Explication systématique des contraintes indexicales

Ludovic Langevine

Explication systématique des contraintes indexicales

Ludovic Langevine*

SICS, Uppsala Science Park, SE-75183 Uppsala, Suède

langevin@sics.se

Résumé

Plusieurs solveurs de contraintes sur domaines finis, tels GNU-Prolog ou SICStus Prolog, utilisent les indexicaux pour définir leurs contraintes primitives. Un indexical exprime un ensemble de valeurs consistantes pour une variable donnée et peut être compilé en un algorithme de filtrage efficace.

L'explication d'un retrait de valeurs est un sous-ensemble du *store* qui suffit à justifier ce retrait. Les explications permettent de traiter des problèmes surcontraints ou dynamiques, et de mettre en œuvre diverses méthodes de recherche.

Traditionnellement, il faut concevoir un algorithme de calcul des explications pour chaque algorithme de filtrage. Cet article montre que, pour les contraintes spécifiées par des indexicaux, il est possible de dériver automatiquement les algorithmes d'explication. Dans le cas de SICStus Prolog, il est même possible de tirer partie de l'implémentation existante pour optimiser ce calcul.

Abstract

Several constraint solvers over finite domains, such as GNU-Prolog or SICStus Prolog, use indexicals to define their primitive constraints. An indexical specifies a set of consistent values for a given variable and can be compiled into an efficient filtering algorithm.

The explanation of a value withdrawal is a set of constraints that is sufficient to justify this withdrawal. Explanations enable the handling of over-constrained or dynamic problems, and allow the development of new search methods.

Traditionally, an explanation algorithm has to be designed for each filtering algorithm. This paper shows that explanation algorithms can be automatically derived from indexical constraints. In the case of SICStus Prolog, explanation computation can also benefit from the existing implementation.

*Travail conduit pendant un séjour post-doctoral ERCIM.

1 Introduction

Les explications sont une variété de techniques qui gardent trace des décisions du solveur de contraintes, principalement pour améliorer la puissance et l'intelligence de la procédure de recherche. Les explications ont montré leur efficacité pour accélérer la résolution de certaines classes de problèmes difficiles, comme la planification hautement disjonctive [13]. Elles permettent également de traiter les problèmes dynamiques et surcontraints au sein d'un même outil. De plus, Ferrand et coll. [11] ont montré que les explications peuvent être utilisées pour déboguer un programme qui ne fournit pas une solution attendue.

Malgré leurs diverses applications, les explications restent peu utilisées. Peu de solveurs fournissent cette fonctionalité, comme le remarquent Douence et Jussien [8]. Le mécanisme de calcul des explications est intrusif : le code source du résolveur doit être instrumenté. Le principal obstacle semble être le cout de ce développement dans le cœur du solveur, mais aussi dans les algorithmes de filtrage des contraintes primitives. Douence et Jussien proposent d'utiliser la programmation orientée aspects pour éviter la modification directe du code source du résolveur. Quoi qu'il en soit, l'algorithme d'explication d'une contrainte donnée doit être conçu et développé. C'est une tâche aisée mais fastidieuse pour les contraintes simples, et bien plus ardue pour les contraintes globales. De plus, l'utilisateur doit expliquer les contraintes qu'il définit.

Certains solveurs utilisent un langage dédié pour définir leurs contraintes primitives. Ces définitions sont ensuite compilées ou interprétées dynamiquement. Les indexicaux sont un moyen pour décrire ces contraintes. Introduits par Van Hentenryck et coll. [19] dans le cadre de `cc(fd)`, il en existe deux mise en œuvres notables : les plateformes de programmation logique

GNU Prolog [7] et SICStus Prolog [3]. Les indexicaux sont à la fois déclaratif et opérationnels : ils spécifient le niveau de consistance à appliquer et offrent un moyen simple de réaliser le filtrage correspondant.

À notre connaissance, aucune implémentation des indexicaux ne fournit d'explications. Dans cet article, nous présentons d'abord un schéma trivial d'explication des contraintes indexicales. Puis nous proposons d'utiliser la structure du langage indexical pour calculer des explications plus précises. Ce schéma est systématique : il évite la conception manuelles d'algorithmes d'explication. Nous montrons également comment bénéficier de l'implémentation existante des indexicaux pour limiter les calculs. Les explications calculées ne sont pas minimales pour l'inclusion, mais sont précises pour un large sous-ensemble du langage indexical.

La section 2 motive notre travail en rappelant les principes des explications et leurs principales applications. La section 3 présente les contraintes indexicales que nous voulons expliquer et leurs implantation classiques. Nous exposons un schéma d'explication triviale en section 4. Un schéma plus précis est ensuite introduit en section 5. Les travaux connexes sont discutés en section 6, avant de conclure en 7.

2 Explications en PPC(DF)

Une explication est un sous-ensemble des contraintes du problème à résoudre suffit à justifier une action du solveur. Deux types d'explications sont largement utilisés : l'explication de retrait de valeurs et l'explications d'échec. La première explique pourquoi une valeur donnée, ou un ensemble de valeurs, doit être retiré du domaine d'une variable. La seconde est un ensemble conflictuel qui rend le problème insoluble. Cette section définit d'abord les explications en 2.1, puis la section 2.2 détaille quelques unes de leurs propriétés. Enfin, nous présentons quelques applications motivantes des explications en 2.3.

2.1 Définitions formelles

Considérons la définition traditionnelle d'un CSP (*constraint satisfaction problem*) : un ensemble $\mathcal{V}$ de variables, une fonction de domaine $\mathcal{D}$ qui associe à chaque variable x un ensemble fini de valeurs possibles $\mathcal{D}(v)$, et un ensemble $\mathcal{C}$ de contraintes. $\mathbb{D}$ est le domaine de calcul, ensemble des valeurs possibles pour toutes les variables. Les domaines initiaux des variables, quand différents de $\mathbb{D}$, sont inclus dans $\mathcal{C}$ sous forme de contraintes unaires.

Une contrainte est complètement définie par les réductions de domaines qu'elle opère. Dans cet article, nous utilisons le formalisme de Ferrand et coll. [10] : la propagation est modélisée par le calcul de la clôture descendante de l'ensemble des opérateurs de consistance locale associés à $\mathcal{C}$. Ce formalisme capture l'incomplétude des solveurs basés sur la propagation. Par souci de simplification, nous noterons $CL\downarrow(\mathcal{V},\mathcal{C})$ la fonction de domaine sur $\mathcal{V}$ résultant de la propagation des contraintes de $\mathcal{C}$.

Déf. 2.1 (Explication de retrait) *Une explication du retrait de la valeur d pour la variable v est un ensemble de contraintes $E \subseteq \mathcal{C}$ tel que :*

$$d \notin CL\downarrow(\mathcal{V}, E)(v)$$

En d'autres termes, la propagation des contraintes de E est suffit à justifier l'élimination de la valeur d de $\mathcal{D}(v)$.

Quand les explications sont calculées pendant la propagation, à la volée, l'explication du retrait de d de $\mathcal{D}(v)$ contient la contrainte qui a procédé au filtrage. Elle contient également les contraintes responsables des retraits de chaque support de la valeur et les explications du retrait de chaque support. Ce raisonnement conduit à une définition récursive élégante de l'explication, comme un arbre de preuve de retraits de valeurs [10] (*explications-arbres*). Ici, nous adoptons une formalisation plus simple où les explications sont des ensembles de contraintes (*explications-ensembles*) [15]).

Déf. 2.2 (Explication d'échec) *Une* explication d'échec *de $(\mathcal{V},\mathcal{C})$ est un ensemble de contraintes $E \subseteq \mathcal{C}$ tel que :*

$$\exists v \in \mathcal{V} \cdot CL\downarrow(\mathcal{V}, E)(v) = \emptyset.$$

En d'autres termes, le sous-ensemble de contraintes E est un conflit : il n'admet aucune solution.

Dans la suite, parlant indifféremment d'explication de retrait ou d'échec, nous noterons « explication d'une décision ».

2.2 Caractérisation des explications

Nous notons $\mathcal{E}$ l'ensemble des explications déjà calculées. $\mathcal{E}$ est une fonction partielle de $\mathcal{V}\times\mathbb{D}$ dans $\mathcal{P}(\mathcal{C})$, où $\mathcal{P}(\mathcal{C})$ est l'ensemble des parties de $\mathcal{C}$. Nous utilisons la notation ensembliste : $\mathcal{E} \in \mathcal{V}\times\mathcal{D}\times\mathcal{P}(\mathcal{C})$

Propriété 2.1 (Explication de tout retrait) *$\mathcal{E}$ est une fonction partielle dont le domaine est l'ensemble des retraits opérés :*

$$\forall v \in \mathcal{V} \cdot (\{d \in \mathbb{D} \mid \exists (v, d, E) \in \mathcal{E}\} = (\mathbb{D} - \mathcal{D}(v)))$$

Pour une décision du solveur donnée, plusieurs explications peuvent être trouvées. Premièrement, si E explique une décision, alors tout sur-ensemble E' de E dans $\mathcal{C}$ est également une explication de cette décision, par suite de la monotonicité des opérateurs de réduction. Ainsi, $\mathcal{C}$ est une explication triviale de toute décision du résolveur. Deuxièmement, plusieurs explications *diffèrentes* de la même décision peuvent coexister du moment qu'il existe deux explications $E_1 \subseteq \mathcal{C}$ et $E_2 \subseteq \mathcal{C}$ telles que $E_1 \cap E_2$ n'est pas une explication de cette décision.

Ågren [1] définit la *plus fine* explication d'une decision comme l'explication qui implique un minimum de variables. Un critère plus courant est la minimalité pour l'inclusion d'une explication [14]. Ici, comme formalisé par Ferrand et coll. [10], nous construisons les explications à partir d'explications existantes, ajoutant la contrainte déclenchant la décision :

Propriété 2.2 (Explication constructive) *Chaque explication de $E \in \mathcal{E}$ est constituée de la contrainte c et d'un ensemble (éventuellement vide) d'explications. Formellement, chaque (v, d, E) de $\mathcal{E}$ est tel que :*

$$E = \{c\} \cup \bigcup_{i=1}^{n} E_i$$

De plus, chaque E_i concerne des variables de $\mathrm{var}(c)$ *(à cause de localité du calcul) :*

$$\forall i \in [\![1, n]\!], \exists v_i \in \mathrm{var}(c), \exists d_i \in \mathbb{D} \cdot (v_i, d_i, E_i) \in \mathcal{E}$$

avec $\mathrm{var}(c)$ *l'ensemble des variables impliquées dans c.*

Par suite, nous considérons la minimalité locale pour l'inclusion [14]. Une explication est localement minimale pour l'inclusion si elle utilise un minimum d'explications antérieures E_i.

2.3 Applications motivant les explications

Nous sommes intéressés par quatre applications principales des explications, à savoir : le débogage, le traitement des problèmes dynamiques, la résolution de problèmes surcontraints et l'enrichissement de la procédure de recherche. Nous allons résumer ces quatre points afin de motiver notre contribution. Ne prétendant nullement prouver l'intérêt des explications, nous renvoyons le lecteur aux travaux cités pour des détails techniques ou quantitatifs.

Premièrement, les explications aident à déboguer un programme incorrect. Elles sont un moyen de trouver la raison d'une réponse manquante. Quand une solution attendue d'un problème n'est pas trouvée par la recherche, cela signifie qu'un ensemble de contraintes interdit cette solution. Une ou plusieurs de ces contraintes peuvent être fausses. Un solveur calculant des explications peut exhiber un ensemble conflictuel précis, c'est-à-dire un ensemble de contraintes justifiant le rejet de la solution. Le programmeur peut ainsi se concentrer sur ce conflit afin de trouver l'origine du bogue. Ferrand et coll. [11] ont adapté le débogage algorithmique en ce sens.

Deuxièmement, le même principe permet de gérer les problèmes surcontraints. Quand un problème correctement formulé et programmé est insoluble, des contraintes doivent être relâchées. Ouis et coll. [17] montrent comment les explications aident l'utilisateur à choisir les contraintes à relâcher. L'utilisation de leur outil par un utilisateur final ne nécessite aucune connaissance de la programmation par contraintes.

Troisièmement, les explications améliorent le traitement des problèmes dynamiques, où les contraintes à prendre en compte peuvent changer au cours de l'exécution. Certaines contraintes peuvent être relâchées, d'autres ajoutées. De tels problèmes se rencontrent dans des systèmes confrontés à une demande ou un environnement changeant. Les explications permettent au solveur de relâcher n'importe quelle contrainte tout en annulant ses effets passés, et ceci avec un minimum de calculs. De plus, l'état après le relâchement est souvent proche de l'ancien état. C'est ainsi qu'Elkhyari et coll. [9] gèrent différents problèmes dynamiques d'ordonnancement où les activités et ressources peuvent être ajoutées ou supprimées, et où les contraintes de précédences peuvent être modifiées. Leurs résultats sur de larges instances montrent l'efficacité de la méthode et la stabilité des ordonnancements proposés par l'outil. De la même manière, l'outil d'emploi du temps de Cambazard et coll. [2] prend en compte l'aspect dynamique inhérent aux ressources humaines.

Enfin, les explications libèrent le solveur du carcan du retour-arrière chronologique. Le solveur pouvant relâcher n'importe quelle contrainte, la procédure de recherche n'est plus nécessairement un parcours d'arbre. À chaque échec, un conflit peut être extrait. Les conflits extraits durant l'exécution peuvent être analysés afin d'adapter l'heuristique. Ici, les explications ont un double intérêt. Elles permettent d'apprendre la cause des échecs et d'effectuer un *backtracking* dynamique. Cette intégration de la propagation de contraintes dans le « *backtracking* intelligent » évite de répéter les mêmes impasses. La plus connue des techiques de ce type est MAC-DBT (*Maintaining Arc-Consistency with Dynamic Backtracking*), proposée par Jussien et coll. [15].

```
plus(X,Y,Z) +:
    X in min(Z) - max(Y) .. max(Z) - min(Y),
    Y in min(Z) - max(X) .. max(Z) - min(X),
    Z in min(X) + min(Y) .. max(X) + max(Y).
```

FIG. 1 – Spécification de la consistance de bornes pour la contrainte $X + Y = Z$ (syntaxe SICStus Prolog)

```
plus(X,Y,Z) +:
    X in dom(Z) - dom(Y),
    Y in dom(Z) - dom(X),
    Z in dom(X) + dom(Y).
```

FIG. 2 – Spécification de la consistance d'hyper-arc pour $X + Y = Z$

3 Contraintes indexicales

Nous rappelons ici les bases des indexicaux : leur expressivité (sec. 3.1), syntaxe (sec. 3.2), ainsi que leurs schémas de compilation et de propagation (sec. 3.3). La section 3.4 présente quelques aspects techniques de leurs implémentations. De plus amples détails sont donnés par Carlsson et coll. [4] pour SICStus Prolog et Codognet et Diaz [5] pour GNU Prolog.initial paper by Van

3.1 Expressivité des indexicaux

Un indexical est une contrainte de la forme V `in` R, où V est une variable de domaine et R une expression de champ, représentant un ensemble de valeurs[1]. L'expression R peut inclure des informations sur les domaines d'autres variables, telles leurs bornes, leur ensemble de valeurs, leur valeur finale ou leur cardinalité (noté respectivement `min`, `max`, `dom`, `val`, et `card`). Ainsi, le domaine de V doit être un sous-ensemble de $R(\mathcal{D})$, évaluation de R dans l'état des domaines $\mathcal{D}$. L'indexical spécifie donc un ensemble de valeurs valides pour V. Comme expliqué en 3.3, un indexical peut être compilé en un algorithme de filtrage. De nombreuses contraintes standards (arithmetiques, booléennes ou symboliques) peuvent être exprimées par la conjonction de plusieurs indexicaux, à raison d'un indexical par variable. En SICStus, les indexicaux sont également utilisés pour spécifier les tests d'implication et de satisfiabilité.

Les indexicaux peuvent coder plusieurs niveaux de consistance. Par exemple, la figure 1 présente comment spécifier la consistance de bornes pour la contrainte arithmetique `X+Y=Z`, grâce à des intervalles délimités par les bornes des variables (`A..B` est l'intervalle entre `A` et `B`). La consistance d'hyper-arc peut être spécifiée en utilisant l'addition et la soustraction point-à-point, comme le montre la Figure 2.

L'intérêt des indexicaux est triple : leur langage est simple à maîtriser, ils sont simples à concevoir et à comprendre, et ils sont des spécifications exécutables. De plus, il est possible de les implémenter très efficacement. Même s'ils manquent d'expressivité, incapables d'exprimer les algorithmes complexes des contraintes globales, ils constituent un outil puissant pour décrire un large éventail de contraintes simples. Pour preuve, la totalité de la bibliothèque *domaines finis* de GNU Prolog est implémentée en utilisant les indexicaux. Les indexicaux permettent aussi de développer rapidement des résolveurs spécialisés.

3.2 Une syntaxe unifiée des indexicaux

L'expressivité et la syntaxe exactes des indexicaux varient légèrement suivant les implémentations. Par exemple, GNU Prolog propose la multiplication point-à-point entre deux champs de valeurs [6, Sec. 7.3.1] tandis que SICStus permet des évaluations conditionnelles. Ici, nous considérons l'union des constructions de ces deux implémentations, adoptant si possible la syntaxe de SICStus, et celle de GNU Prolog pour les constructions qui lui sont spécifiques.

Dans un indexical V `in` R, l'expression de champ R est une construction du langage décrit en table 1. Ce langage a deux niveaux d'expression : les *termes* (valeurs individuelles dans $\mathbb{D}$), qui peuvent être combinés en *champs* (sous-ensembles de $\mathbb{D}$). Les valeurs de champs élémentaires sont soit le domaine d'une variable, une énumération de termes ou un interval entre deux termes. Les champs peuvent être combinés en utilisant des opérations ensemblistes (union, intersection, complément) ou l'arithmétique point-à-point ($\mathcal{P}(\mathbb{D})^2 \rightarrow \mathcal{P}(\mathbb{D})$). Les opérations arithmétiques peuvent également combiner un champ et un terme ($\mathcal{P}(\mathbb{D}) \times \mathbb{D} \rightarrow \mathcal{P}(\mathbb{D})$). Les termes élémentaires sont : une borne inférieure ou supérieure, la cardinalité ou la valeur instanciée d'un domaine, ou bien une constante de $\mathbb{D}$. Les termes peuvent être combinés arithmétiquement. La sémantique dénotationnelle de ces constructions est décrite dans [5]. La table 1 rappelle quatre fonctionnalités avancées. Tout d'abord, l'appel d'une fonction C (aspect non documenté de GNU Prolog), avec des termes ou champs pour paramètres. La fonction f doit renvoyer un champ de valeurs. C'est ainsi qu'est implémentée la contrainte `element` en GNU Prolog. Ensuite, SICStus propose deux constructions d'évaluation conditionnelle où le champ évalué dépend de la valeur d'une autre expression (? et `switch`). En-

[1] Nous traduirons le terme anglais *range* par « champ » dans ce contexte, comme *champ de valeurs possibles*.

Construction type	Syntaxe	Description
Valeurs de champs	$\texttt{dom}(V)$	état courant de $\mathcal{D}(V)$
	$\{T_1, \ldots, T_k\}$	ensemble de termes *(S)*
	T_1 .. T_2	intervalle $[\![T_1, T_2]\!] \subseteq \mathbb{D}$
Composition de champs	$R_1 \,/\backslash\, R_2$	intersection
	$R_1 \,\backslash/\, R_2$	union
	$R_1 + R_2$	addition pàp.
	$R_1 - R_2$	soustraction pàp.
$R = R_1$ op R_2	$R_1 * R_2$	multiplication pàp. *(G)*
	R_1 / R_2	division pàp. *(G)*
	R_1 `mod` R_2	modulo pàp. *(S)*
Op. sur un champ	$\backslash\, R_1$	complément
	$-\, R_1$	négation pàp. *(S)*
Composer un champ et un terme	$R_1 + T_1$	addition pàp.
	$R_1 - T_1$	soustraction pàp.
	$R_1 * T_1$	multiplication pàp.
	$R_1 /< T_1$	division entière pàp. (défaut) *(G)*
$R = R_1$ op T_1	$R_1 /> T_1$	division entière pàp. (excès) *(G)*
Valeurs de termes	$\texttt{min}(V)$	borne inf. de $\mathcal{D}(V)$
	$\texttt{max}(V)$	borne sup. de $\mathcal{D}(V)$
	$\texttt{card}(V)$	cardinalité de $\mathcal{D}(V)$ *(S)*
	$\texttt{val}(V)$	valeur finale de $\mathcal{D}(V)$
	n	entier, $n \in \mathbb{D}$
	`inf`	borne inf. de $\mathbb{D}$
	`sup`	borne sup. de $\mathbb{D}$
Composition de termes	$-\, T_1$	négation *(S)*
	$T_1 + T_2$	addition entière
	$T_1 - T_2$	soustraction entière
	$T_1 * T_2$	multiplication entière
$T = T_1$ op T_2	$T_1 /< T_2$	division entière (défaut)
	$T_1 /> T_2$	division entière (excès)
	T_1 `mod` T_2	modulo entier
Fonctions avancées	$f(a_1, \ldots, a_k)$	appel de fonction, les a_i sont des termes ou champs *(G)*
	R_1 ? R_2	égal à R_2 si $R_1 \neq \emptyset$, $\emptyset$ sinon *(S)*
$R = f(\ldots)$	$\texttt{switch}(T, L)$	égal à $L(T)$ *(S)*
	$\texttt{unionof}(x, R_1, R_2) = \cup_{d \in R_1}(R_2[x \leftarrow d])$	*(S)*

Key : R, R_1 et R_2 sont des champs.
T, T_1 et T_2 sont des termes.
V est une variable de var(V *in* R).
x est une variable locale
$R[x \leftarrow d]$ est la substitution de x par d dans R
L liste de (i, R_i) avec $i = j \Rightarrow R_i = R_j$, et $L(i) = R_i$
(G) signale une spécificité GNU Prolog.
(S) signale une spécificité SICStus Prolog
pàp. signifie « point-à-point »

TAB. 1 – Syntaxe unifiée des contraintes indexicales

fin, « `union` » calcule, en SICStus, l'union des évaluations de R_2 pendant qu'une variable locale x parcourt les valeurs de R_1 [3, Sec. 34.10.2].

3.3 Compilation et Propagation des indexicaux

Les indexicaux ont un shéma de compilation et de propagation simple. Chaque indexical V `in` R est compilé en un moyen d'évaluer $R(\mathcal{D})$ (cf. 3.4). Le réveil d'un indexical provoque cette évaluation, puis le calcul de l'intersection entre l'ensemble résultant et le domaine courant de V. Cette intersection $R(\mathcal{D}) \cap \mathcal{D}(V)$ devient le nouveau domaine de V. Un échec est détecté lorsque cette intersection est vide.

Le réveil d'un indexical est provoqué par toute mise à jour d'un des éléments-domaines impliqués dans son champ R. Par exemple, un indexical dont le champ se réfère à $\texttt{min}(X)$, $\texttt{max}(Y)$, et $\texttt{dom}(Z)$ sera réveillé à chaque augmentation de la borne inférieure de $\mathcal{D}(X)$, diminution de $\max(Y)$ et tout retrait dans $\mathcal{D}(Z)$. La correction de ce schéma de propagation nécessite la *monotonicité* des champs des indexicaux.

Propriété 3.1 (Monotonicité) *Un champ R est monotone ssi $R(\mathcal{D}) \subseteq R(\mathcal{D}')$, pour toute paire de fonctions domaine $(\mathcal{D}, \mathcal{D}')$ sur $\mathcal{V}$ telle que $\forall v \in \text{var}(R) \cdot \mathcal{D}(v) \subseteq \mathcal{D}'(v)$. Lorsque R est non-monotone, l'évaluation de $R(\mathcal{D})$ est retardée jusqu'à ce que $\mathcal{D}$ assure cette monotonicité.*

La stratégie exacte dépend de l'implémentation [4, 5]. Ici, nous supposerons qu'un indexical ne peut effectuer de retrait de valeurs que quand sa monotonicité est certaine. Cette propriété est appelée *monotonicité pour la propagation.*

3.4 Évaluation des indexicaux

En SICStus, le champ d'un indexical est compilé en notation post-ordre puis en un code pour machine à pile. Quand un indexical V `in` R est reveillé, la machine à pile évalue $R(\mathcal{D})$, suivant l'état courant des domaines. Le résolveur calcule alors la position relative de $R(\mathcal{D})$ par rapport à $\mathcal{D}(V)$, l'intersection n'étant pas directement calculée. Si les deux ensembles se révèlent disjoints, un échec est déclenché. Sinon, les valeurs inconsistantes sont retirées de $\mathcal{D}(V)$. En 5, nous proposons de tirer parti de cette implémentation : évaluation de $R(\mathcal{D})$ par une machine à pile et comparaison avec $\mathcal{D}(V)$ ensuite.

En GNU Prolog, chaque indexical est compilé en une fonction C calculant $R(\mathcal{D})$. Lors du réveil de l'indexical, cette fonction est appelée. Si le champ résultat est non vide, l'intersection $R(\mathcal{D}) \cap \mathcal{D}(V)$ est calculée par un *et* bit-à-bit, et affectée à $\mathcal{D}(V)$. Il n'y a donc pas de

comparaison directe entre $R(\mathcal{D})$ et $\mathcal{D}(V)$. Cette compilation native est un des points clés de GNU Prolog et permet une évaluation plus rapide (cf. la comparaison de [4]). Les contraintes primitives bénéficient ainsi des optimisations du compilateur C, au prix d'un code plus volumineux (par rapport au *bytecode* correspondant). Un autre inconvénient est la recompilation obligatoire du solveur FD lorsque l'utilisateur veut définir une nouvelle contrainte primitive.

4 Explications triviales des indexicaux

Comme nous l'avons détaillé en 3, le langage indexical se réfère aux domaines de cinq manières, notées : $\texttt{min}(V)$, $\texttt{max}(V)$, $\texttt{dom}(V)$, $\texttt{val}(V)$ and $\texttt{card}(V)$. Appelons *éléments-domaines* ces références aux (parties de) domaines. L'évaluation d'une expression-champ dépend des éléments-domaines qu'elle contient. Plus précisément, R est une fonction de ses éléments-domaines. L'absence d'une valeur dans $R(\mathcal{D})$ peut s'expliquer par l'état courant des éléments-domaines présents dans R. Nous définissons quelques raccourcis pour expliquer l'état courant des éléments-domaines : $\mathcal{E}_{\min}(V)$ est l'explication de la valeur de $\min(\mathcal{D}(V))$, $\mathcal{E}_{\max}(V)$ est l'explication de la valeur de $\max(\mathcal{D}(V))$, et $\mathcal{E}_{\text{dom}}(V)$ est l'explication de l'ensemble de $\mathcal{D}(V)$. Leur définition est simple :

$$\mathcal{E}_{\min}(V) = \bigcup_{d\in\mathbb{D}\,\wedge\, d<\min(\mathcal{D}(V))} \mathcal{E}(V,d)$$
$$\mathcal{E}_{\max}(V) = \bigcup_{d\in\mathbb{D}\,\wedge\, d>\max(\mathcal{D}(V))} \mathcal{E}(V,d)$$
$$\mathcal{E}_{\text{dom}}(V) = \bigcup_{d\in\mathbb{D}\,\wedge\, d\notin\mathcal{D}(V)} \mathcal{E}(V,d)$$

La cardinalité de $\mathcal{D}(V)$, à l'instar de sa valeur instanciée, dépend de tous les retraits effectués dans $\mathcal{D}(V)$: $\mathcal{E}_{\text{val}}(V) = \mathcal{E}_{\text{card}}(V) = \mathcal{E}_{\text{dom}}(V)$.

En évaluant $c \equiv V$ **in** R, nous expliquons le retrait de $\mathcal{D}(V) - R(\mathcal{D})$ de $\mathcal{D}(V)$ par :

$$\mathcal{T}(V,R,\mathcal{D}) = \{c\} \cup \left(\bigcup_{v\in\text{var}(R)} \left(\cup_{e\in\text{elements}(R,v)} \mathcal{E}_e(v)\right) \right)$$

où $\text{elements}(R,v)$ est l'ensemble des éléments-domaines se rapportant à la variable v dans R.

Théorème 4.1 (Correction du schéma trivial) *Si un indexical V $\texttt{in}$ R est monotone pour la propagation (cf. 3.3), alors $\mathcal{T}(V,R,\mathcal{D})$ est une explication correcte du retrait de valeurs dû à cet indexical.*

Ce résultat peut être prouvé par induction forte sur la taille de $\mathcal{T}(V,R,\mathcal{D})$. Le cas de base est une explication singleton, signifiant que V $\texttt{in}$ R est unaire, ou que tout les éléments-domaines de R ont leur valeur nominale (i.e. $\min(\mathbb{D}) \in \mathcal{D}(X)$ pour $\texttt{min}(X)$, $\max(\mathbb{D}) \in \mathcal{D}(X)$ pour $\texttt{max}(X)$, $\mathcal{D}(X) = \mathbb{D}$ sinon). L'étape d'induction est évidente pour les éléments du type `dom`, `val` et `card` (d'après la propriété 2.2). Enfin, notons que la monotonicité de R implique la monotonicité pour l'augmentation de chacun des éléments `min`, et pour la diminution des éléments de type `max`.

$R(\mathcal{D}) = \emptyset$ provoquant un échec, les explications triviales peuvent être utilisées comme explications d'échec (évident d'après déf. 2.2) :

Théorème 4.2 (Explication d'échec) *Si un indexical V $\texttt{in}$ R est évalué et que $R(\mathcal{D}) \cap \mathcal{D}(V) = \emptyset$, alors $\mathcal{E}_{\text{dom}}(V) \cup \mathcal{T}(V,R,\mathcal{D})$ est une explication d'échec pour $(\mathcal{V},\mathcal{C})$.*

Ces explications ne sont pas minimales, mais elles sont faciles à intégrer dans le processus d'évaluation. L'ensemble $\cup_{v\in\text{var}(R)} elements(R,v)$ étant connu au moment de la compilation, le calcul de l'explication peut être intégré statiquement dans l'évaluation de l'indexical. Le seul calcul supplémentaire est dû à l'opération $\cup$ sur $\mathcal{P}(\mathcal{C})^2$. La section 5 propose un schéma donnant d'autres explications non minimales, mais plus précises néanmoins.

5 Explications précises aux bornes

Nous proposons maintenant un schéma systématique de calcul d'explications, plus précis que les explications triviales. Ce schéma tire parti de la structure de l'indexical exécuté et repose sur une partition des valeurs à retirer. Cette partition privilégie les bornes du domaine. Après avoir motivé ce choix (sec. 5.1), nous définissons les explications calculées (sec. 5.2) et présentons le schéma en regard de la syntaxe des indexicaux (sec. 5.3).

5.1 Bornes des domaines et Propagation

Les bornes des domaines ont souvent un rôle particulier dans les algorithmes de filtrage. Nombre d'entre eux sont basés sur la *consistance de bornes*, qui approxime chaque domaine $\mathcal{D}(v)$ par son plus petit intervalle englobant, c'est-à-dire $[\![\min(\mathcal{D}(v)), \max(\mathcal{D}(v))]\!]$. Cette technique fait peu de cas des « trous » des domaines et offre un niveau de consistance plus faible que l'hyper-arc consistance. Mais sa rapidité de calcul en fait souvent un bon compromis élagage/vitesse.

Comme nous le détaillerons, pour beaucoup des constructions du langage indexical, la mise à jour des bornes d'un domaine est la conséquence de mises à jour de bornes d'un ou plusieurs autres domaines. Cela est également valable pour certaines constructions accomplissant l'hyper-arc consistance.

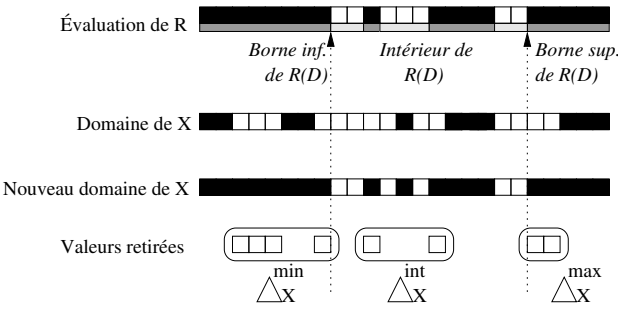

FIG. 3 – Partition des valeurs retirées basée sur les bornes de $R(\mathcal{D})$

5.2 Explications particulièrement précises aux bornes

Soit Δ_V l'ensemble des valeurs à retirer de $\mathcal{D}(V)$ dû à l'indexical V `in` R, $\Delta_V = \mathcal{D}(V) - R(\mathcal{D})$. Nous partitionnons Δ_V en trois ensembles (éventuellement vides) :

1. $\Delta_V^{\min} = \{\delta \in \Delta_V \mid \delta < \min(R(\mathcal{D}))\}$;
2. $\Delta_V^{\max} = \{\delta \in \Delta_V \mid \delta > \max(R(\mathcal{D}))\}$;
3. $\Delta_V^{\text{int}} = \Delta_V - \left(\Delta_V^{\max} \cup \Delta_V^{\min}\right)$.

La figure 3 illustre ces définitions. Les domaines sont un ensemble de carrés blancs, les parties noires étant les valeurs manquantes. $\Delta_V^{\min}$ (resp. $\Delta_V^{\max}$) est l'ensemble des valeurs dont le retrait est dû à la borne inférieure (resp. supérieure) de $R(\mathcal{D})$. Δ_V^{int} est l'ensemble des valeurs retirées entre ces bornes. Trivialement, $\Delta_V^{\min}$, $\Delta_V^{\max}$, et Δ_V^{int} est une partition de Δ_V.

Nous proposons d'expliquer chaque partie Δ_V^i séparément. Dans la suite, nous cherchons à expliquer les bornes du champ de valeurs considéré, adoptant l'extension terminologique suivante :

Déf. 5.1 (Expliquer une expression indexicale) *$E \subseteq \mathcal{C}$ est une* explication *de la propriété P d'une expression (champ ou terme) e du langage indexical ssi $e(CL{\downarrow}(\mathcal{V}, E))$ satisfait P.*

Pour une expression de champ R_1, $\mathcal{E}_{\min}(R_1)$ est l'explication de la valeur de $\min(R_1(\mathcal{D}))$, $\mathcal{E}_{\max}(R_1)$ explique $\max(R_1(\mathcal{D}))$. $\mathcal{E}_{\text{int}}(R_1)$ explique les « trous » de $R_1(\mathcal{D})$. Pour une expression terme T_1, $\mathcal{E}_{\text{val}}(T_1)$ explique la valeur de $T_1(\mathcal{D})$.

Propriété 5.1 (Expliquer la partition Δ_V) *$\{c\} \cup \mathcal{E}_{\min}(R)$ (resp. $\{c\} \cup \mathcal{E}_{\max}(R)$) est une explication du retrait de $\Delta_V^{\min}$ (resp.$\Delta_V^{\max}$) de $\mathcal{D}(V)$, avec $c \equiv X$ `in` R. $\{c\} \cup \mathcal{E}_{\text{int}}(R)$ est une explication du retrait de Δ_V^{int} de $\mathcal{D}(V)$.*

Expression e	$\mathcal{E}_{\text{val}}(e)$
`min`(V)	$\mathcal{E}_{\min}(V)$
`max`(V)	$\mathcal{E}_{\max}(V)$
`card`(V)	$\mathcal{E}_{\text{dom}}(V)$
`val`(V)	$\mathcal{E}_{\text{dom}}(V)$
n, `inf`, `sup`	$\emptyset$
$-T_1$	$\mathcal{E}_{\text{val}}(T_1)$
T_1 [+,−,/<,/>] T_2	$\mathcal{E}_{\text{val}}(T_1) \cup \mathcal{E}_{\text{val}}(T_2)$
$\mathcal{E}_{\text{val}}(T_1 * T_2) =$	$\begin{cases} \mathcal{E}_{\text{val}}(T_1) \text{ si } T_1 = 0 \\ \mathcal{E}_{\text{val}}(T_2) \text{ si } T_2 = 0 \wedge T_1 \neq 0 \\ \mathcal{E}_{\text{val}}(T_1) \cup \mathcal{E}_{\text{val}}(T_2) \text{ sinon} \end{cases}$
$\mathcal{E}_{\text{val}}(T_1$ `mod` $T_2) =$	$\begin{cases} \mathcal{E}_{\text{val}}(T_1) \text{ si } T_1 = 0 \\ \mathcal{E}_{\text{val}}(T_2) \text{ si } T_2 = 1 \wedge T_1 \neq 0 \\ \mathcal{E}_{\text{val}}(T_1) \cup \mathcal{E}_{\text{val}}(T_2) \text{ sinon} \end{cases}$

TAB. 2 – Explications précises aux bornes pour les termes

5.3 Schéma de calcul

Pour un indexical donné V `in` R, le calcul de $\mathcal{E}_{\min}(R)$, $\mathcal{E}_{\max}(R)$ et $\mathcal{E}_{\text{int}}(R)$ est compositionnel, suivant la structure de l'expression R : les explications sont des attributs synthétisés de l'arbre syntaxique associé à R. Pour chaque construction de la table 1, nous spécifions comment synthétiser les explications à partir de celles des sous-expressions. La table 2 présente le schéma de calcul pour le niveau des valeurs termes, reprenant les notations de 4. Le résultat d'une opération binaire est expliqué par l'union des explications des deux opérandes. Quand il existe un élément absorbant (tel 0 pour $\times$ et comme opérande gauche de `mod`, 1 comme opérande droit de `mod`), le terme prenant pour valeur l'élément absorbant suffit à expliquer le résultat (le même raisonnement s'applique à l'opérande gauche de /< et /<).

La table 3 présente le schéma de calcul pour les valeurs de champs. Par manque de place, nous n'en détaillons qu'un échantillon représentatif. Le même principe général s'applique aux constructions manquantes : nous tentons d'utiliser les explications des bornes des opérandes pour expliquer les bornes du résultat.

Certaines constructions perdent la précision aux bornes de nos explications. Ainsi, pour $\{T_1, \ldots, T_k\}$ les bornes du résultat peuvent être dues à n'importe quel T_i. Rafiner le calcul nécessiterait plus d'informations sur les T_i. D'autres constructions conservent toujours la précision aux bornes, telles $R_1 + R_2$. Pour /\, les bornes de $R_1 \cap R_2$ peuvent parfois être expliquées par les bornes d'un des deux opérandes. Afin de garder une complexité raisonnable, $\mathcal{E}_{\text{int}}(e)$ est l'union des explications des deux champs, ce qui manque souvent de précision. Cela évite que la complexité ne dépende de

Expression e	$\mathcal{E}_{\min}(e)$	$\mathcal{E}_{\max}(e)$	$\mathcal{E}_{\text{int}}(e)$
$\texttt{dom}(V)$	$\mathcal{E}_{\min}(V)$	$\mathcal{E}_{\max}(V)$	$\mathcal{E}_{\text{int}}(V)$
T_1 .. T_2	$\mathcal{E}_{\text{val}}(T_1)$	$\mathcal{E}_{\text{val}}(T_2)$	$\emptyset$
$\{T_1, \ldots, T_k\}$	$\bigcup_{i \in [\![1,k]\!]} \mathcal{E}_{\text{val}}(T_i)$		
$-\ R_1$	$\mathcal{E}_{\max}(R_1)$	$\mathcal{E}_{\min}(R_1)$	$\mathcal{E}_{\text{int}}(R_1)$
$R_1\ +\ R_2$	$\mathcal{E}_{\min}(R_1) \cup \mathcal{E}_{\min}(R_2)$	$\mathcal{E}_{\max}(R_1) \cup \mathcal{E}_{\max}(R_2)$	$\mathcal{E}_{\text{dom}}(R_1) \cup \mathcal{E}_{\text{dom}}(R_2)$
$R_1\ -\ R_2$	$\mathcal{E}_{\min}(R_1) \cup \mathcal{E}_{\max}(R_2)$	$\mathcal{E}_{\max}(R_1) \cup \mathcal{E}_{\min}(R_2)$	$\mathcal{E}_{\text{dom}}(R_1) \cup \mathcal{E}_{\text{dom}}(R_2)$
$R_1\ /\backslash\ R_2$	$\mathcal{E}_{\min}(e) = \begin{cases} \mathcal{E}_{\min}(R_1) & \text{si } \min(R_1) \in R_2 \\ \mathcal{E}_{\min}(R_2) & \text{si } \min(R_2) \in R_1 \\ \mathcal{E}_{\text{dom}}(R_1) \cup \mathcal{E}_{\text{dom}}(R_2) & \text{sinon} \end{cases}$ $\mathcal{E}_{\max}(e) = \begin{cases} \mathcal{E}_{\max}(R_1) & \text{si } \max(R_1) \in R_2 \\ \mathcal{E}_{\max}(R_2) & \text{si } \max(R_2) \in R_1 \\ \mathcal{E}_{\text{dom}}(R_1) \cup \mathcal{E}_{\text{dom}}(R_2) & \text{sinon} \end{cases}$ $\mathcal{E}_{\text{int}}(e) = \mathcal{E}_{\text{int}}(R_1)$		
$R_1\ ?\ R_2$	$\mathcal{E}_{\min}(e) = \begin{cases} \mathcal{E}_{\text{dom}}(R_1) & \text{si } R_1 = \emptyset \\ \mathcal{E}_{\min}(R_2) & \text{sinon} \end{cases}$ $\mathcal{E}_{\max}(e) = \begin{cases} \mathcal{E}_{\text{dom}}(R_1) & \text{si } R_1 = \emptyset \\ \mathcal{E}_{\max}(R_2) & \text{sinon} \end{cases}$ $\mathcal{E}_{\text{int}}(e) = \begin{cases} \emptyset & \text{si } R_1 = \emptyset \\ \mathcal{E}_{\text{int}}(R_2) & \text{sinon} \end{cases}$		

TAB. 3 – Explications particulièrement précises aux bornes d'expressions de champs (échantillon représentatif)

la cardinalité des champs, pour l'arithmétique point-à-point notamment.

La correction de ce schéma se prouve par induction forte sur la taille des explications, comme en 4. Ici, le cas de base et l'étape d'induction doivent être prouvés par induction sur la structure de R, nécessitant un cas pour chaque construction du langage.

Quand les indexicaux sont évalués par une machine à pile, comme pour SICStus Prolog, la mise en œuvre de ce schéma est directe : une pile d'explications est ajoutée à la machine. À chaque empilement d'une valeur terme ou champ $e(\mathcal{D})$, un triplet d'explications $(\mathcal{E}_{\min}(e)(\mathcal{D}), \mathcal{E}_{\max}(e)(\mathcal{D}), \mathcal{E}_{\text{int}}(e)(\mathcal{D}))$ est empilé sur la pile d'explications. Quand la machine applique une opération, les explications des opérandes sont dépilées, celles du résultats sont calculées suivant l'opération appliquée et empilées.

5.4 Variante paresseuse

Au moins la moitié des évaluations d'indexicaux sont inutiles, le résultat $R(\mathcal{D})$ étant un surensemble de $\mathcal{D}(V)$ [5].

L'implantation proposée ci-dessus calcule trois explications pour chaque construction. Si, au final, aucune réduction de domaine n'est effectuée, ces calculs se révèlent inutiles. Et si une réduction de domaine est nécessaire, certaines des explications calculées peuvent demeurer inutiles. Nous proposons d'adapter le schéma pour calculer paresseusement les explications utiles, après s'être assuré que $\Delta_V^{\min}$, $\Delta_V^{\max}$ ou Δ_V^{int} soient non vides. Premièrement, une instrumentation légère de la fonction d'évaluation garde trace des résultats intermédiaires. Ensuite, si nécessaire, les explications utiles sont calculées grâce à cette trace.

Instrumentation de l'évaluation. Tandis que la machine à pile évalue $R(\mathcal{D})$, les résultats intermédiaires (termes ou champs) sont stockés dans un arbre. Chaque opération de la machine crée un nœud de cet arbre, contenant le *bytecode* de l'opération et ses résultats intermédiaires. Cet arbre a donc la même structure que l'indexical. Recyclant des résultats intermédiaires, le seul surcout est dû à la construction de l'abre. La figure 4 présente cette procédure instrumentée. I est le flot d'instructions de la machine, S la pile d'évaluation, **E** un ensemble ordonné de nœud représentant la trace des opérations déjà exécutées. **new_leaf** insère une nouvelle feuille dans **E**. **new_node** extrait les arity(i) derniers nœuds de **E** et insère un nouveau nœud, père des nœuds extraits. La fonction est principalement une boucle exécutant les instructions de la machine à pile. À la fin de la fonction, **E** contient la racine de l'arbre et `R` contient $R(\mathcal{D})$.

Calcul à rebours Aussitôt l'évaluation de $R(\mathcal{D})$ terminée, le résolveur calcule $\Delta_V^{\min}$, $\Delta_V^{\max}$ et Δ_V^{int}. Si nécessaire, nous calculons récursivement les explications nécessaires. Pour cela, nous utilisons les résultats intermédiaires stockés dans l'arbre. Ainsi, nous calculons seulement les explications nécessaires et évitons

```
function evaluation(I, 𝒟), returns R,E
  S ← ∅
  E ← ∅
  while ⟨I⟩ ≠ ∅ do
    i ← dequeue(I)
    switch i
      load_min:
        X ← deref_var(dequeue(I))
        push(S, min(X))
        new_leaf(E, ℰ_min(X), int(min(X)))
      load_max:
        X ← deref_var(dequeue(I))
        push(S, max(X))
        new_leaf(E, ℰ_max(X), int(max(X)))
      load_val:
        X ← deref_var(dequeue(I))
        push(S, val(X))
        new_leaf(E, ℰ_dom(X), int(v_x))
      load_dom:
        X ← deref_var(dequeue(I))
        push(S, 𝒟(X))
        new_leaf(E, ℰ_dom(X), set(𝒟(X)))
      load_int:
        t ← deref_int(dequeue(I))
        push(S, t)
        new_leaf(E, ∅, int(t))
      load_set:
        r ← deref_set(dequeue(I))
        push(S, r)
        new_leaf(E, ∅, set(r))
      default:
        R ← compute(S, i)
        push(S,R)
        new_node(E, i, arity(i), R)
    end switch
  end while
  R ← pop(S)
end function
```

FIG. 4 – Procédure d'évaluation instrumentée

les parties non pertinentes de l'expression R.

6 Travaux connexes

Ågren [1] a conçu et mis en œuvre un mécanisme d'explication pour SICStus Prolog. Ce travail est l'une des premières grandes tentatives d'explication des contraintes globales d'un véritable résolveur. Ågren a également mis au point des algorithmes pour calculer les explications les plus fines de quelques contraintes globales classiques, telles que `all_distinct` (réalisation de l'hyperarc-consistance pour la différence deux à deux d'une liste de variables), `circuit` (contrainte qu'un ensemble de sommets d'un graphe forme un circuit hamiltonien). Malheureusement, ce mécanisme fait partie d'une branche « recherche » de SICStus Prolog, non distribuée à ce jour. À notre connaissance, les raisons qui expliquent cet abandon sont les couts d'élaboration de ce mécanisme ainsi que la nécessité d'expliquer les contraintes indexicales.

Ferrand et coll. [12] ont proposé un procédé générique pour calculer les explications à partir d'une trace précise de l'exécution du solveur. Ce procédé n'est pas intrusif, tant que le solveur fournit un tel traceur. On peut citer quatre de ces traceurs, à savoir pour GNU-Prolog [16], Choco, PaLM (qui trace déjà des explications) et CHIP. Si l'on considère le cas de GNU-Prolog, les explications issues de cet algorithme seraient moins précises que les nôtres. Comme la trace ne dit pas quelles informations de domaine l'indexical évalué utilise, leurs explications doivent prendre en considération toutes les réductions de domaine des variables impliquées dans la contrainte, depuis le début de l'exécution. Ce calcul ne prend donc en compte que les explications $\mathcal{E}_{\text{dom}}(X)$ déjà définies dans notre schéma trivial (Sec. 4). À ce jour, aucune implémentation de cet algorithme n'est disponible.

7 Conclusion

Cet article a présenté un schéma de production automatique d'explications pour les solveurs basés sur les indexicaux. Ce schéma utilise la structure des indexicaux pour expliquer leurs filtrages. Il propose un compromis entre précision et cout de l'explication, insistant sur l'explications des bornes des domaines.

Notre schéma permet l'explication automatique de plusieurs résolveurs existants. Il peut également traiter les contraintes que l'utilisateur définit par indexicaux. Les contraintes que nous gérons sont limitées par l'expressivité des indexicaux, excluant les contraintes globales. Cependant, l'utilisation d'un langage spécifique pour décrire les contraintes globales, tel les propriétés de graphes de contraintes, permet d'imaginer un schéma aussi systématique. De premiers résultats en ce sens ont été obtenus récemment par Rochart et Jussien [18].

Remerciements L'auteur tient à remercier l'équipe de SICStus Prolog pour son aide, notamment Mats Carlsson et Per Mildner. Certaines idées ont émergées suite à d'agréables discussions avec Narendra Jussien, Pierre Deransart, Guillaume Rochart et Alexandre Tessier. Un remerciement tout particulier pour Mireille Ducassé, dont l'encre a recouvert un brouillon de cet article, et pour Grégory Lafay, qui a exercé ses talents de traducteur sur une partie de la version finale.

Références

[1] M. ÅGREN – « Tracing and explaining the execution of clp(fd) programs in sicstus prolog »,

Technical Report T2002 :10, Swedish Institute for Computer Science, Uppsala, juil. 2002.

[2] H. Cambazard, F. Demazeau, N. Jussien & P. David – « Interactively solving school timetabling problems using extensions of constraint programming », in *Practice and Theory of Automated Timetabling (PATAT 2004)* (Pittsburgh, PA USA) (M. Trick & E. K. Burke, éds.), aout 2004, p. 107–124.

[3] M. Carlsson et al. – *SICStus Prolog 3.12.0 user's manual*, SICS Research Report, nov. 2004.

[4] M. Carlsson, G. Ottosson & B. Carlson – « An open-ended finite domain constraint solver », in *Proc. of the 9th Int. Symp. on Programming Languages : Implementations, Logics and Programs* (H. Glaser, P. Hartel & H. Kuchen, éds.), LNCS, no. 1292, Springer Verlag, 1997, p. 191–206.

[5] P. Codognet & D. Diaz – « Compiling constraints in CLP(FD) », *Journal of Logic Programming* **27** (1996), no. 3, p. 185–226.

[6] D. Diaz – « Étude de la compilation des langages logiques de programmation par contraintes sur les domaines finis : le système clp(fd) », Thèse, Université d'Orléans, 1995.

[7] D. Diaz et al. – « GNU-Prolog, a clp(fd) system based on Standard Prolog (ISO) », 2003, http ://gprolog.sourceforge.net/ Distributed under the GNU license.

[8] R. Douence & N. Jussien – « Non-intrusive constraint solver enhancements », in *Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS'02)*, CW Reports, vol. 344, juil. 2002, p. 26–36.

[9] A. Elkhyari, C. Guéret & N. Jussien – « Constraint programming for dynamic scheduling problems », in *ISS'04 International Scheduling Symposium* (Awaji, Hyogo, Japan) (H. Kise, éd.), Japan Society of Mechanical Engineers, mai 2004, p. 84–89.

[10] G. Ferrand, W. Lesaint & A. Tessier – « Theoretical foundations of value withdrawal explanations for domain reduction », *Electronic Notes in Computer Science* **76** (2002).

[11] —, « Towards declarative diagnosis of constraint programs over finite domains », in *Proceedings of the Fifth International Workshop on Automated Debugging, AADEBUG2003* (Ghent, Belgium) (M. Ronsse, éd.), sept. 2003, p. 159–170.

[12] — , « Explanations to understand the trace of a finite domain constraint solver », in *14th Workshop on Logic Programming Environments* (Saint-Malo, France) (S. Muoz-Hernàndez, J. M. Gomez-Perez & P. Hofstedt, éds.), sept. 2004, p. 19–33.

[13] C. Guéret, N. Jussien & C. Prins – « Using intelligent backtracking to improve branch and bounds methods : an application to open-shop problems », *European Journal of Operational Research* **127** (2000), no. 2, p. 344–354.

[14] U. Junker – « Quickxplain : Conflict detection for arbitrary constraint propagation algorithms », in *IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, 2001.

[15] N. Jussien, R. Debruyne & P. Boizumault – « Maintaining arc-consistency within dynamic backtracking », in *Principles and Practice of Constraint Programming (CP 2000)* (Singapore), Lecture Notes in Computer Science, no. 1894, Springer-Verlag, sept. 2000, p. 249–261.

[16] L. Langevine, M. Ducassé & P. Deransart – « A propagation tracer for gnu-prolog : from formal definition to efficient implementation », in *Proceedings of ICLP'03,* (Mumbai, India) (C. Palamidessi, éd.), Lecture Notes in Computer Science, Springer-Verlag, déc. 2003.

[17] S. Ouis, N. Jussien & P. Boizumault – « COINS : a constraint-based interactive solving system », in *Proceedings of the 12th Workshop on Logic Programming Environments, (*Wlpe*'02)* (Copenhagen, Denmark) (A. Tessier, éd.), 2002, Computer Research Repository cs.SE/0207046.

[18] G. Rochart & N. Jussien – « Implémenter des contraintes globales expliquées », in *Actes des premières Journées francophones de programmation par contraintes (JFPC 2005)* (C. Solnon, éd.), 2005.

[19] P. Van Hentenryck, V. Saraswat & Y. Deville – « Design, implementation and evaluation of the constraint language cc(FD) », in *Constraints : Basics and Trends* (A. Podelski, éd.), LNCS, no. 910, Springer Verlag, 1995.