Des Propagateurs Indépendants des Solveurs *

Jean-Noël Monette **Pierre Flener** **Justin Pearson**

Uppsala University, Department of Information Technology, Uppsala, Suède
{jean-noel.monette,pierre.flener,justin.pearson}@it.uu.se

Résumé

Cet article présente une extension des indexicals pour décrire des propagateurs de contraintes globales. Ce nouveau langage produit par compilation des propagateurs pour différents solveurs et est indépendant d'un solveur particulier. De plus, nous montrons que cette description de haut niveau aide à prouver certaines propriétés telles que correction et monotonie. Des résultats expérimentaux montrent que les propagateurs compilés à partir de descriptions par indexicals sont parfois presque aussi rapides que des propagateurs natifs de Gecode. Cela montre que notre langage peut être utilisé pour prototyper facilement de nouvelles contraintes globales.

1 Introduction

L'un des principaux atouts de la programmation par contraintes (CP) est l'existence de nombreux algorithmes de filtrage, appelés propagateurs, conçus spécialement pour les contraintes globales et qui permettent de résoudre efficacement des problèmes combinatoires. Les solveurs CP permettent la définition de nouvelles contraintes et de leurs propagateurs respectifs. Cependant, il peut être fastidieux de mettre en oeuvre un propagateur correct, efficace, se conformant à l'interface du solveur et codé à la main dans le langage d'implémentation du solveur, par exemple en C++.

Dans cet article, nous proposons un langage indépendant des solveurs pour décrire une grande classe de propagateurs. Notre contribution est double.

Tout d'abord, nous facilitons la mise en oeuvre et le partage de propagateurs. Les propagateurs sont décrits de façon concise et sans référence aux détails d'implémentation des solveurs. L'implémentation des propagateurs est générée à partir de leurs descriptions. Cela permet de prototyper rapidement un propagateur pour une nouvelle contrainte pour laquelle il n'existe pas de propagateur natif ou de décomposition (assez bonne). Le propagateur généré peut même servir de base à des raffinements manuels. Cela permet aussi d'intégrer une contrainte existante dans un nouveau solveur, comme chaque solveur peut être équipé de son propre générateur de code à partir du langage de description de propagateurs. Nous pensons qu'un tel langage peut jouer le même rôle pour le partage de propagateurs entre solveurs que des langages de modélisation jouent pour le partage de modèles entre solveurs.

Ensuite, nous facilitons la preuve des propriétés d'un propagateur, telles que la correction et la monotonie. Le niveau d'abstraction plus élevé de notre langage de description de propagateurs nous permet de concevoir des outils d'analyse et de transformation de propagateurs. Cette méthode permet d'appliquer des résultats théoriques aux propagateurs de manière systématique.

Notre approche est basée sur le travail fondateur au sujet des indexicals [28]. En résumé, un indexical définit une restriction sur le domaine d'une variable de décision en tenant compte des domaines actuels des autres variables de décision. Les indexicals ont été utilisés pour implémenter des contraintes définies par l'utilisateur dans divers systèmes, tel que SICStus Prolog [6]. Alors que les indexicals originaux ne peuvent faire face qu'à des contraintes d'arité fixe, nous les étendons pour traiter des contraintes d'arité non-fixe (appelées communément contraintes globales) en ajoutant des tableaux de variables de décision et des opérations sur de tels tableaux (itérations et opérateurs n-aire). De plus, contrairement aux implémentations classiques des indexicals, les indexicals ne sont pas interprétés ici, mais compilés en instructions pour le langage source du solveur cible.

*Cet article est une traduction de "Towards Solver-Independent Propagators" publié dans les actes de CP2012[16]. Nous conseillons la lecture de l'article original. Ce travail est supporté par la bourse 2011-6133 du Conseil Suédois de la Recherche (VR). Nous remercions les relecteurs anonymes et Christian Schulte pour leurs commentaires constructifs, et Marie Battisti et Loïc Blet pour l'aide à la traduction.

Le document est structuré comme suit. Après avoir présenté les bases nécessaires (Section 2) et des exemples de descriptions de propagateurs dans notre langage (Section 3), nous donnons la liste de nos décisions pour la conception du langage (Section 4). Ensuite, nous décrivons notre langage (Section 5), montrons comment analyser les propagateurs écrits dans le langage (Section 6), discutons notre implémentation actuelle (Section 7) et l'évaluons expérimentalement (Section 8). Nous terminons le papier par une revue des travaux apparentés et un regard sur nos futures directions de recherche (Section 9).

2 Bases

Soit X un ensemble de variables de décision entières qui prennent leurs valeurs dans un univers $\mathcal{U}$, où $\mathcal{U}$ peut être $\mathbb{Z}$ mais en pratique en est un sous-ensemble. Dans un solveur à domaines finis (FD), un store est une fonction $S\colon X \to \mathcal{P}(\mathcal{U})$, où $\mathcal{P}(\mathcal{U})$ est l'ensemble des sous-ensembles de $\mathcal{U}$. Pour une variable $x \in X$, l'ensemble $S(x)$ est appelé le *domaine* de x et est l'ensemble des valeurs possibles pour x. Un store S est une *affectation* si chaque variable a un domaine singleton ; de telles variables sont dites *déterminées*. Un store est *échoué* si une variable a un domaine vide. Un store S *domine* un store T (noté $S \sqsubseteq T$) si le domaine de chaque variable dans S est un sous-ensemble de son domaine dans T.

Une *contrainte* $C(Y)$ sur une séquence Y de X est une restriction des valeurs que ces variables peuvent prendre en même temps. La contrainte $C(Y)$ est un sous-ensemble de $\mathcal{U}^n$, où n est la longueur de Y. Une affectation A *satisfait* une contrainte $C(Y)$ si la séquence $[v \mid \{v\} = A(y) \wedge y \in Y]$ est membre de $C(Y)$. Étant donné un store S, une valeur $v \in S(y)$ pour un $y \in Y$ quelconque est *cohérente* avec une contrainte C si il existe une affectation $A \sqsubseteq S$ avec $A(y) = \{v\}$ qui satisfait C. Une contrainte C est *satisfaisable* dans un store S si il existe une affectation $A \sqsubseteq S$ qui satisfait C. Une contrainte C est *impliquée* dans un store S si toutes les affectations $A \sqsubseteq S$ satisfont C.

Un *vérificateur* pour une contrainte C est une fonction qui décide si une affectation satisfait C. Un *propagateur* pour une contrainte C est une fonction de stores à stores dont le rôle est de retirer du domaine les valeurs non cohérentes avec C. Nous sommes intéressés ici par l'écriture de propagateurs. Dans de vrais solveurs, les propagateurs ne renvoient pas un store mais modifient le store courant. Il y a un certain nombre de propriétés désirables pour un propagateur :

- *Correct* : Un propagateur correct ne retire jamais de valeurs qui sont cohérentes avec sa contrainte. C'est requis pour tout propagateur.
- *Vérifiant* : A propagateur vérifiant décide si une affectation satisfait sa contrainte. Cela peut se diviser en *correction de singleton* (accepte toutes les affectations satisfaisantes) et *complètude de singleton* (rejette toutes les affectations non satisfaisantes).
- *Contractant* : Un propagateur P est contractant si $P(S) \sqsubseteq S$ pour tout store S.
- *Monotone* : Un propagateur P est monotone si $S \sqsubseteq T$ implique $P(S) \sqsubseteq P(T)$ pour tous stores S et T.
- *Domaine cohérent* (*DC*) : Un propagateur DC retire toutes les valeurs non cohérentes. Des cohérences plus faibles existent, telles que la cohérence de bornes et la cohérence de valeur.
- *Idempotent* : Un propagateur P est idempotent si $P(S) = P(P(S))$ pour tout store S. Cela permet un meilleur ordonnancement du solveur [23].

De plus, pour des raisons d'efficacité, il est bon d'avoir des propagateurs avec une faible complexité temporelle. C'est la raison pour laquelle la cohérence de domaine est souvent remplacée par des cohérences plus faibles. Une autre préoccupation est d'éviter d'exécuter un propagateur qui ne peut retirer de valeur du store courant. Plusieurs mécanismes peuvent être implémentés pour éviter de telles exécutions : déclarer l'idempotence et l'implication, souscrire seulement à des modifications de domaine pertinentes.

Les indexicals ont été introduits dans [28] pour décrire des propagateurs pour le solveur cc(FD). Un(e) (*expression*) *indexical* est de la forme x `in` σ, qui signifie que le domaine de la variable x doit être réduit à l'intersection de son domaine courant avec l'ensemble σ ; cet ensemble peut dépendre d'autres variables et est calculé en fonction de leurs domaines dans le store *courant*. Un indexical est exécuté chaque fois que le domaine d'une des variables apparaissant dans σ est modifié. Un propagateur est typiquement décrit par plusieurs indexicals, un par variable de la contrainte. Les indexicals ont été inclus dans plusieurs autres systèmes, étendus avec des indexicals vérifiant [6], des expressions conditionnelles [24, 6], et des gardes [29].

3 Exemples de Descriptions de Propagateurs

Dans ce papier, nous étendons la syntaxe des indexicals pour utiliser des tableaux de variables et des opération sur ces tableaux (itérations et opérateurs n-aire). Nous commençons par présenter quelques exemples de descriptions de propagateurs qui montrent les principales caractéristiques de notre langage. Cela nous amènera à expliquer les décisions que nous avons prises pendant la conception et à donner une définition

```
1 def SUM(vint[] X,vint N){
2   propagator(v1){
3     N in sum(i in rng(X))(dom(X[i]));
4     forall(i in rng(X))
5       X[i] in dom(N) -
6           sum(j in rng(X):j!=i)(dom(X[j]));
7   }
8   propagator(v2){
9     N in sum(i in rng(X))(min(X[i])) ..
10            sum(i in rng(X))(max(X[i]));
11    forall(i in rng(X))
12      X[i] in min(N) - sum(j in rng(X):j!=i)(max(X[j]))
13           .. max(N) - sum(j in rng(X):j!=i)(min(X[j]));
14  }
15  checker{ val(N) = sum(i in rng(X))(val(X[i])) }
16}
```

FIGURE 1 – Code pour la contrainte SUM, avec deux propagateurs

```
1 def EXACTLY(vint[] X, vint N, int v){
2   propagator{
3     N in sum(i in rng(X))(b2i(entailed(EQ(X[i],v)))) ..
4        sum(i in rng(X))(b2i(satisfiable(EQ(X[i],v))));
5     forall(i in rng(X)){
6       once(val(N) <=
7         sum(j in rng(X):i!=j)
                (b2i(entailed(EQ(X[j],v)))))
8           post(NEQ(X[i], v));
9       once(val(N) >
10        sum(j in rng(X):i!=j)
                (b2i(satisfiable(EQ(X[j],v)))))
11          post(EQ(X[i], v));
12    }
13  }
14        checker{val(N) = sum(i in
                rng(X))(b2i(val(X[i])=v))}
15}
```

FIGURE 2 – Code pour la contrainte EXACTLY

plus précise du langage.

La Figure 1 présente la contrainte SUM(X, N), vérifiée si la somme des valeurs dans le tableau X égale N. Il est possible de décrire plusieurs propagateurs et un vérificateur pour une contrainte. En particulier, il y a ici deux propagateurs : `v1` utilise l'entièreté des domaines des variables (p.ex., `dom(N)`), tandis que `v2` n'utilise que leurs bornes (p.ex., `min(N)`). Une description de propagateur est une liste d'indexicals. Il est possible d'accéder aux domaines des variables avec les quatre fonctions `dom`, `min`, `max` et `val`. Les opérateurs arithmétiques peuvent être appliqués à des entiers ou à des ensembles. L'opérateur `sum` est n-aire, il opère sur une séquence de valeurs de longueur arbitraire. L'opérateur `rng` représente l'intervalle des indices d'un tableau, et $\ell..u$ l'intervalle d'entiers de ℓ à u inclus. Les lignes 9–10 signifient "Le domaine de `N` doit être intersecté avec l'intervalle dont la borne inférieure est la somme des plus petites valeurs du domaine de chaque variable de `X`, et dont la borne supérieure est la somme des plus grandes valeurs du domaine de chaque variable de `X`." Cet exemple montre aussi comment écrire des boucles (`forall` aux lignes 4–6 et 11–13).

La figure 2 présente la contrainte EXACTLY(X, N, v), vérifiée si exactement N variables du tableau X sont égales à la valeur v. Cet exemple illustre l'utilisation de conditions (`when`), de conversions de booléen vers entier (`b2i`$(\mathit{false}) = 0$ et `b2i`$(\mathit{true}) = 1$), et de références à d'autres contraintes (`EQ` et `NEQ`, contraignant une variable à être respectivement égale à et différente d'une certaine valeur). Les fonctions `entailed` et `satisfiable` vérifient l'état d'une contrainte en fonction du domaine courant des variables et `post` invoque un propagateur d'une certaine contrainte. Les lignes 3–4 restreignent le domaine de `N` entre deux bornes. La borne inférieure est calculée comme le nombre de variables de `X` qui *doivent* être assignées à `v`, et la borne supérieure comme le nombre de variables qui *peuvent* être assignées à `v`. Le corps de la boucle retire `v` du domaine d'une variable (lignes 6–8), ou fixe une variable à `v` (lignes 9–11), quand certaines conditions sur les autres variables sont respectées. Les modifications des domaines sont effectuées en invoquant la propagation d'autres contraintes.

4 Décisions de Conception du Langage

Le langage, tel que présenté dans la section précédente et défini plus précisément dans la suivante, a été conçu selon les décisions suivantes.

Le langage est basé sur les indexicals. Les indexicals ont déjà été utilisés avec succès dans plusieurs solveurs, et de nombreux travaux ont été effectués à leur sujet, e.g. in [5] and [9]. De plus, les indexicals sont très simples à comprendre et sont souvent proches du premier raisonnement que quelqu'un ferait au sujet d'un propagateur. Une des restrictions que nous maintenons est que les indexicals sont *sans état*, et ne peuvent donc pas représenter des propagateurs compliqués tels que le propagateur DC pour ALLDIFFERENT [20]. C'est une limitation forte mais un choix doit être fait entre la simplicité du langage et la complication des propagateurs. Un grand nombre de contraintes ont cependant des propagateurs sans état efficaces.

Le langage est fortement typé, pour simplifier la compréhension et la compilation des propagateurs. Cela nécessite l'ajout de l'opérateur `b2i`.

Nous introduisons des tableaux et des opérateurs n-aires pour pouvoir traiter efficacement des contraintes globales. Par exemple, l'expression `sum(i in`

`rng(X))(val(X[i]))` est paramétrée par l'indice de boucle (`i` ici), son domaine (`rng(X)` ici), et l'expression dépendant de l'indice (`val(X[i])` ici) qui doit être agrégée (sommée ici).

Nous introduisons aussi des méta-contraintes, des invocations de contraintes et des variables locales pour aider à écrire des propagateurs plus concis. Voir Section 5 pour les détails.

Pour faciliter l'utilisation, nous ne voulons qu'un petit nombre d'opérateurs et primitives. Cela nous permet aussi d'avoir une procédure de compilation relativement simple. Cependant, il n'est pas impossible que de nouvelles constructions soient ajoutées dans le futur, mais avec précaution.

Pour être général (dans l'approche FD), nous évitons d'ajouter des primitives spécifiques à un solveur. En particulier, notre langage n'a que quatre fonctions pour accéder au domaine d'une variable (voir Section 5). Les autres moyens de communiquer avec un solveur sont les réductions de domaine, fournies par tout solveur FD, et un mécanisme d'échec (qui peut être imité en vidant un domaine).

Notre langage n'a pas non plus certaines constructions présentes dans certaines implémentations de contraintes, telles que la détection d'implication, les littéraux gardés [13], et des évènements avec un grain plus fin [15]. Nous comptons étudier comment ceux-ci peuvent être inclus sans compliquer de trop le langage.

```
CSTR    ::= def CNAME(ARGS){ PROPAG+ CHECKER?)}
PROPAG  ::= propagator(PNAME?){ INSTR* }
CHECKER ::= checker{ BOOL }
INSTR   ::= VAR in SET; | post(CINVOKE,PNAME?); | fail;
            | once(BOOL){ INSTR* } | forall(ID in SET){
            INSTR* }
SET     ::= univ | emptyset | ID | INT..INT | rng(ID) |
            NSETOP(ID in SET)(SET) | dom(VAR) | -SET |
            SET BSETOP SET | {INT+} | {ID in SET:BOOL}
INT     ::= inf | sup | NUM | ID | card(SET)| min(SET)
            | max(SET) | min(VAR) | max(VAR) | val(VAR)
            | INT BINTOP INT | - INT | NINTOP(ID in
            SET)(INT) | b2i(BOOL)
BOOL    ::= true | false | ID | INT INTCOMP INT |
            INT memberOf SET | SET SETCOMP SET | not
            BOOL | BOOL BBOOLOP BOOL | NBOOLOP(ID
            in SET)(BOOL) | check(CINVOKE) | en-
            tailed(CINVOKE) | satisfiable(CINVOKE)
BINTOP  ::= + | - | * | / | mod
NINTOP  ::= sum | min | max
BSETOP  ::= union | inter | minus | BINTOP
NSETOP  ::= union | inter | sum
INTCOMP ::= = | != | <= | < | >= | >
SETCOMP ::= = | subseteq
BBOOLOP ::= and | or | =
NBOOLOP ::= and | or
CINVOKE ::= CNAME | CNAME(ARGS)
```

FIGURE 3 – Grammaire "à la BNF" de notre langage. Les constructions en gris se trouvaient déjà dans d'autres définitions des indexicals [28], [6]. Les règles correspondant à `ARGS` (liste d'arguments), `CNAME`, `PNAME`, `ID`, `VAR` (respectivement identificateur de contrainte, propagateur, constante, et variable), et `NUM` (littéral entier) ne sont pas montrées.

5 Définition du Langage

Dans la Section 5.1, nous définissons la syntaxe et la sémantique de notre langage. Il est fortement typé et a cinq types de base : entiers (`int`), booléens (`bool`), ensembles d'entiers (`set`), variables de décision entières (`vint`) et contraintes (`cstr`). Ce dernier type est discuté à la Section 5.2. Nous supportons des tableaux de n'importe quel type de base (mais pas de tableaux de tableaux pour le moment). Les identificateurs de (tableaux de) variables de décision commencent avec un lettre majuscule. Les identificateurs de constantes dénotant des (tableaux de) entiers, booléens et ensembles commencent avec une lettre minuscule.

5.1 Syntaxe et Sémantique

La Figure 3 présente la grammaire de notre langage. Nous passons en revue les règles de production. La règle principale (`CSTR`) définit une contrainte. Une contrainte est définie par son nom et sa liste d'arguments. Une définition de contrainte peut contenir la description d'un ou plusieurs propagateurs et un vérificateur.

Un *propagateur* possède un identificateur facultatif et contient une liste d'instructions. Une instruction (`INSTR`) peut être un *indexical*, x `in` σ, dont la signification est que le domaine de la variable x doit être réduit à l'intersection de son domaine actuel avec l'ensemble σ. Les autres instructions sont `fail` et `post`. L'effet de `fail` est de transformer le store courant en store échoué. L'instruction `post(`C`,` P`)` invoque le propagateur P de la contrainte C ; si P n'est pas spécifié, alors le premier (ou le seul) propagateur de C est invoqué. Il existe deux structures de contrôle. Le `forall` crée une itération sur un ensemble, et `once` crée un bloc conditionnel ; la raison de ne pas nommer ce dernier `if` est de souligner que les propagateurs doivent être monotone, et qu'une fois que la condition devient vraie, elle doit le rester. Voir Section 6 pour une discussion sur la monotonie.

La plupart des règles sur les ensembles, les entiers et les booléens n'ont pas besoin d'explications ou ont déjà été expliquées dans la Section 3. Certaines constantes sont définies : `univ` désigne l'univers $\mathcal{U}$, `inf` sa borne inférieure, `sup` sa borne supérieure et `emptyset` l'ensemble vide. Les opérations arithmétiques sur les entiers sont étendues point par point aux ensembles.

Il y a quatre moyens d'accéder au domaine d'une

variable : `dom(`x`)`, `min(`x`)`, `max(`x`)` et `val(`x`)` désignent respectivement le domaine de la variable x, son minimum, son maximum et sa valeur unique. Comme `val(`x`)` n'est déterminé que quand la variable x est fixée, le compilateur doit ajouter des gardes pour assurer un traitement correct quand x n'est pas déterminée.

Alors que l'instruction `post` invoque le propagateur d'une autre contrainte, les fonctions `entailed`, `satisfiable`, et `check` interrogent l'état d'un autre contrainte. Soit S le store actuel : `entailed(`C`)` et `satisfiable(`C`)` décident si la contrainte C est impliquée (respectivement, satisfaisable) dans S. Si S est une affectation, la fonction `check(`C`)` peut être appelée et décide si S satisfait la contrainte C (un exemple sera donné dans la sous-section suivante).

5.2 Méta-contraintes

Une nouvelle fonctionnalité de notre langage est ce que nous appelons une méta-contrainte, qui est une contrainte qui prend d'autre(s) contrainte(s) comme argument(s). Les méta-contraintes permettent d'écrire des propagateurs plus concis en encapsulant des fonctionnalités récurrentes.

Par exemple, la contrainte Among(X, N, S), vérifiée s'il y a N éléments dans le tableau X qui ont une valeur dans l'ensemble S, serait décrite presque identiquement à la contrainte Exactly(X, N, v) de la Figure 2. Le code commun peut être factorisé dans la méta-contrainte `Count(Vint [] X, vint N, cstr C, cstr NC)`, dont la description n'est pas donnée ici, mais dont le sens est défini par son vérificateur : `val(N) = sum(i in rng(X))(b2i(check(C(X[i]))))`, c'est-à-dire exactement `N` variables du tableau `X` satisfont la contrainte `C`. L'argument `NC` est la négation de la contrainte `C` (voir [3] pour savoir comment nier même des contraintes globales) et est utilisé dans le propagateur de Count (comme `NEQ` est utilisé à la ligne 8 de la Figure 2). Nous pouvons alors décrire Exactly et Among comme le montre la Figure 4. La méta-contrainte Count est étroitement liée à l'opérateur de cardinalité [26] mais nous permettons à l'utilisateur de décrire d'autres méta-contraintes.

6 Analyse Syntaxique et Outils

Un de nos objectifs est de faciliter la preuve de propriétés des propagateurs. Avant de passer à la compilation proprement dite, nous montrons comment notre langage contribue à cela et à d'autres fonctionnalités liées à l'écriture de propagateurs.

```
def EXACTLY(vint[] X, vint N, int v){
  propagator{
    cstr EQv(vint V):= EQ(V,v);
    cstr N_EQv(vint V):= NEQ(V,v);
    post(COUNT(X,N,EQv,N_EQv));
  }
}

def AMONG(vint N, vint[] X, set s){
  propagator{
    cstr INs(vint V):= INSET(V,s);
    cstr NINs(vint V):= NOTINSET(V,s);
    post(COUNT(X,N,INs,NINs));
  }
}
```

Figure 4 – Exactly et Among, décrits en utilisant la méta-contrainte Count

6.1 Analyse

Parmi les propriétés d'un propagateur présentées à la Section 2, la plupart sont difficiles à prouver pour un propagateur donné (sauf la contraction, que les indexicals satisfont par définition). Toutefois, comme cela a été démontré dans [5], il est possible de prouver la monotonie des indexicals. Ce résultat peut être étendu à notre langage plus général. Nous montrons en outre comment prouver la correction de certains propagateurs par rapport à leurs contraintes

Monotonie. La procédure pour vérifier la monotonie des indexicals est combinée à l'ajout de gardes pour `val`. L'arbre syntaxique représentant les indexicals est traversé par un ensemble de fonctions mutuellement récursives. Pour assurer la monotonie de l'ensemble du propagateur, chaque fonction vérifie un comportement *attendu* de la sous-arborescence auquel il est appliquée. Les fonctions récursives sont étiquetées *monotone* *anti-monotone* ou *fixe* pour les expressions booléennes ; *croissante*, *décroissante* ou *fixe* pour les expressions entières ; *croissante*, *décroissante* ou *fixe* pour les expressions d'ensemble ; et *monotone* pour les instructions. Par exemple, la fonction *croissante* vérifie que son expression entière en argument est (non strictement) croissante entre un store et un store dominant. Ces fonctions renvoient deux valeurs : si l'expression respecte effectivement son comportement attendu, et l'ensemble des variables qui doivent être fixées afin d'assurer une utilisation sûre de `val`. Cet ensemble de variables est utilisé pour ajouter des gardes dans le propagateur généré. Ces gardes sont ajoutées non seulement aux instructions, mais aussi à l'intérieur du corps de l'expression `b2i`.

Par manque de place, nous ne pouvons pas exposer toutes les règles qui composent les fonctions récursives

Expression originale	Expression gardée	Mono
`X in {val(Y)}`	`once(ground(Y)) X in {val(Y)}`	true
`B in b2i(val(X)=v) ..` `  b2i(v memberOf dom(X))`	`B in b2i(ground(X) and val(X)=v) ..` `  b2i(v memberOf dom(X))`	true
`once(min(B)=1) X in {v}`	`once(min(B)=1) X in {v}`	false
`once(min(B)>=1)X in {v}`	`once(min(B)>=1) X in {v}`	true
`once(val(B)=1) X in {v}`	`once(ground(B) and val(B)=1)X in {v}`	true

TABLE 1 – Exemples d'ajout de garde et de vérification de monotonie

(il y a environ 200 règles). Au lieu de cela, nous montrons quelques exemples. A titre d'exemple d'une règle, considérez l'appel à la fonction *croissante* sur une expression de la forme `min(`i `in` σ`)(`e`)`. Pour que cette expression soit croissante, σ doit être *décroissante* et e doit être *croissante*. En outre, l'ensemble des variables de garde est l'union des variables qui doivent être gardées pour ces deux sous-expressions.

Le tableau 1 montre quelques exemples de résultats de la procédure. Dans ce tableau, `ground(`x`)` représente un opérateur (qui ne fait pas partie de notre langage) qui décide si la variable x est déterminée. La deuxième colonne indique où les gardes sont ajoutées aux indexicals donnés dans la première colonne. La troisième colonne indique si l'indexical est prouvé monotone ou non. La première ligne ajoute juste une garde. La deuxième ligne montre qu'une garde peut être ajoutée à l'intérieur d'une expression `b2i` afin qu'il retourne `0` tant que `X` n'est pas fixé. Les trois dernières lignes montrent comment de petites variations modifient la monotonie d'une expression. La condition `min (B) = 1` n'est (syntaxiquement) pas monotone, car en général cette condition pourrait être vraie dans un store et devenir fausse dans un store plus fort, mais si nous savons que `B` représente un booléen (avec un domaine `0..1`), alors nous pouvons remplacer l'égalité par une inégalité comme cela est fait dans le quatrième exemple. La dernière ligne montre une autre façon d'obtenir la monotonie : remplacer le `min` par `val`, ce qui nécessite l'ajout d'un garde.

La correction de la procédure de contrôle de monotonie peut être démontrée par induction sur les règles récursives, comme suggéré dans [5]. Toutefois, cette procédure est syntaxique, donc incomplète. Un exemple de propagateur pour $\text{EQ}(X, Y)$ qui est monotone mais non reconnu comme tel est donné en haut de la Figure 5. La monotonie n'est pas reconnue, car il faut que les ensembles sur lesquelles les boucles `forall` itèrent soient croissants, tandis que `dom(`x`)` ne peut que rétrécir. Cependant, ce n'est pas le plus simple pour décrire la propagation de cette contrainte : un propagateur de 2 lignes se trouve en bas à gauche de la Figure 5.

```
def EQ(vint X1, vint X2){
 propagator{
   forall(i in dom(X1)) once(not i memberOf dom(X2))
     X1 in univ minus {i};
   forall(i in dom(X2)) once(not i memberOf dom(X1))
     X2 in univ minus {i};
 }
}
```

```
def EQ(vint X, vint Y){
 propagator{
   X in dom(Y);
   Y in dom(X);
 }
 checker{val(X) = val(Y)}
}
```

```
def EQ(vint X, int cY){
 propagator{
   X in {cY};
   once(not cY
       memberOf dom(X))
     fail;
 }
 checker{ val(X) = cY }
}
```

FIGURE 5 – Variations sur la contrainte EQ

Correction. Pour prouver algorithmiquement si un propagateur est correct par rapport à sa contrainte, nous utilisons le fait connu qu'un propagateur est correct si il est singleton correct et monotone. Nous proposons une procédure incomplète mais correcte pour prouver qu'un propagateur P est singleton correct par rapport à son vérificateur C, et donc par rapport à sa contrainte (en supposant que le vérificateur est correct). Nous devons prouver que si une affectation satisfait C, alors elle n'est pas supprimée par P. A cette fin, à partir de la description par indexicals de P, on déduit une formule $C(P)$ qui définit les affectations acceptées par le propagateur. La correction de singleton est vraie si la formule $C \wedge \neg C(P)$ est insatisfaisable (c'est à dire, si $C \rightarrow C(P)$). Pour calculer $C(P)$, nous transformons les indexicals en une formule équivalente à l'aide des règles de réécriture suivantes :

`dom(`x`)` $\rightarrow$ `{val(`x`)}`
`min(`x`)` $\rightarrow$ `val(`x`)`
`max(`x`)` $\rightarrow$ `val(`x`)`
`fail` $\rightarrow$ `false`
x `in` σ $\rightarrow$ `val(`x`) memberOf` σ
`once(`b`){`y`}` $\rightarrow$ `(not` b`) or` y
`forall(`i `in` σ`){`y`}` $\rightarrow$ `and(`i `in` σ`)(`y`)`

Notre procédure actuelle pour prouver l'insatisfais-

abilité de $C \wedge \neg C(P)$ essaye de simplifier la formule en `false` en utilisant des règles de réécriture. Nous avons environ 240 règles de réécriture, allant de la simplification des booléens (par exemple, `faux ou` $b \rightarrow G$), à celle d'entiers et d'ensembles (par exemple, `min (`i `in` $\ell..u$`)(`i`)` $\rightarrow$ ℓ) et à l'évaluation partielle (par exemple, $2 + x + 3 \rightarrow x + 5$). Comme cette procédure incomplète n'est en mesure de prouver la correction de singleton que pour une petite partie des propagateurs (voir Section 7.2), nous avons l'intention de l'améliorer en appelant un solveur externe.

À titre d'exemple, l'application de la transformation propagateur-à-vérificateur sur le propagateur dans le coin inférieur gauche de la Figure 5 résulte en la formule `val(X) memberOf {val(Y)} and val(Y) memberOf {val(X)}`. Cette formule peut être démontrée équivalente au vérificateur de la contrainte (en utilisant les règles suivantes : x `memberOf {`y`}` $\rightarrow$ x `=` y, b `and` $B \rightarrow b$ et b `and not` $b \rightarrow$ `false`). En résumé, ce propagateur peut être prouvé automatiquement monotone, singleton correct, singleton complet (voir ci-dessous), et donc correct.

Vérifiant. L'approche ci-dessus peut également être utilisée pour prouver qu'un propagateur est vérifiant. En effet, la complétude de singleton est prouvée par l'implication $C(P) \rightarrow C$ (l'inverse de la correction de singleton), et un propagateur qui est singleton correct et singleton complet est vérifiant ($C(P) \Leftrightarrow C$).

6.2 Transformation

En plus de l'analyse, nous pouvons transformer algorithmiquement un propagateur. Nous avons mis au point deux premières transformations qui semblent intéressantes : modifier le niveau de raisonnement, et fixer des variables de décisions. Par manque de place, cette section ne figure pas dans la traduction en français.

7 Compilation

Nous discutons maintenant nos décisions de conception du compilateur et notre compilateur actuel.

7.1 Décisions de conception du compilateur

Au lieu d'interprétation, nous avons pris la décision de compiler vers le langage dans lequel les propagateurs sont écrits pour un solveur. Cela a le double avantage d'avoir une infrastructure qui est relativement indépendante des solveurs (seule la partie de génération de code dépend des solveurs), et d'avoir du code compilé qui est plus efficace que du code interprété. Le code généré est aussi autonome (il peut être distribué sans le compilateur).

Les propagateurs compilés sont actuellement sans état (comme le sont les indexicals) et utilisent des événements de réveil à gros grain. Ce choix vise à simplifier la compilation. Toutefois, après une analyse appropriée, il devrait être possible de produire des propagateurs qui incorporent certains états ou d'utiliser des événements plus fins.

La compilation produit un propagateur pour chaque description `propagator`. La compilation ne modifie pas l'ordre des indexicals à l'intérieur d'un propagateur. En outre, pour obtenir l'idempotence, le propagateur complet est répété jusqu'à ce qu'il atteigne son point fixe interne. Un autre choix valable aurait été de créer un propagateur pour chaque indexical et laisser le solveur effectuer l'ordonnancement. Nous n'avons pas évalué tous les compromis de ce choix. Une approche intermédiaire serait d'analyser la structure interne de la description du propagateur pour générer une bonne politique d'ordonnancement des indexicals à l'intérieur du propagateur. Cela nécessite beaucoup plus de travail et reste comme travail futur.

Les invocations des propagateurs (`post`) ou vérificateurs (`check`) sont remplacés par le code correspondant. C'est similaire à l'inlining de fonctions dans des langages de programmation classiques. Pour les fonctions `entailed` et `satisfiable`, la description du vérificateur correspondant est d'abord transformée afin de traiter des stores qui ne sont pas des affectations. Cela se fait avec environ 130 règles de réécriture formant une procédure récursive similaire à celle de la vérification de monotonie. Comme différences, les accesseurs `val` sont remplacés par `min`, `max`, ou `dom` lorsque cela est possible, ou sont convenablement gardés autrement. Par exemple, appeler la fonction *décroissante* sur le singleton `{val(`x`)}` renvoie `dom(`x`)`, mais l'appel à la fonction *croissante* ajoute une garde. L'implication nécessite une formule booléenne *monotone* et la satisfaisabilité une formule *antimonotone*. Par exemple, le vérificateur de satisfaisabilité généré pour `EQ(X, Y)` est `not dom(X) inter dom(Y) = emptyset`. À son tour, le vérificateur d'implication généré pour `EQ(X, Y)` est `ground(X) and ground(Y) and val(X) = val(Y)`.

Le choix mettre en ligne les invocations de contraintes simplifie beaucoup le processus de compilation. Toutefois, cela signifie que tous les contraintes référencées doivent être décrites par indexicals. Nous avons l'intention d'explorer comment supprimer cette limitation afin d'être en mesure d'invoquer des propagateurs intégrés dans le solveur cible.

7.2 Implémentation et Solveurs Cibles

Nous avons écrit un prototype en Java. Il utilise Antlr [19] pour l'analyse et StringTemplate [18] pour la génération de code. Actuellement, nous compilons en propagateurs pour Comet [10], Gecode [12] et OscaR [22]. Une grande partie du processus de compilation consiste à réécrire les opérateurs n-aires en boucles. Certaines optimisations sont faites, comme un pré-calcul des tableaux par programmation dynamique (en remplaçant de boucles imbriquées par des boucles successives) [25, Section 9] et la factorisation d'expressions répétées. Le compilateur détecte les événements qui devraient réveiller le propagateur, ce qui est fait en traversant l'arbre syntaxique et collectant les accès aux variables. Le compilateur ajoute également la détection d'implication au propagateur d'une contrainte C, en testant si `entailed(`C`)` est vrai.

Actuellement, nous avons écrit à peu près 700 lignes d'indexicals décrivant 76 propagateurs pour 48 contraintes, dont 14 sont des méta-contraintes, 17 sont des contraintes globales, et 17 sont binaires ou ternaires. Sur les 76 propagateurs, 69 sont prouvés monotones, parmi lesquels 16 sont prouvés singleton corrects et 29 sont prouvés singleton complets, faisant 16 propagateurs prouvés corrects. Ces chiffres pourraient être améliorées avec une meilleure procédure pour prouver l'insatisfaisabilité.

Pour avoir une idée de la concision du langage, notez que notre compilateur produit à partir de la description de 15 lignes de Exactly à la Figure 2 un propagateur pour Comet qui est d'environ 150 lignes de code, et un autre pour Gecode d'environ 170 lignes. Nous estimons que le code pour le propagateur intégré de cette contrainte est environ 150 lignes dans Gecode.

Le prototype, ainsi que les descriptions de propagateurs, sont disponible sur demande auprès du premier auteur ou sur `http://user.it.uu.se/~jeamo371/indexicals`.

8 Evaluation Expérimentale

Pour évaluer si les propagateurs décrits par indexicals se comportent raisonnablement, nous comparons quelques propagateurs générés avec des propagateurs natifs de Gecode et des décompositions simples. Nous ne nous attendons pas à ce que les propagateurs générés soient aussi efficaces que ceux faits à la main, mais le but est de montrer qu'ils sont une alternative viable lorsque l'on a peu de temps pour développer un propagateur d'une contrainte.

Notre cadre expérimental est comme suit. Nous utilisons Gecode 3.7.3. Pour chaque contrainte, nous recherchons l'ensemble de ses solutions. Nous répétons la recherche en utilisant plusieurs heuristiques de branchement pour essayer d'exercer autant de parties des propagateurs que possible.

Les contraintes étudiées sont Sum, Maximum Exactly et Element. Leurs descriptions par indexicals sont représentatives des autres propagateurs que nous avons implémentés. En outre, elles partagent la propriété que l'une des variables dépend fonctionellement des autres. Cela nous permet de comparer les différents propagateurs d'une contrainte avec un problème de référence où la contrainte est absente, mais la variable dépendante est fixée à une valeur arbitraire. Comme l'utilité de la contrainte est seulement de définir la dépendance fonctionnelle, le nombre de solutions est le même et la taille de l'arbre de recherche est la même, mais le temps passé en propagation est nul. On peut alors calculer le temps d'exécution d'un propagateur en soustrayant les totaux des temps d'exécution.

Pour Sum et Maximum, nous utilisons des versions qui raisonnent sur les bornes pour les indexicals et les propagateurs natifs ; pour Exactly et Element, nous utilisons le raisonnement sur le domaine. Les descriptions par indexicals de Sum et Exactly se trouve aux Figures 1 (`v2`) et 2 respectivement. Par manque de place, Maximum et Element ne sont pas montrées. Les décompositions de $\textsc{Sum}(X, N)$ et $\textsc{Maximum}(X, N)$ introduisent un tableaux A de $n = \|X\|$ variables auxiliaires. La décomposition de Sum est exprimée comme $A[1] = X[1] \wedge \forall_{i \in 2..n} (A[i-1] + X[i] = A[i]) \wedge A[n] = N$, et celle de Maximum est $A[1] = X[1] \wedge \forall_{i \in 2..n} (\max(A[i-1], X[i]) = A[i]) \wedge A[n] = N$. La décomposition de $\textsc{Exactly}(X, N, v)$ introduit un tableau B de variables booléennes et est défini comme $\forall_{i \in 1..n} (B[i] \equiv X[i] = v) \wedge N = \sum_{i \in 1..n} B[i]$; la somme des variables booléennes est implémentée par un propagateur natif. La contrainte $\textsc{Element}(X, Y, Z)$ (vraie si $X[Y] = Z$) est décomposée en $Y \in 1..n \wedge \forall_{i \in 1..n} (Y = i \Rightarrow X[i] = Z)$. De plus, nous utilisons des formulations par automate de Maximum et Element, qui se trouvent dans le catalogue des contraintes globales [4]. Les automates de Maximum et Element et la décomposition de Element ne font pas toutes les réductions possibles (mais bien les autres décompositions). Cela génère un surcoût d'environ 15% de noeuds en plus visités pour les automates, et 7% pour la décomposition de Element.

Le Tableau 2 présente les temps relatifs des différentes implémentations des contraintes pour des tableaux de 9 variables avec des domaines de 9 valeurs. Les heuristiques de recherche utilisées sont quelques combinaisons de l'ordre des variables (ordre des arguments de la contrainte, dans un tableau le plus petit ou le plus grand domaine en premier) et de l'or-

TABLE 2 – Temps d'exécution Relatifs (en pourcents)

	MAX.	SUM	EXAC.	ELEM.
Natif	100	100	100	100
Indexicals	125	269	252	118
Décomposition	195	296	313	204
Automate	675	n/a	n/a	487

dre des valeurs (fixer au minimum, diviser en deux, fixer à la médiane). Pour chaque contrainte et chaque propagateur, les temps sont additionnés pour les différentes heuristiques de recherche. Puis la somme des temps pour explorer l'arbre de recherche est soustraite. Enfin, pour chaque contrainte, la somme des temps de chaque propagateur est divisée par la somme des temps des propagateurs de Gecode. Par rapport aux propagateurs natifs, ceux générés induisent uniquement une petite surcharge pour MAXIMUM et ELEMENT, mais ne se comportent pas aussi bien pour SUM et EXACTLY. Cependant, dans tous les cas, les indexicals ont un meilleur temps que la décomposition, mais parfois seulement légèrement. En particulier, pour la contrainte EXACTLY, la décomposition a un meilleur temps pour certains cas plus petits (non présentés). Nous expliquons le comportement des indexicals sur cette contrainte par le fait qu'ils sont éveillés à chaque fois que le domaine d'une variable change même si certaines variables ne peuvent plus affecter le statut de la contrainte. Le propagateur natif et la décomposition sont plus intelligents et ignorent les variables qui ne peuvent plus prendre la valeur donnée. Les indexicals ont un bien meilleur comportement que les automates.

Les propagateurs compilés à partir des indexicals ont un temps d'exécution moyen par appel qui augmente (nécessairement) de façon linéaire avec le nombre de variables. Les temps d'exécution des propagateurs natifs augmentent également mais avec une pente beaucoup plus douce.

Ces expériences montrent que les descriptions de propagateurs par indexicals sont utiles, mais qu'il y a encore de la place pour améliorer la compilation.

9 Conclusion

Nous avons présenté un langage pour décrire des propagateurs indépendemment des solveurs. L'objectif est de faciliter l'écriture et le partage de propagateurs et de faciliter la preuve de leurs propriétés formelles. Le langage qui en résulte, basé sur les indexicals, est de suffisamment haut niveau pour cacher les détails d'implémentation et permettre certaines analyses et transformations. Il est compilé en code source pour des solveurs cibles.

L'idée de laisser l'utilisateur écrire ses propres propagateurs n'est pas neuve. Le système cc(FD) [28] est l'un des premiers à avoir proposé cela grâce à l'utilisation d'indexicals. Depuis lors, beaucoup de solveurs CP sont ouverts, dans le sens où n'importe quel utilisateur peut ajouter de nouveaux propagateurs (surtout pour des contraintes globales) au noyau de propagateurs. Dans de tels systèmes, l'utilisateur écrit le code dans le langage de programmation du solveur et l'intègre au solveur grâce à une interface définissant principalement comment interagir avec les variables et le coeur du solveur. Certains solveurs prennent une approche différente en proposant un langage pour définir la propagation ; citons les constraints handling rules [11] et les actions rules [29]. Notre langage est un niveau d'abstraction au-dessus de ces approches, car il peut être traduit en code pour *n'importe quel* solveur. De ce point de vue, il est proche en esprit de ce qu'est un langage de modélisation indépendant des solveurs : une couche au dessus des solveurs pour décrire facilement des modèles de problèmes qui sont ensuite compilés dans le langage du solveur. Dans notre cas, des propagateurs sont décrits au lieu des modèles.

Des propagateurs ont également été décrits en utilisant des contraintes atomiques et des règles de propagation [8]. Cependant, ils ont été conçus seulement pour raisonner sur les propagateurs, et il ne sont pas pratique pour effectivement implémenter des propagateurs, sauf pour les approches basées sur des solveurs SAT qui utilisent des contraintes de bas niveau (par exemple, la génération paresseuse de clauses [17]).

Les contraintes enfichables de [21] découplent également la mise en oeuvre des contraintes de l'architecture solveur, en utilisant une interface indépendante du solveur pour la communication entre les deux composants. Notre approche vise un niveau plus élevé d'abstraction, avec la possibilité de perdre un certain niveau de contrôle.

Cet article ouvre plusieurs pistes de recherche intéressantes, en plus de celles déjà citées dans les Sections 4 et 7.1 pour surmonter les décisions initiales de conception : le langage doit permettre un meilleur contrôle de l'algorithme de propagation, tout en restant simple et générale. La simplicité est importante car nous croyons que la présence d'outils *automatisés* pour l'analyse facilite l'écriture de propagateurs.

Des futures cibles de compilation sont d'autres solveurs FD (par exemple, Choco [7] et Jacop [14]), les générateurs paresseux de clauses SAT [17], les générateurs de coupes en MIP [1] et les fonctions de pénalité et invariants en CBLS [27], [2].

Références

[1] Achterberg, T. : SCIP : Solving constraint integer programs. Mathematical Programming Computation 1, 1–41 (2009)

[2] Ågren, M., Flener, P., Pearson, J. : Inferring variable conflicts for local search. In : Benhamou, F. (ed.) Proceedings of CP'06. LNCS, vol. 4204, pp. 665–669. Springer (2006)

[3] Beldiceanu, N., Carlsson, M., Flener, P., Pearson, J. : On the reification of global constraints. Tech. Rep. T2012 :02, Swedish Institute of Computer Science (February 2012), available at `http://soda.swedish-ict.se/view/sicsreport/`

[4] Beldiceanu, N., Carlsson, M., Rampon, J.X. : Global constraint catalog, 2nd Edition (revision a). Tech. Rep. T2012 :03, Swedish Institute of Computer Science (February 2012), available at `http://soda.swedish-ict.se/view/sicsreport/`

[5] Carlson, B., Carlsson, M., Diaz, D. : Entailment of finite domain constraints. In : Proceedings of ICLP'94. pp. 339–353. MIT Press (1994)

[6] Carlsson, M., Ottosson, G., Carlson, B. : An open-ended finite domain constraint solver. In : Proceedings of PLILP'97. LNCS, vol. 1292, pp. 191–206. Springer (1997)

[7] CHOCO : An open source Java CP library, `http://www.emn.fr/z-info/choco-solver/`

[8] Choi, C.W., Lee, J.H.M., Stuckey, P.J. : Removing propagation redundant constraints in redundant modeling. ACM Transactions on Computational Logic 8(4) (2007)

[9] Dao, T.B.H., Lallouet, A., Legtchenko, A., Martin, L. : Indexical-based solver learning. In : Proceedings of CP'02. LNCS, vol. 2470, pp. 541–555. Springer (2002)

[10] Dynadec, Dynamic Decision Technologies Inc. : Comet tutorial, v2.0 (2009)

[11] Frühwirth, T.W. : Theory and practice of constraint handling rules. Journal of Logic Programming 37(1–3), 95–138 (1998)

[12] Gecode Team : Gecode : A generic constraint development environment (2006), available from `http://www.gecode.org/`

[13] Gent, I.P., Jefferson, C., Miguel, I. : Watched literals for constraint propagation in minion. In : Proceedings of CP'06. LNCS, vol. 4204, pp. 182–197 (2006)

[14] JaCoP : Java constraint programming solver, `http://jacop.osolpro.com/`

[15] Mohr, R., Henderson, T. : Arc and path consistency revisited. Artificial Intelligence 28, 225–233 (1986)

[16] Monette, J.-N., Flener, P., Pearson, J. : Towards solver-independent propagators. In : Proceedins of CP'12. LNCS, vol. 7514, pp. 544–560 (2012)

[17] Ohrimenko, O., Stuckey, P., Codish, M. : Propagation via lazy clause generation. Contraintes 14, 357–391 (2009)

[18] Parr, T.J. : Enforcing strict model-view separation in template engines. In : Proceedings of the 13th intl. conf. on the World Wide Web. pp. 224–233. ACM (2004)

[19] Parr, T.J. : The Definitive ANTLR Reference : Building Domain-Specific Languages. The Pragmatic Bookshelf (2007)

[20] Régin, J.C. : A filtering algorithm for constraints of difference in CSPs. In : Proceedings of AAAI'94. pp. 362–367. AAAI Press (1994)

[21] Richaud, G., Lorca, X., Jussien, N. : A portable and efficient implementation of global constraints : The tree contrainte case. In : Proceedings of CICLOPS 2007. pp. 44–56 (2007)

[22] OscaR, `https://bitbucket.org/oscarlib/oscar/`

[23] Schulte, C., Stuckey, P.J. : Speeding up constraint propagation. In : Proceedings of CP'04. LNCS, vol. 3258, pp. 619–633. Springer (2004)

[24] Sidebottom, G., Havens, W.S. : Nicolog : A simple yet powerful cc(FD) language. Journal of Automated Reasoning 17, 371–403 (1996)

[25] Tack, G., Schulte, C., Smolka, G. : Generating propagators for finite set constraints. In : Proceedings of CP'06. LNCS, vol. 4204, pp. 575–589. Springer (2006)

[26] Van Hentenryck, P., Deville, Y. : The cardinality operator : A new logical connective for constraint logic programming. In : Proceedings of ICLP'91. pp. 745–759 (1991)

[27] Van Hentenryck, P., Michel, L. : Differentiable invariants. In : Proceedings of CP'06. LNCS, vol. 4204, pp. 604–619. Springer (2006)

[28] Van Hentenryck, P., Saraswat, V., Deville, Y. : Design, implementation, and evaluation of the constraint language cc(FD). Tech. Rep. CS-93-02, Brown University, Providence, USA (January 1993), revised version in *Journal of Logic Prog.* 37(1–3) :293–316, 1998. Based on the unpublished manuscript *Constraint Processing in cc(FD)*, 1991.

[29] Zhou, N.F. : Programming finite-domain constraint propagators in action rules. Theory and Practice of Logic Prog. 6, 483–507 (2006)