille $(v_x)_{x \in \mathrm{var}(c)}$ telle que $t = \{(x, v_x) \mid x \in \mathrm{var}(c)\}$.

Définition 3.3 *Un* support *de* $(x,v) \in \mathbb{D}$ *sur la contrainte* $c \in C$ *telle que* $x \in \mathrm{var}(c)$ *est un tuple* t *sur* $\mathrm{var}(c) \setminus \{x\}$ *tel que* $t \cup \{(x,v)\} \in T_c$.

Il existe donc autant de supports de (x,v) sur c que de solutions de c contenant (x,v).

Un CSP est dit *binaire* si chaque contrainte $c \in C$ est telle que $|\mathrm{var}(c)| \leq 2$, c'est-à-dire si chaque contrainte porte au plus sur deux variables.

3.2.2 Solutions

Pour un CSP donné, on s'intéresse à des instanciations particulières telles que toutes les contraintes soient satisfaites.

Définition 3.4 *Une instanciation t est une* solution *du CSP si* $\forall c \in C, t|_{\mathrm{var}(c)} \in T_c$.

> **Exemple 2** *En considérant $(V, \mathbb{D})$ de l'exemple précédent, le CSP suivant peut être défini :*
> - $C = \{Z = X + Y, X < Y\}$;
> - $\mathrm{var}(Z = X + Y) = \{X, Y, Z\}$ *et* $\mathrm{var}(X < Y) = \{X, Y\}$;
> - $T_{Z=X+Y}$ *est l'ensemble des trois tuples solutions de* $Z = X + Y$: $\{\{(X,1),(Y,1),(Z,2)\},\{(X,1),(Y,2),(Z,3)\},\{(X,2),(Y,1),(Z,3)\}\}$ *et* $T_{X<Y}$ *l'ensemble :* $\{\{(X,1),(Y,2)\}, \{(X,1),(Y,3)\}, \{(X,2),(Y,3)\}\}$.
>
> *Ce CSP possède une unique solution :* $\{(X,1),(Y,2),(Z,3)\}$.

La problématique de la programmation par contraintes est de prouver la satisfaisabilité (i.e. prouver qu'il existe une solution) ou de trouver une ou toutes les solutions d'un problème de satisfaction de contraintes. On parle d'*échec* quand le CSP n'a aucune solution.

Dans cette thèse, on supposera fixé un CSP $(C, \mathrm{var}, (T_c)_{c \in C})$ sur $(V, \mathbb{D})$ et on notera Sol son ensemble de solutions.

3.3 Recherche des solutions

Plusieurs méthodes sont possibles pour obtenir une ou l'ensemble des solutions d'un CSP (s'il en existe).

Une méthode naïve consiste à énumérer toutes les instanciations possibles, celles-ci étant ensuite testées sur les contraintes du problème. Cette méthode se révèle très coûteuse en pratique. On préfère alors des méthodes de recherche tentant de repérer des instanciations impossibles sans avoir à toutes les générer. Ces méthodes sont souvent basées sur une recherche arborescente. Chaque niveau de l'arbre correspond à une variable et chaque branche correspond à une valeur de la variable. Il existe donc autant de feuilles que d'instanciations possibles. Un tel arbre sera appelé *arbre de recherche complet*[1]. La méthode naïve consiste donc à tester chacune de ces feuilles.

Les méthodes abordées essayent de ne pas parcourir l'arbre dans son ensemble. Elles tentent de repérer les instanciations partielles (ou les valeurs) ne pouvant pas appartenir à une solution le plus rapidement possible afin de pouvoir élaguer l'arbre. Certaines de ces méthodes étant l'objet de parties de la thèse, elles ne sont ici que brièvement évoquées.

[1] Le terme d'arbre de recherche étant utilisé pour une de ces méthodes plus loin.

Notons qu'il existe aussi des méthodes dites de recherche locale (souvent incomplètes) qui sortent du cadre de ce travail et ne sont donc pas décrites ici.

3.3.1 Backtrack chronologique

Egalement appelée *backtrack standard*, le *backtrack chronologique* (Golomb & Baumert, 1965) consiste à parcourir l'arbre de recherche complet en profondeur d'abord. Les méthodes suivantes sont basées sur ce parcours de l'arbre. Après chaque instanciation (descente d'un niveau dans l'arbre), l'instanciation partielle est testée pour les contraintes dont toutes les variables sont instanciées. Si aucun échec n'est obtenu, la variable suivante est instanciée. Si un échec est détecté, alors une autre valeur est essayée pour la dernière variable instanciée. Si toutes les valeurs ont été testées, on revient sur l'instanciation de la variable précédente. Cette technique permet donc de repérer des échecs sans avoir à descendre systématiquement jusqu'aux feuilles.

3.3.2 Filtrage avant la recherche

Pour élaguer des branches de l'arbre de recherche avant de commencer, des techniques de filtrage sont souvent employées. Ces méthodes calculent un environnement pour lequel des valeurs ne pouvant participer à aucune solution ont été retirées. Cette réduction de domaine se fait selon une notion de consistance locale qui définie la précision avec laquelle elle est effectuée. Généralement, plus la précision est grande, plus le calcul est coûteux. Une fois l'environnement réduit obtenu, l'arbre de recherche peut être exploré par backtrack standard ou toute autre méthode. La réduction de domaine fait l'objet du chapitre suivant.

3.3.3 Méthodes prospectives

Basées sur le backtrack chronologique, les *méthodes prospectives* effectuent un filtrage pendant toute la recherche. Elles tentent de retirer des environnements des autres variables des valeurs qui ne peuvent pas appartenir à une solution avec les variables déjà instanciées. Ces techniques permettent donc de repérer des échecs plus tôt et de "réduire" l'arbre de recherche (puisque toutes les valeurs n'ont plus à être testées pour une variable). Deux principales techniques existent : le forward-checking et le maintien de consistance. Il faut noter que ces techniques ont été décrites d'abord pour les CSP binaires (c'est-à-dire où les contraintes portent au plus sur deux variables).

Le *forward-checking* (voir (Haralick & Elliott, 1980) pour les contraintes binaires et (Bessière *et al.*, 2002) pour une extension aux contraintes non binaires) se limite à réduire l'environnement par rapport aux contraintes portant sur la dernière variable instanciée. Dans le cas binaire, si la variable x est instanciée à v, alors pour toute contrainte $c \in C$ telle que var(c) $= \{x, y\}$, seuls les supports de (x, v) sont conservés dans l'environnement de y. Cela permet d'élaguer des branches de l'arbre de recherche complet. Il existe plusieurs extensions de cette technique aux CSP non binaires décrites dans (Bessière *et al.*, 2002).

Le *maintien de consistance* ((Sabin & Freuder, 1994) pour les CSP binaires) diffère en ce sens qu'il ne se limite pas à la dernière variable instanciée, mais effectue un filtrage complet après chaque instanciation. C'est donc un mélange de backtrack et de filtrage. L'arbre de recherche exploré par une telle technique est décrit dans la section 4.3. Dans le cas de l'arc-consistance, cet algorithme, appelé MAC, est le plus efficace en général (Sabin & Freuder, 1994; Bessière & Régin, 1996). La réduction de domaine selon une notion de consistance est une méthode clé utilisée par la plupart des solveurs. Nous la formalisons dans le chapitre suivant.

3.3.4 Heuristiques

Des heuristiques sur le choix des variables à instancier permettent également d'améliorer la recherche. En effet, l'ordre d'instanciation des variables a un effet important sur l'efficacité de l'algorithme (Dechter & Meiri, 1989) car celui-ci détermine quand les contraintes seront testées (dans le cas du forward-checking par exemple). On peut citer les travaux de Freuder (Freuder, 1982; Freuder, 1985), Dechter et Pearl (Dechter & Pearl, 1988) qui ont énoncé des propriétés garantissant l'existence d'ordres d'instanciation des variables n'entraînant aucun retour-arrière.

3.3.5 Méthodes rétrospectives

Avant tout il faut noter que les *méthodes rétrospectives* reposent sur le backtrack et peuvent être utilisées en plus des méthodes prospectives citées précédemment (et donc en plus de la réduction de domaine).

Le principe de ces méthodes est d'enregistrer de l'information lors de la recherche et de l'utiliser par la suite pour ne pas revenir sur la dernière instanciation quand celle-ci n'est pas responsable de l'échec. Alors que le backtrack standard revient sur la dernière variable instanciée lors d'un échec, les méthodes rétrospectives tentent d'identifier les causes de cet échec (grâce à des mécanismes proches des explications). Elles tirent parti de cette information soit pour sélectionner un meilleur point de retour, soit pour identifier des instanciations partielles ne participant à aucune solution et ainsi éviter de refaire certaines instanciations menant à un échec. Ces méthodes étant basées sur des mécanismes proches des explications, elles seront largement traitées dans le chapitre 5.

3.4 Le cadre de la thèse

On a rapidement présenté les différentes techniques utilisées pour la résolution de CSP. Certaines sont plus efficaces que d'autres. Entre autre, il semble admis que la réduction de domaine selon des notions de consistance est une des techniques les plus performantes. Elle est d'ailleurs utilisée dans la plupart des solveurs.

Dans cette thèse, on s'intéressera principalement à la réduction de domaine. La réduction de domaine ne permettant pas toujours d'obtenir les solutions du CSP, on l'inclura dans un mécanisme d'énumération dans un soucisde complétude.

Chapitre 4

Réduction de domaine

Sommaire

Les algorithmes de résolution de CSP (les *solveurs*), sont souvent basés sur une recherche systématique (par backtrack standard par exemple) mais utilisent, pour la plupart, des méthodes de réduction de domaines afin de diminuer la taille de l'arbre de recherche à explorer. La réduction de domaine peut-être utilisée comme filtrage, c'est-à-dire avant de faire une recherche systématique, ou tout au long de la recherche, c'est le cas du maintien de consistance.

Dans ce chapitre une formalisation de la réduction de domaine est proposée. Les notations ensemblistes permettent une formalisation basée sur l'application d'opérateurs à l'environnement. Le calcul de l'environnement réduit est décrit comme un calcul de clôture. Il est prouvé que si les opérateurs utilisés sont choisis judicieusement, les solutions du CSP appartiennent toutes à l'environnement obtenu. Une intégration de la réduction de domaine dans une recherche systématique permet d'obtenir les solutions. Enfin une illustration est proposée.

4.1 Programme et calcul de clôture

On considère le cadre de la *réduction de domaine* comme décrit dans (Benhamou, 1996; Codognet & Diaz, 1996; Van Emden, 1995; Van Hentenryck, 1989). Des cadres plus généraux[1] sont décrits dans (Apt, 2000; Montanari & Rossi, 1991). L'intérêt de notre approche repose sur les notations ensemblistes permettant d'une part de considérer la réduction de domaine comme un calcul de point fixe et d'autre part de définir naturellement les notions d'explication dans les chapitres suivants.

La réduction de domaine est dépendante d'une notion de consistance non exprimée par le CSP. La plus connue, l'arc-consistance, s'applique aux CSP binaires. Elle consiste à prendre une contrainte (donc sur deux variables) et à retirer, pour chaque variable apparaissant dans la contrainte, les valeurs ne pouvant pas en être solution, c'est-à-dire n'ayant pas de support dans l'environnement de l'autre variable de la contrainte. Ces valeurs sont dites localement inconsistantes. Cette opération, répétée plusieurs fois pour chaque contrainte du problème permet de n'avoir que des valeurs ayant des supports pour chaque contrainte où elles apparaissent. Plus généralement, les réductions peuvent être effectuées sur une seule variable d'une contrainte ou par rapport à plusieurs contraintes. Tout dépend de la consistance utilisée. Ces réductions sont donc représentées par l'application d'opérateurs monotones, ce qui rend le cadre proposé suffisamment général pour prendre en compte tout type de consistance. Ces opérateurs définissent un programme (le lien entre programme et CSP sera décrit à la fin du chapitre) et permettent de calculer une clôture (plus précisément, un plus grand point fixe).

L'ordre d'application des opérateurs joue un rôle important dans l'efficacité des solveurs et différentes stratégies sont utilisées en pratique. Les notions de run et d'itération utilisées ici permettent de faire abstraction de ces considérations.

[1] Ces travaux décrivent aussi la réduction de contraintes.

Les définitions établies sont illustrées par des exemples suivant le schéma `X in range`[2] (Deville *et al.*, 1991; Codognet & Diaz, 1996) dont s'inspire le solveur sur domaines finis GNU-PROLOG(Diaz & Codognet, 2000).

4.1.1 Programme

De la notion de consistance évoquée précédemment dépend la précision de la réduction de domaine. Plus la consistance est forte, plus le domaine pourra être réduit. Les valeurs retirées du domaine sont bien entendu des valeurs ne pouvant participer à aucune solution du CSP. Dans le formalisme proposé, les valeurs sont retirées par des opérateurs.

Définition 4.1 *Un* opérateur de consistance locale r *est une fonction monotone* $r : \mathcal{P}(\mathbb{D}) \to \mathcal{P}(\mathbb{D})$.

Il est important d'insister sur le fait que toute notion de consistance peut s'exprimer par de tels opérateurs. Plusieurs remarques peuvent être faites.

D'une part, la définition d'opérateur de consistance locale pourrait être plus précise (Ferrand *et al.*, 2002). Un opérateur de consistance locale s'applique à l'environnement dans son ensemble mais le résultat peut ne dépendre que d'une restriction de celui-ci à un sous-ensemble des variables. De plus, il peut ne retirer des valeurs que de l'environnement de certaines autres variables. Dans (Ferrand *et al.*, 2002), un opérateur de consistance locale r possède un type $(\text{in}(r), \text{out}(r))$ avec $\text{in}(r), \text{out}(r) \subseteq V$ et est alors défini par :

- $r(d)|_{V \setminus \text{out}(r)} = \mathbb{D}|_{V \setminus \text{out}(r)}$,
- $r(d) = r(d|_{\text{in}(r)})$

Intuitivement, l'opérateur r ne réduit que l'environnement des variables $\text{out}(r)$ (premier item) et cette réduction ne dépend que des environnements des variables de $\text{in}(r)$ (deuxième item). Il élimine des environnements de $\text{out}(r)$ des valeurs qui sont localement inconsistantes (selon la notion de consistance locale utilisée) avec les environnements de $\text{in}(r)$. Cependant cette précision n'est utile que dans un but pédagogique, nous l'utiliserons pour apporter quelques éclaircissements quand cela sera nécessaire.

D'autre part, les opérateurs de consistance locale généralisent les `X in range` inspirant GNU-PROLOG. Ceux-ci sont un cas particulier de nos opérateurs pour lesquels l'environnement d'une seule variable est calculé (i.e. $\text{out}(r)$ ne contient que la variable `X`). Celui-ci est calculé à partir des environnements des variables $\text{in}(r)$ présentes dans `range`.

[2]Classiquement, on parle du schéma `X in r`. Mais ce `r` n'ayant pas de rapport direct avec les opérateurs utilisés dans le formalisme et déjà notés r, il est remplacé par `range` pour lever toute ambiguïté.

Classiquement (Van Hentenryck, 1989; Benhamou, 1996; Fages *et al.*, 1998; Apt, 1999), les opérateurs utilisés possèdent les propriétés de monotonie, de contractance et d'idempotence. D'un point de vue théorique, l'idempotence est inutile, elle se révèle nécessaire en pratique pour gérer la queue de propagation du solveur. La contractance n'est nécessaire que pour le calcul afin d'obtenir un domaine réduit. Mais la réduction de domaine après l'application d'un opérateur peut s'obtenir en intersectant son résultat avec l'environnement courant. En général, les opérateurs de consistance locale ne sont pas des fonctions contractantes, comme montré plus tard pour définir leurs opérateurs duaux. Ces opérateurs sont intrinsèquement non contractants car ils ne considèrent qu'une partie du CSP, ce sont des opérateurs de consistance *locale*.

Définition 4.2 *L'*opérateur de réduction *associé à l'opérateur de consistance locale r est la fonction monotone et contractante $d \mapsto d \cap r(d)$.*

> **Exemple 3** *En* GNU-Prolog, *l'opérateur*[3] `X in 0..max(Y)-1` *indique que ne sont conservées pour* `X` *que les valeurs comprises entre 0 et la plus grande valeur de l'environnement de* `Y` *(désignée par* `max(Y)`*) moins 1. Pour obtenir un opérateur de réduction (i.e. un opérateur contractant), il suffit d'utiliser l'opération d'intersection (le* `&`*) et donc de remplacer* `X in 0..max(Y)-1` *par* `X in 0..max(Y)-1 & dom(X)` *où* `dom(X)` *représente l'environnement de* `X`*. L'environnement des autres variables n'est pas modifié.*

A un opérateur de consistance locale étant associé un seul opérateur de réduction, on parlera de l'un ou de l'autre. Ces considérations étant faites, on peut définir un programme.

Définition 4.3 *Un* programme *sur $(V, \mathbb{D})$ est un ensemble R d'opérateurs de consistance locale.*

Les opérateurs d'un programme sont choisis pour implanter les contraintes (nous verrons comment plus loin) selon une notion de consistance et vont permettre de calculer un environnement réduit. On suppose fixé un programme R sur $(V, \mathbb{D})$.

4.1.2 Clôture

On s'intéresse à des environnements particuliers : les points fixes communs des opérateurs de réduction $d \mapsto d \cap r(d)$, $r \in R$. Un point fixe de ces opérateurs est un environnement. Un tel environnement d vérifie $\forall r \in R$, $d = d \cap r(d)$, c'est-à-dire que les opérateurs ne peuvent enlever aucune valeur de cet environnement. On peut remarquer que $d = d \cap r(d)$ est équivalent à $d \subseteq r(d)$ utilisé dans la définition suivante.

[3] Les `X in range` sont appelés contraintes primitives dans (Codognet & Diaz, 1996).

Définition 4.4 *Soit un opérateur de consistance locale $r \in R$. Un environnement $d \subseteq \mathbb{D}$ tel que $d \subseteq r(d)$ est dit r-consistant. Un environnement $d \subseteq \mathbb{D}$ est dit R-*consistant *si $\forall r \in R$, d est r-consistant.*

La réduction de domaine à partir d'un environnement d par un ensemble d'opérateurs de consistance locale R calcule le plus grand point fixe commun des opérateurs de réduction associés à R à partir de d, c'est-à-dire le plus grand environnement tel qu'aucun opérateur ne puisse en retirer un élément. Comme annoncé dans le chapitre 2, ce plus grand point fixe commun existe et est unique.

Conformément à la définition 2.3 : la *clôture descendante* de d par R, notée $\mathrm{CL}\downarrow(d, R)$, est le plus grand $d' \subseteq \mathbb{D}$ tel que $d' \subseteq d$ et d' est R-consistant.

En général, on s'intéresse à la clôture de $\mathbb{D}$ par R (le calcul commence à partir de $\mathbb{D}$), mais il sera parfois nécessaire d'exprimer la clôture de sous-ensembles de $\mathbb{D}$ (environnements ou instanciations). Ce sera aussi utile pour prendre en compte les aspects dynamiques ou le maintien de consistance.

On remarque que $\mathrm{CL}\downarrow(d, \emptyset) = d$ et $\mathrm{CL}\downarrow(d, R) \subseteq \mathrm{CL}\downarrow(d, R')$ si $R' \subseteq R$. De plus la clôture descendante est monotone pour les environnements : pour $d, d' \subseteq \mathbb{D}$, si $d \subseteq d'$ alors $\mathrm{CL}\downarrow(d, R) \subseteq \mathrm{CL}\downarrow(d', R)$.

Etant donné que la clôture descendante est le plus grand environnement R-consistant, le lemme suivant est immédiat.

Lemme 4.1 *Si d est R-consistant alors $d \subseteq \mathrm{CL}\downarrow(\mathbb{D}, R)$.*

Preuve. Par la définition 2.3. □

Ce lemme sera utile pour établir le lien entre les solutions d'un CSP et la clôture calculée par un programme.

4.1.3 Calcul de la clôture

Dans le cadre de la réduction de domaine, la clôture descendante est l'ensemble le plus précis qui peut être calculé en utilisant un ensemble d'opérateurs de consistance locale. Comme énoncé dans le chapitre 2, il est facile de vérifier que pour tout $d \subseteq \mathbb{D}$, $\mathrm{CL}\downarrow(d, R)$ existe et peut être obtenue par itération de l'opérateur $d' \mapsto d' \cap \bigcap_{r \in R} r(d')$. Mais il existe une autre façon de calculer $\mathrm{CL}\downarrow(d, R)$ appelée *itération chaotique* qui va être rappelée.

La définition suivante est empruntée à Apt (Apt, 1999).

Définition 4.5 *Un* run *est une séquence* infinie *d'opérateurs de R, c'est-à-dire qu'un run associe à chaque $i \in \mathbb{N}$ ($i \geq 1$) un élément de R noté r^i.*

Le run indique un ordre d'application des opérateurs de réduction associés aux opérateurs de consistance locale. Une fois cet ordre donné, l'environnement va être successivement réduit par ces opérateurs.

Définition 4.6 *L'*itération descendante *de l'ensemble d'opérateurs de consistance locale R à partir du domaine $d \subseteq \mathbb{D}$ par rapport au run $r^1, r^2, \ldots$ est la séquence infinie $d^0, d^1, d^2, \ldots$ inductivement définie par :*

- *$d^0 = d$;*
- *pour chaque $i \in \mathbb{N}$, $d^{i+1} = d^i \cap r^{i+1}(d^i)$.*

Sa limite *est $\cap_{i \in \mathbb{N}} d^i$.*

De même que pour la définition de clôture, une itération peut commencer à partir d'un environnement d différent de $\mathbb{D}$.

L'opérateur $d' \mapsto d' \cap \bigcap_{r \in R} r(d')$ peut davantage réduire un environnement à chaque pas qu'un opérateur de consistance locale seul. Mais les calculs sont plus compliqués et certains peuvent se révéler inutiles. Les itérations descendantes sont alors préférées en pratique, elles procèdent par pas élémentaires, utilisant un seul opérateur à chaque fois.

Le but de l'utilisation des itérations est de calculer la clôture de l'environnement à réduire par l'ensemble d'opérateurs. Cependant certaines itérations ne permettent pas d'atteindre cette clôture (par exemple une itération n'appliquant jamais un des opérateurs). Pour assurer l'obtention de cette clôture, une notion d'équité doit être introduite. Elle permet de ne laisser de côté aucun opérateur de consistance locale.

Définition 4.7 *Un run est* équitable *si chaque $r \in R$ y apparaît infiniment, c'est-à-dire $\forall r \in R, \{i \mid r = r^i\}$ est infini.*

Dès lors les itérations descendantes par rapport à un run équitable peuvent être différenciées des autres.

Définition 4.8 *Une* itération chaotique *est une itération descendante par rapport à un run équitable.*

Le résultat de confluence (Apt, 2000; Cousot & Cousot, 1977) bien connu assure que chaque itération chaotique atteint la clôture. Dans le cadre dans lequel nous nous situons, à savoir les domaines finis, il est facile de voir que $\subseteq$ est un ordre bien fondé car $\mathbb{D}$ est un ensemble fini, et donc chaque itération chaotique commençant à partir de $d \subseteq \mathbb{D}$ (évidemment décroissante) est stationnaire, c'est-à-dire $\exists i \in \mathbb{N}$ tel que $\forall j \geq i, d^j = d^i$.

Théorème 4.2 *La limite de toute itération chaotique de l'ensemble d'opérateurs R à partir de $d \subseteq \mathbb{D}$ est la* clôture descendante *de d par R.*

Preuve. Soit $d^0, d^1, d^2, \ldots$ une itération chaotique de R à partir de d par rapport au run $r^1, r^2, \ldots$. Soit d^ω la limite de cette itération.
[$\mathrm{CL}\downarrow(d,R) \subseteq d^\omega$] Pour chaque i, $\mathrm{CL}\downarrow(d,R) \subseteq d^i$, par induction : $\mathrm{CL}\downarrow(d,R) \subseteq d^0 = d$. Supposons que $\mathrm{CL}\downarrow(d,R) \subseteq d^i$, $\mathrm{CL}\downarrow(d,R) \subseteq r^{i+1}(\mathrm{CL}\downarrow(d,R)) \subseteq r^{i+1}(d^i)$ par monotonie. Donc, $\mathrm{CL}\downarrow(d,R) \subseteq d^i \cap r^{i+1}(d^i) = d^{i+1}$.
[$d^\omega \subseteq \mathrm{CL}\downarrow(d,R)$] Il existe $k \in \mathbb{N}$ tel que $d^\omega = d^k$ car $\subseteq$ est un ordre bien fondé. Le run est équitable, dès lors d^k est un point fixe commun de l'ensemble d'opérateurs de réduction associé à R, donc $d^k \subseteq \mathrm{CL}\downarrow(d,R)$ (le plus grand point fixe commun). □

En pratique, un run est implanté par une *queue de propagation*. Celle-ci contient un ensemble d'opérateurs (l'ensemble R avant le début du calcul) parmi lesquels est choisi l'opérateur à appliquer à chaque étape. Une fois appliqué l'opérateur peut être retiré de la queue de propagation car il ne réduira pas l'environnement à la prochaine étape (les opérateurs de consistance locale étant idempotents en pratique). Par contre l'application de cet opérateur peut “réveiller” d'autres opérateurs qui doivent être rajoutés dans la queue de propagation. Bien entendu, pour un environnement d, tout opérateur r hors de la queue de propagation est tel que d est r-consistant (i.e. r ne réduit pas d). Autrement dit, seuls sont présents dans la queue de propagation des opérateurs susceptibles de réduire l'environnement. Pour optimiser le calcul, différentes stratégies sont utilisées pour déterminer les opérateurs qui doivent être présents dans la queue de propagation et pour déterminer lequel appliquer.

Les runs équitables fournissent un outil théorique permettant de faire abstraction de ces considérations. En effet par le théorème précédent, quelque soit le run utilisé (à partir du moment où celui-ci est équitable), l'itération chaotique atteint toujours la clôture. Dès lors que toute itération chaotique est stationnaire, en pratique, le calcul s'arrête quand un point fixe commun est atteint. Il existe une exception à cette règle qui est le cas où l'environnement d'une variable est vide. En effet, on sait dans ce cas qu'il ne peut y avoir de solutions (on parle d'*échec*), une optimisation consiste donc à stopper le calcul avant que la queue de propagation ne soit vide (i.e. avant que le clôture ne soit atteinte). L'échec est un cas intéressant en programmation par contrainte. Quand un échec se produit, le CSP est dit *sur-contraint*. Les explications peuvent apporter une aide non négligeable pour ces cas de figure. Nous reviendrons sur ces considérations.

4.2 Lien entre CSP et programme

Dans le chapitre précédent, les notions de CSP et de solution d'un CSP ont été définies. La réduction de domaine a été formalisée ici comme un calcul de clôture par un ensemble d'opérateurs. Il reste donc à faire le lien entre les deux, c'est-à-dire comment définir un programme à partir d'un CSP pour que la clôture de celui-ci soit une approximation des solutions du CSP.

En effet, les opérateurs doivent être choisis pour "implanter" les contraintes du CSP (selon une notion de consistance). En pratique, cette correspondance s'exprime par le fait que le programme est capable de tester n'importe quelle instanciation. c'est-à-dire, que si à chaque variable est associée une valeur, le programme doit être capable de répondre à la question : "cette instanciation est-elle une solution du CSP ?". Ce problème de décision fait intervenir deux notions : une notion de correction et une notion de complétude. Plus précisément, une solution du CSP doit être R-consistante (une solution ne doit pas être retirée par le programme) et toute instanciation R-consistante doit être une solution du CSP (si une instanciation n'est pas solution du programme alors elle ne doit pas être R-consistante).

Pour la suite, on considère un programme fixé R sur $(V, \mathbb{D})$ et on rappelle que le CSP $(C, \mathrm{var}, (T_c)_{c \in C})$ sur $(V, \mathbb{D})$ est lui aussi fixé et Sol est son ensemble de solutions.

4.2.1 Consistances usuelles

On rappelle brièvement les notions de consistances usuelles qui seront utilisées par la suite.

Définition 4.9 *Une contrainte $c \in C$ est* nœud-consistante *par rapport à d si et seulement si* $|\mathrm{var}(c)| \neq 1$ *ou si* $\mathrm{var}(c) = \{x\}$ *alors* $\forall (x, v) \in d, \{(x, v)\} \in T_c$.

Un CSP est nœud-consistant *par rapport à d si et seulement si $\forall c \in C$, c est nœud-consistante par rapport à d.*

La *consistance de nœud*, aussi appelée *1-consistance*, assure que chaque élément de l'environnement appartient à au moins une solution de toutes les contraintes unaires impliquant sa variable.

Définition 4.10 *Une contrainte $c \in C$ est* arc-consistante *par rapport à d si et seulement si* $|\mathrm{var}(c)| \neq 2$ *ou si* $\forall (x, v) \in d$ *tel que* $x \in \mathrm{var}(c), \exists (y, v') \in d$ *tel que* $\{(x, v), (y, v')\} \in T_c$.

Un CSP est arc-consistant *par rapport à d si et seulement si $\forall c \in C$, c est arc-consistante par rapport à d.*

L'arc-consistance, aussi appelée *2-consistance*, assure que chaque élément de l'environnement possède un support dans l'environnement pour chaque contrainte binaire dans laquelle sa variable apparaît. L'arc-consistance est une des consistances les plus utilisées en pratique. Elle peut être généralisée en *hyper arc-consistance* pour les contraintes non binaires.

Définition 4.11 *Une contrainte* $c \in C$ *est* hyper arc-consistante *par rapport à* d *si et seulement si* $\forall (x,v) \in d$ *tel que* $x \in \mathrm{var}(c)$, $\exists t \subseteq d$ *tel que* $t \cup \{(x,v)\} \in T_c$.

Un CSP est hyper arc-consistant *par rapport à* d *si et seulement si* $\forall c \in C$, c *est hyper arc-consistante par rapport à* d.

L'inconvénient de l'hyper arc-consistance est son coût élevé. Pour réduire celui-ci, la consistance de bornes[4] ne considère que les bornes du domaine d'une variable. On note $v_{\min(d,x)} = \min\{v \mid (x,v) \in d\}$ et $v_{\max(d,x)} = \max\{v \mid (x,v) \in d\}$.

Définition 4.12 *Une contrainte* $c \in C$ *est* consistante aux bornes *par rapport à* d *si et seulement si* $\forall x \in \mathrm{var}(c)$, $\exists t \subseteq d$ *tel que* $t \cup \{(x, v_{\min(d,x)})\} \in T_c$ *et* $\exists t' \subseteq d$ *tel que* $t' \cup \{(x, v_{\max(d,x)})\} \in T_c$.

Un CSP est consistant aux bornes *par rapport à* d *si et seulement si* $\forall c \in C$, c *est consistante de bornes par rapport à* d.

Une consistance plus forte que l'arc-consistance pour les contraintes binaires est la consistance de chemin[5].

Définition 4.13 *Un CSP est* chemin-consistant *si et seulement si :*

1. *il est arc-consistant et*
2. $\forall c \in C$ *telle que* $\mathrm{var}(c) = \{x,y\}$, *si* $\exists c', c'' \in C$ *telles que* $\mathrm{var}(c') = \{x,z\}$ *et* $\mathrm{var}(c'') = \{y,z\}$ *alors* $\forall (x,v) \in d, \exists (y,v'), (z,v'') \in d$ *tels que* $\{(x,v),(y,v')\} \in T_c$, $\{(x,v),(z,v'')\} \in T_{c'}$ *et* $\{(y,v'),(z,v'')\} \in T_{c''}$.

Il existe de nombreuses autres consistances, celles définies ci-dessus suffiront amplement pour notre propos.

4.2.2 Correction des opérateurs de consistance locale

Tout d'abord, les opérateurs implantant des contraintes ne doivent pas retirer de solutions de ces contraintes.

Définition 4.14 *Un opérateur de consistance locale* r *est dit* préservant *pour un ensemble de contraintes* C' *si, pour chaque instanciation* t, $(\forall c \in C', t|_{\mathrm{var}(c)} \in T_c) \Rightarrow t$ *est* r*-consistant.*

[4] En GNU-Prolog, la consistance de bornes est appelée *arc-consistance partielle*.

[5] La consistance de chemin définie ici est une version simplifiée qui entre dans le cadre de la réduction de domaine. La véritable consistance de chemin réalise aussi de la réduction de contrainte (non considérée dans cette thèse).

En particulier, si C' est l'ensemble des contraintes C du CSP alors on dit que r est *préservant pour le CSP*. Préserver un ensemble de contraintes est une propriété de correction des opérateurs, mais le terme "correct" étant utilisé plus loin dans un autre contexte, on préfère parler d'opérateur "préservant".

Exemple 4 *Poursuivant l'exemple précédent, l'opérateur r défini en* GNU-PROLOG *par* `X in 0..max(Y)-1` *est préservant pour la contrainte $X < Y$. En effet les tuples solutions de cette contrainte sont $\{(X,1),(Y,2)\}$, $\{(X,1),(Y,3)\}$ et $\{(X,2),(Y,3)\}$. Pour $(Y,2)$,* `max(Y)-1`$= 1$ *donc le tuple $\{(X,1),(Y,2)\}$ est r-consistant. Pour $(Y,3)$,* `max(Y)-1`$= 2$ *donc les deux tuples $\{(X,1),(Y,3)\}$ et $\{(X,2),(Y,3)\}$ sont r-consistants.*

Dans le cas de l'arc-consistance, ou plus généralement de l'hyper arc-consistance, les opérateurs sont choisis pour implanter une contrainte du CSP (chaque opérateur est donc préservant pour une contrainte). Dans des cas plus généraux, les opérateurs ne sont pas forcément préservants pour une contrainte mais pour un ensemble de contraintes. C'est le cas par exemple de la consistance de chemin qui s'intéresse aux solutions de trois contraintes binaires.

En choisissant tous les opérateurs de la sorte, c'est-à-dire préservant les solutions d'une ou plusieurs contraintes, il est montré qu'aucune solution du CSP n'est retirée.

Lemme 4.3 *Si l'opérateur de consistance locale r est préservant pour l'ensemble de contraintes C' et $C' \subseteq C''$ alors r est préservant pour l'ensemble de contraintes C''.*

Preuve. $\forall c \in C'', t|_{\mathrm{var}(c)} \in T_c$ donc $\forall c \in C', t|_{\mathrm{var}(c)} \in T_c$ donc t est r-consistant. □

On considère alors le cas où $C'' = C$, l'ensemble des contraintes du CSP.

Corollaire 4.4 *Si l'opérateur de consistance locale r est préservant pour l'ensemble de contraintes $C' \subseteq C$ alors r est préservant pour le CSP.*

Pour implanter un CSP, le programme est donc choisi tel que chaque opérateur soit préservant pour un sous-ensemble de contraintes.

Définition 4.15 *On dira que* le programme R est préservant pour le CSP *si $\forall r \in R$, r est préservant pour le CSP.*

En particulier, le programme vide (i.e. sans opérateur) est préservant pour le CSP. Il ne retire aucune valeur donc aucune solution.

4.2.3 Complétude des opérateurs de consistance locale

Une notion de complétude est également utilisée afin de choisir les opérateurs "implantant" un CSP. Elle permet de rejeter des instanciations qui ne sont pas solutions des contraintes.

Définition 4.16 *Un ensemble d'opérateurs de consistance locale R' est* complet *par rapport à une contrainte c si, pour chaque instanciation t, t est R'-consistant $\Rightarrow$ $t|_{\mathrm{var}(c)} \in T_c$.*

En particulier, si R' est l'ensemble d'opérateurs du programme (c'est-à-dire si $R' = R$), on dit que le programme est *complet par rapport à la contrainte c*.

> **Exemple 5** *Poursuivant toujours notre exemple, chacun des opérateurs définis en* GNU-Prolog *par* `X in 0..max(Y)-1` *et* `Y in min(X)+1..infinity` *est complet*[6] *par rapport à la contrainte $X < Y$.* `infinity` *désigne non pas l'infini puisque on considère les domaines finis mais la plus grande valeur autorisée pour une variable en* GNU-Prolog *et* `min(X)` *désigne la plus petite valeur de l'environnement de X.*

Il faut insister sur le fait que la définition précédente fait intervenir une instanciation et pas un environnement.

> **Exemple 6** *Soit $d = \{(X,0),(X,1),(Y,1),(Y,2)\}$, les solutions de $X < Y$ (pour d) sont $\{(X,0),(Y,1)\}$, $\{(X,0),(Y,2)\}$ et $\{(X,1),(Y,2)\}$. Pour l'opérateur* `X in 0..max(Y)-1` *de l'exemple précédent, d est r-consistant, cependant l'instanciation $\{(X,1),(Y,1)\}$ n'est pas solution de la contrainte.*

En choisissant les opérateurs de cette manière, on montre qu'on assure ainsi de rejeter les instanciations qui ne sont pas solutions du CSP.

Lemme 4.5 *Si l'ensemble d'opérateurs de consistance locale R' est complet par rapport à une contrainte c et si $R' \subseteq R''$ alors l'ensemble d'opérateurs de consistance locale R'' est complet par rapport à la contrainte c.*

> *Preuve.* Si t est R''-consistant alors t est R'-consistant car $R' \subseteq R''$ or R' est complet par rapport à c donc $t|_{\mathrm{var}(c)} \in T_c$. □

On considère le cas où $R'' = R$, c'est-à-dire où R'' est le programme.

Corollaire 4.6 *Si l'ensemble d'opérateurs de consistance locale $R' \subseteq R$ est complet par rapport à une contrainte c alors le programme R est complet par rapport à c.*

---

[6]On considère les solutions de la contrainte sur l'ensemble d'entiers de 0 à `infinity` en GNU-Prolog.

Un programme implantant un CSP est donc choisi de telle façon qu'il soit complet pour chaque contrainte du CSP.

Définition 4.17 *On dira que* le programme R est complet par rapport au CSP *si* $\forall c \in C$*,* R *est complet par rapport à* c*.*

4.2.4 Solutions et clôture

On va considérer un ensemble de solutions $S \subseteq \mathrm{Sol}$ du CSP. Sa projection sur $\mathbb{D}$ est $\bigcup S = \bigcup_{t \in S} t$. S est donc un ensemble d'instanciations et $\bigcup S$ un environnement dont chaque élément appartient à une solution. Les résultats établis seront alors plus généraux et pourront être utilisés pour la mise au point dans le chapitre 9.

On va considérer des opérateurs préservants pour le CSP. La complétude n'est pas nécessaire ici. En effet, c'est la propriété de préservation qui assure de garder des solutions, la propriété de complétude permet quand à elle de retirer des éléments qui ne participent pas aux solutions.

Pour faire le lien entre solutions du CSP et clôture du programme, on va montrer que la projection de l'ensemble des solutions du CSP est incluse dans la clôture calculée par le programme.

Lemme 4.7 *Soit* $S \subseteq \mathrm{Sol}$*, si* r *est préservant pour le CSP alors* $\bigcup S$ *est* r*-consistant.*

Preuve. $\forall t \in S, t \subseteq r(t)$ donc $\bigcup S \subseteq \bigcup_{t \in S} r(t)$. Or, $\forall t \in S, t \subseteq \bigcup S$ donc $\forall t \in S, r(t) \subseteq r(\bigcup S)$. □

Plus généralement, si un opérateur de consistance locale est préservant pour le CSP alors il ne retire pas ses solutions d'un environnement les contenant (ça paraît naturel).

On considère que le programme R fixé est préservant pour le CSP. Le lien entre les solutions du CSP et la clôture calculée par le programme peut alors être établi.

Lemme 4.8 *Si* $S \subseteq \mathrm{Sol}$ *alors* $\bigcup S \subseteq \mathrm{CL}{\downarrow}(\mathbb{D}, R)$*.*

Preuve. par les lemmes 4.1 et 4.7. □

Finalement, en considérant la projection $\bigcup \mathrm{Sol}$ de l'ensemble des solutions du CSP sur $\mathbb{D}$, on obtient le résultat souhaité.

Corollaire 4.9 $\bigcup \mathrm{Sol} \subseteq \mathrm{CL}{\downarrow}(\mathbb{D}, R)$*.*

On a donc prouvé que si les opérateurs sont choisis correctement (i.e préservants pour le CSP) alors les solutions du CSP ne sont pas retirées par le programme et appartiennent donc à l'environnement calculé par celui-ci (la clôture).

Comme on le voit dans l'exemple suivant, l'égalité entre $\bigcup \mathrm{Sol}$ et $\mathrm{CL}\downarrow(\mathbb{D}, R)$ peut être assurée dans le cas très particulier où la notion de consistance utilisée est suffisamment forte et où les opérateurs sont à la fois préservants et complets. Par exemple pour un CSP binaire à k variables, l'égalité est assurée par la k-consistance. Sachant que pour $k > 2$, les calculs sont souvent trop coûteux, les solveurs se limitent à la 2-consistance (i.e. l'arc-consistance).

Exemple 7 *Considérons le CSP défini sur* $(V, \mathbb{D})$ *avec :*

- $V = \{X, Y, Z\}$,
- $\mathbb{D} = \{(X,0),(X,1),(Y,0),(Y,1),(Z,0),(Z,1)\}$
- *et les contraintes* $X \neq Y$, $X \neq Z$ *et* $Y \neq Z$.

On voit facilement que ce CSP n'a pas de solution.

En appliquant des opérateurs d'arc-consistance, aucun élément ne peut être retiré du domaine. Pour la contrainte $X \neq Y$, $(X,0)$ *a le support* $\{(Y,1)\}$ *et* $(X,1)$ *a le support* $\{(Y,0)\}$. *La même remarque peut être faite pour les autres contraintes. La clôture est donc* $\mathbb{D}$ *lui même.*

En appliquant la consistance de chemin (i.e. la 3*-consistance), on cherche à compléter toute instanciation de deux éléments localement consistante avec un troisième élément tel que la nouvelle instanciation soit localement consistante. Par exemple,* $\{(X,0),(Y,1)\}$ *est localement consistante (car* $X \neq Y$*), mais ne peut pas être étendue avec une valeur pour* Z *car* $\{(X,0),(Z,0)\}$ *est localement inconsistante (pour* $X \neq Z$*) et* $\{(Y,1),(Z,1)\}$ *est localement inconsistante (pour* $Y \neq Z$*).* $\{(X,0),(Y,1)\}$ *n'est donc pas chemin-consistante. Aucun tuple ne pouvant satisfaire ces trois contraintes,* $(X,0)$ *est retiré de l'environnement, de même que* $(X,1)$ *et les autres éléments du domaine. La clôture, dans ce cas, est vide et est donc bien égale à la projection des solutions.*

La contrepartie d'une consistance forte est son coût élevé. Ces consistances sont souvent simulées à moindre coût dans des cas particuliers par l'intermédiaire de *contraintes globales*. Pour l'exemple précédent, certains solveurs utilisent la contrainte globale `all_different([X,Y,Z])` qui réduit ici l'environnement à l'ensemble vide en temps polynômial (Régin, 1999). On peut noter que le but des contraintes globales n'est pas seulement l'utilisation de consistances plus fortes. Elles ont aussi pour intérêt d'exprimer facilement des conditions globales difficilement traduisibles avec les contraintes classiques.

Pour conclure, il faut rappeler que la clôture descendante est un sur-ensemble (une "approximation") de $\bigcup \mathrm{Sol}$ qui est elle même la projection de Sol. Mais la clôture descendante est l'ensemble le plus précis qui peut être calculé en utilisant un ensemble d'opérateurs de consistance locale dans le cadre de la réduction de domaine. Pour approcher davantage les solutions, il est souvent nécessaire d'utiliser une recherche systématique.

4.3 Recherche des solutions

On montre ici comment inclure la réduction de domaine dans un mécanisme de recherche de solutions. Plus précisément, on montre comment obtenir les solutions d'un CSP par une méthode de maintien de consistance. On considère toujours fixés un CSP et un programme R tels que R est préservant et complet pour le CSP.

La méthode est la suivante. La clôture de l'environnement $\mathbb{D}$ par le programme R est calculée. Si un échec ou une solution est obtenu, on s'arrête. Sinon, on considère l'environnement obtenu par cette réduction de domaine. Une variable est choisie (contenant plusieurs éléments dans son environnement) et on considère autant de sous-problèmes qu'il existe de valeurs dans son environnement. Pour chacun de ces sous-problèmes, l'environnement des autres variables est conservé, mais une seule des valeurs de l'environnement de la variable choisie est gardée. Il y a donc un sous-problème par valeur de la variable. La clôture de chacun de ces environnements peut être calculée. Le processus est réitéré jusqu'à obtenir un échec ou une solution.

Cette recherche peut être représentée par un arbre, à chaque nœud une clôture est calculée. Le passage d'un nœud à ses fils se fait en restreignant l'environnement d'une variable à une valeur (différente pour chaque fils). On parle de *labeling* ou d'*énumération*.

On ne considère pas ici le splitting qui est une généralisation de l'énumération. Cette technique est surtout utilisée pour les domaines continus (où l'énumération est impossible). Le splitting consiste à partitionner le domaine d'une variable en plusieurs morceaux sans forcément restreindre l'environnement de cette variable à une seule valeur.

4.3.1 Enumération et partition

On définit un nouveau type d'opérateur permettant de restreindre l'environnement d'une variable à une valeur.

Définition 4.18 *L'*opérateur d'énumération *sur* $(x, v) \in \mathbb{D}$ *est l'opérateur constant* $r : \mathcal{P}(\mathbb{D}) \rightarrow \mathcal{P}(\mathbb{D})$ *défini par :* $r(d) = \mathbb{D}|_{V \setminus \{x\}} \cup \{(x, v)\}$.

L'opérateur d'énumération sur (x, v) peut être vu comme un opérateur de consistance locale particulier préservant et complet pour la contrainte $x = v$. Ajouter l'opérateur d'énumération sur (x, v) au programme R revient donc à considérer le CSP auquel est ajouté la contrainte $x = v$. Pour ne pas perdre de solutions, il faut considérer toutes les valeurs de l'environnement de x et donc autant d'opérateurs que de valeurs dans cet environnement.

Définition 4.19 *L'*ensemble partition *de l'environnement* $d \subseteq \mathbb{D}$ *sur la variable* $x \in V$ *est un ensemble d'opérateurs d'énumération* P *tel que :*

- $\exists(x, v), (x, v') \in d|_{\{x\}}, v \neq v'$
- $\forall r \in P, \exists(x, v) \in d|_{\{x\}}, r(d)|_{\{x\}} = \{(x, v)\}$
- $\forall(x, v) \in d|_{\{x\}}, \exists r \in P, r(d)|_{\{x\}} = \{(x, v)\}$

Le premier item rend nécessaire la présence d'au moins deux valeurs dans l'environnement de la variable x afin d'assurer une réelle réduction de l'environnement de x. Le domaine $\mathbb{D}$ étant fini, cela permet de considérer la définition d'arbres de recherche *finis*. Le second item assure que les opérateurs d'énumération portent sur les valeurs de l'environnement de x. Le troisième item assure l'existence d'un opérateur d'énumération pour chaque valeur de l'environnement de x.

Des propriétés importantes des opérateurs d'énumération peuvent alors être établies.

Lemme 4.10 *Si t est une instanciation telle que $t \subseteq d$ et P un ensemble partition de d sur $x \in V$ alors $\exists r \in P$ tel que t est r-consistant.*

Preuve. $t|_{V \setminus \{x\}} \subseteq \mathbb{D}|_{V \setminus \{x\}} = r(d)|_{V \setminus \{x\}}$. Soit $v \in D_x$ tel que $t|_{\{x\}} = \{(x, v)\}$. Donc $(x, v) \in d$ donc $\exists r \in P$ tel que $r(d)|_{\{x\}} = \{(x, v)\} = t|_{\{x\}}$ donc $t \subseteq r(t)$. □

Autrement dit, quand une partition est effectuée sur un environnement, aucune instanciation ne disparaît, donc aucune solution possible. Le lemme suivant sera utile pour considérer les solutions d'un arbre de recherche.

Lemme 4.11 *Soit t une instanciation et N un ensemble d'opérateurs d'énumération. Si $t \subseteq \mathrm{CL}\downarrow(\mathbb{D}, N)$ alors t est N-consistant.*

Preuve. par l'absurde : si t n'est pas N-consistant alors $\exists r \in N$ tel que $t \not\subseteq r(t)$ donc $\exists(x, v) \in t$ tel que $(x, v) \notin r(t)$. r est une fonction constante donc $r(\mathbb{D}) = r(t)$ donc $(x, v) \notin r(\mathbb{D})$ donc $(x, v) \notin \mathrm{CL}\downarrow(\mathbb{D}, N)$ donc $t \not\subseteq \mathrm{CL}\downarrow(\mathbb{D}, N)$. □

Enfin, on montre que calculer la clôture du domaine par un ensemble d'opérateur de consistance locale et d'énumération est équivalent à calculer la clôture par l'ensemble d'opérateurs de consistance locale du domaine réduit par les opérateurs d'énumération.

Lemme 4.12 *Soit R un ensemble d'opérateurs de consistance locale et N un ensemble d'opérateurs d'énumération.* $\mathrm{CL}\downarrow(\mathbb{D}, R \cup N) = \mathrm{CL}\downarrow(\mathrm{CL}\downarrow(\mathbb{D}, N), R)$.

Preuve. Car les opérateurs d'énumération sont des fonctions constantes.
□

Ce lemme sera particulièrement utile pour traiter l'énumération dans le chapitre sur la mise au point. Etant donné que les opérateurs d'énumération sont des fonctions constantes, on peut remarquer que $\mathrm{CL}\downarrow(\mathbb{D}, N) = \bigcap_{r \in N} r(\mathbb{D})$. Autrement dit, calculer une clôture par un ensemble d'opérateurs d'énumération peut se faire en appliquant une seule fois chacun de ces opérateurs.

4.3.2 Arbre de recherche

Une recherche par réduction de domaine et énumération peut se représenter par un arbre de recherche. Pour passer d'un nœud à l'un de ses fils, un opérateur d'énumération est appliqué. L'ensemble des opérateurs d'énumération permettant de passer d'un nœud à ses fils forment une partition de l'environnement d'une des variables. Un nœud est alors défini par l'ensemble des opérateurs d'énumération qui ont été utilisés pour aller de la racine jusqu'à lui, autrement dit, c'est une séquence d'opérateurs d'énumération. A cette séquence peut bien sûr être associé l'ensemble d'opérateurs qui y apparaissent.

Définition 4.20 *Un* état de recherche *est un ensemble d'opérateurs d'énumération.*

Chaque état de recherche étant un ensemble d'opérateurs d'énumération, la clôture du domaine par cet ensemble et le programme peut être calculée. On peut alors distinguer deux types d'états particuliers. Tout d'abord ceux pour lesquels l'environnement d'une des variable est vide.

Définition 4.21 *Un état de recherche N est dit* état échec *s'il existe une variable $x \in V$, $\mathrm{CL}\downarrow(\mathbb{D}, R \cup N)|_{\{x\}} = \emptyset$.*

L'environnement d'une variable étant vide, cette clôture ne peut donc contenir aucune solution du CSP. Cet état correspond donc bien à un échec.

Le deuxième type d'état particulier correspond à l'obtention d'une solution. Cet état est tel que la clôture obtenue est une instanciation.

Définition 4.22 *Un état de recherche N est dit* état succès *si pour toute variable $x \in V$, $\mathrm{CL}\downarrow(\mathbb{D}, R \cup N)|_{\{x\}} = \{(x, v)\}$.*

Contrairement à un état échec, non seulement aucune variable ne possède un environnement vide, mais surtout, comme le montre le lemme suivant, l'instanciation obtenue est une solution du CSP. Il est important de rappeler que le programme R fixé est complet pour le CSP (sans cette remarque, l'instanciation pourrait ne pas être solution du CSP).

Lemme 4.13 *Si N est un état succès alors* $\mathrm{CL}\downarrow(\mathbb{D}, R \cup N) \in \mathrm{Sol}$.

Preuve. soit $s = \mathrm{CL}\downarrow(\mathbb{D}, R \cup N)$ donc $s = \mathrm{CL}\downarrow(s, R \cup N)$ donc s est $(R \cup N)$-consistant donc s est R-consistant donc $\forall c \in C, s|_{\mathrm{var}(c)} \in T_c$ car le programme est complet donc $s \in \mathrm{Sol}$. □

Les arbres de recherche sont des arbres étiquetés par ces états de recherche. Aux états succès et échecs sont associés des nœuds particuliers, ce sont les feuilles de l'arbre. En effet quand une solution ou un échec est obtenu, il n'est pas nécessaire de poursuivre dans cette branche de l'arbre.

Définition 4.23 *Un* arbre de recherche *pour un programme R est un arbre tel que :*

- *ϵ est sa racine et $\emptyset$ l'étiquette de cette racine ;*
- *pour tout nœud n étiqueté par N :*
  - *soit n est un état succès et son étiquette est appelée feuille succès ;*
  - *soit n est un état échec et son étiquette est appelée feuille échec ;*
  - *soit il possède k fils étiquetés par $N \cup \{r_i\}$ pour $1 \le i \le k$ tels que $\bigcup_{1 \le i \le k} r_i$ est un ensemble partition de* $\mathrm{CL}\downarrow(\mathbb{D}, R \cup N)$.

Un opérateur d'énumération restreignant l'environnement d'une variable à un seul élément et étant donné qu'un ensemble partition contient au moins deux éléments, chaque nœud contient au plus un opérateur d'énumération sur chaque variable. De plus, l'ensemble des variables et le domaine étant finis, les arbres de recherche sont aussi finis.

Pour un programme R, il peut exister plusieurs arbres de recherche selon l'ordre dans lequel les variables sont choisies pour l'énumération. Il existe différentes heuristiques de choix de la variable à instancier. Par exemple :

- sélectionner la variable ayant le plus petit nombre d'éléments dans son environnement ;
- sélectionner la variable apparaissant dans le plus grand nombre de contraintes ;
- choisir la variable qui a la plus petite valeur dans son environnement ;
- ...

Une variable sélectionnée, il existe aussi des heuristiques pour choisir la branche à explorer en priorité. Mais ceci sort du cadre de cette thèse.

Il faut insister sur le fait que les arbres de recherche définis ici sont tels qu'une réduction de domaine est effectuée après chaque ajout d'un opérateur d'énumération (i.e. à chaque état de recherche N, $\mathrm{CL}\downarrow(\mathbb{D}, R \cup N)$ est calculée). Cette méthode est une recherche par backtrack standard avec maintien de consistance si l'arbre est parcouru en profondeur d'abord. La clôture étant monotone, en pratique pour un nœud donné, celle-ci n'est pas recalculée à partir du domaine $\mathbb{D}$ mais à partir de la clôture déjà calculée pour le père du nœud. Dans le cas de l'arc-consistance, cet algorithme (appelé MAC) est plus efficace que le forward-checking en général (Sabin & Freuder, 1994).

La définition d'arbre de recherche pourrait être modifiée pour obtenir les arbres de recherche correspondant à d'autres types de recherche. Le forward-checking pourrait être obtenu en remplaçant le calcul de clôture par une application de certains opérateurs. Ce calcul est donc moins coûteux à chaque nœud. Cependant, il est moins précis et les échecs sont détectés moins rapidement.

Deux remarques peuvent être faites à propos des arbres de recherche complets évoqués dans le chapitre précédent. D'une part, un arbre de recherche complet est un arbre de recherche pour le programme vide (programme qui n'est complet que pour le CSP dont toute instanciation est solution). D'autre part (et en conséquence), chaque feuille de cet arbre est une feuille succès.

4.3.3 Solutions

Le but de l'utilisation du labeling étant d'obtenir les solutions, on montre que les clôtures calculées aux feuilles succès de l'arbre de recherche sont exactement les solutions du CSP.

Définition 4.24 *On appelle* solutions d'un arbre de recherche $\mathcal{A}$ *associé au programme* R *l'ensemble défini par* $\mathrm{Sol}_{\mathcal{A}} = \{\mathrm{CL}\downarrow(\mathbb{D}, R \cup N) \mid N$ *est un état succès de* $\mathcal{A}\}$.

On prouve que l'ensemble des solutions d'un arbre de recherche associé à R est exactement l'ensemble des solutions du CSP. Il est important de rappeler que le programme est préservant et complet pour le CSP.

Théorème 4.14 *Si* $\mathcal{A}$ *est un arbre de recherche associé au programme* R *alors* $\mathrm{Sol}_{\mathcal{A}} = \mathrm{Sol}$.

Preuve.

$\mathrm{Sol}_{\mathcal{A}} \subseteq \mathrm{Sol}$: soit $s \in \mathrm{Sol}_{\mathcal{A}}$ donc $\exists N, s = \mathrm{CL}\downarrow(\mathbb{D}, R \cup N)$ donc par le lemme 4.13 $s \in \mathrm{Sol}$.

$\mathrm{Sol} \subseteq \mathrm{Sol}_{\mathcal{A}}$: soit $s \in \mathrm{Sol}$. Par induction sur les nœuds de l'arbre de recherche :
$s \subseteq \mathrm{CL}\downarrow(\mathbb{D}, R)$
on suppose qu'il existe un nœud étiqueté par N qui soit tel que $s \subseteq \mathrm{CL}\downarrow(\mathbb{D}, R \cup N)$ et on prouve que soit N est un état succès, soit ce nœud a un fils étiqueté par $N \cup \{r\}$ tel que $s \subseteq \mathrm{CL}\downarrow(\mathbb{D}, R \cup N \cup \{r\})$.

- de manière évidente, N ne peut pas être un état échec
- si N est un état succès alors $s = \mathrm{CL}\downarrow(\mathbb{D}, R \cup N)$ donc $s \in \mathrm{Sol}_{\mathcal{A}}$
- sinon, soit P l'ensemble partition de $\mathrm{CL}\downarrow(\mathbb{D}, R \cup N)$ sur x utilisé pour ses fils. Par le lemme 4.10, $\exists r \in P$ tel que s est r-consistant. $s \subseteq \mathrm{CL}\downarrow(\mathbb{D}, R \cup N)$ donc $s \subseteq \mathrm{CL}\downarrow(\mathbb{D}, N)$, par le lemme 4.11, s

est N-consistant. s est une solution du CSP et R est préservant pour le CSP donc s est R-consistant. Donc s est $(R \cup N \cup \{r\})$-consistant. Donc $s \subseteq \mathrm{CL}\!\downarrow\!(\mathbb{D}, R \cup N \cup \{r\})$.

□

Un arbre de recherche permet donc d'obtenir les solutions d'un CSP à la condition que le programme soit préservant et complet pour le CSP.

La propriété de préservation permet d'avoir $\mathrm{Sol} \subseteq \mathrm{Sol}_{\mathcal{A}}$. En effet, si un opérateur n'est pas préservant, il peut retirer des solutions. La propriété assure l'appartenance des solutions aux clôtures calculées.

La propriété de complétude permet d'avoir $\mathrm{Sol}_{\mathcal{A}} \subseteq \mathrm{Sol}$. Prenons l'exemple du programme vide. Celui-ci est préservant pour tout CSP mais en général non complet (à part pour les CSP où toute instanciation est solution).

4.4 Application

On propose ici une illustration en GNU-Prolog du formalisme décrit précédemment sur un problème de conférence inspiré de (Cousin, 1993). Dans ce langage, les notions de consistance utilisées sont l'hyper arc-consistance et la consistance de bornes (Marriott & Stuckey, 1998).

4.4.1 Le problème de la conférence

Michel, Pierre et Alain se rencontrent inopinément dans un couloir. Michel interpelle les deux autres et leur rappelle leur promesse de se rencontrer pour se présenter leurs travaux. Chacun doit présenter ses travaux aux deux autres. Il y a donc six exposés : de Pierre à Alain, de Pierre à Michel, d'Alain à Michel, d'Alain à Pierre, de Michel à Pierre et enfin de Michel à Alain.

Ils conviennent de bloquer trois jours pour faire ces exposés. Un exposé prend une demi-journée.

Dès le début, Alain signale qu'il aimerait autant ne pas venir la troisième journée. Michel précise qu'il lui paraît important d'avoir vu ce qu'ont à dire Pierre et Alain avant de faire ses présentations. Alain voudrait lui aussi connaître les travaux de Pierre avant de faire ses exposés. Enfin, il préférerait également ne pas présenter ses travaux à Pierre et Michel en même temps.

4.4.2 Formalisation en CSP

Une fois le problème connu, il faut déterminer les variables, leur domaine et les contraintes. Les six exposés sont représentés par six variables :

- PA pour l'exposé de Pierre à Alain
- PM pour l'exposé de Pierre à Michel
- AM pour l'exposé de Alain à Michel
- AP pour l'exposé de Alain à Pierre
- MP pour l'exposé de Michel à Pierre
- MA pour l'exposé de Michel à Alain

Pour les domaines, une valeur est attribuée chronologiquement à chaque demi-journée. Les domaines contiennent donc les valeurs $\{1, 2, 3, 4, 5, 6\}$.

Finalement, les contraintes sont les suivantes :

- Alain ne souhaite pas rester le dernier jour : $AM \neq 6$, $AM \neq 5$, $MA \neq 6$, $MA \neq 5$, $AP \neq 6$, $AP \neq 5$, $PA \neq 6$, $PA \neq 5$
- Michel souhaite entendre les travaux de ses collègues avant d'exposer les siens : $PM < MA$, $PM < MP$, $AM < MA$, $AM < MP$
- Alain veut connaître les travaux de Pierre avant de faire ses exposés : $PA < AP$, $PA < AM$
- Alain ne veut pas présenter ses travaux à Michel et Pierre en même temps : $AM \neq AP$
- une personne ne peut pas écouter deux orateurs en même temps : $MA \neq PA$, $MP \neq AP$, $AM \neq PM$
- une personne ne peut pas être orateur et auditeur en même temps : $AM \neq MA$, $AM \neq PA$, $AP \neq MA$, $AP \neq PA$, $MP \neq PM$, $MP \neq AM$, $MA \neq PM$, $PM \neq AP$, $PA \neq MP$.

Certaines contraintes sont redondantes (par exemple, l'information $AM \neq MA$ est déjà contenue dans la contrainte $AM < MA$) et peuvent être ignorées. On peut noter que les contraintes sont au plus binaires.

4.4.3 Implantation en GNU-Prolog

En GNU-Prolog, le CSP peut s'écrire comme suit (les contraintes redondantes ont été retirées pour alléger l'exemple, cela ne modifie en rien la suite) :

```
     conf(PA,PM,AM,AP,MP,MA):-
(1)      fd_domain([PA,PM,AM,AP,MP,MA],1,6),
(2)      AM #\=# 6,
(3)      AM #\=# 5,
(4)      MA #\=# 6,
(5)      MA #\=# 5,
(6)      AP #\=# 6,
(7)      AP #\=# 5,
(8)      PA #\=# 6,
(9)      PA #\=# 5,
(10)     PM #<# MA,
(11)     PM #<# MP,
(12)     AM #<# MA,
(13)     AM #<# MP,
(14)     PA #<# AP,
(15)     PA #<# AM,
(16)     AM #\=# AP,
(17)     MA #\=# PA,
(18)     MP #\=# AP,
(19)     AM #\=# PM,
(20)     AP #\=# MA,
(21)     PM #\=# AP,
(22)     PA #\=# MP,
(23)     fd_labeling([MP,PM,PA,AM,AP,MA]).
```

Le prédicat `fd_domain` attribue aux domaines des variables de la liste les valeurs de 1 à 6.

L'écriture des contraintes est plus précise que celle donnée pour le CSP. En effet, comme annoncé précédemment GNU-Prolog utilise deux types de consistance. L'utilisation pour une contrainte de l'hyper arc-consistance ou de la consistance de bornes est précisée par la présence du deuxième dièse dans la contrainte. Par exemple, une contrainte `X #=# Y` indique que les opérateurs de consistance locale associés à cette contrainte effectuent de l'arc-consistance alors qu'une contrainte `X #= Y` indique que les opérateurs associés à cette contrainte effectuent de la consistance de bornes.

Le prédicat `fd_labeling` donne l'ordre d'instanciation des variables pour la phase d'énumération.

4.4.4 Résolution

Les contraintes de GNU-Prolog sont ensuite compilées en `X in range`. Comme annoncé plutôt, ces `X in range` peuvent être considérés comme des opérateurs de consistance locale. Le calcul de clôture se fait donc en appliquant les opérateurs de réduction associés aux opérateurs de consistance locale suivants :

```
(2)     AM in -{6},
(3)     AM in -{5},
(4)     MA in -{6},
(5)     MA in -{5},
(6)     AP in -{6},
(7)     AP in -{5},
(8)     PA in -{6},
(9)     PA in -{5},
(10)    PM in 0..max(MA)-1, MA in min(PM)+1..infinity,
(11)    PM in 0..max(MP)-1, MP in min(PM)+1..infinity,
(12)    AM in 0..max(MA)-1, MA in min(AM)+1..infinity,
(13)    AM in 0..max(MP)-1, MP in min(AM)+1..infinity,
(14)    PA in 0..max(AP)-1, AP in min(PA)+1..infinity,
(15)    PA in 0..max(AM)-1, AM in min(PA)+1..infinity,
(16)    AM in -{val(MA)}, MA in -{val(AM)},
(17)    MA in -{val(PA)}, PA in -{val(MA)},
(18)    MP in -{val(AP)}, AP in -{val(MP)},
(19)    AM in -{val(PM)}, PM in -{val(AM)},
(20)    AP in -{val(MA)}, MA in -{val(AP)},
(21)    PM in -{val(AP)}, AP in -{val(PM)},
(22)    PA in -{val(MP)}, MP in -{val(PA)},
```

Une contrainte primitive `X in range` signifie que la valeur de X doit appartenir à l'intervalle indiqué par `range` qui peut être un intervalle constant (par exemple 1..6 ou $\{6\}$ s'il ne contient qu'une valeur) mais aussi un intervalle "indexical" en utilisant principalement :

- `dom(Y)` représentant le domaine courant de Y.
- `min(Y)` représentant la valeur minimum du domaine courant de Y.
- `max(Y)` représentant la valeur maximum du domaine courant de Y.

Les opérations courantes sont permises sur ces intervalles ou leurs bornes (le $+$ et le $-$ dans notre exemple). Il faut aussi noter que `infinity` ne désigne pas l'infini (puisque les domaines sont finis) mais la plus grande valeur que peut prendre une variable (la plus petite étant 0).

Il reste à traiter le problème des contraintes `X` $\neq$ `Y`. Cette contrainte se compile en `X in -{val(Y)}` et `Y in -{val(X)}`. Ces opérateurs devant être monotones, ils ne sont réveillés que quand la variable de l'intervalle est instanciée. Dans le cas de `X in -{val(Y)}`, celui-ci ne sera appliqué que quand la valeur de Y sera fixée et l'application de cet opérateur supprimera cette même valeur de l'environnement de X. Plus formellement, l'opérateur peut être défini par :

$$\texttt{X in -\{val(Y)\}}(d) = \begin{array}{ll}(\text{si } d|_{\{y\}} = \{(y,v)\}) & \text{alors } \mathbb{D} \setminus \{(x,v)\} \\ & \text{sinon } \mathbb{D}.\end{array}$$

De plus amples détails sur la sémantique des `X in range` sont donnés dans (Codognet & Diaz, 1996).

L'environnement initial contient toutes les valeurs de 1 à 6 pour toutes les variables. Les opérateurs peuvent alors être appliqués. L'application des opérateurs de `(2)` à `(9)` retirent respectivement les éléments $(AM, 6)$, $(AM, 5)$, $(MA, 6)$, $(MA, 5)$, $(AP, 6)$, $(AP, 5)$, $(PA, 6)$ et $(PA, 5)$ de l'environnement. Si on applique ensuite les deux opérateurs de la ligne `(10)`, les valeurs de l'environnement de PM doivent être comprises entre 0 et `max(MA)-1`, c'est-à-dire 0 et 3. Les éléments $(PM, 4)$, $(PM, 5)$ et $(PM, 6)$ sont donc retirés de l'environnement. L'autre opérateur retire l'élément $(AM, 1)$. Les opérateurs sont ainsi tous appliqués (plusieurs fois si nécessaire) jusqu'à obtenir la clôture de l'environnement de départ par l'ensemble des opérateurs.

4.4.5 Solutions

L'exécution du programme précédent sans le prédicat de labeling produit le résultat :

```
AM = _#47(2..3)
AP = _#69(2..4)
MA = _#113(3..4)
MP = _#91(3..6)
PA = _#3(1..2)
PM = _#25(1..3)
```

L'intervalle entre parenthèse représente les valeurs restant dans l'environnement de la variable considérée (par exemple 2, 3 et 4 pour AP). L'ensemble de ses valeurs est donc la clôture calculée par les opérateurs.

Pour obtenir les solutions, il est nécessaire d'ajouter le prédicat de labeling. Le prédicat `fd_labeling` indique l'ordre d'instanciation des variables. La première variable est MP, son environnement contient les éléments $(MP, 3)$, $(MP, 4)$, $(MP, 5)$ et $(MP, 6)$. Le premier opérateur d'énumération restreint l'environnement de MP à $(MP, 3)$. La clôture pour ce nouvel environnement est calculée. Celle-ci est une feuille succès : on obtient la solution

$$\{(AM, 2), (AP, 4), (MA, 3), (MP, 3), (PA, 1), (PM, 1)\}$$

Le solveur essaie ensuite l'instanciation $(MP, 4)$. Le calcul de la clôture ne retire cette fois que l'élément $(AP, 4)$. L'énumération se poursuit donc sur la variable suivante (PM).

Cette phase s'arrête quand toutes les solutions ont été trouvées ou quand l'utilisateur décide de stopper la recherche. L'exécution fournit neuf solutions différentes à ce problème. On peut noter qu'aucune de ces solutions n'utilise les valeurs 2 ou 3 pour PM. Leur réunion est donc strictement incluse dans la clôture calculée sans labeling.

4.5 Conclusion

Dans ce chapitre, le cadre de la réduction de domaine a été décrit par le calcul d'une clôture. L'utilisation d'opérateurs de consistance locale permet de prendre en compte toute notion de consistance. De plus, l'utilisation de runs équitables et d'itérations chaotiques rend le formalisme proposé indépendant de la stratégie de sélection des opérateurs. Ce cadre général permet donc de décrire la réduction de domaine pour la plupart des solveurs sur domaines finis.

Les opérateurs de consistance locale sont choisis pour implanter les contraintes selon deux notions fondamentales. D'une part, si un opérateur est associé à un ensemble de contraintes, il doit être préservant pour ces contraintes (i.e. il ne doit pas retirer de solutions de ces contraintes). Le nombre de contraintes auquel est associé un opérateur dépend de la consistance utilisée. Par exemple, dans le cas de l'arc-consistance, un opérateur est associé à une contrainte, dans le cas de la consistance de chemin, un opérateur est associé à trois contraintes, ...Cette propriété assure que l'environnement calculé (i.e. la clôture) contient toutes les solutions du CSP. L'utilisation de la réduction de domaine comme filtrage permet donc d'approximer les solutions.

D'autre part, ces opérateurs doivent être complets pour chaque contrainte. Autrement dit, ils doivent être capable de décider si une instanciation n'est pas solution d'une contrainte. Cette propriété est utile dès que l'on introduit l'énumération pour obtenir les solutions du CSP.

Il a en effet été montré comment coupler la réduction de domaine avec une recherche systématique (l'énumération). Cette méthode prospective appelée maintien de consistance permet d'obtenir non plus un environnement réduit mais un ensemble d'instanciations. Dans le cas où les opérateurs (en plus d'être préservants) sont complets pour le CSP alors cet ensemble d'instanciations est exactement l'ensemble des solutions du CSP.

Deuxième partie

Explications de retraits de valeurs

Chapitre 5

Etat de l'art des explications

Sommaire

Dans ce chapitre, plusieurs travaux proches des explications sont présentés. Tous sont inspirés des systèmes de maintien de vérité. L'utilisation des explications est motivée principalement par deux objectifs :

- améliorer les algorithmes de recherche de solution basés sur le backtrack (i.e. les méthodes rétrospectives),
- améliorer l'efficacité de la relaxation de contrainte (principalement pour les problèmes de satisfaction de contraintes dynamiques).

Il faut noter que certaines des explications sont présentées pour l'un de ces objectifs mais sont aussi utilisées pour l'autre. Enfin, on aurait pu ajouter une utilisation pour la mise au point ou la compréhension des mécanismes du solveur.

Dans un premier temps, le principe des systèmes de maintien de vérité sera brièvement rappelé. Il est montré comment certaines méthodes de recherche de solutions (appelées méthodes rétrospectives) utilisent les explications. Après un rappel sur les problèmes de satisfaction de contraintes dynamiques, des méthodes utilisant les explications pour la relaxation de contraintes seront présentées. Enfin, le solveur `PaLM` entièrement basé sur les explications sera évoqué.

5.1 Systèmes de maintien de vérité

La plupart des méthodes ont été inspirées des *Assumption-based Truth Maintenance Systems* (ATMS (de Kleer, 1986)) issus des *Truth Maintenance Systems* (TMS (Doyle, 1979)).

L'idée des ATMS est d'enregistrer de l'information (appelée justification) concernant les inférences effectuées par un solveur de problèmes spécifiques afin de pouvoir les utiliser pour améliorer l'efficacité de la recherche. Ces systèmes sont divisés en deux composants : le moteur d'inférence et le TMS qui enregistre ces inférences. Pendant une résolution, les deux composants interagissent : chaque inférence est transmise au TMS pour enregistrement de justifications. Le TMS permet alors de :

- comprendre comment sont obtenues les solutions,
- identifier la source d'une contradiction,
- contrôler la recherche de solution en tenant compte du passé.

Ces informations se révèlent utiles pour améliorer la recherche du moteur d'inférence.

Les méthodes présentées dans ce chapitre sont des versions simplifiées des TMS et ATMS. Elles conservent des justifications de retrait de valeur ou d'échec.

5.2 Méthodes rétrospectives

On revient ici sur les méthodes de recherche dites rétrospectives évoquées dans le chapitre 3. Ces méthodes utilisent de l'information enregistrée pendant la recherche afin de ne pas explorer certaines branches de l'arbre de recherche menant à un échec.

5.2.1 Conflict Direct Backjumping

L'algorithme appelé *conflict direct backjumping* (Prosser, 1993) est une amélioration de l'algorithme de backtrack standard. Cette amélioration porte sur le mécanisme de retour-arrière dans le cas où toutes les valeurs ont été testées sur une variable x. Alors que le backtrack standard revient sur le choix précédent, le conflict direct backjumping revient plus haut (si possible) dans l'arbre de recherche en utilisant l'explication de l'échec constaté sur x. Une telle explication d'échec est appelée un *ensemble conflit*, défini comme suit dans notre formalisme. On rappelle qu'une instanciation partielle t est telle que pour tout $x \in V$, soit $t|_{\{x\}} = \emptyset$, soit $\exists v \in D_x$, $t|_{\{x\}} = \{(x, v)\}$.

Définition 5.1 *Une instanciation partielle t est un* ensemble conflit *si et seulement si* $\forall s \in \mathrm{Sol}, t \not\subseteq s$.

Autrement dit un ensemble conflit est une instanciation partielle qui ne participe à aucune solution du CSP.

Du point de vue du calcul, un nouvel ensemble conflit est calculé à partir de l'affectation courante quand celle-ci aboutit à un échec. Les ensembles conflits sont construits à partir des deux propriétés suivantes.

Propriété 1 *Soit t une instanciation partielle sur $W \subseteq V$. Si $\exists c \in C$ tel que $t|_{\mathrm{var}(c)} \notin T_c$ alors $t|_{\mathrm{var}(c)}$ est un ensemble conflit.*

Quand toutes les valeurs ont été testées pour la variable x, il existe un ensemble conflit pour chacune de ces valeurs.

Propriété 2 *Une instanciation partielle t est un ensemble conflit si* $\forall v \in D_x$, $t \cup \{(x, v)\}$ *est un ensemble conflit.*

Le conflict direct backjumping revient alors sur la dernière variable (selon l'ordre d'instanciation) dont l'instanciation est un élément de cet ensemble conflit. En effet, si la variable instanciée avant x n'appartient pas à l'ensemble conflit ayant éliminé x alors essayer une autre valeur pour cette variable n'empêchera pas d'obtenir un échec en réessayant d'instancier x.

L'algorithme *nogood recording* (Schiex & Verfaillie, 1993) employé pour améliorer la recherche (on verra son utilisation pour les CSP dynamiques après) peut être considéré comme une amélioration du conflict direct backjumping. Dans ces travaux, les ensembles conflits (appelés nogoods) sont enregistrés et peuvent être réutilisés pour réduire l'arbre de recherche non encore exploré. Ici quand une instanciation est défaite, tous les ensembles conflit la contenant sont oubliés.

5.2.2 Backtrack dynamique

Le *backtrack dynamique* introduit par Ginsberg dans (Ginsberg, 1993) se base sur une notion d'*explication éliminante*. Dans le cas du conflict direct backjumping, toutes les intanciations faites entre le point de retour choisi (disons l'instanciation de y) et la dernière instanciation sont effacées. Le backtrack dynamique diffère en ce sens, qu'il retire l'instanciation de y mais conserve toutes les instanciations faites après. Ce n'est donc pas un retour arrière dans l'arbre de recherche mais un changement d'arbre de recherche.

Définition 5.2 *Etant donné une instanciation partielle t sur $W \subseteq V$ du CSP, une* explication éliminante *pour une variable $x \in V$ est une paire (v, S) avec $v \in D_x$ et $S \subseteq W$ telle que $t|_S \cup \{(x, v)\}$ n'appartient à aucune solution du CSP.*

L'idée est que x ne peut pas prendre la valeur v à cause des instanciations des variables S. Dire que $t|_S \cup \{(x, v)\}$ ne participe pas à une solution du CSP revient exactement à dire que $t|_S \cup \{(x, v)\}$ est un ensemble conflit.

Définition 5.3 *Un* mécanisme d'élimination *pour un CSP est une fonction prenant en argument une instanciation partielle et une variable x n'appartenant pas à cette instanciation. La fonction retourne l'ensemble des explications éliminantes pour la variable x.*

5.3 CSP dynamiques

Un grand nombre de problèmes peuvent être modélisés par les contraintes. Cependant, certains problèmes évoluent selon les besoins de l'utilisateur ou simplement par des modifications des contraintes au cours du temps. Les solveurs classiques atteignent leur limite pour ces problèmes dits dynamiques, c'est-à-dire pour lesquels l'ensemble de contraintes se modifie. Cette situation se présente fréquemment en pratique pour les problèmes de gestions de ressources, de configuration, de reconnaissance de situations, ...

Même pour des problèmes a priori classiques, c'est-à-dire pour lesquels l'ensemble de contraintes devrait rester constant, cette situation peut se présenter quand l'utilisateur se retrouve face à un échec (i.e. pas de solutions). Le problème est dit surcontraint et le solveur utilisé répondra qu'il n'existe pas de solution (souvent par un très peu loquace *no*). Mais un utilisateur peut avoir besoin d'une solution quitte à modifier son problème de départ en le relâchant (i.e. en retirant des contraintes). On peut donc aussi parler d'un problème dynamique.

On s'intéresse ici à ces problèmes couramment appelés *Problèmes de Satisfaction de Contraintes Dynamiques* (DCSP) (Dechter & Dechter, 1988).

L'ensemble des contraintes du problème peut évoluer dans deux directions : par l'ajout de nouvelles contraintes ou par la suppression de contraintes (on parlera de contraintes relaxées). Du point de vue du problème cela reste simple : un DCSP est une séquence de CSP dont chacun diffère par l'ajout ou le retrait de certaines contraintes. La difficulté apparaît lors de la résolution de celui-ci.

L'ajout de contraintes ne pose aucune difficulté, il suffit d'ajouter des opérateurs au programme et donc continuer à faire de la réduction de domaine à partir de l'environnement obtenu. En effet, les éléments déjà retirés par les (opérateurs associés aux) contraintes "initiales" du problème ne sont toujours pas solutions de ces contraintes et doivent donc rester en dehors de l'environnement. C'est d'ailleurs souvent ainsi que les contraintes sont ajoutées dans un système de résolution de contraintes classique.

Le cas de la suppression de contrainte est quant à lui plus complexe. Des éléments retirés directement ou indirectement par une contrainte relaxée pourraient désormais appartenir à une solution du nouveau CSP. Relaxer des contraintes revient à trouver les solutions d'un nouveau CSP (le précédent CSP avec les contraintes relaxées en moins). Une approche naïve consiste à s'appuyer sur cette remarque en recommençant le calcul à partir de l'environnement initial avec le nouvel ensemble de contraintes. Or, recommencer un calcul à partir des domaines initiaux avec une ou plusieurs contraintes en moins implique que le solveurs va refaire une partie des calculs qu'il a déjà effectué lors des résolutions précédentes. Il ne va donc tirer aucun profit du calcul précédemment effectué. Or, pour ces problèmes dynamiques, le temps disponible pour produire une nouvelle solution est souvent limité.

Certains travaux (Berlandier & Neveu, 1994; Georget *et al.*, 1999) analysent les opérateurs de réduction pour déterminer les effets passés d'une contrainte et les défaire incrémentalement. D'autres travaux (Bessière, 1991; Debruyne, 1996; Fages *et al.*, 1998; Boizumault *et al.*, 2000), qui nous intéressent davantage, enregistrent de l'information durant la réduction de domaine afin de déterminer les effets des contraintes à relaxer.

5.3.1 Principe de la relaxation de contraintes

Les approches proposées ont un principe commun : elles consistent à effacer les effets passés des contraintes relaxées (c'est-à-dire à réintroduire certaines valeurs dans l'environnement obtenu) et, à partir de là, obtenir un état consistant en calculant la clôture du nouvel ensemble d'opérateurs. Plus précisément, la relaxation de contraintes peut se faire en suivant les étapes suivantes (Georget *et al.*, 1999; Jussien, 2001).

Suppression des opérateurs

La première étape consiste à retirer les contraintes à relaxer de l'ensemble des contraintes, c'est-à-dire faire disparaître les opérateurs de consistance locale associés à ces contraintes afin qu'ils ne soient plus appliqués.

Réintroduction des valeurs

A la seconde étape, les effets passés des contraintes relaxées (ou opérateurs relaxés) sont annulés. Ces effets ont pu être directs (pour les valeurs qui ont été retirées par l'application d'un des opérateurs relaxé) mais aussi indirects (pour les valeurs retirées par d'autres opérateurs mais conséquences de précédents retraits imputables à un ou plusieurs opérateurs relaxés). Cette étape consiste alors en un élargissement de l'environnement : des éléments y sont réintroduits. Une des principales différence entre les algorithmes proposés réside sur la détermination de cet ensemble d'éléments.

Repropagation

Finalement, certaines des valeurs réintroduites peuvent être retirées par l'application d'autres opérateurs (associés à des contraintes non relaxées). Ces retraits doivent alors être effectués, suivis de la propagation habituelle.

A la fin de ce processus, le système est dans un état consistant : c'est exactement l'environnement qui aurait été obtenu si les contraintes relaxées n'avaient jamais été introduites dans le système.

Il faut noter que le retrait d'une contrainte peut intervenir à n'importe quel moment lors d'un calcul de clôture, pas seulement quand celle-ci a été obtenue. Par exemple, si une queue de propagation est utilisée, elle peut ne pas être vide. De plus, le processus décrit précédemment s'applique quel que soit le nombre de contraintes à relaxer.

5.3.2 Les algorithmes DnAC-*

Dans (Bessière, 1991), une adaptation de l'algorithme AC-4 (Mohr & Henderson, 1986) aux DCSP est proposée. L'auteur se place donc dans le cadre de CSP et DCSP binaires, mais précise que les résultats peuvent s'appliquer aux CSP et DCSP généraux. La limitation d'AC-4 pour les DCSP est causée par le fait que l'algorithme oublie les raisons des retraits de valeurs qu'il effectue. L'algorithme DnAC-4 proposé étend AC-4 par l'enregistrement d'informations lors de la réduction de domaine. Plus précisément, pour chaque valeur retirée, l'algorithme enregistre une *justification* qui est la contrainte à l'origine du retrait.

Définition 5.4 *Une* justification *pour* (x, v) *est une contrainte* $c \in C$ *telle que* $\mathrm{var}(c) = \{x, y\}$ *et* $\forall \{(x, v), (y, w)\} \in T_c, (y, w) \notin d$.

En réalité, dans (Bessière, 1991), une justification n'est pas une contrainte, mais une variable. Dans la définition précédente, ce serait la variable y. Comme il existe une seule contrainte telle que $\text{var}(c) = \{x, y\}$, il est équivalent de considérer la contrainte ou la variable.

La justification enregistrée est la première contrainte[1] testée pour laquelle la valeur est sans support. Cela revient donc à n'enregistrer que les effets directs des contraintes. L'étape de réintroduction de valeurs se décompose donc en deux temps. D'abord, en utilisant ces justifications, les valeurs retirées par les contraintes à relaxer sont réintroduites. Ensuite, il faut calculer incrémentalement les autres valeurs à réintroduire : les valeurs retirées par une contrainte impliquant une variable dont des valeurs ont été réintroduites doivent en effet être reconsidérées. La repropagation n'est effectuée que si elle est nécessaire, c'est-à-dire si certaines valeurs réintroduites peuvent être retirées.

On peut noter l'utilisation des justifications également dans DnAC-6 (Debruyne, 1996) qui tire profit du principe d'AC-6 (Bessière & Cordier, 1993) pour améliorer les performances de DnAC-4.

5.3.3 Nogood recording

Dans (Schiex & Verfaillie, 1993), Schiex et Verfaillie s'intéressent également aux DCSP. Notant que l'approche de Bessière est limitée au maintien de consistance et ne permet pas d'obtenir des solutions au problème, ils proposent de traiter le problème du maintien de solutions. Le système proposé s'appuie sur l'enregistrement d'informations pendant la recherche appelées *nogoods*. Ces nogoods sont une extension de ceux définis pour le conflict direct backjumping en y incluant les contraintes.

Définition 5.5 *Un* nogood *est une paire (t, J), constituée d'une instanciation partielle t et d'un ensemble de contraintes J du problème, tels que t ne participe à aucune solution de l'ensemble de contraintes J. L'ensemble J est appelé la justification du nogood.*

L'utilisation des nogoods dans les ATMS nécessite une consommation de mémoire importante : pour chaque instanciation échouant, un nogood est construit. Schiex et Verfaillie remédient à ce problème en proposant plusieurs méthodes de simplifications. Ces méthodes sont sans rapport avec notre travail et ne sont donc pas décrites ici.

Les auteurs montrent comment construire ce système à partir des algorithmes de backtrack et de forward-checking. Dès qu'une feuille échec est obtenue dans l'arbre de recherche, un nogood est enregistré : si $t|_{\text{var}(c)} \notin T_c$ alors $(t, \{c\})$ est un nogood.

Les nogoods permettent également l'ajout de contraintes redondantes au problème. Une contrainte est dite redondante pour le CSP si toute solution du CSP

[1]Dans le cas de l'arc-consistance un opérateur de consistance locale étant associé à une contrainte, on peut parler de l'un ou de l'autre.

(sans cette contrainte) est solution de cette contrainte. Ajouter des contraintes redondantes permet alors de détecter plus rapidement des échecs durant la recherche.

L'utilisation des nogoods implique donc aussi des améliorations de la recherche. Le nogood (t, J) peut servir trois intérêts :

- il est possible de faire du retour arrière à la dernière variable instanciée parmi $\text{var}(J)$. Le retour arrière obtenu est très proche du conflict direct backjumping déjà présenté.
- La contrainte interdisant t est redondante et peut être ajoutée au CSP. Elle peut permettre d'élaguer des branches de l'arbre de recherche et donc d'améliorer les performances de la recherche.
- Lors de la résolution d'un CSP sur C', si $J \subseteq C'$ alors la contrainte interdisant t peut être ajoutée.

Le nombre de nogoods enregistré pouvant être énorme, une limitation est fixée sur la taille des instanciations pouvant apparaître dans un nogood.

5.4 Le système `PaLM`

Le système `PaLM` (Jussien & Barichard, 2000) (Propagation and Learning with Move) est une extension du langage de programmation par contraintes CHOCO (Laburthe & the OCRE project, 2000) intégrant les explications. C'est un solveur basé sur l'utilisation d'explications de contradiction (Jussien, 2001).

Suivant l'auteur, une *explication de contradiction* est un ensemble de contraintes menant à une contradiction. Une explication de contradiction est composée de deux parties : un sous-ensemble des contraintes du problème et un ensemble de contraintes correspondant à une instanciation. L'instanciation d'une variable x à la valeur v est considérée comme la contrainte d'égalité $x = v$. A l'écriture près, une explication de contradiction est donc exactement un nogood.

En considérant $C' \subseteq C$ et $\forall i \in \{1, \ldots, k\}$, $x_i \in V$ et $v_i \in D_{x_i}$. Si $C' \cup \{x_1 = v_1, \ldots, x_k = v_k\}$ est une explication de contradiction alors $C' \cup \bigcup_{i \in \{1,\ldots,k\} \setminus \{j\}} x_i = v_i$ est une *explication éliminante* pour (x_j, v_j).

Ces explications sont générées pendant la réduction de domaine et la recherche systématique. Le solveur sachant évidemment toujours pourquoi il retire une valeur (à cause de quelle contrainte), si une valeur est retirée, son explication éliminante est composée de la contrainte qui l'a retirée et des explications éliminantes de ses supports sur cette contrainte. Il faut noter que pour `PaLM`, le mécanisme d'énumération est simulé par l'ajout et la suppression de contraintes en utilisant les explications de contradictions.

Il est important de préciser que ce mécanisme de génération des explications a une complexité en espace et en temps qui reste polynomiale. Le surcout (pratiquement constant) de ce calcul est compensé par l'utilisation des explications. En pratique, une explication est mémorisée pour chaque valeur retirée.

Il est aussi possible de mémoriser plusieurs explications pour un même retrait. C'est la notion d'*explication k-relevante* (Bayardo Jr. & Miranker, 1996) proposée dans (Ouis *et al.*, 2003). D'une part plusieurs explications peuvent être mémorisées pour un même retrait. D'autre part les explications contenant des contraintes relaxées ne sont pas forcément oubliées. L'idée est de conserver en mémoire toutes les explications ayant au plus $k-1$ contraintes relaxées. La 1-relevance consiste donc à ne garder que les explications dont toutes les contraintes sont encore présentes dans le système. Malgré une consommation d'espace mémoire plus importante, l'utilisation de la k-relevance, pour $k=1$ ou $k=2$, permet d'obtenir de bons résultats pour les algorithmes de recherche basés sur les explications.

Le système `PaLM` entièrement basé sur les explications propose quatre utilisations des explications.

Tout d'abord les explications peuvent aider l'utilisateur à comprendre une situation :

- pourquoi le problème n'a pas de solutions ?
- pourquoi une variable ne peut pas avoir telle valeur ?
- que se passe t-il si une contrainte est ré-introduite dans le système ?

Pouvoir apporter de telles réponses se révèle très utile pour les problèmes sur-contraints et la mise au point de programmes par contraintes (sur laquelle nous reviendrons).

Dans les problèmes sur-contraints, les explications de contradiction fournissent un ensemble de contraintes parmi lesquelles doit être choisie la contrainte à relâcher pour pouvoir obtenir une solution (il n'y a cependant aucune garantie d'obtenir une solution).

Les explications de contradiction peuvent également être utilisées pour le retrait dynamique de contraintes en suivant le principe précédemment décrit (cf. section 5.3.1).

Enfin, les recherches par conflict direct backjumping ou par backtrack dynamique peuvent être réalisées. Les résultats obtenus pour MAC-DBT (Maintien d'arc-consistance et backtrack dynamique) (Boizumault *et al.*, 2000) sont une bonne illustration de l'intérêt des explications pour améliorer la recherche.

5.5 Conclusion

Plusieurs travaux proches des explications ont été brièvement décrits. Leur utilisation est principalement motivée par une amélioration de l'énumération (dans ce cas les explications sont souvent des instanciations menant à un échec) et par la relaxation de contrainte (les explications intègrent alors les contraintes du problème). Cette dernière ayant deux principales applications : les problèmes de satisfaction de contraintes dynamiques et le traitement des problèmes sur-contraints. Enfin le système `PaLM` entièrement basé sur les explications a été rapidement présenté. Ce dernier a montré que l'utilisation des explications se révèle utile (et efficace) pour les utilisations précédentes, mais aussi pour la compréhension des réponses proposées par le solveur. Les explications ont donc un rôle à jouer pour la mise au point de programmes par contraintes !

Chapitre 6

Arbres et ensembles explicatifs

Sommaire

Dans ce chapitre, les notions d'explications de retrait de valeur sont définies. Celles-ci sont données sous deux formes : les ensembles explicatifs et les arbres explicatifs.

Un ensemble explicatif est un ensemble d'opérateurs de consistance locale responsable du retrait d'un élément du domaine. Ils peuvent être considérés comme une généralisation des explications éliminantes de Jussien (Jussien, 2001).

Les arbres explicatifs sont des arbres de preuves pour le retrait d'éléments du domaine. Ces derniers requièrent une vision différente de la réduction de domaine. Alors que la réduction de domaine se concentrait sur les éléments restant dans l'environnement, on va s'intéresser ici aux éléments retirés. On parle de vision duale de la réduction de domaine. Cette vision correspond d'ailleurs davantage au travail du solveur qui en appliquant un opérateur retire des éléments qui ne participent pas aux solutions mais garde des éléments qui ne sont pas assurés de participer à une solution. Des règles sont alors associées aux duaux des opérateurs de consistance locale. Ces règles expriment les retraits d'éléments comme conséquence d'autres retraits et permettent la définition inductive d'arbres de preuve, appelés arbres explicatifs. Ces arbres sont en effet des explications des retraits de valeurs. Parmi ces arbres, certains peuvent être construits à partir d'une itération chaotique et sont appelés arbres explicatifs calculés pour le retrait de valeurs. Les ensembles explicatifs peuvent facilement en être extrait.

6.1 Ensembles explicatifs

Grâce aux opérateurs de consistance locale, on peut définir une notion d'explication suffisamment générale pour prendre en compte tout type de consistance locale. Un ensemble explicatif est un ensemble d'opérateurs responsables du retrait d'un élément d'un environnement.

Lors d'une itération, si un élément est retiré, tous les opérateurs utilisés depuis le début de l'itération ne sont pas nécessairement responsables de ce retrait. A l'opposé, l'opérateur qui a retiré l'élément considéré ne peut pas être considéré comme le seul responsable (il ne l'aurait peut être pas retiré s'il avait été le seul opérateur utilisé). Un ensemble explicatif pour (x, v) est donc un sous-ensemble des opérateurs du programme tel que toute itération chaotique de cet ensemble d'opérateurs retire l'élément (x, v).

Définition 6.1 *Soit* $(x, v) \in \mathbb{D}$ *et* $d \subseteq \mathbb{D}$. *On appelle* ensemble explicatif *pour* (x, v) *par rapport à* d *tout ensemble d'opérateurs de consistance locale* $E \subseteq R$ *tel que* $(x, v) \notin \mathrm{CL}\downarrow(d, E)$.

Plusieurs remarques évidentes peuvent être faites :

- Si $(x, v) \in \mathrm{CL}\downarrow(d, R)$ alors il n'existe aucun ensemble explicatif pour (x, v) par rapport à d.
- Si $(x, v) \notin d$ alors $\forall E \subseteq R, (x, v) \notin \mathrm{CL}\downarrow(d, E)$ donc tout ensemble d'opérateur de consistance locale est un ensemble explicatif du retrait de (x, v) par rapport à d, y compris l'ensemble vide.
- Etant donné que $E \subseteq E' \Rightarrow \mathrm{CL}\downarrow(d, E') \subseteq \mathrm{CL}\downarrow(d, E)$, si E est un ensemble explicatif du retrait de (x, v) par rapport à d alors tout sur-ensemble E' de E est aussi un ensemble explicatif pour le retrait de (x, v) par rapport à d, y compris le programme R.
- Enfin par monotonie de la clôture sur les environnements, si E est un ensemble explicatif pour (x, v) par rapport à d, alors E est un ensemble explicatif pour (x, v) par rapport à tout $d' \subseteq d$.

Il faut noter que l'on préfère des ensembles explicatifs minimaux (pour l'inclusion), c'est-à-dire contenant le moins d'opérateurs possibles. Cependant, calculer des ensembles explicatifs minimaux se fait avec un coût exponentiel (Junker, 2001). On verra à la fin de ce chapitre comment calculer des ensembles explicatifs pour les éléments retirés lors d'un calcul.

Les justifications de Bessière décrites dans le chapitre précédent proposent une version très simplifiée de cette idée, puisque seule la contrainte qui a retiré l'élément est mémorisée. Concernant les nogoods et les explications éliminantes, il est nécessaire d'inclure les opérateurs d'énumération pour pouvoir les obtenir. Ces considérations seront traitées dans ce chapitre.

6.2 Vision duale de la réduction de domaine

Avant de se pencher en détail sur une vision duale de la réduction de domaine, il est important de comprendre ce que cela signifie et pourquoi on s'y intéresse.

Un utilisateur de la programmation par contrainte recherche des solutions à un problème de satisfaction de contraintes. Il regarde donc plutôt ce qui est susceptible de participer à une solution (i.e. les éléments qui restent dans l'environnement) et se désintéresse des autres. La vision duale consiste à s'intéresser non plus à ces environnements mais à leur complémentaire dans $\mathbb{D}$, c'est-à-dire aux éléments retirés. La vision duale de la réduction de domaine revient alors à considérer l'"élargissement du complémentaire de l'environnement" par des opérateurs duaux des opérateurs de consistance locale. Les notions d'itérations et de clôture sont ainsi décrites d'un point de vue dual.

On s'intéresse à la dualité à cause de l'incomplétude des solveurs. En effet, les solveurs assurent que les valeurs retirées n'appartiennent à aucune solution, mais celles restant dans la clôture n'appartiennent pas forcément à une solution (la clôture est un sur-ensemble des solutions). Les solveurs peuvent être considérés comme des constructeurs de preuves de retraits de valeurs. Dès lors, il est plus facile d'avoir des preuves des retraits de valeurs que des preuves de "non-retrait" des valeurs de la clôture.

On notera $\overline{d} = \mathbb{D} \setminus d$. Afin d'aider à la compréhension, on utilisera toujours la notation $\overline{d}$ pour un sous-ensemble de $\mathbb{D}$ si intuitivement il représente un ensemble de valeurs retirées.

6.2.1 Opérateurs duaux

Un opérateur de consistance locale appliqué pour calculer un ensemble de valeurs localement consistantes possède un opérateur dual calculant un ensemble de valeurs localement inconsistantes.

Définition 6.2 *Soit r un opérateur de consistance locale, on notera $\widetilde{r}$ le* dual *de r défini par : $\forall d \subseteq \mathbb{D}, \widetilde{r}(\overline{d}) = \overline{r(d)}$.*

On peut remarquer que $\widetilde{\widetilde{r}} = r$. Pour faciliter la compréhension, on utilisera la notation r pour un opérateur de consistance locale et $\widetilde{r}$ pour le dual d'un opérateur de consistance locale.

Rappelons qu'un type $(\mathrm{out}(r), \mathrm{in}(r))$ peut être associé à un opérateur de consistance locale r (cf. section 4.1.1). Cet opérateur, appliqué à l'environnement d est tel que :

- $r(d)|_{V\setminus\mathrm{out}(r)} = \mathbb{D}|_{V\setminus\mathrm{out}(r)}$;
- $r(d)|_{\mathrm{out}(r)}$ est l'ensemble des valeurs de $\mathbb{D}|_{\mathrm{out}(r)}$ (i.e. le domaine de $\mathrm{out}(r)$) qui sont localement consistantes avec l'environnement (c'est-à-dire que $r(d)|_{\mathrm{out}(r)}$ est tel que $d|_{V\setminus\mathrm{out}(r)} \cup r(d)|_{\mathrm{out}(r)}$ est r-consistant).

Son dual est donc tel que :

- $\widetilde{r}(\overline{d})|_{V\setminus\mathrm{out}(r)} = \emptyset$;
- $\widetilde{r}(\overline{d})|_{\mathrm{out}(r)}$ est l'ensemble des valeurs de $\mathbb{D}|_{\mathrm{out}(r)}$ qui sont localement inconsistantes avec l'environnement.

Le dual d'un opérateur de consistance locale calcule donc exactement un ensemble de valeurs inconsistantes.

Il faut noter que pour $\widetilde{r}$ comme pour r, les éléments calculés sont des éléments du domaine $\mathbb{D}$, pas seulement de l'environnement d auquel l'opérateur est appliqué. De la même façon que le calcul de r ne dépend que des variables $\mathrm{in}(r)$, le calcul de $\widetilde{r}$ ne dépend aussi que de $\mathrm{in}(r)$. L'opérateur $\widetilde{r}$ est donc tel que $\widetilde{r}(\overline{d}) = \widetilde{r}(\overline{d}|_{\mathrm{in}(r)})$.

Dans le cas de l'arc-consistance (où un opérateur est associé à une contrainte binaire), chaque valeur obtenue est telle que tous ses supports (pour la contrainte implantée) sont dans $\overline{d}$, c'est-à-dire ne sont pas dans l'environnement. C'est donc l'ensemble des valeurs de $\mathbb{D}|_{\mathrm{out}(r)}$ telles que pour toutes les solutions de la contrainte où elles apparaissent, un des éléments n'est pas dans l'environnement. Ces valeurs sont donc localement inconsistantes et doivent être retirées de l'environnement.

Exemple 8 *Dans le cas où l'opérateur r est l'opérateur* GNU-Prolog `X in 0..max(Y)-1`*, son dual calcule toutes les valeurs du domaine (et pas seulement de l'environnement) de X qui sont supérieures ou égales à la plus grande valeur de l'environnement de Y. En prenant le CSP défini sur $V = \{X, Y\}$ et $D_X = D_Y = \{0, 1, 2, 3\}$ par l'unique contrainte $X < Y$ implantée en partie (pour X) par l'opérateur ci-dessus et si on considère l'environnement $d = \{(X, 1), (X, 2), (Y, 1), (Y, 2)\}$ alors l'ensemble d'éléments inconsistants pour* `X in 0..max(Y)-1` *est $\widetilde{r}(\overline{d}) = \{(X, 2), (X, 3)\}$. En effet la valeur 2 pour X n'apparaît que dans la solution $\{(X, 2), (Y, 3)\}$ de la contrainte, or $(Y, 3)$ n'est pas dans d. $(X, 3)$ ne participe à aucune solution de la contrainte. Ces deux éléments peuvent donc être retirés de l'environnement (si ils y sont présents).*

Ce sont ces opérateurs duaux qui vont permettre de faire des preuves de retraits. Lors de la réduction de domaine on aurait pu considérer, comme cela est fait classiquement, uniquement les opérateurs de réduction. Mais les opérateurs que l'on vient de définir doivent être les duaux des opérateurs de consistance locale et non des opérateurs de réduction. $\widetilde{r}$ calcule exactement les valeurs retirées (retirables serait plus exact) par r. Le dual d'un opérateur de réduction calculerait en plus les éléments de $\overline{d}$.

De la même façon que l'on considère des ensembles d'opérateurs de consistance locale, on va considérer des ensembles d'opérateurs duaux, plus particulièrement le dual du programme R. Soit $\widetilde{R} = \{\widetilde{r} \mid r \in R\}$.

6.2.2 Clôture et itération

Comme pour la réduction de domaine, on va s'intéresser à des environnements particuliers : les points fixes communs des opérateurs $\overline{d} \mapsto \overline{d} \cup \widetilde{r}(\overline{d})$, $\widetilde{r} \in \widetilde{R}$.

Tout d'abord la notion de r-consistance est exprimée en fonction de $\widetilde{r}$.

Lemme 6.1 *Un environnement d est r-consistant si $\widetilde{r}(\overline{d}) \subseteq \overline{d}$.*

Preuve. $d \subseteq r(d) \Leftrightarrow d \subseteq \overline{\widetilde{r}(\overline{d})} \Leftrightarrow \widetilde{r}(\overline{d}) \subseteq \overline{d}$. □

On peut considérer la clôture ascendante d'un ensemble d'opérateurs $\widetilde{R}$ à partir d'un (complémentaire d') environnement $\overline{d}$. On rappelle la définition de la clôture ascendante (définition 2.4) :

La clôture ascendante de $\overline{d}$ par $\widetilde{R}$, notée $\mathrm{CL}\uparrow(\overline{d}, \widetilde{R})$ est le plus petit $\overline{d'}$ tel que $\overline{d} \subseteq \overline{d'}$ et $\forall r \in R$, $\widetilde{r}(\overline{d'}) \subseteq \overline{d'}$.

La clôture ascendante de $\overline{d}$ par $\widetilde{R}$ est le plus petit point fixe commun des opérateurs duaux des opérateurs de réduction. On peut noter que cette clôture est un ensemble contenant les éléments de $\overline{d}$ et l'ensemble des éléments à retirer de l'environnement d pour que celui-ci soit R-consistant.

En s'appuyant sur le lemme 6.1, cette définition peut être reformulée comme suit : la clôture ascendante de $\overline{d}$ par $\widetilde{R}$, notée $\mathrm{CL}\uparrow(\overline{d}, \widetilde{R})$ est le plus petit $\overline{d'}$ tel que $\overline{d} \subseteq \overline{d'}$ et d' est R-consistant.

La correspondance entre cette clôture ascendante et la clôture descendante des opérateurs de consistance locale apparaît clairement.

Théorème 6.2 $\mathrm{CL}\uparrow(\overline{d}, \widetilde{R}) = \overline{\mathrm{CL}\downarrow(d, R)}$.

Preuve. par le théorème 2.2. □

En particulier, si la clôture ascendante des opérateurs duaux est calculée à partir de l'ensemble vide alors celle-ci est égale au complémentaire de la clôture descendante des opérateurs de consistance locale à partir du domaine $\mathbb{D}$.

Comme pour la réduction de domaine, on prouve que cette clôture peut être calculée par une itération. Les notions de run et de run équitable restent valides pour les opérateurs qu'ils soient opérateurs de consistance locale ou duaux d'opérateurs de consistance locale. De la même façon qu'une itération descendante à partir d'un environnement par rapport à un run d'opérateurs de consistance locale a été définie, une itération ascendante par rapport à un run d'opérateurs duaux peut être définie.

Définition 6.3 *L'*itération ascendante *de $\widetilde{R}$ à partir de $\overline{d} \subseteq \mathbb{D}$ par rapport au run $\widetilde{r^1}, \widetilde{r^2}, \ldots$ est la séquence infinie $\overline{\delta^0}, \overline{\delta^1}, \overline{\delta^2}, \ldots$ inductivement définie par :*

- $\overline{\delta^0} = \overline{d}$,
- $\overline{\delta^{i+1}} = \overline{\delta^i} \cup \widetilde{r^{i+1}}(\overline{\delta^i})$.

En utilisant la définition d'opérateur dual, on peut réécrire le second item de la précédente définition : $\overline{\delta^{i+1}} = \overline{\delta^i} \cup \overline{r^{i+1}(\delta^i)}$, c'est-à-dire que ce pas de l'itération consiste à ajouter à $\overline{\delta^i}$ les éléments de δ^i retirés par r^{i+1} (i.e. les éléments localement inconsistants selon $\widetilde{r}$).

Le lien avec les itérations descendantes décrivant la réduction de domaine peut être mis en évidence. En considérant l'itération descendante à partir de d par rapport au run $r^1, r^2, \ldots$, il suffit de remarquer que : $\overline{\delta^i} \cup \widetilde{r^{i+1}}(\overline{\delta^i}) = \overline{d^i \cap r^{i+1}(d^i)}$. D'où, $\delta^{i+1} = d^{i+1}$.

Bien entendu si l'itération ascendante se fait par rapport à un run équitable alors sa limite est la clôture, on a donc $\cup_{j\in\mathbb{N}}\overline{\delta^j} = \mathrm{CL}\uparrow(\overline{d}, \widetilde{R}) = \overline{\mathrm{CL}\downarrow(d, R)} = \overline{\cap_{j\in\mathbb{N}} d^i}$.

On a donc exprimé une vision duale de la réduction de domaine, c'est-à-dire considéré les complémentaires des environnements. La première approche, celle des environnements réduits, est plus intuitive car elle met en avant les éléments susceptibles d'appartenir à une solution, et donc intéressant l'utilisateur. Mais l'approche duale proposée ici reflète davantage les calculs du solveur qui élimine peu à peu les valeurs localement inconsistantes. De plus, l'application du dual d'un opérateur de consistance locale retourne exactement un ensemble de valeurs inconsistantes (qui doivent être retirées de l'environnement quand elles y sont présentes). Ce sont donc ces opérateurs qui contiennent l'information nécessaire pour expliquer les retraits de valeurs.

6.3 Règles de déductions

6.3.1 Définition

Le calcul des complémentaires des environnements a été décrit. L'application du dual d'un opérateur de consistance locale retourne un ensemble de valeurs inconsistantes. On décrit ici plus précisément les retraits en se focalisant sur la raison du retrait de chacune de ces valeurs inconsistantes. Pour cela, un système de règles (dans le sens de (Aczel, 1977)) est associé à chaque opérateur. Ces règles sont naturelles pour définir le dual d'un opérateur de consistance locale et adéquates pour construire des arbres de preuve.

Définition 6.4 *Une* règle de déduction *est une règle $h \leftarrow B$ telle que $h \in \mathbb{D}$ et $B \subseteq \mathbb{D}$.*

Intuitivement, une règle de déduction $h \leftarrow B$ peut être comprise comme suit : si aucun élément de B n'est présent dans l'environnement, alors h est localement inconsistant (il ne participe donc à aucune solution du CSP) et peut être retiré.

Tout comme les opérateurs de consistance locale et leur duaux, un type peut être associé à une règle de déduction. Une règle de déduction $h \leftarrow B$ de type (V_{out}, V_{in}) est telle que $h \in \mathbb{D}|_{V_{out}}$ et $B \subseteq \mathbb{D}|_{V_{in}}$.

En suivant le lemme 2.3, chaque opérateur monotone peut être défini par un ensemble de telles règles. En particulier, chaque dual d'un opérateur de consistance locale peut être défini par un ensemble de règles de déduction. Etant donné qu'à un opérateur de consistance locale n'est associé qu'un unique opérateur dual, on va considérer qu'un ensemble de règles définissant $\widetilde{r}$ est associé à l'opérateur de consistance locale r. De façon évidente, les règles définissant un opérateur de consistance locale de type $(\text{out}(r), \text{in}(r))$ ont le type (V_{out}, V_{in}) avec $V_{out} \in \text{out}(r)$ et $V_{in} \subseteq \text{in}(r)$.

6.3.2 Application à diverses consistances

Pour un opérateur donné, il peut exister plusieurs ensembles de règles le définissant, mais en général, l'un d'entre eux est plus naturel dans notre contexte. Cet ensemble est souvent fortement lié à la notion de support définie dans la section 3.2. Chaque règle exprime le retrait d'un élément comme la conséquence du retrait d'éléments de ses supports. On fournit plusieurs illustrations sur différents exemples de consistance.

Arc-consistance

Dans le cas le plus simple, celui de l'arc-consistance, un support contient un seul élément que l'on appellera également support par abus de langage. Le corps des règles est directement tiré des supports : B est l'ensemble des supports de h pour la contrainte implantée par l'opérateur. En effet si tous les supports de h sont absents de l'environnement alors h est localement inconsistant et peut être retiré.

Il faut remarquer le cas particulier où h ne peut pas être solution de la contrainte implantée par l'opérateur, c'est-à-dire où h n'a aucun support dans le domaine. Dans ce cas, la règle est un *fait* (i.e. le corps de la règle est vide) : $h \leftarrow \emptyset$.

Pour un opérateur de consistance locale r, il existe exactement une règle par valeur de $\mathbb{D}|_{\mathrm{out}(r)}$.

> **Exemple 9** *Pour le domaine* $\mathbb{D} = \{(X,0),(X,1),(X,2),(Y,0),(Y,1),(Y,2)\}$, *l'opérateur dual de* `X in 0..max(Y)-1` *(associé à la contrainte* $X < Y$*) retire les valeurs de* X *pour lesquelles il n'existe pas de valeur strictement supérieure dans l'environnement de* Y. *Par exemple,* $(X,0)$ *peut être retiré si les valeurs* $(Y,1)$ *et* $(Y,2)$ *ne sont pas dans l'environnement. Autrement dit, la règle de déduction* $(X,0) \leftarrow \{(Y,1),(Y,2)\}$ *est une des règles définissant l'opérateur dual de* `X in 0..max(Y)-1`. *Les deux autres sont* $(X,1) \leftarrow \{(Y,2)\}$ *et* $(X,2) \leftarrow \emptyset$. *On peut noter que ces règles peuvent être obtenues à partir de la règle générique* $(X,v) \leftarrow \{(Y,v') \mid v < v'\}$.

Selon la contrainte implantée, les règles de déduction peuvent avoir un nombre d'éléments différents dans le corps. Une contrainte d'égalité donnera lieu à des règles contenant un seul élément (puisque chaque élément n'a qu'un seul support). Pour une inégalité, au contraire, les règles ont des corps contenant tous les éléments d'un domaine de variable sauf l'élément qui est "égal" à la tête de la règle.

Il faut noter que ces règles reflètent exactement les calculs effectués par le solveur. En effet pour supprimer une valeur, le solveur teste l'existence de supports pour celle-ci. S'il n'en trouve aucun, il la retire (il applique donc la règle de déduction).

Hyper arc-consistance

Dans le cas de l'hyper arc-consistance, un opérateur est toujours associé à une contrainte, mais les tuples solutions de la contrainte peuvent contenir plus de deux variables. Les supports d'un élément sur une contrainte peuvent contenir plus d'un élément. Si un élément de chacun de ces supports n'est pas dans l'environnement alors l'élément peut être retiré. Pour un élément donné, il existe donc autant de règles de déductions que de possibilités de prendre un élément dans chacun de ses supports.

Exemple 10 *Soit* $V = \{X, Y, Z\}$*,* $D_X = D_Y = D_Z = \{0, 1\}$ *et la contrainte* $X = Y + Z$ *dont les solutions sur* $\mathbb{D}$ *sont :*

$$\{(X,0),(Y,0),(Z,0)\}$$
$$\{(X,1),(Y,0),(Z,1)\}$$
$$\{(X,1),(Y,1),(Z,0)\}$$

L'opérateur de consistance locale (de type $(\{X\},\{Y,Z\})$*) associé à cette contrainte (en* GNU-Prolog *cet opérateur est* `X in dom(Y)+dom(Z)`*) peut être décrit par un ensemble naturel de règles de déduction. Ces règles auront pour tête soit* $(X,0)$*, soit* $(X,1)$*. L'élément* $(X,0)$ *ayant un seul support, il suffit qu'un élément de ce support ne soit pas dans l'environnement pour que* $(X,0)$ *soit localement inconsistant. Pour* $(X,0)$*, les règles sont donc :*

$$\begin{array}{lcl} (X,0) & \leftarrow & \{(Y,0)\} \\ (X,0) & \leftarrow & \{(Z,0)\} \end{array}$$

L'élément $(X,1)$ *ayant deux supports (contenant deux éléments), il existe quatre possibilités pour avoir un élément de chaque support hors de l'environnement. Donc pour* $(X,1)$*, les règles sont :*

$$\begin{array}{lcl} (X,1) & \leftarrow & \{(Y,0),(Y,1)\} \\ (X,1) & \leftarrow & \{(Z,0),(Z,1)\} \\ (X,1) & \leftarrow & \{(Y,0),(Z,0)\} \\ (X,1) & \leftarrow & \{(Y,1),(Z,1)\} \end{array}$$

Consistance de chemin

La consistance de chemin est une consistance permettant une précision plus importante que l'arc-consistance car elle considère trois variables du problème. Un opérateur est alors associé à trois contraintes binaires (sur ces trois variables). Le corps des règles n'est plus l'ensemble des supports d'un élément sur une contrainte comme pour l'arc-consistance, mais un ensemble contenant un élément de chaque support sur les *trois* contraintes. Autrement dit, les trois contraintes sont considérées comme une seule contrainte.

Exemple 11 *On considère l'ensemble de variables* $V = \{X, Y, Z\}$ *et le domaine* $\mathbb{D} = \{(X,0),(X,1),(Y,0),(Y,1),(Z,0),(Z,1)\}$*. Aux trois contraintes :* $X \neq Y$*,* $X \neq Z$ *et* $Y \neq Z$*, peut être associer un opérateur de type* $(\{X\},\{Y,Z\})$ *défini par les règles de déduction :*

$$\begin{array}{ll} (X,0) & \leftarrow \\ (X,1) & \leftarrow \end{array}$$

On voit sur cet exemple l'intérêt de la consistance de chemin par rapport à l'arc-consistance. Chaque contrainte possède une solution avec chaque valeur des variables impliquées. Par exemple $T_{X \neq Y} = \{\{(X,0),(Y,1)\},\{(X,1),(Y,0)\}\}$*. Les règles de déduction associées à un opérateur d'arc-consistance ont donc toujours un élément dans leur corps. Mais si on prend (comme le fait la consistance de chemin) les trois contraintes en même temps, il n'existe aucune instanciation* *des trois variables qui soit solution des trois contraintes. Les règles ont donc un corps vide qui permet de retirer chaque élément.*

Les règles de déductions s'appuient sur une plus grande information en regroupant trois contraintes et sont donc plus précises. On peut donc considérer la consistance de chemin comme de l'hyper arc-consistance sur un ensemble de contraintes tel que chaque triplet de contraintes de départ est réuni en une seule contrainte.

6.4 Arbres explicatifs

Dans cette section, on rappelle la définition inductive d'arbre de preuve (Aczel, 1977) par un système de règle. Les règles utilisées étant des règles de déduction associées à des opérateurs, on peut extraire d'un arbre explicatif un ensemble d'opérateurs. Il est prouvé que cet ensemble est un ensemble explicatif pour la racine de l'arbre. Finalement, il est montré comment construire certains de ces arbres à partir d'une itération chaotique et les liens avec la réduction de domaine sont établis pour cette nouvelle notion d'explication.

6.4.1 Arbres explicatifs

Il est possible de construire des arbres de preuve à partir des règles de déduction. Pour cela, on fixe tout d'abord un ensemble de règles de déduction $\mathcal{R}_r$ définissant un opérateur de consistance locale r. On considère alors l'ensemble des règles de déduction pour tous les opérateurs de consistance locale de R : soit $\mathcal{R} = \cup_{r \in R} \mathcal{R}_r$.

Pour les arbres, on utilise les notations définies dans le chapitre 2 :

- $\text{cons}(h, T)$ est l'arbre défini par : h est sa racine et T l'ensemble de ses sous-arbres.
- La racine d'un arbre t est notée $\text{root}(t)$.

Rappelons alors la définition 2.8 d'arbre de preuve par rapport à un ensemble de règles.

Un *arbre de preuve par rapport à* $\mathcal{R}$ est un arbre $\text{cons}(h, T)$ tel que $h \leftarrow \{\text{root}(t) \mid t \in T\} \in \mathcal{R}$ et T est un ensemble d'arbres de preuve par rapport à $\mathcal{R}$.

Dans notre contexte, un arbre de preuve est donc un arbre tel que chaque nœud est lié à ses fils par une règle de déduction.

Définition 6.5 *Un* arbre explicatif *est un arbre de preuve par rapport à un ensemble de règles de déduction.*

On peut noter que le domaine $\mathbb{D}$ étant fini, le corps des règles de déduction est toujours un ensemble fini. Par conséquent, les arbres explicatifs sont des arbres finis.

$$\dfrac{\dfrac{\dfrac{}{(X,2)}}{(Y,1)} \qquad \dfrac{}{(Y,2)}}{(X,0)} \qquad\qquad \dfrac{\dfrac{\dfrac{}{(X,0)}}{(Y,1)} \qquad \dfrac{}{(Y,2)}}{(X,0)} \qquad\qquad \dfrac{}{(X,0)}$$

FIG. 6.1 – Arbres explicatifs pour $(X,0)$

Exemple 12 *Soit le système de règles suivant :*

$$\begin{array}{rcl}
(X,2) & \leftarrow & \emptyset \\
(Y,2) & \leftarrow & \emptyset \\
(X,0) & \leftarrow & \emptyset \\
(Y,1) & \leftarrow & \{(X,0)\} \\
(Y,1) & \leftarrow & \{(X,2)\} \\
(X,0) & \leftarrow & \{(Y,1),(Y,2)\}
\end{array}$$

Les arbres explicatifs de la figure 6.1 sont des arbres de preuve pour ce système de règles[1].

On montre alors que pour tout élément n'appartenant pas à la clôture de $\mathbb{D}$ par R, il existe un arbre explicatif ayant cet élément pour racine.

Théorème 6.3 $\overline{\mathrm{CL}\downarrow(\mathbb{D},R)}$ *est l'ensemble des racines d'arbres explicatifs par rapport à $\mathcal{R}$.*

Preuve. par le théorème 2.4. □

Etant donné qu'on s'intéresse parfois à la clôture d'un environnement différent de $\mathbb{D}$, il faut ajouter des règles afin d'éliminer les éléments de $\overline{d}$. Ces règles sont des faits : pour chaque $(x,v) \in \overline{d}$, on considère la règle $(x,v) \leftarrow \emptyset$. On considère alors $\mathcal{R}^d = \{(x,v) \leftarrow \emptyset \mid (x,v) \in \overline{d}\}$.

Le corollaire du théorème précédent peut alors être énoncé.

Corollaire 6.4 $\overline{\mathrm{CL}\downarrow(d,R)}$ *est l'ensemble des racines d'arbres explicatifs par rapport à $\mathcal{R} \cup \mathcal{R}^d$.*

Donc pour chaque élément n'appartenant pas à la clôture, il existe *au moins* un arbre explicatif l'ayant pour racine. Une règle de déduction exprimant le retrait d'un élément comme conséquence du retrait d'autres éléments, un arbre explicatif est donc bien une preuve du retrait d'un élément.

Il faut noter qu'en pratique on pourra plus facilement obtenir certains arbres explicatifs : ceux qui peuvent être construits à partir d'une itération. Ce n'est pas le cas par exemple du deuxième arbre de preuve de la figure 6.1. En effet, on voit que le retrait de $(X,0)$ (la racine de l'arbre) est une conséquence (indirecte) du retrait

[1] La racine de ces arbres est en bas.

de $(X, 0)$ (une feuille de l'arbre). Or, un élément ne peut être retiré qu'une seule fois lors d'une itération. Donc, un élément peut apparaître plusieurs fois dans différentes branches de l'arbre, mais pas plusieurs fois dans la même branche.

On a uniquement considéré les opérateurs de consistance locale. Bien entendu, le résultat pourrait être étendu en prenant en compte des opérateur d'énumération comme on le verra dans la section 6.5.

6.4.2 Extraction d'ensembles explicatifs

Un arbre explicatif pour un élément de $\overline{\mathrm{CL}\downarrow(d, R)}$ est construit par l'application d'une ou plusieurs règles de déduction. Chacune de ces règles est issue soit d'un ensemble $\mathcal{R}_r$, soit de l'ensemble $\mathcal{R}^d$.

Un arbre explicatif étant défini, il est possible de lui associer un ensemble d'opérateurs de consistance locale.

Définition 6.6 *Soit t un arbre explicatif. Un* ensemble d'opérateurs de consistance locale associé à t *est un ensemble X tel que, pour chaque nœud de t : si h est son étiquette et B l'ensemble des étiquettes de ses fils, alors :*

- *soit $h \notin d$ (et $B = \emptyset$) ;*
- *soit il existe $r \in X, h \leftarrow B \in \mathcal{R}_r$.*

Il faut remarquer qu'il peut exister plusieurs ensembles associés à un même arbre explicatif. D'une part parce qu'une règle de déduction peut apparaître dans plusieurs ensembles de règles et on peut donc choisir entre plusieurs opérateurs de consistance locale. Mais aussi parce que si un ensemble X est associé à un arbre explicatif, tout sur-ensemble de X peut aussi lui être associé. En particulier, l'ensemble d'opérateurs R peut être associé à tout arbre explicatif.

Il est important de rappeler que la racine d'un arbre explicatif n'appartient pas à la clôture de d par l'ensemble d'opérateurs de consistance locale R. Donc, par la définition 6.1, il existe un ensemble explicatif pour cette valeur, le plus grand étant R. Mais il est prouvé dans le théorème suivant que les ensembles définis précédemment sont aussi des ensembles explicatifs pour le retrait de cette valeur. En fait un tel ensemble d'opérateurs de consistance locale est responsable du retrait de la racine de l'arbre.

Théorème 6.5 *Si t est un arbre explicatif, et X un ensemble d'opérateurs de consistance locale associé à t, alors X est un ensemble explicatif pour* $\mathrm{root}(t)$.

Preuve. par le corollaire 6.4. □

On a déjà dit qu'on préférait les ensembles explicatifs minimaux. Si plusieurs ensembles d'opérateurs de consistances locales peuvent être associés à un arbre explicatif de racine (x, v), en pratique, on choisira le plus petit. Il faut bien préciser que celui-ci n'est pas forcément un ensemble explicatif minimal pour (x, v).

On peut noter que la réciproque du théorème est valide, à savoir : si X est un ensemble explicatif pour (x, v) alors il existe au moins un arbre explicatif enraciné par (x, v) auquel il peut être associé.

6.4.3 Arbres explicatifs calculés

Parmi les arbres explicatifs, certains peuvent être obtenus à partir d'une itération chaotique, autrement dit à partir du calcul du solveur.

Avant tout, on peut remarquer que certains arbres explicatifs ne correspondent à aucune itération. Lors d'une itération, un élément du domaine n'est retiré qu'une seule fois, par un seul opérateur.

Exemple 13 *Le deuxième arbre explicatif de la figure 6.1 ne correspond à aucune itération. En effet,* $(Y, 1)$ *est retiré à cause du retrait de* $(X, 0)$*. Ce retrait de* $(Y, 1)$ *(avec celui de* $(Y, 2)$*) cause le retrait de* $(X, 0)$*. Or* $(X, 0)$ *a déjà été retiré. Dans une itération,* $(X, 0)$ *ne peut pas à la fois causé le retrait de* $(Y, 1)$ *et être conséquence du retrait de* $(Y, 1)$*.*

Ici on ne s'intéresse qu'aux arbres explicatifs qui peuvent être déduits d'un calcul, on les appelle *arbres explicatifs calculés*. On considère fixée une itération chaotique $d = d^0, d^1, \ldots, d^i, \ldots$ de R par rapport au run $r^1, r^2, \ldots$. Dans ce contexte, on peut associer à chaque $(x, v) \notin \mathrm{CL}\downarrow(d, R)$, un entier $i \geq 0$ qui correspond à l'étape de l'itération chaotique où (x, v) a été retiré de l'environnement. Chaque élément n'étant retiré qu'une seule fois, cet entier est unique pour chaque élément.

Définition 6.7 *Soit* $(x, v) \notin \mathrm{CL}\downarrow(d, R)$*. On appelle* étape*, notée* $\mathrm{step}(x, v)$*, soit l'entier* $i \geq 1$ *tel que* $(x, v) \in d^{i-1} \setminus d^i$*, soit l'entier* 0 *si* $(x, v) \notin d^0$*.*

Une itération chaotique peut être considérée comme la construction incrémentale d'arbres explicatifs (calculés). On définit l'ensemble d'arbres explicatifs $\mathcal{E}^i$ qui est construit à une étape $i \in \mathbb{N}$. De façon, évidente, avant le calcul cet ensemble contient uniquement les arbres réduits à une feuille étiquetée par un élément qui n'est pas dans d (i.e. les arbres construits par les règles de $\mathcal{R}_d$). A chaque étape $i \in \mathbb{N}$, on construit les nouveaux arbres à partir des arbres des précédentes étapes et des règles associées à l'opérateur de consistance locale utilisé à ce pas de l'itération (i.e. l'opérateur de consistance locale désigné par r^i dans le run).

Définition 6.8 *La famille d'*arbres explicatifs calculés $(\mathcal{E}^i)_{i \in \mathbb{N}}$ *est définie par :*

- $\mathcal{E}^0 = \{\mathrm{cons}(h, \emptyset) \mid h \notin d\}$,
- $\mathcal{E}^{i+1} = \mathcal{E}^i \cup \{\mathrm{cons}(h, T) \mid h \in d^i, T \subseteq \mathcal{E}^i, h \leftarrow \{\mathrm{root}(t) \mid t \in T\} \in \mathcal{R}_{r^{i+1}}\}$.

Dans le second item, $h \in d^i$ assure de ne pas construire d'arbre où un élément apparaîtrait dans un arbre explicatif enraciné par ce même élément. Autrement dit, le retrait d'un élément ne peut pas être la conséquence (indirecte) de son propre retrait (comme le deuxième arbre de la figure 6.1).

On devine clairement le lien entre ces arbres et les éléments retirés de l'environnement. On prouve que les racines des arbres de $\mathcal{E}^i$ sont exactement les éléments retirés de l'environnement aux étapes $j \leq i$ de l'itération.

Lemme 6.6 $\{\mathrm{root}(t) \mid t \in \mathcal{E}^i\} = \overline{d^i}$.

Preuve. Par induction sur i : $\mathcal{E}^0$ vérifie la propriété.
On suppose $\{\mathrm{root}(t) \mid t \in \mathcal{E}^i\} = \overline{d^i}$ et on prouve

- $\{\mathrm{root}(t) \mid t \in \mathcal{E}^{i+1}\} \subseteq \overline{d^{i+1}}$. Soit h la racine de t tel que $t \in \mathcal{E}^{i+1}$. Il existe deux cas :
  - $t \in \mathcal{E}^i$, alors $h \in \overline{d^i}$, donc $h \in \overline{d^{i+1}}$ car $\overline{d^i} \subseteq \overline{d^{i+1}}$.
  - $t \notin \mathcal{E}^i$. Il existe $h \leftarrow B \in \mathcal{R}_{r^{i+1}}, h \in d^i$ et $\forall b \in B, b = \mathrm{root}(t_b), t_b \in \mathcal{E}^i$. donc $b \in \overline{d^i}$, donc $h \in \widetilde{r^{i+1}}(\overline{d^i}) = \overline{r^{i+1}(d^i)} \subseteq \overline{d^{i+1}}$.
- $\overline{d^{i+1}} \subseteq \{\mathrm{root}(t) \mid t \in \mathcal{E}^{i+1}\}$. Soit $h \in \overline{d^{i+1}}$. Il existe deux cas :
  - $h \in \overline{d^i}$, alors $h \in \{\mathrm{root}(t) \mid t \in \mathcal{E}^i\} \subseteq \{\mathrm{root}(t) \mid t \in \mathcal{E}^{i+1}\}$.
  - $h \in d^i$, alors $h \in \widetilde{r^{i+1}}(\overline{d^i})$, donc $\exists h \leftarrow B \in \mathcal{R}_{r^{i+1}}$ et $\forall b \in B, b \in \overline{d^i}$. donc il existe un ensemble T tel que $B = \{\mathrm{root}(t) \mid t \in T\}$ et $\mathrm{cons}(h, T) \in \mathcal{E}^{i+1}$, c'est-à-dire, $h \in \{\mathrm{root}(t) \mid t \in \mathcal{E}^{i+1}\}$.

□

Le lemme précédent est reformulé dans le corollaire suivant qui assure que chaque élément retiré de d durant une itération chaotique est la racine d'un arbre explicatif calculé de $\cup_{i \in \mathbb{N}} \mathcal{E}^i$.

Corollaire 6.7 $\{\mathrm{root}(t) \mid t \in \cup_{i \in \mathbb{N}} \mathcal{E}^i\} = \overline{\mathrm{CL}\downarrow(d, R)}$.

Preuve.

$$\begin{aligned}
\{\mathrm{root}(t) \mid t \in \cup_{i \in \mathbb{N}} \mathcal{E}^i\} &= \cup_{i \in \mathbb{N}} \{\mathrm{root}(t) \mid t \in \mathcal{E}^i\} \\
&= \cup_{i \in \mathbb{N}} \overline{d^i} \text{ par le lemme 6.6} \\
&= \overline{\cap_{i \in \mathbb{N}} d^i} \\
&= \overline{\mathrm{CL}\downarrow(d, R)}
\end{aligned}$$

□

Ce corollaire est important car il assure que, quel que soit l'itération chaotique utilisée, on peut construire des arbres explicatifs incrémentalement pour chaque élément qui n'est pas dans la clôture. Comme évoqué précédemment, tous les arbres explicatifs ne correspondent pas à une itération chaotique, mais pour chaque racine d'un arbre explicatif, on peut trouver (dans $\cup_{i\in\mathbb{N}}\mathcal{E}^i$) un arbre, appelé dans ce cas arbre explicatif calculé, enraciné par le même élément et correspondant à une itération chaotique. Cet arbre explicatif a été construit à l'étape de l'itération où sa racine a été retirée de l'environnement.

Chaque élément n'est donc retiré qu'à une seule étape de l'itération, par un seul opérateur. Dans le cas de l'arc-consistance, on a vu que pour chaque opérateur, il n'existait qu'une seule règle retirant un élément. Mais dans le cas général, par exemple l'hyper arc-consistance, il peut exister plusieurs règles ayant un même élément pour tête. Il est donc possible qu'un élément puisse être retiré par plusieurs règles d'un même opérateur. Il se peut donc qu'il existe dans $\mathcal{E}^i$ plusieurs arbres ayant la même racine.

A partir d'un arbre explicatif calculé, on peut obtenir les opérateurs de consistance utilisés grâce à la fonction step.

Théorème 6.8 *Soit un arbre explicatif $t \in \cup_{i\in\mathbb{N}}\mathcal{E}^i$. L'ensemble d'opérateurs de consistance locale $\{r^{\mathrm{step}(x,v)} \mid (x,v)$ a une occurrence dans t et $\mathrm{step}(x,v) > 0\}$ est un ensemble explicatif pour $\mathrm{root}(t)$.*

Preuve. D'après la définition 6.6 et le théorème 6.5, □

Comme on l'a déjà précisé, cet ensemble explicatif n'est pas forcément minimal pour le retrait de l'élément.

D'un point de vue théorique, l'objet fondamental est l'arbre explicatif. C'est lui qui sera utilisé pour le diagnostic déclaratif.

6.5 Enumération

Les deux notions d'explications définies précédemment prennent en compte exclusivement la réduction de domaine. On peut les étendre en considérant également des opérateurs d'énumération.

On ne considérera ce cadre (réduction de domaine et énumération) que dans le cas de la résolution d'un CSP sur $(V, \mathbb{D})$. Les explications ne sont donc plus définies par rapport à un environnement quelconque $d \subseteq \mathbb{D}$ mais toujours par rapport à $\mathbb{D}$. On ne le précisera donc pas. Par contre, l'ensemble d'opérateurs d'énumération peut varier selon la branche de l'arbre de recherche où l'on se trouve, il est donc nécessaire de le préciser. Soit N un ensemble d'opérateurs d'énumération.

Un ensemble explicatif pour (x, v) dans N est un ensemble d'opérateurs $E \subseteq R \cup N$ tel que $(x, v) \notin \mathrm{CL}\downarrow(\mathbb{D}, E)$.

On peut revenir sur les explications éliminantes (et donc les nogoods) proposées dans le cadre arc-consistance. Une explication éliminante est un ensemble de contraintes qui empêchent l'instanciation d'une variable à une valeur.

On rappelle que dans le cadre de l'hyper arc-consistance (et par conséquent de l'arc-consistance), chaque opérateur de consistance locale est associé à une seule contrainte. De plus on a déjà remarqué qu'un opérateur d'énumération sur (x, v) peut être considéré comme un opérateur de consistance locale pour la contrainte $x = v$. Si on considère l'ensemble de contraintes ainsi associées aux opérateurs d'un ensemble explicatif alors on obtient exactement une explication éliminante.

Notre définition est donc bien :

- plus générale car elle prend en compte toute notion de consistance ;
- plus précise car les opérateurs sont plus précis que les contraintes dans le cas de l'hyper arc-consistance.

Un ensemble explicatif E peut se décomposer en $R' \subseteq R$ et $N' \subseteq N$. En reprenant le lemme 4.12 : $\mathrm{CL}\downarrow(\mathbb{D}, R \cup N) = \mathrm{CL}\downarrow(\mathrm{CL}\downarrow(\mathbb{D}, N), R)$. Dès lors, si R' est un ensemble explicatif pour (x, v) par rapport à N' alors R' est un ensemble explicatif par rapport à l'environnement $\mathrm{CL}\downarrow(\mathbb{D}, N')$.

Cette remarque, qui aura son importance surtout pour la mise au point, souligne que les opérateurs d'énumération qui sont appliqués comme des opérateurs de consistance locale lors du calcul peuvent aussi être "oubliés". On peut en effet considérer que les calculs commencent à partir d'un environnement obtenu en appliquant d'abord tous ces opérateurs d'énumération.

Un opérateur d'énumération étant un opérateur constant, donc monotone, il est aussi possible de lui associer un système de règles. Ici encore, ce système de règles définit en fait le dual de l'opérateur d'énumération. Mais comme pour le cas des opérateurs de consistance locale, ce dual étant unique, on dira qu'un ensemble de règles de déduction est associé à un opérateur d'énumération.

Si r est un opérateur d'énumération sur (x, v) alors il peut lui être associé l'ensemble de règles de déduction $\{(x, v') \leftarrow \emptyset \mid (x, v') \in \mathbb{D}|_{\{x\}} \text{ et } v' \neq v\}$. Ce système de règles est un ensemble de faits.

Les définitions de run, d'itérations peuvent être étendues en prenant en compte l'utilisation d'opérateurs d'énumération en plus des opérateurs de consistance locale.

Pour tout élément retiré du domaine par un ensemble d'opérateurs de consistance locale et d'énumération, il existe au moins un arbre explicatif dont certains peuvent être obtenus lors du calcul. De même, un ensemble explicatif peut en être extrait.

On peut noter que quand une valeur est retirée dans plusieurs branches de l'arbre de recherche, il existe au moins un arbre explicatif calculé pour chacun de ces retraits. Dans (Lesaint, 2002), une notion plus générale d'explication regroupant ces arbres explicatifs a été proposée. Cependant, le formalisme est compliqué et apporte peu de choses par rapport aux arbres explicatifs définis ici.

6.6 Application au problème de la conférence

On reprend le problème de conférence proposé dans la section 4.4. La stratégie choisie pour l'application des opérateurs consistera à les appliquer dans l'ordre d'apparition, puis à recommencer une fois arrivé à la fin jusqu'à l'obtention de la clôture (quand aucun des opérateurs ne réduit plus l'environnement).

Pour faciliter la lecture, on rappelle la liste des opérateurs :

```
(2)   |   AM in -{6},
(3)   |   AM in -{5},
(4)   |   MA in -{6},
(5)   |   MA in -{5},
(6)   |   AP in -{6},
(7)   |   AP in -{5},
(8)   |   PA in -{6},
(9)   |   PA in -{5},
(10)  |   PM in 0..max(MA)-1, MA in min(PM)+1..infinity,
(11)  |   PM in 0..max(MP)-1, MP in min(PM)+1..infinity,
(12)  |   AM in 0..max(MA)-1, MA in min(AM)+1..infinity,
(13)  |   AM in 0..max(MP)-1, MP in min(AM)+1..infinity,
(14)  |   PA in 0..max(AP)-1, AP in min(PA)+1..infinity,
(15)  |   PA in 0..max(AM)-1, AM in min(PA)+1..infinity,
(16)  |   AM in -{val(MA)}, MA in -{val(AM)},
(17)  |   MA in -{val(PA)}, PA in -{val(MA)},
(18)  |   MP in -{val(AP)}, AP in -{val(MP)},
(19)  |   AM in -{val(PM)}, PM in -{val(AM)},
(20)  |   AP in -{val(MA)}, MA in -{val(AP)},
(21)  |   PM in -{val(AP)}, AP in -{val(PM)},
(22)  |   PA in -{val(MP)}, MP in -{val(PA)},
```

L'application des opérateurs des lignes `(2)` à `(9)` créée les arbres de la figure 6.2.

$$\overline{(AM,6)} \quad \overline{(AM,5)} \quad \overline{(MA,6)} \quad \overline{(MA,5)} \quad \overline{(AP,6)} \quad \overline{(AP,5)} \quad \overline{(PA,6)} \quad \overline{(PA,5)}$$

Fig. 6.2 – Arbres créés par les opérateurs `(2)` à `(9)`

L'application des deux opérateurs de la ligne `(10)` construit les arbres explicatifs de la figure 6.3

Le retrait de $(PM,4)$ est donc causé par les retraits de $(MA,5)$ et $(MA,6)$.

L'application des opérateurs suivants continue à retirer des éléments de $\mathbb{D}$ en construisant simultanément les explications pour ces retraits. Des arbres explicatifs plus conséquents peuvent être obtenus, comme pour le retrait de $(PA,3)$ donné dans la figure 6.4. Cet arbre est construit à partir des arbres explicatifs de $(AM,4)$, $(AM,5)$ et $(AM,6)$.

$$\frac{}{(PM,6)} \qquad \frac{\overline{(MA,6)}}{(PM,5)} \qquad \frac{\overline{(MA,5)} \quad \overline{(MA,6)}}{(PM,4)} \qquad \frac{}{(MA,1)}$$

Fig. 6.3 – Arbres créés par les opérateurs de (10)

$$\frac{\dfrac{\overline{(MA,5)} \quad \overline{(MA,6)}}{(AM,4)} \qquad \overline{(AM,5)} \qquad \overline{(AM,6)}}{(PA,3)}$$

Fig. 6.4 – Arbre explicatif du retrait de $(PA,3)$

Etant donné que l'on se trouve dans le cadre de l'arc-consistance, on peut associer chaque opérateur à une contrainte et donc chaque règle définissant cet opérateur à une contrainte. On peut donc réécrire les arbres explicatifs en y faisant figurer la contrainte utilisée à chaque retrait. Le précédent arbre explicatif peut donc se réécrire comme dans la figure 6.5.

$$\frac{\dfrac{\overline{(MA,5)}^{\,MA\neq 5} \quad \overline{(MA,6)}^{\,MA\neq 6}}{(AM,4)}{\scriptstyle AM<MA} \qquad \overline{(AM,5)}^{\,AM\neq 5} \qquad \overline{(AM,6)}^{\,AM\neq 6}}{(PA,3)}{\scriptstyle PA<AM}$$

Fig. 6.5 – Arbre explicatif du retrait de $(PA,3)$

Le retrait de $(PA,3)$ est causé par les retraits des valeurs de AM strictement supérieures à 3, c'est-à-dire $(AM,4)$, $(AM,5)$ et $(AM,6)$. Le retrait de $(AM,4)$ est causé par l'absence de valeurs supérieures dans l'environnement de MA, ces valeurs ayant été retirées par les contraintes $MA \neq 5$ et $MA \neq 6$. Les retraits des valeurs 5 et 6 pour AM ont eux été causés par les contraintes $AM \neq 5$ et $AM \neq 6$.

Autrement dit, c'est parce qu'Alain ne peut pas venir le troisième jour et que Michel veut entendre l'exposé d'Alain avant de faire le sien que l'exposé de Michel à Alain ne peut pas avoir lieu l'une des trois dernières demi-journées (4, 5 et 6). Ce qui entraine que l'exposé de Pierre à Alain qui doit se faire avant ne peut pas se tenir la troisième demi-journée.

6.7 Conclusion

Dans ce chapitre, deux notions d'explications ont été définies.

Un ensemble explicatif pour un élément du retiré domaine est un ensemble d'opérateurs tel que toute itération de cet ensemble retire toujours la valeur considérée. Cette définition déclarative est assez proche des explications décrites dans le chapitre précédent qui considèrent des contraintes (des contraintes du problème ou des contraintes d'énumération) à la place des opérateurs.

Les arbres explicatifs sont un outil plus précis pour comprendre les raisons d'un retrait de valeur. En effet, un arbre explicatif décrit exactement le raisonnement qui permet de retirer une valeur. Ces arbres sont obtenus de façon naturelle en considérant une vision duale de la réduction de domaine, en s'intéressant aux valeurs retirées plutôt qu'aux environnements calculés.

D'un point de vue pratique, il a été expliqué comment extraire de tels arbres du calcul et comment extraire un ensemble explicatif d'un arbre explicatif.

Chapitre 7

Relaxation de contraintes

Sommaire

Dans ce chapitre, une utilisation des explications est proposée. En tant qu'outil théorique, les explications (les ensembles explicatifs ici) permettent de prouver la correction des algorithmes de retrait de contrainte (Debruyne *et al.*, 2003). Ces algorithmes sont utilisés pour les CSP dynamiques décrits dans la section 5.3, mais aussi pour la recherche de solutions aux problèmes sur-contraints ou encore pour les algorithmes de recherche systématique basés sur les explications comme dans le système `PaLM`.

7.1 Propagation

Avant de considérer la relaxation de contraintes, il est nécessaire d'évoquer les mécanismes de propagation utilisés en pratique. Pour la réduction de domaine, on a considéré que les opérateurs étaient appliqués selon un run équitable. En pratique ce run n'est pas connu à l'avance mais construit pendant le calcul à l'aide par exemple d'une queue de propagation. Cette dernière contient un ensemble d'opérateurs pouvant être appliqués. Plus précisément, les opérateurs non présents dans cette queue de propagation ne réduisent pas l'environnement courant, il est donc inutile de les appliquer (pour l'instant). Suivant l'algorithme choisi pour calculer cette queue de propagation, elle peut contenir plus ou moins d'opérateurs. Si l'environnement est $d \subseteq \mathbb{D}$, alors chaque opérateur de consistance locale r hors de la queue de propagation est tel que d est r-consistant. Si tous les opérateurs du programme R sont hors de la queue de propagation alors l'environnement est R-consistant, la clôture a donc été calculée.

Cette propriété des opérateurs hors de la queue de propagation est fondamentale pour assurer que le solveur calcule bien la clôture.

La queue de propagation est construite comme suit : au départ, tous les opérateurs sont présents dans celle-ci. Les opérateurs étant idempotents en pratique, à chaque fois qu'un opérateur est choisi pour être appliqué, il est supprimé de la queue de propagation. Deux situations peuvent se présenter :

- Si son application ne modifie pas l'environnement d, les opérateurs hors de la queue de propagation sont toujours tels que d est r-consistant et sont donc laissés hors de la queue de propagation.
- Par contre si l'opérateur réduit l'environnement d (l'environnement obtenu est noté d'), il se peut que d' ne soit pas r-consistant pour certains opérateurs hors de la queue de propagation. Ceux-ci doivent donc être réintroduits dans la queue de propagation. Bien entendu, les algorithmes ne testent pas réellement pour chaque opérateur si l'environnement est r-consistant (cela serait aussi coûteux que de les appliquer). L'ensemble d'opérateurs ré-introduits dans la queue de propagation peut donc contenir des opérateurs qui ne réduisent pas l'environnement. Cependant une stratégie consistant à remettre tous les opérateurs dans la queue de propagation se montre peu efficace et un compromis doit

donc être trouvé. Souvent, la réintroduction d'un opérateur r dans la queue de propagation se fait si l'environnement d'une des variables de $\mathrm{in}(r)$ a été modifié. Une stratégie plus précise (utilisée dans `PaLM` par exemple) consiste à considérer les bornes de l'environnement de ces variables.

Pour la suite, on va considérer un calcul à partir de l'environnement $d^0 = d$ par le programme R. A l'étape i, on passe de l'environnement d^{i-1} à l'environnement d^i en appliquant l'opérateur r^i. On notera Q^i l'ensemble d'opérateurs de consistance locale présents dans la queue de propagation après l'étape i. Les opérateurs de consistance locale r hors de cette queue de propagation appartiennent donc à l'ensemble $R \setminus Q^i$ et sont tels que d^i est r-consistant.

Le calcul s'arrête à la première étape n pour laquelle $Q^n = \emptyset$ ou pour laquelle l'environnement d'une variable est vide, car dans ce dernier cas aucune solution ne peut être obtenue (i.e. c'est un échec).

7.2 Correction de la relaxation de contraintes

On considère le calcul précédent à partir de l'environnement $d^0 = d$ selon l'ensemble d'opérateurs R. Supposons qu'une contrainte ou plusieurs contraintes doivent être relaxées à l'étape i de cette itération. Le calcul est donc stoppé à l'étape i avec l'environnement d^i et la queue de propagation contenant l'ensemble d'opérateurs Q^i. Remarquons qu'il n'est pas nécessaire que la clôture de d par R soit atteinte. On a donc simplement $\mathrm{CL}\!\downarrow\!(d, R) \subseteq d^i \subseteq d$.

A cette étape i, un ensemble des contraintes, noté C_{relax}, doit être relaxé. On suit alors le processus de relaxation de contraintes décrit dans le chapitre 5, c'est-à-dire :

- suppression des opérateurs
- réintroduction des valeurs
- repropagation

On détermine ici comment sont enlevées ces contraintes, quel ensemble de valeurs réintroduire dans l'environnement et finalement quel ensemble d'opérateurs réintroduire dans la queue de propagation. La minimalité de ces ensembles n'est pas garantie, cependant, les algorithmes classiques évoqués dans le chapitre 5 réintroduisent un sur-ensemble de valeurs dans l'environnement et un sur-ensemble d'opérateurs dans la queue de propagation. Ces algorithmes sont donc prouvés corrects. Entre autre, l'algorithme de relaxation de contraintes du système `PaLM` est correct.

7.2.1 Suppression des opérateurs

Déconnecter l'ensemble de contraintes C_{relax} revient à retirer l'ensemble d'opérateurs associés à ces contraintes, c'est-à-dire, si on note R_c les opérateurs associés à la contrainte $c \in C$, l'ensemble d'opérateurs $R_{\text{relax}} = \bigcup_{c \in C_{\text{relax}}} R_c$.

L'ensemble d'opérateurs restant est alors $R^{\text{new}} \subseteq R$ avec $R^{\text{new}} = \bigcup_{c \in C \setminus C_{\text{relax}}} R_c$. Le retrait des contraintes C_{relax} revient à calculer la clôture de d par R^{new}.

7.2.2 Réintroduction des valeurs

Comme annoncé plutôt, on veut bénéficier du calcul de d^i au lieu de reprendre le calcul depuis d. Grâce aux ensembles explicatifs, il est possible de connaître les éléments de $d \setminus d^i$ qui ont été retirés (directement ou non) par un opérateur relaxé (c'est-à-dire un opérateur de l'ensemble $R \setminus R^{\text{new}}$). Pour chaque $h \in d \setminus d^i$ un ensemble explicatif est choisi et noté $\text{expl}(h)$. Tous les éléments de $h \in d \setminus d^i$ pour lesquels un opérateur relaxé apparaît dans l'ensemble explicatif sont réintroduits. Cet ensemble de valeurs est défini par $d' = \{h \in d \mid \exists r \in R \setminus R^{\text{new}}, r \in \text{expl}(h)\}$. Tous les algorithmes de relaxation de contraintes reviennent à calculer un sur-ensemble (souvent strict) de d'. Le théorème suivant assure l'obtention de la même clôture que le calcul recommence à partir de d ou de $d^i \cup d'$.

Théorème 7.1 $\text{CL}\downarrow(d, R^{\text{new}}) = \text{CL}\downarrow(d^i \cup d', R^{\text{new}})$

Preuve. $\supseteq$: car $d^i \cup d' \subseteq d$ et l'opérateur de clôture est monotone.
$\subseteq$: on prouve $\text{CL}\downarrow(d, R^{\text{new}}) \subseteq d^i \cup d'$. Par l'absurde : soit $h \in \text{CL}\downarrow(d, R^{\text{new}})$ mais $h \notin d^i \cup d'$. $h \notin d^i$, donc $\text{expl}(h)$ existe. soit $\text{expl}(h) \subseteq R^{\text{new}}$, alors $h \notin \text{CL}\downarrow(d, R^{\text{new}})$: contradiction ; soit $\text{expl}(h) \not\subseteq R^{\text{new}}$, alors $h \in d'$: contradiction. Donc, $\text{CL}\downarrow(d, R^{\text{new}}) \subseteq d^i \cup d'$ et donc, par monotonie : $\text{CL}\downarrow(\text{CL}\downarrow(d, R^{\text{new}}), R^{\text{new}}) \subseteq \text{CL}\downarrow(d^i \cup d', R^{\text{new}})$. □

Ce théorème assure que l'ensemble de valeurs d' à réintroduire est suffisant et que tous les algorithmes réintroduisant un sur-ensemble de d' sont donc corrects.

7.2.3 Repropagation

En pratique, l'itération se fait selon une séquence d'opérateurs qui est dynamiquement calculée par la queue de propagation. A la $i^{\text{ème}}$ étape, avant de réintroduire les éléments, l'ensemble d'opérateurs présents dans la queue de propagation est Q^i. Après le retrait des opérateurs à relaxer, les opérateurs qui ne sont pas dans la queue de propagation (i.e. $R^{new} \setminus Q^i$) ne peuvent pas retirer d'éléments de d^i. En effet, pour tous les opérateurs r absents de la queue de propagation à l'étape i, d^i est r-consistant. Cependant, il est possible qu'ils puissent retirer des éléments de d' (les élément réintroduits). Ces opérateurs, formant l'ensemble $R' = \{r \in R^{new} \mid \exists h \leftarrow B \in \mathcal{R}_r, h \in d'\}$, sont réintégrer dans la queue de propagation. Le théorème suivant assure que les opérateurs n'étant pas dans $Q^i \cup R'$ ne modifient pas le nouvel environnement (i.e. $d^i \cup d'$) et qu'il est donc inutile de les réintégrer dans la queue de propagation.

Théorème 7.2 $\forall r \in R^{\text{new}} \setminus (Q^i \cup R'), d^i \cup d' \subseteq r(d^i \cup d')$

> *Preuve.* on prouve $d^i \subseteq r(d^i \cup d')$:
> $d^i \subseteq r(d^i)$ car $R^{\text{new}} \setminus (Q^i \cup R') \subseteq R^{\text{new}} \setminus Q^i$.
> $d^i \subseteq r(d^i \cup d')$ car r est monotone.
> on prouve $d' \subseteq r(d^i \cup d')$:
> Par l'absurde : soit $h \in d'$ mais $h \notin r(d^i \cup d')$. Alors il existe $h \leftarrow B \in \mathcal{R}_r$, c'est-à-dire $r \in \{r' \in R^{\text{new}} \mid \exists h \leftarrow B \in \mathcal{R}_{r'}, h \in d'\}$, alors $r \notin R^{\text{new}} \setminus (Q^i \cup R')$: contradiction. Donc $d' \subseteq r(d^i \cup d')$. □

Ce théorème assure donc la correction de tous les algorithmes réintroduisant un sur-ensemble de R' dans la queue de propagation.

7.3 Discussion

La relaxation de contrainte réalisée dans `PaLM` suivant les travaux de (Jussien, 1997) entre dans ce cadre. L'ensemble de valeurs réintroduites est exactement d' et les opérateurs ajoutés à la queue de propagation sont, à quelques optimisations près, ceux donnés précédemment. Les deux théorèmes 7.1 et 7.2 assurent donc la correction de cette méthode.

Les travaux décrits dans (Fages *et al.*, 1998) et (Codognet *et al.*, 1993) utilisent tous les deux une notion de *graphe de dépendance* pour exécuter la relaxation de contraintes. Ce graphe de dépendance permet une moindre précision que la "méthode" décrite ici. En effet, la réintroduction de valeurs est réalisée de la façon suivante (pour simplifier on prend le cas d'une seule contrainte relaxée) : Si la contrainte c relaxée a retiré des valeurs de la variable x, alors toutes ces valeurs sont réintroduites dans l'environnement. Si une autre contrainte (non relaxée) a retiré des valeurs d'une autre variable y à cause de réductions de l'environnement de x alors toutes ces valeurs sont réintroduites, même si ces retraits étaient causés par une valeur de x retirée par une autre contrainte que c. Cet ensemble de valeurs restaurées est donc clairement un sur-ensemble de notre ensemble d'. Donc, d'après le théorème 7.1, cet algorithme est correct.

Les algorithmes DnAC-* (DnAC-4 (Bessière, 1991), DnAC-6 (Debruyne, 1996)) utilisent un mécanisme de *justifications* comme décrit brièvement plus tôt. Une justification pour le retrait d'une valeur est la contrainte associée à l'opérateur de consistance locale qui a retiré cette valeur lors du calcul. Cela revient donc à n'enregistrer que les effets directs des opérateurs. Les effets indirects doivent donc être calculés récursivement. Dans ce cas aussi, tous les éléments de d' sont réintroduits et selon le théorème 7.1, les algorithmes DnAC-* obtiennent la même clôture qu'un calcul à partir de l'environnement initial. Pour ré-obtenir la consistance (i.e. repropager) les algorithmes DnAC-* ne recherchent pas un support de chaque valeur sur chaque contrainte. Ils vérifient seulement si les valeurs réintroduites ont encore au moins un support sur chaque contrainte et propagent bien sûr les éventuels retraits. Donc, DnAC-* recommence la propagation seulement quand celle-ci peut mener au retraits de valeurs réintroduites. Malgré cela, le théorème 7.2 assure que cette méthode est suffisante.

Les trois méthodes précédentes enregistrent de l'information (explications, graphe de dépendance ou justifications) durant la réduction de domaine. Une autre méthode pour la relaxation de contraintes a été proposée dans (Georget *et al.*, 1999). La principale différence est la non modification des mécanismes du solveur (i.e. pas d'enregistrement d'information). En effet, les dépendances de contraintes sont calculées seulement au moment de la relaxation. Dans ces conditions, la comparaison avec la méthode proposée est difficile. Cependant, la correction de leur algorithme est prouvée par trois lemmes qui sont vérifiés dans notre cadre.

7.4 Conclusion

Les ensembles explicatifs se sont montrés utiles d'un point de vue théorique pour donner des ensembles d'éléments à réintroduire dans l'environnement et d'opérateurs à réintégrer dans la queue de propagation. De plus, ils le sont aussi d'un point de vue pratique puisqu'ils sont utilisées sous la même forme dans le système `PaLM`.

Tout algorithme reprenant le calcul à partir d'un environnement incluant $d^i \cup d'$ et avec une queue de propagation comprenant au moins $Q^i \cup R'$ a été prouvé correct par les deux précédents théorèmes.

Troisième partie

Mise au point

Chapitre 8

Etat de l'Art de la Mise au Point

Sommaire

Ce chapitre se consacre à un rappel des techniques de mise au point existantes en programmation par contraintes. Il existe peu d'outils de mise au point en programmation par contraintes, d'une part parce que ce paradigme de programmation est assez récent, d'autre part parce que les techniques habituelles sont ici peu efficaces.

En programmation par contraintes, on peut distinguer principalement deux types de mise au point : la mise au point de performance et la mise au point de correction. La première a pour but d'améliorer l'efficacité de la résolution du problème, la seconde de corriger d'éventuelles erreurs dans le programme.

La problématique et la difficulté de la mise au point en programmation avec contraintes sont abordées par le biais des travaux réalisés par Meier (Meier, 1995).

Puis les réalisations effectuées dans le cadre du projet ESPRIT DiSCiPl (Deransart *et al.*, 2000) sont présentées. Il faut noter que ces travaux concernent la programmation logique avec contraintes (suivent le schéma clp(X)). Pour certains le solveur n'est qu'un paramètre X du schéma clp, ils ne sont donc que brièvement évoqués.

Enfin d'autres travaux sont rapidement présentés dont des utilisations des explications pour la mise au point.

8.1 Les travaux de Meier

Les premiers travaux sur la mise au point en programmation par contraintes sont à attribuer à Meier (Meier, 1995). Une méthodologie pour le débogage de performance est proposée et l'environnement Grace (Meier, 1996) a été développé pour le système ECL^{i}PS^e (Aggoun *et al.*, 2001). Ces travaux ont l'intérêt de poser clairement la problématique de la mise au point en programmation par contraintes.

8.1.1 Caractéristiques des programmes par contraintes

La difficulté soulignée dans (Meier, 1995) concerne le comportement non algorithmique des programmes par contraintes.

La mise au point de programmes algorithmiques peut elle aussi se faire de façon algorithmique. Cette mise au point peut être locale (par exemple, chaque fonction peut être considérée séparément). Il est possible de décider à chaque point de l'exécution si l'état courant est correct ou pas. Enfin les outils de traçage sont adaptés pour les programmes algorithmiques.

Par comparaison avec les langages algorithmiques, quatre remarques sont formulées à propos de la mise au point de programmes par contraintes :

- La mise au point de programmes par contraintes, et plus spécialement la mise au point de performance, n'est pas algorithmique et peut être difficilement automatisable.

- Elle est globale, il est nécessaire de considérer le programme dans son ensemble ainsi que l'éxécution.
- Spécialement pour la mise au point de performance, il n'est pas possible de décider si un état particulier de l'éxécution est correct ou non. Un état de l'éxécution est, pour Meier, un point de l'espace de recherche et on ne peut pas décider si une solution va être rapidement trouvée ou pas.
- Similairement à la recherche de solution par le solveur, la mise au point elle même est aussi un problème de recherche. L'outil de mise au point doit donc être hautement interactif et ouvert.

8.1.2 Approches de la mise au point

L'auteur considère la mise au point selon deux approches, une approche expérimentale et une approche analytique. Ces deux approches doivent être intégrées dans un environnement de mise au point.

L'approche expérimentale, utilisée pour la mise au point de performance, consiste à tester différentes méthodes. Elle ne demande pas de connaissance de la part de l'utilisateur mais nécessite un large éventail de méthodes. Ces méthodes s'appliquent à trois parties d'un programme par contraintes :

- le choix de modélisation du problème ;
- l'effet de contraintes particulières, la quantité de propagation effectuée, l'ajout de nouvelles contraintes ou de contraintes redondantes ;
- la stratégie de labeling, que ce soit dans le choix de la variable à instancier ou dans le choix des valeurs à essayer.

L'approche analytique suppose une plus grande connaissance des mécanismes utilisés. Un environnement de mise au point approprié devrait être capable de structurer la trace à différents niveaux pour permettre à l'utilisateur d'en déduire de nouvelles informations. Principalement, la mise au point analytique doit permettre à l'utilisateur de :

- découvrir des contraintes redondantes ;
- développer de nouvelles stratégies de labeling plus efficaces (par exemple en affichant l'arbre de recherche de telle façon que l'utilisateur puisse comprendre pourquoi il ne trouve pas de solution rapidement) ;
- trouver la propagation appropriée ;
- voir les raisons d'un échec ;
- comprendre les raisons d'une réponse fausse.

Les trois premiers items se révèlent surtout utiles pour la mise au point de performance alors que les deux derniers sont essentiels à la mise au point de correction.

8.1.3 L'environnement de mise au point

Le support attendu d'un environnement de mise au point peut être vu sous différents angles :

- L'outil doit être capable d'afficher toutes les informations nécessaires. Cependant ces informations étant souvent complexes et larges, elles doivent pouvoir ête affichées à la demande. Ces informations sont les solutions, l'ordre d'instanciation des variables (et des valeurs choisies), les valeurs de certaines ou toutes les variables, les mises à jour des environnements causés par le dernier pas de labeling, la propagation de contraintes.
- L'outil doit être hautement interactif et flexible.
- Idéalement, il devrait pouvoir controler l'application.

Ces considérations ont abouti à la création de l'environnement graphique de traçage de contraintes GRACE qui répond à la plupart d'entre elles. Cet environnement s'est montré efficace et utile (Meier, 1994) pour la mise au point et le traçage de programmes logique avec contrainte.

8.2 Le projet DISCIPL

De nombreux travaux[1] ont été réalisés dans le cadre du projet européen DISCIPL[2]. Ce projet a proposé une méthodologie de la mise au point et a donné lieu à différents types d'outils, principalement :

- les outils basés sur les assertions ;
- le diagnostic déclaratif ;
- les outils de visualisation.

Le but du projet était de fournir des outils pour la mise au point de correction et de performance. Deux approches ont été proposées pour la première : la mise au point statique (c'est-à-dire sans éxécution du programme) et la mise au point basée sur des symptômes obtenus après éxécution. Ces approches ont été concrétisées respectivement par des outils utilisant les assertions et le diagnostic déclaratif. La mise au point de performance quand à elle a été concrétisée par des outils de visualisation.

8.2.1 Outils basés sur les assertions

Les outils de mise au point statique développés dans DISCIPL sont basés sur les assertions. Les assertions sont une description de propriétés qui doivent etre vérifiées par les réponses ou les calculs du programme. Ces assertions portent sur les prédicats

[1]Diverses publications issues de ce projet sont accessibles à l'URL http ://discipl.inria.fr et dans (Deransart *et al.*, 2000).

[2]Debugging Systems for Constraint Programming.

et sont donc essentiellement liées à l'aspect logique des programmes CLP. Les outils basés sur les assertions ont pour rôle de vérifier, de façon automatique si possible, si ces assertions sont respectées par le programme. Plusieurs outils ont été développés : CiaoPP et CHIPRE (Bueno *et al.*, 2000), l'outil d'assertion de Prolog IV (Laï, 2000) et un outil de diagnostic statique localisant les erreurs de type pour CHIP (Drabent *et al.*, 2000).

8.2.2 Outil de diagnostic déclaratif

Le diagnostic déclaratif (Shapiro, 1982) est une technique employée pour localiser une erreur dans un programme sur la base de symptômes apparaîssant après une éxécution. L'idée est d'automatiser en partie l'analyse du calcul. Le système interroge un oracle sur des résultats intermédiaires. Ces réponses permettent d'orienter la recherche vers l'erreur. Cette méthode de diagnostic est dite déclarative car l'oracle n'est questionné que sur des résultats, l'aspect opérationnel lui est caché. Deux types de symptômes sont traités : les symptômes d'incorrection (une réponse fausse, c'est-à-dire non attendue par l'utilisateur) et les symptômes d'incomplétude (une réponse manquante, c'est-à-dire attendue par l'utilisateur et non obtenue). L'algorithme de diagnostic déclaratif (Tessier & Ferrand, 2000) utilisé dépend du type du symptôme traité. Ce sont les mêmes techniques qui sont utilisées ici pour le diagnostic de réponse fausse, à la différence qu'au lieu de déboguer au niveau des clauses d'un programme clp, on se situera au niveau des contraintes d'un CSP.

8.2.3 Outils de visualisation

Des outils de visualisation ont été développés afin d'aider l'utilisateur pour la mise au point. On peut d'abord citer un prototype de débogueur incluant les *s-box* (Benhamou & Goualard, 2000) pour clp(fd)[3]. L'idée des s-box est de structurer le store de contraintes (i.e. l'ensemble des contraintes) afin d'éviter un affichage en "vrac" de celui-ci. Une s-box est un ensemble de contraintes contenues dans un prédicat ainsi que les prédicats appelés à partir de celui-ci (qui sont eux-mêmes des s-box). Le store est ainsi hiérarchisé et la propagation peut alors être concentrée sur une s-box choisie par l'utilisateur, ceci peut permettre de localiser plus facilement une erreur. Les autres outils de visualisation réalisée dans DiSCiPl sont davantage liés à la sémantique opérationelle des solveurs et donc utilisables pour la mise au point de performance. Plusieurs outils ont été proposés pour visualiser l'arbre de recherche : *Prolog IV Search-Tree Visualiser* (Bouvier, 2000), CHIP *Labeling Visualiser* (Simonis & Aggoun, 2000), *INRIA Search-Tre Visualiser* (Aillaud & Deransart, 2000), *APT Visualiser* (Carro & Hermenegildo, 2000).

[3] clp(fd) est devenu GNU-Prolog par la suite.

8.3 Autres travaux et utilisation des explications pour la mise au point

Dans la lignée des outils de visualisation, on peut citer Oz explorer (Schulte, 1997) pour Oz (Smolka, 1995). Plusieurs outils de visualisation sont également en cours d'élaboration dans le cadre du projet RNTL OADymPPaC[4], on peut citer CLPGUI (Fages, 2002) qui permet de visualiser pas à pas l'état des domaines lors d'une résolution (en GNU-Prolog ou SICStus Prolog).

Un modèle de trace générique (Ducassé & Langevine, 2002) pour les solveurs sur domaines finis a également été proposé. Cette trace comporte différents types d'événements, l'ajout d'une contrainte, l'application d'opérateur, la pose de points de choix... L'intérêt de ce modèle est son application à tout type de solveur, ce qui permettra l'utilisation d'un même outil de visualisation quel que soit le solveur utilisé à partir du moment où le modèle de trace est conforme.

L'utilisation des explications pour la mise au point a été rapidement évoquée dans le chapitre 5. Dans (Ouis *et al.*, 2003), les auteurs montrent comment utiliser les explications k-relevantes dans un outil de diagnostic et d'aide à la mise au point. Elles permettent d'obtenir un ensemble de contraintes responsable du retrait d'une valeur ou d'un échec. La k-relevance peut permettre d'avoir plusieurs ensembles et donc de choisir le plus précis (le plus petit au sens de l'inclusion). Les explications permettent de connaître les retraits causés (directement ou non) par une contrainte.

Enfin, dans un session de mise au point, l'information fournie à l'utilisateur doit être la plus compréhensible possible. Les contraintes étant des contraintes de bas niveau, seul un utilisateur expérimenté peut les comprendre correctement. Un outil permettant de fournir des explications compréhensibles (les explications *user-friendly*) a été proposé dans (Ouis *et al.*, 2002). L'idée repose sur le fait que les problèmes rencontrés ont une nature hiérarchique et peuvent alors se représenter par une arborescence dont les feuilles sont les contraintes du problème. Cela permet une présentation plus conviviale et surtout concise d'ensembles de contraintes pouvant parfois être très important.

8.4 Conclusion

Plusieurs travaux concernant la mise au point en programmation par contraintes ont été présentés. La difficulté soulignée par Meier concerne l'aspect non algorithmique de ce paradigme. De plus la quantité d'information à traiter est souvent très importante et la non modularité des problèmes permet difficilement de ne s'intéresser qu'à des parties restreintes du code. Dès lors les techniques habituelles se retrouvent souvent limitées. Les outils proposées tentent donc d'une part de structurer l'infor-

[4]Outils pour l'Analyse Dynamique et la mise au Point de Programmes avec Contraintes.

mation (les s-box ou les explications *user-friendly*) et de proposer des outils efficaces de visualisation de l'arbre de recherche ou de la propagation. C'est essentiellement l'aspect mise au point de performance qui tire profit de ces outils. La mise au point de correction se retrouve bloquée par la quantité d'information à traiter et l'aspect non algorithmique de ces programmes. Dans le chapitre suivant, nous proposons une approche déclarative permettant ainsi de ne conserver que l'information réellement nécessaire à la mise au point de correction.

Chapitre 9

Diagnostic de réponses manquantes

Sommaire

La programmation par contraintes, comme tout langage déclaratif tire sa force de son indépendance vis-à-vis du calcul. Idéalement, un utilisateur programmant par contraintes n'a pas à connaître le fonctionnement du solveur. Sa seule tâche est de modéliser son problème en choisissant les contraintes appropriées. Le solveur se charge de trouver la ou les solutions.

Cependant, il n'est pas rare que le programmeur se trompe en modélisant son problème. On peut distinguer trois types d'erreurs :

- une ou plusieurs contraintes ont été oubliées,
- une ou plusieurs contraintes ont été mal écrites,
- une ou plusieurs contraintes ont été ajoutées à tort.

Les conséquences peuvent être diverses selon le type de l'erreur commise : certaines solutions obtenues ne sont en fait pas des solutions du problème réel ou au contraire certaines solutions du problème réel ne sont pas obtenues (avec comme cas extrême l'échec où aucune solution n'est obtenue). Si l'utilisateur décide de déboguer son programme, c'est parce que ce qu'il obtient ne correspond pas à ce qu'il attendait. Plus précisément, c'est l'apparition de symptômes d'erreur qui prévient l'utilisateur qu'une erreur a été faite lors de la formalisation de son problème. Un symptôme est une incohérence entre ce que l'utilisateur obtient et ce qu'il pensait obtenir.

Il est donc indispensable que l'utilisateur possède une certaine connaissance de ce que doit être le résultat. Cette connaissance est implicite du fait que le programmeur a été capable de formaliser son problème. Bien entendu on ne peut exiger qu'il ait une complète connaissance des solutions (ceci est cependant possible dans le cas de tests) mais seulement d'être capable de répondre à quelques questions.

Par contre, on ne peut pas lui demander de connaître les calculs du solveurs, d'une part parce qu'ils sont complexes et d'autre part parce qu'ils sont cachés. Une approche déclarative semble donc plus appropriée. Le but d'un outil de mise au point est d'aider le programmeur à retrouver son erreur. Dans le cadre du diagnostic déclaratif, cette recherche se fait à partir d'un symptôme. Ce symptôme peut être de deux types : un élément du domaine que le programmeur pensait appartenir à une solution, n'apparaît en fait dans aucune solution ou au contraire un élément apparaît dans une solution alors que l'utilisateur ne l'y attendait pas. On parle respectivement de réponse manquante et de réponse fausse.

Dans le cas d'une réponse manquante, le diagnostic déclaratif semble un outil adapté. Un symptôme de réponse manquante est un élément du domaine qui a été retiré alors qu'il n'aurait pas dû l'être selon le programmeur. Ayant été retiré lors du calcul, il existe donc un arbre explicatif (calculé) du retrait de cet élément. Cet arbre contient un opérateur (plus précisément une règle) ne correspondant pas à la réalité, c'est-à-dire une règle qui a effectué un retrait non souhaité. Le but du diagnostic déclaratif est alors d'aider l'utilisateur à localiser cette règle dans l'explication du symptôme. Cette règle est issue d'un opérateur lui même issu d'une contrainte (ou plusieurs dans le cas général), cet opérateur est donc responsable du retrait.

Pour les réponses fausses, l'utilisateur dispose d'un tuple complet (la solution fausse fournie par le solveur). Comme on considère ici des éléments du dimaine qui n'ont pas été retirés, les arbres explicatifs n'ont pas d'intérêt pour ce cas. Il faudrait donc s'orienter vers d'autres méthodes.

Dans ce chapitre, une formalisation de la sémantique attendue est proposée. Cette formalisation permet de juger de la correction du CSP et du programme par rapport aux attentes de l'utilisateur. Une méthode de diagnostic déclaratif de réponse manquante est alors décrite. Enfin l'explication d'échec est définie. Celle-ci peut se révéler utile pour la mise au point, mais aussi pour les applications concernant les problèmes sur-contraints.

9.1 Sémantique attendue

La notion de *sémantique attendue* est primordiale dès que l'on parle de mise au point. Elle traduit la connaissance du problème que doit avoir le programmeur faisant de la mise au point. Il faut noter que sans cette connaissance, il lui est impossible de dire si le programme se comporte correctement ou non et donc de déceler un quelconque problème. La mise au point, quelque soit l'outil utilisé, fait toujours appel à cette connaissance. Pour pouvoir analyser une trace par exemple, il faut être capable de porter un jugement sur les événements qu'elle décrit, il faut donc posséder une certaine connaissance.

Pour la décrire, il est important de rappeler que la programmation par contraintes est déclarative. En effet, un programmeur utilisant la programmation par contraintes n'est pas censé être la personne qui a implanté le solveur. Sa seule tâche a été de comprendre un problème donné et de le modéliser en choisissant les contraintes appropriées. Le solveur se charge de trouver la ou les solutions de ce problème. Pour programmer avec contraintes, il n'est donc pas indispensable de connaître le fonctionnement du solveur. La connaissance requise pour la mise au point doit être du même type. Il paraît donc peu cohérent d'utiliser des outils de mise au point de bas niveau consistant à analyser les calculs du solveur. De plus ces calculs étant souvent complexes, les traces peuvent atteindre des dimensions gigantesques et l'utilisation d'outils de visualisation n'est pas d'un grand secours quand on ne sait pas où regarder. La sémantique attendue ne peut donc pas être une connaissance du calcul mais de ce qui est calculé, c'est-à-dire des solutions.

La sémantique attendue est donc considérée comme étant un ensemble de solutions.

Définition 9.1 *On appelle* solutions attendues *un ensemble S d'instanciations sur V.*

Il est peu concevable de supposer qu'un utilisateur connaisse exactement l'ensemble des solutions. Ceci peut cependant être envisagé dans le cas de tests, quand une spécification du programme est connue ou quand une nouvelle version d'un programme existant est essayée. Dans ce cas, où une solution complète attendue et non obtenue est connue, un outil de mise au point très simple peut-être utilisé : il suffit de tester l'instanciation attendue sur chaque contrainte (ou opérateur). Les contraintes violées par l'instanciation sont alors responsables de l'absence de cette solution attendue. En terme d'opérateurs, cela revient à chercher l'opérateur $r \in R$ tel que l'instanciation n'est pas r-consistante. Mais en général, l'utilisateur ne connaît pas exactement les solutions attendues et l'utilisation d'un outil de diagnostic se révèle alors nécessaire.

Les solutions du CSP étant calculées par le programme, c'est à partir des résultats du calcul de celui-ci que se fait le diagnostic. Le programme calcule un environnement (une approximation des solutions). C'est sur cet ensemble que va se faire le diagnostic déclaratif. La comparaison entre solutions attendues et solutions obtenues s'étend donc en une comparaison entre la clôture calculée et la projection des solutions attendues sur le domaine $\mathbb{D}$. On utilise la notation $\bigcup S$ pour $\bigcup_{s \in S} s$.

Définition 9.2 *On appelle* environnement attendu*, la projection* $\bigcup S$ *de* S *sur le domaine* $\mathbb{D}$*.*

L'environnement attendu est donc une approximation des solutions attendues. Cette notion d'approximation est différente de l'approximation entre solutions du CSP et clôture du programme. Ici tous les éléments de $\bigcup S$ appartiennent à une solution attendue alors que tous les éléments de la clôture n'appartiennent pas forcément à une solution du CSP. On ne cherche donc pas à avoir $\bigcup S = \mathrm{CL}\!\downarrow\!(\mathbb{D}, R)$.

Il faut insister sur le fait que pour le diagnostic déclaratif, comme on le verra plus tard, l'utilisateur n'a pas à connaître exactement les solutions attendues, ni l'environnement attendu. Il n'a besoin de connaître que quelques éléments de $\bigcup S$ et quelques éléments de $\mathbb{D} \setminus \bigcup S$ pour répondre aux questions d'un outil de diagnostic déclaratif.

On considère fixé l'ensemble de solutions attendues S et donc l'environnement attendu $\bigcup S$.

9.2 Diagnostic de réponse manquante

On se penche sur l'aspect réponse manquante, pour lequel on verra que les explications peuvent apporter une aide pour la recherche d'erreur. Dans un premier temps, les notions de correction du CSP et du programme par rapport à la sémantique attendue sont définies. Les symptômes de réponses manquantes caractérisent les différences entre cette attente et ce qui est obtenu par le calcul. Il est ensuite montré comment le diagnostic déclaratif utilise les explications pour aider le programmeur à retrouver l'erreur ayant causé un symptôme.

9.2.1 Correction

Les notions de sémantique attendue étant définies, on peut alors considérer la correction d'un CSP. Cette correction repose sur la comparaison entre les solutions attendues et les solutions du CSP. Si toutes les solutions attendues sont solutions du CSP alors on considère que celui-ci formalise correctement le problème réel.

Définition 9.3 *Le CSP est dit* correct *par rapport aux solutions attendues S si $S \subseteq \mathrm{Sol}$.*

Si au moins une solution attendue n'est pas solution du CSP alors le CSP n'est pas correct.

Le diagnostic déclaratif se faisant à partir du programme, il est nécessaire que le programme réponde aux mêmes attentes que le CSP vis-à-vis de la sémantique attendue, c'est-à-dire que les éléments appartenant aux solutions attendues ne doivent pas être retirés lors du calcul. Ils doivent donc appartenir à la clôture calculée par le programme. La clôture étant une approximation des solutions, on utilise l'expression *approximativement correct* pour un programme.

Définition 9.4 *R est dit* approximativement correct *par rapport à $\bigcup S$ si $\bigcup S \subseteq \mathrm{CL}\downarrow(\mathbb{D}, R)$.*

Comme on va le voir, la notion de correction est plus forte que la notion de correction approximative.

Rappelons que le programme est préservant pour le CSP. Les deux lemmes suivants mettent en relation la R-consistance avec d'une part le CSP et d'autre part le programme.

Lemme 9.1 *Si le CSP est correct par rapport à S alors $\bigcup S$ est R-consistant.*

Preuve. Par le lemme 4.7. □

En effet chaque solution attendue doit être une solution du CSP (i.e. $S \subseteq \mathrm{Sol}$) et donc si les opérateurs de consistance locale du programme sont appliqués à ces solutions, ils ne doivent pas en retirer d'éléments.

Lemme 9.2 *Si $\bigcup S$ est R-consistant alors R est approximativement correct par rapport à $\bigcup S$.*

Preuve. par la définition 9.4 et le lemme 4.1. □

Si les solutions attendues ne sont pas retirées par le programme alors elles appartiennent à la clôture du domaine par ce programme. Autrement dit le programme est approximativement correct. Finalement, de ces deux lemmes se déduit aisément la relation naturelle entre correction et correction approximative.

Théorème 9.3 *Si le CSP est correct par rapport à S alors R est approximativement correct par rapport à $\bigcup S$.*

Preuve. par les lemmes 9.1 et 9.2. □

La réciproque est évidemment fausse.

Exemple 14 *Soit un CSP ayant pour solutions les tuples $\{(X,0),(Y,1)\}$ et $\{(X,1),(Y,0)\}$. On considère qu'un programme, préservant pour ce CSP, calcule la clôture $\{(X,0),(X,1),(Y,0),(Y,1)\}$. Si les solutions attendues pour ce CSP sont $\{(X,0),(Y,0)\}$ et $\{(X,1),(Y,1)\}$ alors le CSP est incorrect, mais le programme est approximativement correct.*

Le théorème précédent peut être intéressant pour faire de la validation de programme, mais notre propos étant la mise au point, sa contraposée nous sera plus utile.

Corollaire 9.4 *Si R n'est pas approximativement correct par rapport à $\bigcup S$ alors le CSP n'est pas correct par rapport à S.*

Si le programme retire des éléments attendus alors les solutions du CSP ne correspondront pas à l'attente de l'utilisateur. Plus précisément, il manquera des solutions attendues.

9.2.2 Symptôme de réponse manquante et erreur

A partir de la notion d'environnement attendu, on peut définir la notion de symptôme de réponse manquante. Un symptôme de réponse manquante souligne la différence entre ce qui est attendu et ce qui a été calculé.

Définition 9.5 *$h \in \mathbb{D}$ est un* symptôme de réponse manquante *par rapport à un environnement attendu $\bigcup S$ si $h \in \bigcup S \setminus \mathrm{CL}\!\downarrow(\mathbb{D}, R)$.*

Un symptôme de réponse manquante est donc un élément du domaine $\mathbb{D}$ retiré par le calcul alors qu'il était attendu dans une solution du CSP.

On peut remarquer que $\bigcup S \subseteq \mathrm{CL}\!\downarrow(\mathbb{D}, R)$ est équivalent à : il n'y a pas de symptôme par rapport à $\bigcup S$. Les notions de correction peuvent alors être reformulées comme suit :

- un programme est approximativement correct par rapport à $\bigcup S$ s'il n'existe pas de symptôme de réponse manquante par rapport à $\bigcup S$.
- Donc, s'il existe un symptôme de réponse manquante par rapport à $\bigcup S$ alors le CSP n'est pas correct par rapport à S.

Autrement dit, l'apparition d'un symptôme de réponse manquante révèle que quelque-chose ne va pas dans le programme et donc dans le CSP. L'apparition de ce symptôme de réponse manquante est causé par une erreur dans le programme, plus précisément par un opérateur de consistance locale qui a provoqué le retrait de cette valeur (le symptôme de réponse manquante). Il est important de noter que cet opérateur n'est pas obligatoirement celui dont l'application a retiré directement cette valeur. Cependant cet opérateur a retiré une valeur qui n'aurait pas dû l'être, ce qui a provoqué le retrait du symptôme par le jeu de la propagation.

Si un environnement d n'est pas R-consistant alors il existe un opérateur de consistance locale $r \in R$ tel que d n'est pas r-consistant. Cet opérateur est dit erroné si cet environnement est un environnement attendu.

Définition 9.6 *Un opérateur de consistance locale $r \in R$ est un* opérateur erroné *par rapport à $\bigcup S$ si $\bigcup S \not\subseteq r(\bigcup S)$.*

Bien sûr, $\bigcup S$ est R-consistant est équivalent à il n'y a pas d'opérateur erroné par rapport à $\bigcup S$ dans R.

Pour les notions de consistance où un opérateur est associé à une contrainte, trouver un opérateur erroné revient donc à trouver la contrainte fautive. Rappelons qu'en pratique, ce sont essentiellement de telles consistances qui sont utilisées (hyper arc-consistance, consistance de borne). Cette contrainte (on dira *contrainte erronée*) est une mauvaise formalisation du problème car elle est responsable d'un retrait non souhaité. Pour d'autres consistances, un opérateur peut correspondre à plusieurs contraintes. Dans le cas de la consistance de chemin par exemple, un opérateur est associé à trois contraintes.

Le théorème suivant indique la problématique du diagnostic : trouver une erreur à partir d'un symptôme de réponse manquante.

Théorème 9.5 *S'il existe un symptôme de réponse manquante par rapport à $\bigcup S$ alors il existe un opérateur erroné par rapport à $\bigcup S$.*

Il faut noter que la réciproque est fausse. Un programme peut être erroné mais sans symptôme de réponse manquante.

9.2.3 Diagnostic déclaratif de réponse manquante

Le diagnostic déclaratif (Shapiro, 1982) a été utilisé avec succès pour différents paradigmes de programmation (programmation logique (Shapiro, 1982), programmation fonctionnelle (Nilsson & Fritzson, 1994), programmation logique avec contraintes (Tessier, 1997), programmation logico-fonctionnelle (Lloyd, 1995), programmation impérative (Fritzson *et al.*, 1992)). Déclaratif signifie que l'utilisateur n'a pas à connaître la conduite calculatoire du système de programmation.

Précédemment il a été montré que s'il existe un symptôme de réponse manquante alors il existe un opérateur erroné. De plus, pour chaque symptôme de réponse manquante, un arbre explicatif peut être obtenu à partir du calcul. Cette section décrit comment localiser un opérateur erroné à partir d'un symptôme de réponse manquante et de son arbre explicatif en adaptant le diagnostic déclaratif à la programmation par contraintes.

Il faut rappeler qu'il existe un arbre explicatif pour chaque élément retiré du domaine (cf. corollaire 6.7). Les symptômes de réponse manquante étant des éléments retirés par le solveur, il existe donc un arbre explicatif pour chacun de ces symptômes. Celui-ci est construit pendant la réduction de domaine (cf. section 6.4). Le diagnostic déclaratif va s'appuyer sur ces arbres explicatifs.

Les arbres explicatifs étant construits à partir des règles de déduction, il est nécessaire d'étendre la notion d'erreur aux règles définissant un opérateur.

Définition 9.7 *Une règle $h \leftarrow B \in \mathcal{R}_r$ est une* règle erronée *par rapport à $\bigcup S$ si $B \cap \bigcup S = \emptyset$ et $h \in \bigcup S$.*

Une règle est donc erronée si elle retire un élément attendu à partir d'éléments non attendus. En effet elle participe à la définition d'un opérateur r tel que $\bigcup S \not\subseteq r(\bigcup S)$.

Il est facile de vérifier que : r est un opérateur erroné par rapport à $\bigcup S$ si et seulement si il existe une règle erronée $h \leftarrow B \in \mathcal{R}_r$ par rapport à $\bigcup S$. Par conséquent, le théorème 9.5 peut être étendu.

Lemme 9.6 *S'il existe un symptôme de réponse manquante par rapport à $\bigcup S$ alors il existe une règle erronée par rapport à $\bigcup S$.*

Le diagnostic consiste alors à trouver la règle erronée utilisée dans l'arbre explicatif enraciné par le symptôme.

On dit qu'un nœud d'arbre explicatif est un *symptôme* par rapport à $\bigcup S$ si son étiquette est un symptôme de réponse manquante par rapport à $\bigcup S$. Etant donné que pour chaque symptôme de réponse manquante h, il existe un arbre explicatif dont la racine est étiquetée par h, il est possible de parler de minimalité selon la relation père/fils dans un arbre explicatif.

Définition 9.8 *Soit t un arbre explicatif enraciné par un symptôme. Un symptôme de t est* minimal *par rapport à $\bigcup S$ si aucun de ses fils n'est un symptôme par rapport à $\bigcup S$.*

Il faut remarquer que si h est un symptôme minimal par rapport à $\bigcup S$ alors $h \in \bigcup S$ et l'ensemble de ses fils B est tel que $B \subseteq \overline{d}$. En d'autres termes, $h \leftarrow B$ est une règle erronée par rapport à $\bigcup S$.

Théorème 9.7 *Dans un arbre explicatif enraciné par un symptôme de réponse manquante par rapport à $\bigcup S$, il existe au moins un symptôme minimal par rapport à $\bigcup S$ et la règle qui lie le symptôme minimal à ses fils est une règle erronée.*

Preuve. Puisque les arbres explicatifs sont des arbres finis, la relation père/fils est bien-fondée. □

Pour résumer, un symptôme minimal est associé à une règle erronée, celle-ci participe à la définition d'un opérateur erroné. De plus, un opérateur est lui-même associé à, une contrainte (par exemple dans le cas de l'hyper arc-consistance), ou un ensemble de contraintes (par exemple dans le cas de la consistance de chemin). Par conséquent, la recherche de contraintes erronées dans un CSP peut se faire par la recherche d'un symptôme minimal dans un arbre explicatif enraciné par un symptôme de réponse manquante.

9.2.4 Algorithme de diagnostic

Un algorithme de diagnostic déclaratif d'erreur pour un symptôme de réponse manquante (x, v) peut être décrit. Soit un arbre explicatif calculé pour (x, v). Le but est de trouver un symptôme minimal dans cet arbre en interrogeant un oracle avec des questions du type : "(y, v') est-il attendu ?", ou autrement dit "(y, v') est-il un symptôme de réponse manquante ?". Il ne faut pas perdre de vue que tous les éléments apparaîssant dans l'arbre explicatif ont été retirés par le calcul. Parmi eux certains n'auraient pas dû l'être. Différents algorithmes de diagnostic peuvent être utilisés pour découvrir un symptôme minimal.

On propose tout d'abord un algorithme générique (l'algorithme 1).

Cet algorithme teste les nœuds de l'arbre jusqu'à y trouver un symptôme minimal ou jusqu'à les avoir tous tester. L'arbre explicatif peut donc ne pas contenir de symptôme (dans ce cas aucune erreur n'est trouvée). Si un symptôme minimal est découvert, l'algorithme retourne la règle erronée.

Différentes stratégies peuvent être choisies pour parcourir l'arbre. Par exemple, l'algorithme 2, qui utilise une stratégie top-down de parcours de l'arbre explicatif doit permettre de remonter de la racine à un symptôme minimal en suivant une branche de symptômes.

Il faut noter que contrairement à l'algorithme 1, celui-ci considère un arbre explicatif enraciné par un symptôme, h sera donc toujours un symptôme. Dans cet algorithme, la réponse à la question $\exists h' \in B$ tel que $h' \in \bigcup S$ doit être donnée par un oracle, en pratique il peut s'agir d'un utilisateur. Cette question revient à

Algorithme 1 Diagnostic déclaratif générique

données: arbre explicatif, sémantique attendue $\bigcup S$
  erreur := $\emptyset$
  $E := \{n \mid n$ est un nœud de l'arbre $\}$
  tant que erreur $= \emptyset$ et $E \neq \emptyset$ **faire**
    choisir n dans E
    $E := E \setminus n$
    B := $\{n' \mid n'$ est un fils de $n\}$
    si label$(n) \in \bigcup S$ et $\forall n' \in B, \text{label}(n') \notin \bigcup S$ **alors**
      erreur := label$(n) \leftarrow \{\text{label}(n') \mid n' \in B\}$
    fin si
  fin tant que
  retourner erreur

Algorithme 2 Diagnostic déclaratif top-down

données: arbre explicatif t pour un symptôme, sémantique attendue $\bigcup S$
  $n := \text{root}(t)$
  erreur := $\emptyset$
  tant que erreur $= \emptyset$ **faire**
    B := $\{n' \mid n'$ est un fils de $n\}$
    si $\forall n' \in B, \text{label}(n') \notin \bigcup S$ **alors**
      erreur := label$(n) \leftarrow \{\text{label}(n') \mid n' \in B\}$
    sinon
      $n := n'$
    fin si
  fin tant que
  retourner erreur

s'interroger sur les fils d'un nœud pour déterminer s'ils sont ou non symptômes. Différentes stratégies peuvent être choisies : commencer par le fils ayant la plus petite explication, par la plus grande valeur d'une variable, par la plus petite...

Il est possible de considérer d'autres algorithmes pour parcourir l'arbre explicatif. En pratique, on cherchera évidemment à minimiser le nombre de questions. On peut choisir par exemple la stratégie "diviser pour régner" qui consiste à choisir un nœud divisant l'espace de recherche en deux (c'est-à-dire tel que la moitié des nœuds soit dans le sous-arbre enraciné par ce nœud). Dans ce cas, si n est le nombre de nœuds de l'arbre alors, en pratique, le nombre de questions est de l'ordre de $log(n)$, c'est-à-dire peu par rapport à la taille de l'arbre et donc très peu comparé à la taille de l'itération.

Le diagnostic déclaratif possède l'avantage de ne considérer que l'information pertinente. Pour un symptôme de réponse manquante donné, seuls les opérateurs qui ont causé son retrait ont été utilisés pour construire l'arbre explicatif. De plus, certaines branches de l'arbre pouvant être rapidement élaguées, la quantité d'information que l'utilisateur aura à traiter sera très faible par rapport à une trace classique. On peut voir le diagnostic déclaratif comme une utilisation "intelligente" de la trace, c'est le processus que suivrait un programmeur pour déboguer son programme.

Rappelons qu'il n'est pas nécessaire pour l'utilisateur de connaître exactement l'ensemble de solutions, ni une approximation précise de celles-ci. La sémantique attendue est théoriquement considérée comme une partition du domaine $\mathbb{D}$: les éléments qui sont attendus et ceux qui ne le sont pas. En fait, pour le diagnostic de réponse manquante, l'oracle a seulement à répondre à quelques questions (il doit dévoiler pas à pas une partie de la sémantique attendue). La sémantique attendue peut alors être considérée comme trois ensembles : un ensemble d'éléments attendus (i.e. les nœuds symptômes), un ensemble d'éléments non attendus (i.e. les nœuds non symptômes) et un ensemble pour lequel l'utilisateur n'a pas à savoir (i.e. les éléments n'apparaissant pas dans l'arbre ainsi que ceux de l'arbre pour lesquels l'utilisateur ne sera pas questionné). Il est seulement nécessaire pour l'utilisateur de répondre aux seules questions posées.

C'est évidemment sur ce point que la difficulté réside. Il peut en effet être difficile pour l'utilisateur de répondre aux questions posées, cependant il est impossible de trouver une erreur sans connaissance du problème. Il est possible de considérer que l'utilisateur ne peut répondre à certaines questions, mais dans ce cas, il n'y a pas de garantie de trouver une erreur (Tessier & Ferrand, 2000). Sans un tel outil, l'utilisateur se trouvera face à une itération, c'est-à-dire une longue liste d'événements. Dans ces conditions, il semble plus facile de trouver une erreur dans le code du programme que dans cette trace. Même si l'utilisateur n'est pas capable de répondre aux questions, il faut rappeler qu'il dispose d'un ensemble explicatif pour le symptôme qui réduit déjà considérablement l'espace de recherche.

On peut aussi considérer qu'avant de débuter une session de diagnostic, l'utilisateur est capable de donner plusieurs symptômes de réponse manquante. Cette information peut être profitable d'une part pour répondre de façon automatique aux questions de l'outil de diagnostic, d'autre part si plusieurs symptômes peuvent être donnés par l'utilisateur, le diagnostic peut alors se faire sur différents arbres explicatifs.

Enfin, si on suppose qu'il n'existe qu'une erreur dans le programme, alors les ensembles explicatifs de chaque symptôme de réponse manquante contiennent l'opérateur erroné. En calculant l'intersection de ces ensembles explicatifs, on obtient donc un ensemble d'opérateurs contenant l'opérateur erroné. Dans le meilleur cas, cet ensemble peut-être réduit à un seul opérateur, l'opérateur erroné. Il peut en tout cas être possible d'isoler des opérateurs sans faire de diagnostic par la simple donnée de quelques symptômes.

9.2.5 Application au problème de la conférence

On poursuit le problème de la conférence décrit dans les chapitres 4 et 6. Cependant, puisqu'on veut faire du diagnostic, on considère que le programmeur a introduit une erreur dans son CSP. Le CSP considéré est le suivant :

```
        conf(PA,PM,AM,AP,MP,MA):-
(1)         fd_domain([PA,PM,AM,AP,MP,MA],1,6),
(2)         AM #\=# 6,
(3)         AM #\=# 5,
(4)         MA #\=# 6,
(5)         MA #\=# 5,
(6)         AP #\=# 6,
(7)         AP #\=# 5,
(8)         PA #\=# 6,
(9)         PA #\=# 5,
(10)        PM #<# MA,
(11)        MP #<# PM,
(12)        AM #<# MA,
(13)        AM #<# MP,
(14)        PA #<# AP,
(15)        PA #<# AM,
(16)        AM #\=# AP,
(17)        MA #\=# PA,
(18)        MP #\=# AP,
(19)        AM #\=# PM,
(20)        AP #\=# MA,
(21)        PM #\=# AP,
(22)        PA #\=# MP,
(23)        fd_labeling([MP,PM,PA,AM,AP,MA]).
```

Le programmeur connaît bien entendu le problème. Sachant qu'Alain et Michel veulent connaître les travaux de Pierre avant de faire leur exposé, il s'attend à obtenir une solution avec un exposé de Pierre à Alain dès la première demi-journée. Or l'éxécution du programme ne lui donne pas satisfaction (il n'obtient même aucune solution). Le programmeur s'interroge sur la correction de son programme et décide donc de lancer une session de diagnostic pour soit retrouver une possible erreur, soit (s'il n'a pas fait d'erreur) comprendre pourquoi il n'obtient pas de solution avec un exposé de Pierre à Alain dès la première demi-journée. Autrement dit il s'interroge sur le retrait de $(PA, 1)$.

On considère donc l'arbre explicatif[1] du retrait de $(PA, 1)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(MA,5)}{\scriptstyle MA\neq 5}\quad \dfrac{}{(MA,6)}{\scriptstyle MA\neq 6}}{(PM,4)}{\scriptstyle PM<MA}\quad \dfrac{\dfrac{}{(MA,6)}{\scriptstyle MA\neq 6}}{(PM,5)}{\scriptstyle PM<MA}\quad \dfrac{}{(PM,6)}{\scriptstyle PM<MA}}{(MP,3)}{\scriptstyle MP<PM}\quad \ldots\quad \dfrac{}{(MP,6)}{\scriptstyle MP<PM}}{(AM,2)}{\scriptstyle AM<MP}\quad \ldots}{(PA,1)}{\scriptstyle PA<AM}$$

Suivant l'algorithme 2, les questions suivantes vont être posées au programmeur

- "Existe t-il un symptôme parmi $(AM, 2), \ldots, (AM, 6)$?" Pour répondre à cette question, le programmeur exploite ses connaissances du problème concernant l'exposé d'Alain à Michel : cet exposé doit avoir lieu avant les exposés de Michel et après l'exposé de Pierre à Alain. Le programmeur étant parti du fait que ce dernier pouvait avoir lieu dès la première demi-journée $(PA, 1)$, l'exposé d'Alain à Michel peut donc avoir lieu juste après. $(AM, 2)$ est donc un symptôme.
- Le diagnostic se poursuit alors dans l'arbre explicatif du retrait de $(AM, 2)$. La question suivante est : "Existe t-il un symptôme parmi $(MP, 3), \ldots, (MP, 6)$?" Ce retrait est causé par les retraits des valeurs 3, 4, 5 et 6 pour MP. Les seules contraintes connues sur l'exposé de Michel à Pierre sont les exigences de Michel qui veut connaître les travaux de Pierre et Alain avant d'exposer les siens. Cet exposé devrait donc avoir lieu vers la fin. Le programmeur confirme donc que $(MP, 6)$ est un symptôme.
- La règle ayant supprimée $(MP, 6)$ n'ayant pas de corps, cet élément est donc un symptôme minimal. La règle $(MP, 6) \leftarrow \emptyset$ est une règle erronnée. Cette règle définit l'opérateur associé à la contrainte $MP < PM$ qui est donc la contrainte responsable du symptôme $(PA, 1)$. Cette contrainte a en effet remplacée la contrainte $PM < MP$.

[1] Celui-ci étant large, mais peu profond, les parties non explorées sont remplacées par des points de suspension.

Pour l'étage de $(MP, 3)$, il manque les arbres explicatifs de $(MP, 4)$, $(MP, 5)$ et $(MP, 6)$. Il faut noter que ces arbres explicatifs sont plus petits que celui de $(MP, 3)$ et qu'un symptôme minimal existe dans chacun de ses sous-arbres.

Pour l'étage de $(AM, 2)$, il manque les arbres explicatifs de $(AM, 3)$, $(AM, 4)$, $(AM, 5)$ et $(AM, 6)$.

Il a donc suffi au programmeur de répondre à deux questions pour trouver l'erreur. Il faut noter que le choix de $(PA, 1)$ est un des plus mauvais que pouvait faire le programmeur au départ. En effet à l'opposé, le choix de $(MP, 6)$ comme symptôme l'aurait conduit à trouver l'erreur sans la moindre question puisque l'arbre explicatif de $(MP, 6)$ est une feuille.

En supposant maintenant qu'à la seconde étape, le programmeur ait seulement remarqué que $(MP, 3)$ était un symptôme. Il lui faut alors continuer le diagnostic dans un autre sous-arbre. La question suivante serait alors : "Existe t-il un symptôme parmi $(PM, 4), (PM, 5), (PM, 6)$?". L'utilisateur hésitant se renseigne donc d'abord sur les raisons de ces retraits. Il s'avère qu'ils sont tous causés par un opérateur associé à la contrainte $PM < MA$. Autrement dit par le fait que Michel exposera à Alain après avoir vu l'exposé de Pierre. Sachant qu'Alain n'est pas présent le dernier jour, il faut donc que l'exposé de Pierre à Michel ait lieu au plus tard la troisième demi-journée. Les trois retraits paraissent donc justifiés. $(MP, 3)$ est donc un symptôme minimal et la contrainte fautive $MP < PM$ peut ici aussi être localisée. Si l'utilisateur est capable de donner plusieurs symptômes avant de débuter la session de mise au point, il peut être possible de choisir l'arbre explicatif à utiliser pour le diagnostic. Par exemple si l'utilisateur donne les symptômes $(MP, 3)$ et $(PA, 1)$, l'arbre explicatif de $(PA, 1)$ contenant l'arbre explicatif de $(MP, 3)$, le diagnostic peut être effectué sur l'arbre explicatif de $(MP, 3)$ et l'erreur localisée plus rapidement.

9.2.6 Enumération

Dans cette section, un algorithme de diagnostic de réponse manquante a été proposé. Il faut noter que la méthode utilisée ne prend pas en compte l'énumération. On montre ici que cette méthode peut être facilement utilisée avec l'énumération.

Considérons le cas où l'utilisateur attend une solution avec un élément qui appartient à la clôture. Cet élément n'est donc pas un symptôme de réponse manquante d'après la définition 9.5. Supposons également que cet élément n'appartienne à aucune solution du CSP. Lors de l'énumération, cet élément sera donc retiré dans chacune des branches de l'arbre de recherche (soit par un opérateur d'énumération, soit par un opérateur de consistance locale).

Dans tous les cas (i.e. pour chacune des branches), il existe un arbre explicatif pour le retrait de l'élément. Puisque l'élément était attendu dans une solution et que le labeling est complet, au moins un de ces retraits est injustifié, ou plutôt est la conséquence de l'application d'un opérateur erroné. Cet arbre explicatif contient donc un symptôme minimal causé par l'application d'une règle erronée. Le diagnostic de réponse manquante peut donc se faire en adaptant la méthode précédemment décrite.

Deux différences sont à noter. D'une part, étant donné qu'il existe plusieurs arbres explicatifs pour le retrait de l'élément (et que tous ne contiennent pas forcément l'opérateur erroné), l'utilisateur peut choisir l'arbre à utiliser, c'est-à-dire qu'il peut dire quelle est la branche dans laquelle ce retrait n'aurait pas dû avoir lieu. Ne pas pouvoir répondre à cette question n'est pas dramatique, l'utilisateur peut effectuer le diagnostic sur plusieurs arbres explicatif sans erreur. Cela lui prendra seulement plus de temps.

L'autre différence avec une session de diagnostic classique repose sur le fait que la sémantique attendue peut ne plus être la même. En effet puisque des opérateurs d'énumération ont été appliqués, une solution attendue pour le CSP peut ne pas être une solution attendue dans la branche où a été retiré le symptôme. Il faut donc que l'utilisateur en tienne compte pour répondre aux questions de l'outil de diagnostic. En suivant le lemme 4.12, l'utilisateur doit donc considérer un calcul non plus à partir du domaine $\mathbb{D}$, mais à partir de $\mathrm{CL}\downarrow(\mathbb{D}, N)$ où N est l'ensemble des opérateurs d'énumération utilisés dans la branche de l'arbre de recherche où il se trouve.

9.3 Explication d'échec

Pour conclure ce chapitre, la notion d'explication d'échec est définie. Celle-ci peut être utile pour la mise au point, les problèmes sur-contraints et les méthodes de recherche dites rétrospectives.

Une explication d'échec peut être vue comme une réunion d'ensembles explicatifs pour le domaine entier d'une variable. En effet une explication d'échec est un ensemble d'opérateurs responsable d'un échec, c'est-à-dire de l'obtention d'un environnement vide pour une variable. Toute itération contenant cet ensemble d'opérateurs conduira immanquablement à un échec. On ne considère que les calculs à partir de $\mathbb{D}$, l'explication d'échec pour un calcul à partir de $d \subset \mathbb{D}$ n'ayant pas d'intérêt ici.

Définition 9.9 *On appelle* explication d'échec *tout ensemble d'opérateurs E tel que $\exists x \in V, \mathrm{CL}\downarrow(\mathbb{D}, E)|_{\{x\}} = \emptyset$.*

Il faut noter que le type d'opérateurs n'a pas été précisé, il peut s'agir d'opérateurs de consistance locale et d'énumération. Cette explication d'échec peut donc être construite à partir des ensembles explicatifs des retraits de chaque élément du domaine d'une variable. Pour chaque $(x, v) \in \overline{\mathrm{CL}\downarrow(\mathbb{D}, R \cup N)}$ avec R le programme et N un ensemble d'opérateurs d'énumération, un ensemble explicatif $\mathrm{expl}(x, v)$ est choisi.

Théorème 9.8 *Si $\mathrm{CL}\downarrow(\mathbb{D}, R \cup N)|_{\{x\}} = \emptyset$ alors $\bigcup_{v \in D_x} \mathrm{expl}(x, v)$ est une explication d'échec.*

Preuve. évident. □

Il existe une autre façon de construire une explication d'échec en utilisant cette fois un seul ensemble explicatif. On note $r_{x=v}$ l'opérateur d'énumération associé à la contrainte $x = v$.

Théorème 9.9 *Si* $\mathrm{expl}(x, v)$ *est un ensemble explicatif pour* (x, v) *alors* $\mathrm{expl}(x, v) \cup \{r_{x=v}\}$ *est une explication d'échec.*

Preuve. évident. □

On retrouve ici la définition d'explication de contradiction (en passant des opérateurs aux contraintes) et donc celle de nogood.

Face à un problème sur-contraint (c'est-à-dire un problème sans solution), l'utilisateur se retrouve souvent désarmé. Il lui est souvent nécessaire d'obtenir une solution à son problème quitte à relaxer certaines contraintes. Une explication d'échec lui fournit l'ensemble des opérateurs (et donc des contraintes pour les consistances classiquement utilisées) responsable de l'échec. Autrement dit relaxer une contrainte qui n'appartient pas à cet ensemble ne lui fournira pas de solution, il obtiendra toujours un échec. Ce qui ne veut pas dire que relaxer une des contraintes de cet ensemble lui fournira obligatoirement un succès.

Lors d'une recherche systématique, quand un échec est obtenu dans une branche de l'arbre de recherche, l'algorithme de backtrack standard revient sur la dernière instanciation effectuée. Les méthodes dites rétrospectives (cf. section 5.2) utilisent les explications d'échec pour choisir un meilleur point de retour. En effet si la dernière instanciation effectuée n'appartient pas à l'explication d'échec, modifier cette instanciation n'empêchera pas un nouvel échec de se produire. Il faut donc choisir une contrainte associée à un opérateur d'énumération de cet ensemble.

Enfin, pour la mise au point une explication d'échec fournit les opérateurs responsables de l'absence de solutions. Si l'utilisateur sait qu'il devrait obtenir une solution mais n'a aucune idée des éléments qui doivent y appartenir (i.e. il ne peut pas donner de symptôme de réponse manquante), l'information la plus précise que peut lui apporter un outil d'aide à la mise au point est cet ensemble d'opérateurs. Pour que cette information soit compréhensible, il est nécessaire de fournir les contraintes associées plutôt que les opérateurs. De même la vue hiérarchisée proposée dans les explications *user-friendly* (Ouis *et al.*, 2002) peut se révéler très utile pour rendre cette information plus claire.

9.4 Conclusion

Dans ce chapitre, la mise au point de réponse manquante a été abordée d'un point de vue déclaratif. Cette approche semble la plus naturelle étant donné le caractère déclaratif du langage lui même. En effet, il semble incohérent de vouloir faire déboguer un calcul à l'utilisateur alors que celui-ci ne le connaît pas. L'approche déclarative proposée ici permet de se dispenser de cette connaissance du calcul.

Les explications proposent une base solide pour effectuer de la mise au point, non seulement pour expliquer un échec, mais aussi pour effectuer du diagnostic déclaratif de réponse manquante. Un symptôme de réponse manquante n'est pas un tuple dans son ensemble mais simplement un élément de ce tuple (la valeur d'une des variables). Celui-ci est d'un arbre explicatif construit pendant le calcul dans lequel se trouve l'erreur à identifier. En détectant de nouveaux symptômes dans cet arbre, l'utilisateur s'approche peu à peu de l'erreur jusqu'à la localiser. Dans le cas où l'utilisateur ne sait plus répondre, l'ensemble explicatif associé au dernier symptôme repéré peut quand même lui donner un sous-ensemble des opérateurs du programme parmi lesquels se trouve l'opérateur erroné. Pour un outil de mise au point convivial, il faudrait bien entendu fournir à l'utilisateur les contraintes plutôt que les opérateurs.

L'introduction de l'énumération dans ce cadre ne pose pas de difficultés conceptuelles. Si un symptôme de réponse manquante apparaît seulement après une phase d'énumération, il existe plusieur arbres explicatifs pour le retrait de celui-ci (un par branche), l'utilisateur doit alors choisir la branche pour laquelle ce retrait est inattendu. Il doit ensuite simplement garder à l'esprit qu'il se trouve dans une branche de l'arbre de recherche et que par conséquent certaines valeurs de variables ne sont plus autorisées (la sémantique attendue s'en retrouve modifiée). Cela revient à considérer un calcul à partir d'un environnement préalablement réduit (par les opérateurs d'énumération).

Le diagnostic déclaratif de réponse manquante et les explications d'échec devraient pouvoir s'intégrer dans un système complet de mise au point de programmes avec contraintes. Un tel système devrait aussi comprendre, entre autres, des outils pour le débogage de performance et des outils d'aide à la mise au point de réponse fausse.

Le diagnostic de réponse fausse est une problématique intéressante qui requiert une approche différente de celle proposée ici pour les réponses manquantes. D'une part les notions de correction sont totalement différentes. En effet, à cause de l'incomplétude des solveurs, on ne peut pas juger qu'un programme est incorrect si sa clôture contient des éléments non attendus dans les solutions du CSP (ces éléments pourraient en effet être retirés lors de l'énumération). D'autre part, les arbres explicatifs sont des preuves pour les éléments retirés alors que l'on chercherait ici à savoir pourqui des éléments n'ont pas été retirés. La mise au point de réponse fausse nécessite donc une approche différente, ce qui en fait un complément à ce travail intéressant à étudier.

Chapitre 10

Conclusion

Trois contributions au domaine de la programmation par contraintes se dégagent pincipalement de ce travail.

- une reformulation de la réduction de domaine dans un cadre ensembliste ;
- une formalisation des explications sous forme arborescente ;
- une méthode de diagnostic déclaratif de réponse manquante.

La réduction de domaine a été décrite dans un cadre ensembliste issu de la théorie du point fixe. L'utilisation d'opérateurs de consistance locale exprime exactement les calculs du solveurs. Les valeurs inconsistantes sont calculées localement. La réduction se fait ensuite au niveau global en intersectant le résultat de l'application de ces opérateurs avec l'environnement. L'abstraction des stratégies particulières aux implantations obtenue par les itérations chaotiques donne un formalisme général applicable à tout solveur effectuant de la réduction de domaine par des notions de consistance. Le lien entre les opérateurs et les contraintes selon la consistance choisie a été clairement établi.

L'incomplétude des solveurs a pour conséquence l'absence de preuve pour les valeurs restant dans le domaine après la réduction. En revanche, la vision duale proposée est tout à fait naturelle pour prouver les retraits de valeurs. Les arbres explicatifs relatent exactement le raisonnement suivi pour retirer une valeur. Il a été montré comment certains pouvaient être construits de façon incrémentale pendant le calcul. Il faut souligner que la définition proposée est suffisamment générale pour retrouver les notions d'explication existantes et pour prouver la correction des algorithmes de relaxation de contrainte. De plus, l'information contenue dans les arbres explicatifs se révèle très intéressante pour la mise au point de réponse manquante.

Une formalisation des notions d'erreur et de symptôme de réponse manquante a été proposée. Le diagnostic déclaratif adapté à la programmation par contraintes s'effectue sur les arbres explicatifs dont la racine est un symptôme de réponse manquante. Le diagnostic déclaratif revient alors à rechercher une erreur en découvrant peu à peu d'autres symptômes dans l'arbre. Cette méthode semble adéquate car elle restreint l'information à traiter (par rapport à une trace classique) et permet à l'utilisateur de ne pas avoir à se plonger dans les calculs. La connaissance demandée à l'utilisateur concerne les résultats du calcul, elle se situe donc à un niveau déclaratif qu'il est plus apte à appréhender.

Il est légitime de s'interroger sur la difficulté pour l'utilisateur à découvrir des symptômes dans les résultats obtenus par le calcul. D'un côté, aucune session de diagnostic ne peut se faire sans un minimum de connaissance de la part de l'utilisateur, mais d'un autre côté, on ne peut pas lui demander de connaître exactement les éléments qui n'auraient pas dû être retirés. Dans l'exemple traité, découvrir des symptômes reste une tâche facile, mais qu'en serait-il d'un problème plus vaste ? Pour répondre à cette question, l'implantation d'un outil de diagnostic est nécessaire. C'est une suite naturelle à donner à ce travail pour le valider sur des problèmes réels.

Pour cela, il est nécessaire de disposer des arbres explicatifs. Or aucun solveur ne fournit un tel outil pour l'instant. La construction des explications éliminantes dans `PaLM` se fait incrémentalement de la même façon que nos arbres explicatifs à la différence que les explications éliminantes sont des ensembles. L'adaptation de l'algorithme utilisé afin de construire des arbres ne paraît pas trop difficile. Cependant, un outil de mise au point se doit d'être générique et surtout non intrusif. Il serait donc plus logique de considérer un système pouvant obtenir les arbres explicatifs après coup, c'est-à-dire sans modifier le solveur. Il existe déjà quelques travaux (Junker, 2001; Douence & Jussien, 2002) pour obtenir des explications après le calcul. Une perspective intéressante serait d'extraire des arbres explicatifs à partir de la trace générique proposée dans OADymPPaC, ce qui permettrait d'utiliser le diagnostic déclaratif quel que soit le solveur.

L'accès à un tel outil semble intéressant mais il faut garder à l'esprit que celui-ci devrait pouvoir s'intégrer dans un système de mise au point plus complet comprenant, entre autres, des outils de visualisation et d'aide à la mise au point de réponse fausse.

Références

Aczel Peter. 1977. An Introduction to Inductive Definitions. *Chap. C.7, pages 739–782 de* Barwise Jon (ed), *Handbook of Mathematical Logic*. Studies in Logic and the Foundations of Mathematics, vol. 90. North-Holland Publishing Company.

Aggoun Abderrahamane, Chan David, Dufresne Pierre, Falvey Eamon, Grant Hugh, Herold Alexander, Macartney Geoffrey, Meier Micha, Miller David, Mudambi Shyam, Novello Stefano, Perez Bruno, van Rossum Emmanuel, Schimpf Joachim, Shen Kish, Tsahageas Periklis Andreas & de Villeneuve Dominique Henry. 2001. *ECLiPSe user manual release 5.3.* ECRC, ICL-ITC, IC-Parc.

Aillaud Christophe & Deransart Pierre. 2000. *Analysis and Visualisation Tools for Constraint Programming.* Lecture Notes in Computer Science, vol. 1870. Springer-Verlag. Chap. 8. Towards a Language for CLP Choice-Tree Visualisation, pages 209–236.

Apt Krzysztof R. 1999. The Essence of Constraint Propagation. *Theoretical Computer Science*, **221**(1–2), 179–210.

Apt Krzysztof R. 2000. The Role of Commutativity in Constraint Propagation Algorithms. *ACM Transactions On Programming Languages And Systems*, **22**(6), 1002–1036.

Arnold André & Niwinski Damian. 2001. Rudiments of mu-calculus. Studies in Logic and the Foundations of Mathematics, vol. 146. North-Holland Publishing Company.

Bayardo Jr. Roberto J. & Miranker Daniel P. 1996. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. *Pages 298–304 de Proceedings of the 13th National Conference on Artificial Intelligence*, vol. 1. AAAI Press / The MIT Press.

Benhamou Frédéric. 1996. Heterogeneous Constraint Solving. *Pages 62–76 de* Hanus Michael & Rofríguez-Artalejo Mario (eds), *Proceedings of the 5th International Conference on Algebraic and Logic Programming, ALP 96.* Lecture Notes in Computer Science, vol. 1139. Springer-Verlag.

Benhamou Frédéric & Goualard Frédéric. 2000. *Debugging Constraint Programs by Store Inspection.* Lecture Notes in Computer Science, vol. 1870. Springer-Verlag. Chap. 11. Debugging constraint programs by store inspection, pages 273–297.

Berlandier Pierre & Neveu Bertrand. 1994. Arc-Consistency for Dynamic Constraint Problems : A RMS-Free Approach. *Dans* Schiex Thomas & Bessière Christian (eds), *Proceedings ECAI'94 Workshop on Constraint Satisfaction Issues raised by Practical Applications.*

Bessière Christian. 1991. Arc consistency in dynamic constraint satisfaction problems. *Pages 221–226 de Proceedings of the 9th National Conference on Artificial Intelligence, AAAI 91*, vol. 1. AAAI Press.

Bessière Christian & Cordier Marie-Odile. 1993. Arc-consistency and arc-consistency again. *Pages 108–113 de Proceedings of the 11th National Conference on Artificial Intelligence, AAAI 93.* AAAI Press.

Bessière Christian & Régin Jean-Charles. 1996. MAC and Combined Heuristics : Two Reasons to Forsake FC (and CBJ ?) on Hard Problems. *Pages 61–75 de* Freuder Eugene C. (ed), *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, CP 96.* Lecture Notes in Computer Science, vol. 1118. Springer-Verlag.

Bessière Christian, Meseguer Pedro, Freuder Eugene C. & Larrosa Javier. 2002. On forward checking for non-binary constraint satisfaction. *Artificial Intelligence*, **141**(1-2), 205–224.

Boizumault Patrice, Debruyne Romuald & Jussien Narendra. 2000. Maintaining Arc-Consistency within Dynamic Backtracking. *Pages 249–261 de Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming, CP 00.* Lecture Notes in Computer Science, no. 1894. Springer-Verlag.

Bouvier Pascal. 2000. *Analysis and Visualisation Tools for Constraint Programming.* Lecture Notes in Computer Science, vol. 1870. Springer-Verlag. Chap. 6. Visual Tools to Debug Prolog IV Programs, pages 177–190.

Bueno Francisco, Hermenegildo Manuel & Puebla Germán. 2000. *Analysis and Visualisation Tools for Constraint Programming.* Lecture Notes in Computer Science, vol. 1870. Springer-Verlag. Chap. 2. A generic preprocessor for program validation and debugging, pages 63–107.

Carro Manuel & Hermenegildo Manuel. 2000. *Analysis and Visualisation Tools for Constraint Programming.* Lecture Notes in Computer Science, vol. 1870. Springer-Verlag. Chap. 9. Tools for Search-Tree Visualisation : The APT Tool, pages 237–252.

Codognet Philippe & Diaz Daniel. 1996. Compiling Constraints in `clp(FD)`. *Journal of Logic Programming*, **27**(3), 185–226.

Codognet Philippe, Fages François & Sola Thierry. 1993. A Metalevel Compiler of CLP(FD) and its combination with intelligent backtracking. *Chap. 23, pages 437–456 de* Benhamou Frédéric & Colmerauer Alain (eds), *Constraint Logic Programming : Selected Research.* Logic Programming. MIT Press.

Cousin Xavier. 1993. *Application de la programmation logique avec contraintes au problème d'emploi du temps.* Ph.D. thesis, Université de Rennes I.

Cousot Patrick & Cousot Radhia. 1977. Automatic synthesis of optimal invariant assertions mathematical foundation. *Pages 1–12 de Symposium on Artificial Intelligence and Programming Languages.* ACM SIGPLAN Not., vol. 12(8).

de Kleer Johan. 1986. An Assumption-based TMS. *Artificial Intelligence*, **28**(2), 127–162.

Debruyne Romuald. 1996. Arc-consistency in Dynamic CSPs is no more prohibitive. *Pages 299–306 de* 8^{th} *Conference on Tools with Artificial Intelligence, TAI 96.*

Debruyne Romuald, Ferrand Gérard, Jussien Narendra, Lesaint Willy, Ouis Samir & Tessier Alexandre. 2003. Correctness of Constraint Retraction Algorithms. *Pages 172–176 de* Russell Ingrid & Haller Susan (eds), *FLAIRS'03 : Sixteenth international Florida Artificial Intelligence Research Society conference.* AAAI Press.

Dechter Rina & Dechter Avi. 1988. Belief maintenance in Dynamic Constraint Networks. *Pages 37–42 de Proceedings of the 7th National Conference on Artificial Intelligence, AAAI 88.* AAAI Press.

Dechter Rina & Meiri Itay. 1989. Experimental Evaluation of Preprocessing Techniques in Constraint Satisfaction Problems. *Pages 271–277 de* Sridharan N. S. (ed), *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, IJCAI 89.* Morgan Kaufmann.

Dechter Rina & Pearl Judea. 1988. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, **34**(1), 1–38.

Deransart Pierre, Hermenegildo Manuel & Małuszyński Jan (eds). 2000. *Analysis and Visualisation Tools for Constraint Programming.* Lecture Notes in Computer Science, vol. 1870. Springer-Verlag.

Deville Yves, Saraswat Vijay & Van Hentenryck Pascal. 1991. *Constraint Processing in* `cc(FD)`. Draft.

Diaz Daniel & Codognet Philippe. 2000. The GNU-Prolog System and its Implementation. *Pages 728–732 de ACM Symposium on Applied Computing*, vol. 2.

Dincbas Mehmet, Van Hentenryck Pascal, Simonis Helmut & Aggoun Abderrahmane. 1988. The Constraint Logic Programming Language CHIP. *Pages 249–264 de International Conference on Fifth Generation Computer Systems.*

Douence Rémi & Jussien Narendra. 2002. Non-intrusive constraint solver enhancements. *Pages 26–36 de Colloquium on Implementation of Constraint and LOgic Programming Systems, CICLOPS'02.* CW Reports, vol. 344.

Doyle Jon. 1979. A Truth Maintenance System. *Artificial Intelligence*, **12**(3), 231–272.

Drabent Wlodek, Małuszyński Jan & Pietrzak Pawel. 2000. *Analysis and Visualisation Tools for Constraint Programming.* Lecture Notes in Computer Science, vol. 1870. Springer-Verlag. Chap. 4. Locating type errors in untyped CLP programs, pages 121–150.

Ducassé Mireille & Langevine Ludovic. 2002. Automated Analysis of CLP(FD) Program Execution Traces. *Pages 470–471 de* Stuckey Peter J. (ed), *Proceedings of the 18th International Conference on Logic Programming, ICLP 02*. Lecture Notes in Computer Science, vol. 2401. Springer-Verlag.

Fages François. 2002. *CLPGUI User Manual*. INRIA Rocquencourt.

Fages François, Fowler Julian & Sola Thierry. 1995. A Reactive Constraint Logic Programming Scheme. *Pages 149–163 de* Sterling Leon (ed), *Proceedings of the Twelfth International Conference on Logic Programming, ICLP 95*. MIT Press.

Fages François, Fowler Julian & Sola Thierry. 1998. Experiments in Reactive Constraint Logic Programming. *Journal of Logic Programming*, **37**(1-3), 185–212.

Ferrand Gérard, Lesaint Willy & Tessier Alexandre. 2002. Theoretical Foundations of Value Withdrawal Explanations for Domain Reduction. *Electronic Notes in Theoretical Computer Science*, **76**(November).

Freuder Eugene C. 1982. A sufficient condition for backtrack-free search. *Journal of the ACM*, **29**(1), 24–32.

Freuder Eugene C. 1985. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, **32**(4), 755–761.

Fritzson Peter, Gyimothy Tibor, Kamkar Mariam & Shahmeri Nahid. 1992. Generalized Algorithmic Debugging and Testing. *ACM Letters on Programming Languages and Systems*, **1**(4), 303–322.

Georget Yan, Codognet Philippe & Rossi Francesca. 1999. Constraint Retraction in CLP(FD) : Formal Framework and Performance Results. *Constraints*, **4**(1), 5–42.

Ginsberg Matthew L. 1993. Dynamic Backtracking. *Journal of Artificial Intelligence Research*, **1**, 25–46.

Golomb Solomon W. & Baumert Leonard D. 1965. Backtrack programming. *Journal of the ACM*, **12**(4), 516–524.

Haralick R. M. & Elliott G. L. 1980. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Acta Informatica*, **14**, 263–313.

Junker Ulrich. 2001. QUICKXPLAIN : Conflict Detection for Arbitrary Constraint Propagation Algorithms. *Dans IJCAI'01 Workshop on Modelling and Solving problems with constraints.*

Jussien Narendra. 1997. *Relaxation de Contraintes pour les Problèmes dynamiques*. Ph.D. thesis, Université de Rennes I.

Jussien Narendra. 2001. e-constraints : explanation-based Constraint Programming. *Dans CP 01 Workshop on User-Interaction in Constraint Satisfaction.*

Jussien Narendra & Barichard Vincent. 2000. The PaLM system : explanation-based constraint programming. *Pages 118–133 de Proceedings of TRICS : Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000.*

Laburthe François & the OCRE project. 2000. CHOCO : implementing a CP kernel. *Dans Proceedings of TRICS : Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000.*

Laï Claude. 2000. *Analysis and Visualisation Tools for Constraint Programming.* Lecture Notes in Computer Science, vol. 1870. Springer-Verlag. Chap. 3. Assertions with constraints for CLP debugging, pages 109–120.

Lesaint Willy. 2002. Value Withdrawal Explanations : a Theoretical Tool for Programming Environments. *Dans* Tessier Alexandre (ed), *12th Workshop on Logic Programming Environments.*

Lloyd John W. 1995. *Declarative Programming in Escher.* Tech. rept. CSTR-95-013. Department of Computer Science, University of Bristol.

Marriott Kim & Stuckey Peter J. 1998. *Programming with Constraints : An Introduction.* MIT Press.

Meier Micha. 1994. Visualizing and solving finite algebra problems. *Dans Workshop on finite algebras.*

Meier Micha. 1995. Debugging Constraint Programs. *Pages 204–221 de* Montanari Ugo & Rossi Francesca (eds), *Proceedings of the First International Conference on Principles and Practice of Constraint Programming, CP 95.* Lecture Notes in Computer Science, vol. 976. Springer-Verlag.

Meier Micha. 1996. *Grace 1.0 User Manual.* European Computer Industry Research Centre, Munich, Germany.

Mohr Roger & Henderson Thomas C. 1986. Arc and Path Consistency Revisited. *Artificial Intelligence*, **28**(2), 225–233.

Montanari Ugo. 1974. Networks of Constraints : Fundamental Properties and Applications to Picture Processing. *Information Sciences*, **7**(2), 95–132.

Montanari Ugo & Rossi Francesca. 1991. Constraint Relaxation may be Perfect. *Artificial Intelligence*, **48**(2), 143–170.

Nilsson Henrik & Fritzson Peter. 1994. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, **4**(3), 337–370.

Ouis Samir, Jussien Narendra & Boizumault Patrice. 2002. COINS : a constraint-based interactive solving system. *Pages 31–46 de Proceedings of the 12th Workshop on Logic Programming Environments.*

Ouis Samir, Jussien Narendra & Boizumault Patrice. 2003. k-relevant explanations for constraint programming. *Pages 192–196 de* Russell Ingrid & Haller Susan (eds), *FLAIRS'03 : Sixteenth international Florida Artificial Intelligence Research Society conference.* AAAI Press.

Prosser Patrick. 1993. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, **9**(3), 268–299.

Puget Jean-Francois. 1994. A C++ Implementation of CLP. *Dans Proceedings of the Second Singapore International Conference on Intelligent Systems.*

Régin Jean-Charles. 1999. The Symmetric Alldiff Constraint. *Pages 420–425 de* Dean Thomas (ed), *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99*, vol. 1. Morgan Kaufmann.

Sabin Daniel & Freuder Eugene C. 1994. Contradicting Conventional Wisdom in Constraint Satisfaction. *Pages 10–20 de* Borning Alan (ed), *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP 94*. Lecture Notes in Computer Science, vol. 874.

Schiex Thomas & Verfaillie Gérard. 1993. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *Pages 48–55 de Proceeding of the Fifth IEEE International Conference on Tools with Artificial Intelligence, ICTAI 93*.

Schulte Christian. 1997. Oz Explorer : a visual constraint programming tool. *Pages 286–300 de* Naish Lee (ed), *Proceedings of the 14th International Conference on Logic Programming, ICLP 97*. MIT Press.

Shapiro Ehud Y. 1982. *Algorithmic Program Debugging*. ACM Distinguished Dissertation. MIT Press.

Sicstus. 2003. *SICStus Prolog 3.10.1 manual*. Swedish Institute of Computer Sciences.

Simonis Helmut & Aggoun Abder. 2000. *Analysis and Visualisation Tools for Constraint Programming*. Lecture Notes in Computer Science, vol. 1870. Springer-Verlag. Chap. 7. Search-Tree Visualisation, pages 191–208.

Smolka Gert. 1995. The Oz Programming Model. *Pages 324–343 de* van Leeuwen Jan (ed), *Computer Science Today : Recent Trends and Developments*. Lecture Notes in Computer Science, vol. 1000. Springer-Verlag.

Tessier Alexandre. 1997. *Approche, en terme de squelettes de preuve, de la sémantique et du diagnostic déclaratif d'erreur des programmes logiques avec contraintes*. Ph.D. thesis, Université d'Orléans.

Tessier Alexandre & Ferrand Gérard. 2000. *Analysis and Visualisation Tools for Constraint Programming*. Lecture Notes in Computer Science, vol. 1870. Springer-Verlag. Chap. 5. Declarative Diagnosis in the CLP scheme, pages 151–176.

Tsang Edward. 1993. *Foundations of Constraint Satisfaction*. Academic Press.

Van Emden Maarten H. 1995. Value Constraints in the CLP scheme. *Dans International Logic Programming Symposium, post-conference workshop on Interval Constraints*.

Van Hentenryck Pascal. 1989. *Constraint Satisfaction in Logic Programming*. Logic Programming. MIT Press.

Table des définitions

Table des théorèmes

Index

Explications de retraits de valeurs en programmation par contraintes et application au diagnostic déclaratif

Résumé La programmation par contraintes sur domaines finis a montré son efficacité pour traiter les problèmes difficiles, tant au point de vue de leur modélisation que de leur résolution. Les solveurs utilisés pour obtenir leurs solutions mêlent des techniques de réduction de domaine à des techniques d'énumération. Depuis peu, des notions d'explications ont été introduites afin de répondre à diverses problématiques telles qu'améliorer l'efficacité de la recherche, gérer les problèmes dynamiques ou encore expliquer les échecs.
Cette thèse propose principalement trois contributions au domaine de la programmation par contraintes. Tout d'abord la réduction de domaine est reformulée en terme ensembliste et vue comme un calcul de point-fixe. Ensuite, ce cadre permet une définition naturelle et générale d'explication, appelée arbre explicatif. Ces arbres de preuve sont inductivement définis par des règles exprimant le retrait d'une valeur comme conséquence d'autres retraits. Enfin, la vision des arbres explicatifs en tant que trace déclarative du calcul permet de les utiliser pour adapter le diagnostic déclaratif de réponse manquante à la programmation par contraintes.

Mots-clés CSP, contrainte, point-fixe, clôture, réduction de domaine, explication, mise au point, diagnostic.

Abstract Constraint programming over finite domains his efficiency to deal with difficult problems, so to the view point of their modelling that of their resolution. The solvers used to obtain their solutions mix domain reduction with labeling methods. Recently, notions of explanations have been introduced to answer different issues : to improve search, to deal with dynamic problems or to explain failure.
This thesis proposes three contributions to the domain of constraint programming. First, domain reduction is revisited in a set theoretical framework and seen as a fix-point computation. Next, this framework allows a natural and general definition of explanation, called explanation tree. These proof trees are inductively defined by rules expressing the removal of a value as a consequence of other removals. At last, the vision of the explanation trees as a declarative trace of the computation allows using them to adapt the declarative diagnosis of missing answer to constraint programming.

Key words CSP, constraint, fix-point, closure, domain reduction, explanation, debugging, diagnosis.

Thèse du Laboratoire d'Informatique Fondamentale d'Orléans présentée par Willy Lesaint.