UNIVERSITE PARIS VI – LABORATOIRE D'INFORMATIQUE DE PARIS 6 (L.I.P.6)
THEME OABJECT- AGENTS-SIMULATION OF INTELLIGENT SYSTEMS (O.A.S.I.S.)
4, place Jussieu, 75252 PARIS CEDEX 05

THESE DE DOCTORAT DE L'UNIVERSITE PARIS 6

DISCIPLINE : INFORMATIQUE

intitulée

Intégration de Mécanismes de Réécriture dans un Système de Satisfaction de Contraintes

présentée par

Anne LIRET

Soutenue le : Jeudi 12 octobre 2000

Devant le jury composé de :

Yves Caseau	BOUYGUES	Rapporteur
Claude Kirchner	LORIA	Rapporteur
Jean-François Perrot	LIP6	Examinateur
Hachemi Bennaceur	LIPN	Examinateur
Philippe Codognet	LIP6	Président
David Lesaint	British Telecom	Invité
François Pachet	Sony CSL	Directeur de thèse

Résumé

Les techniques de satisfaction de contraintes (CSP pour *Constraint Satisfaction Problems*) permettent de traiter les situations où l'on souhaite résoudre un problème en décrivant les caractéristiques des solutions, plutôt qu'en décrivant une procédure de résolution. Ces techniques, apparues dans les années 70, connaissent depuis quelques années un succès considérable et permettent aujourd'hui de spécifier et de résoudre de nombreux problèmes combinatoires difficiles, dans les domaines tels que l'ordonnancement, la planification ou la configuration de systèmes complexes. Néanmoins, de nombreux problèmes combinatoires qui relèvent en principe de ces techniques restent difficiles, voire hors d'atteinte. Une des causes de ces échecs est la trop grande généralité des techniques de CSP, qui sont conçues pour être indépendantes des problèmes particuliers à résoudre. Cependant, il existe de nombreux contextes dans lesquels il est possible d'améliorer les performances d'un système général de contraintes, à l'aide de connaissances spécialisées, sans pour autant nuire à la généralité des techniques employées. Cette ambivalence de comportement des problèmes de CSP rend essentiel le choix du langage dans lequel exprimer les contraintes et des connaisances qu'il peut vehiculer. Plus précisément, nous nous intéressons dans cette thèse à une manière particulière d'adapter dynamiquement des techniques générales (CSP) à des problèmes spécifiques, en utilisant le paradigme de la réécriture et celui de la programmation par objets.

Nous identifions et étudions d'abord (chapitre II) des situations concrètes dans lesquelles apparaissent les faiblesses d'un système général de CSP. Nous montrons comment une technique de réécriture permet de les contrebalancer. Nous en déduisons un cahier des charges pour un moteur de réécriture intégré à un système de CSP.

Dans une deuxième partie (chapitre III) nous proposons un moteur de Réécriture de Contraintes Symboliques appelé *RCS.* RCS est conçu pour être déclenché sans perturber le bon fonctionnement du système de CSP, à l'aide d'une technique de démons. RCS produit des contraintes redondantes, intégrées à l'ensemble de contraintes en cours de résolution. Nous décrivons l'implémentation de RCS sous la forme d'un *framework*, compatible avec les architectures de solveurs de contraintes existantes. Nous décrivons plus en détail son intégration avec le système de résolution de contraintes à domaines finis *BackTalk*, pris comme système référence.

Nous étudions ensuite (chapitre IV) la mise en œuvre de notre système complet (BackTalk + RCS) sur plusieurs problèmes combinatoires à domaines finis. Les résultats de nos expérimentations nous amènent à proposer des critères permettant de caractériser les situations identifiées au chapitre II, ainsi qu'un langage de définition de stratégies déclaratives pour contrôler précisément l'application de RCS. Pour ce faire, nous introduisons une représentation explicite de l'état de RCS. Nous utilisons ce langage pour observer le comportement de RCS et mesurer dynamiquement son utilité ou prévoir son comportement négatif. Ces stratégies dynamiques, locales aux règles de RCS permettent de guider le déclenchement de RCS et ainsi d'améliorer globalement la résolution des problèmes.

Nous concluons sur la pertinence de l'emploi d'une technique de raisonnement symbolique comme la réécriture dans le contexte de la résolution de problèmes combinatoires. Nous identifions des domaines d'applications qui nous paraissent particulièrement propices à une telle intégration.

Abstract

In the range of Constraint Satisfaction Problems (CSP), a great deal of techniques has been developed to deal with situations where problem solving is described by solution's characteristics, rather than a solving procedure. For several years, these techniques have been successfully applied to specify and solve a lot of difficult combinatorial problems, in numerous domains like planning, scheduling, time-tabling, dynamic network design. However, some combinatorial problems remain inappropriate or out of reach, essentially because of the general scope of the CSP formalism. This dilemma is not new: it's a classic effect of the NP-complete nature of CSP. However, there are a lot of contexts where we can expect increasing the performances of a general constraint system, with the help of specialised knowledge, without decreasing the generality of applied techniques. In this thesis we study a particular manner of dynamically adding local knowledge with the aim to adapt CSP techniques to specific problem properties, using Rewrite mechanisms.

First, we identify concrete situations that clearly put in evidence the drawbacks of a CSP solver. We propose a solver dedicated to Rewriting Symbolic Constraints, called *RCS*. RCS produces redundant constraints, integrated to the constraint store, while solving the problem. Our experiments upon finite-domain CSP problems lead us to propose on one hand, global criteria, which are particularly adapted to the category of cryptogram problems, and on the other hand, a language for defining declarative strategies to precisely control the execution of RCS. We use this language to observe the behaviour of RCS and dynamically measure the utility of RCS, regarding to the solution improvement. Finally we identify application domains which seem to be custom-built to such combination between constraints rewriting and constraint satisfaction-based solving.

Table des Matières

Introduction

> *"Constraints arise naturally in most areas of human endeavor. They are natural medium of expression for formalizing regularities that underlie the computational and (natural or designed) physical worlds and their mathematical abstractions."*
>
> *Pascal Van Hentenryck & Vijay Saraswat*
> *[Van Hentenryck & Saraswat 1997]*

Dans le domaine de la résolution de problèmes combinatoires, l'approche par satisfaction de contraintes tient une large part. Elle consiste à décrire les connaissances sous la forme de variables typées prenant leur valeur dans un domaine fini, et de contraintes, celles-ci pouvant être vues comme des prédicats logiques sur les variables. Un problème de satisfaction de contraintes (ou *CSP* pour *Constraint Satisfaction Problem*) consiste alors à supprimer des domaines les valeurs incompatibles avec au moins un prédicat ; nous présentons un panorama des principaux algorithmes de CSP existants au chapitre I-2. Les contraintes sont définies par des expressions d'une algèbre qui est constituée d'une part du type des variables, et d'autre part d'un ensemble d'opérateurs sur ces types. Le type des variables peut être très varié, par exemple les entiers, les booléens, les figures géométriques, les éléments de documents multimédias, les tâches d'un emploi du temps, voire imprévisible. Une contrainte peut prendre différentes formes, syntaxiques et algorithmiques. Un des problèmes consiste alors à trouver une solution en choisissant parmi les méthodes disponibles celles qui sont adaptées à la forme des contraintes. De nombreux et divers domaines (chapitre I-1) ont pu ainsi être explorés avec succès à l'aide des techniques de satisfaction de contraintes, comme par exemple l'ordonnancement de tâches, le déplacement d'agents mobiles dans un environnement virtuel, l'enseignement des caractères chinois, la reconnaissance de séquences temporelles soit de plans dans des documents audiovisuels [Carrive 2000], soit d'événements médicaux [Ramaux 1998].

Approche par satisfaction de contraintes (CSP)

Cette approche utilise un langage de contraintes général permettant de décrire des problèmes complexes sur des domaines non forcément mathématiques. Ce langage possède un statut particulier dans le cas des contraintes arithmétiques car il possède de bonnes propriétés d'ordre total ainsi qu'un large panel d'interprétations aux théorèmes et axiomes clairement définis. Le domaine des entiers a ainsi servi de cadre privilégié à de nombreuses applications de l'approche par satisfaction de contraintes.

Néanmoins cette approche souffre d'une fragilité chronique sur certains problèmes. Il suffit d'augmenter légèrement la taille du problème pour passer brutalement d'un problème aisé à un problème très difficile. Par exemple le célèbre problème du carré magique de taille 4*4 se résout facilement par les techniques de CSP standards en quelques retour-arrières. Quand on passe à 5*5, le nombre de retour-arrières est pratiquement multiplié par 10. Pour 6*6, la résolution est quasiment impossible… à moins d'introduire des connaissances locales à certains moments de l'énumération, par exemple en ajoutant au problème initial une contrainte redondante sur la somme de toutes les variables (qui est égale à $n*(n+1)/2$). Ce problème typique sera étudié en détail sous différents aspects dans ce document (chapitre I-2.2.1, II-1.1.2, III-3.3.3, IV-1.1.2).

Traditionnellement la notion de redondance dans un problème est interprétée comme une complexification inutile du problème ; en recherche opérationnelle, les contraintes redondantes ont un statut de contraintes qu'il faut supprimer. Notre approche montre que toutes les redondances ne sont pas négatives ; nous proposons une manière automatique d'exploiter ces redondances et de sélectionner celles évaluées comme les plus « positives » pour la résolution.

Notre problème se pose alors simplement en trois questions : 1) quelles contraintes inférer ? 2) comment le faire de manière transparente pour le système ? 3) quand utiliser ces inférences ?

Faiblesse des techniques CSP

Nous identifions au chapitre I-2.2.3, plusieurs situations dans lesquelles apparaissent les faiblesses des techniques de CSP et dans lesquelles la déduction de contraintes symboliques redondantes permet de les contrebalancer. Nous montrons qu'une *transformation syntaxique des contraintes basée sur la réécriture* de termes symboliques, produit un système équivalent ; ce dernier offre surtout l'intérêt d'être traité plus efficacement par les techniques de CSP voire résolu sans retour-arrrière. Nous étudions en détail au chapitre II, plusieurs problèmes combinatoires arithmétiques dans lesquels ce phénomène survient, parfois plusieurs fois durant la résolution du problème. C'est pourquoi il nous semble important d'automatiser l'utilisation combinée des techniques de CSP et de la réécriture de contraintes. Nous décrivons la problématique que cela pose à la section II-2, celle-ci étant par ailleurs introduite dans [Roy et al 1999].

En effet il se peut qu'une contrainte redondante n'implique aucune amélioration de la résolution ; dans ce cas la complexification du problème peut s'avérer néfaste à la résolution. Dans l'exemple du carré magique, la contrainte redondante de la somme est terriblement efficace. Mais la plupart des autres contraintes redondantes que l'on peut produire par combinaison linéaire, en revanche, conduisent à un effondrement des performances, même sur un carré 3*3 ! Nous proposons donc non seulement des techniques permettant de *produire des contraintes redondantes* (chapitre III-1, III-2, III-4), mais aussi des moyens efficaces de contrôler ces techniques pour éviter qu'elles ne deviennent nuisibles (chapitre III-4). Une implémentation est proposée au chapitre IV-2.

Approche symbolique

L'approche symbolique tient également une place importante dans le domaine de la résolution de problèmes. Elle décrit un domaine sous la forme d'une base de connaissances où le problème est un ensemble de prédicats et sa résolution met en œuvre une base de règles. Plus précisément un raisonnement symbolique peut être formalisé par un système de réécriture de termes symboliques. Pour guider vers une solution du problème, des *connaissances de contrôle* générales ou spécifiques du domaine étudié, indiquent quelles règles déclencher à quel moment au cours de la résolution. Ainsi les langages de règles ont été largement utilisés pour représenter les stratégies de contrôle. Par exemple le système

ELAN prototype des systèmes de calcul par des règles de réécriture (C.f. chapitre I-4) ; la plate-forme de programmation Claire intègre des règles de mise à jour au programme afin de clarifier et améliorer les algorithmes d'optimisation (C.f. chapitre I-4).

Faiblesse des connaissances de contrôle

Cependant, il existe peu de domaines où les connaissances de contrôle s'expriment de façon complète et permettent d'atteindre dans tous les cas la meilleure solution au moindre coût. Ces connaissances sont pour la plupart des connaissances empiriques dont l'élaboration nous semble essentiellement basée sur l'expérience [Schoenfeld 1985]. Par exemple, dans le domaine du calcul algébrique, [Foss 1987] regroupe plusieurs critères dynamiques, tels que *la complexité de l'équation croit, la dernière application de l'opérateur a échoué, trop d'opérateurs semblent applicables*. Mais pour construire un tel langage de contrôle, il faut formaliser la notion de *résultat* d'une technique (règles, filtrage de contraintes), résultat qui peut être négatif ou positif. Cette tâche est difficile lorsque l'on ne dispose pas d'un langage de contraintes et de règles où deux représentations cohabitent, symbolique et algorithmique. Cette cohabitation a lieu dans les langages de contraintes qui sont construits comme une extension d'un langage de programmation existant. En particulier, l'implémentation que nous proposons exploite cette double représentation qui est naturellement présente lors de l'intégration des contraintes dans les langages à objets.

La faiblesse de nos connaissances de contrôle provient en quelque sorte des limitations des connaissances humaines sur les processus automatiques. Or une des forces de l'intelligence humaine réside dans le fait que l'on sait que l'on peut faire des erreurs et que l'on sait reconnaître les situations à l'issue aléatoire. Dans ces situations, nous arrêtons nos activités afin d'observer le travail déjà effectué, évaluer l'adéquation des raisonnements utilisés et en déduire si nous sommes "sur la bonne voie ou pas ". La faiblesse des connaissances est alors compensée par le fait de pouvoir s'observer et changer dynamiquement. De nombreux travaux en Intelligence Artificielle [Kornmann 1993], [Clancey 1992] ont été conçus pour contrôler leur raisonnement en utilisant des stratégies dynamiques de même nature que les connaissances humaines ; [Cazenave 1997] propose même un mécanisme d'auto-observation afin d'apprendre des stratégies dans le jeu de GO. Sur le même principe, nous utilisons un contrôle dynamique sur le système de résolution. Ces stratégies font appel aux capacités humaines d'introspection et d'abstraction et sont pour cette raison qualifiées de "méta-connaissances ".

Par ailleurs, le problème du contrôle se pose sous deux aspects : la spécification d'une stratégie et le moment de son activation. Nous ne pouvons exprimer des stratégies de manière globale en raison de nos connaissances limitées. En revanche, les stratégies, elles, sont supposées s'appliquer sur la totalité du problème car l'on ne sait pas *a priori* qualifier les parties de problème les plus importantes à résoudre. Ainsi la spécification de stratégies pose un problème de niveau entre ce que l'on peut exprimer et la portée de ce que l'on veut contrôler. Par ailleurs le contrôle intervient à la suite d'observations globales du système, par exemple soit parce que le système *est arrivé à une impasse* et qu'il doit *rebrousser chemin*, soit à la fin ou au cours d'une étape de la résolution pour évaluer les résultats de cette étape ou encore parce que *le temps* passé à réaliser une étape *semble trop long*…Nous proposons de recourir au concept de méta-contrainte, contrainte manipulant le système de contraintes et d'inférence, pour faisant le lien entre l'analyse, l'action globale et la spécification locale.

Déduction de connaissances locales par réécriture

La réécriture met en œuvre un mécanisme relevant de l'approche symbolique ; en partant de types abstraits décrivant le domaine, *i.e.* les connaissances et leurs propriétés, la réécriture déduit successivement de nouvelles connaissances en exploitant les relations d'équivalence syntaxique, spécifiques du domaine. La terminaison correcte du système de réécriture est soumise à un ordre sur ces connaissances. La grande généralité de la réécriture en a fait un moyen privilégié pour simplifier l'expression d'un ensemble de termes, prouver la forme irréductible d'un terme, inférer de nouvelles connaissances équivalentes qui ne sont pas initialement exprimées dans le problème à résoudre. La réécriture a aussi été utilisée en intelligence artificielle, par exemple pour trouver la preuve de théorème ou de programmes, pour optimiser le code d'un programme… Le raisonnement symbolique proposé peut également être utilisé pour la preuve de théorème exprimé sous forme de CSP (C.F. chapitre IV-1.3). Le chapitre I-3 présente le cadre général de la réécriture de termes ainsi que d'autres méthodes relevant d'une approche symbolique.

La généralité du paradigme de réécriture de termes en fait un moyen de manipulation de contraintes symboliques, sur des domaines très divers, tels que la structure de programme, l'énoncé de problème, la simplification de termes fonctionnels, la normalisation des expressions arithmétiques et booléennes. La réécriture est maintenant intégrée dans les systèmes de résolution de contrainte symboliques ou de simplification de contraintes arithmétiques. Néanmoins son utilisation systématique au sein d'une méthode de résolution non symbolique (recherche systématique par retour-arrière, recherche heuristique) constitue encore une direction de recherche active. C'est dans ce cadre que nous plaçons notre travail.

Proposition dans ce document

Dans cette thèse, nous étudions comment faire fonctionner de manière complémentaire l'approche symbolique et l'approche par satisfaction de contraintes dans le cadre de la résolution de problèmes combinatoires. Plus précisément, nous avons construit un système de réécriture de contraintes symboliques, appelé RCS, qui permet d'introduire des connaissances spécifiques dans un problème de CSP, traité par un système de résolution par satisfaction de contraintes. RCS est ainsi spécifié pour déduire des contraintes redondantes à partir d'un système de contraintes donné (chapitre III-1 et III-2), ce qui permet d'adapter de manière transparente les techniques générales de CSP.

Les deux types de solveurs collaborent *via* un langage de contrôle déclaratif permettant d'adapter dynamiquement le comportement de l'un et l'autre des solveurs. D'après nos expériences, synthétisées aux chapitres II et IV-1, nous proposons des critères et un langage de stratégies déclaratives qui permet d'observer dynamiquement le comportement des techniques utilisées (filtrage de contraintes et règles de réécriture) et d'en déduire des mesures de l'utilité de chacune pour contrôler dynamiquement l'application de l'approche symbolique en collaboration avec un système de résolution de CSP. Le chapitre III-4 présente ce langage de contrôle dynamique. Plus précisément une stratégie peut être vue comme un ensemble de règles de mise à jour contextuelle qui se déclenche lorsqu'une modification particulière survient dans le système global ou dans le problème. De plus, grâce à l'utilisation de démons, l'activation des stratégies se fait de manière transparente sans perturber le bon fonctionnement des systèmes CSP et RCS (chapitre III-4.2.1 et III-4.2.2).

Nous avons défini une typologie des transformations utiles d'un ensemble de contraintes. Notamment la règle de composition de contraintes fonctionnelles (*PerformCtRule*, chapitre III-3.2.1) a été particularisée pour les contraintes linéaires d'égalités et inégalités. L'intérêt de la règle *linlin* (chapitre III-3.3.2) a ainsi pu être en mis en évidence pour la combinaison linéaire de contraintes deux à deux dans un système non carré de contraintes. Sur ce type de système, les méthodes d'algèbre linéaire produisent rarement un résultat intéressant : elles transforment systématiquement le système entier et ne produisent pas de réduction de domaine supplémentaire. Dans certains cas où le problème n'est pas assez dense, une composition ponctuelle de plus de deux contraintes peut débloquer la situation, par exemple dans le cas du problème des carrés magiques ; pour cette raison nous avons proposé *nlinRec* une généralisation de *linlin* à un nombre *n* de contraintes. Des stratégies ont alors été définies à deux niveaux : d'une part, la sélection du système de contraintes à composer et des exigences sur les contraintes résultant de la composition, d'autre part, les différentes règles s'appliquent en fonction les unes des autres. En particulier, la stratégie *DoBestRemaining* est bien adaptée lorsque le nombre de résultats potentiels est important. Les stratégies dynamiques sont indépendantes du système de déduction. La possibilité de les composer permet de prendre en compte la spécificité du problème. Nous avons étudié l'influence de trois stratégies principales (chapitre IV-4.2.3) : 1) une règle est bloquée lorsqu'elle est inutile *p* fois de suite ; 2) une règle R_0 s'active lorsqu'il s'est produit *p* événements d'un certain type (par exemple un échec) parmi *j* règles R_1,..R_j ; 3) une règle se bloque ou s'active lorsqu'une mesure globale atteind son seuil. L'abstraction des événements permet de redéfinir les stratégies existantes ; par exemple la notion d'échec peut regrouper soit un non-déclenchement, soit un ajout de contrainte triviale qui a été court-circuité.

Appliquées sur *linlin et nlinRec*, ces stratégies ont permis d'obtenir la première solution du problème des carrés magiques où la somme est inconnue, pour n=5 en moins de 4 minutes. Pour n=4, la résolution est très rapide, ne necéssitant pas de retour-arrière et seulement 7 choix.

A partir de nos expérimentations, nous proposons des critères caractérisant les situations où RCS est inutile de manière sure ou, au contraire, possède de fortes chances d'avoir des conséquences positives. Ces expériences sont synthétisées dans le chapitre IV-1. Nous en déduisons un langage de stratégies déclaratives afin de contrôler finement le déclenchement de RCS. Les stratégies permettent d'observer et de mesurer dynamiquement l'évolution du comportement de RCS et son utilité *a priori*. L'implémentation du système est décrite brièvement au chapitre IV-2.

Un *framework* de satisfaction et de réécriture de contraintes

Comme nous l'avons dit plus haut, nous nous plaçons dans le cadre de problèmes à domaine fini décrit au moyen d'un langage de contraintes général. Un langage de contraintes peut être construit de deux manières, soit à partir d'un langage étendu avec des contraintes, soit à partir de contraintes que l'on manipule à l'aide d'un langage adapté que l'on a choisi. Le système de satisfaction de contraintes, BackTalk, que nos avons pris comme référence dans cette thèse (C.F. chapitre I-2.2.2), est conçu selon une approche par *framework*, dont la problématique est clairement décrite dans [Roy et al. 1999] et [Roy et al. 2000]. Cette approche utilise un langage non fini supporté par un langage de programmation par objets, où les contraintes sont les instances de classes.

Chapitre I : Contraintes et raisonnement symbolique

"Constraint programming brings us closer to true declarative programming. […] (But) it is one thing to say "all one has to do is express the problem constraints". It is another to express them in manner which permits efficient solution. […] There can be many ways of representing a problem with constraints. In particular redundant constraints may hinder, or dramatically help the solution process."

E.C. Freuder, In Pursuit of the Holy Grail, [Freuder 1997]

La première section de ce chapitre présente le formalisme CSP et conclut sur la pertinence de considérer le *framework* de satisfaction de contraintes BackTalk [Roy 1998] comme cadre de référence. Dans ce cadre, les contraintes sont représentées par un ensemble de *procédures* de filtrage. La seconde partie décrit les techniques permettant de réaliser des raisonnements symboliques sur les contraintes. Dans ce deuxième cadre, les contraintes sont alors considérées comme des *formes syntaxiques* de première classe. La cohabitation de ces deux formes de représentations relève du domaine de la *collaboration* de solveurs, analysé dans la troisième section de ce chapitre.

1 Des contraintes

La notion de contrainte est, de nos jours, utilisée dans de nombreux domaines d'application et revêt en pratique des sens très variés.

Dans les bases de données, les contraintes représentent le plus souvent des «contraintes d'intégrité » portant sur les attributs des objets ou tables de la base. Par exemple on peut exiger que l'attribut *age* de la classe (ou table) *Personne* soit toujours positif. Leur vérification permet d'assurer la cohérence de la base et de filtrer la partie des informations à explorer lors d'une requête. Les contraintes sont testées automatiquement au moment du traitement de la requête. La grande quantité d'information à traiter oblige d'une part à limiter l'arité des contraintes (en général 1 ou 2) à d'autre part les représenter d'une manière optimisée pour la propagation efficace des modifications de la base [Asirelli et al. 1996]. En particulier, une contrainte d'intégrité est compilée statiquement sous une forme procédurale ou déductive jouant le rôle de pré/post-condition à la mise à jour de la base.

Les contraintes ont été historiquement utilisées dans le domaine des interfaces graphiques, pour propager des perturbations dues à des actions utilisateur. Dans ce contexte, les contraintes permettent de rétablir la stabilité d'un système après une perturbation. Le système précurseur Thinglab [Borning 1981] proposait un modèle de contraintes réactives intégré à Smalltalk. Ces contraintes représentent des relations de dépendance géométrique entre des objets graphiques (figures géométriques, et plus généralement composants de l'interface utilisateur). Dans cette lignée, de nombreux travaux ont proposé diverses extensions et améliorations du modèle de base : [Borning et al. 1992] propose un modèle de contraintes pour représenter des hiérarchies de contraintes permettant de représenter la notion de priorité d'une contrainte. [Borning & Freeman-Benson 1995] étend un langage de programmation avec des contraintes à la Thinglab par exemple pour paramétrer l'apparence de graphiques dans des environnements interactifs. De nombreux algorithmes ont été développés (la fameuse lignée des Delta-Blue et Sky-Blue [Sannella 1994]). Une formalisation et une unification de tous ces algorithmes ont été proposées avec la notion de *dataflow-constraints* [Trombettoni & Neveu 1997].

Enfin, les contraintes sont utilisées pour traiter des problèmes combinatoires. Dans ce contexte, les contraintes décrivent les *solutions* d'un problème combinatoire. Les systèmes dits de *satisfaction de contraintes* intègrent à la fois un langage de spécification du problème combinatoire et des techniques de résolution efficaces, comme c'était le cas dans le système précurseur Alice [Laurière 1976].

C'est ce dernier sens du mot contrainte que nous adoptons dans notre travail. Dans le chapitre suivant, nous proposons un état de l'art des systèmes de satisfaction de contraintes.

2 Satisfaction de contraintes

Le formalisme de la satisfaction de contraintes recouvre à la fois un langage de description de problèmes combinatoires et une batterie d'algorithmes généraux pour les résoudre.

Un problème de satisfaction de contraintes est défini par un triplet, noté (V, D, K) tel que :

- V est un ensemble fini de variables ; chaque variable v_i étant associée à un ensemble D_i de valeurs possibles, ou domaine, noté aussi $dom(v_i)$,
- D est défini par le produit cartésien des D_i ;
- K est un ensemble de contraintes reliant les variables de V.

Une contrainte C est une relation entre k variables de V ($k \in [1,N]$, $|V| = N$), notée $C(v_1,\ldots,v_k)$. Par définition, l'arité de C est k, et est notée par *arity*(C) ; on dit que C « porte sur » $v_1,\ldots, v_k$. Pour un n-uplet de valeurs $t = (\alpha_1,\ldots,\alpha_k)$, on note $C(t)=C(\alpha_1,\ldots,\alpha_k)$ la valeur booléenne obtenue en remplaçant dans la relation C d'arité k, chaque occurrence de la variable v_i par la valeur α_i. Dans ce document, nous ne considérerons que des problèmes de CSP sur des domaines finis.

[Mackworth 1977] définit une contrainte de manière extensionnelle comme le sous-ensemble des n-uplets t dans $D_1 x \ldots x D_k$ tel que $C(t) = Vrai$, C étant la relation de la contrainte. En pratique, les solveurs de contraintes exploitent des définitions intentionnelles des contraintes. Dans ce cas une contrainte peut être représentée de deux manières, soit par une formule logique, soit par une fonction vérifiant la satisfaction de la relation de la contrainte. Cette double nature de la contrainte est interprétée différemment selon le langage utilisé.

Une solution d'une contrainte $C(v_1,\ldots,v_k)$ affecte ses variables avec des valeurs $a_1,\ldots, a_k$ appartenant à D1,…,Dk, telles que la relation C soit évaluée à *Vrai*. Un algorithme de résolution de problème de CSP calcule alors les n-uplets ($a_1,\ldots, a_N$) de valeurs appartenant à $D_1 x \ldots x D_N$, qui satisfont toutes les contraintes de K, c'est à dire pour toute contrainte C_i de K, C_i portant sur $vi_1,\ldots vi_{ki}$, on a $C_i(\alpha i_1,\ldots,\alpha i_{ki}) = Vrai$.

Voici un exemple de problème dans lequel les contraintes sont définies de manière intentionnelle : le carré magique 4*4. Ce problème consiste à remplir une grille 4*4 à l'aide d'entiers différents de 1 à 16, de telle manière à ce que les lignes, les colonnes et les deux diagonales soient toutes de somme égale.

Variables et domaines

$x_{11}, x_{12}, x_{13}, x_{14}, x_{21}, \ldots, x_{44}$. Pour tout i,j $\in [1,4]$, $dom(x_{ij}) = [1,16]$.

Pour exprimer simplement l'égalité des sommes, on introduit une variable supplémentaire σ qui représente la somme d'une ligne (ou d'une colonne). Le domaine de cette variable «artificielle» est alors l'ensemble des sommes possibles de quatre entiers, soit [4, 64].

Contraintes

Pour tout k,l $\in [1,4]$, $x_{ij} \neq x_{kl}$.

Lignes et colonnes : Pour tout $i \in [1,4]$, $\sum_{j=1}^{4} x_{ij} = S$, $\sum_{j=1}^{4} x_{ji} = \sigma$.

Les deux diagonales : $\sum_{i=1}^{4} x_{ii} = S$ et $\sum_{i=1}^{4} x_{i,4-i} = \sigma$.

Pour résoudre un tel problème de CSP de manière complète, les systèmes de satisfaction de contraintes peuvent utiliser trois types de méthodes, en fonction de la représentation des contraintes. Dans une approche symbolique, les méthodes syntaxiques transforment une contrainte en une autre équivalente, de manière répétitive ; ces méthodes sont en général basées sur un mécanisme de réécriture [Castro & Kirchner 1998]. La forme résolue d'une contrainte est alors une formule définie comme irréductible. Les méthodes sémantiques utilisent une représentation complexe englobant la formule logique de la contrainte et un automate définissant les différents états de la contrainte. Cet automate correspond à la procédure de filtrage de la contrainte [Dauchet 1994]. Dans une approche CSP, les méthodes de résolution combinent un algorithme d'énumération avec des procédures de cohérence locales à chaque contrainte [Mackworth, 1977]. Les algorithmes fondés sur ce principe comme le *full look-ahea*d [Gaschnig 1977], le *forward-checking* [Haralick & Elliott 1980], réalisent une énumération avec retour-arrière (*BackTracking*). Ils parcourent l'espace de recherche de manière arborescente, en posant des « points de choix » sur les variables. Afin de couper l'arbre de recherche et donc d'accélérer la résolution, les techniques de cohérence réduisent les domaines à chaque cycle d'énumération. Ces techniques retirent des domaines les valeurs incohérentes avec au moins une contrainte du problème. Une valeur β d'un domaine D_i est incohérente avec une contrainte $C(v_1,\ldots v_k)$ si et seulement si pour tout $(\alpha_1,\ldots,\alpha_{i-1},\alpha_{i+1},\ldots\alpha_k)$ de $D_1 x \ldots x D_{i-1} x D_{i+1} x D_k$, $C(\alpha_1,\ldots,\alpha_{i-1},\beta,\alpha_{i+1},\ldots \alpha_k) = Faux$. Autrement dit, β ne peut pas satisfaire C quelles que soient les valeurs des autres variables. La réduction d'un domaine entraîne alors des réductions d'autres domaines, grâce à un mécanisme de propagation.

Plusieurs définitions de la cohérence existent comme nous le verrons dans la prochaine section. De nombreuses variantes de l'algorithme chronologique de base ont été introduites, comme les stratégies de retour-arrière intelligent inspirées des travaux en programmation logique [Prosser 1993a, 1993b], l'exploitation de connaissances expertes [Pfeffer 1992] ou des contraintes disjonctives [Laurière 1986] afin de guider l'énumération.

2.1 Cohérence d'un CSP

Plusieurs techniques de cohérence existent selon le nombre de contraintes considérées ensemble pour réduire les domaines de valeurs, et la complétude de la réduction des domaines de valeurs. La cohérence complète supprime toutes les valeurs impossibles et nécessite pour cela des domaines finis. La cohérence partielle vérifie seulement les bornes des domaines, en supposant un ordre. Dans ce document, nous considérons les problèmes CSP sur des domaines ordonnés et finis. Les algorithmes de cohérence d'arc vérifient les contraintes unes à unes, tandis que les algorithmes de cohérence de chemin vérifie un ensemble de contraintes reliées entre elles. Ces derniers, bien que difficile à metre en oeuvre, sont plus efficaces en termes de réduction de domaines. Un troisième niveau de cohérence consiste à tester une contrainte lorsqu'elle est totalement instantiée. Ce niveau, que nous appelons «*value-cohérence* », ne maintient pas la cohérence contrairement aux algorithmes de réduction de domaine mais la vérifie *a posteriori*. En particulier, il est utilisé dans l'algorithme *Génération-Test* qui ne vérifie que les contraintes totalement instantiées.

Les différents niveaux de cohérence locale sont combinables au sein d'un même réseau de contraintes. En particulier, certaines contraintes peuvent changer de niveau de cohérence en cours de résolution, par exemple en fonction de la taille du domaine, du degré de sollicitation de la contrainte, ou de propriétés du type de contraintes (réflexivité, symétrie).

2.1.1 Cohérence complète

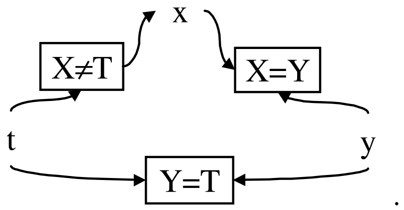
La cohérence complète d'un domaine consiste à supprimer les éléments pour lesquels la ou les contrainte(s) ne peuvent pas être respectées quelles que soient les valeurs des autres variables. Alan Mackworth a introduit les notions de cohérence de nœud, d'arc et de chemin. Les premiers algorithmes [Mackworth & Freuder 1993, Mackworth 1977] s'appliquent aux contraintes binaires, la cohérence d'arc par AC-1 ($O(end^3)$, où n est le nombre de variables, e le nombre de contraintes et d la taille maximale des domaines) et la cohérence de chemin par PC-1 ($O(n^5d^5)$). La cohérence de nœud consiste à supprimer avant le début de la recherche arborescente, les valeurs qui contredisent de manière sure une contrainte, par exemple les valeurs négatives de X dans la contrainte $X>Y^2$. En quelque sorte, la cohérence de nœud reformule le problème en supprimant les valeurs trivialement inutiles. La cohérence d'arc est la notion de cohérence la plus utilisée en pratique, car la plus efficace. Plus précisément, deux variables V_i et V_j ont des domaines arc-cohérents avec une contrainte C portant sur V_i et V_j, si et seulement si V_i et V_j vérifient la propriété suivante [Mackworth 1977] :

$$\forall\, x \in D_i, \exists\, y \in D_j, \text{ tel que } C(x,y) = \text{Vrai} \quad \text{ET}$$
$$\forall\, y \in D_j, \exists\, x \in D_i, \text{ tel que } C(x,y) = \text{Vrai}.$$

La cohérence de chemin est une extension de la cohérence d'arc au traitement de plusieurs contraintes ensemble et réalise une cohérence globale. La cohérence de chemin nécessite de modifier la topologie du problème en complétant le graphe de contraintes. En outre elle est coûteuse et difficilement applicable sur des chemins de contraintes hétérogènes. Au contraire, la cohérence d'arc peut être appliquée à tout type de contrainte indépendamment de la topologie du problème. De plus sa mise en œuvre peut être optimisée en fonction de la nature des contraintes, ce qui facilite la création de bibliothèques de contraintes.

Des améliorations de l'algorithme initial ont porté sur la complexité en temps et en espace. AC-3 et PC-2 réduisent la complexité en temps [Mackworth & E.C. 1985]. [Laurière 1976] et [Mohr & Henderson 1986] trouvent AC-4, dont la complexité en temps dans le pire des cas est optimale (*O(ed²)*) pour des contraintes binaires. En revanche AC-4 requiert une grande place mémoire (*O(ed^r)*, où *r* est l'arité maximale des contraintes). Issu des mêmes réflexions, PC-3 améliore le temps d'un facteur *n²*. Un compromis a été proposé avec la *k*-cohérence, une cohérence sur les chemins de contraintes de longueur *k*.

Exemple 1: X in 1..10, Y in 1..10, T in 3..20



La cohérence d'arc complète trouve X *in* 3..10, Y *in* 3..10, T *in* 3..10.

La cohérence de chemin complète détecte l'incohérence par transitivité de l'égalité lors de la propagation des valeurs de X.

Les algorithmes de cohérence binaire procèdent en testant pour chaque valeur α d'une variable V_1, l'existence d'une valeur β compatible pour l'autre variable V_2 puis en réitérant les tests à partir de V_2. Lorsqu'une valeur β est trouvée, un lien $\alpha \rightarrow \beta$ mémorise le test ; sinon α est retiré du domaine de V_1 et tous les liens pointant sur α sont supprimés. Ensuite, l'algorithme recommence les tests sur toutes les valeurs de V_1 et V_2 non mémorisées. En supposant l'existence d'un ordre total sur les domaines, AC-6 [Bessière 1994] améliore le coût en espace au pire cas en mémorisant les n-uplets de valeurs déjà testés. Quand aucun β n'est trouvé, AC-6 évite les tests inutiles sur les valeurs précédant α dans le domaine de V_1, celles-ci n'étant déjà pas compatibles avant le retrait de α. AC-7 [Bessière et al. 1995] améliore le coût moyen en espace mémoire (*O(ed)*) en mémorisant un lien bidirectionnel $\alpha \leftrightarrow \beta$..Pour certains types de contraintes binaires, la procédure d'arc-cohérence est connue et souvent plus rapide que l'algorithme général. [Deville & Van Hentenryck 1991] propose AC-5, un algorithme général d'arc-cohérence dont la spécialisation pour les contraintes monotones et fonctionnelles a une complexité en temps linéaire en *O(ed)*. Le schéma d'algorithmes d'arc-cohérence *GAC-schema* [Bessière & Régin 1997] généralise AC-7 et s'instancie en algorithmes spécifiques de même complexité que AC-7. [Van Beek 1994] énonce des conditions pour obtenir la cohérence locale sur des contraintes binaires.

2.1.2 Cohérence partielle

La cohérence partielle a été largement utilisée pour faire face à l'inconvénient qu'a la cohérence complète de systématiquement parcourir les domaines. Lorsque le domaine est une union d'intervalles, la cohérence partielle utilise une approximation de celui-ci sous la forme d'un seul intervalle englobant («*box* »). La perte d'information due au fait que des valeurs impossibles peuvent être oubliées est compensée par un gain en temps. [Benhamou & Granvilliers 1996] montre que les systèmes de réécriture sont capables de mettre les contraintes sous une forme telle que la perte d'information soit plus faible, dans le cas de variables réelles.

Exemple 2 : reprenons l'Exemple 1 avec X in [1..5] ∪ [7..10], Y in [1..10], Z in [3..20].

La cohérence partielle d'arc (pour une contrainte) trouve *dom*(X) = [3..5] ∪ [7..10] mais, contrairement à une cohérence complète, elle ne supprime pas la valeur 6 du domaine de Y et Z, qui devient [3..10]. La cohérence partielle de chemin consisterait à changer les bornes de manière à ce qu'elles vérifient toutes les contraintes ; elle détecterait l'incohérence car aucune borne de X, Y et Z ne convient.

La cohérence partielle permet de traiter les contraintes dans le cas où les domaines sont des intervalles réels. Dans ce domaine, la transformation symbolique de contrainte permet de combiner les avantages de plusieurs algorithmes. Ainsi la *int*-cohérence [Granvilliers 1998b] combine les avantages de la *hull*-cohérence et de la *box*-cohérence grâce à une transformation symbolique des contraintes de manière à isoler si possible, une variable n'apparaissant qu'une seule fois dans la contrainte. La *box*-cohérence[1], utilisée dans Numerica et Newton [Van Hentenryck et al. 1997a, 1997b] traite les contraintes dans leur représentation complexe ce qui permet des calculs plus précis dans le cas de domaines réels et plus rapides pour des contraintes complexes (complexité linéaire en nombre de variables à tester). La *hull*-cohérence utilisée dans CLP(BNR) [Benhamou & Older 1997], DecLIC et PrologIV [Benhamou & Touraïvane 1995] s'applique sur les contraintes complexes en supposant qu'elles sont décomposées en conjonction de contraintes simples.

La cohérence de bornes ou borne-cohérence, utilisée dans BackTalk, propose d'imposer la satisfaction d'une contrainte uniquement pour les valeurs extrêmes des domaines, ces derniers étant supposés ordonnés et finis. Plus précisément, dans le cas d'une contrainte binaire C portant sur deux variables V_i et V_j, V_i et V_j ont des domaines *borne-cohérents* avec C, si et seulement si V_i et V_j vérifient la propriété suivante :

$$\forall\ x \in \{min(D_i), max(D_i)\}, \exists\ y \in D_j, \text{ tel que } C(x,y) = \text{Vrai} \quad \text{ET}$$
$$\forall\ y \in \{min(D_j), max(D_j)\}, \exists\ x \in D_i, \text{ tel que } C(x,y) = \text{Vrai}.$$

L'algorithme de borne-cohérence s'exécute en $O(ed)$, où e est le nombre de contraintes, et d la taille du plus large domaine. L'algorithme est également valable pour les contraintes n-aires. Pour les contraintes monotones, [Roy 1998] en propose une version optimisée de manière à être en temps constant ou linéaire ($O(n)$, n étant l'arité de la contrainte).

2.1.3 Cohérence globale

Il existe beaucoup de situations où l'issue de la résolution est évidente si on prend en compte plusieurs contraintes ensembles [Bessière & Régin 1998]. Ainsi les méthodes de filtrage de [Codognet & Nardiello 1996] réalisent la cohérence d'une conjonction de contraintes binaires. Dans le cas de domaines réels pseudo-discrétisés, la 3B-cohérence [Lhomme et al. 1998] réalise une cohérence partielle sur tous les chemins de trois variables reliées par deux contraintes binaires. Dans la même optique, la *conjonctive-consistency* [Bessière & Régin 1998] tente de combiner des contraintes deux à deux, en utilisant une extension de la structure de AC-7. Les expériences montrent qu'elle améliore la qualité de la résolution quand les connaissances de l'expert portent sur plusieurs contraintes indissociables. Néanmoins peu d'algorithmes réalisent la cohérence globale partielle. Dans sa thèse, Pierre Roy écrit (page 73): "La raison essentielle du manque de systèmes fondés sur la cohérence de chemin est précisément qu'il n'existe pas de concept équivalent au filtrage pour cette notion. Le filtrage de chemin de contraintes consisterait à redéfinir une notion plus faible que la cohérence de chemin pour certains types de chemins (ou groupes) de contraintes. Dans la pratique, l'utilisation de contraintes globales est une manière de regrouper plusieurs contraintes en une seule dont le filtrage revient à faire le filtrage d'un

[1] La *box*-cohérence s'applique aux intervalles réels en calculant la cohérence sur des intervalles encadrant chacune des bornes]min-ε, min+ε[et] max-ε, max+ε [.

groupe de contraintes". Nous proposons une concrétisation de la cohérence imaginé par P. Roy et définissons la «2C-cohérence », qui réalise une extension de la cohérence de borne à tous les chemins de deux contraintes, quelles soient binaires ou n-aires.

Notations:

Pour toute contrainte C_i d'arité n sur $v_1,\ldots, v_n$, pour tout w de $D_w = dom(v_1) \times\ldots\times dom(v_n)$, on note $C_i \cap C_j$ l'ensemble des variables communes à C_i et C_j. On définit les ensembles $N_i = \{min(v_1), max(v_1)\}\times\ldots\times\{min(v_n), max(v_n)\}$ et $\overline{N_i} = \{u \in N_i \ / \ C_i(u)\}$. On note $u(v)$ la valeur de la variable v dans le n-uplet u.

Définition 1 : définition de la 2C-cohérence dans le cas général

> Un chemin $C_1,..,C_n$ est 2C-cohérent si et seulement si tous les chemins de longueur 2 sont 2C-cohérents.
>
> Deux contraintes C_i et C_j telles que $Ci \cap C_j \neq \emptyset$ sont 2C-cohérentes si et seulement si :
>
> - C_i et C_j sont cohérentes aux bornes, et
> - Pour tout n-uplet u de $\overline{N_i}$, il existe un n-uplet de valeurs w de D_w tel que $C_j(w)$ et $\forall$ v $\in C_i \cap C_j$, w(v) = u(v), et
> - Pour tout n-uplet w de $\overline{N_j}$, il existe un n-uplet de valeurs u de D_u tel que $C_i(u)$ et $\forall$ v $\in C_i \cap C_j$, u(v) = w(v).

Voici un algorithme de filtrage possible pour la 2C-cohérence d'un système de n contraintes :

```
2C-cohérence(C₁,…,Cₙ) =
  Début
   Borne-cohérence(Cᵢ).
   Borne-cohérence(Cⱼ).
   Pour tout i,j de {1,…,n}, tels que Cᵢ ∩ Cⱼ ≠ ∅, Faire
      Vᵢ  := {v1i,…,vni} les variables de Cᵢ
      Vⱼ  := {v1j,…,vnj} les variables de Cⱼ.
      Nᵢ  := ∅.
      Soit u=(a₁,…,aₙ), où ∀k, aₖ ∈{min(vki) ; max(vki)},
      Si Cᵢ(u) Alors Nᵢ := Nᵢ ∪ {u} Sinon choisir un autre u.
      Pour toute variable v de Δ(Vⱼ , Vᵢ), Faire
         modifier min(v) et max(v) pour qu'il existe au moins un
         n-uplet w tel que Cⱼ(w) et
         ∀k, si il existe p tel que vkj = vpi, w(vkj) = u(vpi),
               sinon w(vkj) ∈ dom(vkj).
      FinFaire
   FinFaire
  Fin
```

Nous comparerons au chapitre II, l'effet du raisonnement symbolique combiné à une cohérence de bornes avec la réduction de domaines résultant de la 2C-cohérence, sur un ensemble donné de contraintes.

2.1.4 Contraintes globales

La cohérence partielle ne vérifie la satisfaction de la contrainte que pour certaines valeurs du domaine. Or dans le cas de domaines non ordonnés, il n'y a pas *a priori* d'élément préférentiel, ce qui rend obligatoire l'utilisation de la cohérence complète pour traiter les contraintes sur des objets. Il est alors avantageux de regrouper plusieurs contraintes au sein d'une seule contrainte complexe globale et d'y associer un algorithme optimisé de cohérence. [Mohr & Masini 1988] montre dans le cas de GAC-4, l'extension de AC-4 aux contraintes n-aires, qu'il n'y a pas de perte d'information par rapport à une cohérence de chemin sur le CSP binaire dual. Dans le cas général, l'utilisation d'une contrainte globale produit une réduction de domaines plus efficace que celle obtenue par sa décomposition en contraintes binaires. Par exemple la contrainte *AllDifférent*(x,y,z) [Régin 1994, 1996] gère le cas où les variables x, y et z ont le même domaine réduit à deux éléments, contrairement aux contraintes binaires $\{x\neq y,\ y\neq z,\ x\neq z\}$. Certaines contraintes spécifiques ne peuvent s'exprimer que par une contrainte globale, comme la contrainte de séquencement et de rotation de tâches [Régin & Puget 1997], les contraintes d'utilisation disjonctive de ressource en ordonnancement [Le Pape & Baptiste 1997]. Néanmoins, devant le grand nombre de méthodes spécialisées, [Bessière & Régin 1997] a proposé un schéma général regroupant les points communs à certaines procédures de cohérence non binaires. En particulier, il permet de traiter les contraintes exprimant un ensemble de combinaisons de valeurs interdites au lieu de combinaisons de valeurs valides. Ces dernières représentent en pratique les contraintes dites «molles » dont les solutions sont nombreuses et les instantiations incorrectes rares ; il est alors plus rapide de tester l'interdiction d'une instantiation plutôt que sa validité.

2.1.5 Changement de niveaux de cohérence

Au sein d'un même réseau de contraintes, les contraintes peuvent posséder différents niveaux de cohérence. La figure ci-dessous résume les différents niveaux et les ordonne selon leur puissance de réduction de domaine.

	Cohérence partielle locale	Cohérence complète locale	Cohérence partielle globale	Cohérence complète globale
Value-Coh.	< Borne-Coh. , 2B-Coh.	< Arc-Coh.* (non comparable)	3B-Coh. < 2C-Coh.	< k-coh. < Path-Coh.

De plus le niveau des contraintes peut évoluer en cours de résolution pour améliorer la réduction de domaines. Ainsi Alice traite les contraintes non linéaires d'abord par arc-cohérence, puis par value-cohérence lorsque les domaines deviennent assez petits. En général une contrainte non binaire est traitée par une borne-cohérence, sauf si elle contient uniquement deux variables non affectées. Autres exemples, une contrainte $x = y/z$ ne propage rien tant que le domaine de z contient 0 ; la contrainte d'appartenance à un ensemble défini analytiquement [Laurière 1976] ne fait rien tant que l'ensemble ne peut pas être calculé explicitement. Par ailleurs, les algorithmes AC-6 et AC-7 fournissent une complexité optimale au pire cas sur l'hypothèse d'une implémentation en temps constant des primitives d'accès aux valeurs du domaine, `n-th-Successeur(value,n)` et `n-th-predecesseur(value,n)`. Cette hypothèse n'est en général pas vérifiée lorsque le domaine est fractionné en union d'intervalles. Dans ce cas les primitives deviennent linéaires en *n*. La diminution du degré de cohérence permet alors de préserver un temps de réponse borné indépendamment de l'implémentation du domaine, mais au prix d'une baisse de qualité de cette réponse.

D'un point de vue technique, si aucune raison n'interdit de modifier le niveau de cohérence au sein d'un ensemble de contraintes, en pratique, il est cablé et non redéfinissable pour les contraintes les plus souvent utilisées. Ainsi en BackTalk [Roy et al. 1998] ou CHIP [Dincbas et al. 1990], les contraintes arithmétiques n-aires réalisent une borne-cohérence, les contraintes binaires et fonctionnelles sont traitées par arc-cohérence. Plus généralement, le filtrage d'une contrainte dépend de la nature de la contrainte. Nous verrons plus tard comme la réécriture automatique des contraintes pendant la résolution permet justement de réaliser un changement de niveau de cohérence dans les cas les plus favorables.

2.2 BackTalk : un système de référence

L'intégration des contraintes dans les langages de programmation a contribué au développement de deux communautés, l'une utilisant les langages logiques, l'autre les langages à objets. Dans cette dernière l'approche par *framework* est très prometteuse car elle permet d'exploiter les propriétés des contraintes. Celle-ci a été concrétisée par le système BackTalk [Roy 1998], qui reprend l'architecture des solveurs de contraintes classiques tout en offrant les bonnes propriétés d'un *framework* objet. Dans le cadre de notre étude, nous avons choisi d'utiliser BackTalk comme système de référence.

2.2.1 Contraintes et langages de programmation

Intrinsèquement une contrainte possède une nature duale. D'une part une contrainte est définie par une formule logique représentant un ensemble en intension, manipulable dans un système d'inférence, interprétable dans une syntaxe donnée. Cette représentation abstraite facilite le traitement collectif de contraintes (combinaison logique). D'autre part les contraintes sont habituellement traitées de manière individuelle. Une des raisons réside dans le fait qu'une contrainte est vue comme une procédure destinée à mettre à jour l'ensemble de données en filtrant les valeurs compatibles. Dans le cadre d'un langage de programmation par objets, la notion de classe englobe l'aspect symbolique et l'aspect procédural des contraintes.

Dans ces langages de programmation logique par contraintes, appelés CLP, les contraintes permettent d'exprimer de manière déclarative, le contrôle du déclenchement des règles [Cohen 1990], [Marriott & Stuckey 1998], [Codognet 1995]. Cette extension est naturelle du fait que la notion de variable logique est compatible avec celle des variables de CSP. L'unification logique de prédicats a ainsi été étendue avec un moteur de satisfaction de contraintes, du fait que l'algorithme de parcours d'une base de règles peut alors être vu comme un algorithme simple d'énumération de CSP. Cette relative simplicité de mise en œuvre explique probablement le succès de l'approche, comme le montre le grand nombre de systèmes de CLP réalisés: Prolog III [Colmerauer 1990], PrologIV [Benhamou & Touraïvane 1995], CHIP [Van Hentenryck 1989], CLP(X) [Jaffar & Lassez 1987], SICStus-Prolog [SICS 1998], CLP(FD) [Diaz 1998]. Neanmoins les techniques de maintien de cohérence sont indissociables de l'interpréteur de règles et par conséquent de l'énumération des domaines de variables.

L'intégration de la notion de contrainte dans les langages de programmation par objets permet essentiellement d'atteindre deux objectifs. D'une part, un objectif de génie logiciel :

en représentant les contraintes par des classes, on peut organiser celles-ci sous la forme d'une bibliothèque, et ainsi séparer les algorithmes d'énumération des différentes techniques de cohérence. En outre, cette manière de procéder permet d'associer des méthodes de filtrage à des contraintes particulières, comme nous le verrons dans la section suivante décrivant le système BackTalk. Il existe plusieurs systèmes basés sur ce principe. Certains systèmes utilisent des langages objets «propriétaires » comme Eclair [Laburthe 1998]. Pour le langage C++, les principaux systèmes sont CHIP et ILOG-Solver [Puget 1992, 1994], [Puget & Leconte 1995] et sont considérés comme des systèmes de référence. Le système LAURE [Caseau 1991a, Caseau & Perron 1991], exploite des techniques des bases de données déductives pour optimiser la propagation des contraintes dans le langage objet sous jacent : les contraintes sont reliées aux objets *via* des démons, ce qui permet d'activer les contraintes d'intégrité uniquement concernées par la requête [Caseau 1989] et de contrôler de manière transparente les modifications de la base.

D'autre part, l'intégration des contraintes dans un langage à objet permet d'enrichir le langage même de spécification des contraintes, comme montré dans [Roy et al. 1997]. L'intégration des contraintes et des objets permet de définir des CSP dans lesquels les domaines sont des ensembles d'objets arbitraires, appartenant à des ontologies spécialisées. Un des intérêts de cette approche réside dans le fait de pouvoir exprimer les contraintes au moyen des relations entre les entités du domaine, plutot que par des relations arithmétiques ou ensemblistes de base. Par opposition aux variables entières dont les domaines sont des ensembles d'entiers, nous appelons «variable objet », une variable ayant un tel domaine, comme des objets musicaux complexes [Pachet & Roy 1995], les tâches de problèmes d'ordonnancement [Paltrinieri 1994], [Caseau et al. 1993, Caseau & Laburthe 1996a].

La double nature des contraintes est unifiée par la notion de classe de contraintes. Une classe de contraintes s'instancie en une contrainte manipulable qui peut simuler une contrainte symbolique. Chaque classe de contraintes implémente un ensemble de procédures spécifiques de filtrage qui maintiennent la cohérence des domaines vis à vis de la contrainte.

L'encapsulation des méthodes de propagation au sein d'une classe offre un double avantage. D'une part chaque contrainte peut stocker des données privées nécessaires à son fonctionnement ou dans le but d'améliorer sa propagation. Par exemple, la contrainte de cardinalité conserve la liste des variables encore candidates [Laburthe 1998], une égalité sur une combinaison linéaire conserve les valeurs minimum et maximum des sommes à coefficients de même signe [Laurière 1976]. D'autre part chaque contrainte est propagée selon son propre niveau de cohérence, ce qui rend transparent la conception de niveau de cohérence hybride au sein d'un réseau de contraintes. D'une manière générale une propagation est organisée en trois phases principales *PréCohérence*, *Cohérence* et *PostCohérence* : la *Précohérence* teste *a priori* si la propagation doit être exécutée, la *Cohérence* effectue la propagation et la *PostCohérence* analyse *a posteriori* l'effet de la propagation. Ainsi il est possible de passer de la cohérence d'arc, à la cohérence de borne si le domaine est jugé trop fractionné après propagation. Par ailleurs, une contrainte étant filtrée plusieurs fois au cours du maintien de cohérence, il est possible d'éviter des calculs inutiles en mémorisant pour quel changement (minimum, maximum) a eu lieu la précédente propagation.

Dans le cadre de notre étude, nous avons choisi d'utiliser le système BackTalk [Roy 1998] qui reprend l'architecture de systèmes comme ILOG-Solver ou CHIP, avec des fonctionnalités et des performances similaires, tout en étant facilement modifiable et accessible.

2.2.2 Description de BackTalk

Le système BackTalk est une bibliothèque de classes Smalltalk implémentant un système complet de satisfaction de contraintes à domaines finis. BackTalk a été réalisé par Pierre Roy pendant sa thèse [Roy 1998][2].

BackTalk est doté d'une efficacité comparable à celle des systèmes existants. Il se caractérise par trois critères : 1) BackTalk est un *framework* extensible par héritage (création de sous-classes) des classes existantes ; 2) son intégration dans le langage hôte permet de poser des contraintes générales sur des structures complexes, qui sont parfois plus performantes que des contraintes numériques [Roy et al. 1997] ; 3) BackTalk est fondé sur un mécanisme de démons. Plus précisément BackTalk intègre outre une bibliothèque de contraintes, un ensemble de classes d'algorithmes d'énumération et d'heuristiques, suivant en cela les *pattern* de conception *Strategy* et *Template Method*. L'algorithme de résolution de BackTalk est fondé sur un schéma abstrait d'algorithmes qui permet d'instancier tous les types d'algorithmes fondés sur le filtrage de contrainte et la recherche arborescente (thèse de Roy page 101).

D'après [Roy 1998], dans le cas général, les procédures de filtrage de BackTalk sont indépendantes du domaine d'application. Cette bonne propriété ne suffit pas à garantir systématiquement l'efficacité du filtrage en terme de réduction de domaine, comme nous l'illustrons dans le chapitre I-2.3. Il faut donc chercher la raison de ce phénomène dans la manière de traiter les contraintes, qui dépend encore de la taille des domaines.

Vie d'une variable et filtrage de contrainte : les démons

Un démon est défini formellement comme l'association d'un type d'événement du système à une liste d'actions ayant une propriété en commun. Le mécanisme de démons est donc adapté à l'implémentation des bibliothèques de contraintes. La vie d'une variable de BackTalk est décrite à travers cinq événements typés identifiant les changements de son domaine (C.f. Tableau 1).

Evénements Event(*, v)	Description
Value	Le domaine de v est réduit à un singleton.
Remove (val)	La valeur val n'appartient plus au domaine de v.
NewDomain	Le domaine de v est réinitialisé.
Min (Si le domaine de v est ordonné)	La valeur minimale est augmentée.
Max (Si le domaine de v est ordonné)	La valeur maximale est diminuée.

Tableau 1: définition des événements rythmant la vie d'une variable.

Ces événements sont traités automatiquement par des démons attachés à la variable. Les actions des démons visent à réduire l'espace de recherche du problème de CSP à résoudre [Roy et al. 1998].

[2] BackTalk est disponible sur le site ftp du LIP6 ainsi qu'à http://www-poleia.lip6.fr/~liret/index.html.

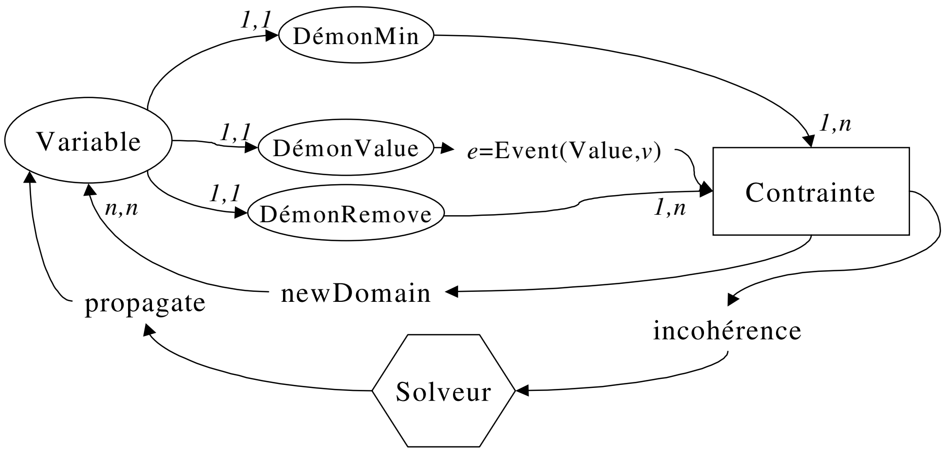


Figure 1[3]: liaison réactive des variables aux contraintes

La procédure de filtrage d'une contrainte *C* est donc décomposée en autant de filtrages spécifiques qu'il y a de types d'événement auxquels C sait réagir. Plus précisément, notons *filtrage*(*C,t*), la procédure de filtrage de *C* si *t* est un événement ; définissons la procédure de filtrage vide, pour le cas où la contrainte ne peut pas être filtrée pour un événement ; définissons T_{Event} comme l'ensemble des types d'événements possibles, on a alors *filtrage*(*C*) = $\coprod_{t \in T_{Event}} filtrage(C,t)$. Par exemple, la contrainte générale *Y = f(X)* réagit quand une valeur est supprimée du domaine de X ou Y, ou quand X ou Y est instantiée ; son filtrage est définie par les actions de démons suivantes :

```
Vx = Value(X) → Y setValue(f(Vx)).
Vy = Value(Y) → X setRemoveValues({i / f(i) ≠ Vy}).
Remove(X,v) →Y setRemoveValue(f(v,i)).
Remove(Y,v) → X setRemoveValues({i / f(i) = v}).
```

Si f est injective la précédente règle se simplifie car l'ensemble à supprimer ne contient qu'une valeur.

`Remove(Y,v) → X setRemoveValue(i)` tel que `f(i) = v, i ∈ dom(X).`

Dans le cas où *f* est une fonction monotone et les domaines sont ordonnés, la contrainte réagit à `MinEvent` et à *MaxEvent*. Pour *MaxEvent*, les règles de filtrage sont données par :

```
m = Max(Y) → X setRemoveValues({i / f(i) > m}).
```

Si f est croissante sur le domaine de X, la règle se simplifie en

```
m = Max(Y) → X setMin(f(i)).
```

Et `m = Max(X) → Y setMax(f(i)).`

Cette division du rôle de la contrainte selon le contexte facilite la définition de nouvelle contrainte. De plus une contrainte est manipulable et un démon redéfinissable. En particulier, on peut définir de nouveau type de démon et contrôler le filtrage d'une contrainte durant la résolution.

Les contraintes en BackTalk

Les contraintes sont maintenues cohérentes essentiellement par cohérence de borne, sauf lorsqu'elles possèdent au plus deux variables libres. Pour les contraintes arithmétiques, le

[3] Les cardinalités des liens sont indiquées en italique.

filtrage de BackTalk s'exécute au pire en *O(dn)*, où *n* est le nombre de variables, et *d* la taille du plus large domaine. La complexité est *O(n)* pour les contraintes arithmétiques non binaires et en temps constant pour les contraintes binaires.

Les contraintes arithmétiques sont décomposées en contraintes primitives comme dans [Alefeld & Herzberger 1983, Hickey et al. 1998] (Figure 2) et en combinaisons linéaires. Les méthodes de filtrage définies dans chaque classe de contraintes primitives correspondent aux opérateurs de réduction d'intervalle de [Van Hentenryck et al. 1997]. La combinaison linéaire correspond à un opérateur d'intervalle.

$$
\begin{tikzcd}
 & T1 - T2 + 3*T3 \text{ @= } 34 \arrow[dl] \arrow[d] \arrow[dr] & \\
f(z) \text{ @= } T1 & X*T4 \text{ @= } T2 \arrow[d] & g(z,u) \text{ @= } T3 \\
 & Y^2 \text{ @= } T4 &
\end{tikzcd}
$$

Figure 2: décomposition d'une contrainte non-linéaire, f et g, deux fonctions numériques.

De plus la représentation d'un domaine comme une union d'intervalles s'inspire des Intervalle-CSP [Hyvönen 1992]. De manière générale, un problème de contraintes à la BackTalk combine des nombres et des objets composites ; il contient donc deux sous-problèmes, un Intervalle-CSP et un CSP objet. Dans la partie Intervalle-CSP, les contraintes sont décomposées et filtrées par les algorithmes de cohérence partielle. BackTalk applique sur le CSP objet un algorithme optimisé et complet (décrit dans la thèse de Roy) qui réalise l'arc-cohérence en ne supposant pas les domaines ordonnés.

BackTalk intègre deux classes de contraintes générales. Une contrainte générale est simplement déclarée en écrivant l'expression de l'envoi de message à une variable objet représentant l'objet receveur. BackTalk distingue les contraintes fonctionnelles et les contraintes prédicatives. Dans une contrainte fonctionnelle `(BTPerformCt)`, la relation est de la forme *Sujet* = *Verbe*(C*omplément*), où le sujet et le complément sont des variables objets et le verbe un symbole de méthode dans la classe des valeurs du sujet. Dans une contrainte prédicative `(BTBlockCt)`, la relation est un prédicat représenté par une fermeture[4].

BTBlockCt

Cette contrainte répond au besoin de pouvoir exprimer rapidement une contrainte compliquée compatible avec les méthodes de filtrage prédéfinies. Elle intervient par exemple lors de la recherche des contraintes d'une situation dont on connaît seulement l'état. C'est le cas lorsqu'un solveur CSP est inséré au sein d'une application existante. Par exemple, dans un jeu à deux joueurs interactif, on peut utiliser un CSP pour faire jouer l'ordinateur contre l'humain ; les contraintes sont les conditions de placement correct d'une pièce. Dans le cas du Morpion, un pion doit être à côté d'une case occupée : BTBlockCt(Pion1, λ*pos*.[∃ *p* ∈ voisins(*pos*), estOccupé(damier[*p*])]). Au L-Game [De Bono 1994], la pièce en 'L' (pL1 ou pL2) doit bouger et un seul des pions (p1 ou p2) peut bouger ; les pièces ne doivent pas se chevaucher. (C.f. Figure 3).

```
BTBlockCt(λp1.p2.pL1.pL2.[p1 inFrontOf: p2,
```

[4] *i.e.* un lambda-terme plus un environnement, quelconque

"*p1 apparaît avant p2 en regardant le damier de droite à gauche et de haut en bas*"

```
and [(pL1 intersect: p1) or ((pL1 intersect: p2) or (pL1 intersect: pL2))] not,
```
"*pL1 ne doit pas chevaucher une autre pièce*"

```
and [(GS neutralP1 = p1) or (GS neutralP1 = p2)
or ((GS neutralP2 = p1) or (GS neutralP2 = p2))],
```
"*un seul pion doit bouger*"

```
and [((p1 intersect: pL1) or (p1 intersect: pL2))
or ((p2 intersect: pL1) or (p2 intersect: pL2))] not].
```
"*p1 doit être une case libre*"

Figure 3 : Contraintes de déplacement de pL1 et p1 en BackTalk.GS est l'état du jeu du L-Game.

BTPerformCt

Cette contrainte permet de définir une variable comme le résultat de l'application d'une fonction d'arité au plus deux, sur d'autres variables. Par exemple, l'expression affecte Y=f(X, Z) avec une variable de domaine $\{i \text{ in } f(dom(x), dom(z)) \mid \exists\, x_i , \exists\, z_i, i = f(x_i,z_i)\}$. La contrainte est postée simplement en écrivant l'expression dans le langage hôte, c'est à dire l'envoi d'un message de sélecteur #f à l'objet X avec pour argument Y.: `(y := (x btperform: #f arg: z) asVariable)`. Toute instance de `BTPerformCt` est une contrainte générale Y = f(X,Z) ou Y = f(X), où X et Y sont des variables, Z une variable ou une valeur et #f le nom d'un opérateur d'arité au plus deux sur tout objet du domaine de X et du domaine ou de la valeur de Z. Dans une approche objet, une fonction sur un ensemble D est représentée par une méthode définie dans les classes des éléments de D. Nous avons étendu BackTalk avec la contrainte Y = f(X,Z) où Z est une variable, BackTalk ne possédant au départ que le cas où Y est une valeur[5]. Par exemple, la relation géométrique "`D1 est parallèle à D2`" est spécifiée par "D2 = parallèle(D1)", soit en BackTalk `(BtPerformCt antécédent: D1 selecteur: #parallèleA image: D2)` ou `((D1 btperform: #parallèleA) @= D2)`. Pour dire que la parallèle à D1 doit passer par le point A, on écrit les contraintes :

```
D2 = parallèleA(D1).
BlockCt(λA.D2.[A belongsTo: D2]).
```

2.3 Exploitation de la redondance

L'expoitation de redondances dans un problème combinatoire consiste à inférer une ou plusieurs contraintes redondantes, c'est à dire des contraintes telle que leur introduction dans le problème de change pas l'ensemble des solutions. Une contrainte redondante revêt plusieurs interprétations selon les domaines d'application. Traditionnellement les méthodes de recherche opérationnelle cherchent à la supprimer en considérent qu'elle complexifie le problème. C'est également le cas dans les processus de modélisation automatique de problèmes d'optimisation, qui génèrent inévitablement des informations redondantes [Nareyek 2000]. Cependant toutes les contraintes redondantes ne sont pas à supprimer, car dès qu'une contrainte redondante est retirée, les autres contraintes peuvent ne plus être redondantes. Il suffit de considérer le problème de programmation linéaire suivant {x 0 ; y

[5] L'expression f(X) est représentée par la classe `BTPerform` tandis que f(X,*a*), où *a* est une constante est représentée par la sous-classe `BTPerformArg`. `BTPerformArgVar` définit f(x,y).

0 ; x – y = 0} : chaque contrainte d'inégalité est redondante mais elles ne peuvent être supprimées simultanément car l'une est déduite de l'autre.

Notre proposition repose sur la constatation expérimentale qu'il existe des redondances « utiles » pour améliorer la résolution d'un problème. Nous interprétons donc une contrainte redondante comme une contrainte dont il faut évaluer l'utilité. Avec cette interprétation plus souple, il est possible de distinguer les contraintes « positives » et les contraintes « négatives ». Cela permet d'adapter le modèle du problème de satisfaction de contraintes, comme dans le cas de satisfaction de contraintes dynamiques [Sabin & Freuder 1996, 1998].

Si les techniques de CSP sont efficaces dans de nombreux domaines, il existe des situations où les techniques de CSP sont inadaptées alors que l'introduction de contraintes redondantes permet de trouver un résultat de manière plus élégante. Cette section en présente quelques-unes (voir aussi [Liret 1997]). Par exemple l'absence de solution nécessite souvent une exploration exhaustive de l'espace recherche en CSP. Pourtant l'ajout de contraintes implicites aboutit sans retour-arrière supplémentaire, à un problème incohérent qui est équivalent au problème initial. Il nous semble important de pouvoir détecter une incohérence sans faire de choix supplémentaire par rapport à la méthode de recherche, car cette situation apparaît en cours de résolution lorsque l'algorithme explore une mauvaise branche de l'arbre de recherche ; cela permet de séparer les choix propres à la résolution et ceux que l'on pourrait éviter par un raisonnement abstrait.

La réécriture, largement utilisé comme approche symbolique de résolution de contraintes [Comon et al. 1999], apparait comme un moyen naturel de déduire automatiquement des contraintes redondantes. Dans ce contexte, les situations peuvent survenir en cours de résolution ; la mise en œuvre d'une telle déduction nécessite la détection automatique de situations favorables.

2.3.1 Détection d'incohérence

Exemple 1 : propagation de contraintes d'intervalle

Considérons l'ensemble de contraintes {(1) X = Y ; (2) Z = Y ; (3) X ≠ Z}. La transitivité de l'égalité implique X = Z et l'opposition des expressions booléennes impliquent la valeur booléenne FAUX. Seule une cohérence globale sur au moins trois variables permet de détecter l'incohérence sans énumération.

Exemple 2 : exploration du domaine

Le problème *P* suivant est tiré de [Horn 1992, Hyvönen 1992] : X, Y, T, Z sont quatre variables réelles dans [–100..100] contraintes par X+Y = T et Y+T = Z. On suppose X instantié avec 1 et Z avec 11. Le problème n'a alors qu'une solution T=6 et Y=5. Or une cohérence de bornes réduit les domaines à *dom*(Y)=[-88..100] et *dom*(T)=[-89..99]. En normalisant les contraintes, la contrainte (1) est équivalente à Y = T-1 et la contrainte (2) à Y+T = 11 ; en substituant Y dans la contrainte (2), on obtient T=6.

2.3.2 Exploitation de la structure des contraintes

Exemple 3 : arité des contraintes

Le problème {XA * X = BXC, [0..9]} contient intuitivement une seule contrainte [(10.X + A) * X = (100.B + 10.X + C)]. En BackTalk, il est modélisé par {[10.X + A = T1] ; [T1 *

X = T2] ; [T2 = 100.B + 10.X + C]}. BackTalk obtient la première solution en 5 retour-arrières et 7 choix et toutes les solutions au minimum au bout de 22 retour-arrières et 22 choix, avec des heuristiques standards fixes pendant la résolution. A partir des équivalences $x*x \Leftrightarrow x^2$ et $(x+y)*z \Leftrightarrow (x*z) + (y*z)$, le problème peut être réécrit en $\{[X^2 = T_1]$; $[A*X = T_2]$; $[T_2 + 10.T_1 = (100.B + 10.X + C)]\}$. Avec la même heuristique, on obtient alors la première solution en 1 retour-arrière et 3 choix et toutes les solutions en 9 retour-arrières et 9 choix. L'introduction de la contrainte $X^2 = T_1$ a amélioré la réduction du domaine de X (C.f. détails au chapitre II). De même, dans le problème $\{Y = X - X, X \in [-3..3]\}$, le domaine de Y est [-6..6] alors que la variation réelle de Y est [0..0].

Exemple 4 : raisonnement sur la parité

Soit l'égalité $4.M + 3.N^2 = 34$, où M et N sont des variables entières [Pitrat 1995]. Un simple raisonnement sur les parités des termes montre que cette égalité n'a pas de solution. Puisque 34.et 4.M sont pairs, nécessairement $3.N^2$ et N^2 sont pairs ; si N^2 est pair alors N^2 doit être multiple de 4. On se retrouve dans la situation où $4.M + 3.N^2$ est multiple de 4 et 34 ne l'est pas. L'interprétation abstraite des nombres par leur parité permet d'appliquer les règles classiques de parité sur N et de conclure.

2.3.3 Contraintes implicites

Exemple 5 : définition en intention d'ensembles en Alice

La contrainte d'appartenance à un ensemble de valeurs défini en intension ne peut être filtrée tant que l'ensemble ne peut pas être déterminé. Le système Alice traite la contrainte en réécrivant la définition de l'ensemble : $\{x_i \in V / R(x_i)\}$, R étant un prédicat unaire, est réécrit en supprimant les x_j ne possédant pas au moins une valeur telle que $R(x_j)$ est vraie.

Exemple 6 : L'expérience de [Davis 1988]

E. Davis montre que les techniques de cohérence sur des contraintes d'intervalles peuvent être améliorées en normalisant les contraintes et en déduisant des variables «imaginaires » sur des contraintes implicites. Par exemple, l'expression (X+1)/X où $X \in V$ et $dom(X) = [1..2]$ est déjà borne-cohérente, tandis que si l'on transforme l'expression en 1+1/X, la cohérence de bornes réduit le domaine de X à [1,5..2]. E. Davis propose un système de contraintes redondantes sur des variables intermédiaires dépendant des variables initiales du problème. Les variables intermédiaires représentent des quantités numériques calculables, par exemple la différence temporelle D_{ij} entre deux dates, $D_{ij} = X_i - X_j$. Ces variables sont contraintes implicitement par le fait que trois dates X_i, X_j, X_k vérifient toujours la contrainte $D_{ij} + D_{jk} = D_{ik}$. Plus généralement pour un CSP à n variables $V_1, V_2, V_3, \ldots, V_n$, on définit les variables $D_{V1,V2}$, D_{V2V3}, D_{V1V3}, $S_{V1,V2}$,... où $D_{ij} = i - j$ et $S_{ij} = i + j$.

Exemple 7 : transformation de contrainte non linéaire

La factorisation de la contrainte $0 = X^2 - Y^2$, $X \in V$, $Y \in V$ produit la contrainte $S_{xy}.D_{xy} = 0$, c'est à dire $(X+Y)*(X-Y) = 0$. Cette contrainte est intéressante car si X et Y sont strictement positives, le domaine de D_{xy} est réduit à {0}. Cela entraîne l'ajout de la contrainte X-Y=0, normalisée ensuite en X=Y. Dans le problème suivant, l'utilisation de la factorisation est intéressante pour la détection d'incohérence.

```
V={x,y,z}. dom(x)=[1..10]. dom(y)=[1..10]. dom(z)=[-10..10]
Contraintes :
(x*x) - (y*y) = 0.
" contraintes additionnelles "
```

```
C1 : x − (3*y) + (5*z) = 0.
C2 : x ≠ y
```

La contrainte implicite `(x+y)*(x−y)=0` étant une contrainte non linéaire, elle est décomposée en {T_1 in [2..20] ; T_2 in [–9..9] ; x+y = T_1 ; x-y = T_2 ; T_1*T_2 =0}. Le filtrage de T_1*T_2 =0 entraîne l'instantiation de T_2 avec 0, ce qui provoque la réécriture de x-y =T_2 en x=y. Le système applique alors la règle suivante : U=V, U≠V → FAUX, et conclut à l'absence de solution sans avoir fait de choix. Dans une résolution où la contrainte implicite n'est pas introduite, aucune propagation n'est faite et le résultat est obtenu au bout du parcours de l'espace de recherche.

Contrainte additionnelle	borne-Cohérence (BC)	Factorisation + Composition + BC + Réécriture	BC + Composition + Réécriture
C1	9 bt	0 bt 0 choix 0.05ms	9 bt 9 choix 0.02ms
C2	4 bt	0 bt 1 choix 0.024ms	4 bt 5 choix 0.023ms

Tableau 2 : incohérence ligne 1 et première solution ligne 2[6]

On constate que la présence ou non dans le problème d'autres contraintes contenant les variables de la nouvelle contrainte redondante est ici un critère de déclenchement de raisonnement symbolique. En présence de C1 et en l'absence de contrainte redondante, l'incohérence n'est pas détectée. Lorsque l'on active la factorisation, l'ajout de la contrainte redondante provoque la composition soit d'égalités linéaires en présence de C2 soit de contraintes binaires en présence de C1. Le simple fait d'introduire la contrainte implicite suffit pour déduire les valeurs des variables restantes sans choix supplémentaire. On remarque que l'introduction d'une étape de factorisation n'implique pas forcement de surcoût en temps. Ainsi à la ligne 2, le surcoût dû à la création et l'ajout d'une nouvelle contrainte en début de résolution, est compensé par un gain en réduction de domaine qui accélère la résolution.

3 Raisonnement symbolique sur les contraintes

Au sein du paradigme de la satisfaction de contraintes, la manière canonique d'améliorer l'efficacité de la cohérence locale, consiste à passer abruptement à des techniques de cohérence de chemin, c'est à dire à traiter plusieurs contraintes simultanément afin de réduire davantage les domaines. Cependant les algorithmes standards de cohérence de chemin traitent systématiquement *tous* les ensembles de contraintes indifféremment. Une manière d'éviter ce saut quantitatif en complexité est de définir de contraintes globales possédant des algorithmes spécifiques. Par ailleurs, la section précédente a exemplifié sur l'intérêt d'introduire des contraintes redondantes pour améliorer la réduction de l'espace de recherche, dans certaines situations. L'approche que nous proposons consiste à déduire ces contraintes en exploitant la représentation symbolique des contraintes existant dans le problème. L'automatisation de ces déductions relève naturellement d'un mécanisme de réécriture de contraintes [Roy 1996], mécanisme que nous avons intégré au système BackTalk, et que nous appelons *RCS.* Nous présentons les résultats principaux de ces techniques dans les sections suivantes.

[6] Résultats avec BackTalk en nombre de backtrack (bt) et millisecondes.

3.1 Techniques basées sur la réécriture

La réécriture est un paradigme général d'expression de calcul dans différentes logiques computationnelles. Les calculs prennent la forme de règles dans un syntaxe donnée. Dans une logique équationnelle, celles-ci résultent de l'interprétation d'équations entre termes. Dans une logique de satisfaction de contraintes, une règle de réécriture peut être interprétée de deux manières, soit comme une transformation syntaxique, soit comme une inférence logique d'une nouvelle formule. La réécriture a été largement utilisée, au point que [Meseguer 2000] définit une « logique de réécriture », comme une interprétation de la réécriture pour le contrôle de systèmes concurrents et l'inférence de connaissances symboliques. Réécrire un terme consiste à le remplacer par un terme équivalent, en vertu des lois de l'algèbre du terme. En particulier, la réécriture peut être utilisée pour transformer l'ensemble des contraintes d'un problème tout en garantissant la conservation de ses solutions. L'application automatique de ce mécanisme produit un résultat unique sous certaines conditions sur la base de règles de réécriture (terminaison, convergence). Sous ces conditions, l'intégration de la réécriture au sein d'un système de satisfaction de contraintes permet de réaliser une transformation syntaxique déterministe d'un problème de CSP avant ou pendant sa résolution. Dans la section suivante, nous ne présentons que les bases de la théorie de la réécriture. [Jounannaud & Lescanne1986], [Dershowitz & Jouannaud 1990], [Bachmair 1991] peuvent être consultés pour davantage de détails.

3.1.1 Réécriture

L'entité de base dans la réécriture est le terme, qui est un arbre dont les noeuds sont étiquetés par des symboles d'opérations et les feuilles par des symboles de constantes ou de variables. La racine de l'arbre s'appelle aussi le symbole de tête et les fils d'un noeud les sous-termes. Un noeud possède autant de fils que l'arité de l'opérateur qui l'étiquette.

A l'origine la réécriture est une mise en œuvre informatique de la preuve de théorèmes équationnels à partir d'axiomes (équations). Celle-ci consiste à utiliser ces équations pour remplacer les termes du théorème par des égaux. Le théorème est vrai si l'on peut obtenir à partir d'un des membres de l'équation, l'autre membre. Devant l'indéterminisme de ce procédé, lié au double sens de l'équation et au choix des bons axiomes, on oriente les équations pour obtenir un système de règles de réécriture. Réécrire un terme t consiste à choisir un sous-terme t' de t et une règle de réécriture $g\text{->}d$, à vérifier que g filtre t' (on dit que t' est instance de g), et à remplacer t' dans t par l'instance correspondante de d. On appelle pas de réécriture le fait d'appliquer une règle valide sur un terme. Une séquence de réécriture est une succession de pas $E_0 \rightarrow E_1 \rightarrow E_2 \rightarrow \ldots \rightarrow E_n$, où pour tout i, E_i est réécrite en E_{i+1}.

La réécriture a rapidement trouvé des applications directes dans le développement de systèmes de calcul formel tels que Macsyma [MathLab 1983] ou Maple. Plus récemment elle a été appliquée dans la conception de systèmes d'aide à la preuve de programme, un programme étant représenté par des spécifications formelles [Bakhouch 1994, Hoffmann and al. 1985], la preuve automatique de théorème en logique du premier ordre, l'écriture d'interprètes de langages logiques. Citons notamment l'atelier B de Digilog développé avec le CNAM et le projet COQ (INRIA) [Cornes and al. 1995]. Les systèmes de réécriture ont été par ailleurs utilisés pour supporter des langages de contraintes symboliques, dans le contexte d'applications graphiques, comme par exemple Delta-C [Roy 1996], Equate [Wilk 1991], Siri [Horn 1992].

Ambiguïté de vocabulaire

Signalons une ambiguïté dans le vocabulaire entre l'approche CSP et l'approche symbolique. Le terme «filtrage » référence une catégorie d'algorithmes de maintien de cohérence dans un problème de satisfaction de contraintes ; en programmation logique et en réécriture, le terme de filtrage (t filtre u) représente l'opération qui consiste, à trouver des substitutions de termes pour les variables de t qui rendent t (où les variables ont été substituées) identique à u. Dans la suite de ce document nous choisissons de conserver le terme de "filtrage" pour le traitement des contraintes ; nous utiliserons le terme "sélection de contraintres" ou "substitution de motif", lorsque le texte se rapportera à un traitement symbolique, comme la réécriture de contraintes.

Propriétés

Deux propriétés souhaitées dans un système de réécriture sont la *terminaison* et la *confluence* :

- Un système termine si pour toute expression E, toute séquence de réécriture est finie.
- Un système est confluent si pour toute expression E, toutes les séquences de réécriture finies aboutissent à la même forme. La confluence exprime le fait que le résultat d'un système de réécriture est indépendant du choix des règles dans le calcul.

Un système est convergent s'il possède les propriétés de confluence et terminaison. Lorsque le système est convergent, on dit que la forme irréductible et unique obtenue est la forme normale du terme initial. Lorsque le système est confluent mais que le calcul de réécriture n'est pas effectué jusqu'à saturation, le résultat est une forme régulière canonique.

Dans le cas d'opération commutative (axiome $x^\circ\ y = y^\circ\ x$) comme dans le cas de contraintes générales pouvant s'exprimer de plusieurs manières équivalentes, il n'est pas toujours possible d'orienter les équations. On définit alors des classes d'équivalence de termes telles que tous les termes d'une classe C puissent être prouvés égaux à une forme régulière dans le système d'équations du domaine. Le lemme du pentagone garantit que tous les termes de C ont des formes irréductibles appartenant à la même classe d'équivalence (éventuellement distincte de C).

Lemme du pentagone

> Soient S un système de réécriture et E un système d'équations. Si deux termes *t1* et *t2* peuvent être prouvés égaux dans E, alors leur forme normale par S peuvent aussi être prouvées égales dans E.

La confluence peut être vérifiée pendant la construction du système de réécriture, *via* un algorithme de complétion [Knuth & Bendix 1970]. Ce dernier, à partir d'un système d'équations et d'un ordre de réduction, oriente les équations pour obtenir des règles, de manière à ce que le terme cible soit toujours plus petit que le terme source. Le programme retourne en outre la liste des règles de réécriture qu'il faut rajouter à la base de règles en construction pour qu'elle soit confluente. La terminaison est assurée par un ordre de réduction sur les termes : s'il existe un ordre > tel que pour toute règle ($l \rightarrow r$) du système, on a «$l > r$ » alors le système possède la propriété de terminaison. Un ordre > sur un ensemble de termes T est un ordre de réduction s'il est 1) bien fondé (pas de chaîne infinie $t_1>t_2>\ldots$), 2) compatible avec la substitution ($s > t \rightarrow s' > t'$ si s' et t' sont des substitués de s et t), 3) compatible avec la structure du terme ($s > t \rightarrow$ pour tout terme U(v), Us > Ut si Us (resp. Ut) est le substitué de U où v est remplacé par s (resp. t)).

Plusieurs modèles d'ordres de réductions existent [Dershowitz & Jouannaud 1990], [Jouannaud & Lescanne 1986], en particulier l'ordre *rpo* (*Récursive Path Order*)..

L'ordre *rpo* est défini par :

> $s= f(s_1,..,s_m) >_{rpo} t= g(t_1,..t_n)$ si et seulement si
> il existe un s_i tel que $s_i >_{rpo} t$, ou
> $f >_{pred} g$ et pour tout f_j, $s >_{rpo} t_j$, ou
> $f = g$ et $(s_1,\ldots,s_m) >_{F\text{-}rpo} (t_1,\ldots,t_n)$.

$>_{F\text{-}rpo}$ est un ordre récursif sur l'ensemble des termes qui s'instancie en l'ordre lexicographique ou l'ordre multiset. $>_{pred}$ est un préordre de précédence sur les têtes des termes fonctionnels. L'ordre *rpo* est étendu avec pour $>_{pred}$ un ordre lexicographique ou un ordre multiset. L'extension lexicographique, *lpo* (*Lexicographic Path Order*) utilise un ordre de précédence fondé sur l'ordre lexicographique des variables et constantes

Algorithme de réécriture

Un algorithme de réécriture est la procédure qui permet de simplifier un terme. La Figure 4 donne le pseudo-code de la procédure principale d'un moteur de réécriture. Il prend en entrée un terme *t* du domaine et un système de réécriture S. Trois résultats peuvent être produits selon les propriétés de S (décrites plus loin) :

1) un terme *t'* syntaxiquement équivalent au terme *t*, plus simple que *t*, irréductible et unique. *t'* est la *forme normale* de *t*. Le système S est convergent.

2) un terme *t'* irréductible selon S mais pas unique, c'est à dire que l'algorithme donne un autre résultat si on change l'ordre des règles. Cela veut dire qu'il existe deux séquences dans S qui n'aboutissent pas au même résultat ; donc S termine mais n'est pas confluent.

3) l'algorithme ne s'arrête pas. Il existe une séquence infinie dans S donc S ne termine pas.

```
Reecrit(t, S)

    Local| exp2 exp1 |
    exp2 := reecritCycle(t, S).
    exp1 := nil.
    Tantque [not(exp2 == exp1)]     "rentre une fois au moins dans la boucle"
                exp2 := trieLex(exp2).
                exp1 := exp2.
                exp2 := reecritCycle(exp1, S)
    finTantque.
    retourne sortLex(exp1)
```

Figure 4 : algorithme itératif de normalisation par réécriture

La fonction `reecritCycle(unTerme, uneBase)` (C.f. pseudo-code donné à la Figure 5) choisit une règle dont la condition filtre `unTerme`, si celle-ci existe, l'applique sur le terme, sinon réécrit les fils du terme. Lorsque le système S termine, il existe un pas de l'algorithme où plus aucune règle ne peut s'appliquer ; l'expression courante est le résultat retourné. La fonction `trieLex(unTerme):unTerme` trie récursivement les sous-termes de son argument en utilisant *lpo*. Elle est nécessaire car les termes ont plusieurs formes syntaxiques équivalentes, par exemple *(-y)+x-z et x –y + (-z)*. On peut éviter de parcourir S à chaque fois en mémorisant les bases spécifiques d'une classe de termes ; la classe est définie par le connecteur principal. Cette optimisation a été implémentée dans le système AliceTalks (C.f. Annexe) [Liret et al. 1998].

```
ReecritCycle(t, S) "ne sature pas"

    Local| fils aChangé filsReecrits anExp2 |
    fils := sousTermes(t).
    Si fils estVide Alors retourne t.          "pas de règle"
    anExp2 := AppliqueRegles(t,S).
    Si not(anExp2 == nil) Alors [retourne anExp2]. "succès"
    aChangé := false.                          "réécrire les sous-termes"
    filsReecrits := OrderedCollection new(size = taille(fils)).
    PourTout f dans fils Faire
          Local| fprime |
          fprime := reecritCycle(f, S).
          Si not(f == fprime) Alors [aChangé := true].
          filsReecrits add(fprime).
    FinPourTout.
    Si not(aChangé) Alors retourne t.          "échec"
    Retourne newTerme(class=classe(t),sons=filsReecrits,
connecteur= tête(t))
```

La classe d'un littéral est `Constante` ou `Variable`.

Exemple 3

Déroulons l'algorithme avec le système convergent sur les sommes binaires (+,0) et des produits binaires (*,1) augmentée de la règle de réécriture E*E → E². L'expression X+(Y*Y)+0+(-X) est tout d'abord réordonnée en 0+X+(-X)+(Y*Y) selon l'ordre lexicographique «*constante < y < (-y)* » et ensuite normalisée en trois pas : réécriture en X+(-X)+(Y*Y) par 0+A → A, puis en Y*Y par A+(-A) → 0 et 0+A → A et en Y² , terme irréductible pour la base du groupe (*,1) étendu.

3.1.2 Réécriture de contraintes

La réécriture est un moyen de calcul formel qui peut être utilisé dans tout domaine où il est possible de définir une algèbre sur des expressions symboliques. En particulier, la réécriture peut s'appliquer aux contraintes : les contraintes arithmétiques étant vue comme des prédicats sur l'anneau (+,*) et les contraintes sur des classes d'objets étant vues comme des prédicats sur l'ensemble des opérations de la classe des objets.

La réécriture de contraintes nécessite de définir un ordre partiel de simplification sur les contraintes. Dans un langage à objets intégrant les contraintes, une contrainte peut être vue comme un terme fonctionnel dont l'entête est la classe de la contrainte ; de ce fait la comparaison de termes intra-classe ne pose pas de difficulté majeure. Pour comparer des contraintes de types différents, il est possible d'étendre l'ordre *rpo* avec un ordre de précédence basé sur les classes de contraintes.

Exemple : comparaison de contraintes de cardinalité sur des nombres

Le symbole de tête est #Cardinalité, les opérandes sont de deux types non comparables, un ensemble de variables et une expression booléenne. On se fixe un ordre fondé sur la complexité de l'expression numérique qui est un polynôme dans ce cas précis :

$\{=\} \succ_{\text{précédence}} \{>\}$ et pour t et g de même symbole de tête, t > g, si arity(t) > arity(g) ou si degré(t) > degré(g).

Alors #Cardinalité(S,x=4) > #Cardinalité(S, x^2>4) .

Historique

Un des buts des langages de programmation logique avec contraintes consiste à optimiser un programme logique en explicitant des contraintes au niveau des règles ou en inférant des contraintes. Par exemple, la règle `pair(Y) :- pair(X+Y), pair(X)` permet au concepteur de demander au système de C.L.P. si un programme implique qu'une des variables est paire. Il peut alors éventuellement rajouter la contrainte dans sa base de prédicats. Cependant la démarche d'inférence en chaînage-arrière des C.L.P fonctionne à l'inverse du raisonnement déductif dont la problématique est de découvrir des contraintes redondantes non connues *a priori* (on ne connaît que le moyen de les déduire). Le programmeur C.L.P. se demande «telle contrainte est-elle redondante avec le problème courant ? » tandis que chaque règle de RCS est fondée sur la question « que peut-on déduire d'un ensemble de contraintes S inclus dans le problème courant ? ».

Plusieurs systèmes ont alors proposé d'implanter un mécanisme de déduction de contraintes redondantes, comme le langage des CHRs (*Constraint Handling Rules*) [Frühwirth 1998] qui étend celui de ECLIPSe [Wallace et al. 1997] avec des règles de réécriture conditionnelle multi-prémisses, CLAIRE [Caseau & Laburthe 1996b] qui intègre des règles de propagation en chaînage avant, et le *framework* de contraintes concurrentes CC [Saraswat 1993] qui définit la notion d'implication de contraintes (*constraint entailment*).

Intérêt de la réécriture pour la résolution de problèmes combinatoires

En considérant une contrainte comme une expression symbolique, les langages logiques sous contraintes formalisent les règles d'inférence comme des règles de réécriture. Un problème de CSP est alors résolu lorsque toutes ses contraintes ont été réécrites dans une forme irréductible. Ainsi [Roy 1996] résout incrémentalement des CSP grâce à une Réécriture de Terme Augmentée au lieu de filtrage ; [Castro & Kirchner 1998],[Castro 1998] formalise les mécanismes de résolution de CSP (cohérence, énumération) sous forme de règles de réécriture dont il contrôle l'application par des «stratégies ELAN » [Kirchner 1993]. L'équivalence entre la contrainte initiale et la contrainte réécrite garantit la cohérence des domaines pendant la modification du problème.

Une contrainte est à la fois un ensemble de méthodes de filtrage et une expression symbolique à valeur booléenne. De même la réécriture de contrainte peut être utilisée dans deux buts : 1) simplifier l'expression de la contrainte ce qui peut avoir pour conséquence de changer l'algorithme de cohérence appliqué dans la résolution ; 2) changer le degré de cohérence de la contrainte en fonction du problème, des propriétés du domaine, de la contrainte elle-même. Si elle est utilisée en combinaison avec les techniques de CSP, la modification du degré de cohérence peut permettre de bloquer certaines contraintes tant qu'une variable n'est pas affectée, par exemple pour isoler un sous-ensemble de contraintes incohérent.

La réécriture apparaît comme un point de convergence naturel entre l'intelligence artificielle et de nombreux domaines computationnels. L'outil syntaxique qu'est la réécriture est utilisé pour transformer la structure d'un ensemble de contraintes sur des domaines réels pour améliorer la précision de la réduction de domaines [Benhamou & Granvilliers 1996], ou couper l'arbre de recherche. La généralite du paradigme de réécriture en fait un cadre adapté à l'implémentation formelle les algorithmes de satisfaction de contraintes [Kirchner 1993], à la normalisation des contraintes de manière à augmenter l'efficacité des méthodes algébriques [Beringer & De Backer 1995]. L'application systématique de la réécriture de contraintes permet enfin de conserver des expressions sous une forme régulière et ainsi de garantir le bon fonctionnement des manipulations algébriques de contraintes [Laurière 1986], [Liret et al 1998].

3.1.3 Constraint Handling Rules

Les règles manipulant des contraintes ou *CHR* ont été appliquées dans de nombreux domaines (contraintes temporelles [Frühwirth 1994], placement de station radio sur un réseau [Frühwirth & Brisset 1998], satisfaction incrémentale de contraintes [Wolf 1997]. Formellement une CHR [Frühwirth 1998],[LMU 1996] est une règle de réécriture conditionnelle, multi-prémisses, portant sur des contraintes, compilée en règles Prolog [Holzbaur & Frühwirth 1998]. Les actions d'une CH-règle ont pour but d'insérer de nouvelles contraintes dans le système de contraintes : la *simplification* (⇔) réécrit un ensemble de contraintes en un autre ensemble de contraintes plus simples ; la *propagation* (➔) ajoute des contraintes en conservant certaines contraintes des prémisses[7]. La confluence et la terminaison d'un programme CHR sont soumises à des conditions, énoncées dans [Abdennadher 1997, Frühwirth et al. 1996]. La syntaxe générale est la suivante :

`H1,...,Hl \ Hl+1,...,Hi ⇔ G1,...,Gj | Body`, où $G_1,\ldots, G_j$ sont des conditions «garde-fous » devant être vérifiées pour que la règle se déclenche. Les gardes-fous permettent de contrôler localement l'application d'une règle mais ce contrôle ne suffit pas pour exprimer des stratégies globales comme nous le verrons au chapitre I section 3. Clairement les CHRs offrent un cadre formel pour l'expression de raisonnement symbolique. Par exemple, `X+Y=Z, Z*Z=R ⇔ X*X+2*X*Y+Y*Y = R` et `X>Y, Y>X ⇔ Fail`.

Composition de AllDifférent(X_1,...,X_n) et X+Y=Z :

```
AllDifférent(L) , X+Y=Z ==>
%garde-fou
X in L, Z in L, 0 in dom(Y) |
%nouvelle contrainte
Y ≠ 0.
```

D'autres exemples sont fournis en annexe. Cependant les règles s'appliquent systématiquement sur toutes les contraintes candidates, ce qui peut provoquer des effets combinatoires difficiles à éviter. De plus le langage des CHRs ne spécifie pas comment manipuler les informations concernant les règles telles le moment de l'application, succès ou échec de l'application. Or le coût en temps et espace des techniques de manipulation symbolique nécessite de contrôler le moment de l'exécution ; en particulier, RCS interdit généralement d'appliquer les règles de manière systématique.

Différence avec notre approche

Notre travail diffère par le contexte de départ. Partant d'un langage à objets et d'une bibliothèque de classes de contraintes, nous avons rajouté un *framework* de règles de réécriture de contraintes où les règles sont des entités de première classe possédant une interface uniforme. La réification des règles permet de rendre leurs états accessibles à

---

[7] Formellement (C1,….,Cn) ➔ D1,…,Dm⇔ $\bigwedge_1^n Ci \Rightarrow (\bigwedge_1^n Ci) \wedge (\bigwedge_1^n Di)$).

d’autres objets. Cela nous a permis ensuite de rajouter un troisième niveau de classes implantant des invariants de contrôle, pour l'affinage du déclenchement des règles. L’utilisation d’un *framework* de satisfaction de contraintes permet d'accéder à des informations sur l'état du problème (nombre de retour-arrières ou de choix déjà faits), sur l'historique de la résolution (nombre de règles qui ont échoué, blocage d'une règle coûteuse ou inutile), informations permettant d'exprimer des situations globales à un niveau général. A notre connaissance, les CHRs n'offrent pas la possibilité de spécifier le moment de leur déclenchement.

3.2 Autres techniques

L’idée de RCS est de faire une interprétation abstraite sur l’ensemble des contraintes, au lieu d’énumérer les domaines. Cependant, dans certains cas, par exemple quand le problème est suffisamment petit, il est plus rapide et efficace de procéder à une énumération bête de l’espace de recherche que de combiner filtrage et énumération. Dans ce cas, seules les méthodes de filtrage interviennent. La compilation des contraintes sous forme de règles de propagation ou de procédure ad hoc permet alors d’optimiser le temps de calcul sur l’ensemble des contraintes. Dans le cas des bases de données, la compilation des contraintes est une étape nécessaire pour des raisons de complexité.

3.2.1 Compilation de contraintes

Lorsque les contraintes doivent être vérifiées très vite, il est avantageux de les compiler sous une forme rapide à exécuter. Par exemple les contraintes des bases de données ne sont pas explicitées mais représentées sous une forme spécifique proche de la logique d'ordre 1 ; le solveur de contraintes doit alors décider de l'une ou l'autre forme de représentation «réifiée » ou «compilée » en fonction de la difficulté de la contrainte (Rabbit) [Laurière 1996].

Contraintes dans les bases de données

Dans un S.I.G. (Système d'Information Géographique) des contraintes implicites existent dans la base, relatives au profil de l'utilisateur. Par exemple lorsque l'utilisateur dit «je veux aller de Paris à Nice en ne faisant pas plus d'un arrêt », il pense qu’il ne veut pas que son trajet total dure plus de vingt heures, ou qu’il ne veut pas rester plus de deux heures dans une même station située sur le trajet. [Mainguenaud 1996] montre que ces contraintes réduisent l'espace de recherche si elles sont ajoutées automatiquement à la requête. Afin d'éviter un surcoût dans le traitement de la nouvelle requête, les contraintes sont automatiquement compilées en règles attachées aux classes du schéma de la base.

Le système Rabbit

Le système Rabbit [Laurière 1996] propose de combiner deux techniques : (1) propagation de contraintes et énumération ; (2) génération et exécution d'un programme combinatoire spécifique. Le système observe l'évolution du problème et détecte quatre situations favorables au point (2) : 1) la taille du problème est importante, 2) la résolution est en phase de preuve d'optimalité, 3) il y a peu de contraintes mais elles sont difficiles à filtrer[8], 4) la propagation est inefficace - on sait que dans ce dernier cas, les choix ne l'améliorent pas car ils ne changent pas la forme des contraintes. Ce cas est difficile à détecter car il suppose de connaître la provenance d'une propagation ; Rabbit utilise donc un critère expérimental en comparant F_0(choix $_i$ - F_0(choix $_{i-1}$) avec un seuil, où $F_0 = \sum_{i \in V} poids(i) + \sum_{c \in K} complexité(c)$. Sur un problème de robot déplaçant des caisses, de type *Jobshop* avec une ressource ayant des contraintes de capacité [David 1992], Rabbit est plus rapide que le système Alice ; car sa stratégie détermine le moment de la compilation de contraintes de manière à ce que le temps de transformation soit compensé par le gain en temps de résolution.

La compilation et la simplification de contraintes impliquent un choix sur le mode de résolution car certaines formes de contraintes sont supprimées du problème et donc ignorées par l'algorithme de résolution. Par exemple dans Rabbit, tout produit nul de deux facteurs est transformé systématiquement en une contrainte disjonctive[9]. Ce choix de transformation provient de la définition du système de résolution : l'objectif étant d'obtenir une solution au moins, on énumère l'espace dès que la taille est restreinte. On préfère donc instancier successivement X et Y avec 0 plutôt que de conserver X*Y=0. Il n'est pas évident à notre sens que ce soit intéressant dans le cadre d'un système dynamique où l'on ajoute incrémentalement des contraintes. En effet cette réécriture interdit les combinaisons avec la contrainte produit, combinaison qui permettrait de simplifier des contraintes non linéaires.

3.2.2 Interprétation abstraite

Les applications intégrant des contraintes où les valeurs sont des objets composites sont difficiles à gérer si les domaines contiennent un grand nombre de valeurs [Caseau & Perron 1991]. L'interprétation abstraite [Cousot & Cousot 1997] offre un moyen d'éviter de manipuler ces ensembles, de remplacer plusieurs éléments par un seul objet, par exemple le signe des parties du domaine, un intervalle construit avec les bornes inférieure et supérieure, le type de l'objet. Formellement un système de raisonnement symbolique, comme celui que nous proposons au chapitre III, abstrait les contraintes implicites en les remplaçant par les règles de réécriture qui les engendrent ; puis il applique les propriétés déductives d'un domaine d'application, afin de déduire un ensemble de contraintes implicites. En effet l'introduction de toutes les contraintes implicites connues avant la résolution rendrait le problème plus complexe qu'à l'origine et plus difficile à résoudre car les contraintes implicites agiraient en parasite dans le maintien de cohérence.

Faire une interprétation abstraite offre plusieurs avantages : d'une part elle explicite des propriétés spécifiques du problème qui sont cachées dans les théories des domaines ou des contraintes ; elle exploite le sens des contraintes du problème. Par exemple la plate-forme de programmation d'algorithmes d'optimisation combinatoire, CLAIRE [Caseau & Laburthe 1996a] propose d'intégrer des règles de propagation portant sur les types d'objets du programme, telle que «POSITIF(X), NEGATIF(Y) → NEGATIF(X*Y) » ou «X ∈ S1, Y ∈ S2, S2 ⊆ S1 -> X ∈ S2 ». En se déclenchant automatiquement dès la modification d'une instance d'un type [Caseau 1989], ces règles permettent d'interpréter le comportement d'un programme afin d'en améliorer les calculs. [Caseau 1991a, 1991b] et [Caseau & Perron 1991] en montre l'intérêt pour optimiser les requêtes à des bases déductives.

---

[8] La complexité est calculée sur la même échelle qu'Alice.
[9] Règle reprise du système de réécriture d'Alice

Dans les exemples section 2.2, les règles de réécriture manipulent une interprétation abstraite dans le domaine d'application (la parité d'un nombre, la transitivité, l'opposition de relations,...). Le processus de résolution combinant satisfaction de contraintes et RCS met en œuvre plusieurs interprétations abstraites différentes selon l'état du problème.

3.2.3 La programmation linéaire revue par RCS

Les techniques de programmation linéaire (LP) existent depuis 1940, moment où G.B. Dantizg conçut la fameuse "méthode du Simplexe" pour résoudre des problèmes de planification linéaire de l' *US Air Force*. Ces techniques sont basées sur une manipulation algébrique du système linéaire, connue sous le nom d'"élimination de Gauss". Lorsque ces méthodes fonctionnent, elles fournissent une solution très rapide. L'intégration de la programmation linéaire et des theniques de satisfaction de contraintes a permis de combiner les avantages de techniques de résolution de contraintes arithmétiques et l'expressivité du paradigme CSP [Hooker et al. 1999]. Cependant, il existe plusieurs inconvénients aux techniques de LP :

1) Elles sont limitées aux expressions linéaires. Or de nombreux problèmes sont modélisés avec des contraintes non linéaires. De plus, grâce à la décomposition des contraintes en contraintes primitives [Hickey et al. 1998], il est possible de représenter une équation polynomiale par une contrainte linéaire sur des variables intermédiaires ; il est alors naturel de vouloir composer linéairement les contraintes non linéaires.

2) Elles fonctionnent souvent mal sur des systèmes non carrés, possédant plus de variables que de contraintes.

3) La méthode du Simplexe necéssite des valeurs d'initialisation compatibles, qui peuvent être difficiles à trouver.

4) Enfin, un sérieux inconvénient à la programmation linéaire réside dans le fait que ce sont des méthodes statiques.

Dans RCS, nous avons revisité la technique de Gauss-Simplexe et sa forme primitive, l'élimination de Gauss, en utilisant le paradigme de la réécriture de contraintes. Cette approche bénéficie de la décomposition des contraintes non linéaires.

La règle *linlin* produit l'ensemble des contraintes obtenues en éliminant une variable pivot au choix dans les variables communes à deux contraintes ; *linlin* conserve les contraintes d'origine ce qui permet d'itérer la composition linéaire plusieurs fois sur une même contrainte et de traiter des systèmes de taille non fixe. La règle *GaussSimplexe* remplace un système linéaire de taille fixe en un autre possédant autant de contraintes, par élimination succéssive de variables (pivots) dans chaque contrainte.

3.3 Le système Alice

Alice [Laurière 1976] a résolu en son temps des problèmes combinatoires réputés difficiles. Le système peut être décrit en cinq points :

1) Le système maintient une double représentation des contraintes, chacune dans une structure optimisée pour un type de traitement (propagation ou inférence) (C.f. Figure

5). Les contraintes dites simples (X=α, X≥α, X≠α, X=Y, X≠Y, X=αY+β, X≥Y), facilement propageables sont représentées par les arcs d'un graphe biparti dont les nœuds sont les variables et les valeurs, tandis que les contraintes complexes sont représentées par un arbre syntaxique car elles sont manipulées symboliquement. Maple et Macsyma utilisent également une représentation arborescente.

2) Les contraintes sont filtrées par un algorithme équivalent à AC-4 pour les contraintes binaires ; le filtrage des contraintes de différence est d'une efficacité comparable à l'arc-cohérence de [Régin 1994] grâce à leur regroupement en «cliques ».

Contraintes simples	Contraintes n-aires
↓	↓
propagation	Inférence
↓	↓
Arc-Graphe	Arbre-Base de règles de type CHR

Figure 5 : deux traitements orthogonaux et deux représentations en parallèle

3) Trois manipulations symboliques[10] sont mises en œuvre : a) retirer les contraintes trivialement satisfaites b) normaliser les expressions arithmétiques, logiques et fonctionnelles dès leur création et après chaque instanciation c) composer systématiquement les contraintes binaires avec les autres contraintes de manière à produire des contraintes contenant moins de variables. La composition s'applique aux égalités et inégalités et consiste à comparer les minimum et maximum des termes de l'expression puis à en déduire une nouvelle contrainte redondante. L'argument fort d'Alice réside dans le fait que ces trois manipulations symboliques combinées à une énumération suffisent pour résoudre des problèmes combinatoires avec très peu de retour-arrières. En particulier, Alice résolvait SEND+MORE=MONEY en 0 retour-arrière. Cette collaboration est possible grâce à une normalisation systématique des contraintes. Les situations de collaboration correspondent aux instantiations de variables survenant dans l'une ou l'autre des représentations (Figure 6).

4) Avec Alice, J.-L. Laurière défend la thèse que la généralité n'implique pas l'inefficacité. La collaboration entre la propagation de contrainte et la manipulation symbolique est générale. Elle s'applique avec succès à des domaines très différents (tournée de véhicules, coloriage, placement, cryptogramme, puzzle) grâce à un jeu d'heuristiques générales et à une procédure de choix dynamique de l'heuristique [Laurière 1979].

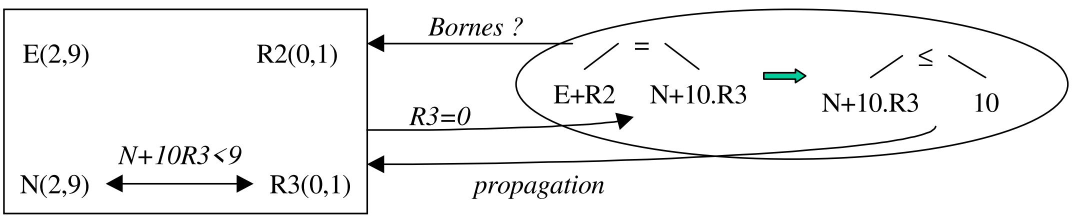


Figure 6: échange entre les deux représentations

De la contrainte [E+R2 = N+10.R3] où *dom*(E) = 2..9, *dom*(R2) = 0..1, *dom*(R3) = 0..1, *dom*(N) = 2..9, Alice déduit que [N+10.R3 ≤ 10] car max(E+R2) = 10, 10 ≥

[10] J.-L.Laurière introduit pour les nommer le terme "raisonnement formel"

max(N+10.R3). Le graphe prend alors en charge cette nouvelle contrainte et déduit que R3=0 ; il signale alors à l'ensemble des contraintes de simplifier celles portant sur R3.

3.3.1 Travaux sur Alice

J.Pitrat a reproduit Alice en codant dans son système à base de règles Maciste les connaissances qui permettent de construire un programme de résolution de problèmes combinatoires entier. Le système s'appelle Malice, *i.e. méta-Alice*.

[Liret et al. 1998] propose une version augmentée du système de réécriture d'Alice pour les expressions arithmétiques et logiques, augmentée de termes fonctionnels unaires et en prouve la convergence. La confluence a été montré en appliquant la procédure de complétion standard de [Knuth & Bendix 1970], disponible dans les contributions de ELAN[11]. Pour garantir que système termine nous avons utilisé comme ordre de simplification l'ordre de décomposition lexicographique (*rdo+lex*). Les connecteurs sont ordonnés selon la précédence :

$$\{\geq, \leq\} < \{<, >\} < \{=, \neq\} \text{ et } \{+, -_2\} < \{*, /\} < -_1 \;{}^{12}$$

Algèbre du système d'Alice

E := O | E + E | E * E | E -2 E | -1E | E / E | T1(E).
F := ¬ F | F ∧ F | F ∨ F | ∃A, F | ∀A, F | A R A.
R := > | = | >= | ≠.
A := T1(A) | O.
T1:= TERMES FONCTIONNELS UNAIRES ENTIERS.
O := VARIABLES | CONSTANTES.

Certaines règles décrites dans la thèse de 1976 ont été exprimées à l'aide de plusieurs règles car le langage d'expression des règles était moins riche que celui utilisé par J.-L. Laurière. Pour gérer la commutativité des expressions, des règles ont été rajoutées : $(p \wedge (p \wedge q) \rightarrow p \wedge q)$ en plus de $(p \wedge p) \rightarrow p$, ainsi qu'une règle pour simplifier les coefficients des combinaisons linéaires. La base obtenue est donnée en annexe.

Cette base de règle est construite en unissant trois bases classiques confluentes : (1) la simplification des expressions d'opérateurs +,-,*,/, (2) celle des expressions logiques (∨,∧,¬, ∀,∃), (3) les règles mettant les expressions relationnelles sous une forme régulière (une différence est transformée en une somme et un terme opposé, une relation inférieur-strict est transformée en une relation supérieur-ou-égal). Nous avons prouvé la convergence de deux sous-bases, l'une contenant la sous-base (1) et les règles de transformation sur les opérateurs de la sous-base (3), l'autre contenant la sous-base (2) et les règles de transformation correspondantes dans la sous-base (3) [Liret et al. 1997][13]. Le système final est complet et réalise la simplification des expressions arithmétiques et logiques associatives, commutatives et ordonnées.

Conclusion

---

[11] téléchargeable sur le site du LORIA.

[12] $-_1$ est l'opérateur inverse dans le groupe (0,+) (signe opposé).
$-_2$ est l'opérateur de différence.

[13] Le théorème sur la modularité de la convergence assure que si deux bases sont confluentes et disjointes alors l'union des bases est confluente.

Nombreux sont les travaux inspirés d'Alice. Mais à notre connaissance, aucun travail n'a porté sur la réutilisation des techniques de manipulation symbolique d'Alice dans un solveur CSP. Par ailleurs, aucune étude systématique de l'influence de la manipulation symbolique sur la résolution n'a été menée. Nous proposons dans cette thèse une réponse à ces deux points.

3.3.2 Comparaison avec les approches I.A.

Alice fut le premier à avoir l'idée d'appliquer des méthodes coûteuses mais efficaces sur une partie du problème et des méthodes rapides mais moins précises sur le reste. D'autres travaux partent de cette idée.

Ainsi les contraintes en CLP-CHR sont normalisées après chaque instantiation d'une variable. Les concepts de CHR sont plus généraux que ceux d'Alice. Les contraintes redondantes déduites sont toujours représentées par un arbre contenant une partie de la contrainte initiale. Alice composait les contraintes simples du type $X \geq \alpha$, $X \geq Y$, où X et Y sont des littéraux inconnus, alors que les CHRs autorisent potentiellement toute composition.

Le couplage d'algorithmes a été introduit par Alice pour combiner propagation locale de contraintes et algorithme d'énumération par *Branch&Bound*. Il a été depuis largement repris dans le domaine des algorithmes hybrides et notamment par les travaux de F. Laburthe sur le système CLAIRE pour les problèmes d'optimisation combinatoire.

L'architecture a été réimplantée en C par [Gouachi & Plateau 1999], dans son système LIRESNE[14] ; ce dernier a été appliqué avec succès aux problèmes SAT en combinant des outils de recherche opérationnelle et des algorithmes de maintien de cohérence.

3.3.3 Les idées non exploitées d'Alice

Un des apports de Alice réside dans l'application de raisonnement symbolique que les contraintes soient linéaires ou non. Dans le cas de contraintes non linéaires, il est d'autant plus avantageux qu'aucun algorithme de cohérence ne gère bien ce type de contraintes. En général les solveurs décomposent les contraintes non linéaires en plusieurs contraintes linéaires ou binaires. Si cette représentation est commode pour réutiliser les algorithmes optimisés, [Granvilliers 1998b] affirme cependant qu'elle a l'inconvénient de diminuer la réduction de domaine et la précision dans le cas de domaines réels.

Alice compile les contraintes binaires linéaires ou unaires dans un graphe adapté à la propagation, et abstrait les contraintes non binaires ou non linéaires. Il est possible d'étendre les fonctionnalités du graphe à d'autres types de contraintes simples à filtrer, par exemple les contraintes d'opérateurs d'intervalles, $X^2 = Y$, $X*Y=Z$, $X+Y=Z$, $\sqrt{X} = Y$. La notion de raisonnement formel en Alice peut également être enrichie avec la composition de plusieurs contraintes hétérogènes.

De plus Alice utilise différentes techniques selon l'état du problème et applique les algorithmes coûteux mais efficaces sur un petit nombre de contraintes plutôt que sur la totalité du problème. L'idée est de faire un compromis entre efficacité et rapidité en exploitant la structure du problème. Par exemple, Alice gère les contraintes de différence en les combinant systématiquement avec d'autres contraintes. Ainsi dans le filtrage de $F(1)+R = F(2)$ où F est une bijection et R une variable, Alice déduit que $R \neq 0$. A chaque accès au

[14] Acronyme pour "Langage pour Interpréter et REsoudre des Systèmes en Nombres Entiers"

domaine d'une variable (F(1),R,F(2)), le graphe vérifie si une contrainte de différence n'est pas violée. Cela permet de détecter des valeurs incompatibles sans faire de choix ou retour-arrière.

4 Collaboration de solveurs

Notre travail met en oeuvre deux systèmes computationnels ou « solveurs », l'un de satisfaction de contraintes (CSP) et l'autre de réécriture symbolique de contraintes (RCS). Ce dernier peut être vu comme un ensemble de sous-solveurs (les règles de réécriture ou de déduction de contraintes redondantes) qui collaborent. Le domaine de la collaboration de solveurs s'intéresse à la représentation des plans de contrôle, spécifiant entre autre l'ordre d'exécution des solveurs et le sous-problème du problème courant qui va être traité par un solveur. Généralement et dans notre cas, la collaboration entre ces solveurs n'est pas figée car les situations où un solveur (ou sous-solveur) est utile dépendent *a priori* de l'état du problème. Les stratégies repose alors sur la capacité des solveurs à évaluer l'intérêt de leur action en fonction de l'état du problème et les possibilités du langage. Il faut donc un langage de stratégies globales pour spécifier de manière abstraire le champ d'application de chaque solveur, comme par exemple interdire le déclenchement d'un solveur, définir un ordre d'exécution en fonction d'un historique des applications précédentes.

Dans ce chapitre, nous étudions dans quelle mesure les différentes approches capables de combiner plusieurs techniques sont adaptées à la définition de stratégies réalisant une combinaison de raisonnements symboliques au sein de RCS.

Plusieurs solutions ont été proposées. La plate-forme ELAN intègre un langage de stratégies dynamiques [Borovansky et al. 1997a]. Le langage de CLAIRE [Caseau & Laburthe 1996a] offre les outils nécessaires à un contrôle réactif (C.f. section 3.2.5). Les algorithmes hybrides combinent des solveurs de manière plus précise, en paramétrant les solveurs, en analysant les résultats des autres solveurs ou d'une fonction d'évaluation avant d'activer un solveur. Le modèle que nous proposons au chapitre III permet quant à lui de définir des stratégies dynamiques entre un solveur CSP et des solveurs de réécriture de contraintes symboliques, c'est à dire des ensembles de règles de réécriture de RCS. Notre modèle permet d'affiner la granularité des stratégies et par conséquent de clarifier le rôle de chacun des solveurs (C.f. Chapitre III). Nous distinguons trois critères de comparaison des langages de stratégies : l'expressivité, la souplesse des stratégies et leur dynamicité. Notre approche regroupe partiellement les 3 critères ; le langage est plus souple que CLAIRE mais offre une qualité de définition de stratégie inférieure à celle des environnements de collaboration comme BALI [Monfroy 1996b]. Néanmoins ces derniers s'adaptent difficilement à la collaboration dont nous avons besoin ; car il n'est pas possible à notre connaissance de récupérer l'historique d'application de règles ni d'évaluer de manière qualitative le déclenchement d'une règle.

Le but d'une stratégie est de définir des situations au moyen de critères simples, où un solveur est susceptible de produire un résultat pertinent ou au contraire n'apporte de manière sure aucune simplification du problème. Le fait que les stratégies portent sur plusieurs règles suppose que les règles de RCS sont distinguables et référençables, par exemple au moyen d'une étiquette unique. Nous privilégions comme moyen d'expression les langages déclaratifs ou procéduraux, comme par exemple la plate-forme de collaboration BALI [Monfroy 1996a], et le langage de stratégies intégré dans COLETTE [Castro & Monfroy 1998], les stratégies dans COLETTE [Castro & Monfroy 1998][15]. Afin de pouvoir définir des stratégies qui évoluent en cours de résolution, nous réifions le concept de stratégie ; ce dernier regroupe d'une part la définition d'un contrat par un ensemble de clauses et d'autre part, l'action de mise à jour dans des situations reconnaissables.

4.1.1 Contraintes concurrentes

Le modèle des contraintes concurrentes CC [Saraswat 1993] tente de réaliser les objectifs communs aux architectures de collaboration de solveurs : 1) les contraintes que l'on sait mal gérer sont traitées en parallèle ; 2) le problème est divisé en sous-problèmes dont la résolution est déléguée à des sous-solveurs ; 3) les résultats des solveurs sont rendus compatibles entre eux. En effet la collaboration de solveurs hétérogènes engendre un problème intrinsèque d'interprétation : les résultats fournis ne sont pas forcément dans un format compatible ; une contrainte peut apparaître dans le problème sous une forme non compréhensible par un solveur qui pourtant sait la traiter. Par exemple dans un problème de programmation linéaire entière, l'application d'un Simplexe pour résoudre les inéquations dans R transforme les contraintes en relations à coefficients réels [Hooker et al. 1999],[Benhamou 1996, Chabrier 1999]. Le modèle CC limite les risques d'incompatibilité en imposant un dialogue entre solveurs grâce à deux performatifs ASK (demande bloquante de validité de contrainte) et TELL (ajout/suppression de contrainte et notification aux solveurs). L'opération ASK vise à détecter l'inclusion implicite d'une information dans l'ensemble des contraintes. Une réécriture de RCS apparaît alors comme une particularisation de l'opération ASK. CC est utilisé pour la résolution de problèmes combinatoires dans Oz [Smolka 1995],[Brim et al. 1996],[Rueher & Solnon 1997]).

4.1.2 Limitation des CHR

Les règles sont contrôlées au niveau de la partie «garde-fou ». Les stratégies sont programmées dans la règle en positionnant des drapeaux en partie action, ces derniers étant testés dans la partie condition, en ajoutant ou retirant un prédicat n'apparaissant que dans une seule règle.

Exemple 3.2.1-1:

> La règle R1 :: A1 ➔ B1 est étendue à R2 :: `AutoriseR1`, A1 ➔ B1. Interdire R1 revient à supprimer le prédicat *AutoriseR1* de la base de faits.

Le langage CHR permet de faire des stratégies dynamiques mais il est limité en expressivité et souplesse, car la règle contient à la fois la définition du RCS et les stratégies qui la contrôlent.

Exemple 3.2.1-2 :

> La stratégie qui consiste à bloquer l'ensemble des règles $R_1,\ldots,R_k$, quand l'ensemble des contraintes contient plus de N contraintes n'est possible *a priori* qu'en rajoutant un test dans chaque règle R :

[15] COLETTE est une abstraction des mécanismes CSP basé sur ELAN via un langage de stratégies similaire à celui de BALI.

```
R.tête {⇔,➔} cardinality(ConstraintsStore)<N,R1.guard | R1.corps.
```

De plus les stratégies dont nous avons besoin pour contrôler les règles de raisonnement symbolique sont capables de changer l'ordre des règles, d'en interdire ou de modifier la définition d'une règle en cours de résolution, par exemple la stratégie «pour toute application de R_1, si R_1 a réussi, R_2 est bloquée et si R_1 a échoué, R_2 est réactivée ». Or les CHRs sont compilées sous forme de procédure, ce qui empêche de modifier la base de règles dynamiquement.

Exemple 3.2.1-3 : déclencher R1 quand la règle R2 a échoué deux fois sur une contrainte de type C.

On étend R1 avec une contrainte qui n'est pas dans la tête de `R2` :

```
R1.tête, D {⇔, ➔} R1.guard | R1.corps
```

On définit un compteur de déclenchement de R2 sur une contrainte C sous la forme d'un terme fonctionnel *C(cptR2)*, où *cptR2* représente un entier. Quand le compteur atteint 2, la contrainte D est ajoutée, le prédicat *C(cptrR2)* retiré et R2 ne peut plus s'appliquer :

```
R2.tête, C(2)  ⇔  D, retract(C(2)).

R2.tête,C(cptR2) ➔ cptR2 < 2, R2.guard | R2.corps.

R2.tête,C(cptR2) ➔ cptR2 < 2 | retract(C(cptR2)), cc is cptR2+1,
C(cc). % cas d'échec de R2
```

Le contrôle des règles de propagation est géré par des *token*, c'est à dire des couples formés d'une règle et d'un ensemble de contraintes candidates. La présence d'un superviseur qui positionne ou retire des *token* sur des règles, permettrait de ne pas exécuter systématiquement les règles coûteuses.

4.1.3 Contrôle dynamique par des règles de réécriture, ELAN

ELAN [Kirchner et al. 1993, Vittek 1996], et par la suite Maude [Clavel et al. 1996] est un *framework* fondé sur la réécriture, pour prototyper des systèmes computationnels sous contraintes. Un système computationnel formalise un solveur dont le calcul est séparé du contrôle. ELAN définit un langage de stratégies, basé également sur des règles de réécriture, qui sert à indiquer dans quel sous-terme une règle ELAN peut s'appliquer et combien de fois. Par certains côtés, ELAN peut être vu comme une plate-forme adaptée à la combinaison d'algorithmes CSP et de règles exprimant des raisonnements symboliques. L'application d'une stratégie ELAN sur une règle et un terme renvoie la liste des termes résultants des applications valides de la règle. En cela, une stratégie définit la granularité d'application d'une règle ELAN et peut contribuer à limiter la complexité du raisonnement symbolique.

Le *framework* de construction de stratégies que nous proposons pour contrôler les raisonnements symboliques. (C.f. chapitre III et IV) diffère de celui d'ELAN, dans le sens où la collaboration n'est pas exprimée dans un programme de stratégies, mais dispersée au niveau de chaque règle abstraite. Les possibilités de combinaison de stratégies primitives sont plus étendues en ELAN grâce à des opérateurs proches des structures de base d'un langage de programmation (concaténation, séquence, boucle-*iterate,* boucle-*repeat*)

[Borovansky et al. 1997b, 1997c][16]. Néanmoins dans le cas où le comportement des solveurs en fonction de l'évolution du problème n'est pas connue *a priori,* ce langage nous semble inadapté. Dans notre approche nous proposons de définir des stratégies qui se modifient en fonction des résultats des solveurs eux-mêmes. Seul un opérateur de composition y est autorisé : la séquence, qui évalue successivement les conditions des stratégies et prend en compte la première stratégie dont la condition est vraie.

Un module ELAN (`RCS`) peut contenir des règles de RCS sur un type abstrait *Constraint* représentant l'expression symbolique d'une contrainte ; les opérateurs sont `add`(*Constraint*), `remove`(*Constraint*), `transform`(*Constraint*), `execute`((*Constraint*)`list`), `candidate`(*Constraint*).

ELAN propose trois stratégies de base : *dont know choose(r)* renvoie l'ensemble des résultats obtenus en appliquant r sur l'ensemble des contraintes candidates ; *dont care choose(r)* renvoie un résultat aléatoire parmi ceux renvoyés par *dont know choose(r)* ; *first(r)* renvoie le résultat de la première application valide. La stratégie d'application d'une règle de RCS consiste en une itération de la base de règles sur l'ensemble des contraintes candidates, ce qui correspond en ELAN au programme de stratégie suivant :

```
     Strategy_reasoning
         (iterate (forall r in RCS, strategy_r(r)) enditerate).   (*)
end of strategy
```

`strategy_r` équivaut à *dont care choose* pour les règles spécifiques renvoyant un seul résultat (une seule possibilité d'application).

En revanche, les règles générales de composition de contraintes linéaires requièrent une stratégie *dont know choose,* car il faut choisir parmi les différentes contraintes produites. Enfin pour exécuter une seule fois une règle sur l'ensemble des contraintes, il suffit en ELAN de ne pas englober la règle dans un opérateur d'itération.

```
module symbolicReasoning
sort : Rule Constraint ; end
op global
    blocked(@,@): ( Constraint nelist ) Bool
    blocked: Bool
    failureCounter: integer
    limit(@): ( Rule ) integer
endop
strategy strategy_r_1
declare c : Constraint
      l : nelist
      r : Rule
// exécution de r uniquement sur les termes pour lesquels elle n'est pas bloquée
    [strategy_r_1]  blocked => end         //pas d'exécution de r
    [strategy_r_1]  blocked(c,l) => end    //pas d'exécution de r sur c
    [strategy_r_1]  true => r end          // exécution de r
strategy strategy_r_2
declare r : Rule
```

[16] Une stratégie ELAN *Strat(rList, e)* est un moyen de description et de calcul de l'ensemble des termes possibles résultants de l'application d'une liste de règles *rList* sur un terme de départ *e*. La séquence *s1;s2;s3* signifie que le résultat de *s1* est l'entrée de s2 et le résultat final est celui de s3. La boucle *repeat* est l'application récursive d'un ensemble de règles jusqu'à l'échec. Si la suite $((S_n)_t)$ est ainsi construite, *repeat* rend comme résultat le dernier terme $((S_n)_{t\infty})$ tandis que *iterate* rend l'union des $(S_n)_{ti}$.

// exécution de r uniquement si moins de limit échecs ont eu lieu

```
    [strategy_r_2]  failureCounter < limit(r) => r
    [strategy_r_2]  failureCounter >= limit(r) => end
end of strategy
```

4.1.4 Système d'inférence de contraintes, les outils ILOG

Le système d'inférence d'ILOG, ILOG-Rules regroupe les techniques de programmation logique avec le monde de la programmation par objets. ILOG-Rules est un générateur de systèmes experts basés sur des règles générales sur des objets composites. Il permet en outre d'ajouter et retirer des objets pendant l'application de règles. En particulier, il peut faire des inférences sur les contraintes définies dans ILOG-Solver.

Plus précisément, un prédicat d'ILOG-Rules signifie l'existence d'un objet dans la mémoire. Par exemple, le prédicat suivant est vrai si la mémoire contient un cercle noir : `(Form couleur = noir, contour = cercle)`. La règle suivante ajoute un triangle rouge à chaque fois qu'il y a simultanément un cercle noir et un carré blanc :

```
(Form couleur = noir, contour = cercle),
(Form couleur = blanc, contour = carre)
➔ (Form couleur = rouge, contour = triangle).
```

Exemple : théorème des deux droites perpendiculaires à une troisième[17]

```
?d1 : (Form contour = droite), ?d2 : (Form contour = droite),
?d3 : (Form contour = droite),
(PerpendicularConstraint  x = ?d1  y = ?d3),
(PerpendicularConstraint  x = ?d2  y =  ?d3)
➔ assert((ParallelConstraint  x = ?d1  y  = ?d2)).
```

Les prédicats portant sur les contraintes binaires de perpendicularité et parallélisme `(PerpendicularConstraint` et `ParallelConstraint)` sont des *User Defined Predicate* fortement typés selon le *ILOG-Rules Manual*.

Par ailleurs, en mode interprété, ILOG-Rules permet d’ajouter ou retirer des règles ce qui permet de contrôler dynamiquement la base, par exemple en retirant une règle si celle-ci ne doit pas s'appliquer. Cependant lorsque la base de règles de ILOG-Rules est compilée, le contrôle ne peut qu’être intégré au niveau des règles elle-mêmes car la base est dans ce cas non modifiable. Le contrôle de chaque règle peut être représenté par un objet particulier que nous appelons «*objet de contrôle* », qui représente les différents paramètres d’une stratégie. Cet objet doit alors être différent dans chaque règle à contrôler. La stratégie est alors spécifiée dans les règles sous la forme d’un test d'existence de cet objet de contrôle. Cette approche offre les mêmes inconvénients que celle présentée pour les CHRs ; néanmoins l'utilisation d'objets donne plus de liberté d'expression.

Exemple : une règle de RCS contrôlée par un objet ControlApplication(5).

R1 s'applique au bout de 5 déclenchements des autres règles avec succès.

```
R1 :: (ControlApplication n=5),….➔ ….,(ActionR1).
```

"une règle de collaboration qui compte le nombre d'applications de R1"

```
R2:: (…),… ➔retract(controleApplication n = ?d4),
             assert(controleApplication n = ?d4 + 1)
```

[17] ?d1 est une variable représentant l'objet du premier prédicat .

Le contrôle se trouvant au niveau des règles, il est plus difficile d'implanter une stratégie tenant compte des échecs ; car on ne peut récupérer les situations d'échecs d'application, au niveau de la règle elle-même !

4.1.5 Algorithmes hybrides

Les algorithmes hybrides sont apparus à partir de l'idée que l'utilisation conjointe de plusieurs traitements permet d'adapter la résolution à différentes parties d'un problème et par conséquent d'améliorer la solution. Cette approche connaît un essor certain notamment grâce à l'élaboration de langages de spécifiacation d'algorithme hybride comme SalSA [Laburthe & Caseau 1998]. Dans le cadre de problèmes à domaine réel, la programmation logique (CLP) a été combinée avec la programmation linéaire (LP) (appariement de vols d'avions [Chabrier 1999]). Le modèle MLLP [Hooker et al. 1999] distingue le modèle de contraintes symboliques et des contraintes mathématiques, et propose de décrire la communication par des règles de mise à jour du type `H(X)`→ `AX>=B` où `H(X)` est une condition sur les variables du modèle CLP. Les nombreux travaux de F. Laburthe [Laburthe et al. 1998] montre l'intérêt d'utiliser la satisfaction de contraintes pour déterminer les choix faits par les techniques de recherche opérationnelle (insertion gloutonne, recherche tabou,...).

Ces travaux montrent également la difficulté d'exprimer les hybridations dans un cadre formel. Le projet ESPRIT CHIC-2[18] a d'ailleurs pour but de développer une plate-forme et une méthodologie pour construire et utiliser des algorithmes hybrides [CHIC-2 1997, CHIC-2 1998]. Les langages SalSA et Localizer [Michel & Van Hentenryck 1997] proposent une solution à ce problème en introduisant notamment la notion d'invariant. Un invariant est une relation fonctionnelle entre des variables, qui se propage de manière transparente lorsqu'une des variables change. Nous pensons que la notion d'invariant est adaptée à la description de stratégies à un haut niveau car un invariant est à la fois réactif, déclaratif et combinable à d'autres invariants. Nous proposons donc de définir une stratégie comme un invariant sur des règles de RCS, réagissant à chaque évaluation du résultat de l'application d'une règle (échec ou succès).

4.1.6 Système déductif et réactif, CLAIRE-ECLAIR

La plate-forme CLAIRE dispose de règles de propagation multi-prémisses portant sur des objets. Ces règles sont intégrées au programme de manière transparente et sont reliées aux attributs des objets par un démon. A chaque fois que la valeur d'un attribut change, les règles ayant une condition sur cet attribut sont déclenchées[19]. Le langage de CLAIRE inclut des primitives de gestion de sauvegarde-restauration d'état (`world_push()`, `world_pop()`, `world_remove()`). Ces dernières mémorisent ou restaurent les états d'un ensemble d'objets du programme ; en particulier, elles permettent de maintenir des informations nécessaires aux stratégies, comme l'historique des états de RCS. Grâce à la bibliothèque de contraintes ECLAIR [Laburthe et al. 1998][20], construite au-dessus de CLAIRE, les règles de propagation peuvent être exploitées pour modéliser les règles de RCS ou pour modéliser des stratégies.

[18] Creating Hybrid Solutions for Industry and Commerce

[19] Nous notons une restriction purement technique : le démon déclencheur de la règle n'est relié qu'aux objets en prémisse et non pas aux attributs de ces objets ; si une variable est changée dans une contrainte, la règle doit porter sur le domaine de la variable pour en être informée.

[20] Nous voyons ECLAIR comme un sous-ensemble de BackTalk, le modèle est similaire mais les démons en ECLAIR ne sont pas surchargeables.

La première solution entraîne une restriction au niveau des stratégies du fait que les règles sont compilées sous forme procédurale.

Combinaison de règles de propagation en Claire

Cette approche utilise les règles de propagation Claire pour représenter les règles de RCS. Elle permet de bénéficier de la rapidité d'exécution des déductions. Cependant nous notons deux limites : 1) Les règles se déclenchent systématiquement à chaque accès en écriture à un des attributs de contrainte, ce qui peut amener à des calculs symboliques combinatoires ; 2) Le démon déclencheur n'est pas redéfinissable, ce qui empêche par exemple d'interdire dynamiquement des règles ou de retarder leur exécution jusqu'à la fin du maintien de cohérence.

Exemple 3.2.5-1: Scheduling

Considérons la règle `ressourceExclusiveNTaches` sur n tâches $T_1,\ldots T_n$ qui utilisent la même ressource M : si $T_1,\ldots T_n$ doivent se terminer avant le début d'une tâche T donnée, alors elles ne peuvent s'effectuer que les unes après les autres.

```
[Tache <: thing(ressource:Ressource, duration,start:integer)]
[ressourceExclusiveNTaches (SetTi : List [PrecedenceCt]) ::
forall (i in SetTi,
ressource(x(i))=ressource(x(first(SetTi))) &
y(i)=y(first(SetTi)))
  ➔
min(list{start(i)|i in SetTi}) + sum(list{duration(i)|i in
SetTi}) <= y(first(SetTi))]
```

`PrecedenceCt` est la classe de la contrainte sur deux tâches X et Y imposant que «X se termine avant le début de Y ». Une tâche possède ici trois attributs `ressource`, `duration et start` (date de départ).

Exemple 3.2.5-2 : détection d'incohérence

Une contrainte d'occurrence exprime le fait que le nombre d'éléments d'un ensemble S de valeur A est borné par un intervalle I. Si la taille de S est inférieure au minimum de I alors la contrainte ne sera jamais satisfaite. On lève donc une exception `Inconsistency` signalant au programme principal (le solveur) que le contexte courant est incohérent.

```
[Inconsistency <: exception(s : string, o : constraint)]
[occurrenceInconsistency (c : OccurrenceCt)::
 size(variables(c)) < minValue(targetVariable(c))
 ➔
 Inconsistency(s= "incoherence structurelle", o = c)]
```

Détection de l'incohérence x>y et y>x :

```
[inequationInconsistency (ineq1, ineq2 : GreaterThanCt) ::
 x(ineq1) = y(ineq2)
 ➔Inconsistency(s="incohérence globale", o = ineq1)]
```

Dans cette approche, les stratégies visent à contrôler les règles de propagation en Claire. Le déclenchement de règles est supervisé en modifiant l'état d'objets de contrôle, spécifiques de chaque règle. La notion d'objet de contrôle global représente les stratégies contrôlant plusieurs règles ensemble, les règles de CLAIRE faisant appel aux instances de ces objets.

```
ObjetControle <: thing(mainRule = unknown)]
```

Les stratégies nécessitant d'accéder à des informations sur les règles, celles-ci doivent être étiquetées. Une règle est alors augmentée d'un test T1 et d'une action A1. T1 rend *Vrai* si l'objet de contrôle autorise le déclenchement et A1 met à jour le contrôle avec le nouvel état du problème :

```
"R1" : [oc : ObjetContrôle, …]
(oc testEtatDe("R1")),condition(r1)            /*T1*/
  ➔
 action(r1),(oc metAJour("R1")).               /*A1*/
```

Les méthodes de récupération d'information `testEtatDe(aRuleLabel):boolean` et `metAJour(aRuleLabel)` sont spécifiques d'un type de contrôle en fonction des informations nécessaires pour vérifier `T1`. La méthode `metAJour()` s'applique quand le déclenchement de la règle a réussi ; cependant il n'est pas possible de modéliser un contrôle similaire qui compte le nombre d'échecs, contrairement à notre architecture (C.f.. chapitre III).

Dans la mesure où la représentation des règles dans le code compilé est *built-in,* une solution consiste à modifier le compilateur de règles Claire de manière à 1) stocker les étiquettes des règles dans une liste RULES de l'environnement Claire ; 2) définir une liste ETATACTION de même taille que RULES dont les éléments appartiennent à {"succès", "échec", "inutile"}. Ces deux listes doivent être accessibles depuis l'environnement utilisateur de Claire et depuis les objets de contrôle.

Exemple 3.2.5-3 : Stratégie sur des règles de propagation manipulant des contraintes

Les stratégies visent à changer les états des règles. Le chapitre III décrit les états d'une technique de RCS. (bloqué, réussi, échoué, libre). Par exemple, soit la règle `AllDifférent(x,y), x-y=z ➔ z≠0`. L'exécution de cette règle n'est utile qu'une seule fois pour chaque égalité linéaire : on n'applique la règle sur une contrainte C que si son execution sur C n'est pas explicitement bloquée. Pour cela la règle est étendue en *"R"* : `AllDifférent(x,y), x-y=z, T1 ➔ z≠0, A1`. Le prédicat T1 retourne *Vrai* si *"R"* n'est pas bloquée et l'action A1 bloque *"R"*.

```
ObjetContrôleExemple1. TestEtatDe(aRuleLabel)
     Return (étatDe(aRuleLabel)) = "libre"

    Objet-Contrôle : étatDe(aRuleLabel)
     Return (RULES elementAt(aRuleLabel))

    ObjetContrôleExemple1 : metAJour(aRuleLabel)
     If (ETATACTION elementAt(aRuleLabel) = "succès"
     Then RULES elementAtPut(aRuleLabel, "bloquée")
```

Exemple 4.2.5-4 : Stratégie sur plusieurs règles

Les stratégies peuvent être définies en fonction des états et exécutions de plusieurs règles. Par exemple, soient trois règles r1, r2, r3 (S = {r1, r2, r3}) et r une règle coûteuse. Pour limiter une trop grande combinatoire, on ne veut appliquer r que si au moins deux succès ont eu lieu dans S (ItExecutionAfterSuccès(r, S, 2)).
`ObjetContrôleExemple2` contient un attribut `compteur` de type entier, un seuil entier et une liste de règles distinctes de `mainRule`.

```
[ObjetControleExemple2 <: ObjetControle(mainRule = unknown,
seuil : integer, otherRules : List[String], n : integer = 0)]
```

```
ObjetContrôleExemple2 : testEtatDe(aRuleLabel)
 Return (aRuleLabel  != "bloqué") & (((aRuleLabel = mainRule
and (n <= seuil)) | (aRuleLabel != mainRule))

ObjetContrôleExemple2 : metAJour(aRuleLabel)
 If (aRuleLabel = mainRule) n := 0
 Else if (contains(otherRules, aRuleLabel)) n := n+1
```

Stratégie réactive et déclarative sous forme de règles de propagation

Comme nous l'avons vu à la section 3.2.2, les stratégies s'expriment naturellement de manière réactive et déductive ; une seconde approche consiste donc à exploiter le mécanisme d'inférence de CLAIRE pour modéliser les stratégies. Cela permet en outre de séparer l'action de RCS de son contrôle. Cette approche a deux avantages dans notre problématique : 1) le contrôle est réactif donc plus flexible et plus adapté à la détection de situations favorables ; 2) RCS ne s'applique pas systématiquement mais seulement sous le contrôle des règles. En revanche les règles étant compilées, il n'est pas possible de rendre les stratégies dynamiques.

La classe des techniques de RCS possède un attribut `status` de type *symbole*, qui définit le statut de toute instance. Les règles implantent des stratégies déductives de manière à réagir à toute modification de l'attribut `status`, en particulier lors d'un échec.

```
[RCSRule <: thing(candidatesCts : list[Constraint], status =
"applicable")]
```

Conclusion

Une des restrictions des règles dans CLAIRE est due au fait qu'elles ne peuvent pas être référencées dans du code. Dans cette section, nous avons considéré deux approches :

- Les règles de propagation intègrent à la fois la réécriture et le test des stratégies, bénéficiant ainsi de la structure optimisée d'un programme CLAIRE mais perdant en lisibilité. Le compilateur de CLAIRE est étendu de manière à ce que les règles de propagation soient indexables, par exemple par une étiquette unique "LABEL" et deux procédures ayant des noms spécifiques de cette étiquette `testConditionOfRuleLABEL: arg, doActionOfRuleLABEL: arg`.
- Les techniques de raisonnement symbolique sont réifiées et les stratégies implantées par des règles de propagation sur ces objets. Celles-ci se déclenchent à chaque fois qu'un des attributs d'un objet est modifié. Pour qu'elles jouent leur rôle de stratégie, les règles sont reliées à un attribut booléen `declenche`, représnetant le fait qu'on tente d'appliquer la règle. Cette approche permet de séparer RCS du mécanisme de collaboration et réutilise la structure optimisée de CLAIRE pour implanter le mécanisme d'imposition de stratégies.

Néanmoins une question reste concernant l'influence des modifications de la plate-forme sur les performances de CLAIRE, de même que les conséquences sur la collaboration, de la réification des techniques de RCS combinées à un contrôle réactif rapide.

4.1.7 Stratégies dynamiques

Une stratégie dynamique spécifie des situations où l'application de règle(s) peut produire un résultat pertinent, quelle que soit la contrainte sur laquelle elle s'applique *a priori*. L'action des stratégies consiste alors à activer ou désactiver des règles en fonction d'une connaissance non triviale, à savoir l'évaluation de la qualité d'un résultat (son intérêt, son arité, sa définition) produit par un solveur.

Le langage de stratégies de ELAN exprime une collaboration statique au sens où il impose de connaître l'enchaînement d'application des solveurs (règles). L'extension de ce langage par des règles de réécriture [Borovansky et al. 1997a] contrôlant une stratégie ELAN a permis de modéliser des stratégies dynamiques. De plus, les stratégies étant des termes du langage, elles sont normalisables, c'est à dire que le système simplifie les stratégies avant de les appliquer. Néanmoins les systèmes basés entièrement sur la réécriture imposent une contrainte forte sur le typage des données, contraintes qu'il n'est pas toujours possible de satisfaire dans les problèmes complexes ; les experts fournissent en général des relations causales «situation-action » qui rapproche le contrôle de solveurs d'un raisonnement à partir de cas.

Notre approche

Dans notre langage présenté au chapitre III, chaque stratégie est réifiée et peut être statique ou modifiée par le programme. La bibliothèque conçue permet le retrait et l'ajout de stratégie en cours de résolution (C.f. chapitre IV).

Exemple : stratégie dynamique avec notre approche

```
retract(S1). add(StratBlockIfUseLess(r, 2))
```

Cette stratégie composite retire la condition d'application relative à `S1` et en rajoute une relative uniquement à `r` (si le nombre d'échecs de r dépasse 2 alors bloquer `r`).

5 Conclusion

Le caractère orthogonal des techniques CSP et de RCS nécessite de mettre en œuvre une collaboration fine en fonction d'une évaluation qualitative du résultat par rapport à l'objectif : la priorité des solveurs change en cours de résolution. Les systèmes existants offrent certes un cadre général pour modéliser à la fois les contraintes et les manipulations symboliques ; néanmoins rares sont ceux qui manipulent les règles de déduction ou de réécriture de contraintes, ce qui empêche de séparer le contrôle du langage de règles. De plus si la précision des stratégies peut être résolue dans une certaine mesure (par exemple par des algorithmes hybrides intégrant des règles de mise à jour sur des objets composites), les stratégies dynamiques restent un point difficile qui nécessite des expérimentations sur des problèmes connus.

Chapitre II : Etude expérimentale et motivation

Ce qui importe le plus, c'est la recherche elle-même. Elle est plus importante que les chercheurs. La conscience doit rêver; il lui faut un territoire pour ses rêves, et, rêvant, elle doit invoquer des rêves toujours nouveaux (Morgan Hempstead, Conférences de Lunabase).

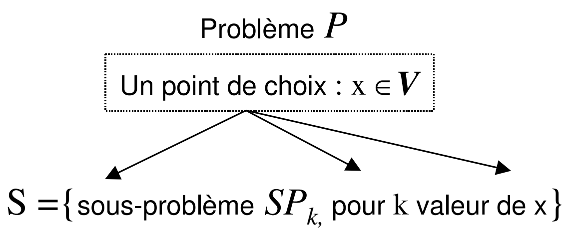
Franck Herbert, Destination : vide, R. Laffont, chapitre 3, 1978

Nous décrivons ici une étude expérimentale de l'influence de la réécriture de contraintes symboliques, réalisée sur le *framework* BackTalk. L'étude porte sur des problèmes pour lesquels on connaît une solution de manière à faciliter l'analyse des conséquences de la manipulation symbolique de contraintes. La première section de ce chapitre synthétise l'étude expérimentale réalisée. La seconde section présente la problématique dégagée par ces expériences, problématique qui étend celle de la programmation par contraintes et objets, présentée dans [Roy et al 2000].

1 Etude expérimentale

Le but de cette étude est de comparer la résolution de problèmes connus pour différentes techniques de filtrage et de RCS appliquées avant ou pendant la résolution. Un algorithme de type *backtracking* chronologique est appliqué pour la recherche. Le choix de problèmes connus et d'espace de recherche restreint permet de comparer l'effet d'une collaboration par rapport à une résolution purement CSP.

L'étude fait appel à la notion de «point de choix » que nous définissons par une variable de **V** dont le domaine est énuméré selon une heuristique. Un point de choix correspond à un nœud dans l'arbre de recherche. Le nombre de points de choix indique alors la dépendance de la solution vis à vis des heuristiques : plus ce nombre est faible, plus la résolution est indépendante des heuristiques.



Le nombre de retour-arrières correspond à la cardinalité de l'ensemble des sous-problèmes explorés (S) ; il dépend du point de choix engendrant S et renseigne sur la capacité des techniques de traitement des contraintes, à réduire l'espace de recherche avant un choix. Par ailleurs, nous définissons la notion de composition «utile » comme un calcul symbolique sur des contraintes, qui engendre une contrainte dont la propagation provoque une réduction de domaine.

Dans cette étude les contraintes sont manipulées selon trois approches : (1) simplification d'un ensemble de contraintes, (2) combinaison linéaire de contraintes, (3) réécriture d'une contrainte à chaque instantiation, dite «réécriture d'instantiation ». Les approches (1) et (3) sont implantées dans une base de réécriture, décomposée en fonction du moment de déclenchement, *i.e.* à la création d'une contrainte ou dans la procédure de résolution. La réécriture d'instantiation simplifie toute contrainte en une contrainte de même classe en remplaçant les occurrences de la variable instantiée par sa valeur. Son utilisation garantit que toutes les contraintes sont dans une forme régulière et facilite l'implantation de compositions de contraintes, en particulier pour la reconnaissance de contraintes équivalentes à un coefficient près[21]. Les contraintes linéaires sont composées deux à deux de quatre manières : deux égalités ou inégalités, ou bien une égalité et une inégalité, ou encore une égalité et une différence. L'approche (2) effectue une composition de deux contraintes C_1 et C_2, ayant au moins une variable en commun et sera appelée dans ce chapitre «composition (linéaire) de contraintes ». Celle-ci produit l'ensemble SC = $\{C_1\text{-}k{*}C_2\}$, où k est calculé en fonction d'une certaine propriété des contraintes redondantes, par exemple leur arité est inférieure à celle de C_1 et de C_2, ou leur portée ne contient pas au moins une variable commune à C_1 et C_2 ; la taille de SC est alors bornée. Dans cette étude, les contraintes de SC ajoutées sont celles qui ont l'arité la plus petite ; s'il n'existe pas de contrainte dans SC, une composition plus souple est réalisée en utilisant le second critère et en conservant les contraintes où sont retirées le plus de variables communes. Par ailleurs, les compositions linéaires sont effectuées après vérification de la cohérence des contraintes ; chaque retour-arrière est suivi du retrait des contraintes qui ont été déduites, puis des compositions tenant compte du nouveau choix.

Nous avons mené deux études, l'une sur une série de cryptogrammes standards, l'autre sur les problèmes de carré magique. Ces problèmes relativement simples offrent l'avantage d'être connus ; leur résolution par des techniques de CSP peut donc être facilement reproduite. Dans le cas des cryptogrammes, nous avons constaté que les techniques de RCS énoncés ci-dessus ont des effets positifs sur la résolution, auxquels nous proposons une explication informelle. Nous définissons au chapitre IV-1 des critères d'évaluation de ces situations en fonction de la densité d'un problème et les validons sur des séries de cryptogrammes générées aléatoirement. Dans la seconde étude, nous avons constaté que la composition de contraintes accroit le nombre de contraintes du problème, de manière dramatique pour les algorithmes de filtrage de CSP. Dans ce contexte, plusieurs stratégies permettent de limiter le nombre de contraintes redondantes ajoutées, par exemple interdire des réécritures dans certaines situations, composer les contraintes trois par trois. En effet le problème est difficile car peu dense ; deux contraintes ont au plus une variable en commun. Les compositions linéaires fonctionnent mal car elles sont nombreuses et ajoutent beaucoup de contraintes qui ne provoquent pas d'instantiation de variable.

Notation :

On note CSP+RCS le solveur intégrant la manipulation symbolique de contraintes et le maintien de cohérence.

[21] On retrouve le besoin d'avoir une forme canonique dès qu'une manipulation symbolique est utilisée de manière automatisée.

1.1 Etude de cryptogrammes standards

1.1.1 SEND+MORE=MONEY

On modélise le problème avec 12 variables (S E N D M O R Y) de domaine 0..9 et (R1 R2 R3 R4) de domaine 0..1.

(0) AllDifférent(S,E,N,D,M,O,R,Y) (1) D+E=Y+10.R1 (2) R1+N+R=E+10.R2 (5) R4=M
(3) R2+E+O=N+10.R3 (4) R3+S+M=O+10.R4 (6) M ≥ 1 (7) S ≥ 1

Borne-cohérence seule

Le problème initial est rendu borne-cohérent (1ère ligne du Tableau 3). Avec une heuristique «fixe » (*i.e.* ne change pas pendant la résolution) et «standard » (C.f liste en annexe), on trouve la solution en au minimum deux retour-arrières (heuristique minSize&maxValue&ordreLexicographique) et au minimum un point de choix. (heuristique mostConstrained&minValue&ordreLexicographique) (C.f. Tableau 4).

	Actions	S	E	N	D	M	O	R	Y	R1	R2	R3	R4
1	BC	9	2..8	2..8	2..8	1	0	2..8	2..8	0..1	0..1	0	1
2	Choix R1=0									0			
3	BC		2..6	2..7	4..8			2..8	4..8				
4	Choix R2=0										0		
5	BC échec!		4	4				2..4					
6	Choix R2=1										1		
7	BC échec!	9	3..4	5	4..6			7..8	5..8				
8	Choix R1=1									1			
9	BC		4..8	4..8	4..8			2..8	2..6				
10	Choix R2=0										0		
11	BC échec!		7..8	4..5	4..8			2..3	2..6				
12	Choix R2=1										1		
13	BC		4..7	5..8	5..8			5..8	2..5				
14	Choix E=4		4										
15	BC échec!			5	8			8	2				
16	Choix E=5		5										
17	BC			6	7			8	2				

Tableau 3 : borne-cohérence, Full Look-ahead, minSize&minValue&ordre lexicographique.

Ligne 4 : Le choix R1=0 suivit de R2=0 entraîne la réécriture de (2) en E+R=N, dont le filtrage instancie N et E avec la valeur 4. La contrainte AllDifférent provoque un échec.

Ligne 6 : Le choix R2=1 entraîne la réécriture de (2) en N+R=E+10, dont le filtrage provoque un échec car aucune solution n'est dans les domaines. Le filtrage de (2) et (3) réduit les domaines de E à 3..4, de R à 7..8, de N à 5, puis celui de E à {4} d'où l'échec.

Ligne 10 : Le choix R1=1 suivi de R2=0 entraîne la réécriture de (3) en N=E. Or le filtrage de (2) (réécrite en N+R+1=E) réduit les domaines de N et E tel qu'ils soient disjoints.

Ligne 12 : Le choix R2=1 entraîne la réécriture de (3) en E+1=N.

Ligne 14 : Le choix E=4 entraîne la réécriture de (1) en D-Y=6 et (2) en N+R=13.

Ligne 16 : Le choix E=5 entraîne la réécriture de (1) en D-Y=5 et (2) en N+R=14.

Avec l'heuristique minSize&maxValue&ordreLexicographique, les choix R2=0 et R1=0 sont évités : après filtrage (ligne 13), E est affectée successivement avec 7, 6 et 5.

Avec l'heuristique mostConstrained, E a pour domaine 2..8 lorsqu'elle sert de point de choix. Il faut alors trois retour-arrières pour trouver la solution, avec minValue ou MaxValue (C.f. Tableau 4).

	Actions	S	E	N	D	M	O	R	Y	R1	R2	R3	R4
1	BC	9	2..8	2..8	2..8	1	0	2..8	2..8	0..1	0..1	0	1
2	Choix E=2		2										
3	BC échec!			3	3..8			8	3..8	1	1		
4	Choix E=3		3										
5	BC échec!			4	2,4..5			2,4..8	5..8	0	1		
6	Choix E=4		4							1			
7	BC échec!			5	8			8	2	1	1		
8	Choix E=5		5										
9	BC			6	7			8	2				

Tableau 4: BC, Full Look-ahead, MostConstrained&MinValue&OrdreLexicographique

Borne-cohérence, réécriture d'instantiation et composition de contraintes

La composition d'équations linéaires est inspirée de l'algorithme de Gauss : la contrainte engendrée possède moins de variables que les deux contraintes composées.

	Techniques	S	E	N	D	M	O	R	Y	R1	R2	R3	R4
1	BC,Réécriture	9	2..8	2..8	2..8	1	0	2..8	2..8	0..1	0..1	0	1
2	AllDiff+(3)	R2 ≠ 0											
3	BC,Réécriture		2..7	3..8	2..8			2..8	3..8	0..1	1		
4	(3)+(2)	R1+R=9											
5	BC,Réécriture		5..7	3..8	7..8			8	2..3	1			
6	Choix E=5		5										
7	BC,Réécriture			6	7				2				

Tableau 5 : composition, Réécriture, Borne-cohérence, Full Look-ahead, minSize&MaxValue&ordre lexicographique.

Ligne 1 : Réécriture de (3) en R2+E=N.

Ligne 2 et 4 : Quand plus aucune réduction n'est possible, la contrainte AllDifférent est composée (ligne 2) avec (3), ce qui provoque l'ajout de la contrainte R2 ≠ 0 car N ≠ E. A la ligne 4, toutes les contraintes linéaires sont composées deux à deux, dont celle de (3) et (2) qui provoque l'ajout de R1+R=9.

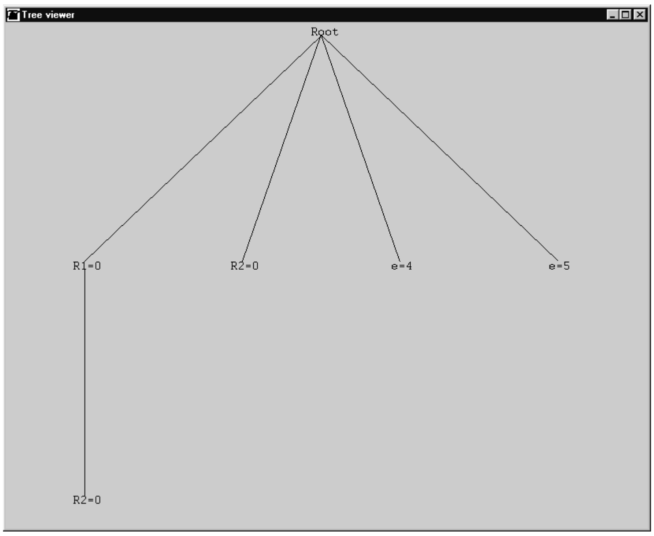
Ligne 3 et 5 : Après chaque ajout d'une contrainte, le réseau CSP est maintenu cohérent. Le filtrage de R1+R=9 entraîne l'instantiation de R1 avec 1 et de R avec 8.

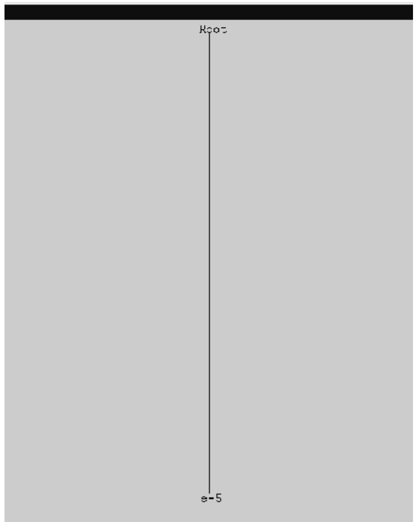
A la ligne 5, les domaines sont davantage réduits que, dans la même situation, avec la borne-cohérence seule (C.f Tableau 3, ligne 13). La composition de contraintes linéaires a permis de déterminer des variables car la contrainte engendrée portait sur une variable (R1) dont le domaine contenait deux valeurs. De plus aucun choix n'a été nécessaire auparavant ! Plus précisément, la composition d'une contrainte linéaire et d'une contrainte *AllDifférent* supprime le choix de R2 (ligne 2 du Tableau 5 versus lignes 4 et 10 du Tableau 3). Une composition linéaire supprime le choix de R1 (lignes 2 et 8 du Tableau 3). Finalement une variable est énumérée, contre trois variables en l'absence de RCS.

Heuristiques	BC, Réécriture	Arc-C, Réécriture	BC, Réécriture, Composition
MinSizeMinValue Lexicographique	4bt,5choix R1,R2,E	4bt,5choix R1,R2,E	0bt,1choix E, 2 compositions (Alldifflin, linlin) [22]
MinSize MaxValue	2bt,5choix R1,R2,E	2bt,3choix	2bt,2choix E, 2 compositions (Alldifflin, linlin)
Mostconstrained, MinValue	3bt, 4choix E	3bt,4choix E	0bt,1choix E, 2 compositions (Alldifflin, linlin)
GreatestRegret, MinValue	3bt,4choix E	3bt, 4choix E	0bt,1choix E

Tableau 6 : SEND+MORE=MONEY.

Dans le cas de la ligne 1 du Tableau 6, les arbres de recherche avec et sans composition sont les suivants :





Borne-Cohérence *Borne-Cohérence, Réécriture, Composition*

Conclusions

La réécriture d'instantiation ne changeant pas la classe de la contrainte, son utilisation en combinaison avec la borne-cohérence ne change pas le nombre de points de choix ou de retour-arrières de la résolution, ni l'ordre des choix faits par le système à heuristiques identiques (C.f. chapitre III-2.1). Par ailleurs, le Tableau 6 montre que la composition permet de rendre la résolution plus indépendante des heuristiques en repoussant le moment où un choix sera nécessaire jusqu'à ce que les domaines soient de petite taille. Le nombre de points de choix s'en trouve diminué : le seul choix qui est fait est celui de E, au moment où son domaine est {5,6,7}. Il y a donc au maximum 2 retour-arrières (si on choisit la valeur maximale du domaine) et au minimum 0 retour-arrière (si on choisit la valeur minimale). Par conséquent, dans tous les cas étudiés, la composition linéaire permet de gagner au minimum

[22] Alldifflin signifie « composition de AllDifférent et d'une égalité linéaire » ; linlin signifie « composition de deux contraintes linéaires »

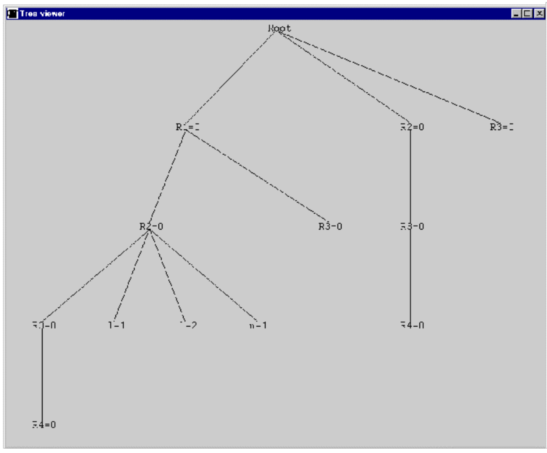
trois points de choix ; elle évite le choix de deux variables (ligne 1 et 2) ou évite le choix de deux valeurs (ligne 3 et 4).

1.1.2 DONALD+GERALD=ROBERT

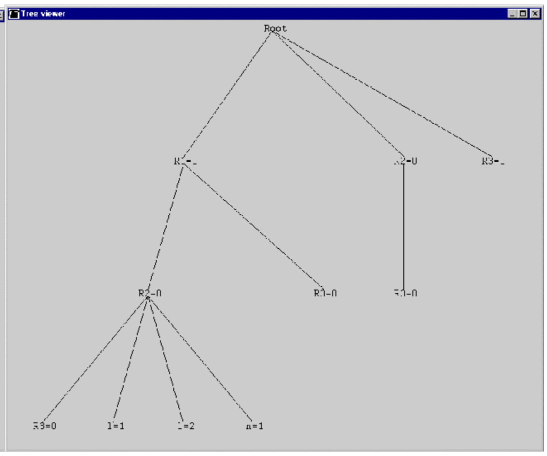
On modélise le problème par 10 lettres (D O N A L G E R B T) dans 0..9 et 5 retenues (R1 R2 R3 R4 R5) dans 0..1.

(0) AllDifférent(D,O,N,A,L,G,E,R,B,T) (1) D+D=T+10.R1
(2) L+L+R1=R+10.R2
(3) A+A+R2=E+10.R3 (4) N+R+R3=B+10.R4 (5) O+E+R4=O+10.R5
(6) D+G+R5=R (7)D $\geq$ 1 (8)R $\geq$ 1 (9) G $\geq$ 1

Heuristiques	Borne-Cohérence	Composition,Réécriture,Borne-Cohérence
MinSize, MinValue	11bt,12choix	9bt,10choix
MinSize, MaxValue	11bt,12choix	11bt, 13 choix
MostConstrained, MinValue	12bt,13chix	18bt,19choix
GreatestRegret, MinValue	27bt,28choix	9bt, 10choix



MinSize&MinValue, sans composition



MinSize&MinValue, avec composition

Avec l'heuristique MinSize&MinValue, la première composition utile est 2.G-2.R+T=0 , 2.L+R=0 ➔ 2.G-4.L+T=0 ; elle réduit le domaine de G de (1..4) (6..7) à 1..4 et instancie R3 avec 1, R4 et R5 avec 0. Cette composition a lieu alors que deux choix ont déjà été faits R1=0 et R2=0. Ces deux choix s'avèrent mauvais et la composition permet de le détecter sans autre énumération, ce qui n'est pas le cas avec la borne-cohérence seule. Dans le contexte où R1=1 et R2=0, une autre composition -2.D+T=-10 et -D-G+R-R5=0 ➔ 2.G-2.R+2.R5+T=-10 permet de réduire les domaines : L [(1..9) => (3..9)], G [(1..8) => (1..4)], R [(2..9) => (6..9)]. Ensuite le filtrage des contraintes ajoutées suffit pour détecter l'incohérence du choix R3=0. Sans composition, R5 n'est pas instanciée et un choix est nécessaire sur R4 ou R5 pour invalider le choix R3=0.

Conclusions

On constate que RCS est utile pour détecter les états incohérents plus tôt. En revanche, le reste de la résolution est le même avec ou sans composition linéaire ; l'ordre des choix n'est pas changé et les compositions n'améliorent pas le filtrage. Cette expérience met donc en évidence l'existence de seuil de densités au delà desquels l'ajout de contraintes redondantes est inefficace.

1.1.3 Le problème ALPHA

Une description du problème se trouve dans la thèse de Pierre Roy (partie 2, chapitre I, paragraphe 1.2). Le problème contient 20 égalités linéaires et une contrainte de différence globale. Chaque contrainte exprime que la somme des lettres d'un mot vaut une constante.

ballet = 45	glee = 66
cello = 43	jazz = 58
concert = 74	lyre = 47
flute = 30	oboe = 53
fugue = 50	opera = 65
polka = 59	song = 61
quartet = 50	soprano = 82
saxophone = 134	theme = 72
scale = 51	violin = 100
solo = 37	waltz = 34.

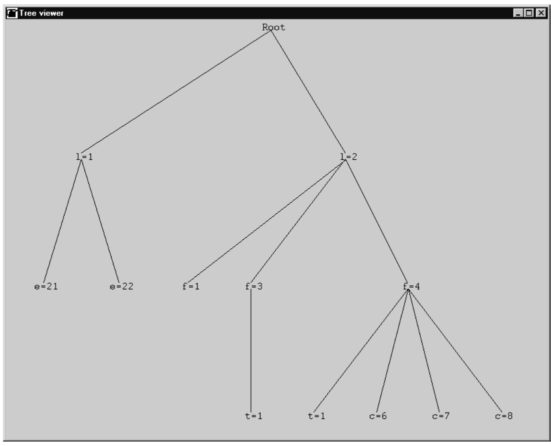
Le problème n'a qu'une solution[23]. L'introduction de la réécriture d'instantiation ne change pas la résolution, quelle que soit l'heuristique. En ajoutant la composition linéaire, le nombre de retour-arrières se réduit jusqu'à 6 et le nombre de points de choix à 8. Le calcul des compositions conserve les termes contenant moins de variables que le nombre maximal de variables dans les contraintes initiales.

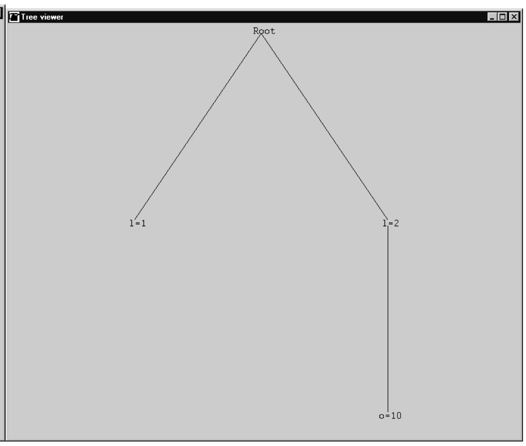
Heuristiques	Borne-C	BC,Réécriture	Composition,Réécriture,BC
MinSizeMinValue	10bt,12 choix 5 ptCh(l,e,f,t)	10bt,12 choix	1bt,3choix, (linlin) 2 ptCh (l,o)
MostConstrained MinValue	10bt,13choix	10bt,13choix	10bt,13 choix
GreatestRegret MinValue	>6492bt	Exponentiel	Exponentiel
MinSizeMaxValue	13bt,16choix 6 ptCh(e,u,g,u,g,o)	13bt, 16 choix	7bt,9choix (linlin) 2 ptCh(e,u)

Tableau 7: comparaisons de résolution pour trouver la solution.[24]

[23] (a=5 b=13 c=9 d=16 e=20 f=4 g=24 h=21 i=25 j=17 k=23 l=2 m=8 n=12 o=10 p=19 q=7 r=11 s=15 t=3 u=1 v=26 w=6 x=22 y=14 z=18)

[24] *ptCh* signifie « point de choix ».





Sans composition, avec MinSize&MinValue   *Avec composition et MinSize&MinValue*

Dans ce problème la composition ne permet pas de déterminer directement la valeur de variable. En revanche les contraintes ajoutées influencent le nombre de points de choix. Avec minSize&MinValue, l'incohérence du premier choix est détectée sans énumération supplémentaire du problème, ceci grâce à l'introduction de contraintes linéaires avant le premier point de choix. Le dernier point de choix (O=10) est correct et un filtrage des contraintes permet de trouver les inconnues restantes.

Conclusions

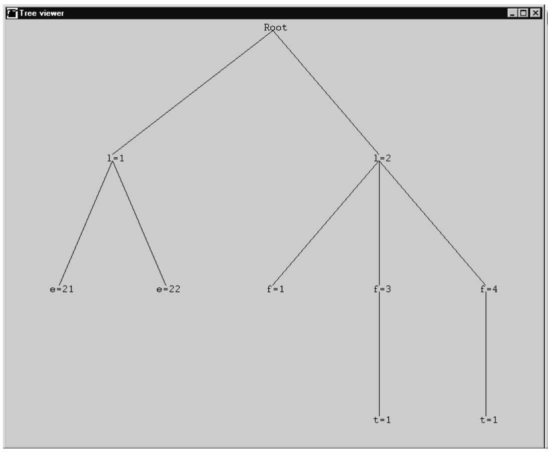
La composition a pour effet de contraindre davantage certaines variables, entraînant potentiellement une réduction supplémentaire de leur domaine. Cette réduction est d'autant plus avantageuse qu'elle a lieu avant le premier choix. Si l'on était capable au moyen de métriques de prévoir à l'avance si un type de composition est intéressant, cela permettrait d'éviter des calculs inutiles et coûteux.

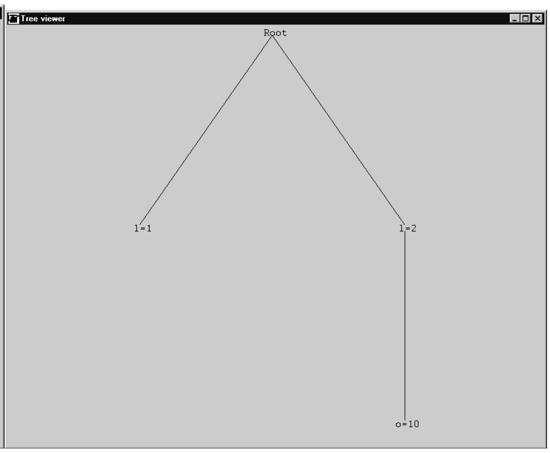
1.1.4 Le problème BETA

A partir du problème ALPHA, on construit le problème BETA en retirant deux contraintes (ballet = 45, oboe = 53) et ajoutant quatre contraintes (band = 46, blues = 54, doublebass = 113, tuba = 25). Le problème est plus contraint que ALPHA[25].

Heuristiques	Borne-Cohérence	B-C,Réécriture	Composition,Réécriture,B-C
MinSize MinValue	7bt,9 choix 4 ptCh (l,e,f,t) 42bt,2choix (unicité)	7bt,9 choix 42bt, 42 choix	1bt,3choix, (linlin) 2 ptCh (l,o) 23bt,23choix (unicité) détection d'incohérence plus tôt
MostConstrained MinValue	103bt,106 choix 194bt,194choix (unicité)	103bt,106choix 194bt,194choix	76bt,79 choix 146bt,146choix
MinSize MaxValue	29bt,31choix 13 ptCh 48bt,48choix (unicité)	29bt, 31 choix 48bt,48choix	26bt,28choix 5 ptCh (e,g,u,g,e,u) 45bt,45choix

[25] Là encore, il y a une unique solution évaluant toutes les lettres de l'alphabet par ordre lexicographique (a=5 b=16 c=9 d=13 e=20 f=4 g=24 h=21 i=25 j=17 k=23 l=2 m=8 n=12 o=10 p=19 q=7 r=11 s=15 t=3 u=1 v=26 w=6 x=22 y=14 z=18).





Sans composition avec MinSize&MinValue *Avec composition et MinSize&MinValue*

Comparons les deux résolutions ci-dessus. Le premier point de choix est identique (L). Après celui-ci, la composition linéaire change l'ordre des variables calculé par l'heuristique choisie. C'est pourquoi le choix suivant diffère : U=2 au lieu de E=21. Cela s'avère être un meilleur choix car lors du retour-arrière de U=2, le système s'aperçoit par un simple filtrage qu'il n'y a pas de solution pour U>=3 et L=1. Lorsque L est instanciée avec 2, les compositions dans ce nouveau contexte provoquent la réduction du domaine de U telle qu'elle devient la variable de plus petit domaine. Après le choix U=1, d'autres compositions sont faites et le filtrage des contraintes générées réduit le domaine de E à [19..20]. E est alors instanciée successivement avec 19 et 20. On obtient la solution en 4 points de choix soit un de moins par rapport à une résolution CSP standard.

Conclusions

Nous constatons que BETA est un exemple de problème vérifiant l'idée selon laquelle un problème plus contraint est plus facile à résoudre. L'instantiation de seulement 2 variables (L=2 et U=1) sur 26 a permis de composer les contraintes de manière à ce que l'énumération d'une seule variable suffise pour trouver la solution.

1.1.5 Contraintes non linéaires : cas des multiplications cryptées

Dans le cas des multiplications cryptées, la réécriture unaire permet de reformuler le problème en appliquant les lois de distributivité et les simplifications systématiques des expressions arithmétiques. Ainsi la règle $(X+Y)*Z \rightarrow (X*Z)+(Y*Z)$ transforme un produit en somme linéaire de deux variables contraintes par des contraintes primitives de produit binaire : T_1+T_2, où $T_1 = X*Z$ et $T_2 = Y*Z$.

*Exemple : XA*X = BXC*

Le problème est modélisé par trois contraintes :

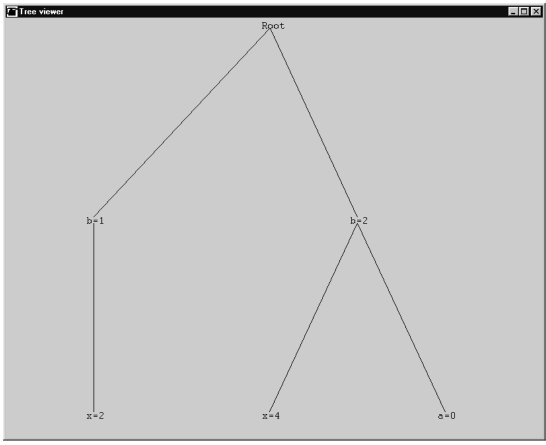
(1) $(10*X + A) * X = T_1$ (2) $100*B + 10*X + C = T_2$ (3) $T_1 = T_2$

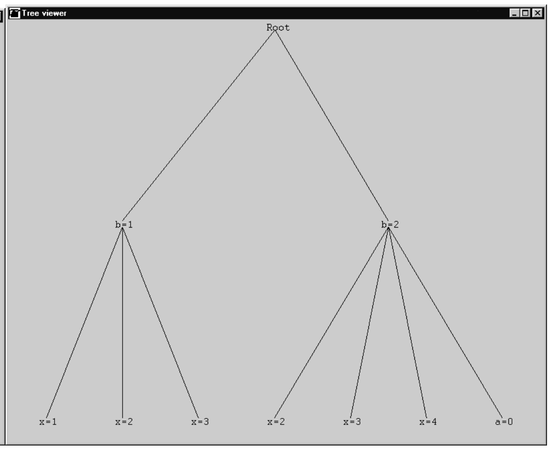
Dès sa création, la contrainte (1) est reformulée en la conjonction de trois contraintes (4) $10*X*X = U_1$, (5) $A*X = U_2$ et (6) $U_1+U_2 = T_1$. Toute contrainte étant mise en forme régulière dès sont ajout dans le problème, la contrainte (4) est simplifiée en $10*X^2 = U_1$. Le filtrage de la nouvelle contrainte, binaire réduit davantage le domaine de X que celui de la contrainte ternaire (4). Cela est dû au fait que l'algorithme de cohérence d'une contrainte binaire ne suppose pas de lien entre ses opérandes et n'exploite donc pas le fait que les deux opérandes du produit sont identiques.; qualitativement la contrainte est équivalente à {Y = X*Z , Z = X}.

Résolution	Borne-Cohérence	Composition,Réécriture,Borne-Cohérence
1ère solution	5bt,7choix	1bt,3choix
2ème solution	6bt,7choix	2bt,5choix
3ème solution	9bt,11choix	3bt,5choix
Toutes les solutions	22bt,22choix	9bt,9choix

Tableau 8 : Résultats avec l'heuristique minSize&MinValue

La composition linéaire n'intervient pas. Seule la réécriture unaire est donc responsable de la réduction du nombre de retour-arrières et de points de choix (un seul pour la première solution et trois pour toutes les solutions).





Avec réécriture $x*x->x^2$ et distributivité *Sans réécriture*

La contrainte sur X^2 réduit le domaine de X à {2,3}, lorsque B=1, ce qui évite le retour-arrière de X=3. Quand B=2, le domaine de X est réduit à [4..5] alors qu'il reste à [3..9] si la contrainte sur X*X n'est pas simplifiée ; cela permet d'éviter un mauvais choix sur X (X=3).

Conclusion

La réécriture unaire permet de reformuler le problème avant sa résolution de manière à faire apparaître des égalités linéaires. Nous constatons que la présence d'égalité linéaire permet de filtrer davantage les domaines par rapport aux contraintes primitives de produit. De plus l'utilisation de réécriture produisant des contraintes plus complexes comme la règle de distributivité est compensée par la simplification systématique des contraintes dès leur ajout dans le problème.

1.2 Composition de contraintes à effet retardé : le problème des carrés magiques

La seconde étude porte sur les carrés magiques. Elle met en évidence la nécessité de contrôler finement RCS. Le problème des carrés magiques est difficile car les instances sont peu denses : deux contraintes ont au plus une variable en commun, ce qui limite la propagation au parcours d'un chemin sur l'hyper-graphe de contraintes. Sur cette catégorie de problèmes, les compositions linéaires permettent au plus de supprimer une variable commune, engendrant alors des contraintes redondantes d'arité plus grande que les contraintes d'origine. L'effet peut s'avérer positif car le problème devient plus dense d'où une propagation de contraintes plus efficace. Néanmoins nous avons constaté que les compositions linéaires fonctionnent mal car elles ajoutent beaucoup de contraintes qui ne regroupent pas d'instantiation de variables. Il est alors nécessaire d'imposer une stratégie de composition des contraintes linéaires, pour limiter le nombre de contraintes ajoutées, tout en préservant la déduction des contraintes redondantes utiles (*i.e.* celles qui réduisent le domaine).

Analyse

Le problème contient pour un côté de N cases, 2*N+2 contraintes et N*N variables. Chaque contrainte est N-aire.

Une première solution est de rajouter une contrainte exprimant le fait que l'on cherche en réalité une bijection : la somme de toutes les variables est égale à ½(N*(N+1)) (somme des N premiers nombres). L'ajout de cette contrainte redondante (appelée BijC) peut être automatisé lors d'un pré-traitement du problème (Cf. Chapitre III-3.4.1). Chaque contrainte initiale se combine avec la contrainte redondante, celle-ci contenant toutes les variables, et crée au moins une contrainte redondante plus petite qu'au moins une des contraintes sources.

Cependant cette solution pose un problème car la complexité de la composition linéaire croit de manière exponentielle : pour chaque contrainte initiale, il y a N compositions possibles soit au total N*(2N+2). De plus chaque contrainte ajoutée contient au moins N*(N-1) variables et peut se combiner avec au plus N-1 contraintes du problème initial, distinctes. De plus on observe un effet retardé des contraintes redondantes qui provoquent une réduction de domaine quand elles sont en assez grand nombre dans le problème. En pratique avec cette solution, la taille du problème explose avant que l'on obtienne une réduction de domaine significative.

Nous étudions la résolution du problème pour N=3, défini comme sur la grille suivante :

Bt 1	Bt 2	Bt 3
Bt 4	Bt 5	Bt 6
Bt 7	Bt 8	Bt 9

<u>Contraintes</u> : L1 : Bt1 + Bt2 + Bt3 = S,…, C1 : Bt1 + Bt4 + Bt7 = σ,…,
D1 : Bt1 + Bt5 + Bt9 = σ, D2 : Bt3 + Bt5 + Bt7 = σ.

1.2.1 Problème simple où la somme est connue

Ce problème est facile car σ peut être évaluée à partir de la contrainte de bijection en substituant les sommes de L1, L2, L3 par S, soit σ+σ+σ = 45, réécrit en σ = 15. Un solveur CSP[26] résout le nouveau problème en au minimum 1 retour-arrière et 3 choix (2 points de choix Bt1 et Bt8), avec l'heuristique MinSize&MinValue. BackTalk choisit comme premier point de choix, le coin à gauche (Bt1). Au deuxième choix (Bt1 = 2), après filtrage, Bt8 a la plus petite borne minimale parmi les variables de plus petit domaine ; elle est donc choisie comme second point de choix. Après son affectation avec 1, le filtrage permet de trouver directement la solution.

[26] Ces expériences ont été réalisées avec BackTalk et l'heuristique minSize&MinValue.

Bt 1 [(1..9)] → Bt 1 int[1]   erreur lors du filtrage ➔ retour-arrière
Bt 1 [(2..9)] → Bt 1 int[2]   filtrage valide
Bt 2 [1 (3..5)] → Bt 8 int[1]

Figure 7 : l'énumération de la variable Bt1 (2 choix) implique Bt8 comme second point de choix.

Dans le meilleur des cas, le problème initial se résout en 2 retour-arrières et 4 choix par un solveur CSP. La reformulation du problème permet donc globalement d'éviter un retour-arrière.

La place des compositions dans la résolution

Toutes les solutions ont pour valeur 5 dans la case du milieu. Les compositions permettent de retrouver ce résultat et d'instantier Bt5 avec 5 sans faire de choix, après 194 compositions. Le problème contient à ce moment 38 contraintes. La trace d'exécution affiche les réductions de domaine jusqu'à la détermination de Bt5 :

```
'Bt 1 [(1..9)] ==>  Bt 1 int[(1..4) (6..9)]'
'Bt 2 [(1..9)] ==>  Bt 2 int[(1..4) (6..9)]'
'Bt 3 [(1..9)] ==>  Bt 3 int[(1..4) (6..9)]'
'Bt 4 [(1..9)] ==>  Bt 4 int[(1..4) (6..9)]'
'Bt 5 [(1..9)] ==>  Bt 5 int[5]'
'Bt 6 [(1..9)] ==>  Bt 6 int[(1..4) (6..9)]'
'Bt 7 [(1..9)] ==>  Bt 7 int[(1..4) (6..9)]'
'Bt 8 [(1..9)] ==>  Bt 8 int[(1..4) (6..9)]'
'Bt 9 [(1..9)] ==>  Bt 9 int[(1..4) (6..9)]'
Csaturate Bt1 + Bt5 + Bt9 =15 et Bt1 - 2.Bt5 + Bt9 =0 ➔ Bt5 =5
'Magic Square 3: nbContraintes : 38'
```

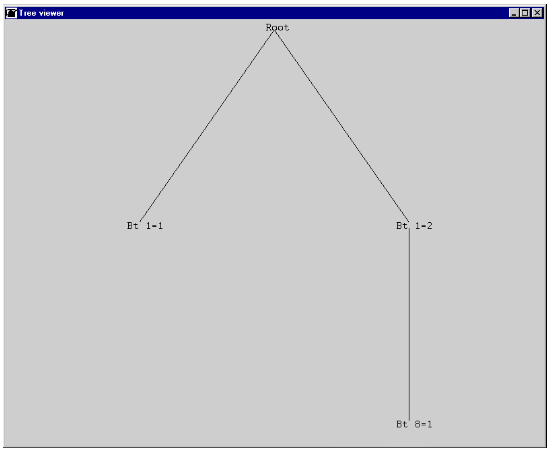
Le problème est réduit à 17 contraintes après la réécriture d'instantiation de Bt5. Une fois Bt5 instantiée, la résolution reste similaire à celle d'un solveur CSP car les variables ont des domaines identiques. Deux choix sont faits sur Bt1, en accord avec l'heuristique minSize et ordre lexicographique ; Bt1 = 1 est incorrect tandis que Bt1 = 2 provoque la réécriture de toutes les contraintes et leur composition (73 compositions). Le filtrage qui suit immédiatement ces compositions provoque une réduction de domaine :

```
'Bt 6 [1 (3..5)] ==>  Bt 6 int[1 3]'
'Bt 8 [1 (3..5)] ==>  Bt 8 int[1 3]'
Csaturate Bt4+Bt7 =13 et Bt4+Bt6+Bt7+Bt8 =17 ➔ Bt6+Bt8 = 4
'Magic Square 3: nbContraintes : 17'
```

Après le choix Bt8 = 1, un filtrage suffit pour trouver la première solution. Cependant le processus de composition linéaire étant systématique, 9 compositions sont réalisées après le choix sur Bt8, inutilement. Au total, 276 compositions sont réalisées ; la taille mémoire augmente d'un facteur 3,8.[27]

CM 3x3	σ connue	σ inconnue
CSP	0.061s 2bt 4ch 2 ptCh (Bt1, Bt4)	0.453s 50bt 53ch 13 ptCh
CSP+Composition-saturate	5.829s 1bt 3ch 2 ptCh (Bt1, Bt8)	14.155s 12bt 15ch 2 ptCh (Bt1, S)

[27] Temps réalisés sur un pentium PII 64 MoRam. Mis à titre indicatif.



Résolution si la somme est connue avec MinSize

Conclusion

Dans cette catégorie de problèmes, ni la réécriture, ni la composition de contraintes n'améliore pas la résolution et ne l'aggrave pas non plus. En revanche, le nombre de compositions réalisées est très grand et l'ensemble des contraintes croit de manière rédhibitoire puisque le temps mis pour résoudre le problème est multiplié par 95,557. Dans le cas où la somme est connue, l'intérêt de composer les contraintes est limité ; car le problème est trop simple pour que les techniques de CSP rencontrent des difficultés à conclure.

1.2.2 Problème compliqué où la somme n'est pas connue

Dans le cas ou la somme σ n'est pas connue, on obtient paradoxalement un problème plus dense car toutes les contraintes ont au moins une variable en commun (la variable qui justement représente la somme inconnue). Par ailleurs, la détermination de Bt5 est cruciale car elle provoque par filtrage l'affectation de S avec 15 et permet de se ramener au cas précédent. La Figure 8 illustre la combinatoire de la résolution dans le cas où le solveur CSP ne choisit pas Bt5 comme point de choix.

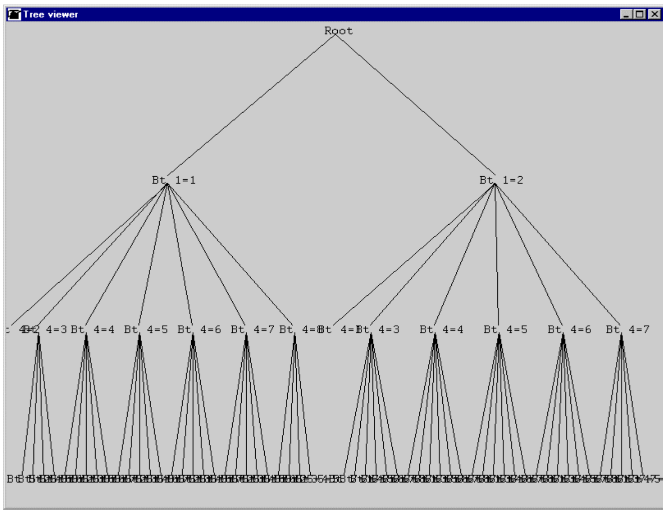


Figure 8: arbre des choix parcouru par un solveur CSP avec l'heuristique minSize

Nous avons montré dans la section 2.2.1 que la valeur de peut être déterminée formellement par composition linéaire ; elle ne peut être obtenue par les seules méthodes générales de cohérence. Les techniques de raisonnement symbolique permettent de ramener le problème à un problème connu simple, avant le premier point de choix. Cela nous a amené a étudier un sous-problème contenant la contrainte [Bt5=5] (C.f Tableau 9).

Problèmes	Borne-Cohérence	Composition,Réécriture,Borne-Cohérence
Sans Bt5 = 5	50 bt, 53 choix 13 ptCh	19 bt, 22 choix 4 ptCh (Bt1,Bt2,Bt4,Bt6)
Avec Bt5 = 5	12 bt, 13 choix 3 ptCh (Bt1, Bt4, Bt7)	1 bt, 3 choix 1 ptCh(Bt1)

Tableau 9 : première solution du carré magique 3x3 avec minSize&minValue.

Nous avons constaté expérimentalement que le résultat de CSP+RCS lorsque Bt5 = 5 (ligne 2, colonne 3) est vrai pour toutes les heuristiques décrites en annexe. Par conséquent les compositions linéaires réalisées après la détermination de la valeur de Bt5 n'apportent pas d'amélioration supplémentaire.

L'utilité des compositions dans la résolution

Pour une amélioration en qualité, le but est que le temps mis pour trouver Bt5 ne soit pas plus grand que le temps mis pour faire les mauvais choix avant l'instantiation de Bt5, dans la résolution par CSP. Dans son livre, J. Pitrat décrit la suite minimale de compositions linéaires (seulement quatre) aboutissant à la valeur de Bt5. Dans le système automatisé CSP+RCS, les compositions linéaires sont plus nombreuses car réalisées par un mécanisme systématique avec pour seul contrôle le fait d'autoriser les nouvelles contraintes plus simples que l'une au moins des contraintes sources. Après 50 compositions, la valeur de Bt5 est trouvée grâce à la composition :

```
Bt3 + Bt5 + Bt7 = 15 (-) Bt3 - 2.Bt5 + Bt7 = 0 ➔ Bt5 = 5
```

Ces compositions ne sont possibles que si la contrainte redondante BijC est présente. Dans la suite de la résolution, les compositions sont moins nombreuses et n'impliquent plus BijC car les contraintes sont de plus en plus simples. Cependant la taille du problème est multipliée par 5 ! Au chapitre IV-1.2 nous montrons que l'on peut contrôler les compositions linéaires de manière à ce que le problème ne dépasse jamais plus de 21 contraintes (*i.e.* le double de la taille du problème d'origine). L'idée est d'imposer une stratégie dynamique sur la sélection des compositions réalisées : toutes les meilleures compositions quand au moins une des contraintes est binaire et une seule des meilleures compositions quand les deux contraintes ont une arité supérieure à deux. Malheureusement avec ce contrôle, RCS ne trouve plus la valeur de Bt5 et des choix doivent être faits pour se ramener au problème simple. Cette stratégie est néanmoins satisfaisante puisque le temps mis par BackTalk+RCS est identique voir plus petit que celui de BackTalk : le temps de manipulation des contraintes compense le temps d'annulation des mauvais choix (restauration des domaines de valeurs).

Conclusions

Dans son livre [Pitrat 1993], J. Pitrat montre une résolution exclusivement formelle basée sur des compositions linéaires, tout en faisant appel à des heuristiques humaines et non systématiques pour choisir les contraintes et coefficients de composition. La composition systématique de contraintes linéaires avec une stratégie fixe permet d'aboutir au même résultat mais au prix d'un accroissement de la taille du problème et d'un temps de calcul long. De plus beaucoup de contraintes ne servent pas à une réduction de domaine supplémentaire par rapport à l'état du problème avant la composition. Finalement cette expérience montre les intérêts et limites de la composition binaire et fournit une motivation pour construire des outils d'affinage de la composition, de construction de stratégie.

1.3 Lien qualitatif entre cohérence globale et combinaison de contraintes linéaires

Si les études des propriétés de la cohérence locale ne manquent pas [Güsgen & Hertzberg 1988], [Van Beek 1994], il est plus difficile de trouver des propriétés sur l'action du raisonnement symbolique ; car celui-ci porte sur un ensemble de contraintes en général non binaires. Nous avons constaté des similarités expérimentales entre les algorithmes de maintien de cohérence globale sur des contraintes arithmétiques et la combinaison linéaire de contraintes d'égalité et d'inégalité. Ces deux techniques manipulent plusieurs contraintes en même temps afin de maintenir le problème dans un état cohérent.

Reprenons l'expérience décrite au chapitre I-2.2.1 avec le problème $\{X^2-Y^2=0 ; X \neq Y; X \text{ in } 1..2 ; Y \text{ in } 1..2\}$. Le tableau ci-dessous compare les résultats de différentes méthodes de résolution.

Méthodes	Résultats avant le premier choix d'instanciation
Cohérence de bornes, cohérence d'arc	X in 1..2 ; Y in 1..2
Filtrage partiel sur 2 contraintes (2C-cohérence)	Incohérence détectée car X et Y sont positifs
R.F. + cohérence de bornes = Réécriture : X2-Y2=0 → (X+Y)*(X-Y) = 0 Borne-cohérence et réécriture : X = Y Composition : X = Y, X ≠ Y → False.	Déduction d'une contrainte redondante X = Y. Incohérence détectée par composition de deux contraintes.

Tableau 10: raisonnement symolique et filtrage partiel de chemin de 2 contraintes.

L'effet sur le domaine est identique dans les cas 2 et 3. La 2C-cohérence s'applique systématiquement sur tous les couples de contraintes du problème. Au contraire, la composition de contraintes (ligne 3 du Tableau 10) ne considère que certains couples de contraintes. Pour les contraintes arithmétiques manipulées, l'expérience montre que la composition linéaire produit un résultat comparable voire identique à celui de la 2C-cohérence. La détermination formelle de la catégorie de contraintes concernées par la propriété entre dans le cadre des perspectives de ce travail. D'une manière informelle, on peut voir l'utilisation conjointe du raisonnement symbolique et de la cohérence locale comme une cohérence partielle de chemin sur une partie du problème.

Propriété expérimentale :

> Dans le cas particulier des opérateurs relationnels binaires (=, ≠, >), la composition des contraintes deux à deux revient à une cohérence sur les chemins de deux contraintes basée sur l'arithmétique des intervalles, la 2C-cohérence (C.f. chapitre I). En particulier, les deux méthodes détectent l'incohérence des mêmes situations.

La chemin-cohérence dans un CSP non binaire correspondrait à la vérification de la cohérence de tous les hyper-chemins, que nous appelons *cohérence d'hyper-chemin*. Dans le cas particulier d'un CSP binaire, la cohérence d'hyper-chemin est équivalente à la chemin-cohérence (terme réservé aux CSP binaires). La chemin-cohérence implique l'ajout systématique d'autant de contraintes intermédiaires qu'il y a de couples de variables reliées par un chemin dans le problème. Pour deux contraintes d'arité `m` et `n`, pas moins de `(m-p)*(n-p)` contraintes binaires sont ajoutées (`p` est le nombre de variables communes aux deux contraintes). De la même manière *RCS* implique l'ajout de contraintes, et la cohérence d'hyper-chemin ajouterait les contraintes intermédiaires tant que cela ne provoque pas d'incohérence ou qu'une solution n'est pas trouvée. Cependant, au contraire de la cohérence d'hyper-chemin, *RCS* contrôle le nombre de nouvelles contraintes.

Un filtrage par 3B-cohérence propagerait des événements sur une variable, en parcourant tous les hyper-chemins de longueur 2 qui atteignent cette variable. Dans le cas de la cohérence d'hyper-chemins de longueur 2, tous les chemins de la même longueur sont visités à chaque tour.

Le traitement de plusieurs contraintes permet d'affiner la propagation des valeurs des domaines. Par exemple, on peut pour certaines contraintes, parcourir les chemins sur les contraintes adjacentes et pour d'autres ne tester la cohérence que sur un hyper-arc. Le choix du type de cohérence dépend du type de la contrainte, de sa complexité, éventuellement de l'événement traité. Ainsi le système BackTalk+RCS autorise le choix des stratégies de composition, par exemple n'appliquer les règles considérées comme coûteuses, lorsque les autres règles (plus simples) ont échoué deux fois.

2 Conclusion

Les résultats présentés montrent l'existence de situations où 1) les techniques CSP sont inadaptées, 2) le couplage avec une suite de transformations symboliques du problème permet de trouver le résultat plus directement. Celles-ci déduisent les valeurs d'un certain nombre de variables avant le premier choix, évitant ainsi d'explorer des mauvaises branches de l'arbre de recherche. Cette amélioration de la qualité de la résolution n'est pas gratuite car l'application de la réécriture et la création de contraintes apportent un coût supplémentaire.

2.1 De l'adaptation de l'algorithme de résolution aux spécifités du problème

La combinaison de plusieurs méthodes non forcément homogènes, comme la réécriture de contraintes et les techniques de CSP, nous a amenés à nous poser les questions suivantes :

- Quel est le champ d'application des méthodes : des parties distinctes du problème, des parties ayant une intersection, la totalité du problème ?
- Comment choisir les méthodes dans une situation donnée ? Cette décision résulte d'une analyse nécessairement *a posteriori* car la stratégie de combinaison se veut la plus générale possible et se base sur les caractéristiques préalablement définies d'un état de problème. Cette analyse porte sur des connaissances résultant des choix précédents, *i.e.* de l'évaluation de l'utilité des méthodes dans les situations précédemment survenues.

- Quelles sont les actions possibles pour contrôler l'application des méthodes ? En général, une unité de contrôle est dédiée à cette tâche qui consiste à bloquer ou activer une méthode. Le blocage est valide jusqu'à validation d'une condition sur une partie du problème[28], tandis que la méthode s'exécute à différents niveaux de modification du problème[29].

D'un point de vue formel, on peut voir une contrainte comme l'abstraction d'un algorithme de cohérence. De même, une contrainte redondante avec un ensemble de contraintes E peut être vue comme l'abstraction d'une procédure de filtrage telle que toutes les solutions trouvées sont également trouvées par la propagation des contraintes de E. Nous identifions une équivalence entre l'introduction d'une contrainte et la génération d'une nouvelle procédure de calcul. Plus précisément, si les contraintes sont remplacées par des contraintes de même classe, la méthode de propagation ne change pas, seules ses paramètres changent ; si les nouvelles contraintes sont de type différent, la propagation change car une des méthodes de filtrage est différente ; si les anciennes contraintes sont conservées, la propagation est enrichie avec les méthodes de filtrage des nouvelles contraintes. L'absence de conflit entre les nouvelles procédures et les procédures existantes est garantie par le fait que seules des contraintes redondantes sont ajoutées.

2.2 Amélioration qualitative du maintien de cohérence

Nous proposons de combiner des transformations symboliques de manière à ce que l'algorithme de maintien de cohérence s'adapte davantage à l'état du problème. L'algorithme de résolution utilisant la satisfaction de contraintes bénéficie de cette amélioration. En effet la satisfaction de contraintes applique systématiquement sur le problème entier, soit un algorithme coûteux et efficace soit un algorithme rapide mais moins efficace. Nous partons de l'idée suggérée par Alice, qui consiste à appliquer l'algorithme rapide sur la totalité du problème, et parfois, l'algorithme coûteux sur un sous-problème que l'on sait caractériser à l'avance ou de manière automatique. Un des buts de notre travail est de proposer un cadre de modélidation de techniques symboliques sur les contraintes et de représentation de critères d'application de ces techniques.

Les expériences de ce chapitre et les travaux de [Granvilliers 1998a], [Granvilliers 1999] prouvent que la diminution de l'arité des contraintes arithmétiques apporte des améliorations dans la cohérence des domaines, lorsqu'elles contiennent plusieurs occurrences d'une même variable. Néanmoins, techniquement, l'utilisation de la simplification de contraintes requiert des mécanismes non gratuits en temps et espace ; par exemple la simplification suppose que les occurrences d'une même variable soient représentées par le même objet physique.

Comme le souligne J.Pitrat dans [Laurière 1996], «le traitement formel des contraintes ralentit considérablement ». Le problème est donc d'expliciter des informations sur les situations où les manipulations symboliques sont intéressantes. L'expérience montre que ces informations dépendent des contraintes déjà présentes dans le problème. Par exemple de l'équivalence $X^2-Y^2 = S_{xy}.D_{xy}$, on peut déduire deux règles selon que l'on développe ou

[28] Par exemple, une classe de contraintes, ou une variable et ses contraintes.

[29] La méthode est différente selon le niveau de cohérence de la contrainte (celui-ci pouvant a priori changer en cours de résolution), la sélection des instances de règles peut prendre en compte des modifications antérieures des contraintes.

factorise la contrainte. Dans l'exemple de E. Davis (chapitre I), la factorisation de l'expression X^2-Y^2 a permis de conclure parce qu'il y avait une contrainte contenant X et Y dans le reste du problème. Au contraire dans l'exemple XA*X = BXC (chapitre I), c'est un développement qui est utile car il n'y a pas d'autre contrainte pour une éventuelle composition. Alors comment choisir ? Un compromis pourrait être de factoriser tous les produits binaires et, lorsque l'un des facteurs contient une seule variable, de développer en somme. Ce critère est valable sur les exemples que nous avons testés mais reste à adapter à de nouvelles situations. Il faut donc définir une structure qui rende cette édition facilement accessible à l'utilisateur.

L'idée est aussi de faire un traitement en plus avant un choix de manière à supprimer des domaines les valeurs impossibles qu ele filtrage ne détecte pas, et ainsi éviter les mauvais choix. Quand un mauvais choix est fait, il faut restaurer les domaines et toutes les données du contexte précédent. Quand en plus une transformation intervient avant le choix, il faut restaurer l'ensemble des contraintes. Dans le cas idéal, le temps mis pour faire le traitement symbolique compenserait le temps mis pour annuler les mauvais choix. Pour des problèmes numériques cela arrive rarement car les techniques CSP contrôlées par des heuristiques adaptées sont déjà très rapides. Néanmoins, pour les problèmes exprimés sur des ensembles d'objets, les techniques d'énumération-propagation sont difficilement optimisables car le domaine n'est pas ordonné *a priori*. Or le temps mis par la déduction de contraintes reste dans le même ordre de grandeur, car il dépend essentiellement de la structure de la contrainte.

2.3 Proposition

A partir de ces expériences, nous avons intégré dans le *framework* BackTalk un mécanisme de manipulation symbolique de contraintes fondé sur la réécriture, appelé *RCS*. De plus nous avons intégré un module de contrôle dynamique du déclenchement de ces réécritures, appelé *méta-RCS*, permettant de définir de manière déclarative des stratégies dynamiques à partir des états définis pour le solveur symbolique RCS (Chapitre III). BackTalk est alors considéré comme le support d'une collaboration entre plusieurs solveurs, ce qui nous a permis d'étudier au chapitre IV, l'expression de critères décrivant les situations favorables aux règles de RCS.

Chapitre III : Raisonnement symbolique et contrôle en RCS

La chasse a passionné l'humanité depuis toujours, et avec juste raison. Ce que beaucoup n'ont pas compris, cependant, c'est que la chasse peut être exaltante même si la chose poursuivie n'est qu'une idée, qu'un concept, une théorie. A mesure que s'est développée la conscience, il est devenu évident que c'est de toutes, la poursuite la plus importante, celle dont l'issue conditionne la survie ou l'échec de toute l'humanité. (Raja Lon Flatterie, Le Livre de la Nef)

Franck Herbert, Destination : vide, R. Laffont, chapitre 10, 1978

RCS est défini par l'ensemble des solveurs symboliques qui visent à transformer un problème combinatoire en un problème équivalent dont la résolution ou la solution obtenue est plus simple ou de meilleure qualité. La complexité d'une procédure de résolution peut être évaluée de deux manières : 1) qualitativement, étant données des métriques sur les connaissances manipulées par la procédure (nombre de contraintes ou variables visitées à chaque étape, nombre de n-uplets supprimées de l'espace de recherche,…), on met à jour ces mesures à chaque étape ; 2) quantitativement, on mesure globalement le temps mis pour résoudre le problème. La simplicité d'un ensemble de contraintes est caractérisée par la qualité d'exécution de la procédure de résolution considérée. Par exemple, dans un un ensemble de contraintes symboliques, la simplicité mesure le nombre de pas de réécriture necéssaire pour résoudre les contraintes par transformation syntaxique. La simplicité est également fonction du type de contrainte. Par exemple, une contrainte arithmétique est d'autant plus facile à propager que son arité est petite (cas trivial des contraintes unaires) ; la résolution sera d'autant meilleure que le nombre de retour-arrières ou de points de choix sera faible.

Un solveur symbolique est un système computationnel déduisant d'un ensemble de contraintes, un nouvel ensemble équivalent mais plus simple. Nous considérons deux manières de construire cet ensemble à partir des contraintes existantes, soit en remplaçant des contraintes, soit en produisant de nouvelles contraintes par interprétation de la sémantique des contraintes existantes. Un solveur symbolique peut alors être représenté dans le formalisme des règles de réécriture conditionnelles multi-prémisses. Cette représentation uniforme nous permet de construire une architecture de modélisation de stratégies permettant d'exploiter les informations sur l'exécution des méthodes de cohérence et de manipulation de contraintes elles-mêmes. Ces dernières sont nécessaires pour éviter une explosion combinatoire ou une action dégradant la résolution. En effet les techniques symboliques sont un moyen de traiter plusieurs contraintes ensemble ; néanmoins cet ensemble ne peut être défini *a priori* car le problème change. Il est donc défini de manière abstraite par les prémisses des règles. Les contraintes pouvant être traitées de plusieurs manières, l'intersection des ensembles abstraits n'est en général pas vide.

Dans ce chapitre, nous analysons tout d'abord la sémantique et les conséquences du raisonnement symbolique dans le cadre de la résolution d'un CSP. Les sections 2 et 3 proposent une typologie hiérarchique de manipulations symboliques de contraintes. La dernière section décrit le modèle général de définition de règles de contrôle par observation et réaction au comportement de CSP-RCS.

1 Analyse

Une méthode algébrique peut être abstraite par une fonction à laquelle est attaché un ensemble de conditions devant être satisfaites avant son exécution. Nous étendons cette définition avec la possibilité de manipuler la méthode elle-même. Pour cela nous donnons une définition réflexive d'un solveur symbolique, comme un ensemble de règles de réécriture, augmentées chacune d'une phase de traitement introspectif (sur la règle elle-même). Cette phase contient par exemple la notification des changements de priorité, l'évaluation de la qualité des actions de la règle sur le reste du problème, le blocage de la règle. Nous verrons au chapitre III-4, l'utilisation concrète de ces connaissances refléxives pour le contrôle de RCS.

Un solveur symbolique de contraintes agit de deux manières, soit par réécriture, soit par ajout de contraintes redondantes. Nous pouvons alors le modéliser par une règle de réécriture, augmentée de pré-conditions et de post-actions :

```
R: Existe(i₁,…,iₙ) ➔ condition(R) |
 Calcul(i₁,…,iₙ),ajoute(r₁,…,rⱼ),retire(p₁,…,pₖ)        |
  notificationEtat(R).
{ p₁,…, pₖ } ⊆ { i₁,…, iₙ }
{ r₁, …, rⱼ } ⊄ { i₁,…, iₙ }
```

Si le problème contient n contraintes satisfaisant les conditions, alors l'exécution de `SolveurCalculus` ajoute de nouvelles contraintes et supprime un sous-ensemble des contraintes d'origine. Lorsque j = 0, le solveur réécrit, lorsque k = 0, le solveur propage de nouvelles contraintes.

L'équivalence entre deux problèmes est soumise à certaines conditions :

1. Les contraintes ajoutées ne suppriment pas de solution, c'est à dire qu'elles sont redondantes.
2. Les contraintes ajoutées sont équivalentes à un ensemble de contraintes présentes dans le problème.
3. Ne peut être retirée qu'une contrainte trivialement satisfaite ou obsolète, *i.e.* qui vient d'être réécrite ; car celle-ci n'apporte plus de réduction de domaine ou a été remplacée par une contrainte équivalente.

Définition 1-1 : Redondance d'une contrainte

Soit un système de raisonnement symbolique *RS* sur l'ensemble des contraintes *E*. Une contrainte *C* est redondante avec *E* relativement à *RS*, si l'application en une passe au moins de *RS* sur une contrainte de *E* produit *C*.

Pour un système de réécriture *R* donné, une contrainte redondante avec une autre lui est en particulier équivalente, car si en une passe p_0, on transforme C_1 en C_2, alors en particulier il existe une séquence de *k* passes dont la première est p_0, qui réécrit C_1 en C_2. Cette propriété garantit la conservation de l'ensemble des solutions et de la sémantique du problème, durant un raisonnement symbolique.

Définition 1-2 : Equivalence forte entre deux contraintes

> Soit un système de réécriture convergent *R* sur l'ensemble des contraintes du domaine. Deux contraintes C_1 et C_2 sont syntaxiquement équivalentes si l'application saturante de *R* sur C_1 puis sur C_2 produit une expression identique.

Par définition deux contraintes équivalentes produisent la même réduction de domaine. Une contrainte redondante produite par *RS* est rendue canonique par *R* ; en cas d'équivalence avec une des contraintes du problème, la contrainte n'est pas ajoutée évitant ainsi des redondances qui ne provoquent pas de coupure de l'arbre de recherche (que nous appelerons redondance «négative » ou «inutile »).

Dans les domaines moins formalisés, il n'est pas évident de trouver un système de réécriture convergent ; en revanche, il est aisé de trouver un système qui termine et qui produit une forme canonique au bout d'un certain nombre de passes. Cette forme canonique n'est pas unique mais elle suffit pour obtenir des calculs efficaces en temps et qualité, tant dans la réduction de domaine que dans la transformation symbolique.

Définition 1-3 : Equivalence faible entre deux contraintes

> Soit un système de réécriture *R* qui termine et produit une forme canonique au bout de *k* passes au maximum. L'équivalence faible entre deux contraintes C_1 et C_2 signifie qu'il existe une séquence de réécriture dans *R* qui part de C_1 et arrive à C_2.

Cette équivalence plus faible est utile pour les contraintes générales sur des domaines objets (contraintes fonctionnelles, contraintes prédicatives), dans la mesure où elle supprime les occurrences de contraintes volumineuses lourdes à manipuler.

Par ailleurs, pour garantir un coût raisonnable, les méthodes symboliques doivent être indépendantes de l'implantation et de la taille des domaines des variables. Sinon il faudrait garantir un temps constant pour l'accès à certaines valeurs utiles pour les méthodes symboliques (minimum, maximum).

Notations :

On note *c(val)* le résultat de l'évaluation de la relation de la contrainte *c* pour le n-uplet de valeurs *val*. On note *contraintes(v)*, la liste des contraintes de la variable *v*. On note *dom(v)*, le domaine de la variable *v*. On note *rfRegles(c)*, la liste des règles de raisonnement symbolique susceptible de s'appliquer sur *c*. On note *n* le nombre de variables, *e* le nombre de contraintes, *d* la taille du plus grand domaine.

1.1 Extension du modèle CSP

L'objectif du système CSP+RCS est d'intégrer les opérations de raisonnement symbolique et de maintien de cohérence d'un ensemble de contraintes, dans la procédure d'énumération de l'espace de recherche. Cela pose la question de la terminaison de la procédure de résolution hybride et nécessite l'ajout de primitives de gestion d'un problème dynamique.

1.1.1 Algorithme de résolution hybride

Dans cet algorithme, RCS est utilisé en complément des méthodes de cohérence, lorsque ces méthodes sont dans une situation d'impasse. L'objectif est de provoquer un changement dans le problème, en l'occurrence une modification des contraintes ou une réduction de domaine.

Procédure (1) de maintien de cohérence

```
PropageCohérence(V,D,K)
Données : un problème CSP = (VD,K) ; VariablesAPropager : Liste
Variables locales : v : Variable;
    TantQue VariablesAPropager ≠ ∅
          v: = first(VariablesAPropager);
          ∀ c ∈ contraintes(v), filtrage(c,v);
    FinTantQue
```

La procédure `filtrage(c,v)` maintient la cohérence de la contrainte `c` quand `v` est modifiée[30].

Procédure (2) de raisonnement symbolique

```
ProcèdeRS(V,D,K)
Données : un problème CSP = (V,D,K) ; ContraintesPourRS : Liste
Variables locales : c : Contrainte
Début
    TantQue ContraintesPourRS ≠ ∅
    c := first(ContraintesPourRS);
      ∀ r ∈ rfRegles(c),
          S := appliquerRegle(r,c);
          ∀ c' ∈S, ContraintesPourRS := {c'} ∪ ContraintesPourRS;
          propageCohérence(V,D,K);
      Fin∀
    FinTantQue
Fin
```

La procédure `appliquerRegle(r,c)` déclenche la règle `r` avec `c` en prémisse. Son action est répercutée dans la liste `ContraintesPourRS` : toute nouvelle contrainte est candidate à un raisonnement symbolique. Après chaque ajout de contrainte, le problème est rendu cohérent. La contrainte `c` joue un rôle particulier puisqu'elle initie l'application de la règle `r` ; pour cette raison nous appelons la contrainte `c`, «la contrainte active de `r` ».

Procédure (3) de résolution hybride

```
Début
 Repeat
      Repeat
           PropageCohérence(Vi,Di,Ki);
           ProcèdeRS(Vi,Di,Ki);
      Until (pas de changement)
      faireUnChoix;
 Until (uneSolution ou PasDeSolution)
Fin
```

[30] Elle ajoute dans la liste `VariablesAPropager`, les variables dont les domaines sont modifiés en conséquence.

Terminaison

La terminaison de la *procédure (3)* est soumise à l'existence d'un point fixe dans la boucle de traitement des contraintes. Or, les algorithmes de cohérence locale basés sur AC-4 terminent, comme le montrent [Güsgen & Hertzberg 1988]. De plus, un solveur symbolique termine s'il satisfait aux conditions de terminaison des règles manipulant des contraintes, c'est à dire la terminaison du système de réécriture de contraintes et la quantité finie de contraintes ajoutées par déduction automatique. Dans notre cas, la réécriture termine car elle est basée sur un ordre de réduction sur les classes de contraintes (donnée en annexe) ; la déduction de contraintes redondantes possède un point fixe pour deux raisons : 1) les contraintes candidates sont déjà présentes dans le problème, 2) chaque composition est mémorisée pour n'être appliquée qu'une fois. Finalement, la propriété de terminaison est conservée lors de l'intégration du maintien de cohérence avec la réécriture ou la déduction de contraintes.

1.1.2 Primitives d'ajout et retrait de contrainte

Nous avons étendu le *framework* de satisfaction de contraintes avec deux primitives réalisant les opérations à la base de RCS : l'ajout et le retrait d'une contrainte.

Ajout d'une contrainte

Le fait d'ajouter une contrainte ne modifie pas l'ensemble des solutions dans le contexte courant. Si une incohérence survient à la suite de l'ajout de contraintes redondantes, elle n'est donc pas due à ces contraintes mais aux choix faits en amont du contexte courant. Les contraintes redondantes ne servent qu'à mettre en évidence l'incohérence.

Coût :

> Soit *c* une contrainte. La création de *c* est une opération coûteuse dont la complexité dépend du langage hôte et de la puissance de la machine. L'ajout de *c* au problème consiste à ajouter *c* à l'ensemble des contraintes puis connecter *c* à toutes ses variables. Toutes les opérations sont des accès à des listes et se font en temps constant.

Retrait d'une contrainte

Dans un contexte donné de la résolution, le retrait d'une contrainte est valable tant que ce contexte n'est pas remis en cause (C.f. preuve). Un contexte est associé à un choix d'énumération et mémorise l'ensemble des contraintes et domaines de variable avant de prendre en compte ce choix. Si un contexte est restauré (retour-arrière), les contraintes qui ont été supprimées après son choix sont restaurées également.

Preuve:

> Lorsqu'une contrainte est retirée du problème dans le contexte C_i, elle est remplacée par des contraintes équivalentes. Appelons C_j le contexte du nouveau problème. *cts*(C_j) est équivalent à *cts*(C_i). De plus les domaines sont susceptibles d'être réduits après la création des contraintes ; donc *dom*(C_j) $\subseteq$ *dom*(C_i). Par conséquent, tout retrait de contrainte fait au niveau de C_j suppose les affectations déjà faites à la création de C_i.

Le retrait de contrainte dont nous avons besoin est différent du retrait de contrainte défini dans le cadre des CSP dynamiques[31] [Bessière 1991]. Contrairement à cette approche, l'objectif de notre retrait de contrainte est de trouver une solution à un sous-problème du problème initial, en garantissant la conservation de l'ensemble des solutions. Par ailleurs, lors du retrait de contrainte défini dans [Georget and al. 1997], le problème est remplacé par un problème moins contraint et qui n'a pas *a priori* le même ensemble de solutions ; pour ne pas oublier de solution, les domaines sont restaurés après chaque retrait d'une contrainte. Dans notre cas, le retrait assure l'équivalence entre les deux problèmes car toute contrainte retirée ne réduit plus les domaines dans le contexte courant[32]. Intuitivement, le nouveau problème tient compte des réductions de domaines précédentes, provenant de la contrainte retirée. La contrainte n'existe plus en tant qu'objet du problème mais elle est en quelque sorte remplacée par son résultat sur les domaines de valeurs des variables.

Coût :

> Soit *c* une contrainte. Le retrait de *c* du problème consiste à retirer *c* de l'ensemble des contraintes, déconnecter *c* des toutes ses variables, restaurer les contraintes réécrites dynamiquement et dont est issue *c* et enfin détruire l'objet. Toutes les opérations, des parcours de listes se font en temps linéaire avec *n et e*. Le restauration est en temps constant si l'objet représentant la contrainte réécrite a été mémorisé. La destruction de *c* dépend des caractéristiques matérielles.

1.1.3 Exploitation de la double représentation d'une contrainte objet

Comme nous l'avons déjà dit au chapitre I, une contrainte est à la fois l'abstraction d'un ensemble de procédures de calcul (les méthodes de filtrage) et une entité du problème de satisfaction de contraintes. Le paradigme de Programmation Par Objets regroupe ces deux représentations au sein d'une même structure, la classe de la contrainte. Nous avons vérifié expérimentalement que la réification des contraintes facilite la collaboration entre le maintien de cohérence et la réécriture de contraintes. En effet la même entité informatique contient les connaissances adaptées à chaque solveur : RCS manipule la contrainte en tant qu'objet composite, tandis que le *framework* CSP maintient les domaines des variables cohérents en exécutant les procédures de filtrage des contraintes qui y sont connectées.

1.2 Spécification des règles de RCS

Les règles sont distinguées selon trois critères : 1) le type de comportement, 2) la catégorie de prémisses, 3) le moment de déclenchement. Les règles agissent soit comme des règles de réécriture, par exemple la simplification de contrainte, soit comme des règles de déduction de contrainte redondante. Ces dernières sont appelées *règle de propagation*[33] et incluent la composition de contraintes, par exemple la composition linéaire selon l'algorithme de Gauss (C.f. section 3.3.1). Les règles sont regroupées en trois catégories :

[31] Un CSP dynamique est une séquence de CSP (statiques) tels que chacun résulte de l'addition ou le retrait d'une contrainte dans le CSP précédent. [Bessière 1991] propose un algorithme de filtrage pour maintenir la cohérence de CSP dynamique qui, dans le cas d'un retrait de contrainte, met à jour les domaines de manière plus fine qu'en réinitialisant les domaines.

[32] Si, au moment du retrait, on restaurait les domaines initiaux et appliquait successivement les choix des contextes mémorisés, on obtiendrait les mêmes domaines que le problème sans la contrainte.

[33] Nous employons le terme de "règle de propagation" par analogie avec les *"simpagation rules"* des CHR qui correspondent à notre définition d'un solveur symbolique.

règles unaires, règle n-aires et règles dont toutes les prémisses sont des contraintes de même classe[34]. RCS peut être utilisé pour reformuler un problème statiquement ou dynamiquement, selon que les règles s'appliquent à la création d'une contrainte ou pendant le traitement du problème avant un choix d'énumération. Notamment les règles de simplification sont statiques car elles servent à formater les contraintes en vue de faciliter l'application des règles dynamiques. Les règles de déduction de contraintes fonctionnelles $Z = f(X,Y)$ et de composition linéaire de contraintes sont dynamiques ; pour des raisons de complexité, il est conseillé de déclencher les règles n-aires dynamiques un nombre limité de fois. Par construction, certaines règles dynamiques ne se déclenchent qu'une seule fois et s'apparentent à des règles statiques (développement d'un carré parfait, distributivité d'un produit, composition de contraintes de cardinalité générique (C.f. section 3.2.2). Par ailleurs grâce aux stratégies, le moment de déclenchement est paramétrable ; par exemple le déclenchement d'une règle de propagation coûteuse peut être imposé en amont de la résolution.

1.2.1 Typologie

Les règles sont organisées hiérarchiquement selon les trois catégories ; pour chacune, deux classes de règles sont distinguées selon leur comportement. L'arbre d'héritage des règles est donnée en annexe. Le modèle proposé autorise l'écriture de règles sur des objets complexes, telles que la simplification d'une contrainte prédicative générale sur des variables objets[35], la déduction de contraintes de type $X = f(Y,Z)$ à partir de contraintes du même type.

Types de règles	Propriétés communes aux règles
Règles à n prémisses (n>1)	*ActiveCt* : contrainte active
Règles de propagation	*ForbiddenTuples :* historique pour éviter les exécutions redondantes
Règles de réécriture	retirer au moins une contrainte

Les concepts de propagation et de réécriture unaire et n-aire de RCS généralisent les raisonnements d'Alice. La normalisation, le retrait des contraintes triviales sont des cas particuliers de la réécriture unaire. La composition de contraintes est un cas particulier de réécriture n-aire. De plus la composition de contraintes s'applique sur toutes les contraintes linéaires, sans se limiter aux contraintes binaires.

Les règles à plusieurs prémisses distinguent la contrainte active qui est responsable du déclenchement de la règle. Dans un même contexte, les règles de propagation ne s'appliquent pas deux fois sur le même ensemble de prémisses, quel que soit l'ordre des contraintes dans cet ensemble. Pour ce faire, un historique mémorise les compositions ; il est mis à jour lors d'un retour-arrière en supprimant les compositions dont les contraintes sources sont annulées.

La typologie proposée dans ce travail contient deux règles générales qui ne spécifient pas de relation particulière : la réécriture d'instantiation (*RwInstantiationRule*), la déduction de contraintes à partir de contraintes fonctionnelles $Z=f(X,Y)$ (*FunctionalCtRule*). Ces règles sont des sortes de raisonnement sur une seule classe de contraintes car leurs prémisses sont de même type. Les règles modélisant un raisonnement unaire sur une classe, schématisé par

[34] Ces règles peuvent être unaires ou n-aires.
[35] BTBlockCt en BackTalk

$C \rightarrow C$ sont stables dans l'ensemble des instances de la classe C et de ses sous-classes. Elles représentent des lois sur l'extension ensembliste de la classe C. Une transformation naire stable dans une classe données est schématisé par $C_1,\ldots,C_n \rightarrow C'_1,\ldots,C'_n$, où chaque C_i est égale à C ou une sous-classe de C. Il généralise la règle unaire de type $C \rightarrow C$; en particulier si les contraintes déduites sont de type C, la loi est stable dans l'algèbre de la classe abstraite de C. Notons que si une règle porte sur une seule contrainte et si celle-ci est en pratique décomposée en une conjonction de contraintes primitives, la règle est considérée comme une règle unaire.

Réécriture de contraintes

La réécriture n-aire suppose de pouvoir supprimer *n* contraintes tout en gardant l'équivalence de l'ensemble. Dans les problèmes objets elle est difficile à utiliser ; en revanche dans le cas des égalités et inégalités arithmétiques, la composition linéaire inspirée de l'algorithme Gauss-Simplexe peut être considérée comme une règle de réécriture puisque le système linéaire résultant se substitue exactement au système initial. De plus, si le *framework* décompose les contraintes en contraintes primitives[36], la composition linéaire s'applique aux contraintes non linéaires et aux contraintes sur des termes fonctionnels.

Soit un système de réécriture, la réécriture d'un ensemble de contraintes S est définie comme la transformation de S en un nouvel ensemble S' telle qu'au moins une des contraintes de S est réécrite en une contrainte de S' : $c \in S, \exists c' \in S'$ tel que $(c \neq c')$ et $(c \Leftrightarrow c')$ et $(c' < c)$. Ainsi la réécriture d'un problème possède la propriété de préserver l'ensemble des solutions du problème (C.f. preuve en annexe).

2 Réécriture et propagation unaire

Les règles unaires sont toutes des propagations sauf la réécriture d'instantiation, et les règles de simplification.

2.1 Simplification d'une contrainte

Une règle de simplification est modélisée par une réécriture contextuelle en fonction des événements survenant sur les variables ou les contraintes. La prise en compte du contexte de la résolution est justifiée par la nécessité d'éviter une application systématique. Elle étend la définition d'une règle de réécriture comme suit :

Evénement(*X*), $C1 \rightarrow C2$, où *X* est une variable ou une contrainte ;
Evénement(*X*) est un événement modifiant l'entité *X*.

Dans le cas d'une simplification dynamique, l'événement concerne par exemple la modification d'un domaine, l'ajout ou le retrait d'une contrainte. Dans le cas particulier où l'événement concerne la détermination d'une variable, la réécriture remplace dans les contraintes toutes les occurrences des variables instantiées, par leur valeur ; nous l'avons appelée «réécriture d'instantiation ». Le processus de remplacement est la base de la résolution de contraintes en CLP [Colmerauer & Benhamou 1993]. La réécriture d'instantiation est générale ; en particulier elle s'applique sur les contraintes fonctionnelles et permet de simplifier l'algorithme de filtrage. Par exemple, la contrainte ternaire $Z=f(X,Y)$ ne

[36] C'est le cas de BackTalk pour les contraintes arithmétiques et objets.

peut être filtrée que lorsqu'une au moins des variables est instantiée. Sa réécriture en Z=f(X,*a*) ou Z = g(X) permet d'appliquer un algorithme d'arc-cohérence rapide tel AC-7.

2.2 Simplification statique

La simplification statique unaire normalise une contrainte à sa création. Elle est un cas particulier de la réécriture unaire où l'événement concerne la création d'une expression. Nous avons fait un compromis entre complétude et rapidité car le système de simplification n'est pas saturé ; les contraintes ne sont pas mises en forme normale mais sous une forme régulière suffisante pour appliquer les règles de RCS.

2.2.1 Simplification de contraintes générales sur des objets

Dans sa thèse, Pierre Roy a décrit des règles de normalisation de contraintes prédicatives définie par un lambda-terme booléen ; ces règles visent à supprimer les parties de code inutile dans la spécification du prédicat, par exemple les arguments du lambda-terme ou variables locales non utilisés dans le corps du bloc. Quand un bloc n'a pas d'argument, il est évalué car la contrainte est totalement instantiée.

c: $\lambda x_1 \ldots x_i \ldots x_n$.EXP $\xrightarrow[\textit{réécriture}]{}$ remove(c), $\lambda x_1 \ldots x_n$.EXP, si $x_i \notin$ EXP.
c: $\lambda x_1 \ldots x_n$.EXP $\xrightarrow[\textit{réécriture}]{}$ remove(c), si eval(EXP) = True.
c: $\lambda x_1 \ldots x_n$.EXP $\xrightarrow[\textit{réécriture}]{}$ Incohérence, si eval(EXP) = False.
c: : λ.EXP $\xrightarrow[\textit{réécriture}]{}$ eval(EXP), remove(c).

Cette réécriture reste une reformulation en amont de la résolution car on ne sait pas *a priori* transformer les lambda-termes en cours de résolution. Une transformation dynamique demanderait une interprétation du bloc ; cela suppose de connaître la sémantique de la relation sous-jacente, ce qui n'est pas possible dans le cas général.

2.2.2 Simplification statique arithmétique et booléenne

Le système de simplification de RCS est décomposé selon le type d'expression et selon le type des opérandes (variables ou constantes) et reprend partiellement celui d'Alice. L'existence d'une structure optimisée pour les additions n-aires permet, au niveau de RCS, de fusionner les règles d'Alice sur les sommes binaires. Le concept de produit n-aire que nous avons défini par analogie avec la somme n-aire nous a permis de supprimer les règles d'associativité sur la multiplication. Certaines règles d'Alice sont inutiles car le filtrage réalise une action qui aboutit au même résultat. C'est le cas des règles $0 = a.x \rightarrow 0 = x$, $b + a*y \geq 0 \rightarrow ((b/a) + y) \geq 0$.

Règles déclenchées à la création d'une combinaison linéaire

`a(b.x) → (ab).x ; -(b.x)→ (-b).x ; -(x+y)→ -x+-y`

Distributivité, déclenchée à la création d'un produit

`x.(y+z)→ (x.y)+(x.z) ; (x+y).(z+t) → (x.z) + (y.t)`
condition d'application : `x` ou `z` sont des constantes.
`x.x → x² ; (a+y)² → a²+2.a.y + y²`
condition d'application : `a` est une constante.

2.2.3 Simplification et décomposition de contraintes

La décomposition des contraintes transforme les contraintes complexes en une conjonction de contraintes primitives orientées. Par exemple l'expression -E est représentée par une variable intermédiaire R et la contrainte "opposé" -E = R, R est la variable en position de résultat dans la contrainte primitive[37]. La simplification est un processus de reformulation du problème qui remplace la décomposition initiale des contraintes par une décomposition plus adaptée au filtrage et avec laquelle la réduction de domaine est plus performante. Elle est appliquée à chaque étape de la construction de la contrainte ; de cette manière l'expression déjà construite est toujours en forme simplifiée.

Les bases de règles de simplification sont implantées dans les classes de contraintes. Cela permet de normaliser l'expression dès sa création, évitant ainsi le parcours d'une base de règle ou la construction d'un ensemble des conflits comme dans une modélisation classique. Par exemple dans la classe des contraintes "opposé", la règle -(-x) → x s'applique à chaque apparition de l'expression -R, si R est déjà le résultat d'une contrainte "opposé". La simplification se combine donc avec la décomposition.

Définition 2.2.3-1 : interdépendance de deux contraintes

> L'interdépendance qualifie l'intersection entre deux contraintes, c'est à dire le nombre de variables en commun. Deux contraintes sont interdépendantes quand elles ont au moins deux variables en commun.

L'interdépendance des contraintes favorise le filtrage ; elle rend ainsi le problème plus facile à résoudre.

Décomposition d'un carré parfait

$T_1^2=z \;;\; T_1=a+y \rightarrow \quad 2a.y + T_4 - z = -T_2 \;;\quad T_2=a^2 \;;\; T_4=y^2$

Condition d'application : a est une constante

De même qu'une expression arithmétique est modélisable par un arbre n-aire ou binaire, une décomposition peut se modéliser par une transformation d'arbre.

$$(T1)^2 = R \quad \Rightarrow \quad T2 + 2.a\,.\,Y - R = a^2$$

$$T1 = a + Y \qquad\qquad T2 = Y^2$$

La seconde décomposition est potentiellement plus profitable au filtrage car Y est davantage contraint.

Elimination des opposés

$-(-x) + y \quad \rightarrow \quad x + y \;;\; -(-x) \quad \rightarrow \quad x$

La règle se déclenche à la construction d'une contrainte "opposé" ou "addition binaire". S'il existe une expression F telle que -F = E, la simplification consiste à retirer la contrainte -E = T et à poser la contrainte F = T. La règle de réécriture qui s'applique est donc : {-F=E ; -E=T} → {-F=E ; F=T}.

Regroupement de coefficients dispersés dans l'arbre de décomposition

$a(b.x) \rightarrow (ab).x \;;\; a(b.x) + y \quad \rightarrow \quad (ab).x + y$

---

[37] Nous l'appelons "variable résultat".

Ces deux règles sont regroupées en une seule qui s'applique au niveau de la création de la multiplication de `(b.x)` par `a`. Elles sont susceptibles de poser de nouvelles contraintes sur les variables initiales du problème et parfois de retirer des variables intermédiaires.

$$a.T1 + c.Y \longrightarrow (a.b).X + c.Y$$
$$T1 = b.X$$

2.3 Propagation unaire

Les règles de propagation n-aire sur des contraintes objets peuvent se particulariser en règles unaires car leur spécification ne dépend pas du nombre de prémisses. En particulier, nous définissons à la section 3 une règle générale qui déduit à partir de contraintes fonctionnelles, un ensemble de contraintes fonctionnelles ayant une intersection éventuellement non vide avec les contraintes en prémisses. Dans le cas où cette règle a une seule prémisse, elle abstrait les implications entre les messages que peuvent s'envoyer des objets.

2.3.1 Propagation unaire arithmétique

Certaines règles de simplification statique peuvent s'interpréter comme des règles de propagation statique, c'est à dire conservant des contraintes initiales. C'est le cas des règles de développement d'un produit en somme (distributivité du + sur le *) et du développement d'un carré parfait $(x+a)^2$.

Développement d'un produit en somme (distributivité)

$$x \,.\, T_1 = u \,;\, T_1 = y + z \quad \rightarrow \quad T_2 + T_3 = u \,;\, T_2 = x \,.\, y \,;\, T_3 = x \,.\, z$$

La contrainte initiale x*T1 = u est retirée et remplacée par T2+T3 = u. La contrainte est retirée si la variable intermédiaire T1 n'est contrainte que par T1 = y+z ; cette dernière est retirée du problème car T1 n'intervient plus dans le maintien de cohérence.

Propagation de AllDifférent sur une bijection (BijectionRule)

Cette règle s'applique quand toutes les variables sont distinctes et de même domaine. Elle explicite le fait que le problème peut être posé comme la détermination d'une bijection F : I→I[38].

Soit $f \;:\; (a_i)_1^n \rightarrow (v_i)_1^n \,,\; AllDifférent((f(a_i))_1^n) \Rightarrow \sum_{1,n} f(a_i) = \sum_{1,n} v_i$.

L'introduction de cette contrainte a pour conséquence avantageuse d'initialiser la composition de contraintes linéaires. En particulier, J. Pitrat [Pitrat 1993] montre que la présence de cette contrainte permet de trouver une solution au carré magique 3x3 en ne faisant que des compositions linéaires.

[38] Alice rajoutait cette contrainte de manière automatique quand le but du problème était de trouver une bijection.

3 Réécriture et propagation n-aire

Les règles n-aires de RCS supposent que les contraintes sont présentes sous une forme simplifiée ; cette hypothèse est également nécessaire pour reconnaître quand deux contraintes générées sont équivalentes. Les règles de propagation se spécialisent dans deux domaines d'application, les contraintes fonctionnelles générales et les contraintes arithmétiques. Le coût d'application d'une règle de propagation est d'autant plus grand que le nombre de prémisses est élevé. Les règles générales de raisonnement sur des objets peuvent s'instancier en règle de réécriture ou de déduction ; néanmoins, les règles n-aires modélisent en général des déductions de contraintes redondantes car il est difficile d'expliciter une situation où un ensemble de contraintes peut être strictement remplacé par un autre.

3.1.1 Orientation des règles de propagation

Un des moyens de garantir la terminaison de la propagation consiste à orienter les règles de propagation. En effet à chaque déclenchement d'une règle, une des contraintes en jeu ne doit pas avoir encore été traitée formellement par cette règle dans le contexte courant. Nous choisissons donc de propager les règles qui portent sur la contrainte active, *i.e.* la contrainte qui a initié le raisonnement symbolique idoine. Cette restriction du champ d'application nous permet de spécialiser l'action des propagations en fonction du type de la contrainte qui les a déclenchées (C.f. exemple).

Exemple : orientation de la composition de x=y avec (f(x) op c)

> Considérons la composition de (x = y) avec une contrainte du type (f(x) *op* c), où *op* représente un comparateur arithmétique qui substitue la variable x dans l'expression f(x) et remplace la contrainte (f(x) *op* c) par (f(y) *op* c). Cette substitution est dangereuse car elle supprime une interconnexion dans le problème, alors que ce lien reste utile puisque la variable x n'est *a priori* pas déterminée. Par ailleurs la règle est symétrique et ne spécifie pas de comparateur *op* particulier.
>
> Si la contrainte active est (x = y), alors l'exécution de la règle introduit une ambiguïté si l'on ne dispose pas de critère pour connaître la substitution la plus pertinente. Dans ce cas nous choisissons d'orienter la propagation de manière à conserver les variables les plus contraintes et à remplacer les variables les moins contraintes, ceci afin d'augmenter la densité du problème. Si la contrainte (f(x) op_0 c)[39] est la contrainte active, nous restreignons la substitution à (f(x) op_0 c).

Plus généralement l'affinage de la propagation en fonction de la contrainte initiatrice est un moyen simple à mettre en œuvre pour lever les ambiguïtés liées à une composition n-aire détruisant au moins une des contraintes sources.

3.1.2 Stratégies de propagation n-aire

Nous distinguons deux stratégies d'application d'une propagation n-aire. Avec la stratégie *saturate*, RCS compose la contrainte active avec toutes les contraintes candidates du problème. L'arbre de composition se développe nettement en largeur car, pour chaque

[39] op_0 est une instance de l'opérateur abstrait *op*.

contrainte ajoutée, d'autres compositions sont faites. Le processus est complet mais la taille du problème diverge. Avec la stratégie *doBest*, RCS sélectionne un sous-ensemble de compositions selon un certain critère. Celui-ci compare une fonction d'évaluation avec une valeur critique. La taille de l'ensemble des contraintes reste dans le même ordre de grandeur mais le processus n'est pas complet. Dans le cas où il n'y a que deux contraintes combinables, *doBest* est équivalent à *saturate*.

Les critères suivants sont utilisés pour distinguer différents types de propagation n-aire :

a) Celle qui retire le plus de variables dans la contrainte générée. La fonction d'évaluation à minimiser est le nombre de variables supprimées par rapport aux variables des contraintes sources (en général les variables communes).

b) Parmi celles qui retirent des variables, la première qui génère une contrainte C permettant de réduire le domaine immédiatement. La fonction d'évaluation est la différence entre les domaines avant et après composition et doit être positive.

c) Celle qui engendre la contrainte dont les variables sont les plus contraintes. La fonction d'évaluation à maximiser est la somme des degrés des variables de la contrainte.

Ces critères requièrent une connaissance pour prévoir la topologie de la contrainte résultante ce qui n'est pas toujours le cas. Néanmoins dans le cas particulier des compositions linéaires, on peut vérifier certains critères sans créer réellement la contrainte.

3.2 Raisonnement sur des contraintes générales

Comme nous l'avons vu au chapitre I, les contraintes générales ou «contraintes objets » sont de deux types : les contraintes prédicatives définies par un prédicat n-aire et les contraintes fonctionnelles défini par une fonction (n-aire) sur des ensembles d'objets. Les relations des contraintes générales étant potentiellement plus complexes que les relations arithmétiques, RCS s'exécute sur ces contraintes en une passe, en accord avec des stratégies pré-conditionnelles. La relation des contraintes objets n'étant pas fixe, le choix des stratégies dépend de chaque domaine d'application (temporel, géométrique,…) et des connaissances expertes de ces domaines.

3.2.1 Contraintes fonctionnelles générales (PerformCtRule)

Soit une contrainte fonctionnelle, Z = *f*(X,Y), avec X, Y et Z trois variables. L'ensemble des domaines de X,Y et Z constitue une algèbre dont les opérateurs sont les symboles de fonctions sur *dom*(X) x *dom*(Y), c'est à dire l'ensemble des messages que comprennent les objets de *dom*(X) avec pour argument une valeur de *dom*(Y). En particulier, le symbole *f* représente le nom de l'opérateur binaire sur chaque objet dans le domaine de X et de Z. L'objectif de la règle *PerformCtRule* est d'expliciter les relations causales entre les symboles de fonction. La règle se particularise pour différents domaines d'application en spécifiant les égalités fonctionnelles *via* un langage textuel à base de motif (voir plus loin). Le coût de la détection des prémisses est linéaire en *O(en)* (*e* le nombre de contraintes et *n* le nombre de prémisses)[40].

Définition 3.2.1-1 : Règle PerformCtRule

[12] La règle s'utilise en pratique avec n=2.

$\forall i \in [1, p], \alpha_i, \beta_i, \gamma_i \in [1,n], \forall j \in [1,q], \alpha_j, \beta_j, \gamma_j \in [1,n],$

$$Y_1=\Phi_1(X_1,\ldots,Z_1) \; ; \; Y_n=\Phi_n(X_n,\ldots,Z_n) \Rightarrow \text{add}(\bigwedge_1^p (Y_{\alpha_i} = \Gamma_i(X_{\beta_i}, Z_{\gamma_i})) \; ;$$

$$\text{remove}(\bigwedge_1^q (Y_{\alpha j} = \Phi_j(X_{\beta_j}, Z_{\gamma_j}))$$

Dans le *framework* CSP+RCS nous avons considéré la règle suivante :

{(*x message1: y)* ; (*y message2: z*) } → (*o_1 messageR: o_2*) ,où *o_1*, *o_2* sont deux variables objets.

Cette règle est une instance de *PerformCtRule* pour n=2, Y_1=Y_2=True, p=1, q=0 et Y_α=True. Elle s'instancie par exemple pour modéliser le théorème « deux droites perpendiculaires à une troisième sont parallèles »[41] (C.f. chapitre IV).

La tâche de sélection vérifie qu'il existe deux contraintes fonctionnelles de sélecteur *message1:* et *message2:* telle que la variable en position de receveur dans la seconde est identique à la variable en position d'argument dans la première. Le langage de spécification du format des prémisses et du résultat construit un motif à trois chaînes de caractères de la forme "*Sujet sélecteurDuVerbe: Complément*". La règle contient une liste de motifs *condition* (`ifSelectors`) et un motif *action pour l'ajout* (`thenAddSelector`). Un motif représente un envoi de message. Il se compose d'un sélecteur et de deux symboles représentant l'objet receveur et l'objet argument qui sont les inconnues à unifier dans la règle.

Algorithme de sélection sur des contraintes fonctionelles générales

Cette section décrit l'algorithme de la règle PerformCtRule de RCS. Un motif est implanté par une chaîne de caractères. Cette règle ayant été implantée au sein du *framework* BackTalk, un motif représente l'envoi de message avec une syntaxe Smalltalk : le receveur est le premier mot lu dans le motif, le sélecteur le second mot et l'argument le troisième mot. La détection des prémisses est réalisée par un *parser* ; un dictionnaire (*Mémoire*) gère l'unicité des symboles dans les motifs en mémorisant les correspondances entre les variables contraintes et les symboles des motifs.

1) *`receveurFrom: motif`*

cherche dans *Mémoire* la variable qui correspond au symbole en position de receveur dans *motif*. Rend *nil* si le symbole n'est pas lié.

2) *`argumentFrom: motif`*

cherche dans Mémoire la variable qui correspond au symbole en position d'argument dans *motif*. Rend *nil* si le symbole n'est pas lié.

3) *`executeSchémaAction: actionMotif`*

crée la contrainte spécifiée par le motif *actionMotif* et qui référence les variables stockées dans *Mémoire*. En Smalltalk la contrainte a pour expression :

```
(self receveurFrom: actionMotif)
      btPerform: (self selecteurFrom: actionMotif)
      with: (self argumentFrom: actionMotif).
```

4) *`unifieMotif: motif constraint: aConstraint`*

[41] *y* et *z* sont alors égaux, *messages1:* et *message2:* valent #perpendicularTo.

fait correspondre *aConstraint* avec *motif*. Si ce n'est pas possible renvoie *Faux,* sinon renvoie *Vrai*. L'algorithme utilisé est une particularisation de l'algorithme de *sélection* par résolution d'un CSP ; celui-ci est présenté au chapitre IV-2.

Nous supposons connues, pour une contrainte fonctionnelle `BTPerformCt`, les primitives d'accès suivantes : `receveur(aConstraint)` renvoie la variable en position d'objet receveur ; `argument(aConstraint)` renvoie la variable en position d'argument.

3.2.2 Contraintes de cardinalité indépendantes du domaine d'application

La gestion d'emploi du temps et d'allocation de ressources humaines amène en général à contraindre le nombre d'employés présents à une tâche ou à une période de la journée. Ces contraintes sont des relations qui s'expriment bien en langage naturel et difficilement en langage mathématique, comme la contrainte de rotation de personnel qui impose au plus trois jours de travail le soir dans une semaine. Dans un langage à objets, cette contrainte s'exprime simplement en envoyant des messages aux variables «objets » qui représentent les tâches. Afin d'exprimer cette contrainte, nous avons créé une classe de contraintes de cardinalité générique[42], dont l'expression représente le nombre d'éléments i d'un ensemble *S* vérifiant une expression booléenne f(i). L'expression f(I) est un lambda-terme booléen (*bloc* Smalltalk) prenant en argument la valeur d'une des variables de *S*.

Exemple 1 :

Soit S un tableau de 100 variables de domaine 0..9. Soit T dans 1..10 et U dans 5..5. Le CSP suivant contraint le nombre des éléments du tableau S dont la valeur V vérifie V² > 4, à être borné par l'intervalle [1 ; 10] et le nombre d'occurences de la valeur 2 dans S à être égal à 5.

```
|{ x in S, x2 > 4}| = T, |{ x in S, x = 2}| = U.
```

Néanmoins la contrainte de cardinalité générique est difficile à utiliser telle quelle pour maintenir les domaines cohérents. Elle est traitée par une cohérence d'arc, complète et générale car elle ne spécifie pas le type de valeur des domaines ni la sémantique de l'expression à vérifier. De plus pour des raisons d'efficacité, cette cohérence d'arc est réalisée lorsque la contrainte possède au plus deux variables inconnues. Cette section propose deux règles déduisant des contraintes redondantes qui évitent ces inconvénients ; les contraintes modifient immédiatement les domaines. Ces deux règles s'exécutent en pré-traitement du problème. Elles héritent de la règle abstraite *CtClassRule* car leurs prémisses sont toutes de type *Cardinalité*.

Composition de cardinalités (CardGen-CardGen)

> Soit S, un ensemble de variables, EXP, EXP0,…,EXPk, des expressions unaires booléennes, B,B0,…,Bk, des variables entières.
> SI CardinalityGen(S, EXP) = B, CardinalityGen(S,EXP0) = B0,…,
> CardinalityGen(S,EXPk) = Bk,
> ET Pour tout i $\in$ 0..k, EXPi implique logiquement EXP,
> ALORS
> $\forall i \in 0..k, \max(B_i) \leq \max(B)$ et $\min(B) \geq \sum_{i \in 0..k} \min(Bi)$

[42] La contrainte de cardinalité générale (`BTCardinalityGenCt`) est une extension de la contrainte d'occurrences de BackTalk (`BTCardinalityCt`).

EXPi implique EXP lorsque pour toute variable x de S, EXPi(x value) implique EXP(x value). Ce test peut être optimisé par une méthode exploitant la sémantique de EXPi et EXP. Cette composition est valable pour tout type d'expression, en particulier on peut l'utiliser avec des contraintes d'occurrence. De manière informelle cette règle signifie : si |{x in S,f(x)}| = T et |{x in S,g(x)}| = U et si pour tout $i \in$ S, f(i) implique logiquement g(i), alors min(U)≤ T et U ≤ max(T).

Composition occurrence-somme linéaire(Card-Lin)

Soit S, un ensemble de variables, B1,….Bk, des variables entières ; toutes les variables de S ont le même domaine D de taille k. SI Pour tout i dans D, Occurrences(S, x=i) = Bi, ET Pour tout x de S, *Dom*(x) = D, ALORS $(\sum_{i \in D} i \times Bi) = \sum_{x \in S} x$

Exemple 2 :

Soit S un tableau de 100 variables de domaine 0..9. Soit T dans 0..10 et U dans 5..5.

```
(1): |{ x in S, x2 = 4}| = T, (2): |{ x in S, x = 2}| = U.
```

Le filtrage des contraintes (cohérence partielle pour (2) et arc-cohérence binaire pour (1)) ne réduit pas le domaine de T ; la résolution énumère donc les valeurs 0,1,2,3,4 pour T et cherche pour chacune un n-uplet de valeurs des variables de S, qui contienne exactement T valeurs dont le carré est supérieur à 4. Le système énumère donc les domaines des variables de S alors que une valeur correcte de T doit être supérieure ou égale à 5 pour que la contrainte d'occurrence soit satisfaite ; en effet la contrainte d'occurrence impose déjà l'existence de 5 variables avec la valeur 2.

En appliquant la règle *CardGen-CardGen*, le domaine de T est réduit tel que T≥ *min*(U),*i.e.* T≥5, puisque x=2 implique x^2=4. La résolution du nouveau problème est très rapide car 5 est la première valeur de T qui est solution.

Exemple 3 :

Soit S un tableau de 100 variables de domaine 0..9. Soit T dans 0..20 et $U_0,\ldots,U_9$ telles que U_i est dans i..i.

$|\{x \text{ in } S, x^2 > 4\}| = T,\ |\{x \text{ in } S,\ x = 0\}| = U_0,$
$|\{x \text{ in } S,\ x = 1\}| = U_1, \ldots,\ |\{x \text{ in } S,\ x = 9\}| = U_9.$

Une résolution par cohérence et énumération ne détecte l'incohérence du problème qu'après avoir exploré le produit scalaire des domaines si le filtrage de la contrainte d'occurrence réalise une cohérence partielle[43] (100*10*21 tests au pire cas). Or il est clair qu'une valeur correcte de T ne peut être inférieure à $U_3+U_4+\ldots+U_9$; car les contraintes d'occurrences imposent déjà l'existence de U_3 variables dont la valeur est 3,..., U_4 variables dont la valeur est 4, etc.. On a $T \geq \sum_{i \in [3;9]} \text{Occurrences}(S, x = i)$.

L'application de la règle *CardGen-CardGen* avant la résolution a pour effet d'augmenter le minimum de T et de mettre en évidence l'incohérence : puisque toutes

[43] L'algorithme utilisé dans BackTalk est similaire à celui énoncé dans la thèse de F. Laburthe.

les expressions x=3,…et x=9 impliquent x²>4, alors $T \geq \sum_{i \in [3;9]} \min(Ui)$ *i.e.* T ≥ 35. Le coût de cette déduction consiste à évaluer l'implication logique des expressions, ce qui se fait en temps constant car les EXPi sont des égalités unaires.

3.3 Compositions de contraintes linéaires

La composition linéaire de contraintes s'applique sur les contraintes d'égalités et d'inégalités. Elle s'appuie sur le principe de fonctionnement de l'algorithme de Gauss : elle choisit une contrainte pivot et la compose avec les contraintes ayant des variables communes. La contrainte pivot est déterminée automatiquement ; c'est exactement la contrainte active. Cette section présente trois types de compositions : la composition de deux contraintes (*linlin*), l'application récursive de la composition binaire successivement sur un nombre fini de contraintes (*nlinRec*), enfin la transformation d'un système de contraintes par la méthode d'élimination de Gauss étendu aux inégalités (*Gaus-Simplexe*). Les compositions ajoutent de nouvelles contraintes sauf *Gauss-Simplexe* qui réécrit une partie du problème. Nous considérons plusieurs types de compositions binaires dans ce document :

- Celle qui utilise la contrainte linéaire la plus grande du problème.
- Celle qui retire le plus de variables dans la contrainte générée,
- Celle qui génère la contrainte la plus petite.
- Celle qui supprime au moins une variable commune
- Celle qui génère une contrainte permettant de réduire le domaine immédiatement.

La dernière option nécessite de pouvoir prévoir les réductions de domaines que l'ajout d'une contrainte va éventuellement entraîner[44].

Dans le cas où les contraintes non linéaires sont décomposées en contraintes primitives, la composition linéaire s'applique sur des contraintes polynomiales à plusieurs inconnues ; le degré n'est pas limité *a priori*, néanmoins pour des degrés élevés, des méthodes spécialisées comme le calcul des bases de Gröbner sont plus adaptées.

3.3.1 Associativité des compositions linéaires

Si la contrainte C_1 est composée avec la contrainte C_2 puis, les contraintes résultantes avec C_3, il est inutile de composer C_1 avec les contraintes résultant de la composition de C_2 et C_3, d'où l'équivalence logique suivante[45] :

`compose([C1,C2],U),compose(U,C3) ⇔ compose([C2,C3],W),compose(C1,W)`

De plus, si le résultat de la composition de C_1 et C_2 est la contrainte C_3 alors la composition de C_2 avec C_3, ou de C_1 avec C_3 produira uniquement des contraintes syntaxiquement équivalentes à C_2 ou C_3 :

`compose([C1,C2],C3) ⇔ compose([C1,C3],C2) ⇔ compose([C2,C3],C1)`

[44] La manière basique consiste à ajouter la contrainte effectivement ; cela oblige à retirer la contrainte dans le cas où elle ne réduit pas le domaine ; les domaines n'ont pas à être restaurés puisqu'ils n'ont pas été modifiés.

[45] `compose` est un prédicat binaire vrai si le deuxième argument est la liste des contraintes résultant des combinaisons des contraintes du premier argument.

D'après ces équivalences, la réalisation d'une composition linéaire rend inutile d'autres compositions potentielles. Il est possible de les éviter en mémorisant pour chaque contrainte impliquée dans une composition, la liste des contraintes avec lesquelles elle ne doit plus ou pas se composer.

3.3.2 Composition de deux contraintes linéaires (linlin)

La composition de deux contraintes c_1 et c_2 consiste à calculer un ensemble de ratios non nuls et, pour chaque ratio *p,* construire $c_1 - p*c_2$. Cette opération est réalisée par la fonction compose(c_1,c_2) qui renvoie la liste des contraintes redondantes créées. L'arité de la contrainte résultant de $c_1 - p*c_2$ est donnée par *arity(c_1) + arity(c_2)- size(RemovedVariables(K , c_1, c_2)). RemovedVariables(p , c_1, c_2)* est l'ensemble des variables communes qui n'apparaissent pas dans la contrainte créée. Comme nous l'avons vu à la section 3.1.2, la propagation est paramétrée par le critère de sélection des contraintes résultantes ; dans le cas de *linlin*, ce critère calcule l'ensemble des ratios.

Propriétés de *linlin* quelque soit la stratégie

compose(c1,c2) ≡ {c2 - k * c1, k ∈ Ratios_{12}}

Chaque k définit une classe d'équivalence {∃ α ∈ N / α*c2 - α*k*c1} ; du fait que les contraintes sont simplifiées de manière à avoir les coefficients les plus petits, seule la contrainte la plus simple de la classe d'équivalence est ajoutée. Cela garantit qu'aucune contrainte trivialement équivalente à c2 -k*c1 n'est ajoutée. La propriété suivante provient de de l 'équivalence entre les contraintes de compose(c1,c2) pour un ratio k donné :

compose(c1,c2) <=> compose(c2,c1) si et seulement si ∀k ∈ Ratios_{12}, ∀k' ∈ Ratios_{21}, k * k' = 1.

Stratégies de composition LinLin

Les deux stratégies *saturate* et *doBest* sont combinables au niveau utilisateur ; par exemple si c1 ou c2 est binaire la règle est appliquée selon *saturate* car il y a au plus 2 variables en commun (le danger d'explosion combinatoire est limité), si c1 et c2 sont d'arité supérieure à deux, il se peut qu'il y ait beaucoup de variables en commun, donc la stratégie *doBest* est préférable.

Définition 3.3.2-1 : stratégies de saturate

Soit `ratiosOfTupleOption` la variable spécifiant le calcul des coefficients à utiliser dans la composition. Nous distinguons cinq critères :

RatiosOfTupleOption	Critère
maxRemoved	Le nombre de variables supprimées est maximum.
SmallerRemaining	La contrainte résultat contient le minimum de variables.
WeakMaxRemoved	Autorise la contrainte à être plus grande mais son arité doit être inférieure au nombre total de variables distinctes dans c_1 et c_2.
All	Tous les ratios qui retirent au moins une variable.
one	Le premier ratio qui conserve la variable la plus contrainte.

Remarquons que les options *all* et *weakMaxRemoved* sont coûteuses car il y a beaucoup de compositions valides.

Définition 3.3.2-2 : stratégies de doBest

Les ratios sont calculés conjointement avec la recherche des contraintes à combiner. Les critères sont définis par rapport aux critères de *saturate*.

RatiosOfTupleOption	Critère
doBestRemaining	sélectionne les contraintes et les ratios qui génèrent la plus petite contrainte.
doBestRemoved	sélectionne les contraintes et ratios qui retirent le plus de variables communes.
biggerCt	crée la contrainte la plus grande.
doOneBestRemaining	choisit une contrainte parmi celles trouvées par *doBestRemaining*.
doOneBestRemoved	choisit une parmi celles trouvées par *doOneBestRemoved*.

Les ratios étant combinables, doOneBestRemaining équivaut à doBestRemaining+one et doOneBestRemoved équivaut à *doBestRemoved+one.*

Comparaison

Soit c_1, composable avec $c_2,c_3,\ldots,c_k$. r_{1i} est un ratio pour compose(c_1,c_i).

- *Saturate/smallerRemaining*

La stratégie *saturate/smallerRemaining* consiste à faire les *k* compositions possibles, en choisissant à chaque fois les meilleurs ratios. On obtient alors l'ensemble des meilleurs ratios $R = \{r_{jb_j}, j \in [2,k]\}$ pour chaque composition (c_1,c_j). Un ratio est d'autant meilleur qu'il élimine des variables communes.

- *DoBestRemaining*

La stratégie *doBestRemaining* consiste à choisir la composition qui engendre la contrainte ayant le moins de variables, en calculant pour chaque composition (c_1,c_j), l'ensemble R_{1j} défini plus haut, puis en choisissant la contrainte c_j dont la composition avec un ratio mémorisé dans R_{1j}, aboutit à la contrainte de plus petite arité. Enfin on applique la stratégie *saturate* sur la composition de c_1 avec c_j uniquement.

Influence des différentes stratégies sur un exemple

Au chapitre II, nous avons montré expérimentalement que pour le problème DONALD+GERALD=ROBERT, aucune composition binaire n'est réalisée avant le premier choix, dans le cas où la stratégie est *smallerRemaining*. Avec la stratégie plus souple *weakMaxRemoved*, il se produit une explosion combinatoire.

Stratégies	Borne-cohérence		Borne-cohérence + composition binaire
	Heuristiques	Résolution[46]	
DoBestRemoved	MinSize (minValue, maxValue)	11bt,12choix 0,16s	6,113s ; 7 bt ; 9 choix
Saturate/smallerRemaining	’’		9,644s ; 6 bt ; 8 choix

[46] Résultat : 526485+197485=723970

DoBestRemoved, DoBestRemaining, Saturate/maxRemoved, Saturate/smallerRemaining	MostConstrained ‘’ ‘’ ‘’	10bt,11choix 0,117s	1,14s 10 bt ; 11 choix
DoBestRemoved	GreatestRegret, MinValue	27bt, 28choix 0,212s	17.907s ; 9 bt ; 10 choix
Saturate/smallerRemaining	‘’		17.178s ; 6 bt ; 7 choix
DoBestRemaining	‘’		23.913s ; 9 bt ; 10 choix

Tableau 11[47] *: stratégies sur DONALD+GERALD=ROBERT.*

Le tableau ci-dessus présente les différentes stratégies. Notons que la plus rapide n'est pas la plus directe en terme de retour-arrières. Avec *mostConstrained*, la composition binaire est inefficace, ceci quelle que soit la stratégie.

Définition mathématique

Cette section définit les stratégies *saturate/smallerRemaining* et *doBestRemaining*.

Soient i,j $\in$ [1,*n*], où *n* est le nombre de contraintes composables, soit $N_{ij} = \min_{k \in \mathbb{R}} (N_{C_i - kC_j})$. $N_{C_i - k.C_j}$ représente l'arité de la contrainte générée par composition linéaire de C_i et C_j. En informatique, on restreint le domaine de variation de k à un sous-ensemble de ℝ, l'ensemble des k qui enlèvent au moins une variable dans la composition. Soit $\chi_{ij} = \{x_{\alpha_1}, x_{\alpha_2}, ..., x_{\alpha_p}\}$, l'ensemble des coefficients non nuls des variables communes à C_i et C_j. Pour tout q, on note t_{iq} le coefficient de la variable x_q de la contrainte C_i et l'expression de C_i, $t_{i1}x_1 + ... + t_{iz_i}x_{z_i} = b_i$.

On définit informatiquement N_{ij} par $N_{ij} = \min_{k \in K_{ij}} (N_{C_i - kC_j})$, où K_{ij} est l'ensemble des ratios *k* impliqués dans les compositions[48]. $K_{ij} = \left\{ \frac{t_{i\alpha_s}}{t_{j\alpha_s}}, x_{\alpha_s} \in \chi_{ij} \right\}$.

De manière générale, "composer linéairement C_i et C_j" signifie choisir un ensemble de coefficients (réel) *k* particuliers inclus dans l'ensemble K_{ij} et ensuite ajouter pour chaque k, la nouvelle contrainte C_i - *k**C_j. Par la stratégie *smallerRemaining*, les contraintes contiennent le moins de variables possible. Cela se traduit mathématiquement par le fait que, pour C_i et C_j fixées, l'ensemble des compositions de C_i et C_j est $S_{ij} = \{C_i - k \cdot C_j, k \in \overline{K_{ij}}\}$, où $\overline{K_{ij}} = \{k_l, l \in [1,m], N_{C_i - k_l \cdot C_j} = N_{ij}\}$. N_{ij} est la valeur minimale de l'arité des contraintes résultant de compose(C_i,C_j). $\overline{K_{ij}}$ est l'ensemble des coefficients tels que le nombre de variables de la combinaison linéaire de C_i et C_j soit minimum. L'arité de la contrainte C_i - h*C_j est calculée par : size(C_i) + size(C_j) – p - size(*RemovedVariables*(h,i,j)). $\mathrm{RemovedVariables(i, j, h)} = \{x_{\alpha_s} \in \chi_{ij}, t_{i\alpha_s} = h * t_{j\alpha_s}\}$.

[47] Les tests ont eu lieu sur un Pentium I 90 avec 24 Mo de mémoire vive. Les temps de résolution doivent être considérés relativement les uns par rapport aux autres.
[48] On calcule K_{ij} en divisant les coefficients des variables communes entre C_i et C_j.

Soit C_i une contrainte à composer. Soit Φ_i l'ensemble des contraintes ayant au moins une variable en commun avec C_i. $\Phi_i = \{C_u, \chi_{iu} \neq \emptyset\}$. Comme nous l'avons déjà dit, deux stratégies existent selon le choix de C_j, des coefficients k et le nombre de contraintes créées.

- La stratégie *saturate/smallerRemaining* réalise pour toutes les contraintes C_j candidates, toutes les compositions issues de l'ensemble $\overline{K_{ij}}$. Aucun choix particulier n'est fait sur la contrainte C_j. Le nombre de contraintes créées est indéterminé. L'ensemble des compositions est $S_i = \{S_{iw}, C_w \in \Phi_i\}$.
- La stratégie *doBestRemaining* commence par choisir les contraintes C_j dont la composition avec C_i engendre la contrainte la plus petite. On cherche donc un j ∈ [1,n]/{i} tel que $N_{ij} = \min_{k \in [[1,n]]/\{i\}} (N_{ik})$. On réalise alors toutes les compositions de C_i et C_j créant des contraintes ayant la même arité N_{ij}. L'ensemble des compositions est S_{ij}. S_{ij} est un sous-ensemble de S_i.

3.3.3 Composition linéaire n-aire (nlinRec)

Dans son principe, la composition *nlinRec* généralise *linlin*. Plus précisément, la composition récursive d'un système de plus de deux contraintes est définie comme une séquence de compositions binaires. Les contraintes sont ordonnées selon leur position dans le système linéaire. A chaque étape, les contraintes résultant de la i$^{\text{ème}}$ composition deviennent les contraintes sources de la i+1$^{\text{ème}}$ composition. "composer k contraintes" signifie prendre k-1 contraintes, les composer récursivement, puis composer chaque résultat avec la contrainte restante. Soit $c_0, c_1, \ldots, c_k$, k+1 contraintes combinables. *NlinRec* produit $\{(\ldots((c_0 - p_1{*}c_1) - p_2{*}c_2)\ldots) - p_k{*}c_k\ ,\ p_1 \in \mathrm{Ratios}_{10},\ p_2 \in \mathrm{Ratios}_{210}, \ldots, p_k \in \mathrm{Ratios}_{k..210}\}$. Les ratios sont calculés selon les stratégies de composition binaire, éventuellement distinctes à chaque étape. Pour des raisons de complexité, un seul ratio est sélectionné en pratique, pour chaque composition binaire ; si la stratégie permet d'en calculer plusieurs, le premier est choisi automatiquement.

Voici un algorithme réalisant la composition *nlinRec* :

```
Compose_nlinRec(c0, n)

    Données : n ∈ℕ ; S = ensemble des contraintes linéaires du problème
    Variables locales : ci = contrainte à composer ; res = résultat de
    composition intermédiaire ; vars = variables de res.
Début
    vars := Variables(c0)
    res := c0
    Tantque (n>0 et res ≠ nil)
          ci := sélection(S) telle que Variables(ci) ∩ vars ≠ ∅
          S := S - {ci}
          res := compose(res,ci)
          n := n-1
    FinTantque
Fin
```

Globalement, si l'on prend une stratégie telle que la composition binaire ne produise qu'une seule contrainte, la composition *nlinRec* est un réordonnancement des compositions binaires qui auraient lieu sur le même ensemble S de contraintes. En particulier, les deux traitements détectent l'incohérence si elle existe. En effet sur le même ensemble S, la composition binaire réaliserait ($c_0 - p_1 * c_1$) puis ($c_0 - p_2 * c_2$),….($c_0 - p_k * c_k$) et réitèrerait sur les contraintes résultats à l'étape de manipulation symbolique suivante dans la procédure de résolution.

Dans ce cas, RCS produit en quelque sorte un arbre k-aire de composition commençant par les compositions linéaires avec c_0 ,et le parcourt en largeur. La composition *nlinRec* regroupe les compositions de chaque branche de cet arbre, ce qui a pour effet de changer l'ordre de parcours. Dans tous les cas, à la fin de la résolution, si aucune incohérence n'est détectée, le problème contient les mêmes contraintes.

3.3.4 Vision ensembliste de la composition linéaire

Dans une vision ensembliste, nous représentons toute contrainte ci par l'ensemble de ses variables *Ci* = *variables*(ci)). Dans ce cas, nous ne nous intéressons pas à la sémantique mais aux intersections entre les contraintes.

Notations :

On note *ci*, la contrainte dans sa forme en intension, objet analytique, et *Ci* l'ensemble correspondant. On note *Variables*(*ci*) l'ensemble des variables de la contrainte ci et *arity*(*ci*) la taille de cet ensemble. On a toujours *Ci* = *Variables*(*ci*).

Quelques opérateurs d'ensemble

Soit A et B deux ensembles, la différence symétrique entre A et B est :

$$\Delta(A,B) = A \cup B \ - \ (A \cap B).$$

Soit Δ_k ($A_1,\ldots,A_k$) la différence symétrique entre les k ensembles A_1, …, A_k, est définie récursivement par :

$$\Delta_2(A_1,A_2) = \Delta(A_1,A_2).$$
$$\Delta_k(A_1,\ldots,A_k) = \Delta((\Delta_{k-1}(A_1,\ldots,A_{k-1})) , A_k).$$

Soit P un CSP et *V* l'ensemble des variables de P. Soit alors la fonction F de l'ensemble des parties de *V*, à valeurs dans *V* définie par : $F \ : \ \wp(V) \rightarrow V$

$$F(A_1,A_2) = A_1 \cap A_2.$$
$$F(A_1,A_2,A_3) = \Delta(A_1,A_2) \cap A_3.$$
$$F(A_1,\ldots,A_p) = (\Delta_{p-1}(A_1,\ldots,A_{p-1})) \cap A_p.$$

F exprime le fait que dans une composition sur un système de contraintes, on cherche les variables communes pour chaque couple de contraintes. F va nous servir pour la définition ensembliste de la composition *nlinRec*.

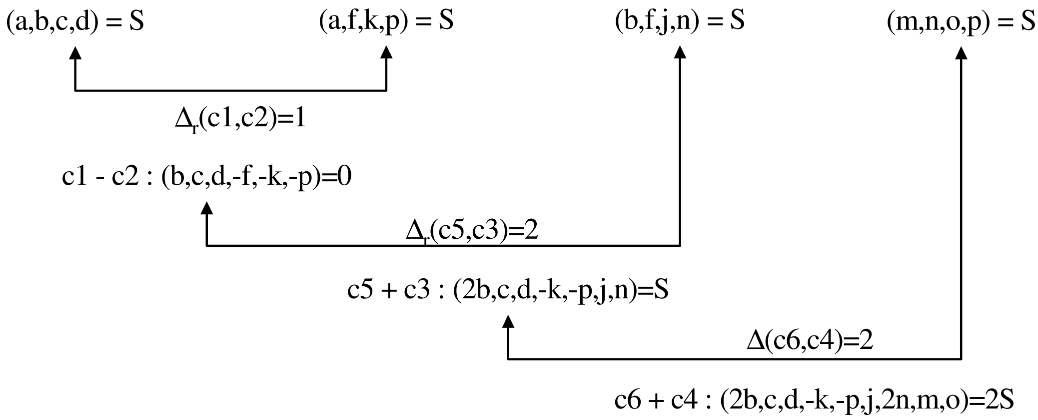
Définition 3.3.4-2 : connectivité d'un système de contraintes

La connectivité d'un système $A_1,\ldots,A_p$ de contraintes est la taille de $F(A_1,\ldots,A_p)$. La connectivité mesure la taille en nombre de variables du système après composition.

Exemple : carré magique 4x4

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

`(1)a+b+c+d = S, (2)a+f+b+p = S, (3)b+f+j+n = S, (4)m+n+o+p = S.`



Vision ensembliste de linlin

Dans le cas idéal, la composition binaire peut être vue comme l'application de l'opérateur Δ sur les deux ensembles représentant les contraintes car Variables(c_1 - k*c_2) = Δ(Variables(c_1),Variables(c_2)).

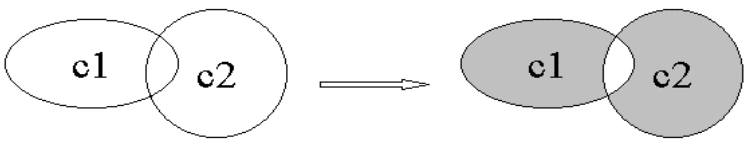


Figure 9 : compose(C1, C2) = Δ(C1,C2)

Mais dans le cas réel, l'opérateur appliqué n'est pas exactement Δ car il existe un sous-ensemble de $C_1 \cap C_2$ qui appartient à la contrainte résultat [49]:

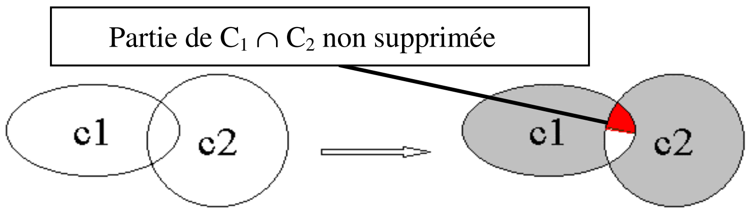


Figure 10 : composition non idéale. $C_1 \cap C_2$ est non vide. MaxRemovedVariables est la partie de $C_1 \cap C_2$ qui est supprimée.

La distance au cas idéal *i.e.* la différence de taille entre la contrainte générée réellement et la taille minimale possible, mesure l'efficacité d'une composition. Mais elle suppose de connaître la contrainte générée, c'est à dire de réaliser déjà une partie de la composition[50].

Vision ensembliste de nlinRec

Généralisons l'opération de composition d'ensemble à *k* contraintes. Formellement on définit la composition de *k* contraintes C_1, .. C_k par l'application des opérateurs

$$\Delta_k(C_1,...,C_k) = \Delta(\Delta_{k-1}(C_{\alpha_1},...,C_{\alpha_{k-1}}),C_{\alpha_k}) .$$

[49] C'est le décalage entre le cas réel et le cas idéal.
[50] La partie la plus couteuse.

Dans le cas idéal, la contrainte générée ne possède aucune des variables communes à au moins deux contraintes parmi les k contraintes. Voici quelques exemples de situations idéales :

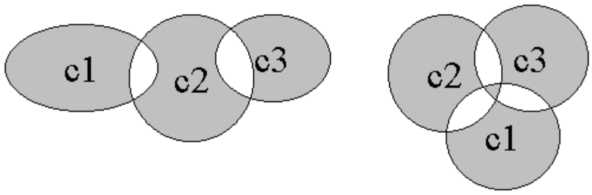


Figure 11 : composition idéale des contraintes 1,2, puis 3. On obtient le même résultat si on compose (cas idéal) 1,3 puis 2.

Exemple : Soit le système (L1,L2,L3) de trois contraintes : $\begin{cases} x + 2y + u = 4 \\ -x - t + z = 0 \\ -3x + y \geq 20 \end{cases}$.

La composition (L3 -3L2) est idéale et produit $-3z + 3t + y \geq 20$; (L1 + L2) est également idéale. Mais (L3 + 3L1) ne l'est pas car on ne peut retirer toutes les variables communes. Si une composition des trois contraintes était idéale, la contrainte résultante porterait uniquement sur u et z. Trivialement les compositions ternaires ne sont pas idéales[51].

Propriétés

Dans cette section, on notera χ.*coefficient*(ϖ) le coefficient de la variable ϖ dans le polynôme de la contrainte linéaire χ. On note *arity*(χ) le nombre de variables de la contrainte χ. On note *CommonVariables*(χ_1, χ_2) l'ensemble des variables communes aux deux contraintes χ_1 et χ_2. Par hypothèse, l' heuristique de composition binaire est *maxRemoved*.

Soit *MaxRemovedVariables* l'ensemble des variables supprimées lors de la composition de χ_1 et χ_2. Alors on a toujours *MaxRemovedVariables* $\subset$ *CommonVariables*(χ_1, χ_2). Dans le cas idéal, toutes les variables communes disparaissent, *i.e.* *MaxRemovedVariables* = *Commonvariables*(χ_1, χ_2).

Propriété 3.3.4-1

> Pour tout couple de contraintes (c_1,c_2), $\Delta(C_1,C_2)$ est une borne inférieure de l'ensemble de toutes les contraintes productibles par composition binaire. Pour tout k entier, *arity*(c_1-k*c_2) ≤ $|\Delta(C_1,C_2)|$.

Preuve :

Dans le meilleur des cas, pour tout v de *CommonVariables*(c_1,c_2), on a c_1.*coefficient*(v) = k*c_2.*coefficient*(v), où k est le ratio de la composition. Dans le cas réel, au moins une variable commune reste donc $(C_1 \cap C_2) \supseteq$ *MaxRemovedVariables* et on a :

arity(c_1 –k*c_2) = $|C_1| + |C_2|$ - |*MaxRemovedVariables* $\cap C_2$ | - |*MaxRemovedVariables* $\cap$ C1 |.

arity(c_1 –k*c_2) ≤ | $C_1 \cup C_2$ – (*MaxRemovedVariables* $\cap C_2$) $\cup$ (*MaxRemovedVariables* $\cap C_1$) |

autrement dit,

[51] Par exemple, (L3 -3L2)+ (-½)L1) produit -3z + 3t - ½ x ≥ 18 ; (- ½ (L1+L2)+L3) produit –3x – ½ z + ½ t ≥ 18.

arity(c_1 –k*c_2) $\leq |C_1 \cup C_2 - (MaxRemovedVariables \cap (C_1 \cup C_2))|$
Or *MaxRemovedVariables* $\subseteq$ *CommonVariables*(c_1,c_2) $\subseteq (C_1 \cup C_2)$.
D'où *arity*(c_1 –k*c_2) $\geq |\Delta(C_1,C_2)|$.
De plus toute contrainte non commune à c_1 et c_2 apparaît dans c_1-k*c_2 . Par conséquent il n'existe pas de contrainte résultante d'arité inférieure à $|\Delta(C_1,C_2)|$.

Propriété 3.3.4-2

Généralisons la propriété 1 à n contraintes :

> Pour tout *n* entier, *arity*(compose(c_1,...,c_n)) $\leq |\Delta_n(C_1,..., C_n)|$.

On peut prouver ce résultat par récurrence sur *n*. La propriété précédente assure que la base de la récurrence pour *n*=2 est vraie.

Propriété 3.3.4-3

> F n'est pas symétrique.
> Pour tout ensemble i, j, k, l, on a F(...,i,j,k,l,...) $\neq$ F(...,i,k,j,l,...).

Preuve :

Il suffit de trouver un contre-exemple. Prenons trois ensembles comme sur la figure suivante :

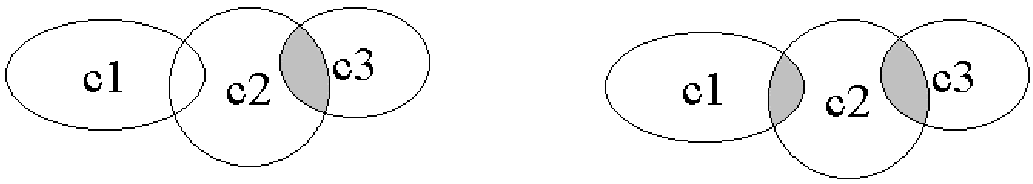


Figure 12 : non symétrie de F

$F(C_1,C_2,C_3) = C_2 \cap C_3$ tandis que $F(C_1,C_3,C_2) = (C_1 \cap C_2) \cup (C_2 \cap C_3)$.

Corollaire

La composition *nlinRec* calcule les intersections entre les contraintes en utilisant la fonction F. Elle n'est donc pas symétrique. En général le résultat d'une composition *nlinRec* dépend de l'ordre des compositions binaires intermédiaires.

Théorème 3.3.4-1

> Pour tout p, $\Delta_p(A_1,...,A_p)$ est l'union des sous-ensembles appartenant à un nombre impair d'ensembles parmi les A_1,..., A_p.

Preuve par récurrence :

Pour p = 1 :
$\Delta_1(A_1) = A_1$. Le résultat appartient à un seul ensemble ; Δ_1 est l'identité. La propriété est vraie.

Supposons le théorème vrai pour p-1 et montrons que la propriété et encore vraie pour p :
On a $\Delta_p(A_1,...,A_p) = (\Delta_{p-1}(A_1,...,A_{p-1})) \cup A_p - ((\Delta_{p-1}(A_1,...,A_{p-1})) \cap A_p)$ par définition de Δ_n. Par hypothèse de récurrence, $\Delta_{p-1}(A_1,...,A_{p-1})$ est l'union de sous-ensembles

appartenant à un nombre impair d'ensembles. Donc $\Delta_{p-1}(A_1,...,A_{p-1}) \cap A_p$ contient des sous-ensembles appartenant à un nombre pair d'ensembles. En effet ces sous-ensembles appartiennent à A_p, et aussi aux ensembles de $\Delta_{p-1}(A_1,...,A_{p-1})$, qui sont en nombre impair par hypothèse. Pour calculer $\Delta_p(A_1,...,A_p)$, on enlève ces sous-ensembles ; donc il ne reste plus que les sous-ensembles appartenant à un nombre impair d'ensembles. Donc le théorème est vrai au rang p.
Finalement le théorème est vrai pour tout p entier positif strictement.

Propriétés de symétrie de Δ_p

Pour tout *p* entier, L'opérateur Δ_p est symétrique par permutation de deux de ses arguments. On a toujours la relation suivante :

$$\Delta_p(\ldots,i,j,k,l,\ldots) = \Delta_p(\ldots,i,k,j,l,\ldots).$$

La propriété s'exprime plus généralement. Pour tout sous-ensemble Φ de q contraintes ($q \leq p$), pour toute permutation σ des éléments de ce sous-ensemble, on a toujours $\Delta_p(\ldots,\Phi,\ldots) = \Delta_p(\ldots,\sigma(\Phi),\ldots)$.

Preuve :

On va montrer que, quel que soit l'ordre des ensembles, on obtient toujours le même résultat en appliquant l'opérateur Δ_k. L'idée est d'exprimer $\Delta_k(\ldots,\Phi,\ldots)$ d'une autre manière, équivalente, qui soit indépendante de l'ordre des arguments, ce qui n'est pas le cas de la définition que nous avons donnée précédemment. D'après le théorème 3.3.4-1, si A est l'ensemble des arguments de Δ_k (contraintes), pour tout sous-ensemble S inclus dans A, S appartient à la différence symétrique Δ_k si et seulement s'il appartient à un nombre impair d'ensembles parmi A. Par conséquent le résultat est indépendant de l'ordre des arguments, et Δ_k est symétrique par permutation.

Corollaire

Pour tout p entier, Δ_p est symétrique. Cela signifie que dans le cas idéal, la composition *nlinRec* de *n* contraintes (n>2) est symétrique. Autrement dit, une fois un ensemble de p contraintes choisi, dans le problème, l'ordre des compositions ne change pas le résultat final.

Ces résultats sont valables dans le cas idéal. Mais en pratique, il existe un décalage entre la contrainte générée et la différence symétrique des contraintes sources. Cette différence est la cause de la non symétrie de la composition n-aire dans le cas réel.

Considérons l'exemple de deux compositions du carré magique 4x4 ci-dessous.

(a,b,c,d) = S (a,f,k,p) = S (b,f,j,n) = S (m,n,o,p) = S

d_{inter}(c1,c2)=1

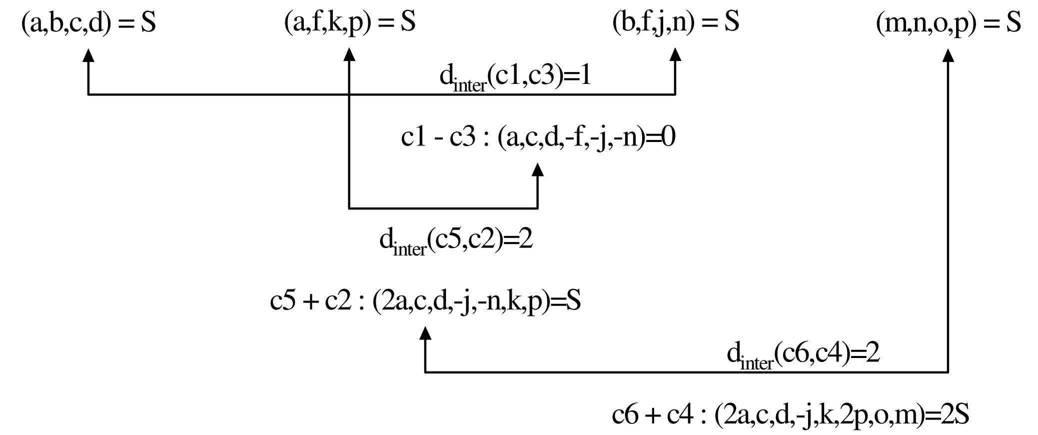
c1 - c2 : (b,c,d,-f,-k,-p)=0

d_{inter}(c5,c3)=2

c5 + c3 : (2b,c,d,-k,-p,j,n)=S

d_{inter}(c6,c4)=2

c6 + c4 : (2b,c,d,-k,-p,j,2n,m,o)=2S



Ici, les deux compositions ne sont pas équivalentes mais l'arité de la contrainte générée est la même quel que soit l'ordre de composition.

3.3.5 Réécriture d'un système de contraintes linéaires (Gauss-Simplexe)

La méthode d'élimination de Gauss permet de transformer un système d'équations linéaires en conservant une équation et en supprimant au moins une variable des autres. La méthode du Simplexe est basée sur le même principe mais s'applique aux inéquations en ramenant le système à un système d'équations. Cette section propose une composition, *Gauss-Simplexe*, qui s'applique sur des systèmes de contraintes hétérogènes contenant à la fois des équations et des inéquations. *Gauss-simplexe* choisit itérativement une contrainte, dite pivot, réécrit le système en éliminant au moins une variable de la contrainte pivot, des autres contraintes.

Ce n'est pas tant l'algorithme qui est coûteux, que la recherche du système de contraintes sur lequel l'exécuter. Le système de contraintes peut être construit par résolution d'un CSP (C.f. chapitre IV-2). Dans ce cas les contraintes du CSP décrivent la topologie des contraintes qui seront incluses dans le système et le nombre de contraintes à prendre. L'utilisation des CSP pour décrire les conditions de construction du système permet d'essayer différents critères. Un mécanisme de contrôle est alors envisageable pour choisir le critère approprié en fonction de l'état du problème. Si le système est construit à partir d'une contrainte C donnée, il faut au moins choisir autant de contraintes que C a de variables ; de plus toutes les variables de C doivent apparaître dans au moins une de ces contraintes. Le chapitre IV-1 propose des fonctions d'évaluation de la densité d'un problème qui sont applicables à un système de contraintes linéaires.

Exemples :

Pour *n* fixé, choisir *n* contraintes telles que la $d_{intersection}(C_1,\ldots, C_n)$ est maximum.
Pour *n* fixé, choisir *n* contraintes telles que pour tout $i \neq j$, $d_0(C_i,C_j)$ est maximum.

Principe

L'élimination d'une variable pour chaque contrainte pivot est réalisée par l'opération de composition binaire compose(ci, cp_{ivot}) avec pour stratégie de supprimer au moins une variable et de calculer un seul ratio (*doOneMaxRemoved*). Cependant une restriction sur ci-k*c_{pivot} doit être imposée dans le cas où la contrainte pivot est une inégalité : 1) k doit être négatif si c_{pivot} est une inégalité ; 2) le sens de l'inégalité est conservé dans la contrainte résultat ; 3) l'équivalence entre le système initial et le système final est perdue. En effet quand le pivot est une inégalité, compose(c_i,c_{pivot}) transforme les égalités du système en inégalité ; le nouveau système a donc un espace de solutions plus large que le système d'origine. Cela nous amène à considérer *Gauss-Simplexe* comme une propagation au lieu d'une réécriture, dans le cas où l'on autoriserait les inégalités à jouer le rôle de contrainte pivot.

Voici un algorithme réalisant la composition par *Gauss-Simplexe :*

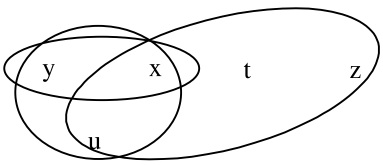
`La fonction calculePivot(cPivot)` renvoie la première variable de la contrainte en argument, capable de servir de pivot. Par défaut il s'agit de la première contrainte commune aux autres contraintes de S. La fonction `compose(C1, C2, variablePivot)` laisse c_1 et c_2 intactes et renvoie la contrainte $c_1 - \left(\frac{c_1.coefficient(variablePivot)}{c_2.coefficient(variablePivot)}\right) * c_2$.

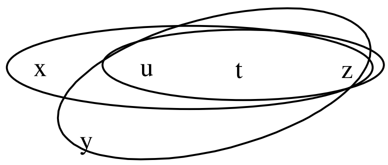
```
ComposeGauss-Simplexe(n, S)
Données : n = entier ; S = ensemble de n contraintes linéaires
Variables locales : c_pivot= contrainte pivot; v_pivot= variable de c_pivot
Début
   Pour tout k dans {1,…,n-1}, Faire
      c_pivot := S(k)          "kème élément de S"
      v_pivot := calculePivot(c_pivot)
      Pour tout j dans {1,…,n}, j ≠ k, S(j) := compose(k,j, v_pivot)
  Finfaire
Fin
```

Exemple 3.3.5-1 : réécriture par Gauss-Simplexe

Système initial : S   Système final : S'

$$\begin{cases} x+2y+u=4 \\ -x-z+t+u=0 \\ -3x+y\geq 20 \end{cases} \rightarrow \begin{cases} (1): x+2y+u=4 \\ (1)+(2): 2y-z+t+2u=4 \\ 3(1)+(3): 7y+3u\geq 20 \end{cases} \rightarrow \begin{cases} (1)-(2): x+z-t-u=0 \\ (2): 2y-z+t+2u=4 \\ 3(3)-7(2): 7z-7t-8u\geq 36 \end{cases}$$





Gauss-Simplexe est utilisé en prétraitement de reformulation d'une partie du problème.

Propriété

La connectivité du système mesure les intersections entre les contraintes. Nous constatons expérimentalement que *Gauss-Simplexe* génère un système de connectivité plus élevée. De manière générale, la valeur de la fonction F est toujours l'ensemble vide sur le système résultant (C.f. propriété 3.3.5-1).

Définition 3.3.5-1 : intersection deux à deux

L'intersection deux à deux est la somme des intersections des contraintes d'un système donné, avec une contrainte de référence.

Reprenons le système de l'exemple 3.3.5-1.

Système contraint	Arité de F	Intersections deux à deux
Avant Gauss- Simplexe	1 ou 2	4
Après Gauss- Simplexe	Toujours 0	6

F n'étant pas symétrique, sa valeur change selon l'ordre dans lequel on prend les contraintes : F(1,2,3) = {y} ; F(2,3,1) = {y,u} ; F(3,1,2) = {u}.

Propriété

Pour tout système S obtenu par application de la composition *Gauss-Simplexe*, F(S) = 0.

Preuve:

Soit $S_0 = \{A_1,\ldots, A_{p-1}\} \cup \{A_p\}$. $F(S_0)$ est définie par l'intersection de $(\Delta_{p-1}(A_1,\ldots,A_{p-1}))$ et de A_p. *Gauss-Simplexe* transforme S_0 en isolant une variable pivot et en la supprimant de toutes les autres contraintes de S_0, par conséquent dans le système final, S'_0, il y a *p-1* variables (pivot) qui n'apparaissent que dans une seule contrainte (la dernière contrainte n'est pas composée). Les autres variables du système sont communes à au moins deux contraintes ; la différence symétrique $\Delta_{p-1}(A_1,\ldots,A_{p-1})$ est donc identique à l'ensemble des variables pivot. D'où l'intersection entre Δ_{p-1} et A_p est vide car A_p ne contient aucune des variables pivot du système $\{A_1,\ldots, A_{p-1}\}$. Finalement $F(S_0) = 0$.

3.4 Compositions de contraintes hétérogènes

Cette section présente deux classes de règles de propagation binaire : la première combine une égalité linéaire et une contrainte de différence ; la seconde combine une égalité linéaire et une contrainte primitive "produit". Le contexte d'application de ces règles n'étant pas imposé, elles peuvent donc être utilisées statiquement ou dynamiquement. Dans le cas où le moment de déclenchement de ces règles est restreint à l'affectation d'une variable, ces règles réalisent une propagation de valeurs comparable à la propagation de *dataflow constraints* [Hoover 1987].

3.4.1 Composition d'une égalité linéaire et d'une différence n-aire (LinDiff)

Elle regroupe trois règles primitives :

Détection d'incohérence : x = y et AllDifférent(...,x,....,y,....) → Incohérence.

Maintien de cohérence : k.x = y ($k \neq 0$) et AllDifférent(...,x,...,y,...) → $x \neq 0$ et $y \neq 0$

Maintien de cohérence : ax = ay + k et AllDifférent(...,x,...,y,...) → $k \neq 0$

La règle *LinDiff* sélectionne les règles primitives de la manière suivante : si la contrainte active est une différence alors appliquer les trois règles ; si la contrainte active est une égalité alors appliquer les règles qui correspondent.

Application aux fonctions injectives

Dans le cas des fonctions injectives, la cohérence partielle des combinaisons linéaires de termes fonctionnels n'est pas optimale. La règle *linDiff* améliore ce filtrage. Le filtrage d'une combinaison linéaire $\sum \alpha_i x_i = \beta$ est donné par deux algorithmes basés sur le calcul de la valeur minimale (*min*) et maximale (*max*) de la somme. Voici l'algorithme de filtrage quand la borne minimum d'une variable a changé.

Procédure FiltrageBorneMinimum
Début
min := *min*($\sum \alpha_i v_i$), $\forall v_i$, valeur possible de x_i dans *dom*(x_i)).

Pour toute variable x_j, **si** $\alpha_j > 0$ **alors** $x_j \leq \left(\frac{\beta - \min}{\alpha_j}\right)$

sinon $x_j \geq \left(\frac{\beta - \min}{\alpha_j}\right)$

Fin

Or dans le cas où les x_i sont toutes différentes, le problème revient à la détermination d'une fonction F injective de $\{x_i\}_i$ vers I, I étant un intervalle contenant autant d'éléments qu'il y a de x_i. Considérons la fonction V telle que $V(x_i) = min(x_i)$ si $\alpha_i>0$ et $V(x_i) = max(x_i)$ si $\alpha_i<0$, et la fonction *W* telle que $W(x_i) = max(x_i)$ si $\alpha_i>0$ et $W(x_i) = min(x_i)$ si $\alpha_i<0$.

F étant injective, on déduit que, s'il existe au moins deux variables x_1 et x_2 telles que $V(x_1) = V(x_2)$, *min*($\sum \alpha_i F(x_i)$) > $\sum \alpha_i . V(x_i)$. En effet, le n-uplet $(\ldots,V(x_1),V(x_2),\ldots)$ n'est trivialement pas solution. De même s'il existe au moins deux variables x_1 et x_2 telles que $W(x_1) = W(x_2)$, alors *max*($\sum \alpha_i F(x_i)$) < $\sum \alpha_i . W(x_i)$.

Par ailleurs le calcul de *min*($\sum \alpha_i F(x_i)$) (et *max*($\sum \alpha_i F(x_i)$)) est basé sur un algorithme de type *Enumeration-Test*. Pour calculer *min*($\sum \alpha_i F(x_i)$), le principe est le suivant :

1) essayer toutes les possibilités en remplaçant successivement chaque occurrence de variable par sa borne inférieure ;
2) quand la valeur a déjà été choisie, prendre la valeur suivante dans le domaine[52] ;
3) une fois le calcul fait pour une possibilité, recommencer en essayant de trouver une valeur de *min* plus petite.

Exemple :

Soit $x_1 = \{a_1,a_2,a_3\}$, $x_2 = \{a_1,a_4,a_5\}$, $x_3 = \{a_2,a_3\}$, où pour tout i, a_i est une valeur et $a_i < a_{i+1}$, et b_1,b_2,b_3 strictement positifs. On déduit que
min$(b_1x_1 + b_2x_2 + b_3x_3)$ = *min*$\{b_1a_1 + b_2a_4 + b_3a_2,\ b_1a_2 + b_2a_1 + b_3a_3,\ b_1a_3 + b_2a_1 + b_3a_2\}$.

Ce calcul peut être optimisé, comme en Alice, en représentant explicitement la fonction F. En Alice, la contrainte de différence est représentée par des cliques, ce qui permet de prendre en compte la contrainte d'injection dans le filtrage des égalités linéaires. En quelque sorte, Alice combine la contrainte *AllDifférent* avec toute égalité linéaire. Néanmoins ce calcul a une complexité non négligeable[53] ($O(d^n)$); de plus, d'après nos expériences, le gain en réduction de domaine par rapport au temps de calcul est modeste. C'est pourquoi, dans le

[52] Les domaines sont supposés ordonnés.
[53] Pour une combinaison linéaire de *n* variables dont le domaine le plus large contient *d* valeurs.

cas où une contrainte *AllDifférent* relie les variables, le filtrage par cohérence partielle ne calcule pas les bornes les plus «contraignantes ». Pour compenser cette perte d'information, une solution consiste à ajouter une contrainte d'égalité sur toutes les variables de la combinaison linéaire ; la règle de propagation unaire *BijectionRule* réalise cet ajout.

3.4.2 Composition linéaire-Produit

Soit la règle générique, x = $\sum_i a_i y_i$ et $x * \prod v_j$ = T → $\sum_i (a_i y_i \prod v_j)$

Condition d'application : il existe une variable x à la fois facteur d'un produit et contrainte par une égalité linéaire.

Cette règle est définie pour toutes les égalités linéaires mais elle est d'utilisation difficile et coûteuse telle quelle. Les instance utilisées en pratique mettent en jeu des égalités du type x = EXP(y) où EXP est d'arité petite (inférieure à 3) et des produits binaires. Si elle est contrôlée avec la stratégie consistant à remplacer la variable la moins contrainte, cette règle augmente les interconnexions du problème [Laurière 1976].

Voici deux instances utilisées en pratique :

$$x = a.y + c\ ,\ \ x\ .\ y = T \rightarrow \quad a(y\ .\ y) + c.y = T$$
$$x = y \text{ et } x\ .\ y = T \quad \rightarrow \quad y\ .\ y = T$$

Cette règle est une propagation sur deux contraintes primitives. Son utilisation, par exemple, dans l'approximation de coordonnées réelles de points dans un référentiel permet d'améliorer la précision des valeurs des coordonnées polaires.

4 Contrôle de RCS par adaptation à la résolution

Les expériences décrites au chapitre II, ont montré la nécessité d'un contrôle sur les règles de raisonnement symbolique ; ce contrôle se devait d'être facile à spécifier, déclaratif et capable d'évoluer en cours de résolution, pour pouvoir s'adapter à de nouvelles situations favorables au raisonnement symbolique. Dans ce chapitre, nous proposons une architecture de spécification de stratégies pour répondre à ce besoin. Une telle stratégie est définie par un contrat entre un ensemble de règles (éventuellement une seule) et un moteur CSP. Les éléments du contrat sont la règle et la contrainte, reliés par un ensemble de clauses. De plus les clauses des stratégies sont évaluées à des moments prédéfinis de la résolution : lorsqu'une contrainte vient d'être propagée ou ajoutée, lorsqu'une règle devient candidate (C.f. section 2.1).

Le contrôle est d'autant plus adapté qu'il est capable de prédire l'évolution d'un solveur dans la prochaine étape de résolution, c'est à dire qu'il dispose de moyens calculatoires pour évaluer l'action d'un solveur et en déduire son utilité *a priori*. Très souvent ces calculs nécessitent des informations globales sur l'état du système (problème, utilité des solveurs dans les traitements effectués jusqu'alors). Le modèle de contrôle dynamique de Meta-RCS est fondé sur la représentation générale d'un solveur symbolique donnée au chapitre III-1. I ; elle expoite les méta-connaissances traitées dans les parties pre-conditions et post-actions du solveur. Ces deux parties constitue le point de liaison des stratégies aux règles de RCS : à la fois pour la mise à jour des stratégies dynamiques, la notification de changement de statut du solveur, l'action des stratégies.

Par ailleurs la relation est susceptible d'évoluer en fonction des exécutions de ses solveurs, ce qui permet de représenter des stratégies dynamiques qui tiennent compte de la qualité du résultat produit par un solveur. De manière informelle la relation représente l'état relatif d'un ensemble de règles, qui doit être maintenu dans le système CSP+RCS. Pour représenter une stratégie nous avons eu recours au concept de *méta-contrainte*, c'est à dire des contraintes portant sur des règles et des contraintes. Ces méta-contraintes sont essentiellement testées ; leur propagation consiste à faire un effet de bord. Elles s'apparentent ainsi à des *invariants* au sens des invariants définis dans Localizer [Michel & Van Hentenryck 1997][54]. Le maintien de la relation entre plusieurs solveurs est incrémental : les caractéristiques de la relation sont mises à jour en fonction des actions des mêmes solveurs. Pour ce faire, le modèle intègre un mécanisme réactif reliant une stratégie à un ensemble d'événements prédéfinis, susceptible de survenir durant la vie d'un solveur. Par exemple, considérons la stratégie imposant à la règle R_0 de se déclencher quand il y a eu au moins k échecs dans S={R_1,R_2,R_3}. Nous la modélisons par un objet qui 1) bloque R_0, 2) comptabilise le nombre d'échecs à chaque fin d'exécution d'une règle de *S* et 3) débloque R_0 si ce compteur atteint *k* (C.f. section 4.2).

4.1 Analyse comportementale du système CSP+RCS

Le système CSP+RCS se comporte de manière réactive autour de trois acteurs, le solveur CSP, la contrainte et le solveur RCS, chaque règle de RCS étant un sous-solveur. La réactivité du système est obtenue grâce à un mécanisme de démons et à l'explicitation d'événements décrivant la vie des trois acteurs (C.f. figure ; les acteurs communiquent entre eux *via* ces événements grâce aux démons. Les informations échangées concernent la manipulation du problème ; elles représentent les contraintes dont le statut change (ajoutée, candidate à un raisonnement symbolique) et influence l'action des solveurs. Par exemple, à chaque événement, seuls les solveurs symboliques capables de manipuler la contrainte idoine sont exécutés.

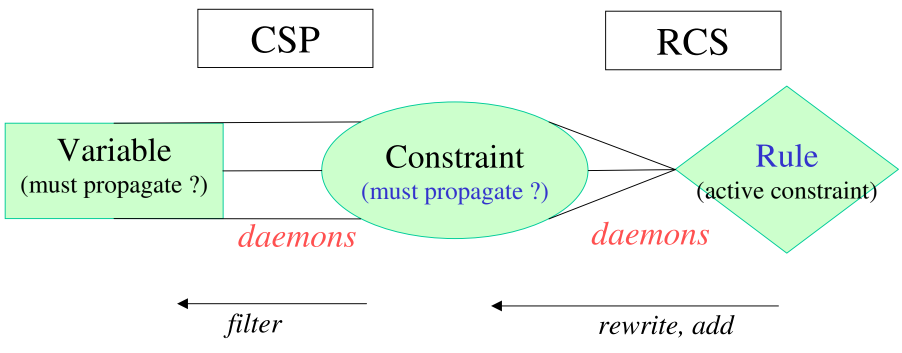


Figure 13 : intégration de CSP, RCS et Méta-RCS

Dans cette section, nous avons extrait les différents états d'un solveur qui interviennent dans la construction de stratégies globales. L'échec et le succès d'un solveur sont caractérisés

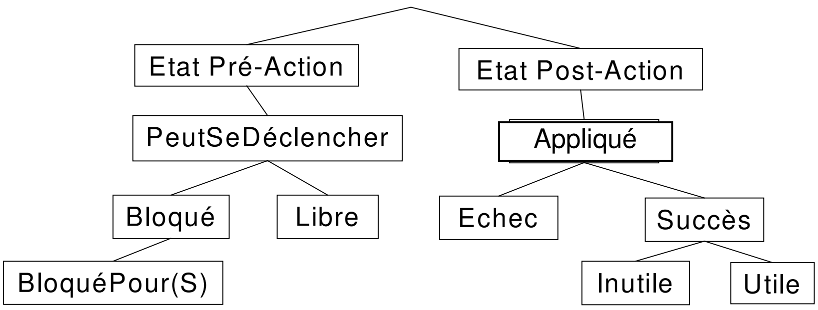
[54] Dans le langage Localizer, un invariant est une variable dont la valeur est définie par une fonction n-aire d'autres variables.

par rapport à l'effet attendu sur la résolution ; dans le cas de CSP, nous évaluons la réduction de domaine lors du filtrage des contraintes modifiées ou ajoutées dans le problème.

Etats	Caractérisation
PeutSeDéclencher	L'exécution est sur le point de s'appliquer.
Echec	La dernière exécution se termine par un échec.
Succès	La dernière exécution se termine par un succès.
Appliqué	L'exécution vient de s'appliquer. *Appliqué* est un événement abstrait dont *Echec* et *Succès* sont deux particularisations.
Bloqué	Le solveur est interdit d'exécution.
BloquéPour(S)	Le solveur est interdit d'exécution sur les contraintes de l'ensemble S.
Libre	Le solveur n'est pas *bloqué*.

Tableau 12 : Etats d'un solveur

Dans le cas d'un solveur symbolique (une règle de RCS), les états sont hiérarchisés. Les deux événements principaux sont *PeutSeDéclencher* et *Appliqué*. *Appliqué* se particularise en deux événements, *Succès* et *Echec*, selon l'issue de l'application de la règle.



Les états traduisent la capacité et l'autorisation du solveur à s'exécuter, ainsi que la caractérisation du résultat après son exécution.

Deux statuts caractérisent une règle de RCS selon l'utilité de son exécution. Dans ce document, nous considérons une définition particulière de l'utilité d'une règle : une règle est considérée comme *utile* si elle provoque au moins une réduction de domaine. Si une règle échoue alors toutes ses instances sont inutiles. Si au moins une instance de règle est utile, alors la règle réussit. Le statut *Echec* implique l'état *Inutile*, de même que *Utile* implique *Succès*.

4.1.1 Vie d'une contrainte

Comme nous l'avons vu au chapitre I-1, dans le cadre de la programmation par contraintes et objets, la notion de contrainte encapsule à la fois la définition logique et la définition procédurale. Autrement dit une contrainte réifie à la fois l'objet algorithmique responsable de la méthode cohérence et la donnée que manipulent les règles de réécriture. Ce double aspect est réutilisé au niveau des stratégies : le contrôle de la collaboration entre RCS et le moteur CSP implique la partie procédurale de la contrainte, tandis que le contrôle d'une règle de RCS considère la contrainte en tant que donnée présente dans une situation.

Etats d'une contrainte dans CSP+RCS

Etats	Spécification
Ajoutée	Quand la contrainte vient d'être ajoutée dans le solveur, elle se connecte aux variables par des démons -Min, Max, Domain, Value, Remove.
Obsolète	Quand la contrainte est retirée du solveur mais pas déconnectée des démons, elle est obsolète. Toute contrainte réécrite devient obsolète.
Supprimée	Quand la contrainte est réécrite, elle devient d'abord obsolète. Puis elle est déconnectée des démons. Elle est à ce moment complètement supprimée du solveur.
CSP-active	la contrainte est CSP-active à chaque fois qu'une de ses variables est modifiée par sa méthode de cohérence locale.
Active	La contrainte est active quand elle initie un raisonnement symbolique.
Passive	La contrainte est impliquée dans un raisonnement symbolique mais n'est pas active.
FilteringOn	La contrainte est associée à une technique de réduction de domaines.
RSOn	La contrainte est reliée à au moins une règle de RCS.

Tableau 13 : états d'une contrainte (C.f. chapitre I-1)

Les états *filteringOn* et *RSOn* sont fixes *a priori* : la liste des méthodes de traitement d'une contrainte est connue dès sa création puisqu'elle ne dépend que de la classe de la contrainte. Toute contrainte est *filteringOn*, si l'on se place dans un *framework* de satisfaction de contraintes. Une contrainte devient *active* quand elle est ajoutée au solveur, elle résulte alors d'une réécriture ou d'une propagation. Dans tous les autres cas, une contrainte est *inactive*. En particulier, si une contrainte est *obsolète* alors elle est aussi *inactive*.

4.1.2 Vie d'un solveur symbolique

Les règles de RCS sont reliées au solveur CSP de manière paresseuse. L'ensemble des contraintes candidates des règles est mis à jour à chaque fois qu'une nouvelle contrainte est ajoutée. Certaines règles comme les règles de simplification n'ont pas besoin d'être conservée car elles sont spécifiques d'une contrainte.

D'un point de vue objet, la règle peut être vue sous deux aspects, à la fois une procédure déductive et un objet manipulable par le solveur, par exemple pour choisir la règle à appliquer lors du contrôle de la collaboration.

Statut d'une règle

Une règle de RCS possède les états d'un solveur symbolique. En particulier, une règle *peut se déclencher* si elle possède au moins une contrainte active. L'issue du déclenchement d'une règle peut être récupérée *via* les deux statuts *Succès* et *Echec.* Deux statuts particuliers correspondent au moment où la règle est déclarée utile ou inutile selon un certain critère redéfinissable. Par exemple, le statut *Utile* peut signaler que l'exécution de la règle a provoqué au moins une réduction de domaine, instantiation ou incohérence. Quand son exécution n'a pas changé le problème, la règle est mise dans l'état *Inutile.*

Une règle ignore si elle a changé de statut. Il faut donc le lui dire explicitement. Les statuts *Succès*, *Echec*, *PeutSeDéclencher* et *Appliqué* sont signalés automatiquement. Néanmoins il n'est pas obligatoire pour une règle de connaître son statut ; cela devient nécessaire lorsque les règles sont contrôlées. En effet la réactivité du contrôle dépend de la génération des événements. Si la règle n'a pas été avertie de son changement de statut, les stratégies ne fonctionnent pas. La règle échoue quand aucun n-uplet de prémisses n'a été trouvé. Par défaut elle réussit lorsque l'application parvient à terme.

4.2 Méta-RCS, un modèle de stratégie dynamique pour RCS

La spécification de stratégies poursuit un objectif simple : pouvoir manipuler un raisonnement symbolique sans devoir gérer des connaissanes globales à tout le système. Dans cette section nous présentons l'architecture de *méta-RCS*, pour spécifier un contrôle dynamique souple sur le solveur de résolution CSP+RCS.

Dans notre approche une stratégie abstrait une relation invariante entre les solveurs ; elle est mise en œuvre par une contrainte, qui encapsule les données nécessaires au maintien de la relation. La donnée par défaut est le solveur principal sur lequel porte la stratégie. Dans notre cas, les stratégies servent à contrôler RCS ; par conséquent le solveur principal est toujours un solveur symbolique. Afin de représenter de manière uniforme les stratégies de collaboration entre les règles de RCS, nous avons réutilisé le concept de *méta-contrainte* : une *méta-containte* est vue ici comme une relation globale qui doit être vérifiée à certains moments de la résolution, dédiée au contrôle. Ellle influence l'algorithme de résolution en autorisant ou interdisant le déclenchement d'une règle. De plus elle est utilisée de manière déclarative et réactive : chaque règle s'enregistre comme étant pilotée par un invariant de contrôle, celui-ci étant caractérisé par un ensemble de réactions à des événements survenant au cours de l'application de ses règles. Ce modèle permet un contrôle précis et dynamique ainsi qu'une collaboration à une fine granularité.

Nous définissons une méta-contrainte dédié au contrôle ou «contrainte de contrôle », par une contrainte globale sur des variables représentant l'état des solveurs : ContrôleCt(s_1,..,s_k), où $s_i \in$ {*PeutSeDéclencher, Echec, Succès,…, BlockéPour*}; sa propagation a lieu à chaque fois qu'un des solveurs est sur le point de s'exécuter sur une contrainte active. Cette contrainte doit être vérifiée pour que le solveur se déclenche. La notion de « contrainte de contrôle » est proche de la notion d'invariant de [Michel & Van Hentenryck 1997] : l'état des solveurs est mis à jour incrémentalement durant la résolution en fonction de l'évolution du système CSP+RCS. Ce mécanisme permet de déclarer des stratégies dynamiques.

Exemple 4.2-5 : collaboration entre maintien de cohérence et manipulation symbolique

Considérons la contrainte ensembliste [$X \in S$], où $S=\{a_1,a_2,...,a_n\}$; son filtrage est peu opérationnel si l'ensemble S n'est pas ordonné. Par conséquent les règles d'inclusion sur les ensembles sont utiles pour déduire des contraintes redondantes [Müller & Müller 1997]. Celles-ci le sont beaucoup moins quand la taille de S est petite par rapport au domaine de X ; dans ce cas, il est plus avantageux de réaliser l'arc-cohérence de la contrainte. Une stratégie possible pour la gestion des règles d'inclusion d'ensembles s'écrit donc ainsi :

Procédure `Contrôle`(r)
Données : `r` = règle de RCS sur la contrainte ensembliste C : [$X \in S$]
Début
  si *size*(*set*(C)) ≤ *size*(*variable*(C)) **alors** *BTFilteringAC*(C).
  sinon *apply*(`r`, C)

Fin

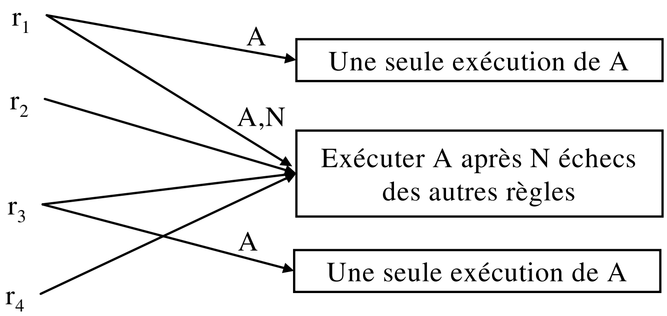
set(C) renvoie S ; *variable*(C) renvoie X. la procédure *BTFilteringAC* réalise l'arc-cohérence sur la contrainte C [55].

Exemple 4.2-6 : contrôle dynamique et relatif de plusieurs règles

Soient quatre règles r_1, r_2, r_3, r_4 telles que r_1 ne s'exécute que si r_2 a échoué au moins deux fois. La question est «comment spécifier le contrôle vérifiant cette condition en une seule stratégie simple ? » Un élément de réponse consiste à pouvoir déclarer plusieurs contrôles sur une même règle à l'aide de trois invariants de contrôle, comme suit :

`r1 onceRuleExecution. r3 onceRuleExecution.`
"impose à r_1 et r_3 de ne s'exécuter qu'une seule fois."

`r1 executionAfterNbEchecs: 2.`
"impose à r_1 de ne s'exécuter qu'après au moins deux échecs dans la base de règles. Une fois r_1 exécutée, on ignore les précédents échecs."



Le troisième invariant est implanté *via* un compteur d'échecs `NbRèglesEchouées`. L'invariant exprime le fait qu' au déclenchement de r1, r1 n'a pas encore été exécutée et 2 échecs ont eu lieu consécutivement (`nbRèglesEchouées >= 2`). Après le déclenchement, les invariants mettent à jour le statut des règles : r1 est bloquée (1er invariant), et `nbRèglesEchouées` est initialisé à 0.

Exemple 4.2-7: contrôle global pour déterminer un système linéaire

La composition linéaire par *Gauss-Simplexe* construit le système de contraintes de départ en fonction de deux questions, combien (*n*) de contraintes prendre et lesquelles ? Les invariants de contrôle offrent le moyen de guider cette construction en fonction de l'état global du solveur. Par exemple, 1) choisir un *n* très grand au départ puis vers la fin de la résolution prendre *n* plus petit car le problème est plus contraint ; 2) fixer *n* en fonction de la densité du problème[56] ; 3) pour *n* fixé, prendre *n* contraintes telles que $d_{intersection}(C_1,..., C_n)$ est maximum ou bien ; 4) pour *n* fixé, prendre *n* contraintes telles que pour tout $i \neq j$, $d_0(C_i,C_j)$ est maximum.

De plus la construction du système de contraintes se faisant par résolution d'un problème de CSP ayant pour domaine l'ensemble des contraintes du problème à résoudre, le contrôle consiste alors à choisir les contraintes de ce «méta-CSP » appropriées selon l'état de la résolution.

[55] En BackTalk c'est une classe conçue selon le pattern Strategy.
[56] Les densités sont définies au chapitre IV, section "Expérimentations".

4.2.1 Contrôle dynamique en méta-RCS

Le contrôle dynamique de Meta-RCS se décompose en trois étapes :

1) l'observation. Elle suit le *pattern* "*Event Notification*", implémenté à l'aide d'un mécanisme réactif de démons reliant chaque règle à un ensemble de stratégies. L'information observée est traduite par un événement, notifiée aux stratégies.

2) l'analyse. Elle est effectuée de manière incrémentale et transparente à chaque notification d'événement.

3) l'action. Elle a lieu au niveau des pré-conditions et post-actions (un test sur la règle candidate et une mise à jour du statut de la règle).

Les stratégies dynamiques sont construites selon un schéma prédéfini dans le *framework* de spécification : Action avant / après Evénement

- L'action consiste à bloquer ou activer une règle.
- Un type d'événement est défini par une règle et par un critère global ou un changement de statut. Par exemple si la règle R devient inutile, l'événement "R est *Inutile*" sera notifiée.
- Le moment de l'analyse peut être spécifié explicitement.

Avec ce schéma, deux types deux situations peuvent être reconnues : 1) l'exécution d'une règle sera probablement utile ; 2) l'exécution d'une règle a été certainement inutile.

Remarquons que le contrôle dynamique sert aussi à éviter le déclenchement des règles de RCS dans les situations jugées défavorables (C.f. plus loin la stratégie *ItStopIfUseless*). En effet, au chapitre II, nous avons montré l'importance d'identifier les situations favorables au raisonnement symbolique. Néanmoins il est tout aussi important de trouver les situations défavorables car RCS fonctionne de manière automatisée. En revanche avec cette approche on ne peut définir de contrôle sur les phases de raisonnement symbolique, par exemple on ne peut imposer qu'une règle s'applique pendant deux tours dans la boucle de traitement symbolique (rappelons que RCS s'applique itérativement tant qu'il y a des contraintes à traiter -C.f. chapitre III-1).

4.2.2 Réactivité du contrôle

Un invariant est un objet réactif qui réagit à un ensemble d'événements spécifiques en exécutant une méthode. Les événements sont définis au tableau 2 ; le tableau 4 donne la correspondance entre événement et action. Le lien entre un invariant et un solveur est réalisé par un mécanisme de démons : l'invariant s'enregistre au près d'une règle dont il doit contrôler l'application. Le résultat d'une méta-contrainte de contrôle est hétérogène : 1) Si l'événement est *PeutSeDéclencher*, elle renvoie le résultat d'un test ; 2) si l'événement concerne la fin d'exécution de la règle (*Appliqué* et ses dérivés), elle met à jour la relation et modifie l'état des règles.

Les invariants de contrôle de méta-RCS réagissent à des événements sur le changement de statut de RCS ou sur des critères globaux. Ce sont des objets globaux (réactifs) qui réalisent une stratégie de traitement symbolique. En ce sens, ils se rapprochent des agents de stratégies implantés dans BS-Solve [Granvilliers et al. 1998].

Le langage de spécification stratégies de Meta-RCS modélise une stratégie comme un ensemble de règle contextuelle dont la partie condition teste le type d'événement qui vient d'être notifié, et la partie action fait un effet de bord sur la ou les règles.

Changement de statut d'une règle	Méthodes de la classe d'invariant
PeutSeDéclencher	TestRuleBeforeTriggering(aRule)
Utile	UpdateRuleIsUseful(aRule, aConstraint)
Inutile	UpdateRuleIsUseless(aRule, aConstraint)
Succès	UpdateRuleSucceeds(aRule)
Echec	UpdateRuleFails(aRule)
Appliqué	UpdateAfterTriggering(aRule)

Tableau 14 : correspondance entre les événements engendrés par un changement de statut et les méthodes d'invariant.[57]

Une stratégie réagit par défaut à l'événement *Appliqué*, sauf lorsqu'un événement prioritaire survient auparavant, par exemple *Echec*. Les deux événements *Utile* et *Inutile* sont engendrés à la fin de l'exécution de la règle. Ils surviennent avant l'événement *Appliqué*. Le lecteur pourra trouver en annexe le détail des principales opérations du système qui sont utilisées dans le modèle de stratégies.

Reprenons l'exemple 4.2-1. Comment représenter la stratégie à l'aide d'invariants de contrôle ? Ce contrôle réagit aux événement *Apply* et *Applied*. L'invariant de contrôle implémente deux méthodes. La méthode `TestRuleBeforeTriggering(r,c)` vérifie que la taille de l'ensemble de `c` est inférieure ou égale à la taille du domaine de la variable de `c` ; dans ce cas, alors elle bloque `r` et renvoie FAUX, sinon elle débloque `r` si nécessaire et renvoie VRAI. La méthode `updateAfterTriggering(r,c)` ne fait rien car le contrôle ne nécessite pas de mise à jour d'informations.

4.2.3 Contrôles prédéfinis

Les stratégies primitives sont organisées hiérarchiquement selon le type d'action sur la manière dont va s'exécuter une règle. Des classes de stratégies générales ont été prédéfinies, particularisées, validées par les expérimentations décrites au chapitre IV. Par exemple, *ItComposition* paramètre la composition linéaire ; trois sortes de stratégies générales contrôlent l'exécution d'une règle : *ItOneExecution* n'autorise qu'une seule exécution, *ItBlockAfter* et *ItExecutionAfter* interdisent et autorisent l'exécution en fonction de l'exécution d'autres règles. *StopIfUseless* contrôle les règles de propagation générales en bloquant une règle lorsqu'au moins N de ses applications ont échoué.

[57] L'exécution des méthodes de mise à jour est encapsulée dans la méthode `update(R)`.

- **BlockAfterRulesApplication** (R, N, {R_1,....,R_k}, *EventType*)
  où *EventType* ∊ {Failure, Success, Useless, Useful, Executed}
  - BlockAfterFailure (R, N, {R})
  - **OneExecution** (R, P) où P vaut *Vrai* ou *Faux*
  - **StopIfUseless** (R,N)

- **ExecuteAfterRulesApplication** (R,N, {R_1,....,R_k}, *EventType*)
  - **ExecuteAfterFailure** (R,N,{R_i })

- **BlockIfGlobalCondition** (R,P) où P est un prédicat
  - BlockIfForbiddenConstraint (R,C)
  - **BlockIfLimitIsReached** (R, densité, d_o)

Une stratégie peut porter sur plusieurs règles et ainsi gérer la priorité des règles les unes relativement aux autres. Par exemple, la stratégie ExecuteAfterFailure(R,N,{Ri}) retarde le déclenchement de la règle R tant que moins Napplications de règles de {Ri} ont échoué.

Une seule application : ItOneExecution

En général les règles coûteuses comme la composition n-aire ou la composition de contraintes de cardinalité générique interviennent une seule fois, en pré-traitement avant le début de la résolution. Pour ce faire, *ItOneExecution* contraint sa règle à s'appliquer une seule fois et la bloque ensuite jusqu'à la fin de l'exécution. Cet invariant est prioritaire sur les autres invariants contrôlant le blocage de la règle ; une fois la règle appliquée, les autres invariants sont sans effet.

```
ItOneExecution(mainRule = r1) :
      updateAfterTriggering(r)
        Si(mainRule = r AND mainRule.state = Succès) alors block(r).
      testRuleBeforeTriggering(r)
         Si(mainRule = r) retourne not(r.state= Bloqué)
         Sinon retourne FAUX.
```

Figure 14 : Spécification de ItOneExecution, r étant une règle de R..S.

Changement de statut dans une situation donnée

Soit une situation que l'on sait caractériser, soit par un ensemble de conditions, qui constituent les pré-conditions de l'invariant, soit par l'apparition d'un événement sur une règle. La définition d'une stratégie devant s'appliquer dans cette situation consiste à définir un invariant de contrôle qui implante la méthode de décision correspondant à l'événement idoine. A chaque fois qu'une règle va ou vient de s'appliquer, la situation est évaluée par une fonction λ à deux arguments, la règle r et sa contrainte active c. Le critère de détection de la situation donnée compare $\lambda(r,c)$ avec une valeur limite ϕ. Parmi les instances de λ implantées dans la bibliothèque d'invariants, citons le nombre de contraintes, de choix faits par le moteur CSP ou le nombre d'exécutions de RCS jugées inutiles. Par exemple, si la situation concerne l'identification de la contrainte active avec une contrainte χ donnée, la fonction retourne simplement la contrainte active de r et le critère teste $\lambda(r,c) = \chi$. Ce critère est utilisé en particulier pour bloquer l'exécution de la règle *r* sur certaines contraintes sur lesquelles l'expert sait que *r* est inadaptée (C.f. section suivante). Cette stratégie suppose de mettre à jour l'invariant de contrôle lorsque la contrainte χ est supprimée car dans ce cas le contrôle n'est plus valide.

De nos expériences, nous avons identifié une situation de collaboration, dans le cas où le nombre de contraintes du problème est supérieur à un seuil. Le problème peut être considéré comme suffisamment dense pour que le maintien de cohérence produise de bons résultats. RCS n'est alors pas adapté car il contribuerait à complexifier le problème ce qui dégraderait le filtrage des contraintes. La stratégie *ItBlockAfter* exploite cette situation. Sa fonction λ renvoie simplement la taille de l'ensemble des contraintes du problème.

Ce type de contrôles possède deux caractéristiques communes : avant application, le critère est évalué ; après application, la règle change éventuellement d'état. Les invariants de contrôle réagissent donc au moins aux événements *PeutSeDéclencher* et *Appliqué* ; ils peuvent être affinés en tenant compte des autres événements, comme *ItStopIfUseless.*

Exécution après une situation : ItExecutionAfter

L'action de la stratégie (blocage ou autorisation) n'a lieu que si le critère est vérifié avant exécution de la règle principale (`mainRule`), *via* la méthode de test `testRuleBeforeTriggering(aRule, aConstraint)`. La mise à jour des données de l'invariant est nécessaire dans le cas de stratégie dynamique. Elle a lieu à la fin de l'exécution de `mainRule` et dépend de la sémantique. En particulier, deux contrôles héritent de *ItExecutionAfter*. L'un n'exécute `mainRule` que si `nbSeuil` règles ont été appliquées auparavant. L'autre n'exécute la règle principale que si le problème contient au moins `nbSeuil` contraintes. Dans le premier cas, l'invariant maintient un compteur des applications de la règle principale (`mainRule`); celui-ci est incrémenté dans la méthode `updateAfterTriggering(aRule,aConstraint)`. Dans le second cas, l'invariant ne contient pas d'information à maintenir car la taille de l'ensemble de contraintes est calculable au niveau du moteur CSP.

La stratégie *ExecuteAfterRulesApplication*(R,N,{R_i}, `TypeEvent`) retarde l'éxécution d'une règle R tant que moins de `N` exécutions de règles de {R_i} ont eu lieu avec pour type d'événement `TypeEvent`.

```
ItExecutionAfterRulesApplication(mainRule = r1, nbSeuil,
 rules = {r2,…,rk}, TEvent : booléen) :
 Variable locale : counter = 0
 TestRuleBeforeTriggering(r)
     Retourne (r.state ≠ Bloqué) ET
             ((r = mainRule ET counter >= nbSeuil)
               OU (r ≠ mainRule)).
 Update(r)
     Si (r.state = Appliqué ET r = mainRule) alors counter := 0.
     Si (r.state = TEvent  ET r ∈ rules)
                          alors counter := counter + 1.
     Si (r.state = Inutile ET r ∈ rules ET TEvent = Echec)
                                alors counter := counter + 1.
     Si (r.state = Utile ET r∈ rules ET TEvent = Succès)
```

```
                                alors counter := counter + 1.
```

Figure 15 : Spécification de ItExecutionAfterNbRulesExecution, r étant une règle de RCS

La stratégie ci-dessus a été particularisée pour contrôler précisément l'utilité ou l'inutilité d'une règle. L'invariant est paramétré par un attribut booléen, `ifEchec`. Si `ifEchec` est VRAI, seule les exécutions inutiles sont comptabilisées.

```
ItExecutionAfterRulesApplication(mainRule = r1, nbSeuil,
 rules = {r2,…,rk}, ifEchec : booléen) :
 Variable locale : counter = 0
 TestRuleBeforeTriggering(r)
    Retourne (r.state ≠ Bloqué) AND
           ((r = mainRule AND counter >= nbSeuil)
             OR (r ≠ mainRule)).
 Update(r)
    Si (r = mainRule ET r.state = Appliqué) alors counter := 0.
    Si (rules includes(r) ET r.state = Inutile)
             Si (ifEchec) alors counter := counter + 1.
    Si (rules includes(r) ET r.state = Utile)
             Si (not(ifEchec)) counter := counter + 1.
```

Blocage après une situation : ItBlockAfter

Les instances de ce contrôle bloque la règle principale lorsqu'une caractéristique du système CSP+RCS dépasse la valeur `nbSeuil`. Le critère, testé lors de l'événement *PeutSeDéclencher* est $\lambda(r,c) \leq$ `nbSeuil`. En particulier, deux contrôles héritent de *ItBlockAfter*. L'un bloque la règle après `nbseuil` choix faits par le moteur CSP, l'autre après `nbSeuil` applications inutiles parmi un ensemble donné de règles. Cette dernière est initialisée avec un ensemble de règles ; tout événement survenant sur l'une d'elles provoquera alors la réaction de l'invariant de contrôle sous-jacent. Le nombre d'applications inutiles est comptabilisé *via* un compteur ; celui-ci est incrémenté lorsque la règle est déclarée inutile. La spécification de cet invariant est donnée `ItExecutionAfterRulesApplication(mainRule = r1, nbSeuil,`

```
 rules = {r2,…,rk}, TEvent : booléen) :
 Variable locale : counter = 0
 TestRuleBeforeTriggering(r)
    Retourne (r.state ≠ Bloqué) ET
           ((r = mainRule ET counter >= nbSeuil)
             OU (r ≠ mainRule)).
 Update(r)
    Si (r.state = Appliqué ET r = mainRule) alors counter := 0.
    Si (r.state = TEvent  ET r ∈ rules)
                        alors counter := counter + 1.
    Si (r.state = Inutile ET r ∈ rules ET TEvent = Echec)
                                alors counter := counter + 1.
    Si (r.state = Utile ET r∈ rules ET TEvent = Succès)
                                alors counter := counter + 1.
```

Figure 15 par *ItExecutionAfterNbRulesExecution(r1, 0, nbseuil, rules, VRAI).*

La spécification ci-dessous décrit la stratégie consistant à bloquer une règle quand le problème contient au moins `nbSeuil` contraintes ou quand la contrainte active est une contrainte déclarée bloquante.

```
ItBlockIfCt(mainRule = r1, ctBloquante, counter = 0, nbSeuil)
 TestRuleBeforeTriggering(r,c)
       Retourne (r.state ≠ Bloqué) ET
     ((ctBloquante notNil ET not(c equivalentTo: ctBloquante))
       OU (size(constraintStore(CSPsolver(r))) < counter)
     ).
 updateAfterApplying(r,c)
     Si (ctBloquante notNil AND (c equivalentTo: ctBloquante))
     alors block(r)
     sinon si (counter notNil AND
               (size(constraintStore(CSPsolver(r))) ≥ counter)
             )
           alors block(r).
```

Figure 16: Spécification de ItBlockIfCt

`r` est une règle de RCS, dont `c` est la contrainte active. Chaque règle connaît le moteur CSP avec lequel elle collabore : la fonction `CSPsolver(r)` renvoie le moteur `CSP` idoine.

Contrôle d'une propagation : ItStopIfUseless

Cette stratégie vise à limiter la taille de l'ensemble des contraintes du problème. Elle est dynamique car elle prend en compte un historique des exécutions pour détecter une situation défavorable à RCS. Elle se base sur le constat suivant : lorsqu'il s'avère que plusieurs applications d'une règle semblent inutiles compte tenu du peu de conséquences qu'elle a sur les domaines lors du filtrage de la contrainte ajoutée, alors il vaut mieux abandonner cette composition et continuer la résolution. Par conséquent, l'invariant de contrôle *ItStopIfUseless* réagit aux événements *PeutSeDéclencher,Appliqué et Inutile.*

Nous l'avons validée sur les problèmes de carrés magiques où le nombre de composition de contraintes linéaires augmente très vite. Soit C la contrainte active d'une composition linéaire et S l'ensemble des contraintes linéaires combinables avec C. Il y a, *a priori,* plusieurs compositions avec C. La stratégie *ItStopIfUseless* autorise toute composition linéaire avec C et les contraintes de S, tant que moins de N compositions ont été inutiles. Autrement dit s'il existe N contraintes dans S telles que leur composition avec C est inutile, alors la règle *linlin* est bloquée pour C et le reste des contraintes n'est pas considéré.

```
ItStopIfUseless(mainRule = r1, counter = 0, n = 5):
UpdateRuleIsUseless(r)
    Si (mainRule = r) alors counter := counter + 1.
    Si (counter >= n) alors block(mainRule).

TestRuleBeforeTriggering(r)
       Retourne VRAI.
UpdateAfterTriggering(r) "rien"
```

Figure 17 : Spécification de ItStopIfUseless, r étant une règle de RCS

Lorsque l'application de la composition linéaire avec C est terminée le compteur d'échecs est remis à 0 pour le contrôle d'une prochaine application de la règle de composition linéaire.

Remarquons qu'avec ce contrôle, la résolution est susceptible de manquer des compositions intéressantes car la contrainte passive se situe après la $N^{ème}$ contrainte rendant la règle inutile. Un moyen pour éviter cela, est d'ordonner les contraintes passives, par exemple par nombre décroissant de variables en commun[58].

Collaboration entre compositions linéaires : ItComposition

Cette stratégie a été créée pour le contrôle des règles de propagation particulièrement « gloutonnes » afin de circonscrire le champ de leur application. Une règle de propagation *r* se déclenche sur une contrainte active C et un ensemble ordonné de contraintes passives C_1,...C_k. Par exemple, la composition d'une contrainte active C peut donner lieu à plusieurs déclenchements de *r* s'il y a plus d'un ratio valide.

ItComposition contrôle la manière d'engendrer les contraintes redondantes. Nous l'avons validée sur la règle de composition linéaire. Soit *S* l'ensemble des contraintes linéaires distinctes de la contrainte active *C*. La stratégie suit le principe suivant :

1. Si l'arité de *C* vaut 2, alors calculer toutes les compositions possibles de *C* avec *S* (*saturate*(*C*, *S*));
2. sinon si l'arité de *C* est supérieure à 2, ne calculer que les compositions qui produisent la contrainte la plus petite (*doBest*(*C*, *S*)) ;
3. sinon ne rien faire.

Cette stratégie a l'avantage d'être valable à la fois pour les compositions non binaires et binaires. *saturate(ct, SetOfConstraints)* exécute la procédure `compose(ct, c)`, pour toutes les contraintes `c` de *SetOfConstraints*, tandis que *doBest*(*ct, SetOfConstraints*) sélectionne les contraintes c_i dans *SetOfConstraints*, telle que `compose(ct, ci)` produise la plus petite contrainte (C.f. chapitre III-3). L'invariant *ItComposition* qui définit cette stratégie réagit à l'événement *PeutSeDéclencher*, comme le décrit la spécification ci-dessous.

```
ItComposition (mainRule = r1):
 UpdateAfterTriggering(r)    "rien"
 TestRuleBeforeTriggering(r)
     Si (arity(r.activeConstraint) = 2)
       Alors r.heuristicForPartnerCts := #saturate.
              r.ratiosOfTupleOption = #smallerRemaining.
             Retourne Vrai.
     Si (arity(r.activeConstraint) > 2)
       Alors r.heuristicForPartnerCts = #doBestRemoved.
     Retourne Vrai.
```

Figure 18 : Spécification de ItComposition, r étant une règle de RCS

---

[58] Dans le cas général, le fait de rater une composition n'a pas de mauvaise influence si les contraintes intéressantes ont été mémorisées comme devant être propagées par le raisonnement formel. En effet, si la composition n'est pas faite à ce moment, elle reste néanmoins valide tant que C existe dans le problème.

4.2.4 Mélange de stratégies

Dans ce travail, la bibliothèque de stratégies proposée permet le retrait et l'ajout de stratégie. Par exemple, l'action `retract(S1)` dé-enregistre l'invariant de contrôle S1, ce qui a pour effet de retirer la condition d'application relative à S1, puis l'action `add(StratBlockIfUseLess(r, 2)` connecte un invariant relatif uniquement à r dont la sémantique est « si le nombre d'échecs de r dépasse 2 alors bloquer r ». Toute nouvelle stratégie imposée sur une règle est en position de priorité minimale par défaut. Par ailleurs le langage de stratégie est construit dans le but de prendre en compte plusieurs situations favorables ou défavorables. Plus précisément, une stratégie complexe est définie soit par surcharge des invariants de contrôle déjà présents, soit en imposant plusieurs invariants sur une même règle. Dans ce dernier cas, cependant, des conflits peuvent survenir du fait que certains invariants ont des actions contradictoires. Pour lever cette ambiguïté, les invariants sont traités par ordre de priorité, l'ordre de base étant l'ordre chronologique d'imposition des invariants.

Contrairement à ELAN, un seul opérateur de combinaison est proposé, la séquence ; celui-ci évalue successivement les conditions des stratégies jusqu'à une stratégie de condition vraie ou une stratégie qui bloque la règle. La première stratégie évaluée qui bloque la règle est prise en compte.

Niveau de définition du contrôle

Le contrôle est défini au niveau de la classe de la règle à contrôler, dans la méthode publique `setExecutionControl`. Par exemple, dans la classe des règles de composition linéaire, on trouve :

```
LinearCompositionRule.setExecutionControl
     (ItComposition on: self).
     (ItStopIfUseless on: self nbFailure: 5).

LinearNaryCompositionRule. setExecutionControl
     (ItExecutionAfterRuleApplication
                on: self
                otherRules: solver rfRules nbSeuil: 5).
     (ItStopIfUseless on: self nbFailure: 5).
```

4.2.5 Dynamicité sur le contrôle

L'objectif du contrôle est d'affecter à chaque solveur une tâche, un espace de recherche et éventuellement un temps d'exécution. Dans cette architecture, nous considérons le contrôle des solveurs symboliques et plus généralement la collaboration CSP+RCS comme un problème dynamique d'optimisation, qui est géré par méta-RCS. En effet RCS modifie le problème à résoudre. Comme on ne connaît pas *a priori* la tendance de son évolution, le modèle décrivant la collaboration entre les solveurs symboliques est incomplet. Les changements de situations influencent l'utilité des solveurs et forcent à reconsidérer les stratégies précédemment établies. Par conséquent, pour supporter l'amélioration des décisions du contrôle, le système de contrôle doit répondre à un besoin de flexibilité et de facilité de construction de nouvelles stratégies. C'est le cas de l'architecture proposée. En effet notre approche, basée sur un mécanisme réactif d'invariants permet de modifier les stratégies et d'adapter les solveurs en réponse à des changements du problème. De ce point de vue, cette architecture répond aux mêmes exigences que les systèmes de configuration dynamique OZONE et OPIS [Smith 1995], [Smith et al. 1996][59].

5 Conclusion

La représentation uniforme des techniques symboliques est un atout certain pour la flexibilité de construction de stratégies. L'ouverture du système est d'autant plus importante que la détection de situations critiques relève essentiellement d'un processus expérimental. De plus la typologie hiérarchique de RCS offre un cadre pour la définition d'autres règles par héritage des actions et de la complexité des règles existantes.

[59] OZONE et OPIS sont deux *frameworks* de configuration de système d'ordonnancement et de contrôle, intégrés à la plate-forme DITOPS pour la génération, l'analyse et la révision de plans.

Chapitre IV : Expérimentations et implémentation

Les ordinateurs ne sont que des systèmes dotés d'une grande inconscience : tout y est tenu en mémoire de façon immédiatement accessible, et sujet à des programmes lancés par l'opérateur. L'opérateur, donc, est la conscience de l'ordinateur (Raja Lon Flatterie, Le Livre de la Nef).

Franck Herbert, Destination : vide, R. Laffont, chapitre 15, 1978

La première section de ce chapitre présente une série d'expérimentations dans le but d'identifier des critères, servant au niveau des stratégies à détecter les situations importantes. La caractérisation des situations favorables ou défavorables relève d'un processus d'apprentissage par analyse incrémentale, grâce à une représentation réfléxive de RCS : à partir d'une analyse *a posteriori* de l'intérêt de RCS, on déduit des critères qui sont utilisés pour prévoir l'utilité de chaque règle. Dans un premier temps, une étude a été menée sur des instances aléatoires de la catégorie des problèmes arithmétiques linéaires et quadratiques, ceci afin de s'abstraire des particularités des domaines d'application. Dans un deuxième temps nous avons étudié la classe des carrés magiques et plus particulièrement le contrôle de la composition linéaire binaire ; nous avions constaté au chapitre II la necéssité d'un tel contrôle pour que cette règle soit opérationnelle sans conséquence rédhibitoire. La troisième étude concerne la validation de la règle de propagation symbolique sur des domaines d'objets, dans le cadre de la preuve de théorèmes géométriques.

La seconde section de ce chapitre décrit les principaux éléments de l'implémentation de RCS et du module de contrôle de RCS, appelé *méta-RCS,* selon le cahier des charges spécifié au chapitre III.

En particulier le système dispose d'une interface (Figure 19) permettant de configurer avant résolution l'heuristique de choix d'instantiation ainsi que l'ensemble des stratégies qui est appliquées sur les règles de RCS.

Cette interface a entre autre permis d'adapter au mieux RCS pour un problème connu et difficile les carrés magiques (section IV-1.2).

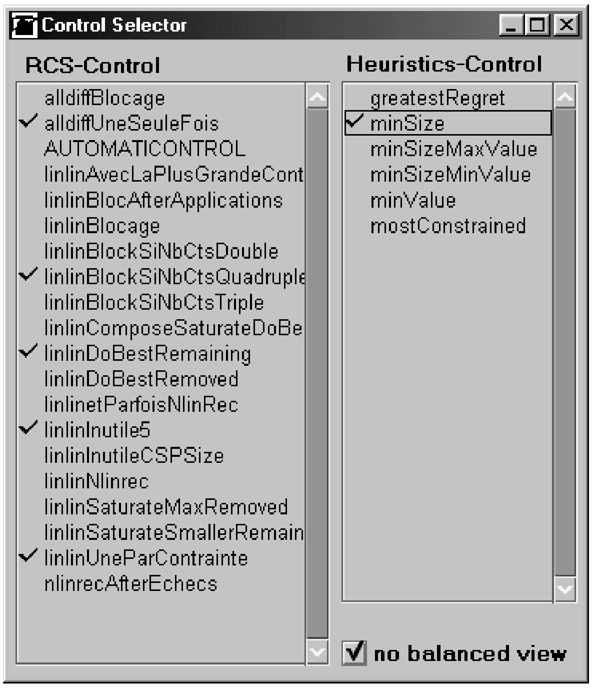


Figure 19 : Contrôleur de paramètre pour l'utilisation de RCS, les choix de BackTalk et le mode d'affichage de l'arbre de recherche du premier résultat obtenu.

1 Expérimentations

Ce chapitre analyse différentes expérimentations sur la combinaison entre les techniques de raisonnement symbolique et de satisfaction de contraintes. Nous avons utilisé le *framework* BackTalk comme moteur CSP. Comme nous l'avons déjà vu, RCS améliore la qualité de la résolution tout en ayant l'inconvénient d'être intrinsèquement lent. Ainsi, l'intégration de RCS au sein de techniques de CSP pose la question suivante : «le coût de l'introduction du raisonnement symbolique peut-il être compensé par le gain en qualité de résolution et sous quelles conditions ?».

Dans un premier temps nous avons étudié le comportement de la composition de contraintes arithmétiques pour une classe de problèmes sur des contraintes d'égalités et d'inégalités, linéaires ou quadratiques, en domaine fini. Les instances de problème ont été générées aléatoirement en faisant varier la densité, le nombre de variables et de contraintes et n'avaient pour la majorité pas de solution : dans 10% des cas, le moteur CSP a des difficultés à conclure et le gain de temps est positif ; dans 25% des cas, le raisonnement symbolique a permis de réduire le nombre de retour-arrières et augmente le temps d'un facteur inférieur à deux. Dans un deuxième temps, nous étudions le contrôle de ces composition de contraintes, par notre module de stratégies, *méta-RCS*. Le contrôle des techniques symboliques permet de réduire la complexité de la collaboration en sélectionnant le solveur symbolique le plus adapté en fonction de l'état du problème et de l'utilité des précédentes applications. En particulier, sur le problème difficile des carrés magiques (somme non connue), les stratégies réduisent le temps d'exécution tout en évitant les compositions inutiles. En l'absence de contrôle, la résolution du carré 3x3 requiert un retour-arrière et 11 fois plus de temps qu'avec BackTalk seul. En contrôlant les compositions linéaires, la somme est déterminée avant le premier choix quelle que soit la taille du carré, ramenant le problème au cas simple où la somme est connue. RCS a par ailleurs a été validé dans le cadre de problèmes sur des contraintes complexes.

1.1 Mesure de l'intérêt du raisonnement symbolique

L'étude dépend *a priori* du domaine d'application et des manipulations symboliques considérées. Nous avons don mené cette analyse sur une classe particulière de problèmes, les cryptogrammes et une classe de manipulation particulière, la composition linéaire de contraintes ; de plus les critères proposés indiquent si RCS est susceptible d'aider les algorithmes de cohérence à améliorer la réduction de l'espace de cherche, *i.e.* d'apporter une amélioration quantitative directe (C.f. section 1.1.1). Notre recherche a suivi le plan suivant : à partir d'un ensemble de fonctions d'évaluation d'un système de contraintes, nous comparons les résolutions avec et sans RCS (nombres de choix et de retour-arrières, temps) ; nous avons constaté expérimentalement l'existence de valeurs seuil et en avons déduit un ensemble de critères. Afin d'avoir un résultat général, nous avons mené une étude sur des séries de problèmes aléatoires appartenant tous à la classe des problèmes numériques à équations polynomiales du second degré[60]. Nous avons constaté que pour les problèmes

[60] Les contraintes polynomiales de degré supérieur à 1 sont automatiquement décomposées en une

contenant les contraintes difficiles pour le moteur CSP, RCS réduit le nombre de retour-arrières et parfois le temps de résolution. Cette expérience confirme le fait qu'il existe des cas où le coût de la manipulation symbolique de contraintes est compensé voire annulé par un gain en temps ; ce gain étant dû à la diminution du nombre de mauvais choix.

Notre deuxième étude s'est focalisée sur les carrés magiques. Le chapitre II indique que le nombre de contraintes utiles est faible en comparaison des contraintes générées : la composition ajoute énormément de contraintes avant que le filtrage détermine une variable et provoque la suppression de contraintes triviales. De manière générale, RCS complexifie le problème ce qui augmente la complexité du filtrage de l'ensemble des contraintes et les possibilités de compositions ultérieures. Le problème doit atteindre une densité seuil pour que les contraintes redondantes influencent positivement la résolution.

Nous avons cherché des explications à ce comportement :

1. Le mécanisme de composition linéaire, trop général doit être spécialisé.
2. La composition de k contraintes ($k \geq 2$) augmente la dépendance des contraintes entre elles, mais elle est basée sur les contraintes ayant déjà une intersection ; elle ne permet donc pas de relier des contraintes totalement indépendantes.
3. Le problème n'est pas assez dense, au sens où les contraintes ont peu de variables en commun. RCS fonctionne très bien sur SEND+MORE=MONEY, où il existe toujours deux contraintes ayant au moins deux variables en commun ; il fonctionne moins bien sur DONALD+GERALD=ROBERT, où aucun couple de contraintes n'a plus d'une variable en commun

Conjecture prouvée expérimentalement :

> Il existe une relation entre le nombre de contraintes à composer et la densité du problème ; moins le problème est dense, plus les contraintes à composer ensemble doivent être nombreuses.

1.1.1 Mesure des effets de RCS

S'il est aisé de trouver un intérêt à la simplification de contraintes et à la déduction formelle d'incohérence (C.f. exemple ci-dessous), il n'en va pas de même pour l'intérêt de l'ajout de contraintes redondantes, car celui-ci complexifie le problème. Dans ce cas l'utilisation de RCS relève d'un compromis car le filtrage des nouvelles contraintes augmente le coût du maintien de cohérence, d'autre part la résolution est plus directe car l'ensemble des solutions potentielles cohérentes est plus petit.

Exemple : schéma de détection formelle d'absence de solution

> Soit (V,D,K) un CSP de n contraintes.
> $\exists\, k \in [1,n], \{c_1,\ldots,c_k\} \subseteq K \Rightarrow$ FAUX

Nous avons trois manières de mesurer l'intérêt du raisonnement symbolique. Au niveau global, la qualité de la résolution est évaluée en termes de nombre de retour-arrières et de points de choix. Au chapitre II et dans la première étude de cette section, nous mesurons ainsi la différence de retour-arrières entre deux résolutions. Au niveau local, l'exécution de RCS est évaluée en fonction de la réduction de domaine qui est effectuée par le filtrage du nouveau problème. Mais l'intérêt de RCS n'apparaît pas toujours lors du filtrage suivant la

---

conjonction de contraintes d'intervalles.

réécriture du problème ; par exemple dans les problèmes de carrés magiques, les premières introductions de contraintes redondantes n'ont pas d'effet sur le filtrage ; néanmoins elles engendrent des compositions ultérieures qui provoqueront des réductions de domaines. Finalement nous proposons deux mesures de la conséquence de RCS, le délai entre son exécution et son exploitation pour une réduction effective de l'espace de recherche, et le type d'exploitation. Le premier critère évalue la conséquence sur le maintien de cohérence , tandis que le second évalue l'effet sur RCS lui-même :

1. *Conséquence immédiate, locale et quantitative*
   RCS restreint effectivement l'espace de recherche car le filtrage de la nouvelle contrainte ajoutée provoque une réduction de domaine. Cette réduction est en supplément par rapport à une résolution sans RCS.
2. *Conséquence retardée, globale et qualitative*
   RCS engendre des contraintes qui permettent, ultérieurement dans la résolution, d'exécuter une manipulation symbolique ayant une conséquence immédiate.

1.1.2 Critères de prévision

Soit un problème (Ω, Δ, K)[61]. On note n_Ω le nombre de variables, n_K le nombre de contraintes.

$$d_0 = \frac{n_K}{\frac{n_\Omega * (n_\Omega + 1)}{2}} \qquad d'_0 = \frac{\sum_{c_i \in K, n_i = arity(c_i)} \frac{n_i * (n_i + 1)}{2}}{\frac{n_\Omega * (n_\Omega + 1)}{2}}$$

$$d_1 = \frac{1}{n_\Omega} \sum_{ci \in K} arity(ci) \qquad d_2 = \frac{\sum_{(c1,c2) \in K} |commonVariables(c1, c2)|}{n_\Omega^{\ 2}}$$

$$pM = \max_{v_i \in \Omega}(poids(v_i)) \text{, où } poids(v_i) = |\{c \in K, v_i \in \text{variables}(c)\}|$$

commonVariables(c1,c2) renvoie l'ensemble des variables communes à *c1* et *c2*. *arity(c)* renvoie le nombre de variables de la contrainte *c*.

La densité d_0 évalue la quantité de contraintes par rapport au nombre total de contraintes binaires possibles. Lorsque les contraintes de K ne sont pas toutes binaires, cette fonction est ambiguë et remplacée par d'$_0$. En effet en première approximation, une contrainte non binaire peut être traduite en une union de contraintes binaires [Tsang 1993] et une contrainte binaire, restreinte à un couple de variables non ordonnées. Ainsi, pour une contrainte *i*, d'arité n_i, il y a au plus $\frac{n_i * (n_i + 1)}{2}$ contraintes binaires.

Par ailleurs nous avons défini des fonctions afin d'évaluer l'interdépendance entre les contraintes d'un ensemble, celle-ci étant modifiée par l'exécution de RCS. La densité d_1 indique la charge maximale de chaque variable en calculant le rapport entre la charge de toutes les variables et le nombre de variables. La densité d_2 définit une fonction qui va nous permettre de mesurer l'intérêt des compositions linéaires binaires (*linlin*), à partir du fait que plus les contraintes ont de variables en commun plus la contrainte générée est susceptible

[61] Aucune condition n'est posée sur la nature et l'arité des contraintes.

d'être petite. La fonction d_2 calcule le rapport entre les intersections des couples de contraintes du problème et le nombre de total de couples de variables, *i.e.* n_Ω^2.

De plus, le poids des variables peut nous guider dans le choix du nombre de contraintes à combiner. Nous définissons le poids d'une variable *v* comme le nombre de contraintes portant sur *v*. Soit pM_Ω le poids maximal des variables de Ω et pM_{12}, le poids maximal des variables non communes à deux contraintes c_1 et c_2. Plus pM_{12} est élevé plus forte est probabilité que la composition de c_1 et c_2 produise une contrainte de grande arité.

Remarquons que la densité du problème change en cours de résolution ; les fonctions sont donc évaluées dans la boucle de la procédure de résolution hybride (C.f. chapitre I). A la suite des expériences décrites en section 1.1.3 et 1.1.4, à partir de la courbe d'évolution de la résolution en fonction des valeurs de densité en moyenne, nous avons déduit la définition de critères pour prévoir si RCS sera intéressant : $d_0 \geq d_{0Seuil}$; $d_1 \geq d_{1Seuil}$; $d_2 \geq d_{2Seuil}$; $pM_\Omega \leq pM_{\Omega Seuil}$, où d_{0Seuil}, d_{1Seuil}, d_{2Seuil}, $pM_{\Omega Seuil}$ sont déterminés expérimentalement.

Conjecture prouvée expérimentalement :

> Dans un état donné du problème, l'intérêt de la composition de contraintes est mesuré en fonction du nombre de critères qui sont vérifiés et de l'écart entre la densité du problème courant et la valeur seuil correspondante.

Densité d'intersection

Contrairement aux précédentes définitions, la densité d'intersection prend en compte les compositions de plus de 2 contraintes. Pour tout sous-ensemble de contraintes susceptibles de se composer, elle évalue le rapport entre le nombre de variables de la contrainte générée et le nombre total de variables. La densité d'intersection[62] de c_1 et c_2 est définie par le rapport entre l'intersection entre C_1 et C_2 et l'intersection maximale possible ($C_1=C_2$).

$d_{\text{inter}2}(A_1, A_2) = \dfrac{A_1 \cap A_2}{A_1 \cup A_2}$, où A_i = *Variables*(c_i).

Généralisons à *k* contraintes. On a besoin d'étendre cette définition à Δ_k, de même que nous avons étendu la définition ensembliste de la composition à l'aide de cet opérateur.

$d_{\text{inter}k}(A_1,..., A_k) = \dfrac{\left|\Delta_{k-1}(A_{\alpha_1},...A_{\alpha_{k-1}}) \cap A_{\alpha_k}\right|}{k}$, où le dénominateur est l'intersection maximale entre les *k* contraintes et α_i varie dans [1, *k*].

Propriétés de la densité d'intersection

Au chapitre III nous avons prouvé que la fonction F n'est pas symétrique. Par conséquent la densité d'intersection, basée sur la valeur de F, est différente pour deux compositions permutables, c'est à dire équivalentes dans le cas idéal. On a en effet : $d_{\text{inter}k}(A_1,.., A_k) = \dfrac{F(A_1,.., A_k)}{k}$. Cette non-symétrie assure que la densité d'intersection distingue toutes les compositions indépendamment de leur résultat.

[62] notations identiques à celles du chapitre III.

Remarquons que la densité d'intersection d'un système obtenu par l'algorithme de Gauss-Jordan est toujours nulle. En effet, nous avons vu au chapitre III qu'un tel système possède au moins une variable par contrainte, qui n'est commune à aucune autre contrainte du système[63]. Par conséquent la différence symétrique n-aire est strictement égale à l'ensemble de toutes les variables pivots ; aucune de ces variables n'apparaît plus d'une fois dans le système. Par conséquent F = 0.

1.1.3 Comportement de *linlin* sur des équations linéaires

Pour cette expérience et la suivante, nous nous sommes inspirés du générateur de [Freuder & Hubbe 1995] qui propose de générer des CSP binaires de plus en plus denses et propose comme paramètres le nombre de variables, la tailles des domaines, la connectivité du graphe de contraintes (*i.e.* la proportion de contraintes existantes) et la proportion de paires de valeurs interdites entre deux variables d'une contrainte. Nous avons construit un générateur de CSP où seuls les nombres de contraintes et de variables varient.

Pour limiter l'influence de la topologie du problème, les problèmes considérés ont tous la même forme, qui est celle du problème ALPHA. Les problèmes sont définis par :

V est un ensemble de variables entières,
$D = [0, size(V)]\text{x}\ldots\text{x}[0, size(V)]$,
Toutes les v_i dans *V* ont des valeurs différentes deux à deux,
Toute contrainte c appartient à *K* si et seulement si c est définie par $\sum_{i=1}^{n_c}(v_i) = \delta_c$, pour n_c variant dans [5,40], où δ_c est une constante calculée aléatoirement dans $[5, size(V)]$.

Le tableau 1 en annexe regroupe les résultats des valeurs des densités avec 10 séries de problèmes obtenus en faisant varier le nombre de contraintes de 2 à 45, pour une heuristique fixe (*minSize&minValue*). Nous observons également le nombre de retour-arrière et le temps d'exécution. L'intérêt de RCS est représenté par le surcoût en temps *cpu* et la réduction du nombre de retour-arrières.

La réduction de retour-arrière ainsi que le surcoût de temps entre les deux résolutions sont calculés en pourcentage. En abscisse, la série de problèmes aléatoires regroupés par nombre de contraintes croissant et densités d_0, d_1, d_2 croissantes et poids maximal (pM) décroissant. Tous les problèmes au-delà de la série 11, vérifient les critères sur d_1, d_2, d_0 et pM. Il y a donc un seuil à N=11. Nous remarquons aussi que plus les valeurs sont éloignées du seuil, plus le pourcentage de réduction du nombre de retour-arrières est élevé. Si celui-ci est égal à 100%, cela signifie qu'aucun retour-arrière n'a eu lieu.

[63] Ces variables sont les variables utilisées comme pivot.

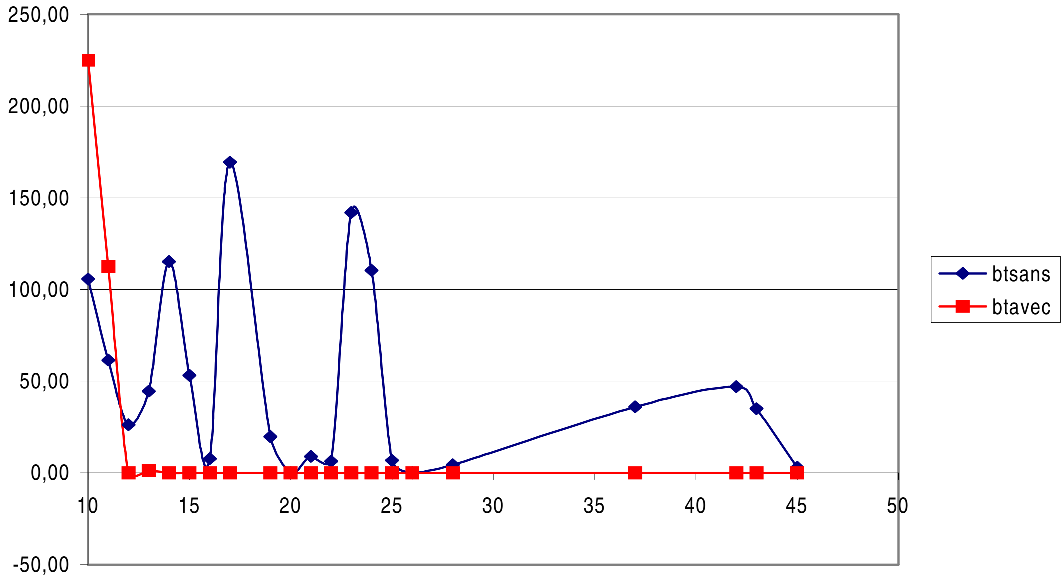
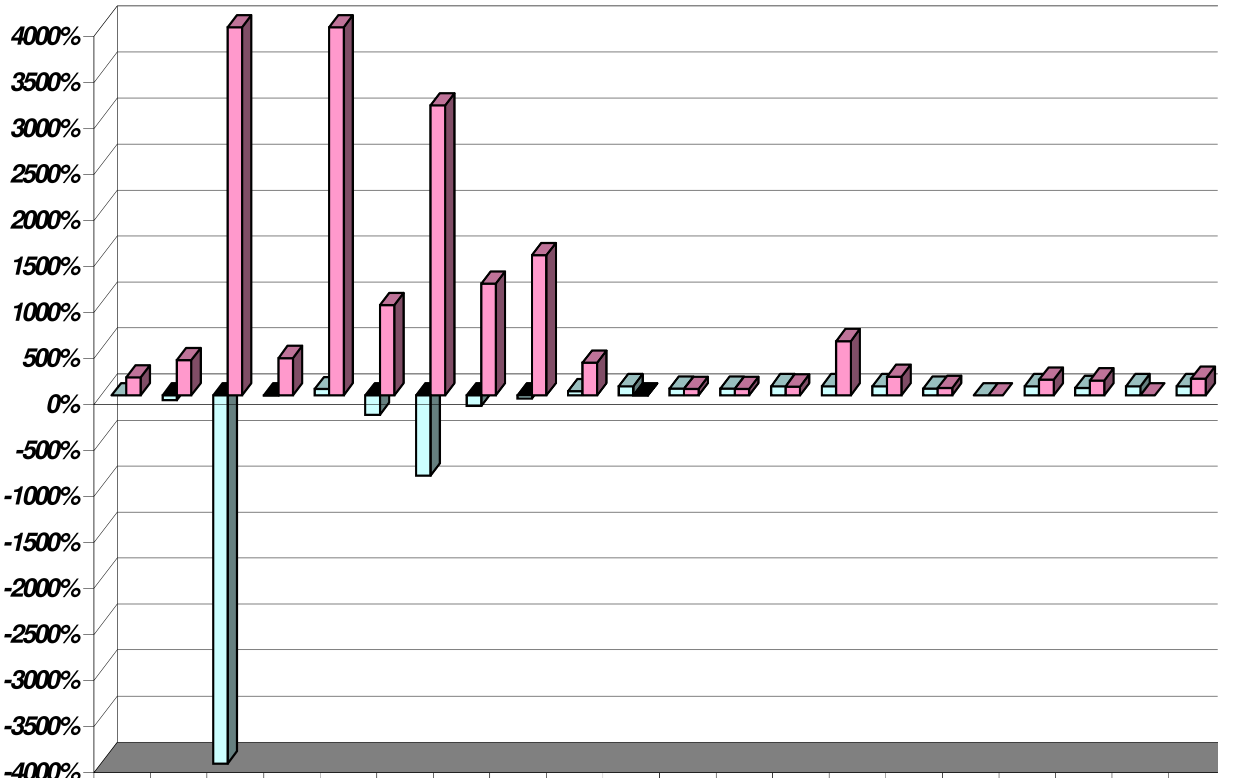


Figure 20 : réduction de retour-arrière avec et sans saisonnement symbolique.

La Figure 20 et la Figure 21 confirment l'existence d'un seuil à partir duquel RCS est intéressant globalement. La plupart des problèmes étant générés aléatoirement, ils n'ont pas de solution. Ces critères mesurent donc surtout l'intérêt de la composition pour la détection de l'absence de solution. En moyenne les temps d'exécution sont similaires à ceux d'une résolution standard : le coût de RCS est compensé par les déductions qu'entraîne l'ajout de contraintes redondantes[64].

[64] Les déductions sont soit des réductions de domaines soit des exceptions signifiant qu'une incohérence est détectée.



	2	3	4	5	6	8	9	10	11	12	13	14	15	17	19	20	21	22	23	24
pourcentred	0	-0,54	-109	-0,02	0,684	-8,75	-1,14	-0,35	0,451	1	0,742	0,75	1	1	0,75	0	1	0,8	1	1
pourcentcpu	1,951	3,812	384,1	4,053	57,08	31,54	12,14	15,23	3,519	-0,05	0,69	0,695	0,946	2,004	0,801	0	1,696	1,625	0	1,797

Figure 21 : évolution de la réduction du nombre de retour-arrière et du surcoût en temps CPU en fonction du nombre de contrainte

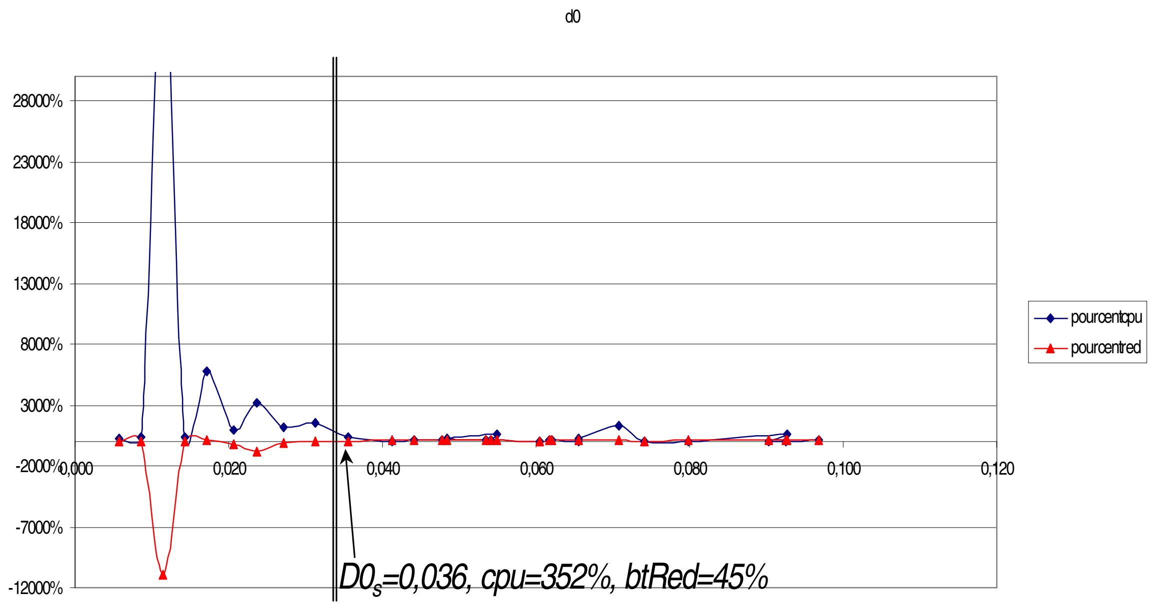


Figure 22 : surcoût en temps cpu et réduction du nombre de retour-arrière en fonction de d_0

Dans cette expérience, dans 56,7 % des cas, le gain en retour-arrières est non nul et le surcoût inférieur à 500% (*i.e.* moins de 10 fois plus long) ; dans 10,3% des problèmes le gain en temps d'exécution est non négligeable.

Valeurs seuil en moyenne :	Critères expérimentaux :
D_{0S}= 0,036 ; d_{1S}= 4 ; d_{2S}= 0,68 ; pM_S= 0,6	$d_0 \geq d_{0S}$; $d_1 \geq d_{1S}$; $d_2 \geq d_{2S}$; $pM \leq pM_S$

Pour des problèmes petits et peu denses ($d_2 < d_{2s}$; $d0 < d_{0s}$), l'intérêt de RCS dépend énormément de la topologie de l'instance. Pour des problèmes ayant une densité élevée ($pM < pM_s$, $d_2 \geq d_{2s}$, $d_1 \geq d_{1s}$, $d_0 \geq d_{0s}$), RCS produit un résultat uniforme quelle que soit l'instance du problème :

> Il existe un réel A (A entre 100% et 500%) et un intervalle I, tel que pour n appartenant à I, *pourcentcpu*(n) ≤ A,
> Et pour tout n appartenant à I, tel que *pourcentcpu*(n) > 0, il existe un réel θ grand tel que *pourcentred*(n) > θ.

De plus nous avons évalué la distance Q des valeurs des fonctions à leur valeur seuil : plus Q est grand, plus la composition est susceptible de réduire un domaine directement après l'ajout de la contrainte redondante. Cela confirme la conjecture énoncée à la section 1.1.2 dans le cas d'une mesure de la qualité des compositions binaires de contraintes pour la détection d'incohérence.

La phase d'observation de la résolution a abouti à des critères basés sur l'état du problème courant. La seconde phase consiste à programmer ces critères et à les utiliser effectivement. Néanmoins le calcul de ces critères globaux est coûteux. Des critères locaux à chaque classe de contraintes, basés sur sa sémantique ou sa structure seraient peut être plus rapides à évaluer car ils ne dépendraient que des caractéristiques d'une contrainte, au lieu de l'ensemble des contraintes du problème.

1.1.4 Résultats des critères sur quelques cryptogrammes standards

RCS a été appliqué avec un contrôle spécifié dans méta-RCS. Dans le problème DONALD+GERALD=ROBERT, le contrôle préserve les choix d'énumération. Nous interprétons ce phénomène ainsi : des compositions inutiles sont évitées et des contraintes qui ne servent pas pour améliorer la résolution[65] ne sont pas ajoutées. Dans tous les autres cas, le contrôle influence RCS qui lui-même influence l'ordre des choix.

Problème	Résolution CSP		Résolution CSP+RCS +méta-RCS		%Red-Ret-Arr[66]	%Sur-cpu[67]*
	Temps (seconde)	Retour-arrières	Temps (seconde)	Retour-arrières		
Send+more=money	0,022	1	0,009	0	100	-144,4 (2,5 fois)
Donald+gerald=robert	0,045	11	0,198	7	36,4	77,2 (4,4 fois)
alpha 1ère solution	0,112	10	4,11	1	90	97,27 (36,6 fois)
alpha unicité de la solution	0,5912	27	0,127	10	62,9	-365,5 (4.6 fois)
Magic 3x3 (Pentium II)	0,201	50	0,323	1	97,1	37,7 (1.19 fois)
Magic 4x4 (Pentium II)	0,938	849	0,213	0	100	-340,3 (1.1 fois)
Magic 5x5 (Pentium II)	Interrompu	-	227,787	676	interrompu	-
xa*x=bxc 1ère solution	0,028	5	0,028	3	40	0
xa*x=bxc 2ère solution	0,032	6	0,032	4	33,3	0
xa*x=bxc 3ère solution	0,04	9	0,038	5	44,5	-5,26
xa*x=bxc les 8 autres solutions	0,076	22	0,066	11	50	-15,15
Ain+aisne+drome+marne=somme les trois solutions	0,628	196	2,4	182	-7,14	73,8 (3,8 fois)

Tableau 1 : tableau de comparaison pour les cryptogrammes.

Le tableau 1 ci-dessus montre les temps de résolution en millisecondes. Quand le nombre de retour-arrières est nul, le solveur a résolu le problème sans mauvais choix. Le tableau 2 en annexe présente les valeurs des fonctions de densité sur le problème initial. Le seul critère toujours vérifié est $d_0 \geq d_{0s}$. Les critères sur d_1 et d_2 ne sont vérifiés que pour les lignes 2, 3, 4. Or dans tous les problèmes RCS est efficace. Ce décalage par rapport aux résultats généraux sur les problèmes aléatoires s'explique par le fait que les problèmes (sauf les lignes 2, 3, 4) sont très petits (moins de 10 contraintes initiales). La densité n'y est pas aussi déterminante.

D'après nos expériences, les critères étudiés permettent de distinguer les cryptogrammes adaptés à des transformations de contraintes numériques. Celles qui sont utilisées ici agissent principalement sur des contraintes linéaires par composition linéaire ou parce qu'il y a une contrainte de différence. En revanche ces critères ne fournissent aucune condition sur la complexité des contraintes lorsque les solveurs symboliques opèrent sur des contraintes de

[65] Elles sont inutiles car le filtrage qui est réalisé sur elles ne réduit pas davantage les domaines que le filtrage des autres contraintes du problème.

[66] %Red-Ret-Arr = nombre de retour-arrière en moins entre CSP et CSPR+RCS , ramené par rapport à CSP.

[67] %Sur-cpu = différence de temps d'exécution entre CSP+RCS et CSP, ramené par rapport à CSP+RCS.

types différents. D'autres critères prenant en compte la structure de la contrainte elle-même pourraient être envisagés.

1.1.5 Comportement de *linlin* sur des équations quadratiques

En procédant de la même manière que dans l'expérience de la section 1.1.3, nous avons vérifié expérimentalement que RCS se comportait de manière similaire et que les taux de réussite étaient maintenus. De plus les critères restent valables moyennant une adaptation des valeurs seuils. Considérons la classe des problèmes polynomiaux suivante :

> *V* est un ensemble de variables entières,
> *D* = [0, size(*V*)]x….x[0, size(*V*)],
> toute contrainte c appartient à *K* si et seulement si c est définie par $\sum_{i=1}^{n_c}(\alpha_i \cdot v_i^{\beta_i}) = \delta_c$, pour n_c variant dans [5, 40], où δ_c est une constante calculée aléatoirement dans [5, size(*V*)²], les coefficients α_i et β_i sont calculés aléatoirement respectivement dans [0, 10] et {1,2}.

K est un sous-ensemble des équations polynomiales de degré au plus deux, contenant par exemple, le triplet de Pythagore, les équations diophantiennes et les cryptogrammes. Ce type de problème est assez complexe car les algorithmes dédiés uniquement aux équations quadratiques ne peuvent s'appliquer que sur une partie des contraintes. De plus, sur ce type de contraintes, un moteur CSP réalisant la borne-cohérence ne fonctionne pas bien : les résultats Figure 22 confirment cette tendance dans 60,4% des problèmes. La décomposition des contraintes non-linéaires permet d'appliquer la composition linéaire sur ce type de problème (C.f. annexe).

Nous avons réalisé 7 séries de 30 problèmes et avons sélectionné 123 tests[68]. Parmi ceux ci, seulement deux donnent un résultat négatif pour RCS (augmentation du nombre de retour-arrière et du temps d'exécution !); dans ces cas, le moteur CSP a du mal à prouver l'absence de solution et RCS aggrave la situation. Par ailleurs pour certains problèmes ayant au moins une solution, nous constatons l'utilité de RCS, mais les temps d'exécution sont bien plus élevés que ceux du moteur CSP ; cela peut être dû au fait que ces problèmes ont un petit nombre de contraintes.

Pour 46% des problèmes, RCS produit des résultats avec moins de retour-arrières. De plus dans 10% des cas, le temps mis par le RCS est inférieur ou égal à celui mis par le moteur CSP. Cette première analyse amène à la conclusion que plus les contraintes complexes sont décomposées, moins le filtrage par borne-cohérence est efficace et plus forte est la probabilité que le surcoût de CSP+RCS par rapport à CSP soit proche de zéro voire négatif.

La Figure 23 présente l'évolution, d'une part de la différence du nombre de retour-arrières et, d'autre part de la différence de temps d'exécution entre les résolutions CSP et CSP+RCS. Les valeurs sont calculées en pourcentage pour amoindrir l'effet des résultats extrêmes. La première quantité est rapportée par rapport à la résolution par CSP+RCS ; nous l'appelons *pourcentred* car l'on cherche à réduire le nombre de retour-arrières. La seconde quantité est rapportée par rapport au temps mis par le moteur CSP seul (BackTalk) ; nous

---

[68] Beaucoup de problèmes étaient trivialement incohérents aussi bien pour les CSP que pour CSP+RCS et quand la résolution dépassait 24 heures, nous avons interrompu le solveur.

l'appelons *pourcentcpu* car RCS introduit un coût et l'on veut prévoir s'il est compensé par le coût des mauvais choix.

Equations des courbes :

pourcentcpu (n)=100* ((*tempscpuRS*(n) - *tempscpuCSP*(n)) * (*tempsCSP*(n))$^{-1}$).

pourcentred(n)=100* ((*retour-arrièreCSP*(n) - *retour-arrièreRS*(n))* (*retour-arrièreRS*(n))$^{-1}$).

> Courbe de tendance de pourcentcpu : *Polynomiale_pourcentred*(x) =
>
> 4.10-11 .x^6 – 6.10-8 .x^5 + 6.10-6 .x^4 - 0,0002 .x^3 + 0,0004 .x2 + 0,0209 .x + 0,2165.
>
> Courbe de tendance de pourcentred : *Polynomiale_pourcentcpu*(x) =
>
> 10-7 .x^6 – 2.10-5 .x^5 + 0,0016 .x^4 - 0,0651.x^3 + 1,388.x^2 - 14,128.x + 53,588.

L'axe des abscisses représente le nombre de contraintes des séries de problèmes. Pour chaque point, les valeurs des courbes sont les moyennes des valeurs obtenues pour chacune des séries. Les deux courbes de tendance polynomiale sont calculées par Excel, avec les polynômes de degré 6 énoncés ci-dessus. Quand la courbe de tendance de *pourcentcpu* est la plus basse, le coût des manipulations symboliques est à son minimum. Si la valeur des abscisses des points de la courbe est négative, cela dénote un gain de temps. Quand la courbe de tendance de *pourcentred* est la plus haute, le nombre de mauvais choix évités avec RCS est à son maximum. Les plages où les valeurs de la courbe sont positives correspondent à un gain en qualité de résolution.

Les courbes ont été divisées en 4 parties selon ce qu'elles induisent sur le comportement de RCS dans la résolution. Les quatre flèches notent les 4 cas en moyenne pour lesquels RCS apporte un gain en temps non nul :

Cas 0 : Cas limite des petits problèmes.

> Les problèmes initiaux ont peu de contraintes (faible densité). Certains problèmes ont au moins une solution, et dans ce cas le surcoût en temps *cpu* est faible. Pour d'autres problèmes le petit nombre de contraintes limite la quantité de compositions possibles en début de résolution.

Cas 1 : Pour tout n tel que 14 ≤ n ≤ 38, pourcentcpu(n) ≤ 500%.

> Le surcoût est borné, relativement élevé, jusqu'à 500%, c'est à dire une résolution avec CSP+RCS qui est 7,5 fois supérieure à celle avec CSP. Néanmoins la moitié des problèmes sont résolus avec moins de 100% d'augmentation, ce qui est acceptable. Dans cette fourchette, 58% des cas sont résolus plus directement avec CSP+RCS et pour 13 % des cas, sans retour-arrière. La gamme des densités couvre de 14 à 38 contraintes initiales.

Cas 3 : Pour tout n tel que 39 ≤ n ≤ 42, 200% ≤ pourcentcpu(n) ≤ 851%.

> Cette tranche regroupe les mauvais résultats : le surcoût est rédhibitoire et le gain en nombre de retour-arrières est négatif ou nul. Cette tranche couvre très peu de cas (abscisses 39 à 42).

Cas 4 : Pour tout n tel que 43 ≤ n ≤ 91, pourcentcpu(n) ≤ 150%.

> Les problèmes initiaux sont très denses et n'ont pas de solution. RCS n'améliore pas la résolution mais les temps d'exécution restent corrects (inférieurs à 100% d'augmentation pour 94% des points). RCS change les choix qui sont faits mais pas en

les améliorant, c'est à dire qu'une résolution par le moteur CSP nécessite autant de mauvais choix, mais ce ne sont pas les mêmes. Ce résultat n'est pas surprenant car nous avions vu au chapitre II que, si RCS est appliqué sans contrôle particulier, il n'est pas certain qu'il soit toujours intéressant.

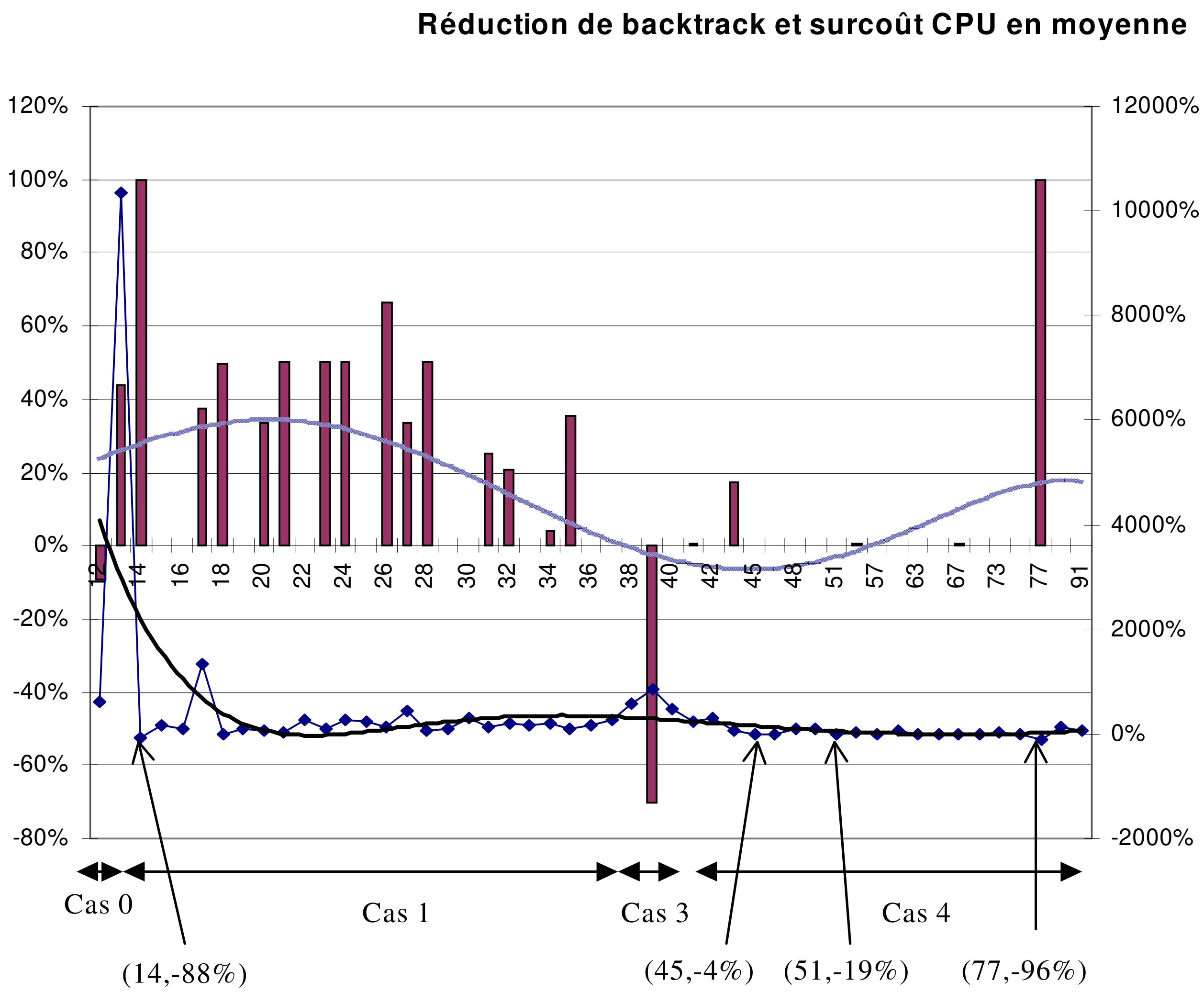


Figure 23 : influence de RCS sur des problèmes quadratiques aléatoires

Analyse des courbes

Signalons que, dans cette expérience, RCS n'utilise pas de stratégie particulière si ce n'est d'éviter d'appliquer deux fois la même règle sur les mêmes contraintes. En particulier, la composition linéaire de contraintes est appliquée systématiquement sur tous les couples de contraintes valides *i.e.* ayant une intersection et génère les contraintes plus petites que l'une au moins des contraintes sources.

Globalement, pour tous les problèmes aléatoires sauf trois cas, on a la relation suivante : (*retour-arrièreCSP*(*n*) – *retour-arrièreRS*(*n*)) ≥ 0.

Dans le *Cas 1*, la différence de temps d'exécution est négative ou nulle. De plus la résolution de CSP+RCS fait moins de choix mais prend plus de temps pour traiter l'ensemble des contraintes entre chaque choix. Cette partie de l'expérience confirme la conjecture énoncée à la suite des expériences du chapitre II et que nous proposions de valider dans cette thèse, à savoir : il existe des situations identifiables où le coût introduit par la manipulation symbolique des contraintes avant chaque choix est compensé par le gain en temps dû au faible nombre de mauvais choix.

Soit *reductionBT* la fonction définie par *reductionBT* = (*retour-arrièreCSP* - *retour-arrièreRS*).

- Si *reductionBT* > 0, 25% des cas en moyenne sont tels que *pourcentcpu*(n) ≤ 100% et 42% sont tels que *pourcentcpu*(n) ≤ 500%.
- Si *reductionBT* = 0, RCS n'a pas amélioré la qualité de la résolution et néanmoins le surcoût reste faible. 54 % des cas en moyenne sont tels que *pourcentcpu*(*n*) ≤ 100%, dont 25% avec une augmentation de moins de 50% du temps d 'exécution. Nous considérons une augmentation de 100% du temps de résolution comme tout à fait acceptable[69]. On prend comme convention que l'augmentation est aussi acceptable quand CSP+RCS a mis moins d'une seconde pour trouver le résultat et que le moteur CSP a résolu le problème par un simple filtrage (le temps est alors de 0 seconde).
- Au total, pour tous les cas où *reductionBT* ≥ 0, 10 % sont tels que *tempscpuRS*(*n*) ≤ *temspcpuCSP*(*n*) (gain en temps et qualité). Nous pouvons donc affirmer que pour la classe des problèmes de polynômes quadratiques entiers, la composition binaire de contraintes est utile à la fois qualitativement et quantitativement, dans 10 % des cas.

[69] Cela signifie que le système CSP+RCS a mis deux fois plus de temps que le moteur CSP.

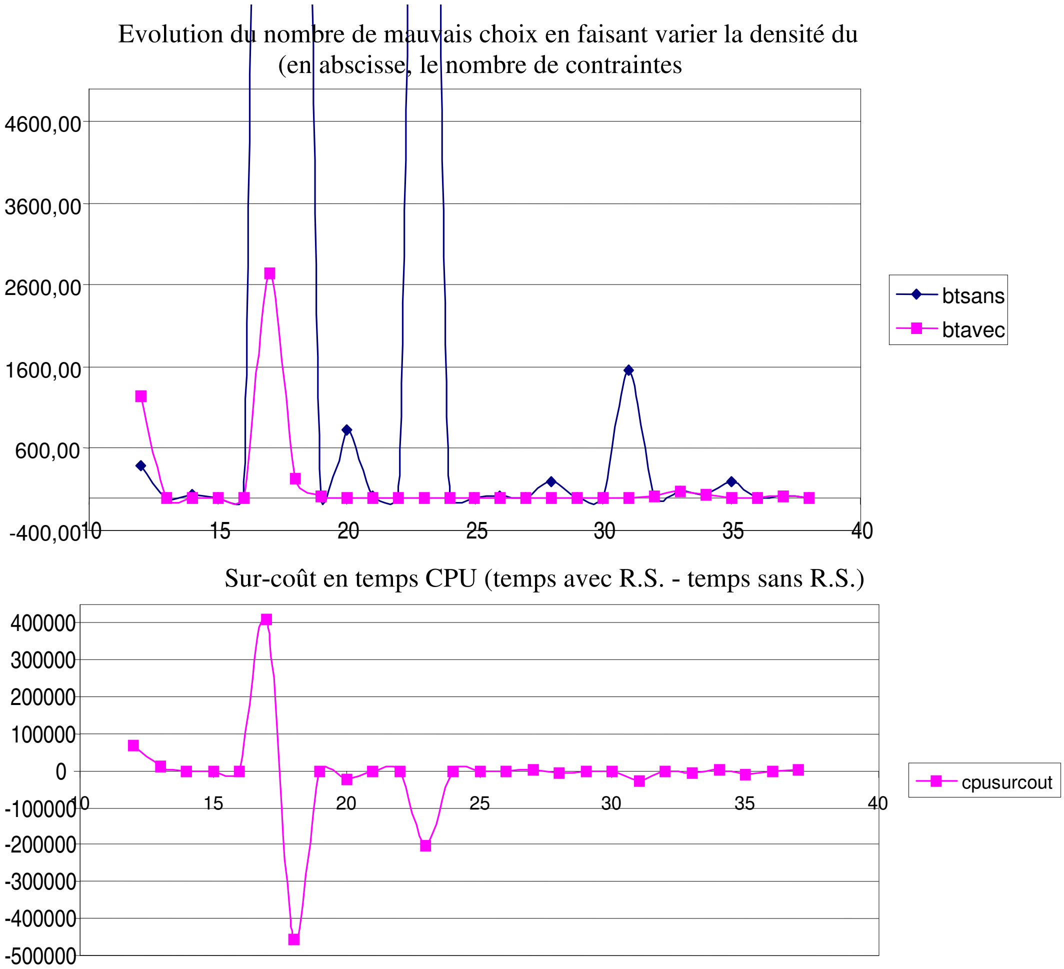


Figure 24 : comparaison du nombre de mauvais choix et du temps d'exécution pour CSP et CSP+RCS

Evaluations des critères

Les critères trouvés sur des problèmes linéaires sont encore valables dans ces problèmes quadratiques ; plus précisément ils sont tous vérifiés dans les cas où RCS est le plus utile, moyennant une adaptation des valeurs seuils des fonctions d'évaluation.

Valeurs seuil en moyenne :	Critères :
$d0_S$= 0,027 ; $d1_S$= 3.25 ; $d2_S$= 0,68 ; pM_S= 0,6 ; $d2_{Max}$ = 1,01 ; pM_{Min} = 0,37	$d0 \geq d0_S$; $d1 \geq d1_S$; $d2_{Max} \geq d2 \geq d2_S$; $pM_{Min} \leq pM \leq pM_S$

De cette expérience nous retirons deux conclusions : 1) Il existe des cas où effectivement RCS améliore la résolution en temps et en qualité mais la caractérisation de ces cas est délicate et doit faire l’objet d’étude plus approfondie pour des contraintes plus complexes. 2) L’utilisation de RCS n’est pas toujours positive et cela est en partie dû au fait que les techniques utilisées ne permettent pas de déduire les contraintes redondantes pertinentes ; par exemple pour remplacer la composition binaire, que se passerait-il si l'on composait trois contraintes en imposant que deux contraintes engagées dans une composition ternaire ne peuvent plus se composer deux à deux ?

1.2 Les carrés magiques : utilisation de stratégies

Le chapitre II a conclu sur l'importance du contrôle pour combiner efficacement composition linéaire binaire et cohérence sur les problèmes où l'interdépendance des contraintes est faible. Or dans le problème des carrés magiques, les contraintes ayant toutes la même arité et la même structure, elles sont systématiquement toutes soumises aux transformations symboliques. La distinction de ces contraintes nécessite une stratégie. Nous avons constaté qu'un compromis entre les deux stratégies de compositions *saturate* et *doBest* améliore le comportement des manipulations symboliques : la stratégie consiste à appliquer *doBest* sur les contraintes complexes et *saturate* sur les contraintes binaires[70]. Par ailleurs même avec cette stratégie, l'application pendant toute la résolution des compositions linéaires augmente beaucoup la taille du problème sans que cela provoque immédiatement une réduction de domaines. Dans cette section nous menons une étude comparative pour N=3,4,5,6 et validons une stratégie qui, d'une part limite l'accroissement de la taille du problème et, d'autre part évite de composer une contrainte quand cela s'avère inutile depuis un certain nombre de compositions.

Cette section compare deux stratégies de résolution. La première approche consiste à se ramener à un problème plus simple, c'est à dire où σ est correctement instantiée et où les valeurs des variables les plus contraintes sont correctes. Pour cela les heuristiques ne suffisent pas. Par exemple, pour N=3, l'heuristique *mostConstrained* aggrave la résolution car les valeurs choisies ne sont pas des solutions, malgré le fait qu'elle instantie d'abord Bt5. Par ailleurs, l'introduction de la contrainte redondante *Cbij* permet d'augmenter la densité d'intersection (de $^2/_9$ à $^1/_2$) et d'initialiser le raisonnement symbolique. La valeur de σ peut alors être trouvée par une suite de compositions linéaires ; par exemple, il faut 24 compositions pour N=6, impliquant des contraintes non données initialement tandis que 7 compositions suffisent pour N=3. Mais la combinatoire des couples de contraintes fait que la résolution met 1000 fois plus de temps. La seconde approche consiste à imposer une borne supérieure au nombre de compositions linéaires inutiles dans un cycle d'exécution de RCS ; la complexité est nettement réduite. Si cette borne est atteinte, la composition est arrêtée, d'où une accélération dans la suite de la résolution. Néanmoins des contraintes pertinentes peuvent être oubliées, d'où l'augmentation du nombre de retour-arrières par rapport à la première approche.

Remarquons que les contraintes des carrés magiques ont toutes le même poids et les variables jouent toutes le même rôle, sauf celle du centre pour N impair. Cela se voit concrètement dans l'arbre de recherche pour différentes heuristiques : les arbres ont tous la même forme au nom de variable près (C.f. Figure 25). Par conséquent, aucune composition n'est meilleure qu'une autre.

Figure 25 : forme grossière des arbres choix pour deux heuristiques

---

[70] Rappelons que la stratégie *saturate* est coûteuse mais complète, tandis que la stratégie *doBest* est rapide mais peut court-circuiter des déductions pertinentes.

1.2.1 Le carré magique 3x3 résolu par Alice

Alice exprime les variables les moins contraintes en fonction des variables les plus contraintes et supprime les variables les moins contraintes ; l'ensemble des contraintes est réécrit en un ensemble où les éléments ont davantage d'intersection ce qui favorise les compositions de contraintes. La stratégie d'Alice (page 83 de [Laurière 1978]) revient à orienter les compositions telles que les contraintes générées conservent les variables les plus contraintes.

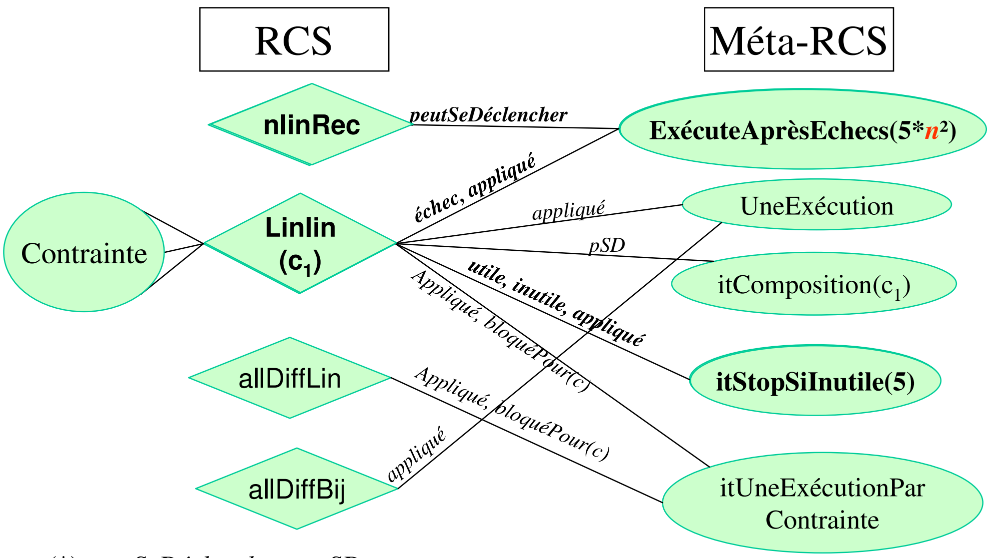
Dans CM 3x3, avec la stratégie d'Alice, dans chaque composition linéaire Bt5 est conservée ; la contrainte $3*Bt5=\sigma$ est déduite après trois compositions. Par ailleurs la valeur de σ est trouvée par le raisonnement formel décrit dans [Pitrat 1993]. Finalement Alice trouve la valeur de Bt5 directement et la première solution est trouvée en 0 retour-arrière et 2 choix par Alice. Néanmoins la suite de compositions par J. Pitrat requiert des heuristiques astucieuses ; un traitement automatique ne peut pas suivre la même démarche. La résolution décrite au chapitre II trouve la valeur de Bt5 en composant systématiquement les contraintes linéaires et aboutit à une solution différente en 1 retour-arrière et 3 choix car les choix sont différents. Cependant le coût en temps et taille mémoire reste élevé. Par ailleurs les techniques de CSP peinent dès que N vaut 5 ; ILOG-Solver entre dans une combinatoire exponentielle pour N = 5 si σ n'est pas connue, de même que BackTalk.

1.2.2 Utilisation de stratégies

Nous présentons une approche basée sur les stratégies `ItComposition` et `ItStopIfUseless`, où ces invariants de méta-RCS sont reliés à la règle *linlin*. `ItComposition` décide de la manière dont il faut engendrer les contraintes composées, en fonction de l'arité de la contrainte active. Cette stratégie a l'avantage d'être valable quel que soit le nombre de contraintes composées simultanément (2 ou plus). L'objectif de cette stratégie n'est pas d'instantier à tout prix la variable la plus contrainte ni de trouver S le plus tôt possible, mais de conserver l'efficacité des compositions linéaires tout en limitant la combinatoire. Cela est loin d'être évident car un grand nombre de compositions ne sont pas utiles immédiatement mais ont un effet retardé (C.f. section 1.1.1). La stratégie `ItStopIfUseless(N)` est utilisée conjointement avec `ItComposition`. Cette stratégie limite la taille du problème et donc la complexité de la résolution mais, puisque les contraintes sont considérées dans l'ordre d'apparition, il se peut qu'une composition intéressante soit oubliée car elle se trouvait après la N+1$^{\text{ème}}$ contrainte dont la composition était inutile. Dans le cas des carrés magiques, le seuil N a été fixé à 5 expérimentalement.

Le mécanisme de collaboration est externe aux règles ce qui permet de contrôler leur déclenchement. En particulier, pour comparer les résolutions, il suffit de changer le contrôle des règles. Pour interdire l'application d'une règle R, il suffit de relier la règle à l'invariant de contrôle `ItBlockRule(0)`(C.f. chapitre III-4).

Les figures ci-dessous montre la configuration optimale des stratégies prédéfinies, pour la résolution des carrés magiques par BackTalk, en instantiant la variables de plus petit domaine en priorité (minSize) et avec l'intégration de RCS comme décrit au chapitre III.



(*) *peutSeDéclencher = pSD*

1.2.3 CM3x3

Dans le cas N=3, deux stratégies sont comparées, l'une n'utilise que *saturate* pour composer les contraintes linéaires, l'autre combine s*aturate* et d*oBest* et bloque les compositions au bout de 5 échecs. Dans le premier cas, la résolution est très longue mais ne fait qu'un seul retour-arrière. Cela est dû au fait que σ=15 est trouvé au bout de 10 compositions, avant le premier choix. Une fois σ instantiée, la valeur de BT5 est déduite 60 compositions plus tard, avant le premier choix.

Dans le second cas, la résolution est plus rapide. Cela est dû au fait que σ=15 est trouvé au bout de 3 compositions, avant le premier choix. Le contrôle court-circuite les compositions qui déduisent la valeur de BT5, ce qui privilégie le moteur de cohérence locale.

Solveur	Qualité et temps CPU (sur différentes machines)		Nb contraintes
	σ connue	σ inconnue	
Ilog-Solver 4.3.1	2bt, 4choix 0s (Sparc 5)	48bt 51choix 0.01s (Sparc 5)	9
BackTalk MinSize&MinValue	2bt, 4choix 0.061s (Pentium I166)	50bt 53choix 201ms (Pentium I166)	10 (1 contrainte redondante)
BackTalk+RCS – CMComposition MinSize&MinValue	1bt 3choix 3.522s (Pentium I166) (104 contraintes avant le premier choix)	1bt, 3choix 11fois plus long que BackTalk (Pentium I166)	De 10 à 50, puis de 50 à 14
		1bt, 3choix 323ms (Pentium I166)[71]	De 10 à 47, puis de 47 à 10.
Alice (Laurière 1976)	0bt, 2choix 400ms (IBM 370/168)	0bt, 2choix non connu	9 au départ
GNU Prolog (CLP(FD)) (Pentium I90)	0ms 3bt	20 ms 65 bt	9
	0ms 0bt (tableau optimisé par Diaz)	10 ms 0 bt (tableau optimisé par Diaz)	

Tableau 15 : une solution de Magic 3x3

Les résultats pour Alice sont pris dans [Pitrat 1993], ceux de Ilog-Solver, GNU Prolog et BackTalk ont été implémentés. Les résultats sur BackTalk sont obtenus avec l'heuristique *minSize*. Les résultats sur BackTalk+RCS sont obtenus d'une part en saturant toutes les compositions (1 retour-arrière). D'autre part avec les stratégies `ItComposition` et `ItStopIfUseless(5)` et en bloquant les compositions quand la taille du problème dépasse 100 contraintes, on obtient le même nombre de retour-arrières que si S est connue. Avec BackTalk, le premier choix n'est pas correct, au contraire de BackTalk+RCS.

Par ailleurs dans la seconde résolution, la présence de Bt5 = 5 est plus importante que la contrainte *CBij* : la présence de Bt5 = 5 suffit pour trouver la première solution en 1 retour-arrière et 3 choix. La deuxième résolution avec BackTalk+RCS est même plus rapide sans la contrainte *CBij* (division du temps par deux) ; le nombre de retour-arrières est inchangé. On peut donc conclure que *CBij* n'est pas utile quand on utilise la stratégie `ItComposition`. Le fait de laisser une contrainte inutile C peut avoir une conséquence négative car RCS va engendrer d'autres contraintes également inutiles - sinon la contrainte inutile C ne le serait pas !

Evaluation des critères

[71] résultat sans application de la règle déduisant Cbij.

Cas de la première stratégie

Le raisonnement symbolique est intéressant dans deux cas :

1) A la composition `(Bt3 + Bt6 + Bt9 - σ = 0 et -Bt3 - Bt6 - Bt9 - 2.σ = -45 ➔ σ = 15)`, les critères valent `d0 = 0,327327 d1 = 8,5 d2 = 7,87 pM = 0.5556`.

Tous les critères de prévision sont vérifiés.

2) A la composition `(Bt1 + Bt2 + Bt3 + Bt4 + Bt6 = 25 et Bt1 + Bt2 + Bt3 + Bt4 + Bt5 + Bt6 = 30 ➔ Bt5 = 5)`, un filtrage supplémentaire a lieu. Les critères valent `d0 = 0,85454 d1 = 19,2 d2 = 43,56 pM = 0,617021`.

Les critères sur d0, d1, d2 sont vérifiés. Le résultat de la section 1 est confirmé.

Cas de la seconde stratégie

Grâce à la stratégie *ItStopIfUseless*(5), l'ensemble des contraintes ne contient jamais plus de 47 contraintes pendant l'exécution. Voici la trace obtenue au début de la résolution :

```
  σ [(3..27)] => σ int[(9..21)]
  2e composition, LINdoBestRemoved
Bt2 +Bt5 +Bt8 -σ = 0 et -Bt2 -Bt3 -Bt5 -Bt6 -Bt8 -Bt9 -σ = -45
       ➔-Bt3 - Bt6 - Bt9 - 2.σ = -45

  nbVariables = 10  nbContraintes = 12  d1 = 6.1  d2 = 3.99  d3 = 399.0  pM = 0.833333.
σ[(9..21)] => σ int[15]
  5e composition, LINdoBestRemoved
   Bt3 +Bt6 +Bt9 - σ = 0 et -Bt3 -Bt6 -Bt9 - 2.σ = -45 ➔ σ = 15

  NbVariables = 10  nbContraintes = 13  d1 = 5.5  d2 = 3.31  d3 = 331.0  pM = 0.538462
```

1.2.4 CM4x4

BackTalk+RCS met seulement trois fois plus de temps que BackTalk. Grâce à *ItStopIfIUseless*(5), le problème ne dépasse pas 34 contraintes. La somme σ = 34 est trouvée au bout de 10 compositions. Voici la trace obtenue au début de la résolution :

```
  3e composition LINdoBestRemoved

  Magic Square 4 : nv= 17  nc= 15  d1= 6.41176  d2= 2.6609  d3= 769.0  pM= 0.866667

  Bt11 + Bt15 + Bt3 + Bt7 - σ =0 et -Bt11 - Bt12 - Bt15 - Bt16 - Bt3 -Bt4 - Bt7 - Bt8 - 2.σ= -136 ➔ -Bt12 - Bt16 - Bt4 - Bt8 - 3.σ =-136

  σ [(4..64)] => σ int[(24..44)]

  4e composition LINdoBestRemoved

  Magic Square 4 : nv= 17  nc= 15  d1= 5.47059  d2= 1.88581  d3= 545.0  pM= 0.466667

  Bt12 + Bt16 + Bt4 + Bt8 -σ =0 et -Bt12 - Bt16 - Bt4 - Bt8 - 3.σ = -136 ➔ σ = 34

  σ [(24..44)] => σ int[34]

  '0.213s ; 0 bt ; 7 ch'
```

La suite de la résolution utilise la procédure d'énumeration et filtrage standard, sur le problème contenant des contraintes redondantes dont l'effet est qualitatif. En effet le filtrage de ces contraintes réduit davantage l'espace de recherche puisque la première solution de ce nouveau problème est trouvée par BackTalk <u>sans retour-arrière</u> !

<table>
<tr><th>Solveur</th><th>σ connue</th><th>σ inconnue</th><th>Nb contraintes</th></tr>
<tr><td>Ilog-Solver 4.3.1</td><td>0.01s 15bt, 19choix (Sparc5)</td><td>0.2s 849bt , 853 choix</td><td>11</td></tr>
<tr><td>BackTalk</td><td>0.082s 15bt, 19choix
(pentium I166)</td><td>2,818 s 849 bt,
(pentium I166)</td><td>12</td></tr>
<tr><td rowspan="2">BackTalk+RCS - CmComposition</td><td rowspan="2">7.317s 27bt, 31choix
(pentium I166)</td><td>11.795s
15 bt, 19 choix</td><td rowspan="2">12 à 33</td></tr>
<tr><td>0,633s
0 bt, 7 choix</td></tr>
<tr><td rowspan="2">GNU Prolog (CLP(FD)) (pentium I90)</td><td>30ms 41bt</td><td>2.090s 4419 bt</td><td>11</td></tr>
<tr><td>120ms 0bt (tableau optimisé de Diaz)</td><td>1.940s 0bt (tableau optimisé de Diaz)</td><td></td></tr>
</table>

Tableau 16 : une solution de Magic 4x4

Solutions trouvées :

1	2	15	16
12	14	3	5
13	7	10	4
8	11	6	9

1	8	12	13
7	14	2	11
10	3	15	6
16	9	5	4

1.2.5 CM5x5

Le système applique toutes les compositions linéaires possibles, dans la limite des stratégies, jusqu'à obtenir un point fixe. Chaque nouvelle contrainte est composée à son tour. Quand un choix est fait, les contraintes sont simplifiées puis la propagation recommence. La somme inconnue est trouvée à la 21ème composition linéaire. Notons que cette composition est due à des contraintes issues des compositions initiales. Voici la trace obtenue jusqu'à la détermination de σ.

```
   σ [(5..125)] => σ int[(25..105)]
   5e composition LINdoBestRemoved
   Bt13 + Bt18 + Bt23 + Bt3 + Bt8 - σ = 0 et -Bt10 - Bt13 - Bt14 -
Bt15 - Bt18 - Bt19 - Bt20 - Bt23 - Bt24 - Bt25 - Bt3 - Bt4 - Bt5 -
Bt8 - Bt9 - 2.σ = -325 ➔ -Bt10 - Bt14 - Bt15 - Bt19 - Bt20 - Bt24 -
Bt25 - Bt4 - Bt5 - Bt9 - 3.σ = -325
NbVariables = 26  nbContraintes = 19  d1 = 7.96154  d2 = 2.67012  d3
= 1805.0  pM = 0.894737
σ [(25..105)] => σ  int[(50..80)]
   10e composition LINdoBestRemoved
   Bt14 + Bt19 + Bt24 + Bt4 + Bt9 - σ = 0 et -Bt10 - Bt14 - Bt15 -
Bt19 - Bt20 - Bt24 - Bt25 - Bt4 - Bt5 - Bt9 - 3.σ = -325 ➔  -Bt10 -
Bt15 - Bt20 - Bt25 - Bt5 - 4.σ = -325
NbVariables = 26  nbContraintes = 24  d1 = 10.6538  d2 = 4.80325  d3
= 3247.0  pM = 0.916667
σ [(50..80)] => σ int[65]
   21e composition LINdoBestRemoved
   Bt10 + Bt15 + Bt20 + Bt25 + Bt5 - σ = 0 et -Bt10 - Bt15 - Bt20 -
Bt25 - Bt5 - 4.σ = -325
   ➔ σ = 65
```

`nbVariables = 26  nbContraintes = 32  d1 = 13.6923  d2 = 7.66864  d3 = 5184.0  pM = 0.59375`

Solveur	σ connue	σ inconnue	Nb contraintes
BackTalk	3.745s 791bt, 799choix (pentium I166)	Interruption	14
BackTalk+RCS-Cmcomposition	501.563s 2008bt, 2021 choix (pentium I166)	227,787s 676 bt, 684 choix	14 à 34
Ilog-Solver 4.3.1	0.18s 791bt, 799choix (Sparc5)	Interruption	13
GNU Prolog (CLP(FD))	5,240s 9404 bt	Interruption	13
	250ms 55bt (tableau optimisé de Diaz)	255.250s 48 bt (tableau optimisé de Diaz)	

Tableau 17 : une solution de Magic 5x5 [72]

Un effet dramatique de RCS consiste à ajouter des contraintes inutiles en grand nombre, changeant ainsi l'ordre des choix. Le nouvel ordre n'est pas *a priori* meilleur que l'ordre des choix de la résolution par le moteur CSP. Dans le cas N=5 et σ inconnue, pour BackTalk+RCS, il n'y a pas plus de 5 contraintes non immédiatement utiles pour chaque traitement d'une contrainte, donc en tout pour un cycle il n'y a pas plus de 5*(*Nombre de contraintes à propager*) contraintes inutiles, soit au pire 5*(*Nombre de contraintes initiales*) = 5*(2*N+2). Ces contraintes servent dans la résolution après le premier choix, puisque le problème produit par RCS est résolu plus directement que lorsque σ est connue dès le départ.

Solution trouvée :

1	2	13	24	25
3	23	17	6	16
20	21	11	8	5
22	4	14	18	7

1.2.6 Détail de la résolution de CM3x3

Nous approfondissons ici l'étude. Les valeurs des critères sont notés à chaque fois qu'une composition provoque une réduction de domaine. Cela arrive quand les domaines sont réduits par le filtrage qui suit systématiquement toute composition. Au moment où Bt5 est trouvée, les fonctions sont les suivantes : `d1=23.7778  d2=70.0247  d3=5672.0  pM=0.58`. Les critères sont tous vérifiés et de plus leur valeur est très loin des seuils. Après le choix Bt1 = 2, le problème retrouve sa taille initiale et les domaines sont réduits :

```
Bt 2 [(1..9)] => Bt 2 int[(6..9)]
Bt 3 [(1..9)] => Bt 3 int[(4..7)]
Bt 4 [(1..9)] => Bt 4 int[(6..9)]
Bt 5 [(1..9)] => Bt 5 int[(4..7)]
Bt 6 [(1..9)] => Bt 6 int[1 (3..5)]
Bt 7 [(1..9)] => Bt 7 int[(4..7)]
Bt 8 [(1..9)] => Bt 8 int[1 (3..5)]
Bt 9 [(1..9)] => Bt 9 int[(6..9)]
```

Ensuite la contrainte `Bt6 + Bt8 = 4` est ajoutée par composition `Bt2 + Bt3 = 13 (-) Bt2 + Bt3 = 13,` ce qui réduit Bt6 et Bt8 à {1,3} ; le problème a alors 19

[72] Pour S inconnue, les tests sur Ilog-Solver et BackTalk ont été interrompus.

contraintes. Remarquons qu'Alice déduit aussi la contrainte `Bt6 + Bt8 = 4` et trouve le même domaine. A ce moment, les fonctions sont les suivantes : `d1=9.44444 d2=11.0247 d3=893.0 pM=0.631579.` Les critères sont tous vérifiés mais leur valeur sont moins loin des seuils.

Il ne reste plus qu'un choix trivial à faire, Bt8 = 1. Quatre contraintes sont supprimées et le filtrage instantie Bt6 par 3 à cause de Bt6 + Bt8 = 4. Une seule composition est alors efficace : `Bt2 + Bt3 + Bt7 + Bt9 = 27 (-) Bt3 + Bt5 + Bt7 = 15` ➔ `-Bt2 + Bt5 - Bt9 = -12.`

La présence de la contrainte redondante `-Bt2 + Bt5 - Bt9 = -12` permet au filtrage de réduire les domaines :

```
Bt 2 [(6..9)] => Bt 2 int[(7..9)]
Bt 3 [(4..7)] => Bt 3 int[(4..6)]
Bt 5 [(4..7)] => Bt 5 int[(4..6)]
Bt 9 [(6..9)] => Bt 9 int[(7..9)]
```

Les fonctions valent à ce moment : `d1=6.44444 d2=6.2716 d3=508.0 pM=0.66667.` Les critères sont vérifiés mais leur valeur est très proche des seuils.

Une seule composition est efficace : `Bt2 + Bt3 + Bt7 + Bt9 = 27 (-) Bt3 + Bt4 + Bt7 + Bt9 = 25` ➔ `-Bt2 + Bt4 = -2,` les nouveaux domaines sont :

```
Bt 2 [(7..9)] => Bt 2 int[9]
Bt 3 [(4..6)] => Bt 3 int[4]
Bt 4 [(6..8)] => Bt 4 int[7]
Bt 5 [(4..6)] => Bt 5 int[5]
Bt 7 [(5..7)] => Bt 7 int[6]
Bt 9 [(7..9)] => Bt 9 int[8]
```

Les fonctions valent : `d1=3.66667 d2=1.98765 d3=161.0 pM=0.5.` Seul le critère sur d2 est vérifié.

Conclusion

Nous induisons qu'il y a un lien entre l'efficacité de la composition linéaire et la distance des fonctions d'évaluation aux seuils (C.f. section 1.1.2). Plus les fonctions sont éloignées des valeurs seuil, plus il y a de chance que la composition linéaire provoque de larges réductions de domaine. Son intérêt quantitatif immédiat peut être mesuré par la somme des distances de chaque fonction au seuil correspondant pour lesquelles le critère est vérifié.

1.3 Raisonnement symbolique sur des objets

Nous avons étudié deux domaines d'application :

1) Les problèmes d'emploi du temps nous ont fourni des exemples d'utilisation de contraintes de cardinalité générique et d'étude du comportement de la règle *CardGen–CardGen* et *linlin* pour aboutir plus vite à l'optimum.

2) La preuve de théorème est une des applications de la satisfaction de contraintes objets. Supposons un théorème modélisé par la déduction d'un but à partir d'un ensemble d'hypothèses. D'après le principe de réfutation, le théorème est vrai si et seulement si la conjonction des hypothèses et de la négation de la conclusion est absurde. L'énoncé d'un théorème est exprimable par un CSP, dont les contraintes représentent les hypothèses et la négation de la conclusion et les variables représentent les variables mathématiques. Leur domaine est théoriquement un ensemble infini d'objets mathématiques (nombres,

figures géométriques, coordonnées de points, domaine d'une fonction…) ; il est choisi très grand en pratique[73]. La résolution de ce CSP donne la preuve du théorème. Si le CSP est incohérent, le théorème est vrai, s'il existe au moins une solution, le théorème est faux.

1.3.1 Déduction symbolique sur des objets géométriques

Supposons connue une ontologie des objets mathématiques comprenant les concepts de point du plan, de droite du plan, de cercle, de triangle et les propriétés géométriques de perdendicularité, parallélisme, intersection. Une droite est construite à partir de deux points par le message `P1 lineTo: P2`. L'ensemble des droites est créé par une contrainte sur V1 et V2 deux variables représentant l'ensemble des positions de P1 et P2. Un cercle défini par un centre et un rayon est représenté par son équation analytique. Il peut aussi être défini par trois points. Les points sont définis par deux coordonnées entières. Le domaine de variation des coordonnées est l'intervalle 0..N, N fixé. Pour être sûr de prouver le théorème, N doit être pris suffisamment grand. Les contraintes fonctionnelles permettent de définir des variables dont le domaine est connu en intention, par exemple le domaine d'un point est déduit par application d'un opérateur de construction de points (@) *i.e.* *P* = (0..N)x(0..N), et celui d'une droit par un opérateur de construction de droites *i.e.* *P* x *P*. Les nouvelles règles définies dans RCS modélisent les axiomes et théorèmes géométriques supposés prouvés.

Ontologie des contraintes géométriques utilisées

- D1 isPerpenducarTo: D2 ⇔ D1 $\perp$D2. D1 isParallelTo: D2 ⇔ D1 // D2.
- P1 isInLine: D1 ⇔ P1 est sur la droite D1.
- P1 isMiddleOf: A segmentTo: B ⇔ P1 est le milieu du segment AB.
- *extremity*(AB) renvoie B
- *origin*(AB) renvoie A.
- centre(C1) = P ⇔ P est le centre du cercle C1.
- Rayon(C1) = R ⇔ R est le rayon du cercle C1.
- D1 isNotPerpendicularTo: D2 <= >D1 $\neg\perp$ D2.

Exemple

Si deux droites sont perpendiculaires à une troisième, alors elles sont parallèles :

D1 IsPerpendicularTo: D2 , D3 IsPerpenducularTo: D2 ➔ D1 isParallelTo: D3.

Si deux droites sont perpendiculaires, le produit de leur coefficients directeurs vaut –1 :

D1 IsPerpendicularTo: D2, D1 = A1 lineTo: B1, D2 = A2 lineTo: B2
➔

$$\frac{Y_{B1} - Y_{A1}}{X_{B1} - X_{A1}} = -\left(\frac{Y_{B2} - Y_{A2}}{X_{B2} - X_{A2}}\right), \text{ soit } (Y_{B1} - Y_{A1})*(X_{B2} - X_{A2}) + (Y_{B2} - Y_{A2})*(X_{B1} - X_{A1}) = 0.$$

Réciproquement, D1 notPerpendicularTo: D2 ➔
$(Y_{B1} - Y_{A1})*(X_{B2} - X_{A2}) + (Y_{B2} - Y_{A2})*(X_{B1} - X_{A1}) \neq 0$.

Soit un cercle passant par trois points A,B,C. Son centre I et sont rayon R sont donnés par la contrainte :

$$[(X_A - X_I)^2 + (Y_A - Y_I)^2 = R^2] \wedge [(X_B - X_I)^2 + (Y_B - Y_I)^2 = R^2] \wedge [(X_C - X_I)^2 + (Y_C - Y_I)^2 = R^2]$$

P isMiddleOf: A segment: B ➔ P isInLine: (A lineTo: B)

[73] Le domaine est un plan, une droite ou un espace selon le type de l'objet.

P isMiddleOf: A segment: B ➔ PA = PB

<u>Symétrie de la perpendicularité et du parallélisme:</u>

D1 isPerdendicularTo: D2 ⇔ D2 isPerpendicularTo: D1.
D1 isParallelTo: D2 ⇔ D2 isParallelTo: D1.

Cette équivalence est gérée au niveau de la substituion des prémisses des règles impliquant une telle contrainte.

En BackTalk, toutes les contraintes du type V = f(U,W) s'expriment par `V := (U btPerform: #f arg: W) asVariable` ; cette expression Smalltalk construit une contrainte de type `BTPerformCt` sur une relation à deux arguments variables. Par exemple, le domaine de A est défini par {(X,Y) / X in (0..N) et Y in (0..N)} et mis automatiquement à jour quand les domaines de a1 et a2 changent, grâce à la contrainte BackTalk `A := (a1 @ a2)asVariable.`

Notations :

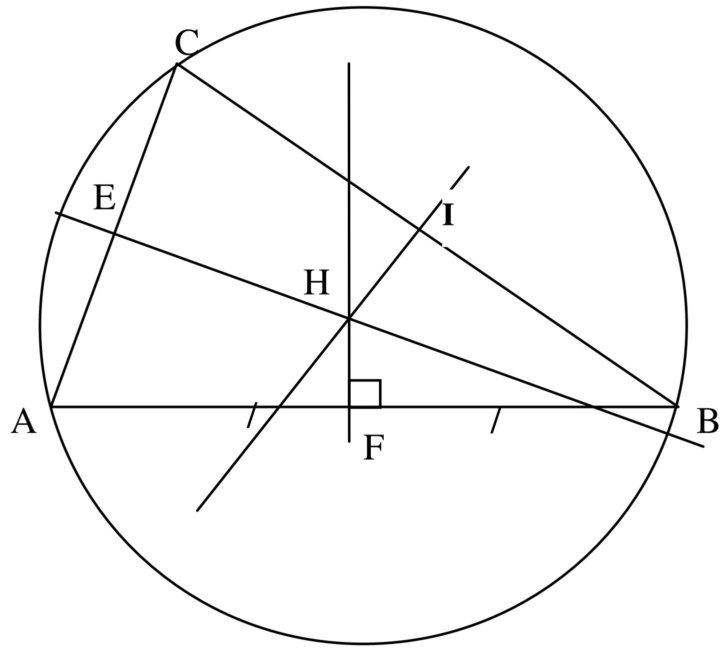
On note *AB* la variable définie par *AB = A lineTo: B*. La partie condition des règles gère la vérification de l'existence de cette contrainte. On utilisera indifféremment les notations géométriques et les notations des contraintes objets fonctionnelles.

Le théorème du cercle circonscrit à un triangle

Enoncé : "Dans un triangle, le centre du cercle circonscrit est le point d'intersection des médiatrices."On suppose connu le fait qu'elles se coupent en un seul point.

<u>Hypothèses</u> : les trois médiatrices, leur point d'intersection unique et le cercle circonscrit passant par les sommets du triangle.

<u>Conclusion</u> : le centre du cercle est le point d'intersection des médiatrices.



<u>Variables de base (coordonnées de points) :</u>

a1,a2,b1,b2,c1,c2,e1,e2,f1,f2,h1,h2, i1,i2 de domaines 0..N, N étant un entier.

Contraintes de construction des objets mathématiques :

Les points :

A = a1 @ a2. B = b1 @ b2. C = c1 @ c2. E = e1 @ e2.

I = i1@i2. H = h1 @ h2. F = f1 @ f2.

Les droites :

AB = A lineTo: B. BC = B lineTo: C. AC = A lineTo: C.

FH = F lineTo: H. IH = I lineTo: H. EH = E lineTo: H.

Contraintes des médiatrices :

AC isPerpendicularTo: EH. E isInLine: AC. AB isPerpendicularTo: FH.

BC isPerpendicularTo: IH. F isInLine: AB. I isInLine: BC.

I isMiddleOf: B segment: C. F isMiddleOf: A segment: B. E isMiddleOf: A segment: C.

Contraintes du cercle :

Soient j1,j2 dans 0..N. J = j1 @ j2. JA = JB. JA = JC. JB = JC.

Négation de la conclusion :

J ≠H, HA ≠ HB ∨ HA ≠ HC ∨ HB ≠ HC

Raisonnement symbolique effectué :

Propriété des points de la médiatrice d'un segment (*MediatriceEqui*):

Equidistance des extrémités du segment ⇔ Pour tout P, A et B,

P isMiddleOf: A segment: B , EP isPerpendicularTo: (A lineTo: B) ➔ EA = EB.

Propriété d'unicité des points vérifiant les mêmes proriétés (*UnicityPoint*) :

Pour tout E, H, A et B, EA = EB, HA = HB, E isInLine : D1, H isInLine: D2 ➔ E = H.

I isMiddleOf: B segment: C, BC isPerpendicularTo: IH → HC = HB.

F isMiddleOf: A segment: B, AB isPerpendicularTo: FH → HA = HB.

E isMiddleOf: A segment: C, AC isPerpendicularTo: EH → HA = HC.

A ce moment deux règles du même type s'activent et détectent l'incohérence :

P = Q, P ≠ Q ➔ *fail* et P = Q, (P ≠ Q) ∨ R ➔ R. Cette dernière s'applique deux fois avant que la première puisse s'activer.

HB = HA, (HA ≠ HB) ∨ (HA ≠ HC ∨ HB ≠ HC) → HA ≠ HC ∨ HB ≠ HC.

HA = HC, (HA ≠ HC) ∨ HB ≠ HC → HB ≠ HC

HB = HC, HB ≠ HC → ***fail***

Ce raisonnement repose sur le fait que les variables résultat de contraintes fonctionnelles générales sont uniques ; en effet une contrainte fonctionnelle abstrait la définition intentionnelle de sa variable résultat.

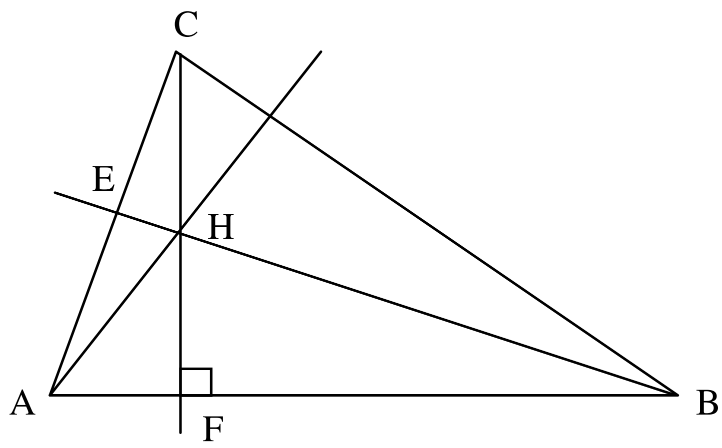
Le théorème de l'orthocentre

Enoncé : Dans un triangle, les trois hauteurs se coupent en un seul point

Hypothèses : trois points A,B,C, deux hauteurs issues de B et C, une demi-droite issue de A qui coupe les hauteurs en leur point d'intersection.

Conclusion : la demi-droite issue de A est perpendiculaire à un côté du triangle.

Cet exemple est inspiré de [Monfroy 1996a].



<u>Variables de base (coordonnées de points) :</u>

a1,a2,b1,b2,c1,c2,e1,e2,f1,f2,h1,h2 de domaines 0..N, N étant un entier.

<u>Contraintes de construction des objets mathématiques :</u>

Les points :

A = a1 @ a2. B = b1 @ b2. C = c1 @ c2. E = e1 @ e2. F = f1 @ f2. H = h1 @ h2.

Les droites :

AB = A lineTo: B. EB = E lineTo: B. AC = A lineTo: C. CF = C lineTo: F.
AH = A lineTo: H. BC = B lineTo: C. EB = E lineTo: B.

<u>Contraintes des hauteurs :</u>

AC isPerpendicularTo: EB. AB isPerpendicularTo: CF.
F isInLine: AB. H isInLine: CF. H isInLine: EB. E isInLine: AC.

<u>Négation de la conclusion :</u>

AH isNotPerpendicularTo: BC.

Une des démonstrations géométriques de ce théorème consiste à ajouter les parallèles aux côtés passant par le sommet opposé. Ces droites se coupent car (A,B,C) est un triangle. Supposons donc les droites d1, d2 et d3 ajoutées et montrons que l'on peut prouver le théorème par un raisonnement symbolique. Le problème contient alors les contraintes supplémentaires :

x1,x2,y1,y2,z1,z2 dans 0..N. X = x1@ x2. Y = y1@ y2. Z = z1@ z2.
XY = X lineTo: Y. YZ = Y lineTo: Z ; XZ = X lineTo: Z.
XY isParallelTo: AB. YZ isParallelTo: AC. XZ isParallelTo: BC.
B isInLine: YZ. A isInLine: XZ. C isInLine: XY. (0)

De plus les règles de RCS suivantes sont définies :

Pour tout point A,B et droite D1,
(*PointLine*) D1 = A @ B ➔ A isInLine: D1, B isInLine: D1.
(*ParallelPoint*) P isInLine :D1, D1 = A lineTo: B, D1 isParallelTo: D2
➔ PA isParallelTo:D2 , PB isParallelTo: D2

<u>Règle de définition d'un parallélogramme (*Parallelogram*)</u>

Pour tout point E,F, G, H, EF // GH, EG // FH ➔ EF = GH, EG = FH.

<u>Règle ParaPerpen</u>

Pour toute droite D1, D2, D1 // D2, D1 ⊥ D3 ➔ D2 ⊥ D3.

Règle de transitivité (*TransitivityEqual*) : V = W, W = U ➔ V = U

Règle de cohérence des points sur une droite (*linePoint*)

Pour tout op ∈ {⊥,//}, T isInLine: AB, D1 *op* AT ➔ D1 *op* AB.

Pour tout op ∈ {⊥,//}, T isInLine: AB, D1 *op* AB ➔ D1 *op* AT, D1 *op* BT.

Règle de concourance des médiatrices d'un triangle (*MédiatriceTriangle*)

Pour tout I, A, B, C, , A', B', C',
A' isInLine: BC. C' isInLine: AB. B' isInLine: AC.
A'B=A'C, B'A = B'C, C'A = C'B , IA'⊥ BC, IB'⊥ AC ➔ IC' ⊥ AB.

Par ailleurs pour modéliser les équivalences sans introduire de cycle (garantie de terminaison de RCS), il faut choisir une orientation. Ainsi, A ⇔ B est implémenté par A' ➔ B' , où A'et B' sont les formes normalisées de A et B, *i.e.* il existe une séquence de réécriture terminante B→…→ B_0, alorsB_0=B'. Le fait d'imposer que les règles manipulent les contraintes simplifiées évite que des règles soient court-circuitées alors qu'elles sont sémantiquement candidates. Par exemple, le système de simplification contient la règle

P isMiddleOf: A segment: B ➔ P isInLine: AB, PA = PB

La règle *MédiatriceEqui* est donc implémentée par

PA = PB, P isInLine: AB, EP isPerpendicularTo: AB ➔ PA = PB.

Raisonnement symbolique effectué :

(*ParallelPoint*) : CY isParallelTo: AB, CX isParallelTo: AB
BY isParallelTo: AC, BZ isParallelTo: AC
AX isParallelTo: BC, AZ isParallelTo: BC

(*Parallelogram*) : CY = AB, BY = AC
CX = AB, AX = BC
AZ = BC, BZ = AC.

(*TransitivityEqual*) : CY = AB , AB = CX →CY = CX (1)
idem on déduit AX = AZ et BZ = BY. (2)

(*ParaPerpen*) : AC ⊥ EB, BY isParalelTo: AC → BY ⊥ EB.
AC ⊥ EB, BZ isParalelTo: AC → BZ ⊥ EB.
AB ⊥ CF, CY isParalelTo: AB → CF ⊥ CY.
AB ⊥ CF, CX isParalelTo: AB → CF ⊥ CX.

(*linePoint-2*) : H isInLine: EB, BY ⊥ EB → BY ⊥ HE, BY ⊥ HB.
H isInLine: CF, CY ⊥ CF → CY ⊥ HC, CY ⊥ HF.

(*MédiatriceEqui*) : BY ⊥ EH, BZ = BY→ HY = HZ
CY ⊥ HF, CX = CY → HX = HY.

(*Transitivity*) : HX = HY, HY = HZ → HX = HZ.

(*linePoint-1*) : C isInLine : XY, CY ⊥ HF → XY ⊥ HF (3)

B isInLine : YZ, BY ⊥ EH → YZ ⊥ EH (4)

(*MédiatriceTriangle*) : A partir de (0), (1), (2), (3), (4)
B isInLine: YZ, A isInLine: XZ, C isInLine: XY,
XY ⊥ HF, YZ ⊥ EH, CY = CX, AX = AZ , BZ = BY,

➔ HA ⊥ XZ. (5)

(*linePoint-2*) : A isInLine: XZ, HA ⊥XZ → HA ⊥ XA, HA ⊥ ZA

(*ParaPerpen*) : HA ⊥ ZA, BC // AZ → HA ⊥ BC.

L'ajout de cette dernière contrainte redondante provoque une incohérence car la règle D1 isPerpendicularTo: D2, D1 isNotPerpendicularTo: D2 ➔ ***fail***

1.3.2 Déduction symbolique sur des contraintes temporelles

Les problèmes d'ordonnancement sont des problèmes d'optimisation combinatoire. BackTalk résout ce type de problèmes en approchant la solution optimale pas une méthode systématique par satisfaction de contraintes : chaque fois qu'une solution est trouvée par le système CSP+RCS, une contrainte est rajoutée pour forcer le système à ne chercher que les solutions meilleures que la solution déjà trouvée ($coût_{i+1} < coût_i$). L'influence de RCS sur des problèmes d'optimisation avec contraintes temporelles se manifeste globalement par une réduction du nombre de retour-arrières, au prix d'un surcoût en temps variant de 14% à 53,5% ! (C.f. tableau 4).

Problème	Résolution CSP		Résolution CSP+RCS		Nombre de contraintes en début de résolution
	Temps (sec)	Retour-arrières	Temps (sec.)	Retour-arrières	
Chr3 (coût optimal 180)[74]	0,08	79	42,88	44	13
MT06 (coût optimal : 55)	9,379	593	36,311	537	54
Newspaper (coût optimal : 210)	0,205	31	3,043	28	73
Bridge (coût optimal : 104)	2,096	555	79,513	680	95

Tableau 18 : résolution de problèmes d'optimisation, avec et sans utilisation du raisonnement symbolique[75].

En pratique RCS est plus adapté en prétraitement du problème afin d'isoler automatiquement les contraintes redondantes pertinentes, avant de faire le premier choix. Ensuite ces contraintes sont rajoutées à la main, puis le problème est résolu avec des techniques d'optimisation combinatoire dédiées [Laburthe 1998].

Gestion d'emploi du temps

Nous avons étudié la gestion de planning d'un centre d'éducation ; le planning est prédécoupé en plages horaires différentes pour chaque jour et requérant la présence d'un nombre minimum d'employés. L'objectif du système d'emploi du temps est d'affecter à chaque plage horaire la quantité de personnes nécessaire telle que la charge totale soit équilibrée et répartie, tout en respectant les contraintes liés au règlement et le maximum de préférences. Les données du problème sont : un ensemble d'éducateurs salariés, une matrice des plages horaires et une matrice des charges de chaque plage horaire[76].

Contraintes :

[74] Définition présentée en annexe. Problème témoin dans l'expérimentation.
[75] Le tableau montre les temps de résolution en secondes. Quand le nombre de retour-arrières est nul, le solveur a résolu le problème sans mauvais choix. Dans tous les tests on a restreint RCS aux règles de propagation et de réécriture unaire et binaire.
[76] La plage horaire joue le même rôle qu'une tâche d'un problème d'ordonnancement.

- Assurer la présence du nombre minimum d'éducateurs requis dans chaque plage horaire,
- Respecter un nombre d'heures hebdomadaire minimum et maximum, utiliser le cota d'heures de chaque éducateur sans les gaspiller,
- Répartir les difficultés entre les éducateurs,
- Respecter les séquences de travail (par exemple si un éducateur ferme le centre le soir alors il ne travaille pas le lendemain matin)
- Tenir compte des préférences de chaque éducateur.
- Un éducateur ne peut être affecté qu'à une seule plage horaire par jour et ne peut effectuer une partie seulement de plage horaire.

Manipulation de contraintes de cardinalité

Afin de répartir les difficultés entre éducateurs, on poste une contrainte de cardinalité sur les tâches de chaque éducateur. Le nombre de tâches par semaine, incluant une fermeture du centre doit être borné par un intervalle constant 1..3. Pour chaque éducateur, une telle contrainte est créée. Cette contrainte s'exprime ainsi :

> " Soit E un éducateur, soit S l'ensemble des plages horaires à affecter à E, le nombre de variables *x* de S telles que (*x* estUneFermeture) est égale à V, où V est une variable entière de domaine 1..3." [77]

Le centre fonctionne selon la règle de gestion suivante : Il y a au plus autant de fermetures que de plage horaires ne faisant pas l'ouverture. Autrement dit, soit S, l'ensemble des affectations d'employés, de domaine l'ensemble des plages horaires possibles pour un jour. Soient T dans 1..3 et U dans 0..N (N est la taille du domaine initial d'une variable, N>3).

Card(S, (x estUneFermeture)) = T, Card(S,(NON(x estUneOuverture))) = U. T ≤ U

Le filtrage déduit que U est dans 1..N après avoir énuméré tous les n-uplets de S pour U = 0. La règle *cardGen-CardGen* déduite que, puisque si une plage horaire est une fermeture elle ne peut être une ouverture, *i.e.* une fonction a été définie afin de rendre VRAI pour l'implication logique d'expression (*x* estUneFermeture ➔ NON(*x* estUneOuverture)), alors U ≥ 1.

2 Implantation

Les règles sont organisées en arbre d'héritage simple, chaque classe de règle étant conçue selon le *pattern Strategy*. Chaque classe de règle abstraite se particularise en règles de réécriture ou en règles de propagation. Une instance représente une règle générique. Chaque type de règles est implanté par une sous-classe de la classe abstraite `RCSRule`. Lorsque le problème est résolu dans son ensemble, RCS possède une seule instance de chaque type de règle. Dans ce cas le type de règle est équivalent à la règle générique que l'on aurait définie si l'on avait été dans un langage logique. En revanche, si le problème est résolu par morceaux, une instance de RCS est affectée à chaque sous-problème. Les sous-problèmes n'étant pas

[77] Le message estUneFermeture (resp. estUneOuverture) envoyé à une plage horaire renvoie vraie si la plage se termine après 21h00 (resp. commence avant 9h00) et faux sinon.

forcément disjoints, une règle est susceptible d'intervenir dans différentes instance de RCS, qui inclut alors chacun une instance de la classe idoine.

Un certain nombre d'informations sont stockées au niveau des règles.

- le solveur qui contient le problème manipulé par la règle (attribut `solver`),
- la liste des contraintes candidates (attribut `ctCandidates`),
- Les règles multi-prémisses distinguent la contrainte active des autres contraintes vérifiant la partie condition (attribut `activeCt`),
- Les règles de propagation maintiennent à jour l'historique des n-uplets de contraintes qui ont déjà été utilisés (attribut `forbiddenTuples`), ceci afin d'éviter les exécutions redondantes,
- Les règles de réécriture mémorisent la liste des contraintes à supprimer à la fin de l'exécution de la règle (attribut `contraintesToRemove`),
- Une règle supprime ou non sa contrainte active à la fin de son exécution. Ce paramètre est initialisé en exécutant `mustRemoveActiveCt(aBoolean)`.

2.1 Comportement des règles

Les règles suivent un protocole. A l'initialisation de la règle, la méthode `selectCts` trouve les n-uplets de contraintes structurellement candidates, i.e. dont le type correspond aux prémisses, qui vérifient les conditions de la règle, ne sont pas obsolètes ni totalement instantiées. L'application complète de la règle consiste à trouver toutes les prémisses dans le CSP et à exécuter toutes les instances de la règle. La méthode responsable, `applyGlobal` s'utilise au niveau du système CSP+RCS dans la méthode `processRCSGlobal`.

```
CSPRCSRulesSolver.processRCSGlobal
       rcsrules do: [:rfr | rfr applyGlobal]

RCSRule.applyGlobal
  | setOfPremisses |
    setOfPremisses := self findPremissesGlobal.
    setOfPremisses do:  [:tuple |
                     self execute: tuple.
                     forbiddenTuples add: tuple]
```

L'application partielle consiste à sélectionner les règles dont au moins une prémisse est unifiée avec une contrainte C (mémorisée dans `activeCt`) ; il reste alors à trouver les contraintes s'unifiant avec les prémisses restantes, puis à exécuter les instances de règles portant sur C (méthode `apply(C)`). Dans ce cas, la contrainte C est choisie dynamiquement parmi les contraintes mémorisées au niveau du système CSP+RCS. Initialement l'ensemble équivaut à l'ensemble des contraintes du problème[78].

```
RCSRulesSolver.processRCS
   ctsToPropagate do: [:c | c propagateCompositionsAndRewrite]

RCSRule.apply: ct
  | setOfPremisses ntuple |
    self activeCt: ct.
    setOfPremisses := self findPremisses.
    setOfPremisses do:  [:tuple |
        activeCt isEqualCt
```

[78]La méthode `propagateCompositionsAndRewrite` applique toutes les règles connectées à la contrainte.

```
        ifTrue: [ntuple := Array with: tuple with: activeCt]
        ifFalse: [ntuple := Array with: activeCt with: tuple].
      self execute: ntuple.
      forbiddenTuples add: ntuple
   ]
```

Une règle possède trois opérations principales. La sélection des contraintes est réalisée soit par `findPrémisses` soit par `makeConditionCSP`. L'action est exécutée par `execute(tuple)`. L'application de la règle regroupe la substitution des prémisses compatibles avec la contrainte active et l'exécution de l'action pour chaque n-uplet. De plus elle signale à la règle si elle a réussi ou non. La méthode publique `apply(aConstraint)` implante ce comportement au niveau des classes abstraites.

Enfin il est possible de déconnecter une règle du problème, *via* l'envoi du message `destroy`.

2.1.1 Eviter les raisonnements redondants

Lorsqu'une règle s'applique sur un n-uplet *T* de contraintes et produit un ensemble *T'* de contraintes, si l'exécution de la règle n'a supprimé aucune contrainte de *T*, il se peut que le même n-uplet *T* soit de nouveau sélectionné plus tard. Pour éviter cela la règle mémorise les n-uplets de *T* comme des ensembles de contraintes interdites. Grâce à la fonction `isNotForbiddenTuple(aTuple)`, l'application de la règle sélectionne les n-uplets dont les contraintes ne sont pas interdites entre elles.

```
LinearCompositionRule.applyGlobal
  |setOfPremisses newCts|
    setOfPremisses := self findPremisses.
    SetOfPremisses isEmpty ifTrue: [self fail. ^nil].
    NewCts := BTCollection new: 10.
    setOfPremisses do: [:tuple |
        res := self execute: tuple.
        newCts addAll: res.
        ForbiddenTuples add: tuple.
        res do: [:ct | ct isForbiddenCtFor: activeCt.
                       ct isForbiddenCtFor: tuple]
        ]
```

Figure 26 : code de la méthode d'application de la composition de contraintes linéaires

2.1.2 Déclenchement de règles

Nous distinguons les règles créées à la volée et les règles persistantes. Les premières sont en général des règles simples, telles les règles de réécriture unaires et sont déclenchées en même temps que leur création. Ce mode est particulièrement ntéressant lorsque le langage hôte gère la destruction des objets créés non référencés, par exemple *via* un « ramasse-miette informatique » (*Garbagge Collector*). Lorsqu'une règle générique possède un grand nombre d'instances candidates, il est préférable de représenter la règle avec un seul objet au lieu d'en recréer un à chaque fois. La règle est alors créée dès le début de la résolution ; l'ensemble des contraintes candidates est mis à jour en cours de résolution.

2.1.3 Lien contrainte – règle

L'utilisation de règle persistante nécessite de conserver un lien bidirectionnel entre les contraintes et la règle. D'une part une règle connaît l'ensemble des contraintes candidates à la substitution de ses prémisses ; d'autre part une contrainte est connectée aux types de règles pour lesquelles elle est structurellement candidate (attribut *rulesClasses*). Le second lien sert à mettre en place le premier lien lors de l'initialisation d'une résolution : pour chaque contrainte, le système CSP+RCS crée les règles génériques dont le type appartient à *rulesClasses*, telle qu'il y ait un seul objet-règle pour chaque problème. Chaque règle générique est ainsi initialisée avec les contraintes structurellement candidates du problème. Notons que l'ensemble des règles structurellement candidates est l'ensemble maximal des contraintes susceptibles d'être effectivement candidates, *i.e.* semi-unifiées avec les prémisses de la règle. Toute contrainte est automatiquement reliée à la règle de réécriture d'instantiation.

Par ailleurs deux démons relient une contrainte à ses règles de réécriture et de propagation ; ils servent à la mise à jour du premier lien, quand une contrainte structurellement candidate est ajoutée ou supprimée du problème. L'utilisation de démons permet de limiter la taille des objets-contraintes.

```
CSPRCSRulesSolver.addConstraint: aCt
   super addConstraint: aCt.
         …
   self initializeRulesFor: aCt.

CSPRCSRulesSolver.initializeRuleFor: aCt
   aCt ruleClasses do: [:rcsClass |
                     self initializeRules: rcsClass with: aCt]

CSPRCSRulesSolver.initializeRules: aRCSClass withCt: aCt
    | rr |
         rr := rcsRules detect: [:r | r class == aRCSClass]
                ifNone: [rcsRules add: (aRCSClass solver: self)].
         rr addCt: aCt.
         rr postConstraintDemonOn: aCt[79]
```

Figure 27 : création du lien à l'ajout d'une contrainte

2.2 Sélection des contraintes

La sélection des règles se compose de deux étapes, le préfiltrage sur le type des contraintes et la sélection des contraintes ou vérification des conditions. Le préfiltrage statique sert à initialiser les règles persistantes, à la création d'une contrainte et n'existe pas si la règle est créée à la volée. La sélection dynamique est réalisée soit de manière algorithmique (*findPremisses*) soit par résolution d'un problème de satisfaction de contraintes dont les solutions sont les n-uplets de contraintes qui vérifient les prémisses (*findPremissesCSP*). Les contraintes du CSP expriment les conditions des prémisses de la règle.

[79] *PostConstraintDemonOn(aConstraint)* initialise le démon de *aConstraint*. La réécriture d'instantiation possède un démon particulier car toute contrainte est candidate à cette règle. Ce démon pourrait être remplacé par un pointeur sur la règle, dans la mesure où une seule règle implante la réécriture d'instantiation. Il est conservé par soucis d'homogénéité.

2.2.1 Préfiltrage statique

Il réalise l'association décrite comme le premier lien dans la section 2.1.3, au niveau des types. Il a lieu à l'ajout d'une contrainte et utilise la liste de classes de règles rendue par la méthode `rulesClasses`, comme par exemple celle des contraintes linéaires ci-dessous.

```
BTLinearCt ruleClasses
  ^super ruleClasses,
    (Array with: AllDiffLinRule with: LinearComposeRule)
```

Modélisation par le pattern Visitor

En BackTalk, les contraintes sont organisées en arbre d'héritage simple ; une classe de contraintes hérite du lien de sa super-classe vers les types de règles. Si la classe de contraintes CtA hérite de la classe CtB, alors pour toute règle R contenant une prémisse Pi préfiltrée par CtB, R est aussi préfiltrée par CtA.

CtB $\longrightarrow$ {R1,R2 : [(…, Pi: CtB,…) → uneAction]}

$\uparrow$

CtA $\longrightarrow$ {R1,R2} $\cup$ {R3 : [(…, Pj: CtA,…) → uneAction]}

Figure 28 : héritage de règles de raisonnement symbolique

Néanmoins dans le cadre d'un outil d'expérimentations, la liste des règles de RCS est appelée à être modifiée. Afin d'éviter de modifier la classe de la contrainte, nous proposons d'implanter le préfiltrage statique selon le *pattern Visitor.*

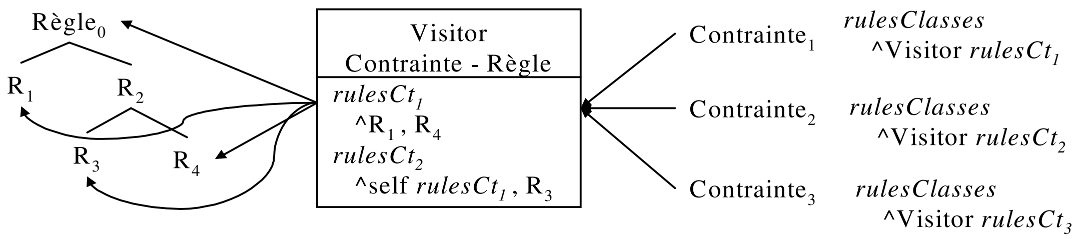


Figure 29 : préfiltrage selon le pattern Visitor

L'utilisation d'un *visitor* accroît la modularité de l'architecture ; en contre partie, elle augmente la quantité d'envois de message entre objets. L'implantation d'un lien direct automatiquement initialisé nécessiterait l'analyse de la partie condition des règles, analyse qui serait à la charge du compilateur ou de l'interpréteur de règles. Cependant, cette représentation est difficilement envisageable dans les langages compilés ou partiellement compilés, lorsque la partie condition est représentée par une méthode, car le contenu des méthodes n'est pas accessible depuis un programme.

2.2.2 Algorithme de sélection de contraintes

La méthode `findPremissesGlobal` calcule toutes les instances de la règle générique. Si une des contraintes est connue[80], la méthode `findPremisses` est utilisée conjointement avec `apply(aCt)`; la sélection revient alors à trouver les instances de la règle générique contenant `activeCt`. Par exemple, la règle *ProduitSubstition1* (section 3.4.2) s'exécute

[80] La contrainte active en l'occurrence.

globalement ou à partir d'une contrainte Produit (`ProductCt`) ou Egalité (`EqualityCt`). Les détails d'implémentation de cette règle sont donnés à titre d'exemple en annexe.

Dans cette approche, le filtrage est ramené à l'évaluation d'une conjonction de conditions implantée en dur. En remplacement d'un algorithme de substitution sur des objets, nous proposons d'utiliser un langage de contraintes générales pour exprimer cette condition à un niveau plus haut.

2.2.3 Substitution par résolution d'un CSP

D'un point de vue informel, la substitution des prémisses vise à trouver les éléments corrects parmi un ensemble donné de contraintes d'un problème (notons le P_0). Nous proposons de la modéliser comme la résolution d'un problème de satisfaction de contraintes (notons le P_r) ; les variables de P_r ont pour domaine un ensemble de contraintes inclues dans P_0 (notons les *cVariables*); les contraintes de P_r traduisent les conditions de la règle. La résolution de P_r produit des n-uplets de contraintes de P_0 vérifiant les conditions en prémisse de la règle. Cette approche est possible si d'une part, les contraintes sont réifiées et donc manipulables, et d'autre part si le langage autorise la définition de contraintes prédicatives générales sur des domaines non ordonnés d'objets ; en BackTalk, les contraintes `BTPerformCt et BTBlockCt` permettent d'exprimer de telles relations.

Contraintes pour l'expression des prémisses des règles de RCS

Les *cVariables* sont représentées dans une classe particulière de variables où les éléments du domaine sont des objets-contraintes. De plus nous avons défini un langage de contraintes prédicatives et fonctionelles sur les *cVariables* ; elles portent sur les caractéristiques des contraintes de P_0. Soit *c* une *cVariable* de P_r à déterminer. Les contraintes et mesures suivantes ont été définies :

1) <u>Arité de *c*</u>. La contrainte unaire d'arité `ArityCt` signifie que le nombre de variables d'une contrainte est bornée. Par exemple, *c.arityIs*(α).

2) <u>Fonctions donnant accès aux variables d'une contrainte</u>. Ces fonctions sont modélisées par une variable-objet, elle-même contrainte par une contrainte fonctionnelle générale (`PerformCt`) ; la création de cette dernière n'est pas visible depuis le langage de contraintes, l'utilisateur manipulant des envois de messages. Par exemple, *c.nthVariable*(i) = T, *c.x* et *c*.y si *c* représente une contrainte binaire, *c.résultat* pour une contrainte fonctionnelle de type Z= f(X,Y) ou Z= f(X).

3) <u>Nombre de variables communes à deux contraintes</u>. La contrainte binaire `CommonElementsCt` borne la quantité $|Variables(c1) \cap Variables(c2)|$. Par exemple, *c1.commonVariablesSizeWith*(*c2*) = U.

4) <u>Appartenance (`InSetCt`) ou non appartenance (`NotInSetCt`) à un ensemble défini en intention</u>, par exemple les variables d'une contrainte qui est la valeur d'une *cVariable* à déterminer. La sémantique de la contrainte binaire *InSetCt*(*setV1, setVar*) est la suivante :

- *setV1* est une *cVariable*.
- *setVar* est une variable-objet dont le domaine est soit un ensemble d'objets soit un ensemble d'ensemble. Le premier cas correspond à une instantiation du second cas et revient à une contrainte de domaine. Dans ce dernier cas sa valeur est

soumise à une contrainte fonctionnelle rendant un ensemble. Par exemple, si *setVar* = *Variables*(c_i), c_i étant une *cVariable*, le domaine de *setVar* est l'ensemble de tous les ensembles de variables pour toute contrainte du domaine de $c_{i.}$

- **si** *dom*(*setVar*) est une collection d'atomes **alors** *value*(*setV1*) appartient à *setVar*[81]

  sinon *dom*(*setVar*) est un ensemble de sous-ensembles $\{S_1,\ldots, S_k\}$ et *value*(*setV1*) appartient à l'un des S_i.

- Il doit exister une valeur *x* dans *dom*(*setV1*) et un ensemble *s* dans *dom*(*setVar*) tels que *x* appartient à *s* si x est un atome, **ou** *x* est un sous-ensemble de *s*, si x est un ensemble.

- Toutes ces contraintes sont au plus binaires et filtrées par arc-cohérence complète. Une méthode de cohérence partielle pour le second cas effectuerait au minimum :

  Si *setVar* est instantiée **alors**
  $dom(setV1) := \{c \in dom(setV1), c \in value(setVar)\}$ ou
  $dom(setV1) := \{c \in dom(setV1), size(c) \leq size(value(setVar))\}$.
  Si *setV1* est instantiée **alors**
  $dom(setVar) := \{c \in dom(setVar), value(setV1) \in c\}$ ou
  $dom(setVar) := \{c \in dom(setVar), size(c) \geq size(value(setV1))\}$

  Dans le cas où *setV1* est une *cVariable*, c est une contrainte de même que *value*(*setV1*) ; la fonction *size*(aConstraint) renvoie alors l'arité de la contrainte.

Voici à titre d'exemple le CSP spécifié pour l'instantiation de la règle Produit-Substitution1 :

---

*"x*y= T ; x = b*y+c ➔ b*y^2 + c*y = T ; x*y=T ; x=y ➔ y²=T"*

Soit P_0 le problème à résoudre.
Soient c_1 une *cVariable* telle que $dom(c_1) = \{ c \in cts(P_0), c.class = \#Produit\}$
Soient c_2 = cVariable telle que $dom(c_2) = \{ c \in cts(P_0),$
$c.class = \#EgalitéLinéaireBinaire\}$

Arity(c_1) = 3. Arity(c_2) = 2.
|Variables(c_1) Variables(c_2) | = 2.
*"c1 est une contrainte orientée x*y =T"*
$c_1.x \in Variables(c_2)$. $c_1.y \in Variables(c_2)$. $c_1.result \in Variables(c_2)$.

(coefficientForX(c_2) ≠0) OU (coefficientForY(c_2) ≠ 0).

L'algorithme général de *substitution* par résolution du CSP P_r et la construction des n-uplets sont implantés dans la classe abstraite `RCSRule` (méthode publique *findPremisesCSP*). La méthode *makeConditionCSP* qui construit et retourne le CSP est redéfinie dans les sous-classes.

Exemple : CSP de sélection pour la règle ProduitSubstitution1 (section 3.4.2)

```
ProductSubstitutionRule. makeConditionCSP
```
*"x*y= T ; x = b*y+c ➔ b*y^2 + c*y = T*

[81]premier cas

```
x*y=T ; x=y ➔ y²=T"
 | pbm c1 c2 vars |
  pbm := BTSolver new label: 'ProductSubstitionRule-CSP'.
  c1 := BTConditionVariable label: 'c1' fromCtClass: ProductCt
                          inSolver:solver.
  c2 := BTConditionVariable label: 'c2' fromCtClass: EqualityCt
                          inSolver: solver.
  vars := Array with: c1 with: c2.
  c1 arityIs: 3. c2 arityIs: 2.
  (c1 commonVariablesWithSize: c2) @= 2.
  (c1 x) inSet: (c2 variables). "x et y en commun ,mais pas T ."
  (c1 y) inSet: (c2 variables).
  (c1 result) notInSet: (c2 variables).
  "un des coefficient non nul dans l'égalité"
  BTBlockCt on: (Array with: c2)
         block: [:v2 |(v2 coefficientForX) ~= 0 or:
                      [v2 coefficientForY) ~= 0]].
^pbm variablesToInstantiate: vars  "initialiser Pr"
```

Exemple : code de la méthode responsable de la sélection de contraintes par résolution d'un CSP de contraintes générales objets

```
RCSRule.findPremissesCSP
  | goodValues |
  V domainWipedOut handle:[:ex|^#()]
  "En cas d'incohérence, rendre un tableau vide."
   do: [| pbm sols s premisses |
       pbm := self makeConditionCSP.
       sols := pbm allSolutions.
       premisses := OrderedCollection new: sols size.
       s := pbm variablesToInstantiate size.
       s = 1
         ifTrue: [ sols do: [:dsol |
             sol keysAndValuesDo: [:k :v |
                (pbm variablesToInstantiate includes: k)
                    ifTrue: [premisses add: v]]]]
         ifFalse:[sols do:  [:dsol |
             goodValues := Array new: s.
             dsol keysAndValuesDo: [:k :v |
                (pbm variablesToInstantiate includes: k)
                    ifTrue:
       [goodValues at: (pbm variablesToInstantiate indexOf: k)
                  put: v]
                  ].
       premisses add: goodValues asBTCollection]].
       ^premisses
   ]
```

P_r est reconstruit à chaque application d'une règle de RCS car l'ensemble des contraintes change. Il est possible d'optimiser en mémorisant les contraintes de P_r, car celles-ci ne sont pas modifiées. Les domaines des variables de P_r doivent être mis à jour quand l'ensemble des contraintes de P_0 change en cours de résolution (retrait d'une contrainte, retrait d'une variable d'une contrainte, ajout d'une contrainte). La mise à jour est incrémentale ; elle peut être automatisée *via* une relation fonctionnelle invariante (au sens des invariants de Localizer), qui définit le domaine de chaque variable comme étant égal à l'ensemble des contraintes candidates d'une certaine classe.

Application aux contraintes fonctionnelles générales dans PerformCtRule

Soit la règle de type PerformCtRule

$$c_1 :(x_1\ op_1\ y_1) = u_1,...,c_k :(x_k\ op_k\ y_k) = u_k \rightarrow\ z_1\ f_1\ t_1,..., z_n\ f_n\ t_n$$

Rappelons qu'une contrainte fonctionnelle possède trois fonctions pour accéder à ses variables, *receveur*(c_i) = x_i, *argument*(c_i) = y_i et *résultat*(c_i) = u_i. On suppose construites les données suivantes :

- k et n.
- Le réseau de dépendance entre motifs. Un motif contient l'ensemble des contraintes qui lui correspond sous la forme d'une variable de type *cVariable*. Pour chaque variable *vM* de motif, trois ensembles sont construits : *motifsReçus*(*vM*), l'ensemble des motifs où *vM* apparaît en position de receveur, *motifsArgument*(*vM*) et *motifsRésultat*(*vM*).
- Un dictionnaire destiné à mémoriser les correspondances avec les variables de motif déjà trouvées.

Soient c_1, …, c_k les *cVariables* de P_r, telles que *dom*(c_i) = {c ∈ *contraintes*(P_0), c.class = #PerformCt}. La construction des contraintes sur la procédure suivante :

Pour tout i ∈ [1..2k],
Soit v_i une des variables des motifs en prémisse,
Pour tout o_i ∈ *MotifsReçus*(v_i), **Pour tout** a_i ∈ *MotifsArgument*(v_i),
Pour tout r_i ∈ *MotifsRésultat*(v_i),
`contraindre` *receveur*(o_i)= *argument*(a_i).
`contraindre` *receveur*(o_i) = *résultat*(r_i).
`contraindre` *argument*(a_i) = *résultat*(r_i).
`CréerIntersectionMotifReceveur`(*MotifsReçus*(v_i)).
`CréerIntersectionMotifArgument`(*MotifsArgument*(v_i)).
`CréerIntersectionMotifRésultat`(*MotifsRésultat*(v_i)).

`Procédure CréerIntersectionMotifReceveur(list)` *"list = liste de cVariables"*
Pour tout j ∈ [0.. *size*(*list*)-2], `contraindre` *receveur*(`list`[i]) = *receveur*(`list`[i+1]).
`Procédure CréerIntersectionMotifArgument(list)` *"list = liste de cVariables"*
Pour tout j ∈ [0.. *size*(*list*)-2], `contraindre` *argument*(`list`[i]) = *argument*(`list`[i+1]).
`Procédure CréerIntersectionMotifRésultat(list)` *"list = liste de cVariables"*
Pour tout j ∈ [0.. *size*(*list*)-2], `contraindre` *résultat*(`list`[i]) = *résultat* (`list`[i+1]).

L'utilisation d'un problème de CSP offre ici un avantage en clarté sur un algorithme figé et récursif ; de plus les contraintes décrivant les connexions entre prémisses étant des égalités binaires elles sont traitées efficacement par un système de résolution CSP.

2.3 Implantation du contrôle dans méta-RCS

Dans une approche objet, une stratégie est une sous-classe de la classe abstraite des invariants de contrôle, `InvariantControle`. La déclaration d'un invariant de contrôle est faite simplement en créant une instance de la classe idoine avec les paramètres *ad hoc*, dont la règle principale. Cette règle est exactement celle qui héberge la création de l'invariant (C.f. section 3). Cette manière de poser des conditions s'inspire des langages de contraintes en programmation par objets, tel que celui de BackTalk. Une stratégie complexe peut être modélisée par une succession d'invariants uniquement.

Tout invariant de méta-RCS doit implanter deux actions de base : 1) tester avant le déclenchement si la règle principale est autorisée ; 2) mettre à jour l'information après déclenchement. Si une stratégie dépend d'un événement particulier, par exemple le passage au statut *Useless*, alors les invariants inclus dans la modélisation de la stratégie doivent implanter la méthode associée à cet événement. Par conséquent la classe doit redéfinir deux méthodes publiques au minimum, `testRule(r,c)` rendant le résultat de l'évaluation des pré-conditions et `updateAfterTriggering(r,c)` mettant à jour le statut des règles et les informations propres au contrôle. Un type de contrôle possède plusieurs instances si plusieurs règles sont contrôlées par celui-ci. Notons que, même si rien ne l'empêche, il est conseillé de ne pas imposer de contrôle sur des règles créées à la volée.

RCS et méta-RCS sont liés *via* les invariants de contrôle et les règles grâce à un mécanisme de démon. En accord avec le *pattern Observer*, les objets observés sont les règles. Le démon est stocké dans un attribut de la règle (`controlDemon`). A chaque fois que la règle est informée de son changement d'état, ce démon est également informé et la transmission aux invariants connectés est automatique. Par exemple, la règle est automatiquement informée de l'échec de son application lorsque l'ensemble des prémisses est vide, la méthode responsable est `fail`, implanté comme suit :

```
fail
   succes := false.
   controlDemon setNotSucceed
```

Les états *PeutSeDéclencher* et *Appliqué* sont gérés au-dessus de la règle par les démons reliant chaque contrainte à un ensemble de règles.

Exemple : informer l'invariant qu'une composition a été inutile

L'attribut `controlDemon` se charge d'informer tous les invariants. Dans l'action de la règle (méthode `execute` ou `apply`), lorsque l'on sait qu'une instance a été utile, la règle est informée en envoyant le message `setInutile` ; cette tâche est à la charge de l'utilisateur.

Exemple: plusieurs contrôles sur une même règle

Soit d1 le démon de r1, d3 le démon de r3.
`r1 onceRuleExecution. r3 onceRuleExecution.`
"met dans le démon d1 une instance de ItOneExecution sur r1; idem pour r3"
`r1 executionAfterNbEchecs: 2.`
"met dans d1 une instance de ItExecutionAfterEchecs(2, mainRule=r1, otherRules=rules(solver))

La méthode `trigger` des *constraints-démons*[82] a été modifiée de manière à prendre en compte les stratégies.

```
ConstaintDemon.trigger
     rules do: [:rsr | rsr applyWithControl: constraint]

RCSRule.applyWithControl: aConstraint
     "exécute une instance de la règle générique représentée par self."
     self activeCt: aCt.
     controlDemon trigger

RuleDemon.trigger
     (self testInvariants: rule activeCt)
          ifTrue:
               [rule apply: rule activeCt.
               self updateInvariants: rule activeCt]
```

La fonction `testInvariants(aRule,activeConstraint)` rend vrai si toutes les stratégies imposées sur la règle `aRule` autorisent son exécution avec la contrainte

[82] Démons reliant une variable à ses contraintes.

`activeConstraint`. Par ailleurs une règle peut être contrôlée par une conjonction d'invariants de contrôle, ceux-ci étant alors testés dans un ordre statique relatif à la priorité des stratégies. La question de la compatibilité est à la charge de l'utilisateur ; l'ordre de priorité par défaut est l'ordre d'apparition. Bien que primitif, ce langage permet l'expression de stratégies intuitives. Cependant le modèle de coopération gagne à être complété en tenant compte du coût des manipulations symboliques et des méthodes de cohérence, de l'influence des transformations symboliques sur celles-ci et des erreurs d'arrondis sur les valeurs réelles.

2.4 Une vision *framework* du système CSP+RCS+méta-RCS

D'après Ralph Johnson[83], un *framework* est une conception réutilisable d'une partie ou de tout un système. Un *framework* réalise un objectif qui peut être abstrait ; il est implanté par un ensemble de classes abstraites et la description précise de la manière dont leurs instances communiquent pour réaliser cet objectif[84].

Dans ce travail nous avons proposé une typologie du raisonnement symbolique et une bibliothèque d'unités de calcul déductives suivant cette typologie. De plus nous avons construit une architecture pour la construction de stratégies d'exécution de ces solveurs symboliques et proposons un ensemble de stratégies primitives. Ces deux modèles se greffent au dessus d'un *framework* de satisfaction de contraintes. Le système total peut être vu comme un *framework* pour la résolution de problèmes combinatoires par collaboration entre algorithme de cohérence et raisonnement symbolique. La réutilisation du modèle de contrôle Méta-RCS est facilitée par la mise en évidence d'événements de base concernant le changement d'état des solveurs et survenant en cours de résolution.

Toujours d'après Ralph Johnson, un *framework* doit être facile à apprendre pour être rapidement utilisable, par conséquent il doit être accompagné d'une documentation décrivant ses caractéristiques et comment l'utiliser. Autrement dit par J.-P. Briot (LIP6), "Un *framework* est une structure avec des trous et des contraintes sur les trous". Il ne peut être conçu sans un mode d'emploi décrivant la bonne manière de combler ces trous. Cette section décrit 1) les fonctionnalités d'une règle de RCS, et 2) comment réutiliser la bibliothèque de solveurs symboliques et de stratégies, l'étendre selon la typologie proposée au chapitre III.

2.4.1 Création d'une nouvelle règle de propagation ou de réécriture

Toutes les définitions de règle sont stockées dans la catégorie `RCS-Rules`. Deux règles unaires particulières, la réécriture d'instantiation et la simplification possèdent une procédure de création particulière, car la classe de la contrainte doit être étendue avec les connaissances spécifiques nécessaires au fonctionnement de la règle. Dans le cas général, le mode d'emploi est le suivant :

- Dans la catégorie `Ct-RCSRules`, sous-classer `ComposeRule` ou `RewriteRule`.
- Si la contrainte active doit être supprimée après exécution, positionner `mustRemoveActiveCt`.

La règle est définie en indiquant

1. ce qu'elle fait : l'action et les conditions,

[83] http://st-www.cs.uiuc.edu/users/johnson/frameworks.html

[84] "A framework is a set of classes that embodies an abstract design for solutions to a family of related problems" [Johnson & Foote 1988].

2. comment la relier aux contraintes : la sélection des classes de contraintes structurellement candidates (*ruleClasses*).
3. comment la déclencher : les conditions que nous voulons que le problème vérifie, au moment où la règle s'applique (*setExecutionControl*).

Dans la première phase, chaque classe redéfinit deux méthodes au minimum, une des deux méthodes publiques dédiées au test des conditions (*findPremisses* ou *makeConditionCSP*) d'une part, et l'action de la règle sur chaque n-uplet valide (*execute(tuple)*).

La seconde phase n'est *a priori* pas redéfinissable car le type de lien est automatiquement fixé par le type de la règle et géré par la super-classe dans la méthode *postConstraintDemon*.

De plus chaque règle est reliée à un ensemble d'invariants de contrôle visant à la faire collaborer avec les autres règles. Pour modifier la stratégie de collaboration sur une règle (troisième phase), il suffit d'éditer la méthode *setExecutionControl* de manière à ce qu'elle crée les invariants désirés (protocole *controlling-decisions*). L'invariant de contrôle est responsable des pré-conditions et post-actions de déclenchement.

La méthode générale de sélection des contraintes est *findPremisses,* une fonction renvoyant une collection de n-uplets valides d'associations (prémisses, contraintes). Dans le cas où les conditions sont complexes et difficiles à exprimer algorithmiquement, il est possible de les exprimer dans le langage à objets hôte sous forme de contraintes globales, dans la méthode *makeConditionCSP.* Si l'attribut *conditionParCSP* est positionné à VRAI, la sélection des prémisses renvoie les solutions du CSP. L'action de la règle sur chaque prémisse est implantée dans la procédure *execute(tuple)* ; chaque n-uplet renvoyé par *findPrémisses* ou par *makeConditionCSP* est mis en argument de la méthode d'exécution *execute(tuple*). Le comportement par défaut d'une règle est implanté dans la méthode *apply(aConstraint)* au niveau des classes abstraites ; il inclut la substitution des prémisses compatibles avec *aConstraint* et l'exécution de l'action.

Ajouter une réécriture d'instantiation

Le schéma de la règle d'instantiation s'applique à tout type de contraintes et renvoie une instance de la même classe de contrainte que la contrainte source, sans les variables valuées. Il délègue la transformation proprement dite qui supprime les variables instantiées, à la classe de la contrainte. Au niveau des contraintes, la méthode *computeNewConstraint* en est responsable. Celle-ci a été définie pour toutes les contraintes de base du *framework* de CSP. Lors de la création d'une classe de contraintes, il faut donc :

- implémenter la fonction booléenne `mustBeRewritten` qui renvoie `VRAI` ;
- implémenter la méthode `rewriteIfValue(aVar)` qui renvoie la nouvelle contrainte simplifiée quand `aVar` vient d'être instantiée.

Ajouter une règle de simplification

Le schéma de la règle de simplification d'une contrainte délègue son action à la contrainte. Celle-ci par défaut renvoie la contrainte elle-même. Elle est redéfinie dans chaque classe des contraintes arithmétiques de manière à déclencher à la volée des règles de réécriture unaires. Celles-ci fonctionnent par effet de bord en modifiant les attributs de la contrainte ou en la remplaçant par une autre. Lors de la création d'une nouvelle classe de contrainte, il est préférable, dans la mesure du possible, d'exécuter la méthode responsable de la simplification avant la création complète de la contrainte de manière à éviter la création d'objets intermédiaires. Lors de la création d'une classe de contraintes, il faut donc :

- implémenter la méthode `normalizeExpression` qui applique les règles de réécriture unaire mettant la contrainte sous une forme régulière.
- exécuter `normalizeExpression` lors de la création de la contrainte[85].

Le déclenchement d'une règle

Toutes les règles associées à une contrainte active sont exécutées automatiquement. Mais il est possible de spécifier les règles que l'on veut voir se déclencher en éditant la méthode `ruleClasses`. `ruleClasses` renvoie la liste des classes de règles reliées aux contraintes de cette classe. Ce lien est effectuée lors de la création de la contrainte.

Options du solveur

La réécriture n'a lieu que si le solveur est en position `RéécritureAuthorized`. De même la composition n'a lieu que si le solveur est en position `ComposeAuthorized`.

2.4.2 Création d'une classe d'invariant de contrôle

Il suffit de sous-classer la classe `InvariantControl` dans la catégorie RCS-`Control`. Un invariant de contrôle possède au minimum une procédure de test et une procédure de mise à jour ; cette dernière maintient des informations globales sur la résolution par exemple, ou locales à un ensemble de règles de RCS. La classe doit donc implanter deux méthodes publiques, `testRule(r,c)` et `updateAfterApplying(r,c)` ; la première renvoie l'évaluation du test d'autorisation de déclenchement de la règle, tandis que la seconde implante la réaction du contrôle après application des solveurs (mise à jour des états entre autres). Si l'invariant réagit à d'autres changements de statut, les méthodes correspondantes doivent être implantées. Ces méthodes sont spécifiées aux chapitres III-4 et IV-3.

2.4.3 Imposer une stratégie *via* un invariant de contrôle

Pour imposer un contrôle sur une règle, il suffit d'éditer la méthode publique `setExecutionControl` au niveau de la classe de la règle. La déclaration d'un contrôle se fait en instantiant la classe d'invariant correspondante *via* le message `on(aRule)`. Pour cumuler des contrôles, la seule manière implantée dans le système consiste à les déclarer en séquence dans la méthode `setExecutionControl`, par exemple :

```
LinearCompositionRule.setExecutionControl
     (ItComposition on: self).
     (ItStopIfUseless on: self) nbFailure:5.
```

3 Conclusion

Dans ce chapitre nous avons vérifié expérimentalement que les compositions de contraintes définies au chapitre III aident le solveur CSP, en améliorant les algorithmes de cohérence utilisés, ce qui permet de détecter une incohérence plus tôt ou d'éviter des choix infructueux. Durant la résolution, la taille de l'ensemble des contraintes est augmentée par

[85] En Smalltalk c'est possible au niveau de la méta-classe.

les règles de propagation et diminuée par les règles de réécriture. Filtrage de contrainte et reécriture de contraintes s'alternent, permettant d'associer une augmentation du nombre de contraintes à la diminution de la taille des domaines qu'elles entraînent éventuellement.

Ces expériences ont servi de support à la mesure de l'utilité de RCS en fonction du délai entre la déduction de contraintes et l'exploitation de ces contraintes par le solveur CSP pour une réduction effective des domaines. De cette étude, nous avons déduit des critères globaux pour contrôler dynamiquement la composition de contraintes et adapter les stratégies au cours de la résolution. Ces critères généraux s'avèrent coûteux à calculer, nous avons donc prédéfini des stratégies utilisant des critères plus simples.

La seconde section présente l'implémentation de RCS et du contrôle de RCS que nous appelons Méta-RCS en utilisant la programmation par objets. Le système global CSP+RCS+Méta-RCS étend le mecanisme de démons de BackTalk et fait ainsi coopérer trois systèmes sur trois niveaux. Grâce au mécanisme d'héritage, la hiérarchie de classes de stratégies de Méta-RCS constitue une première approche d'une typologie du contrôle évolutif.

Conclusion et perspectives

Les techniques de satisfaction de contraintes fondées sur le mécanisme de filtrage mettent en œuvre des algorithmes efficaces pour résoudre de nombreux problèmes combinatoires ou d'optimisation. Cependant l'efficacité de ces techniques dépend entièrement de l'existence de méthodes de filtrage adaptées à la forme des contraintes au moment de leur spécification. Les deux limites principales de ces approches sont alors 1) lorsque les méthodes de filtrage ne réduisent pas suffisamment les domaines, ces techniques se ramènent à des procédures d'*Enumération-Test* qui sont inefficaces sur de gros problèmes ; 2) les contraintes gardent la même forme tout au long de la résolution, empêchant ainsi l'utilisation de méthodes différentes éventuellement mieux adaptées à l'état courant de la résolution.

Nous avons proposé de répondre à ces deux problèmes en augmentant les mécanismes de CSP classiques par une manipulation symbolique des contraintes, fondée sur la réécriture. Plus précisément nous exprimons sous forme de règles de réécriture de contraintes symboliques, des connaissances intrinsèquement liées aux lois du domaine d'application, comme un axiome du groupe des entiers, un théorème de géométrie, une implication de relations temporelles d'Allen, une loi d'interaction entre deux agents d'un monde virtuel, une propriété d'homogénéité sur des suites de plans de séquences audiovisuelles,.… Etant donné un solveur de contraintes en domaine fini, dans certaines situations pouvant survenir en cours de résolution, la transformation résultant de l'application de ces règles a un effet favorable sur la résolution. Ces raisonnements symboliques permettent à la fois de provoquer des filtrages supplémentaires en produisant des contraintes redondantes et d'exploiter les modifications potentielles de la forme des contraintes au cours de la résolution. Nous avons montré au chapitre II et IV comment certains problèmes classiques de CSP à domaines finis peuvent être ainsi résolus plus directement par notre extension.

Cependant cette amélioration de la qualité de la résolution n'est pas systématique ; de plus le processus de réécriture doit être contrôlé en cours de résolution, pour éviter le risque d'une explosion combinatoire de la taille du problème et donc de la complexité du système de contraintes. Nous avons donc construit RCS, un système de réécriture de contraintes symboliques à base de règles multi-prémisses, dont la partie condition et l'action sont observables et manipulables par une application externe. Nous avons en outre introduit une *représentation réflexive des règles* de réécriture qui permet de spécifier des stratégies pour *contrôler dynamiquement* le déclenchement de ces règles. Notre langage de définition de *stratégies déclaratives* exploite la particularité de notre langage de règles de réécriture pour l'évaluation du résultat du déclenchement d'une règle, le blocage de règles coûteuses en fonction de propriétés topologiques du problème (par exemple le nombre de contraintes d'un certain type, présentes dans le problème), de propriétés du solveur de CSP (réduction de domaines issue d'une contrainte, nombre de choix) ou de propriétés de RCS (historique des déductions, effet des déductions sur le filtrage du problème).

Notre modèle de stratégies répond à une volonté de construction par affinage de stratégies existantes, lié à un besoin de flexibilité d'utilisation et motivé par l'aspect dynamique de la combinaison des solveurs symboliques avec un solveur CSP. Par ailleurs la collaboration de RCS avec un solveur CSP pose un problème d'ordonnancement : il faut décider dynamiquement quand déclencher les règles de RCS. Or les stratégies de contrôle ne peuvent se déterminer que dynamiquement, puisqu'elles portent sur l'état courant du système. Nous avons donc proposé une représentation uniforme des stratégies de manière à pouvoir spécifier et combiner plusieurs stratégies au cours de la résolution.

Parallèlement, ce mémoire a abordé deux questions : comment et quand manipuler les contraintes symboliquement ? Nous avons défini une typologie des transformations utiles d'un ensemble de contraintes. Notamment Nous avons défini une typologie de transformations algégriques. La règle générale de composition de contraintes fonctionnelles (*PerformCtRule*) a été particularisée dans le cas de contraintes d'égalités et d'inégalités linéaires (chapitre III-3.3) : la règle *Gauss-SimplexeRule* spécifie une revisite de l'algorithme de Gauss-Simplexe sur les systèmes carrés en utilisant la réécriture. La règle *linlin* est plus simple et plus générale : *linlin* compose une contrainte donnée avec une contrainte du problème et produit une nouvelle contrainte redondante dont les caractéristiques sont configurables. *Linlin* réalise ainsi une étape de l'algorithme de Gauss et permet par déclenchements successifs de traiter un système de contraintes linéaires non carré et de taille non fixe. Nous avons commencé une étude théorique de la combinaison de contraintes linéaires réalisée par linlin et nlinRec (chapitre III-3.3). L'intérêt de trois stratégies dynamiques a été particulièrement mis en évidence sur un large panel de problèmes arithmétiques à domaines d'entiers. La stratégie *StopIfUseless* exploite l'évaluation des résultats successifs et évite ainsi les applications de RCS lorsqu'il n'a pas été utile pendant un certain nombre de cycles. La stratégie *ExecuteAfterRuleApplication* retarde l'application d'une règle jusqu'au moment où d'autres règles ont échoué un certain nombre de fois. Cette stratégie est impliquée dans la collaboration entre *linlin* et les règles plus coûteuses comme *Gauss-Simplexe* et *nlinRec*. Le langage que nous avons développé permet donc de spécifier à la fois des situations où RCS est (ou a été) inutile et des situations où RCS est susceptible d'être utile. Enfin l'implémentation de RCS et Méta-RCS met en œuvre le *pattern* de conception *Event Notification* pour la coopération réactive entre un système de contrôle et deux solveurs.

Nous avons de plus développé un outil graphique au dessus de BackTalk+RCS+Méta-RCS, outil permettant de configurer les heuristiques de choix et stratégies à appliquer sur les règles de RCS. Son utilisation a été bien utile pour déterminer l'ensemble des stratégies le mieux adapté pour le problème des carrés magiques.

Contraintes objets versus contraintes arithmétiques

Le langage de contraintes est construit par extension d'un langage de programmation par objets (Smalltalk). Le langage permet d'exprimer des contraintes génériques fonctionnelles et prédicatives sur tout domaine fini. Ainsi, un langage uniforme est utilisé pour représenter des contraintes arithmétiques et des contraintes sur des variables à domaine d'objets. Le langage possède des primitives spécialisées pour les expressions arithmétiques, construites au-dessus du langage de base, dans le but de bénéficier d'un langage plus naturel que le langage de contraintes général. Notre travail s'articule autour de ces deux aspects du langage de contraintes de BackTalk. D'une part RCS est défini d'une part par un ensemble de règles transformant et déduisant des contraintes génériques sur des objets quelconques. Ces règles sont destinées à exprimer les lois du domaine d'application, par exemple le parallélisme des cotées d'un carré, le théorème de Pythagore donnant la valeur de l'hypoténuse d'un triangle rectangle,... D'autre part, RCS a été particularisé dans le cas des contraintes arithmétiques. La représentation particulière de ces dernières permet d'exploiter facilement les axiomes et propriétés du groupe des relations arithmétiques et logiques (distributivité, développement d'un carré parfait, substitution dans une combinaison linéaire).

Afin de motiver notre travail nous avons étudié des problèmes classiques de CSP, sur des domaines d'entiers. Nous avons fait ce choix afin de pouvoir comparer les résultats obtenus en combinant CSP et RCS avec les résultats existant et ainsi constater l'influence positive ou négative de la réécriture contrôlée de contraintes symboliques.

Validation

Notre validation de RCS et de notre langage de stratégies, Méta-RCS, porte sur deux aspects principaux (C.F. chapitre IV) : 1) l'application de RCS et du langage de stratégies sur des problèmes de CSP à objets quelconques et l'étude du comportement du système global sur des domaines particuliers d'objets (chapitre IV-1.3) ; 2) la stabilité système global durant la résolution d'une certaine catégorie de problèmes que nous avons identifiée (C.F. chapitre IV-1.1.5) ; et la qualité des stratégies que nous avons définies au moyen de critères expérimentaux (C.F. chapitre IV-1.1.3 et 1.1.4).

Dans la première optique, nous avons choisi un domaine d'application particulier, relativement bien formalisé et dans lequel existe un grand nombre de propriétés structurelles. Nous avons spécifié ces propriétés dans RCS et validé leur utilisation pour la preuve de théorèmes. Dans cette application les stratégies jouent un faible rôle, limité à la sélection des contraintes instantiant les prémisses des règles de réécriture. D'autre part, afin de trouver des invariants sur le comportement global du système CSP-RCS-Méta-RCS, nous avons étudié des problèmes aléatoires issus de la même catégorie que les problèmes initialement étudiés, c'est à dire des cryptogrammes entiers. L'uniformité du rôle et du type des contraintes et des variables permet de trouver des critères globaux indépendants de la structure des contraintes et des stratégies générales portant sur l'évolution du problème quel que soit le type de valeurs des domaines.

Plus précisément, nous avons validé l'intégration de RCS, ainsi que le contrôle de RCS, avec le *framework* de satisfaction de contraintes BackTalk. Notre étude expérimentale a porté d'une part sur des séries de problèmes aléatoires homogènes, et d'autre part sur le problème des carrés magiques de taille 4, 5 et 6. Cette étude a permis de spécifier des critères caractérisant les cas où le raisonnement symbolique s'applique avec efficacité. Nous avons enfin mené plusieurs expérimentations sur des cryptogrammes arithmétiques aléatoirement afin de valider la combinaison de CSP et RCS, étudié et montré l'influence de stratégies générales sur la résolution des problèmes difficiles de carrés magiques (C.F. chapitre IV-1.2) et enfin validé notre langage de réécriture de contraintes symboliques sur des problèmes structurés pour la preuve de théorèmes géométriques (chapitre IV-1.3).

Perspectives

De cette expérience nous retirons deux constats : 1) La caractérisation des cas où RCS améliore la résolution en temps et en qualité est une tâche délicate. Nous avons simplement ébauché cette étude mais celle-ci devrait être approfondie en particulier avec des contraintes complexes comme celle que l'on trouve en ordonnancement ou dans les domaines structurés; 2) La plupart des effets négatifs de RCS sont dus au fait que les techniques utilisées ne permettent pas de déduire les contraintes redondantes pertinentes ; par exemple pour remplacer la composition binaire, on pourrait étudier le comportement d'une composition de trois contraintes en imposant que deux contraintes engagées dans une composition ternaire ne peuvent plus se composer deux à deux. Il serait intéressant d'étudier aussi la composition de plus de deux contraintes, étude difficile car autant les gains obtenus peuvent être importants, autant l'explosion combinatoire due au nombre de compositions possibles est dangereuse. De plus la notion d'utilité de RCS dépend des objectifs poursuivis par le sytème. Cette notion a été définie dans cette thèse par rapport aux techniques de recherche par backtracking et réécriture ; l'approfondissement de cette définition ouvrirait une perspective d'application de RCS à d'autres techniques de recherche (heuristiques, génétiques,…).

Dans de nombreux cas, la recherche des solutions d'un système de contraintes ne constitue qu'un des aspects de la résolution d'un problème. En effet l'utilisateur souhaite souvent obtenir une solution "optimale", cet optimum étant exprimé comme le maximum ou le minimum d'une fonction de coût. Or l'optimisation sous contraintes fait souvent appel à un pré-traitement du problème afin d'améliorer les méthodes d'optimisation, par exemple en introduisant des contraintes redondantes [Caseau et al 1993]. Dans le cas du problème de visualisation optimale d'un ensemble de contraintes [Goumairi & Tessier 1998] fait appel à une étape préalable de simplification afin de traiter un système équivalent plus lisible. Les techniques de réécriture de contraintes symboliques sous contrôle déclaratif, que nous proposons, peuvent en particulier servir à ce type de mise en forme avant résolution.

Le langage de stratégies peut s'appliquer pour le contrôle de système dynamique, comme par exemple les systèmes d'optimisation incrémentale de problème dynamique, tels l'ordonnancement dynamique, la construction d'emploi du temps. Un ordonnanceur dynamique met en œuvre des méthodes heuristiques de recherche locale, couplées à des algorithmes de filtrage de contraintes. Or la modélisation du problème évolue au cours du temps lorsqu'un événement change les données du système (tâche obsolète, ressource bloquée pour un temps supplémentaire,...). La construction du nouveau problème constitue le but de l'analyse des événements et du contrôle des contraintes (ajout, retrait, transformation) du problème courant. Nous voyons une utilisation possible du langage de stratégies Méta-RCS au niveau du contrôle et notamment le contrôle déclaratif [Tessier 1996] qui offre des avantages d'expressivité et de manipulation : les stratégies sont sous forme de contraintes globales sur les contraintes du problème, elles peuvent être utilisées comme briques de base pour faire du diagnostic déclaratif.

Nous avons vu que RCS est particulièrement efficace pour la détection d'absence de solution. En particulier, RCS aider à prouver plus efficacement l'optimalité d'une solution, et peut alors avoir un intérêt au sein de systèmes d'optimisation dynamique. En effet ces derniers sont souvent confrontés à des problèmes combinatoires sur-contraints auxquels ils doivent donner rapidement une «bonne » réponse.

Par ailleurs RCS permet de traiter un ensemble de contraintes globalement par réécriture suivi d'un filtrage de chaque contrainte ; de même la cohérence de chemin est un moyen de filtrer plusieurs contraintes reliées entre elles deux à deux, en ajoutant systématiquement des contraintes redondantes. Nous avons au chapitre II ébauché une comparaison entre l'effet de RCS et celui de la cohérence de chemin. La poursuite de cette étude permettrait d'élaborer un compromis entre la complétude de la cohérence de chemin et l'effet contrôlé de RCS.

Enfin le contrôle dynamique à partir de connaissances observées s'applique naturellement au domaine du diagnostic sur le déroulement des algorithmes de filtrage lors de la propagation de contraintes. Par exemple il peut être utilisé pour sélectionner l'algorithme le plus adapté en fonction des propriétés des contraintes.

Bibliographie

[Abdennadher 1997] S. Abdennadher. Operational semantics and confluence of Constraint Handling Rules. *CP'97, Hagenberg, Austria,* LNCS, Springer-Verlag, vol. 1330, pp. 252-266, 26-30 octobre 1997.

[Alefeld & Herzberger 1983] G. Alefeld & J. Herzberger. Introduction to Interval Computations. Academic Press, 1983.

[Aoki 1994] A. Aoki. Object-Oriented Analysis and Design Techniques. Software Research Center. Tokyo, Japan, 1994.

[Asirelli et al. 1996] P. Asirelli, P. Inverardi and G. Plagenza. Integrity Constraints as Views in Deductive Databases. *6th International Workshop on Foundations of Models and Languages for Data and Objects, Schloss Dagsthul, Germany,* electronic proceedings (http://wwwiti:cs:uni-magdeburg.de/~conrad/IDB96/Proceedings.html, pp. 133-140, September 1996.

[Bachmair 1991] L. Bachmair. Canonical Equational Proofs. Progress in Theoretical Computer Science. Birkhäuser, Romual V. Book, University of California, 1991.

[Bakhouch 1994] M. Bakhouch. Manipulation d'algorithmes numériques. Approches transformationnelles et modélisation par objets. Ph.D., LAFORIA, Univeristé PARIS VI, Paris, 1994.

[Benhamou 1996] F. Benhamou. Heterogeneous Constraint Solving. *Proceedings of the 5th ALP'96 (Algebraic and Logic Programming), Aachen, Germany,* Springer Verlag, LNCS, vol. 1139, pp. 62-76, September, 25-27 1996.

[Benhamou & Granvilliers 1996] F. Benhamou & L. Granvilliers. Combining Local Consistency, Symbolic Rewriting and Interval Methods. *Proceedings of Artificial Intelligence and Symbolic Mathematical Computation, International Conference (AISMC'96), Steyr, Austria,* LNCS, Springer-Verlag, vol. 1138, pp. 144-159, September 23-25 1996.

[Benhamou & Older 1997] F. Benhamou & William J. Older. Applying Interval Arithmetic to Real, Integer and Boolean Constraints. *The Journal of Logic Programming,* Elsevier Science eds, vol. 32(1), pp. 1-24, 1997.

[Benhamou & Touraïvane 1995] F. Benhamou & Touraïvane. PrologIV : langage et algorithmes. *Actes des IVèmes Journées Francophones de Programmation en Logique (JFPL), Dijon, France,* pp. 51-65, 1995.

[Bessière 1994] Ch. Bessière. Arc-Consistency and Arc-Consistency Again. *Artificial Intelligence*, vol. 65, pp. 179-190, 1994.

[Bessière et al. 1995] C. Bessière, E.C. Freuder et J.-Ch. Régin. Using Inferences to reduce Arc-Consistency Computation. *Proceedings of IJCAI'95, Montréal,* vol. 1, pp. 592-598, August 1995.

[Bessière & Régin 1997] C. Bessière & J.-Ch. Régin. Arc-consistency for General Constraint Networks : Preliminary Results. *Proceedings of IJCAI'97, Nagoya, Japan,* vol. 1, pp. 398-404, août 1997.

[Bessière & Régin 1998] C. Bessière & J.C. Régin. Local Consistency on Conjunctions of Constraints. *Workshop on non binary constraints, ECAI'98, Brighton, UK,* vol. 1, pp. 53-60, août 1998.

[Borning 1981] A. Borning. The Programming Language Aspects of ThingLab, a constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, vol. 3(4), pp. 353-387, October 1981.

[Borning et al. 1992] A Borning, B. Freeman-Benson et M. Wilson. Constraint hierarchies. *LISP and Symbolic Computation: An International Journal*, vol. 5(3), pp. 223-270, 1992.

[Borning & Freeman-Benson 1995] A. Borning & B. Freeman-Benson. The OTI Constraint Solver: A Constraint Library for Constructing Interactive Graphical User Interfaces. *CP'95, Cassis,* Springer-Verlag, Lecture Notes in Computer Science n. 976. U. Montanari & F. Rosi Eds, pp. 624-628, 1995.

[Borovansky et al. 1997a] P. Borovansky, C. Kirchner et H. Kirchner. Controlling Rewriting by Rewriting. *Theoretical Computer Science (Elsevier Science)*, vol. 5 (1997), pp. 1997.

[Borovansky et al. 1997b] P. Borovansky, C. Kirchner, H. Kirchner, P.-E. Moreau et M. Vittek. ELAN: A logical *framework* based on computational systems. *Theoretical Computer Science (Elsevier Science)*, vol. 5(1997), pp. 1997.

[Borovansky et al. 1997c] P. Borovansky, C. Kirchner, H. Kirchner, P.-E. Moreau et M. Vittek. ELAN: User Manual version 2.00 beta. copyright INRIA Lorraine & CRIN, report Villers-lès-Nancy, 9 mai 1997.

[Brim et al. 1996] L. Brim, D. Gilbert, J.M. Jacquet et M Kretinsky. A Process Algebra for Synchronous Concurrent Constraint Programming. *Proceedings of ALP'96, Aachen, Germany,* Springer Verlag, LNCS, vol. 1139, pp. 165-178, September, 25-27 1996.

[Carrive 2000] J. Carrive. Indexation de documents audiovisuels. Thèse de doctorat, Université Paris 6 et Institut National de l'Audiovisuel, septembre 2000.

[Caseau 1989] Y. Caseau. A formal system for Producing Demons from Rules. *Proceedings of the first international conference of Deductive and Object-Oriented Databases,* march 1989.

[Caseau 1991a] Y. Caseau. Abstract Interpretation of Constraints on Order-Sorted Domains. *Proceedings of ILPS'91,* 1991.

[Caseau 1991b] Y. Caseau. An Object-Oriented Deductive Language. *Annals of Mathematics and Artificial Intelligence*, vol. 3, pp. 211-258, 1991.

[Caseau et al. 1993] Y. Caseau, P.Y. Guillo and E. Levenez. A Deductive and Object-Oriented Approach to a Complex Scheduling Problem. *Proceedings of DOOD'93,* 1993.

[Caseau & Laburthe 1996a] Y. Caseau & F. Laburthe. CLAIRE: Combining Objects and Rules for Problem Solving. *Proceedings of the JICSLP'96 workshop on Multi-paradigm Logic Programming, Berlin, Germany,* Y.GUO M.T. Chakravarty, T.Ida eds., 1996.

[Caseau & Laburthe 1996b] Yves Caseau & François Laburthe. Introduction to the CLAIRE Programming Language. LIENS, report Paris, 1996.

[Caseau & Perron 1991] Y. Caseau & L. Perron. A Type system for Object-Oriented Database Programming and Querying Languages. *Proceedings of DBPL'91,* 1991.

[Castro 1998] C. Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae, IOS Press*, vol. 34 (3), pp. 263-293, 1998.

[Castro & Kirchner 1998] C. Castro & C. Kirchner. Using Computational Systems as a General *Framework* for Handling CSPs. *Proceedings of the Annual workshop of the Constraints in Computational Logics Working Group, CCL'98, Jerusalem, Israel,* September 1998.

[Castro & Monfroy 1998] C. Castro & E. Monfroy. A Strategy Language for Solving CSPs. *Proceedings of the 3rd Workshop of the Working Group on Constraints of ERCIM'98, Amsterdam, The Nederlands,* Ph. codognet and E. Monfroy K. Apt, eds, CWIAmsterdam, 1998.

[Cazenave 1997] T. Cazenave. Système d'apprentissage par Auto-Observation : Application au jeu de Go. Rapport de recherche LIP6 1997/034, décembre 1997

[Chabrier 1999] A. Chabrier. A Cooperative CP and LP Optimizer Approach for the Pairing Generation Problem. *CP-AI-OR'99, Workshop on Integration of AI and OR techniques in*

Constraint Programming for Combinatorial Optimization Problems, University of Ferrara, Italy, AI*IA & AIRO, 25 - 26 February 1999.

[CHIC-2 1997] CHIC-2. Methodology User Guide On Line. Rapport de recherche, Projet Esprit, report http://www-icparc.doc.ic.ac.uk/chic2/, 1997.

[CHIC-2 1998] CHIC-2. European Chapter on Combinatorial Optimization (ECCO XI), industrial sessions on Esprit Projects, Copenhage, Danemark,May 1998.

[Clavel et al. 1996] M. Clavel, S. Eker, P. Lincoln et J. Meseguer. Principles of Maude. *First International Workshop on Rewriting Logic, Asilomar, USA,* Electronic Notes in Theoretical Computer Science, North-Holland, vol. 5, September 1996.

[Codognet 1995] P. Codognet. Programmation logique avec contraintes: une introduction. *Technique et Sciences Informatiques*, vol. 14, pp. 665-692, 1995.

[Codognet & Nardiello 1996] P. Codognet & G. Nardiello. Path-consistency in clp(FD). *Proceedings of the International Conference on Constraints in Computational Logic (CCL'94), Munich, Germany,* J.-P. Jouannaud editor, Springer-Verlag, vol. 845, pp. 201-216, September 1996.

[Cohen 1990] J. Cohen. Constraint Logic Programming Languages. *A.C.M. Communications*, vol. 33(7), July, pp. 52-68, 1990.

[Colmerauer 1990] Alain Colmerauer. An Introduction to Prolog-III. *Communication of the ACM*, vol. 33 (7), juillet, pp. 69-90, 1990.

[Comon et al. 1999] H. Comon, M. Dincbas, J.-P. Jouannaud et C. Kirchner. A Methodological View of Constraint Solving. *Constraints Journal*, vol 4(4), pp.337-361, décembre 1999.

[Cornes et al. 1995] C. Cornes, J. Courant, J-C. Filliatre, G. Huet, P. Manoury et al. The Coq Proof Assistant Reference Manual, version 5.10. INRIA, report 0177, 1995.

[Cousot & Cousot 1977] P. Cousot & R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Constructions or Approximation of Fixpoints. *Proceedings of 4th ACM Symposium of Principles of Programming Languages*, 1997.

[David 1992] Alain David. Utilisation de propagation de contraintes pour un problème de robot. *Journée Enseignement de la Recherche Opérationnelle de l'AFCET, Nancy, France,* AFCET, juin 1992.

[Davis 1988] E. Davis. Constraint Propagation with Interval Labels. *Artificial Intelligence*, vol. 32, pp. 281-332, 1988.

[De Bono 1994] E. De Bono. De Bono's Thinking Course. Revised Edition. USA, 1994.

[Dershowitz & Jouannaud 1990] N. Dershowitz & J.-P. Jouannaud. Rewrite Systems. chapter 6, in Handbook of theoretical computer science, vol B, elsevier Science Publishers, 1990.

[Deville & Van Hentenryck 1991] Y. Deville & P. Van Hentenryck. An Efficient Arc-Consistency Algorithm for a Class of CSP Problems. *Proceedings of IJCAI'91, Chambéry,* pp. 125-130, 1991.

[Diaz 1998] D. Diaz. CALYPSO PROLOG version 1.0 beta 6. User's and Reference Manual. GNU freeware. INRIA Rocquencourt, 1998.

[Dincbas et al. 1990] M. Dincbas, H. Simonis et P. Van Hentenryck. Solving Large Combinatoral Problems in Logic Programming, *Journal of Logic Programming*, vol. 7(1), 1990.

[Foss 1987] C.L. Foss. The Acquisition of Error Management Skills, *Proceedings of Cognitiva'87*, Paris, France, pp. 117-122, 1987.

[Freuder 1997] E.C. Freuder. In Pursuit of the Holy Grail, *Constraints: An International Journal*, vol 2(1), ed. E. Freuder, Kluwer Academic Publishers, avril 1997.

[Freuder & Hubbe 1995] Eugene C. Freuder & Paul D. Hubbe. Extracting Constraint Satisfaction Subproblems. *IJCAI-95,* pp. 548-555, 1995.

[Frühwirth 1994] T. Frühwirth. Temporal Reasoning with Constraint Handling Rules. ECRC, report 94-05., février 1994.

[Frühwirth 1998] Th. Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming*, vol. 37(1-3) (October), pp. 95-138, 1998.

[Frühwirth et al. 1996] Th. Frühwirth, S. Abdennadher et H. Meuss. On Confluence of Constraint Handling Rules. *Conference on Constraint Programming (CP'96), Cambridge, USA,* Springer-Verlag, LNCS, vol. 1118, 1996.

[Frühwirth & Brisset 1998] Th. Frühwirth & P. Brisset. Optimal Placement of Base Stations in Wireless Indoor Telecommunication. *Fourth International Conference on Principles and Practice of Constraint Programming (CP98), Pisa, Italy,* October 1998.

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson et J. Vlissides. Design Patterns - Elements of reusable Object-Oriented Software. Addison-Wesley, 1995.

[Gaschnig 1977] J. Gaschnig. A General Backtrack Algorithm that Eliminates most Redundant Tests, presented at IJCAI'77, Cambridge MA, 1977.

[Gouachi & Plateau 1999] I. Gouachi & G. Plateau. LIRESNE: A system for solving integer satisfaction constraint. *CP-AI-OR'99, Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems, Jointly Organized by DEIS and University of Ferrara, Italy,* AI*IA &AIRO, February,25 - 26 1999.

[Granvilliers 1998a] L. Granvilliers. Consistances locales et transformations symboliques de contraintes d'intervalles. Thèse de doctorat en Informatique, Université d'Orléans, 1998.

[Granvilliers 1998b] L. Granvilliers. On the Combination of Box-consistency and Hull-consistency. *Workshop on Non-binary Constraints, ECAI'98, Brighton, UK,* pp. 19-26, août 1998.

[Granvilliers 1999] L. Granvilliers. Symbolic Transformations and Box-consistency of Real Constraints. *Techniques et Sciences Informatiques,* 1999.

[Granvilliers et al. 1998] L. Granvilliers, G. Hains, Q. Miller et N. Romero. A system for the high-level parallelization and cooperation of constraint solvers. *Proceedings of the conference on Parallel and Distributed Computing and Systems (PDCS), Las Vegas, USA,* Akl and Li editors Pan, IASTED/ACTA Press, pp. 596-601, 1998.

[Güsgen & Hertzberg 1988] H.-W. Güsgen & J. Hertzberg. Some Fundamental Properties of Local constraint Propagation. *Artificial Intelligence*, vol. 36(2), pp. 237-248, 1988.

[Haralick & Elliott 1980] R. Haralick & G. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial intelligence*, vol. 14, pp. 263-313, 1980.

[Hickey et al. 1998] T.J. Hickey, M.H. Van Emden et H. Wu. A Unified *Framework* for Interval Constraints and Interval Arithmetic. *Proceedings of Constraint Programming (CP'98), Pise, Italy,* LNCS, M. Maher & J.F. Puget eds, Springer-Verlag, vol. 1520, pp. 250-264, 26-30 octobre 1998.

[Hofstedt 1998] P. Hofstedt. An Architecture for the Combination of Constraint Solvers. *ERCIM workshop on Constraints, Amsterdam, Netherlands, S*eptember 1998.

[Holzbaur & Frühwirth 1998] Ch. Holzbaur & Th Frühwirth. Compiling Constraint Handling Rules. *Workshop of the CompulogNet arear " Constraint Programming ", ERCIM Working Group on Constraints,* pp. 1-14, September 23-25 1998.

[Hooker et al. 1999] J. Hooker, G. Ottoson, H.-J. Kim et E. Thorsteinsson. On Integrating Constraint Propagation and Linear Programming for Combinatorial Optimization. *Proceedings of CP-AI-OR'99,* AAAI/IAAI Press, January 1999.

[Hoover 1987] Roger Hoover. Incremental Graph Evaluation. PH. D. thesis, Cornell University, Ithaca, 1987.

[Horn 1992] B. Horn. Constraint Patterns As a Basis for Object Oriented Programming. *Proceedings of OOPSLA'92,ACM SIGPLAN Notices,* vol. 27 (10), pp. 218-233, 1992.

[Hyvönen 1992] E. Hyvönen. Constraint Reasoning Based on Interval Arithmetic: the tolerance propagation approach. *Artificial Intelligence*, vol. 58, pp. 71-112, 1992.

[Jaffar & Lassez 1987] J. Jaffar & J.L. Lassez. Constraint Logic Programming. *Proceedings of 4th International conference on Logic Programming, Melbourne,* MIT Press, May 1987.

[Jouannaud & Lescanne 1986] J.P. Jouannaud & P. Lescanne. La Réécriture. in Techniques et Sciences informatiques.edited by North-Holland, vol. B, pp. 433-448, 1986.

[Kirchner 1993] C. Kirchner. Rewriting techniques and applications. *5th International Conference on Rewriting Techniques and Applications, RTA'93,, Montreal, Canada,* LNCS, Sprinwer-Verlqg, vol. 690, june 1993 1993.

[Kirchner et al. 1993] C. Kirchner, H. Kirchner and M. Vittek. Implementing computational systems with constraints. *Proceedings of the first Workshop on Principles and Practise of Constraint Programming, Providence (R.I., USA),* aris Kanellakis, Jean-Louis Lassez, and Vijay Saraswat editors, pp. 166-175, 1993.

[Knuth & Bendix 1970] D.E. Knuth & P.B. Bendix. Simple word problems in universal algebras. in Computational Problems in Abstract Algebra.edited by Pergamon Press. J.Leech, pp. 263-297, 1970.

[Kornmann 1993] S. Kornmann. SADE : un système réflexif de surveillance à base de connaissances . Thèse de l'université Paris 6, février 1993.

[Laburthe 1998] F. Laburthe. Contraintes et Algorithmes en Optimisation Combinatoire. Informatique, Université Paris VII-Denis Diderot, Paris, France, 1998.

[Laburthe & Caseau 1998] F. Laburthe & Y. Caseau. SaLSA: A Specification Language for Search Algorithms. *Proceedings of the conference on Constraints Principle and Practice (CP'98), Pise, Italy,* LNCS, Springer-Verlag, vol. 1520, pp. 310-324, October, 26-30 1998.

[Laburthe et al. 1998] F. Laburthe, P. Savéant, S. De Givry et J. Jourdan. Eclair : A Library of Constraints over Finite Domains. Laboratoire Central de Recherches, Thomson-CSF, report ATS 98-2, domaine de Corbeville, 91404 Orsay, France, 1998.

[Laurière 1976] J.L. Laurière. Un langage et un programme pour énoncer et résoudre des problèmes combinatoires. Ph. D., University Pierre et Marie Curie, Paris, 1976.

[Laurière 1978] Jean-Louis Laurière. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial intelligence*, vol. 10, pp. 29-127, 1978.

[Laurière 1979] J.L. Laurière. Toward efficiency through generality. *Sixth International Joint Conference on Artificial Intelligence, Tokyo, Japan,* vol. 2, pp. 519-521, 1979.

[Laurière 1986] J.L. Laurière. Intelligence Artificielle, Résolution de problèmes par l'homme et la machine. Eyrolles, Paris, 1986.

[Laurière 1996] J.L. Laurière. Propagation de contraintes ou programmation automatique. Laforia Institut Blaise Pascal-CNRS, University Pierre et Marie Curie, report n. 96/19, Paris, juin 1996.

[Le Pape & Baptiste 1997] Cl. Le Pape & Ph. Baptiste. The Claire Schedule documentation. Bouygues scientific department, report St Quentin, France, 1997.

[Lhomme et al. 1998] O. Lhomme, A. Gotlieb et M. Rueher. Dynamic optimization of interval narrowing algorithms. *The Journal of Logic Programming, Elsevier Science eds*, vol. 37(2), pp. 165-183, 1998.

[Liret 1996] A. Liret. Projet AliceTalks. Université de Paris 6, DESS G.L.A. report, Paris, mai 1996.

[Liret 1997] A. Liret. Memento sur AliceTalks, LIP6, pôle IA. 1997.

[Liret et al. 1997] A. Liret, P. Roy et F. Pachet. Combining Formal Reasoning Techniques and CSP. *ERCIM Workshop Constraint Programming and Processing (en conjonction avec CP'97), Linz, Autriche,* octobre,27, 28,29 1997.

[Liret et al. 1998] A. Liret, P. Roy et F. Pachet. Conception par objets d'un système pour combiner raisonnement formel et satisfaction de contraintes. *Journées Francophones des Langages Applicatifs (JFLA'98), Lac de Côme, Italie,* vol. 1, pp. 159-172, February, 2,3 1998.

[LMU 1996] LMU. Constraints Handling Rules Manual. Copyright (C) LMU (Ludwig-Maximilians-University), Munich, Germany, 1996.

[Mackworth & Freuder 1993] A. Mackworth & E.C. Freuder. The Complexity of Constraint Satisfaction revisited. *Artificial Intelligence*, vol. 59, pp. 57-62, 1993.

[Mackworth 1977] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial intelligence*, vol. 8(1), pp. 99-118, 1977.

[Mainguenaud 1996] Michel Mainguenaud. Constraint-based Queries in a Geographical Database for Network Facilities. *Computer, Environment and Urban Systems (Pergamon Press)*, vol. 20(2), pp. 139-151, 1996.

[Marriott & Stuckey 1998] Kim. Marriott & Peter J. Stuckey. Programming with Constraints, An Introduction. The MIT Press, Cambridge, Massachusetts, 1998.

[MathLab 1983] MathLab. MACSYMA Reference Manual. The MathLab Group, Laboratory for Computer Science, report Cambridge, Mass., January 1983.

[Meseguer 2000] J. Meseguer. Rewriting Logic and Maude: Concepts and Applications. *RTA'2000*, Lecture Notes in Computer Science, Springer-Verlag, vol. 1833, pp. 1-26, 2000.

[Michel & Van Hentenryck 1997] L. Michel & P. Van Hentenryck. Localizer: A Modeling Language for Local Search. *Proceedings of the Conference on Constraint Programming (CP'97), Linz, Austria,* LNCS, Springer-Verlag, vol. 1330, pp. 237-251, 1997.

[Mohr & Henderson 1986] R. Mohr & T. Henderson. Arc and Path Consistency revisited. *Artificial Intelligence*, vol. 28, pp. 225-233, 1986.

[Mohr & Masini 1988] R. Mohr & G. Masini. Good Old Discrete Relaxation. *Proceedings of ECAI'88, Munich (Austria),* vol. 1, pp. 651-656, August 1988.

[Monfroy 1996a] E. Monfroy. Collaboration de Solveurs pour la Programmation Logique à Contraintes. Thèse d'université, Université Henri Poincaré-Nancy I, 1996.

[Monfroy 1996b] E. Monfroy. An Environment for Designing/Executing Constraint Solver Collaborations. CRIN, Centre de Recherche en Informatique de Nancy, report 96-R-044, Vandoeuvre-lès-Nancy, February 1996.

[Nareyek 2000] Alexander Nareyek. Alexander Nareyek.AI Planning in a Constraint Programming *Framework. Communication-Based Systems,* Hommel, G. (ed.), Kluwer Academic Publishers, pp.163-178, 2000.

[Pachet & Roy 1995] François Pachet & Pierre Roy. Mixing Constraints and Objects: a Case Study in Automatic Harmonization. *TOOLS Europe, Versailles, France,* Prentice-Hall, pp. 119-126, 1995.

[Paltrinieri 1994] M. Paltrinieri. Integrating Objects with Constraint-Programming Languages. *Proceedings, Object-Oriented Methodologies and Systems,* S.Urban edition, Sprinwer-Verlqg, LNCS, 858, pp. 248-265, 1994.

[Pfeffer 1992] N. Pfeffer. Intégrer le maintien de cohérence dans un système combinant inférences expertes et propagation de contraintes. *International Avignon Conference on Artificial Intelligence, Expert Systems and Natural language, Avignon, France,* Alcatel Alsthom Recherche, vol. 1, pp. 197-208, 1-6 juin 1992.

[Pitrat 1993] J. Pitrat. Penser autrement l'informatique. Hermès, Paris, 1993.

[Pitrat 1995] J. Pitrat. De la machine à l'intelligence. Hermès, Paris, 1995.

[Prosser 1993a] P. Prosser. Domain filtering can degrade intelligent backjumping. *Proceeding of the 13th. International Joint Conference on Artificial Intelligence., Chambery,* pp. 262-267, August 28 - September 3 1993.

[Prosser 1993b] Patrick Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, vol. 9, pp. 268-299, 1993.

[Puget 1994] Jean-Francois Puget. A C++ implementation of CLP. Ilog, report Ilog Technical Report, Paris, 1994.

[Puget 1992] J.F. Puget. Programmation Par Contraintes Orientée Objet. *12th International Conference on Artificial Intelligence, Expert Systems and Natural Language,* Avignon France, pp. 129-138, 1992.

[Puget & Leconte 1995] J.-F. Puget & M. Leconte. Beyond the black box: Constraints as objects. *Proceedings of the International Logic Programming Symposium,* J. Loyd editor, The MIT Press, pp. 513-527, 1995.

[Ramaux 1998] N. Ramaux. Supervision de systèmes dynamiques par reconnaissances de scénarios. Thèse de doctorat, université de Technologie de Compiègne, 1998.

[Régin 1994] J.-Ch. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. *Twelfth National Conference on Artificial Intelligence,* MA Cambridge, AAAI Press / MIT Press, vol. 1, pp. 362-367, 1994.

[Régin 1996] Jean-Charles Régin. Generalized Arc Consistency for Global Cardinality Constraint. *Thirteenth National Conference on Artificial Intelligence, Portland, Oregon,* 1996.

[Régin & Puget 1997] J.-Ch. Régin & J.-F. Puget. A Filtering Algorithm for Global Sequencing Constraints. *Proceedings of the Conference on Constraint Programming (CP'97), Linz, Austria,* LNCS, Springer-Verlag, vol. 1330, pp. 32-46, October 1997.

[Roy 1996] G. Roy. Satisfaction incrémentielle de contraintes par le biais de la réécriture. Thèse de Ph.D., Département d'informatique et de recherche opérationnelle, Université de Montréal, Montréal, 1996.

[Roy 1998] P. Roy. Satisfaction de contraintes et programmation par objets. Thèse de doctorat en Informatique, LIP6, pôle Intelligence artificielle, Université Paris 6, Paris, France, 1998.

[Roy et al. 2000] P. Roy, A. Liret et F. Pachet. A *Framework* for Object-Oriented Cnstraint Satisfaction Problems. *ACM Computing Surveys Symposium on Object-Oriented Application Frameworks,* March 2000.

[Roy et al. 1999] P. Roy, A. Liret et F. Pachet. Constraint Satisfaction Problems *Framework*. in Implementing Application *Framework*s: Object-Oriented *Framework*s at Work.edited by D. Schmidt and R. Johnson M. Fayad. Wiley & sons, vol. 2(17), 1999.

[Roy et al. 1997] P. Roy, J.-F. Perrot et F. Pachet. A *Framework* for Expressing Knowledge About Constraint Satisfaction Problems. *FLAIRS, Daytona Beach (Florida),* Douglas D. Dankel II, Mark Fishman, vol. 1, pp. 47-51, 1997.

[Rueher & Solnon 1997] M. Rueher & C. Solnon. Concurrent Cooperative Solvers over Reals. *Reliable Computing (Kluwer Academidc Publishers)*, vol. 3(3), pp. 325-333, 1997.

[Sabin & Freuder 1996] M. Sabin & E.C. Freuder. Automated Formulation of Constraint Satisfaction Problem. *Proceedings of AAAI'96*, 1996.

[Sabin & Freuder 1998] M. Sabin & E.C. Freuder. Detecting and Resolving Inconsistency and Redundancy in Conditional Constraint Satisfaction Problems. *Proceeding of Constraint Programming (CP'98)*, 1998.

[Sannella 1994] Mickael Sannella. The SkyBlue Constraint Solver and Its Applications. *Proceeding of the 1993 Workshop on Principles and Practice of Constraint Programming,* Saraswat & Van Hentenryck eds., MIT Press, 1994.

[Saraswat 1993] Vijay A. Saraswat. Concurrent Constraint Programming. MIT Press, Cambridge, Massachusetts, 1993.

[Schoenfeld 1985] A.H. Shoenfled. Mathematical Problem Solving, Academic Press, 1985.

[SICS 1998] SICS. SICStus-Prolog User's manual. version 3.0 (release 7.1), Suède, 1998.

[Smith 1995] S.F. Smith. Reactive Scheduling Systems. in Intelligent Scheduling Systems.edited by eds. D.E. Brown & W.T. Scherer. Kluwer Press, 1995.

[Smith et al. 1996] S.F. Smith, O. Lassila et M. Becker. Configurable, Mixed-Initiative Systems for Planning and Scheduling. in Advanced Planning Technology.edited by ed. A. Tate. AAAI Press, Menlo Park, CA, 1996

[Smolka 1995] G. Smolka. The Oz programming model. *Computer Science Today*, vol. 1000 (Lecture Notes in Computer Science), pp. 324-343, 1995.

[Tessier 1996] A. Tessier. Declarative Debugging in Constraint Logic Programming, *Asian Computing Science Conference* , J. Jaffar (réd.), *Lecture Notes in Computer Science*, vol. 1179, Springer-Verlag, pp. 64-73,1996.

[Trombettoni & Neveu 1997] Gilles Trombettoni & Bertrand Neveu. Computational Complexity of Multi-way, Dataflow Constraint Problems. *Proceedings of IJCAI'1997,* pp. 358-363, 1997.

[Tsang 1993] E. Tsang. Foundations of Constraint Satisfaction. Computation in cognitive science. Academic Press limited. Harcourt Brace & Company, Great Britain, 1993.

[Van Beek 1994] Peter. Van Beek. On the Inherent Level of Local Consistency in Constraint Networks. *Twelfth National Conference on Artificial Intelligence,* MA Cambridge, AAAI Press / MIT Press, vol. 1, pp. 368-373, 1994.

[Van Hentenryck 1989] P. Van Hentenryck. Constraint Satisfaction in Logic Programming. Logic Programming. MIT Press, Cambridge,MA, 1989.

[Van Hentenryck & Saraswat 1997] P. Van Hentenryck & V. Saraswat. Constraint Programming: Strategic Directions. *Constraints: An International Journal*, vol 2(1), ed. E. Freuder, Kluwer Academic Publishers, avril 1997.

[Van Hentenryck et al. 1997a] P. Van Hentenryck, L. Michel et Y. Deville. Numerica : a Modeling Language for Global Optimization. MIT Press, Cambridge,MA, 1997.

[Van Hentenryck et al. 1997b] P. Van Hentenryck, L. Michel et Benhamou F. Newton - Constraint Programming over non-linear Constraints. *Science of Computer Programming,* Elsevier Science eds, 1997.

[Vittek 1996] M. Vittek. A Compiler for Nondeterministic Term Rewriting Systems. *RTA'96,* Ganzinger, Harald editor, LNCS, vol. 1103, pp. 154-1668, 1996.

[Wallace et al. 1997] M. Wallace, S. Novello et J. Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. IC-Parc, Imperial College, report Londre, Royaume-Uni, août 1997.

[Wilk 1991] M. Wilk. Equate : An Object-Oriented Constraint Solver. *Proceedings of OOPSLA'91, Phoenix, Arizona, USA,* ACM SIGPLAN Notices, vol. 26 (11), pp. 286-298, November 1991.

[Wolf 1997] A. Wolf. Incremental Adaptation of CHR derivation. *CP'97 Workshop on the Theory and Practice of Dynamic Constraint Satisfaction, Hagenberg, Austria,* 1 novembre 1997.

ANNEXES

1 EXEMPLES DE TRANSFORMATIONS SYMBOLIQUES EN CHR

Composition de deux contraintes linéaires :

```
somme(Vars1,Coefs1) = B1 , somme(Vars2, Coefs2) = B2 ==>
%guard
intersection(Vars1,Vars2,CommonVars), CommonVars estNonVide,
size(CommonVars) < max(size(Vars1), size(Vars2)) |
%calculus
gaussRatio(Coefs1, Coefs2, Result),
%new constraint
somme(Vars1 ∪ Vars2, Coefs1 ⊕(Coefs2 ⊗  Result)) = (B1+(B2*R)).
```

Le prédicat `gaussRatio` calcule le meilleur pivot pour éliminer le plus de variables dans la contrainte résultat.

Normalisation d'une combinaison linéaire :

```
somme(Vars,coefs) = B <=>
%guard
gcd(Coefs, G), divideList(Coefs, G, Result), true |
%new constraint
somme(Vars, Coefs) = (B/G).
```

Le prédicat `divideList` divise les éléments d'une liste par un nombre. La prédicat `gcd` calcule le plus grand commun diviseur des nombres de la liste en argument.

Normalisation d'un carré :

```
X+Y=Z, square(Z) = U <=>
%guard
square(X) = T1, square(Y) = T3, 2*X*Y = T2 |
%new constraint
T1+T2+T3 = Z.
```

Développement d'un produit par distributivité :

```
X+Y = Z, T+U = V, Z*V = W <=>
%guard
X*T = T1, X*U = T2, Y*T = T3, Y*U = T4 |
%new constraint
T1+T2+T3 = Z.
```

Transformation de structure en diminuant l'arité de la contrainte :

```
X*X = R <=>
%guard
%new constraint
square(X) = R.
```

2 COLLABORATION

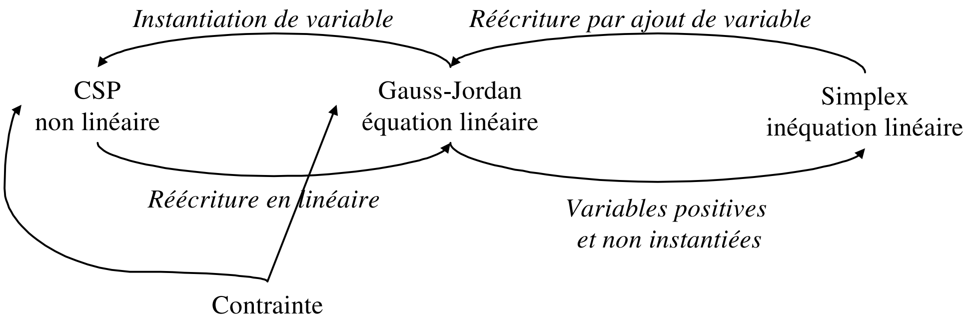


Figure 30 : modèle de collaboration de Prolog III.

Prolog IV remplace le composant CSP par un calcul sur l'arithmétique des intervalles sur les contraintes arithmétiques.

La **méthode de Newton** sert de méthode de filtrage sur des équations réelles du type f(x)=0. Plus précisément c'est une méthode itérative pour approcher les racines d'un polynôme de degré quelconque. L'algorithme considère la fonction F(x) = x – f(x)/f'(x), où f(x)=0 est le polynôme dont on cherche les racines. S'il existe un intervalle I assez petit pour que F soit une contraction (la distance entre deux images est bornée par la distance entre les antécédents), et f'(x) est non nul sur l'intervalle I, le théorème du point fixe dans un espace complet assure que F possède un point fixe l tel que F(l) = l, *i.e.* f(l)=0 ; l'intervalle I est donc une approximation d'une racine de f(x). En réitérant le processus on obtient tous les intervalles contenant une solution au moins de l'équation f(x)=0. La méthode est d'autant plus efficace que les intervalles sont petits, et que le nombre d'itération est grand.

	Algorithme hybride CLAIRE	*Combinaison ELAN, CLAIRE*	*Langage de collaboration de solveurs BALI, [Hofstedt 1998]*
Stratégie	Règle cablée non publique	Règle ou procédure évoluant en cours de résolution, combinable (concaténation, énumération)	Opération contrôlant l'exécution de processus (concurrence, séquence, parallélisme)
Niveau	Paramètre d'algorithme	Base de règles ou système de calcul ayant une base de contraintes en commun	Système de calcul fermé autonome.
Informations disponibles	ex. taille d'un arbre de recherche	Evaluation des applications sur une contrainte, interprétation des résultats	Interprétation des résultats

Figure 31 : Comparaison des principaux systèmes capable de mettre en oeuvre une collaboration de solveurs.

3 ALICETALKS

D'un point de vue objet, Alice possède deux entités capables de traiter des contraintes différentes, et un contrôle qui décide de la structure qui doit traiter chaque contrainte du problème. AliceTalks reprend ce modèle en utilisant les *patterns* de conception [Gamma et al. 1995]. De plus il étend les algorithmes de déduction de contraintes redondantes et rajoute une interface permettant de suivre dynamiquement la trace des étapes de résolution ainsi qu'un module de création d'heuristiques (Figure 32). Le fonctionnement est détaillé dans [Liret 1996] et [Liret 1997].

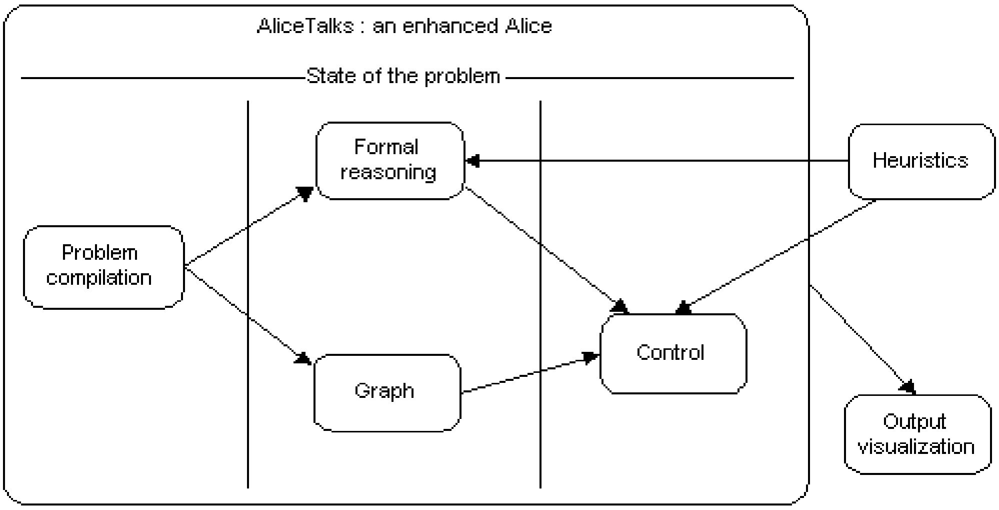


Figure 32: les modules d'AliceTalks[86]

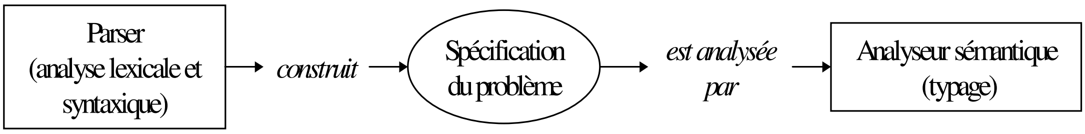


Figure 33: étapes de construction du problème

Un problème se spécifit en quatre parties, la déclaration, le but, les contraintes à respecter et la partie DATA contenant les initialisations des données déclarées si nécessaire.

```
moneyFunctions
    titre
    "SEND + MORE = MONEY. 9567 + 1085 = 10652. "
    commentaires
soit      déclaration des constantes et ensembles (les domaines sont des intervalles)
          constante s, e, n, d, m, o, r, y.
          constante r1, r2, r3 , r4
          constante B.
          Letters = intervalle 1,8
          Carries = intervalle 1,4
          Numbers = intervalle 0,9
          CarriesDomaine = intervalle 0,B.
trouver   déclaration des fonctions à déterminer <nom> : <domaine> → <codomaine>
          fonction injective letter : Letters -> Numbers.
          fonction           carry : Carries -> CarriesDomaine .
verifiant    déclaration des contraintes à satisfaire, utilisant les fonctions et les constantes
    letter(d) + letter(e) = letter(y) +10 * carry(r1).
    carry(r1) + letter(n) + letter® = letter(e) + 10 * carry(r2).
carry(r2) + letter(e) + letter(o) = letter(n) + 10 * carry(r3).
carry(r3) + letter(s) + letter(m) = letter(o) + 10 * carry(r4).
    letter(m) = carry(r4).
    letter(m) ~= 0.
    letter(s) ~= 0.
Avec déclaration des données à prendre en compte avant la résolution (dmi,dma, facilité
d'écriture)
    s=1   e=2   n=3    d=4  m=5   o=6   r=7   y=8
    r1=1  r2=2  r3=3  r4=4
```

[86] L'analyse lexicale et syntaxique de l'énoncé du problème sont implantées en suivant le pattern de comportement *Visitor*.

```
    B=1
```

Figure 34: Spécification du problème de crypto-addition SEND+MORE=MONEY.

L'ouverture vers des contraintes exprimées par du code Smalltalk permet d'énoncer des problèmes combinatoires sur des ensembles énumérables d'objets ; cela permet de modéliser un ensemble sans connaître explicitement ses éléments, par exemple les instances d'une classe. La relation de la contrainte est décrite dans un bloc Smalltalk, c'est à dire une fonction booléenne dont les arguments sont les valeurs possibles des inconnues.

```
  problemeDeTrajetVilleAVille
      "Luc et Vincent partent des villes A et B distinctes à la recherche l'un de l'autre. Sachant que l'on connait un moyen de calculer la distance parcourue par une personne depuis sa ville de départ, trouver la distance parcourue par Luc et Vincent au moment où ils se rencontrent"
  classe cible : Personne class
  soit
      constante vincent.
      constante luc.
      instants = intervalle 0, 1000.
      distance = intervalle 100, 1000.
  trouver
      constante x : instants.
      constante distanceVincentEnx : distances.
      constante distanceLucEnx : distances.
  verifiant
      distanceVincentEnx = {vincent vitesseMoyenne}*(x -
      {vincent tDepart}).
      distanceLucEnx = {luc vitesseMoyenne}*(x - {luc tDepart}).
      distanceVincentEnx = distanceLucEnx.

  avec
      vincent = {Personne vincent}.
      luc = {Personne luc}.
```

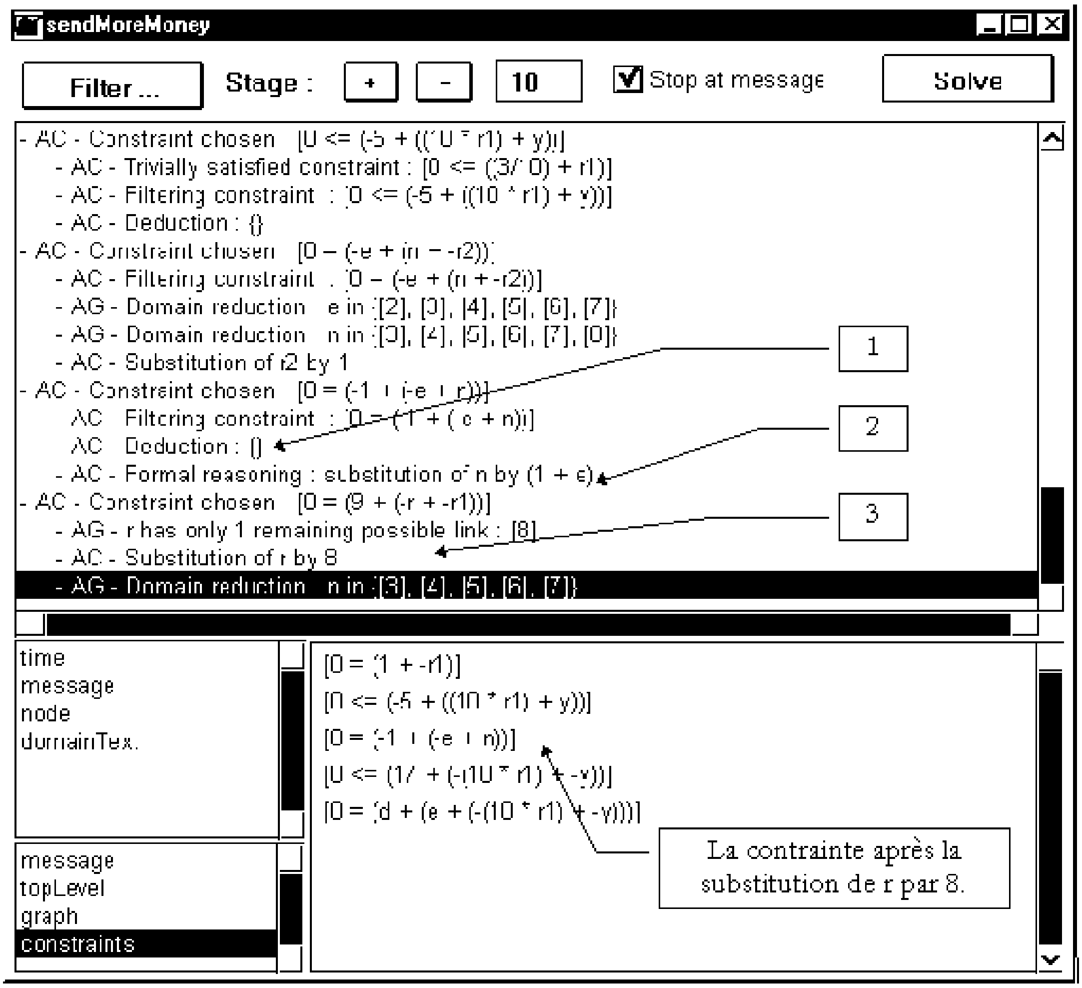
Figure 35: Enoncé d'un problème sur des données objets[87]

AliceTalks fournit une explication succincte en vue de comprendre la résolution : le traceur affiche dynamiquement un message signalant l'action d'un module[88]. En mode debug, le système s'arrête à chaque message qui a été spécifié comme point d'arrêt. La trace de résolution peut être affichée avec plus ou moins de détails grâce à une indentation des messages lorsqu'une action se décompose en plusieurs sous-actions.

Exemple : trace d'exécution du cryptogramme SEND+MORE=MONEY au moment où le module de raisonnement formel évite un choix

[87] La partie en accolades est du code dans le langage hôte, compilé et traduit en un bloc Smalltalk lors de la construction des contraintes. La partie DATA contient les initialisations des variables pointant sur les objets du problème.

[88] Le message est spécifique de l'action et étiqueté pas les initiales du module.



A ce moment plus aucune réduction de domaine n'a lieu (1). Le domaine de R1 est {0,1}, celui de N, R et E est [1..8]. Le module de raisonnement formel déduit la contrainte [R+R1=9] (2) à partir de [N = 1+E] et [10+E = R+R1+N]. Le filtrage de [R+R1 = 9] entraîne R=8 et R1=1 (3). Plus de détails sont disponibles dans [Liret et al. 1997] et la thèse de J.-L. Laurière.

3.1 Réécriture des expressions dans AliceTalks

Les expressions arithmétiques, relationnelles, fonctionnelles et logiques sont représentées en DAG.

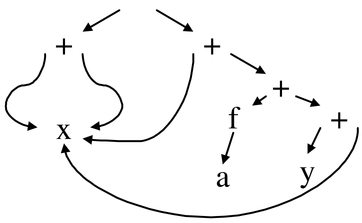


Figure 36: graphe de (x+x)- (x+f(a)+y+x).

Pour enlever l'ambiguïté des opérations commutatives, l'ordre de simplification modulo AC suivant est utilisé :

Procédure `Compare(t1,t2)`
  soit $<_{pred}$ l'ordre de précédence sur les opérateurs, préordre.
  soit $<_{lex}$ l'ordre lexicographique, total.
  Début
    si `OP1` $<_{pred}$ `OP2` **alors** `t1 < t2`
    sinon si `OP1 = OP2`

```
            si n < m alors t1 < t2      "ordre sur la taille"
        sinon
            ordonner(x1,...,xn)         "ordre lexicographique"
            ordonner(y1,...,yn)         "ordre lexicographique"
            si {x1,...,xn} <lex+rdo {y1,...,ym} alors t1 < t2
    "ordre lexicographique étendu à des chemins"
    sinon t1 ≥ t2
Fin
```

Figure 34 : Procédure de comparaison entre deux termes de AliceTalks, t1=OP1(x1,...,xn) et t2=OP2(y1,...,ym)

Le pattern de comportement Interpreter s'adapte naturellement aux opérations sur la structure arborescente des expressions et contraintes, par exemple la comparaison selon l'ordre de décomposition récursif rdo défini plus haut.

La méthode `AEBinaryRelation.compareWith(anExp)` compare chaque opérande d'une relation avec `anExp` tandis que la méthode du même nom dans la classe des littéraux (variables et constantes) applique l'ordre de précédence défini dans Alice.

Par ailleurs toute variable instantiée est supprimée des contraintes et remplacée par sa valeur. L'absence d'occurrence de variables instanciées évite le cas, souligné dans [Jouannaud & Lescanne 1986], où il est impossible de comparer deux variables distinctes dont l'une est instanciées mais pas l'autre. La simplification des contraintes à l'instantiation rend donc l'ordre total, sur les expressions arithmétiques, logiques et relationnelles.

Le pattern Interpreter est utilisé également pour mettre les contraintes sous une forme canonique. Partant de l'hypothèse qu'une règle de réécriture est l'abstraction d'une règle d'inférence, la première version d'AliceTalks utilisait un interpréteur Prolog en Smalltalk (MeiProlog [Aoki 1994]). Dans une seconde version, nous l'avons remplacé par un mécanisme de réécriture spécifique et moins lourd, implanté directement dans la hiérarchie des expressions. En effet un langage à objets ne possède pas à la base de moteur d'inférence ni d'interprétation symbolique d'un objet. En revanche l'héritage permet d'organiser la base de réécriture en regroupant les règles en fonction du type de leur prémisse. Le système de réécriture est divisé en paquets de règles. La méta-règle de division est de regrouper les règles dont les prémisses s’unifient avec le même type d’expression. Le moteur de saturation est implanté au niveau de la classe abstraite.

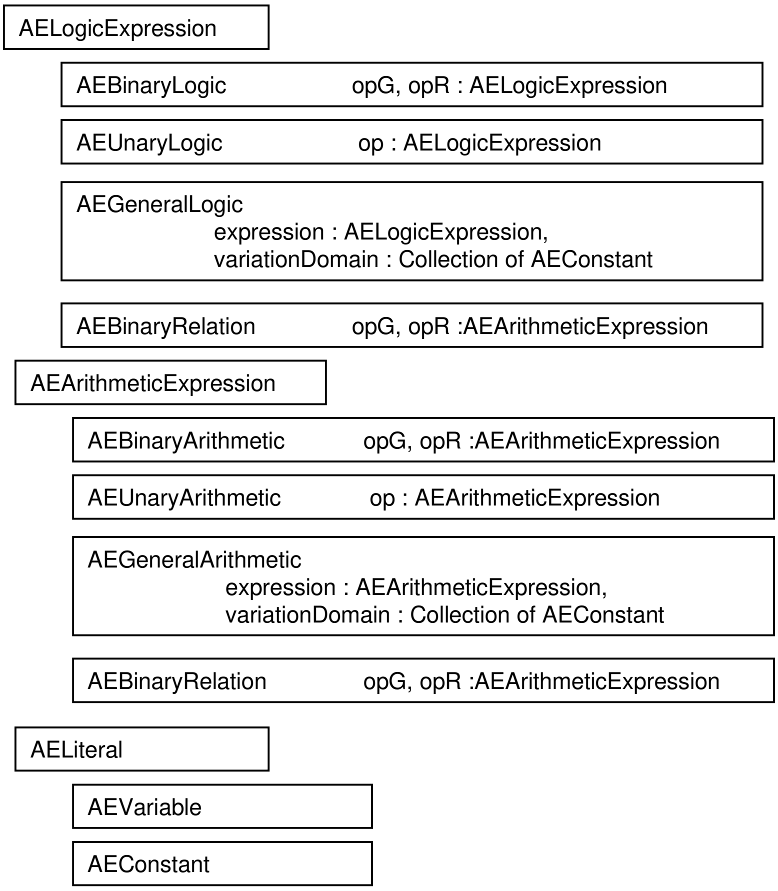


Figure 35 : typologie des expressions

Le *filtrage logique* sur des termes à plus de deux opérandes étant combinatoire, les expressions n-aires sont transformées en arbre binaire. De ce fait, il faut dupliquer les règles à plusieurs niveaux. Par exemple, la règle X+X → 2*X s'applique sur un terme ayant deux opérandes identiques, tandis que la règle X+X+Y → 2*X+Y s'applique sur un terme ayant une opérande contenant une occurrence de son autre opérande (C.f. Figure 37). Notons que ces règles ne sont pas automatiquement déduites par la procédure de complétion. Mais si l'on introduit une partie des règles sur deux niveaux, la complétion rajoute les règles manquantes.

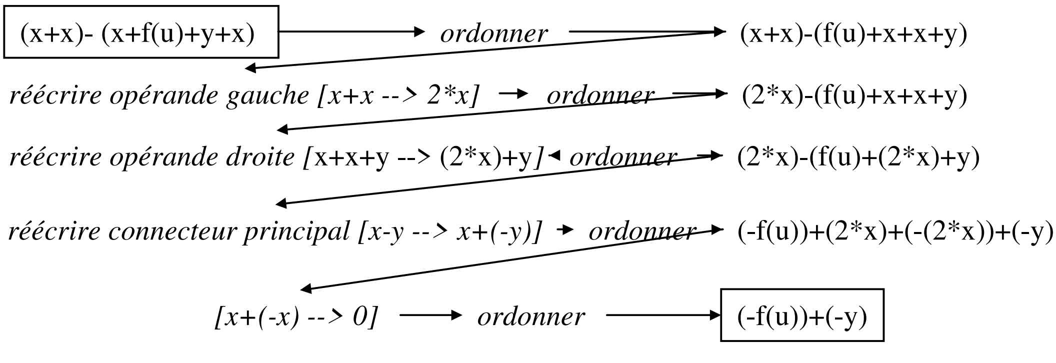


Figure 37 : exemple de séquence de normalisation

Au niveau de chaque classe C sont implantées les règles de réécriture qui s'appliquent sur des termes issus de la classe C. La méthode `applyOwnRules` qui applique les règles spécifiques de la classe ou héritées, y est redéfinie. La version actuelle ne dispose pas d'interpréteur ; une règle est représentée par deux méthodes, l'une contenant un test vrai si le terme est filtré par la prémisse et l'autre implantant l'action de la règle, *i.e.* le remplacement du terme par un nouveau terme.

```
AEExpression            → normalize →     applyOwnRules
                                          (règle générique d'évaluation)
    AEBinaryRelation
      AEEquality                          applyOwnRules
                                          (a=a → true ; a=b → a-b=0)
    AEArithmeticExpression
      AEUnaryArithmetic
          AEOpposite                      applyOwnRules
                                          (-(-x) → x ; -(x+y) → (-x)+(-y))
      AEBinaryArithmetic
          AEAddition                      applyOwnRules
                  (k*a+l*a → (k+l)*a, si k et l sont des constantes ; a+0 → a)
        AEGeneralArithmetic
          AENaryAddition                  applyOwnRules
                  (v1*a+...+vk*a → (v1+...+vk)*a si tout vi est une constante)
    AELiteral                             applyOwnRules ()
      AEVariable
      AEConstant
```

Figure 38 : décomposition de la base de réécriture selon le type de contraintes

La décomposition de la base de règles revient à pré-filtrer les règles selon le connecteur principal du terme candidat à la réécriture ; le *filtrage logique* de la règle opère lui sur les opérandes du terme. Ce modèle d'implantation a pour conséquence une réduction du coût d'exécution car la base n'est pas parcourue en entier. Par ailleurs chaque base est confluente pour les expressions instances de la classe où la base est implantée.
L'algorithme suivant est appliqué jusqu'à obtention du point fixe pour la réécriture d'un terme :

1) Supprimer les connecteurs $-_2$ et /
2) Distribuer $-_1$ de manière à ce qu'il ne porte que sur des atomes.
3) Appliquer les règles de distributivité jusqu'à saturation, de manière à obtenir des sommes n-aires.
4) Ordonner les termes suivant l'ordre de précédence $+>*>-_1>$atomes. On utilise un tri lexicographique pour ordonner les termes des expressions n-aires et les atomes.
5) Appliquer le système de réécriture jusqu'à saturation sur les sommes et produits.
6) Si l'expression réécrite est distincte de l'expression obtenue en 3), recommencer en 3).

4 RESULTATS DE RABBIT SUR UN PROBLEMES DE ROBOTS

Alice résout un problème d'optimisation en trois phases : la phase 0 trouve une solution, la phase 1 trouve une bonne solution à heuristiques variables, la phase 2 prouve l'optimalité.

Temps-Rabbit = *tA+tC+tG*, ou *tA* est le temps mis par Alice en début de résolution pour détecter une situation favorable au programme spécifique, *tC* est le temps de compilation de celui-ci et *tG* est son temps d'exécution. Les temps sont mesurés en secondes. Nous ne disposons pas des nombres de retour-arrières.

Exemple 1 : génération du programme spécifique après la phase 0

TA = 4. *TC* = 5.

Problème avec 4 caisses (8 tâches)	30	34	60	100
Temps Charme	4	6	10	13
Temps Rabbit	0.1	0.2	0.5	0.6

Exemple 2 : génération du programme spécifique en la phase 2

$TA = 9$. $TC = 8$.

Problème avec 8 caisses (16 tâches)	30	34	60	100	138
Temps Charme	1978	3747	7334	10707	10817
Temps Rabbit	11	53	215	678	810

Ces résultats onte été obtenues par J.-L. Laurière en 1992 sur des machines aux performances comparables. Les résultats en Charme ont été obtenus sur une HP 9000- 433 S (horloge 33 Mhz, performance=26MIPS). Les résultats en Rabbit ont été obtenus sur une SUN SPARC2 (horloge=40 Mhz, performance=28,5 MIPS).

5 LISTE DES HEURISTIQUES UTILISES POUR LES COMPARAISONS DE RESOLUTION

Mot clé	Type de choix	Description
minSize	Variable	rend la variable de plus petit domaine
mostConstrained	Variable	rend la variable la plus contrainte
greatestRegret	Variable	rend la variable qui a le plus grand regret
minValue	Domaine	rend la plus petite valeur du domaine
maxValue	Domaine	rend la plus grande valeur du domaine
Ordre (inversement) Lexicographique	Variable	ordonne la liste des variables et rend le premier élément.

6 SYSTEME DE NORMALISATION DE ALICETALKS

Le système normalise les expressions arithmétiques, logiques et relationnelles sur l'ensemble des entiers relatifs.

Opérateurs arithmétiques : somme binaire (+), soustraction ($-_2$), produit (.), division (/), opposé ($-_1$).

Opérateurs relationnels : supérieur (>), inférieur (<), supérieur ou égal (≥), inférieur ou égal (≤), égalité (=), différence (≠).

Opérateurs logiques : négation (¬), conjonction (∧), disjonction inclusive (∨), disjonction exclusive (xor), implication (⇒), équivalence logique (⇔), quantification existentielle (∃), quantification universelle (∀).

Ordre de précédence des opérateurs : $\{\geq , \leq\} <_{pred} \{<, >\} <_{pred} \{=, \neq\}$

$\{+, -_2\} <_{pred} \{. , /\} <_{pred} -_1$

$\{\exists , \forall, \Rightarrow , \Leftrightarrow\} <_{pred} \{\vee, \text{xor}\} <_{pred} \wedge <_{pred} \neg$

Les opérateurs associatifs et commutatifs sont = , ≠, +, . , ⇔, ∨, xor et ∧. La normalisation supprime les opérateurs ⇒ et ⇔ des expressions logiques.

6.1 Expressions arithmétiques

Les expressions n-aires sont représentées par des sommes ou produits n-aires dont les termes ou facteurs sont mémorisés dans une liste de taille indéfinie. Les règles s'appliquent sur tout élément de la liste. On peut ainsi voir une expression n-aire comme un arbre non binaire. La représentation en arbre n-aire, associée à un ordre des éléments en fonction de l'ordre de précédence permet de ne pas avoir de règles parcourant plusieurs niveaux de l'arbre, comme c'est le cas si l'on choisit une représentation en arbre binaire. Par exemple, la règle $x + x \rightarrow 2.x$ parcourt le niveau des termes de + ; si l'addition a 3 termes il faut avoir aussi la règle $x + (x + y) \rightarrow 2.x + y$.

<u>Regroupement de coefficients (simplification de combinaison linéaires) :</u>

$a(b . x) \rightarrow (ab) . x$

$-(b . x) \rightarrow (-b) . x$

$-(x + y) \rightarrow -x + -y$

$x . (-y) \rightarrow -(x . y)$

$a + (b+x) \rightarrow (a+b) + x$

$a . (b.x) \rightarrow (ab) . x$

$1.x \rightarrow x$

0.x	→	0
x + 0	→	x
0 + x	→	x
b.x + a.x	→	(b+a) . x
x + a.x	→	(a+1) . x
-x + a.x	→	(a-1) . x
x+x	→	2.x
(-x) + x	→	0
x+ (-x)	→	0

<u>Elimination des opposés :</u>

-(-x) + y	→	x + y
-(-x)	→	x
-0	→	0

<u>Elimination du moins binaire :</u>

x – y	→	x + (-y)

<u>Distributivité :</u>

x . (y + z)	→	(x . y) + (x . z)
(x + y) . (z + t)	→	(x . z) + (y . t)

<u>Réécriture en sus de simplification mais utile en combinaison avec la réduction de domaine de la satisfaction de contraintes :</u>

x . x	→	x^2
$(a + y)^2$	→	$a^2 + 2.a . y + y^2$, si a est une constante.

6.2 Relations logiques

x ≠ x	→	False [89]
x = x	→	True
x > x	→	False
x ≥ x	→	True
0 = 0	→	True

<u>Regroupement des termes à gauche :</u>

x = y	→	y - x = 0 , si x > y
x ≠ y	→	y - x ≠ 0 , si x > y
y ≥ x	→	y - x ≥ 0 , si x > y

<u>Elimination des opposés et coefficients [89]:</u>

-x = 0	→	x = 0
-x ≠ 0	→	x ≠ 0
a . y = 0	→	y = 0 , si a est une constante.
a . y ≥ b	→	y ≥ b/a , si a>0
a . y ≥ b	→	-y ≥ b/a , si a<0

<u>Simplification des coefficients [89] :</u>

$$\sum_i a_i x_i = b \xrightarrow[\textit{réécriture}]{} \frac{(\sum_i a_i x_i)}{\mathrm{p\,gcd}(a_i)} = \frac{b}{\mathrm{p\,gcd}(a_i)}.$$

$$\sum_i a_i x_i > b \xrightarrow[\textit{réécriture}]{} \mathrm{p\,gcd}(a_i) > 0 \Big| \ \frac{(\sum_i a_i x_i)}{\mathrm{p\,gcd}(a_i)} > \frac{b}{\mathrm{p\,gcd}(a_i)}.$$

$$\sum_i a_i x_i > b \xrightarrow[\textit{réécriture}]{} \mathrm{p\,gcd}(a_i) < 0 \Big| \ \frac{(\sum_i a_i x_i)}{\left|\mathrm{p\,gcd}(a_i)\right|} < \frac{b}{\left|\mathrm{p\,gcd}(a_i)\right|}.$$

---

[89] Le filtrage du système de contraintes correspondant aux prémisses aboutit à un résultat qui inclut celui de cette règle.

7.3 Expressions logiques

True $\vee$ p	→	True
False $\vee$ p	→	p
p $\vee \neg$(p)	→	True
p $\vee$ p	→	p
p $\vee$ (p $\vee$ q)	→	p $\vee$ q
False $\wedge$ p	→	False
p $\wedge$ p	→	p
True $\wedge$ p	→	p
p $\wedge$ (p $\wedge$ q)	→	p $\wedge$ q
p $\wedge \neg$(p)	→	False
p xor $\neg$(p)	→	True
p xor p	→	False
p xor $\neg$(p)	→	True
False xor p	→	p
True xor p	→	$\neg$(p)
p xor (p xor q)	→	q
$\neg(x \neq y)$	→	$x = y$
$\neg(x = y)$	→	$x \neq y$
$\neg$(False)	→	True
$\neg$(True)	→	False
$\neg(x \geq y)$	→	$y > x$
$\neg(\neg(x))$	→	x

<u>Distributivité :</u>

p $\wedge$ (q xor r)	→	(p $\wedge$ q xor (p $\wedge$ r)
p $\wedge$ (q $\vee$ r)	→	(p $\wedge$ q $\vee$ (p $\wedge$ r)

<u>Conversion et lois de De Morgan :</u>

$(p \Rightarrow q)$	→	$\neg(p) \vee q$	
$(p \Leftrightarrow q)$	→	$(p \Rightarrow q) \vee (q \Rightarrow p)$	
$\neg(p \wedge q)$	→	$\neg(p) \vee \neg(q)$	lois de De Morgan
$\neg(p \vee q)$	→	$\neg(p) \wedge \neg(q)$	lois de De Morgan
$\neg(\exists x, p))$	→	$\forall x, \neg(p)$	
$\neg(\forall x , p)$	→	$\exists x, \neg(p)$	

<u>Evaluation des constantes :</u>

Toutes les expressions constantes sont évaluées par la fonction polymorphique *eval*.

a + b	→	*eval*(a + b)
a * b	→	*eval*(a * b)
(a = b)	→	*eval*(a = b)

8 TYPOLOGIE DU RAISONNEMENT SYMBOLIQUE

Ordre de simplification sur les contraintes générales

$<$ est défini sur les classes de contraintes pour les opérateurs arithmétiques et sur l'arité pour les contraintes générales (globales).

$c_1 < c_2$ si et seulement si une des deux conditions suivantes est satisfaite :

classe(c_1) = classe(c_2) et arity(c_1) $<_i$ arity(c_2)

classe(c_1) $<_p$ classe(c_2)

$<_i$ est l'ordre total sur $\mathbb{N}$. $<_p$ est un ordre de précédence basé sur les connecteurs des contraintes :

$\{+,-_2 < \{*,/\}<\{-_1,\hat{}2\}$; $(x \geq y) < (x > y)$; $\{=_n , \geq_n\}< \{=_2, \geq_2\}$

8.1 Réécriture et propagation unaire

`CtClassRule`

La contrainte cible est de même classe que la contrainte source mais son expression est plus simple. Elle inclut par exemple la réécriture d'instantiation (`RwInstantiationRule`) et ses dérivées (réécriture d'une fonction Y = f(X), d'une contrainte linéaire), simplification des coefficients d'une combinaison linéaire (`LinearCtRule`) ou d'une relation quelconque exprimée par un lambda-terme à n arguments (`BlockCtRule`).

- ⇨ `BTPerformCtRule Y=f(X,V)` → `Y=f(X,α)` (réécriture dynamique)
- ⇨ `BTBlockCtRule λ.xyz.(T y z)` → `λ.yz.(T y z)` si `T` n'utilise pas `x` (réécriture statique)

`Règles unaires`

- ⇨ `SquareRule` : $(-x)^2 \rightarrow x^2$ (réécriture statique)
- ⇨ `OppositeRule` : $-(-x) \rightarrow x$ (réécriture statique)
- ⇨ `TimesVarRule` : $x*x \rightarrow x^2$ (réécriture statique)
- ⇨ `DeveloppeSquareRule` : $(a+b)^2 \rightarrow a^2 + 2*a*b + b^2$ (propagation dynamique)
- ⇨ `DeveloppeProduitLinéaire` : $(x*(y + z)) \rightarrow x*y + x*z$ (propagation dynamique)
- ⇨ `FactoriseRule` : $x^2 - y^2 \rightarrow (x+y)*(x-y)$ (inverse de la règle précédente). Ces deux règles sont en conflit *a priori*. On résout le conflit en empêchant que l'une ne s'applique pas sur les contraintes issues de l'autre (propagation statique).
- ⇨ `AllDifférentRule` : Si la fonction (Variables → Domaines) est une bijection alors la somme de ses images vaut la somme des nombres du domaine (propagation statique).

8.2 Exemple de simplification et décomposition

La simplification se combine avec la décomposition. Par exemple, soit `a` une constante et `y` une variable ; la contrainte $(a+y)2 = z$ est décomposée d'abord en $\{t_1 = a+y \; ; \; t_1^2 = z\}$ puis t_1^2 est normalisée en $a^2 + 2.a.y + y^2$ qui est elle-même décomposée en $\{t_2 = a^2 \; ; t_3 = 2.a.y \; ; t_4 = y^2\}$. Enfin la contrainte $t_1^2 = z$ est décomposée en $\{t_3 + t_4 - z = -t_2\}$. Apres la simplification de t_1^2, la contrainte $t_1 = a+y$ n'est plus utile et n'a même plus aucun sens puisque t_1 n'est plus une inconnue du problème transformé. On retire donc la contrainte $t_1 = a+y$. Remarquons que si on garde la contrainte, l'ensemble des solutions ne change pas ; le problème est équivalent. Finalement le problème qui est résolu est $\{t_2 = a^2 \; ; \; t_3 = 2.a.y \; ; \; t_4 = y^2 \; ; \; t_3 + t_4 - z = -t_2\}$.

Pour représenter les contraintes non linéaires, BackTalk décompose la contrainte en introduisant des variables intermédiaires liées à des contraintes primitives représentant chaque terme non linéaire de la contrainte. Cette décomposition est utilisée en BackTalk car elle permet d'appliquer les algorithmes de filtrage optimisés pour les contraintes linéaires, sur des contraintes non-linéaires. Par exemple, toute contrainte de la forme $\sum_{i=1}^{n_c}(\alpha_i \cdot v_i^{\beta_i}) = \delta_c$ se décompose en :

- $\{W1,\ldots,Wn_c\}$, un ensemble de variables définies par les contraintes $v_i^{\beta i} = Wi$, pour tout i de 1 à n_c,
- la contrainte $\sum_{i=1}^{n_c}(\alpha_i.W_i) = \delta_c$.

8.3 Réécriture et propagation n-aire

La bibliothèque de base implantée au dessus de BackTalk contient deux pôles principaux et pour chaque pôle, des règles générales et des règles spécifiques :

Raisonnement dans une même classe : $C,\ldots, C \rightarrow C'_1,\ldots,C'_k$

- Règle générale pour les contraintes fonctionnelles,

- Règle spécifique pour les contraintes de différence

Règles n-aires

- Règles générales arithmétiques : composition linéaire de *k* contraintes

  Elles s'appliquent à des égalités et inégalités linéaires. Trois cas particuliers sont distingués : k = 2, k quelconque et application récursive de la composition linéaire binaire, k quelconque et application de la méthode d'élimination de variable Gauss-Simplexe.

  - La composition linéaire binaire de C1 et C2 consiste à calculer un coefficient *p* et à ajouter la contrainte C1 – *p**C2.
  - L'application récursive de la composition binaire consiste à calculer une contrainte . $(\ldots((C_1 - k_1{*}C_2) - k_2{*}C_3)\ldots - k_{n-1} * C_n)$.

- Règles spécifiques

  Implication des expressions de cardinalités génériques et contraintes d'occurrence.

  - Propagation de la contrainte de différence sur une somme ternaire linéaire ou sur un produit de deux facteurs.

9 IMPLEMENTATION

9.1 Contrainte de cardinalité générique

De même que pour définir une contrainte d'occurrence en BackTalk, on instancie `BTCardinalityCt` en évaluant `(BTCardinality on: cases value: 4)`, pour définir une contrainte de cardinalité générique, on évalue `(BTCardinalityGen on: cases expression:[:x|x squared > 4])`.

Exemple 1

```
| S pbm cardV |
pbm := BTSolver new label: ` cardinalityGenExample'.
S := ArraySize: 100 domain: (0 to: 9).
cardV := V domain: (0 to: 10).
"| { x in S / x²=4} |) = cardV"
  (BTCardinalityGen on: S expression: [:x|x squared =4])@= cardV.
"|{x in S / x=2} |, Occurrence(S, x=2)"
  (BTCardinality on: S value: 2) @=5.
pbm variablesToInstantiate: (Array with: cardV), S.
```

Exemple 2

```
| cases pbm cardV |
pbm := BTSolver new label: `cardinality test'.
cases := V arraySize: 100 domain: #(0 1 2 3 4 5 6 7 8 9).
cardV := V label: `cardVar' dom: #(0 20).
"| { x in S / x² > 4} | = cardV"
(BTCardinalityGen on: cases expression: [:x|x squared >4]@= cardV.
"Pour tout i de 0 à 9, |{x in S / x=i} | @= i."
  0 to: 9 do: [:i | (BTCardinality on: S value: i) @= i].
pbm variablesToInstantiate: (Array with: cardV), cases
```

9.2 Le système CSP+RCS construit au-dessus de BackTalk

Cette section décrit les détails d'implémentation en Smalltalk de la communication entre les trois acteurs du système CSP+RCS, moteur CSP, contrainte et règle.

CSP+RCS

Le moteur CSP implante deux performatifs de résolution, `firstSolution` et `nextSolution`, ce qui facilite la résolution incrémentale de problème d'optimisation. L'étape de base (boucle `forward)` consiste à sauver le contexte, propager les événements et règles de R.F. puis choisir la variable à instancier et la valeur d'instanciation et propager le choix dans les contraintes. La commande `saveContext` sauve un contexte. Le choix d'instantiation qui a provoqué la sauvegarde du contexte sera interdit si le contexte est remis en cause

```
Solveur. forward
   self saveContext.
   isComposing ifFalse: [self processRSetCohérence].
   self makeAChoice
```

```
Solveur. ProcessRSetCohérence
  "combinaison de toutes les contraintes actives. Filtrage après chaque composition.. "
   | c |
   isComposing := true.
   [      c :=self bestCtToPropagate.
          c == nil ifTrue: [isComposing := false. ^self].
          c propagateCompositionsAndRewrite.
          self mustNotPropagateRSOnCt: c.
          self removeObsoleteConstraints
   ] repeat.
   isComposing := false
```

L'attribut `isComposing` décrit l'état du solveur selon qu'il est entrain de transformer le problème ou non. Lorsque d'autres contraintes sont ajoutées pendant la propagation symbolique, elles sont mémorisées dans la liste `ctsToPropagate`, en vue d'une manipulation ultérieure. La commande de mémorisation est `mustPropagateRSOnCt(aConstraint)`. La composition n'aura lieu que si `aConstraint` n'est pas supprimée avant.

L'opération de traitement symbolique à partir d'une contrainte est implantée par la primitive `propagateCompositionsAndRewrite`.

```
Contrainte.propagateCompositionsAndRewrite
   composeDemon isActive ifTrue: [composeDemon trigger;
   deactivate].
   rwDemon isActive ifTrue: [rwDemon trigger; deactivate]
```

L'envoi du message propagateCompositionsAndRewrite délègue le traitement symbolique aux démons reliés à la contrainte. Le lien entre une contrainte et les règles capables de la traiter est basé sur des démons et détaillé dans le chapitre IV-2. Lorsque le moteur CSP est informé d'une demande de propagation ou de réécriture, il mémorise la contrainte dans la file d'attente comme candidate pour la réécriture par RCS. Les primitives `willBeComposed` ou `willBeRewritten` sont responsables de l'enregistrement d'une contrainte pour un type de traitement symbolique .

```
Contrainte.willBeComposed
 "En attendant la fin des traitements éventuellement en cours dans le solveur, mémorise moi-même
    dans la file d'attente"
   composeDemon activate.
owner mustPropagateRSOnCt: self
```

Solveur symbolique

Les règles de RCS ont deux modes de création : à la volée, ou à l'ajout d'une contrainte candidate. Le message `normalize`, envoyé à une contrainte retourne une nouvelle contrainte simplifiée et retire la contrainte elle-même. Celle-ci est la contrainte active des règles appelées ou créées dans la méthode.

Exemple : simplification d'une contrainte carré [x^2=y]

```
BTSquareConstraint.  normalize
   c := (SquareRule solver: owner) applySquare: self ; destroy.
   "(-x)² → x²"
   ^(DevelopSquareRule solver: owner) apply: c ; destroy.
   "(x+y)² → x² +2xy + y²"
```

Dans l'exemple ci-dessus, *self* dénote la contrainte active. Les règles sont réellement déclenchées si seulement si la contrainte vérifie les conditions.

Par ailleurs, en raison de la représentation des règles par des objets, une règle de RCS ignore si elle a changé de statut. Il faut donc le lui dire explicitement. C'est pourquoi des messages de signalisation de changement de statut sont implantés dans la classe abstraite `RCSRule`. Par exemple, les événements *Succès*, *Echec, Utile et Inutile* sont générés par envoi d'un message à la règle elle-même au niveau de l'action. Les primitives de changement d'état sont `succeed`, `fail`, `setUtile` et `setInutile`. Le message `succeed` met à jour un attribut booléen `succes` à VRAI, le message `fail` met `succes` à FAUX.

La commande `apply(aConstraint)` est responsable de l'exécution de la règle, c'est à dire 1) la sélection des contraintes candidates vérifiant les conditions et les pré-conditions des stratégies attachées à la règle ; 2) le déclenchement de l'action pour chaque prémisse et 3) la signalisation du changement d'état. La commande `block` (respectivement `unblock`) autorise (respectivement interdit) le déclenchement. Le statut est implanté par un attribut booléen.

9.3 Implémentation de la règle Produitsubstitution1 (section 3.4.2)

La règle *ProduitSubstition1* (section 3.4.2) s'exécute globalement ou à partir d'une contrainte Produit (`ProductCt`) ou Egalité (`EqualityCt`).

```
ProduitSubstitutionRule.findPremissesGlobal
| r partner |
r := BTCollection new: produits size.
products do: [:tv |
 partner := equalities select: [:c | c hasSameVariablesAs: tv].
 partner size > 0 ifTrue: [r add: tv -> partner]
].
^r
```

La méthode `apply(aCt)` est utilisée si une des contraintes est connue[90] ; le filtrage logique revient alors à trouver la liste des contraintes partenaires de la contrainte connue.

```
ProductSubstitutionRule.findPremisses
 ^activeCt isEqualCt
    ifTrue: [(equalities includes: activeCt)
       ifFalse: [^self error: 'equality activeCt is not
candidate ']
       ifTrue: [produits select: [:c |
                   activeCt hasSameVariablesAs: c]]
     ]
    ifFalse: [(products includes: activeCt)
       ifFalse: [^self error: 'product activeCt not candidate
']
       ifTrue: [equalities select: [:c |
                   c hasSameVariablesAs: activeCt]]
     ]
```

10 EXPERIMENTATIONS

10.1 Cryptogramme SEND+MORE=MONEY

Détails de la résolution avec : borne-cohérence, Full Look-ahead, minSize&minValue, ordre des variables inversement lexicographique.

Actions	S	E	N	D	M	O	R	Y	R1	R2	R3	R4
BC Choix R2=0	9	2..8	2..8	2..8	1	0	2..8	2..8	0..1	0..1 0	0	1

[90] La contrainte active en l'occurrence.

BC		4..8	4..8	4..8			2..8	2..6				
Choix R1=0									0			
BC échec!		4	4				2..4					
Choix R1=1									1			
BC échec!		7..8	4..5	4..8			2..3	2..6				
Choix R2=1										1		
BC		2..7	3..8	2..8			3..8	2..8	0..1			
Choix R1=0									0			
BC échec!	9						9					
Choix R1=1									1			
BC		4..7	5..8	5..8			5..8	2..5				
Choix E=4		4										
BC échec!			5	8			8	2				
Choix E=5		5										
BC			6	7			8	2				

Ligne 4 : Le choix R2=0 suivit du choix R1=0 entraîne la réécriture de (2) en E+R=N, dont le filtrage instancie N et E avec la valeur 4. Le filtrage de la contrainte AllDifférent est alors déclenché et provoque un échec.

Ligne 6 : Ensuite le choix R1=1 (R2 étant égal à 0) entraîne la réécriture de (2) en E+R+1=N, dont le filtrage provoque un échec car aucune solution n'est dans les domaines.

Ligne 8 : Le choix R2=1 entraîne la réécriture de (3) en E+1=N.

Ligne 14 : Le choix E=4 entraîne la réécriture de (1) en D-Y=6 et (2) en N+R=13.

Ligne 16 : Le choix E=5 entraîne la réécriture de (1) en D-Y=5 et (2) en N+R=14.

ALPHA

Voici la trace de la résolution du problème ALPHA (C.f. chapitre II) lorsque les compositions sont activées et avec l'heuristique minSize&MinValue :

alpha: contraintes=21 d0=0.0598291 d1=4.57692 d2=1.24408 pM=0.666667
doBest: 'a [(1..26)] ==> [#a int[(1..23)]]'
'c [(1..26)] ==> [#c int[(1..20)]]'
'e [(13..26)] ==> [#e int[(15..26)]]'
'f [(1..14)] ==> [#f int[(1..12)]]'
'g [(1..26)] ==> [#g int[(4..26)]]'
'i [(9..26)] ==> [#i int[(11..26)]]'
'l [(1..14)] ==> [#l int[(1..10)]]'
'o [(1..17)] ==> [#o int[(6..17)]]'
's [(1..26)] ==> [#s int[(1..24)]]'
't [(1..14)] ==> [#t int[(1..12)]]'
'u [(1..14)] ==> [#u int[(1..12)]]'
'z [(3..26)] ==> [#z int[(5..26)]]'
'alpha: contraintes=22 d0=0.0626781 d1=4.73077 d2=1.36538 pM=0.636364'
'a+b+e+2.l+t=45'
doBest: doBest: doBest: doBest: doBest: 'r [(1..26)] ==> [#r int[(1..25)]]'
'y [(1..26)] ==> [#y int[(1..25)]]'
'alpha: contraintes=29 d0=0.0826211 d1=5.88462 d2=2.15828 pM=0.62069'
'e+l+r+y=47'
doBest: 'g [(4..26)] ==> [#g int[(10..26)]]'
'alpha: contraintes=30 d0=0.0854701 d1=6.03846 d2=2.27071 pM=0.6'
'e+f+l+t+u=30'
doBest: doBest: doBest: doBest: doBest: 'b [(1..26)] ==> [#b int[(1..23)]]'
'c [(1..20)] ==> [#c int[(1..17)]]'
'e [(15..26)] ==> [#e int[(18..21)]]'
'f [(1..12)] ==> [#f int[(1..7)]]'
'g [(10..26)] ==> [#g int[(23..26)]]'
'i [(11..26)] ==> [#i int[(12..26)]]'
'k [(1..26)] ==> [#k int[(6..26)]]'

'l [(1..10)] ==> [#l int[(1..7)]]'
'r [(1..25)] ==> [#r int[(1..21)]]'
't [(1..12)] ==> [#t int[(1..9)]]'
'u [(1..12)] ==> [#u int[(1..4)]]'
'alpha: contraintes=36 d0=0.102564 d1=6.88462 d2=2.96302 pM=0.611111'
'g+n+o+s=61'
doBest: 'p [(1..26)] ==> [#p int[(2..26)]]'
'q [(1..26)] ==> [#q int[(1..25)]]'
'alpha: contraintes=37 d0=0.105413 d1=7.07692 d2=3.09467 pM=0.594595'
'a+e+q+r+2.t+u=50'
doBest: 'h [(1..26)] ==> [#h int[(6..26)]]'
'x [(1..26)] ==> [#x int[(6..26)]]'
'alpha: contraintes=38 d0=0.108262 d1=7.23077 d2=3.21006 pM=0.605263'
'a+n+2.o+p+r+s=82'
doBest: doBest: doBest: 'c [(1..17)] ==> [#c int[(1..14)]]'
'i [(12..26)] ==> [#i int[(13..26)]]'
'o [(6..17)] ==> [#o int[(6..16)]]'
'p [(2..26)] ==> [#p int[(3..26)]]'
'q [(1..25)] ==> [#q int[(1..24)]]'
's [(1..24)] ==> [#s int[(3..19)]]'
'alpha: contraintes=41 d0=0.116809 d1=7.73077 d2=3.63462 pM=0.585366'
'a+c+e+l+s=51'
doBest: 'i [(13..26)] ==> [#i int[(16..26)]]'
'v [(1..26)] ==> [#v int[(6..26)]]'
'alpha: contraintes=42 d0=0.119658 d1=7.92308 d2=3.81657 pM=0.595238'
'2.i+l+n+o+v=100'
doBest: doBest: *** MakeAChoice ***l = 1
triggerBackward
*** MakeAChoice ***l = 2
doBest: doBest: doBest: doBest: 'a [1 (3..20)] ==> [#a int[(5..20)]]'
'b [1 (3..20)] ==> [#b int[1 (3..14)]]'
'c [1 (3..15)] ==> [#c int[1 (3..10)]]'
'i [(15..26)] ==> [#i int[(16..26)]]'
'n [(4..26)] ==> [#n int[(5..26)]]'
'o [(5..16)] ==> [#o int[(10..15)]]'
'q [1 (3..26)] ==> [#q int[1 (3..22)]]'
'r [1 (3..25)] ==> [#r int[1 (3..22)]]'
's [(3..25)] ==> [#s int[(5..15)]]'
'v [(4..26)] ==> [#v int[(5..26)]]'
'w [1 (3..24)] ==> [#w int[1 (3..20)]]'
'z [(6..26)] ==> [#z int[(6..25)]]'
'alpha: contraintes=25 d0=0.0712251 d1=4.69231 d2=1.3284 pM=0.64'
'a+c+e+s=49'
doBest: 'k [1 (3..26)] ==> [#k int[(12..26)]]'
'r [1 (3..22)] ==> [#r int[1 (3..15)]]'
'y [1 (3..25)] ==> [#y int[(5..25)]]'
'alpha: contraintes=26 d0=0.0740741 d1=4.80769 d2=1.40385 pM=0.653846'
'a+k+o+p=57'
doBest: 'e [(19..25)] ==> [#e int[(19..24)]]'
'g [(14..26)] ==> [#g int[(16..26)]]'
'y [(5..25)] ==> [#y int[(6..25)]]'
'alpha: contraintes=27 d0=0.0769231 d1=4.92308 d2=1.45266 pM=0.62963'
'e+f+t+u=28'
doBest: 'y [(6..25)] ==> [#y int[(11..25)]]'
'alpha: contraintes=28 d0=0.0797721 d1=5.0 d2=1.47041 pM=0.607143'
'e+r+y=45'
doBest: doBest: doBest: 'a [(5..20)] ==> [#a int[(5..17)]]'
'n [(5..26)] ==> [#n int[(6..26)]]'
'o [(10..15)] ==> [#o int[(10..14)]]'

's [(5..15)] ==> [#s int[(7..15)]]'
'v [(5..26)] ==> [#v int[(6..26)]]'
'w [1 (3..20)] ==> [#w int[1 (3..18)]]'
'z [(6..25)] ==> [#z int[(8..25)]]'
'alpha: contraintes=31 d0=0.0883191 d1=5.34615 d2=1.72929 pM=0.580645'
'a+b+e+t=41'
*** MakeAChoice ***o = 10
ptChoix=2 (alpha) 7.995s ; 1 bt ; 3 ch
(5 13 9 16 20 4 24 21 25 17 23 2 8 12 10 19 7 11 15 3 1 26 6 22 14 18)

Voici la trace de la résolution sans composition linéaire et avec l'heuristique minSize&Minvalue

```
*** MakeAChoice ***l = 1
*** MakeAChoice ***e = 21
triggerBackward
*** MakeAChoice ***e = 22
triggerBackward
*** MakeAChoice ***l = 2
*** MakeAChoice ***f = 1
triggerBackward
*** MakeAChoice ***f = 3
*** MakeAChoice ***t = 1
triggerBackward
*** MakeAChoice ***f = 4
*** MakeAChoice ***t = 1
triggerBackward
*** MakeAChoice ***c = 6
triggerBackward
*** MakeAChoice ***c = 7
triggerBackward
*** MakeAChoice ***c = 8
triggerBackward
ptChoix=5 (alpha) 0.506s ; 10 bt ; 12 ch
(5 13 9 16 20 4 24 21 25 17 23 2 8 12 10
19 7 11 15 3 1 26 6 22 14 18 )
```

10.2 Carré magique

Solution pour N=3 : 2 7 6
9 5 1
4 3 8

	+ CtBijection	+CtBijection+BT5=5	- CtBijection -BT5=5	-CtBijection + BT5=5
BackTalk *–linlin*	0,158s 69 bt 72 ch	0,054s 11 bt 13 ch	0,141s 70 bt 73 ch	0,038s 11 bt 13 ch
BackTalk *+linlin*	0,0685s 19 bt 22 ch	0,0184s 1 bt 3 ch	0,405s 19 bt 22 ch	0,112s 1 bt 3 ch

Trace d'exécution avec BackTalk+RCS et ItComposition (N=3)

```
   sum [(3..27)] => sum int[(9..21)]
   LINdoBestRemoved 'C Bt 1+Bt 4+Bt 7-sum=0 et Bt 1+Bt 2+Bt 4+Bt 5+Bt
7+Bt 8+sum=45 ==> -Bt 2-Bt 5-Bt 8-2.sum=-45'
   '7   || Magic Square 3 : nv= 10  nc= 17  d1= 9.6  d2= 9.6  d3=
960.0  pM= 0.882353'
   sum [(9..21)] => sum int[15]
   LINdoBestRemoved 'C Bt 3+Bt 6+Bt 9-sum=0 et -Bt 3-Bt 6-Bt 9-
2.sum=-45 ==> sum=15'
   '10  || Magic Square 3 : nv= 10  nc= 18  d1= 8.5  d2= 7.87  d3=
787.0  pM= 0.555556'
   Bt 9 [(1..9) ] => Bt 9 int[(1..8)]
   LINdoBestRemoved 'C -Bt 2-Bt 4+2.Bt 9=0 et Bt 1+Bt 2+Bt 4+Bt 6+Bt
8+Bt 9=30 ==> Bt 1+Bt 6+Bt 8+3.Bt 9=30'
   '60  || Magic Square 3 : nv= 10  nc= 44  d1= 20.5  d2= 44.21  d3=
4421.0  pM= 0.590909'
   Bt 1 [(2..9)] => Bt 1 int[(2..4) (6..9) ]
   Bt 2 [(1..9) ] => Bt 2 int[(1..4) (6..9) ]
   Bt 3 [(1..9) ] => Bt 3 int[(1..4) (6..9) ]
   Bt 4 [(1..9) ] => Bt 4 int[(1..4) (6..9) ]
   Bt 5 [(1..9) ] => Bt 5 int[5]
   Bt 6 [(1..9) ] => Bt 6 int[(1..4) (6..9) ]
   Bt 7 [(1..9) ] => Bt 7 int[(1..4) (6..9) ]
   Bt 8 [(1..9) ] => Bt 8 int[(1..4) (6..9) ]
   Bt 9 [(1..8)] => Bt 9 int[(1..4) (6..8) ]
   LINdoBestRemoved 'C Bt 1+Bt 2+Bt 3+Bt 4+Bt 6=25 et Bt 1+Bt 2+Bt
3+Bt 4+Bt 5+Bt 6=30 ==> Bt 5=5'
   '71  || Magic Square 3 : nv= 10  nc= 47  d1= 19.2  d2= 43.56  d3=
4356.0  pM= 0.617021'
   (Magic Square 3) 443.698s ; 1 bt ; 3 ch
   2 7 6
   9 5 1
   4 3 8
```

Trace d'exécution avec ItComposition et StopIfUseless(5) (N=3)

```
   sum [(3..27)] => sum int[(9..21)]
   LINdoBestRemoved 'C Bt 1+Bt 4+Bt 7-sum=0 et Bt 1+Bt 2+Bt 4+Bt 5+Bt
7+Bt 8+sum=45 ==> -Bt 2-Bt 5-Bt 8-2.sum=-45'
   '2   || Magic Square 3 : nv= 10  nc= 17  d1= 9.6  d2= 9.6  d3=
960.0  pM= 0.882353'
   sum [(9..21)] => sum int[15]
   LINdoBestRemoved 'C Bt 2+Bt 5+Bt 8-sum=0 et -Bt 2-Bt 5-Bt 8-
2.sum=-45 ==> sum=15'
   '5   ||  Magic Square 3 : nv= 10  nc= 18  d1= 8.5  d2= 7.87  d3=
787.0  pM= 0.555556'
```

```
(Magic Square 3) 0.893s ; 2 bt ; 4 ch
2 7 6
9 5 1
4 3 8
```

Trace d'exécution avec ItComposition et StopIfUseless(5) (N=4)

```
sum [(4..64)] => sum int[(24..44)]
LINdoBestRemoved 'C Bt 11+Bt 15+Bt 3+Bt 7-sum=0 et -Bt 11-Bt 12-Bt
15-Bt 16-Bt 3-Bt 4-Bt 7-Bt 8-2.sum=-136 ==> -Bt 12-Bt 16-Bt 4-Bt 8-
3.sum=-136'
'11  ||  Magic Square 4 : nv= 17  nc= 23  d1= 12.2941  d2= 9.29066
d3= 2685.0  pM= 0.913043'
sum [(24..44)] => sum int[34]
LINdoBestRemoved 'C Bt 12+Bt 16+Bt 4+Bt 8-sum=0 et -Bt 12-Bt 16-Bt
4-Bt 8-3.sum=-136 ==> sum=34'
'16  || Magic Square 4 : nv= 17  nc= 26  d1= 12.5294  d2= 9.78893
d3= 2829.0  pM= 0.576923'
Bt 16 [(8..10)] => Bt 16 int[(8..9)]
 (Magic Square 4) 11.795s ; 15 bt ; 19 ch
1 2 15 16
12 14 3 5
13 7 10 4
8 11 6 9
```

10.3 Résultats de l'étude de critères de prévision

nbct	tpssans	btsans	tpsavec	btavec	pourcentred	pourcentcpu
37	3994,75	391,50	73985,75	1242,00	0%	272%
35	52,00	4,00	10775,50	0,50	36%	82%
71	1027,00	40,00	123,00	0,00	0%	6%
91	34,00	0,50	125,50	0,50	0%	50%
65	140,00	4,00	255,00	4,00	0%	2%
49	727398,25	25062,75	1139570,50	2743,25	0%	100%
34	536070,50	21019,75	79983,25	232,25	4%	209%
41	208,00	24,00	408,00	24,00	1%	249%
75	21117,33	819,00	143,00	1,33	0%	5%
39	481,50	12,50	324,50	4,50	-70%	851%
67	114,00	2,00	411,00	2,00	0%	12%
73	203113,50	15533,00	278,00	0,00	0%	17%
45	101,00	4,50	374,50	4,00	0%	-4%
32	97,67	2,33	417,33	2,33	21%	195%
51	284,33	6,33	511,67	0,67	0%	-19%
57	250,67	2,67	1896,00	1,67	0%	0%
77	3377,00	200,75	209,00	0,75	100%	-96%
43	358,00	3,00	671,00	3,00	17%	65%
81	177,00	0,00	704,00	0,00	0%	147%
61	27156,75	1554,50	1759,25	2,00	0%	74%
31	430,75	17,00	1075,50	7,75	25%	141%
48	3680,67	71,33	458,67	71,33	0%	100%
42	1497,00	40,25	5251,75	33,75	0%	293%
53	10100,17	191,00	1219,50	3,50	1%	26%
36	64,00	1,00	394,33	1,00	0%	170%
28	258,80	13,80	2146,00	13,80	50%	69%
47	168,00	1,00	1143,00	1,00	0%	11%
24	489378,00	12435,75	10696534,25	30875,75	50%	269%
26	204,00	0,00	1169,00	0,00	67%	139%
25	277777,60	5181,80	276619,80	5051,80	0%	245%
27	130,50	2,50	769,50	2,50	33%	448%
33	245111,83	4977,83	267176,33	4931,67	0%	157%
63	2312234,50	57920,00	2154429,50	57920,00	0%	10%
22	3143605,00	54554,50	3408451,00	54531,50	0%	261%
40	0,00	0,00	19,00	0,00	0%	473%
23	0,00	0,00	15,00	0,00	50%	101%
18	2246629,50	32558,00	2073148,50	32552,75	50%	6%
38	1149757,50	11985,00	1328428,50	11792,00	0%	580%
13	4977730,00	50329,00	4941339,50	50329,00	44%	10353%
21	1829651,50	16427,00	1910499,00	16427,00	50%	45%
17	223381,00	2306,00	246411,00	2306,00	38%	1328%
20	1866526,50	18382,50	1890243,50	18382,50	33%	60%
30	260214,00	2563,00	291502,00	2551,00	0%	298%
29	60648,00	363,00	64140,00	363,00	0%	87%
12	43310,00	452,00	50797,00	452,00	-9%	608%
16	10686525,00	100001,00	11265323,00	100001,00	0%	82%
15	77764,00	429,00	3251,00	0,00	0%	171%
19	5515428,00	42903,50	5489317,00	42903,50	0%	96%
14	61982,00	343,00	92720,00	343,00	100%	-88%

Tableau 1:résultat des résolutions avec et sans raisonnement symbolique pour 14 série de problèmes aléatoires linéaires du même type que le cryptogramme ALPHA.

Problème	Résolution CSP		Résolution CSP-RCS		d1	d2	pM	d0	N[91]
	Temps (seconde)	Retour-arrières	Temps (seconde)	Retour-arrières					
Send+more=money	0,022	1	0,009	0	0,20	0,38	0,66	0,07	6
Donald+gerald=robert	0,045	11	0,198	7	2,20	0,35	0,57	0,05	7
Alpha 1ère solution	0,112	10	4,11	1	4,57	1,24	0,66	0,05	21
Alpha unicité de la solution	0,5912	27	0,127	10	4,57	1,24	0,66	0,16	58
Magic 3x3	9,472	69	9,25	19	5,0	2,64	0,8	0,18	10
xa*x=bxc 1ère solution	0,028	5	0,028	3	1,44	0,25	1,00	0,06	3
xa*x=bxc 2ère solution	0,032	6	0,032	4	1,66	0,55	1,00	0,14	3
xa*x=bxc 3ère solution	0,04	9	0,038	5	1,16	0,30	0,66	0,14	3
xa*x=bxc les 8 autres solutions	0,076	22	0,066	11	1,33	0,38	0,66	0,14	3
Ain+aisne+drome+marne=somme les trois solutions	0,628	196	2,4	182	2,0	0,44	1,02	0,044	2

Tableau 2 : tableau de critères pour les cryptogrammes.

10.4 Problèmes de contraintes temporelles

Problème Chr 3 inspiré des documentations sur les Constraint Handling Rules

```
x in [-100, 100], y in [-100, 100], z in [-100, 100], t in [-100, 100],
u in [-100, 100].
x= 0 ; 10 ≤x-y ≤ 20. 10 ≤t-z ≤ 20. 30 ≤y-z ≤ 40. 40 ≤t-u ≤ 50.
60 ≤x-u ≤ 70.
VariablesToInstantiate = {x,y,z,t,u}.
Minimiser f(x,y,z,t,u) =x+(2*y)-(3*z)-t-u.
```

Système de gestion de l'emploi du temps d'un centre

En BackTalk, la répartition des fermetures et ouvertures du centre peut s'interpréter sous la forme du CSP suivant :

```
|pbm s e cardV varNbFermeture |
   pbm := BTSolver new.
   e := Educateur nom : 'Dupont'.
   s := BTMatrix width: 6 height: 1.
   s allItemsPut: [ :p |
         V label: 'affectationPlage' domain: EnsemblePlages
     at: p].
"EnsemblePlages est un tableau de collections de toutes les plages indexées sur le jour de la semaine"
   CardV := V label: 'nbFermeture' from: 1 to: 3.
   varNbFermeture := (BTCardinalityGen on: s
               expression: [:x | x estUneFermeture])
               asVariable.
```

[91] N = Nombre de contraintes au début de la résolution.

```
 varNbFermeture @= cardV.
 varNbFermeture <= (BTCardinalityGen on: s
        expression:   [:x   |   x   EstUneOuverture   not])
        asVariable.
pbm variablesToInstantiate: s.
```

Contraintes sur des objets temporels

Afin d'équilibrer la charge totale par éducateur, la somme des charges des tâches réalisées doit être incluse dans un intervalle fixé au départ par le règlement afin d'éviter les déséquilibres entre les charges des éducateurs. Par exemple, soit un ensemble de deux tâches de charges respectives 3 et 4 unités de travail, le code BackTalk suivant contraint la somme des charges des tâches à être plus petite que 6 unités.

```
|p t v d x y s|
     p := BTSolver new.
     x := Tc newInit: 3.
     y := Tc newInit: 4.
     t := Array with: x with: y.
     d := (Array with: 1 with: 2).
     v := BTMatrix width: 2 height: 1.
     1 to: 2 do: [:i | 1 to: 1 do: [:j |
                           v  at:  i  @  j  put:  (V  domain:
                    d)]].
     s := 0.
     1 to: 2 do:[:i|
                    s := s +
                    ((t  atVar:  (v  at:  i@1))  btPerform:
              #charge) asVariable].
     s  <= 6.
     p print: v; variablesToInstantiate: v allItems.
     p printAllSolutionsWithInformation
```

*Figure 39 : cont***r***ainte BTPerformCt pour la répartition des charges.*

La classe `Tc` représente les tâches ; la méthode `newInit: aNumber` crée une instance de charge `aNumber` ; la méthode charge renvoie la charge de la tâche.

Le message `btPerform: aMessageSelector` crée une expression générale, convertible en variable et donc pouvant intervenir dans n'importe contrainte, à une compatibilité de type près. L'expression `(x btPerform: #charge) asVariable @= r` crée une contrainte entre `x` (variable objet dont la valeur est une tâche) et `r` (variable dont la valeur est un entier) signifiant que `r` vaut la charge de la valeur de `x`.