Prototyper des Moteurs de Propagation avec un DSL

Charles Prud'Homme, Xavier Lorca, Rémi Douence, Narendra Jussien

Prototyper des Moteurs de Propagation avec un DSL

Charles Prud'homme[1] **Xavier Lorca**[1] **Rémi Douence**[2] **Narendra Jussien**[1]

[1] EMNantes, INRIA TASC, CNRS LINA, FR-44307 Nantes Cedex 3, France
[2] EMNantes, INRIA ASCOLA, CNRS LINA, FR-44307 Nantes Cedex 3, France
{prenom.nom}@mines-nantes.fr

Résumé

La propagation des contraintes est au cœur des solveurs de contraintes. Deux tendances co-existent : les algorithmes de propagation orientés variables et ceux orientés propagateurs. Ces deux approches assurent le même niveau de consistance, généralement la consistance d'arc, mais leur efficacité peut varier selon l'instance traitée. Il est généralement admis qu'il n'y a pas de meilleure approche et les solveurs de contraintes modernes n'en implantent finalement qu'une seule. Dans cet article, nous souhaitons aller un peu plus loin en fournissant un langage indépendant de tout solveur qui permette de prototyper des moteurs de propagation. Nous validons notre proposition avec une implémentation de référence basée sur Choco et le langage de modélisation MiniZinc. Finalement, nous évaluons la réalisabilité de notre proposition et en illustrons la flexibilité.

Abstract

Constraint propagation is at the heart of constraint solvers. Two main trends co-exist : variable-oriented propagation engines and constraint-oriented propagation engines. Those two approaches ensure the same consistency level, generally arc-consistency, but their efficiency (computation time) can be quite different depending on the instance solved. It is usually accepted that there is no best approach in general, and modern constraint solvers implement only one. In this paper, we would like to go a step further providing a solver independent language at the modeling stage to enable the design of propagation engine prototypes. We validate our proposal with a reference implementation based on Choco and the MiniZinc modeling language.

1 Introduction

La propagation des contraintes, élément central des solveurs de programmation par contraintes, a été beaucoup étudiée ces dernières années [1, 15]. Différents algorithmes existent [9] ; les principaux sont des versions modernes d'algorithmes assurant la consistance d'arc [1, 2, 3]. Pour un algorithme donné, plusieurs orientations existent. Les plus fréquemment utilisées sont celles orientées "variables" ou "contraintes". Par exemple, IBM CPO [20], Choco [5], Minion [10] et or-tools [23] reposent sur un algorithme orienté variables, alors que Gecode [21], SICStus Prolog [4] et JaCoP [22] reposent sur un algorithme orienté contraintes. Concevoir un moteur de propagation, et plus généralement un solveur, est un processus fait de choix et compromis pour faire conjuguer adaptabilité (pour résoudre un large éventail de problèmes) et efficacité (pour les résoudre efficacement). Les stratégies de propagation de contraintes sont étroitement liées à l'implémentation même d'un solveur. Implémenter un algorithme qui soit correct, efficace, respectant l'interface spécifique du solveur ainsi que son langage de programmation, peut se révéler être un vrai défi. Il est vraisemblable que seuls les développeurs du solveur est en position de modifier cet algorithme. En conséquence, le moteur de propagation tend à devenir monolithique et relativement inaccessible dès la phase de modélisation. Donc, même s'il n'est pas raisonnable de donner accès l'algorithme de propagation de contraintes aux utilisateurs novices, il peut se révéler fort utile aux modeleurs avancés ainsi qu'aux développeurs. Les premiers y verront l'opportunité d'optimiser le processus de résolution. Les seconds pourront rapidement prototyper de nouvelles stratégies de propagation, avant la phase, complexe, de développement de la-dite stratégie.

Dans cet article, nous proposons un langage indépendant de tout solveur qui permet de configurer la propagation des contraintes dès la phase de modélisation. Notre contribution est triple. Premièrement, nous

présenterons un Langage Dédié (en anglais "*Domain Specific Language*" ou DSL) à la configuration du moteur de propagation des contraintes. Nous montrerons qu'il permet d'exprimer un large éventail de stratégies déjà existantes. Deuxièmement, nous exploiterons les propriétés de base d'un DSL pour assurer à la fois la complétude et la correction du moteur de propagation produit. Troisièmement, nous présenterons une implémentation concrète de ce langage dédié dans Choco ainsi que son intégration dans MiniZinc [19]. Enfin, nous montrerons que le surcoût induit par l'utilisation du DSL est acceptable pour le prototypage de moteurs de propagation.

Dans la suite, la Section 2 rappelle les bases de la propagation de contraintes, ainsi que l'état de l'art des améliorations apportées. La Section 3 est dédiée à la description du DSL. La Section 3.1 définit l'ensemble des techniques et services requis par le solveur sous-jacent pour accepter le langage. La Section 3.2 présente et discute des détails du langage ainsi que ses garanties. La Section 3.3 présente son intégration dans MiniZinc, alors que la Section 3.4 discute des possible limites de notre approche. Enfin, la Section 4 évalue le DSL à la fois lors de la phase d'analyse syntaxique et celle de résolution, et montre combien, sur un cas d'utilisation, le prototypage des moteurs de propagation est simple.

2 Contexte

La Programmation par Contraintes est un paradigme de programmation où les relations entre les variables sont établies par les contraintes. Un *Problème de Satisfaction de Contraintes* (PSC) est défini par le triplet $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ et consiste en un ensemble $\mathcal{V}$ de n variables, leurs domaines associés $\mathcal{D}$, et une collection $\mathcal{C}$ de m contraintes.

Domaine et Variable Le domaine $d_v \in \mathcal{D}$ associé à une variable $v \in \mathcal{V}$ définit l'ensemble fini de valeurs entières auxquelles la variable v peut être affectée. L'*affectation* $v^\star$ d'une variable v à une valeur x se traduit par la réduction de son domaine d_v à un singleton, $d_v = \{x\}$.

Constrainte Une contrainte $c \in \mathcal{C}$ est définie sur un ensemble de variables $V_c \subseteq \mathcal{V}$. Une contrainte est équipée d'une fonction booléenne définie par $f_c : V_c^\star \rightarrow \{\texttt{true},\texttt{false}\}$. Une solution de c est une affection de toutes ses variables telle que $f_c(V_c^\star) = \texttt{true}$. Une contrainte est dite *satisfiable* si il existe une solution pour c. Une solution pour un PSC est une affectation de toutes les variables de $\mathcal{V}$ telle que toutes les contraintes de $\mathcal{C}$ sont satisfaites simultanément ; un PSC est satisfiable si et seulement si il a une solution.

Réduction de domaines et propagation Une contrainte est caractérisée par son comportement face à la modification d'une de ses variables. Une contrainte est équipée d'un ou plusieurs algorithmes de filtrage, nommés *propagateurs*. Un propagateur $p \in \mathcal{P}$ est une fonction définie par $p : \mathcal{D} \rightarrow \mathcal{D}$. Il retire du domaine de ses variables les valeurs qui ne sont désormais plus cohérentes. Soit $c \in \mathcal{C}$, P_c indique l'ensemble des propagateurs de c et V_p l'ensemble des variable de p tel que pour tous $v \in V_p, v \in V_c \wedge p \in P_c$; l'ensemble des propagateurs d'un PSC est dénoté $\mathcal{P}$.

Propagation de contraintes La programmation par contraintes est basée sur la propagation de contraintes [9, 17]. Les déductions locales liées à l'application de chaque propagateur forcent à reconsidérer la satisfiabilité du PSC ; ceci est assuré par l'exécution de l'algorithme de propagation [1]. L'algorithme AC5 [3] est une variante d'un tel algorithme basée sur les événements. L'algorithme de propagation est un mécanisme qui *planifie* et *exécute* des *arcs* pour atteindre le point fixe de l'étape de propagation, et ainsi obtenir la satisfiabilité du PSC. Un arc a est une paire $\langle p^a, v^a \rangle$ qui réunit un propagateur p^a et une variable v^a, tel que $v^a \in V_{p^a}$; $\mathcal{A}$ représente l'ensemble des arcs du PSC. L'algorithme de propagation est déclenché par la modification d'un domaine (par exemple, l'application d'une décision) et se déroule en deux étapes. Premièrement un arc a est retiré d'un ensemble Q. Cet arc réunit le propagateur p^a à exécuter et la variable modifiée v^a. Deuxièmement, le propagateur p^a est exécuté, avec d_{v^a} pour paramètre. L'exécution peut résulter soit en un *échec*, par exemple un domaine devient vide, alors l'algorithme s'arrête ; soit, en la planification d'autres arcs dans Q. Ces opérations sont exécutées jusqu'à ce que Q devienne vide, ce qui signifie alors qu'un point fixe a été de nouveau atteint.

La manière dont Q est *orienté* peut avoir une incidence sur la manière dont les propagateurs doivent être implantés. L'orientation désigne le type d'objets manipulés par Q. Les alternatives aux arcs sont (a) de planifier des variables [2], (b) de planifier des propagateurs [4]. Dans le cas d'un moteur de propagation *orienté variables* (a), lorsqu'une variable est retirée de l'ensemble Q, une boucle exécute ses propagateurs un à un. Dans le cas d'un moteur de propagation *orienté propagateurs* (b), lorsqu'une variable est modifiée, une boucle planifie ses propagateurs un à un. La complexité reste inchangée ; cependant, le nombre maximum d'objets ajoutés dans Q peut varier, et l'ordre d'exécution des propagateurs également. Plus récemment, l'introduction des contraintes globales dans les solveurs de contraintes a souligné l'importance d'organiser l'exécution des propagateurs en fonction de leur complexité. Ainsi, il a été montré que la planification des propagateurs des contraintes globales devait être effectuée soigneusement. D'un côté,

dans les solveurs orientés variables, ceci est géré en ajoutant un ensemble dédié aux propagations *à gros grains* [5, 10]. Cette approche peut être améliorée en associant des *watched literals* [11] aux propagateurs. De l'autre côté, [15] ont introduit la notion de *propagateurs à états* et ont proposé une plage de priorités pour discriminer les propagateurs dans les moteurs orientés propagateurs. [13] ont introduit la notion d'*advisors* comme concept additionnel. Evidemment, ces approches sont similaires : elles ont pour but de réduire le nombre d'appels à la planification de propagateurs et d'améliorer l'efficacité du filtrage.

En parallèle de ces travaux sur les conditions de planification, l'*ordre de révision* des objets à propager a été étudié. L'ordre de révision définit comment un élément est sélectionné et retiré de l'ensemble Q. Il est généralement admis qu'une file, qui choisit l'élément le plus vieux d'abord, est un bon choix pour l'ensemble Q car il traite tous les éléments de manière équitable. Cependant, des alternatives existent. D'une part, [8] propose la sélection dynamique d'un élément en se basant sur un critère tel que la taille du domaine de la variable ou l'arité d'une propagateur. D'autre part, [15] ont proposé un moteur orienté propagateurs avec sept niveaux de priorités. Un propagateur est planifié dans une file en fonction de sa priorité dynamique [1]. Enfin, de récents progrès on été faits concernant l'accès à l'algorithme de propagation en introduisant la notion de groupe de propagateurs [16]. Néanmoins, l'implantation des groupes reste à la charge de l'utilisateur.

Enfin, contrairement aux stratégies de recherche qui sont communément disponibles sous la forme de portfolios ou de combinateurs [20], les moteurs de propagation sont plus ou moins imposés aux utilisateurs des solveurs. Cela réside, bien entendu, essentiellement dans la difficulté d'implanter un algorithme de propagation. En plus d'une connaissance de l'architecture du solveur, de solides compétences en programmation sont nécessaires pour ne pas dégrader la performance et la correction du solveur. Pour toutes ces raisons, nous proposons l'introduction d'un langage dédié au prototypage, rapide et robuste, des algorithmes de propagation.

3 Un DSL pour décrire les moteurs de propagation

Pour simplifier l'expression, ou la déclaration, d'un moteur de propagation pour l'utilisateur, nous introduisons maintenant un *langage dédié* [6] (DSL). Tout d'abord, nous donnons une courte liste des pré-requis nécessaires pour bénéficier entièrement du DSL. Ensuite, nous présentons le DSL en deux étapes : la composition des groupes et la structuration des groupes entre eux. Puis, nous rappelons ce qu'est MiniZinc [19] et décrivons comment ce langage de modélisation a été étendu avec notre DSL. Nous listons les propriétés et garanties qui accompagne notre DSL. Enfin, nous discutons des éventuels problèmes liés à notre proposition.

3.1 Accéder au propriétés du solveur

Pour entièrement bénéficier de notre DSL, des fonctionnalités sont requises. Parce qu'elles agissent sur l'efficacité des solveurs, il est très vraisemblable que ces fonctionnalités soient déjà implantées largement.

- Les propagateurs *à états* [15] : les propagateurs sont ordonnés par rapport à leur priorité. Les propagateurs *simples* (au sens de la complexité) sont exécutés avant ceux plus complexes ;
- Les groupes de propagateurs [16] : un propagateur *contrôleur* est lié à un groupe de propagateurs ;
- L'accès aux propriétés des variables et des propagateurs : par exemple, la cardinalité d'une variable, l'arité d'un propagateur ou sa priorité.

Les propriétés mentionnées dans ce dernier point doivent pouvoir être évaluées dynamiquement pendant la propagation. Une évaluation dynamique implique bien évidemment un coût et peut nécessiter des structures de données dédiées [8].

Pour plus de flexibilité et de précision, nous assumons que tous les arcs $\mathcal{A}$ sont accessibles explicitement. Ceci peut être fait en représentant explicitement chacun d'entre eux et en leur associant des watched literals [11] ou des advisors [13]. La représentation explicite des arcs d'un problème induit un surcoût mémoire : $\mathcal{O}(|\mathcal{V}| \times |\mathcal{P}|)$ objets supplémentaires doivent être créés, chacun d'entre eux maintenant deux pointeurs, l'un vers une variable, l'autre vers un propagateur. Cependant, les arcs apportent plus de flexibilité en donnant à la fois accès aux propriétés de la variable et à celles du propagateur. De plus, les arcs peuvent être regroupés autour de variables, de propagateurs ou de propriétés.

3.2 Le langage dédié

En considérant $\mathcal{A}$ en entrée, l'objectif du DSL est de décrire un ordre global sur cet ensemble. Le DSL est divisé en deux parties (Figure 1) : d'abord, l'affectation d'un arc à un groupe, puis, la structuration du moteur de propagation basée sur ces groupes. Ce processus en deux étapes permet de se concentrer d'abord sur les propriétés des arcs, puis de composer les groupes entre eux. Pour plus de clarté, chacune des parties sera présentée individuellement. La syntaxe suit une

1. La priorité dynamique combine l'arité et la priorité pour fournir une évaluation fine du coût d'exécution du propagateur [15].

FIGURE 1 – Le point d'entrée du DSL.

⟨*propagation_engine*⟩ : := ⟨*group_decl*⟩+ ⟨*structure_decl*⟩ ;

FIGURE 2 – Définition des groupes

```
⟨group_decl⟩   ::= ⟨id⟩ : ⟨predicates⟩;
⟨id⟩           ::= [[a-zA-Z]][[a-zA-Z0-9]]⋆
⟨predicates⟩   ::= ⟨predicate⟩ | true
                 | (⟨predicates⟩ ((&& | ||) ⟨predicates⟩)+)
                 | !⟨predicates⟩
⟨predicate⟩    ::= in(⟨var_id⟩ | ⟨cstr_id⟩)
                 | ⟨attribute⟩ ⟨op⟩ (⟨int_const⟩ | "string" )
⟨op⟩           ::= == | != | > | >= | < | <=
⟨attribute⟩    ::= var[.(name|card)]
                 | cstr[.(name|arity)]
                 | prop[.(arity|priority|prioDyn)]
⟨int_const⟩    ::= [+−][[0-9]][[0-9]]⋆
⟨var_id⟩       ::= -⋆⟨id⟩
⟨cstr_id⟩      ::= -⋆⟨id⟩
```

notation BNF avec les conventions suivantes : `tt` indique un symbole terminal ; ⟨*it*⟩ indique un symbole non terminal ; des crochets [*e*] définissent une option ; des doubles crochets [[a-z]] définissent un intervalle ; le symbole plus *e*+ définit une séquence de une ou plus répétitions de *e* ; le symbole étoile *e*⋆ définit une séquence de zéro ou plus répétitions de *e* ; l'ellipse *e*,... définit une séquence non vide de *e* séparés par des virgules ; l'alternance *a*|*b* indique des alternatives.

3.2.1 Déclaration des groupes

Nous présentons maintenant le langage dédié à la déclaration des groupes. La notion de groupe est définie dans [16]. La notion de groupe a un objectif simple : contrôler un ensemble de propagateurs en une fois en définissant, pour le groupe, une politique interne de planification et d'exécution. Ici, nous substituons aux propagateurs des arcs. Constituer des groupes d'arcs doit se faire dans le respect des règles d'affectation (décrites dans la Figure 2). L'utilisateur n'a accès qu'aux variables et aux contraintes lors de la modélisation. Celles-ci, et leurs propriétés, peuvent être directement référencées dans cette partie du DSL. Les propagateurs, quant à eux, ne peuvent être référencés que via leurs propriétés, puisqu'ils ne sont habituellement pas accessibles à l'étape de modélisation. Les propriétés listées ici sont présentes dans la majorité des solveurs.

Un groupe est déclaré avec l'aide d'un identifiant et d'une liste de prédicats. L'identifiant ⟨*id*⟩ est un nom unique commençant par une lettre, par exemple `G1`. Un prédicat est une fonction booléenne $\mathcal{A} \rightarrow \{\texttt{true}, \texttt{false}\}$, elle définit la condition d'appartenance à un groupe. Un prédicat est dit *en extension* s'il repose sur une variable ou une contrainte (via `var_id` et `cstr_id`). Les arcs associés à une variable ou une contrainte sont éligibles pour le groupe. Par exemple, `in(`v_1`)` définit un ensemble d'arcs A_1 tel que pour tout $a \in A_1$, $v^a = v_1$; `in(`c_1`)` définit A_2 tel que pour tout $a \in A_2$, $p^a \in P_{c_1}$. Un prédicat est dit *en intension* s'il repose sur une ou plusieurs propriétés. Tous les arcs retenus doivent alors satisfaire les propriétés. Les prédicats en intension sont construits à l'aide de trois paramètres : un attribut ⟨*attribute*⟩, un opérateur ⟨*op*⟩ et une valeur entière ou une chaîne de caractères.

FIGURE 3 – Structure d'un moteur de propagation

```
⟨structure⟩    ::= ⟨struct⟩ | ⟨struct_reg⟩
⟨struct⟩       ::= ⟨coll⟩ of {⟨elt⟩,...} [key ⟨comb_attr⟩]
⟨struct_reg⟩   ::= ⟨id⟩ as ⟨coll⟩ of {⟨many⟩} [key ⟨comb_attr⟩]
⟨elt⟩          ::= ⟨structure⟩
                 | ⟨id⟩ [key ⟨attribute⟩]
⟨many⟩         ::= each ⟨attribute⟩ as ⟨coll⟩ [of {⟨many⟩}]
                   [key ⟨comb_attr⟩]
⟨coll⟩         ::= queue(⟨qiter⟩)
                 | [rev] list(⟨liter⟩)
                 | [max] heap(⟨qiter⟩)
⟨qiter⟩        ::= one | wone
⟨liter⟩        ::= ⟨qiter⟩ | for | wfor
⟨comb_attr⟩    ::= (⟨attr_op⟩.)⋆⟨ext_attr⟩
⟨ext_attr⟩     ::= ⟨attribute⟩ | size
⟨attr_op⟩      ::= any | min | max | sum
```

Un ⟨*attribute*⟩ fait référence à une propriété d'une variable (`var.name`, le nom d'une variable ; `var.card`, sa cardinalité), d'une contrainte (`cstr.name`, le nom d'une contrainte ; `cstr.arity`, sa cardinalité) ou d'un propagateur (`prop.arity`, l'arité d'un propagateur ; `prop.priority`, sa priorité statique ; `prop.prioDyn`, sa priorité dynamique). Par exemple, `prop.priority` < 4 définit un ensemble d'arcs tel que la priorité statique de chacun des arcs de cet ensemble est strictement inférieure à 4. Les prédicats peuvent être combinés avec les trois opérateurs booléens `&&`, `||` and `!`.

Le ⟨*group_decl*⟩ peut être adapté à chaque solveur en étendant ⟨*attribute*⟩, c'est à dire, en déclarant de nouvelles propriétés aux contraintes et aux variables. Lorsque des variables ou des contraintes sont pointées, ⟨*groups_decl*⟩ est dépendant du modèle, autrement il en est complètement indépendant.

Le but de ⟨*groups_decl*⟩ est de contruire des sous-ensembles d'arcs. Cette partie du DSL est évaluée de manière impérative, *i.e.*, chaque évaluation de prédicat peut déplacer un ou plusieurs arcs de $\mathcal{A}$ vers le sous-ensemble défini par le prédicat. Cependant, un arc peut répondre `true` à des prédicats de différents groupes. Pour empêcher d'avoir à discriminer les prédicats entre eux, cette partie du DSL est évaluée de haut en bas et de droite à gauche. Ainsi, un arc qui répond `true` à un prédicat devient indisponible pour les prédicats suivants.

3.2.2 Déclaration de la structure

La seconde partie du DSL (Figure 3) porte sur la description, basée sur les groupes, des structures de données. Le but ici est double : indiquer "où" stocker un arc et "comment" le sélectionner. De la déclaration de la structure résulte une structure d'arbre hiérarchisée basée sur une collection ⟨*coll*⟩.

Une ⟨*coll*⟩ est une structure de données abstraite (SDA) composée d'autres SDA et d'arcs. Une SDA définit comment les éléments vont être planifiés et retirés pour une exécution. Il en existe trois types : une `queue`, une `list` et un `heap`. Chacun d'entre eux définit également un itérateur qui décrit la manière de le traverser. Une file `queue` est une collection d'élé-

FIGURE 4 – Moteur de propagation basé sur un tas.

```
1. heap(wone) of {
2.     G1 key var.card,
3.     list(for) of {G2} key any.var.card
4. }
```

ments de la forme *first-in-first-out*. Une liste `list` est une collection d'éléments ordonnés statiquement (`rev` renverse la liste). Un tas `heap` est une collection d'éléments ordonnés dynamiquement grâce à un critère (la fonction de comparaison par défaut est $\leq$, le mot-clé `max` le transforme en $\geq$).

Les itérateurs de bases sont `one` et `wone`, définis pour toute ⟨*coll*⟩. `one` traverse un élément d'une liste ⟨*coll*⟩. `wone`, version courte pour 'while one', appelle `one` jusqu'à ce que tous les éléments aient été traversés. Une `list` définit deux itérateurs supplémentaires : `for` and `wfor`. `for` traverse un à un les éléments d'une liste dans l'ordre croissant de son index. `for` n'autorise pas de retours arrières ni d'interruption de la traversée. `wfor`, version courte pour 'while for', appelle `for` jusqu'à ce que tous les éléments de la liste aient été traversés.

Un `heap` requiert la déclaration d'un critère de comparaison (qui est optionnel pour une `list`). Le critère doit être statique pour une `list` et dynamique pour un `heap`. Comme il n'y a aucune garantie qu'une ⟨*coll*⟩ contienne exclusivement des arcs, le critère de tri doit être défini par (a) le mot-clé `key` et un ⟨*attribute*⟩ d'arc ou (b) le mot-clé `key` et une ⟨*comb_attr*⟩ pour un ensemble d'éléments. L'évaluation d'un ensemble est faite en évaluant les arcs de l'ensemble (grâce à ⟨*attr_op*⟩). Une extension de ⟨*attribute*⟩, nommée ⟨*ext_attr*⟩, est introduite dans la Figure 3. Une ⟨*ext_attr*⟩ permet d'atteindre tout attribut d'un élément d'une collection d'élément, comme sa taille `size`. Par exemple, la Figure 4 déclare un moteur de propagation basé sur un tas. Le tas haut niveau impose que les éléments qui le composent soit évaluables (comparables entre eux). Tous les arcs de `G1` sont évalués en retournant la cardinalité de leur variable (ligne 2). Le second groupe est une liste d'arcs `G2`. Comme une liste n'est pas évaluable en tant que telle, son évaluation doit être déléguée à l'un des arcs qui la composent. Un arc de `G2` est alors sélectionné arbitrairement (`any`) pour obtenir la cardinalité de sa variable (ligne 3). Le mot-clé `any` indique à la fois de déléguer l'évaluation du critère à un élément de `list` et de le choisir arbitrairement.

Maintenant, nous décrivons comment utiliser les structures de données abstraites dans une structure complète. Le point d'entrée d'une déclaration de structure est ⟨*structure*⟩. Une ⟨*structure*⟩ définit la structure du plus haut niveau qui pilotera la propagation. Elle doit être de type ⟨*struct*⟩ ou ⟨*struct_reg*⟩.

Une ⟨*struct*⟩ définit une collection ⟨*coll*⟩ composée d'éléments ⟨*elt*⟩. Un ⟨*elt*⟩ peut faire référence soit à l'identifiant d'un groupe et une instruction d'ordre (⟨*id*⟩ [`key` ⟨*attribute*⟩]), soit une autre⟨*structure*⟩. Une ⟨*struct*⟩ est définie en *extension*. La Figure 4 montre un `heap` composé d'arcs de `G1` et d'une liste.

Une ⟨*struct_reg*⟩ définit une structure régulière. L'expressivité impliquée par sa régularité justifie sa présence : elle permet l'expression compacte de structures imbriquées, où l'utilisation de ⟨*struct*⟩ impliquerait une expression verbeuse et sujette à l'introduction de bugs. Une ⟨*struct_reg*⟩ est définie en *intension*. Elle prend en entrée un ensemble d'arcs défini par un identifiant de groupe et elle *génère* autant de ⟨*coll*⟩ que requis par l'instruction ⟨*many*⟩, potentiellement imbriquées les uns dans les autres. ⟨*many*⟩ implique une itération sur les valeurs de l'⟨*attribute*⟩ et regroupe les éléments avec la même valeur pour l'⟨*attribute*⟩ dans une même ⟨*coll*⟩. La Figure 5 montre la déclaration

FIGURE 5 – Exemple de structure régulière.

```
1. G as list(wfor) of{
2.   each prop.priority as queue(for) of{
3.     each var as list(for)}
4. }
```

d'une structure régulière. La construction de la structure de données prend `G` en entrée : autant d'ensembles d'arcs qu'il y a de `prop.priority` parmi les arcs de `G` sont créés (ligne 2). Pour rappel, [15] en définissent 7. Puis, les ensembles sont traités un par un et leurs arcs sont regroupés par variables dans des listes. Toutes les listes sont ajoutées dans une `queue` qui correspond à une `priority`. Les files sont ajoutées dans la liste de haut niveau. Si un critère dynamique le nécessite, l'étape d'affectation à la bonne collection sera effectué pendant la propagation.

3.2.3 Propriétés

Un langage dédié apporte des propriétés, mais également des garanties. Nous décrivons maintenant toutes les propriétés et garanties de notre langage.

Description indépendante du solveur Un langage dédié bénéficie naturellement d'un haut niveau d'abstraction. Le fait qu'il ne repose sur l'architecture d'aucun solveur permet de l'implanter facilement dans un large éventail de solveurs de contraintes.

Expressivité Notre DSL regroupe les structures de données et les caractéristiques usuelles (de la simple file aux files de priorités). Il permet d'aller au-delà en les combinant avec les propriétés et décrire une grande variétés de moteurs de propagation de manière concise, incluant le mélange d'orientations.

Extensibilité Le DSL peut être étendu de deux manières. Premièrement, la déclaration de nouveaux attributs permet de définir de manière plus précise les groupes. Deuxièmement, l'ensemble des collections proposé peut être enrichi, tout comme les itérateurs. Cela donne l'opportunité de déclarer des propagations incomplètes, telles que décrites dans [12].

Garantie de couverture Tous les arcs d'un modèle sont représentés dans le moteur de propagation. L'expressivité de notre DSL autorise la description de moteurs de propagation dans lesquels un ou plusieurs arcs seraient absents. Il en résulte un moteur de propagation qui ignore une partie du modèle. Actuellement, nous en informons l'utilisateur et stoppons l'évaluation du DSL. Une alternative est le *collecteur d'arcs* qui réclame les arcs *libres* pour assurer la couverture.

Unicité de la propagation Un arc peut satisfaire plusieurs prédicats, et donc être éligible à différents groupes. L'évaluation vérifiée plus haut du DSL garantie que chaque arc n'est représenté qu'une seule fois dans le moteur de propagation. Donc, un événement intervenant sur une variable ne peut pas être traité plus d'une fois. Cela garantit que la complexité reste inchangée en branchant le DSL.

Conformité Les arcs liés à aucune paire (p,v) existante du modèle ne sont pas représentées dans le moteur de propagation. Grâce à cette propriété, les arcs *dénués de sens* ne peuvent être ni planifiés ni exécutés durant la propagation. Une fois de plus, cela préserve la complexité du moteur de propagation.

Complétude La complétude est assurée par une collection haut niveau définie avec un itérateur "while", tel que `wone` and `wfor`. Ainsi, quelques soient les itérateurs des sous-structures, la collection de plus haut niveau assure qu'il n'y ait plus d'événement en attente avant de terminer la propagation. Sélectionner `one` ou `for` peut ne plus assurer la complétude. Cependant, [12] explique pourquoi une propagation incomplète peut être utile.

Fiabilité Le DSL autorise la description de moteurs de propagations mal formés, bien que corrects. Par exemple : un groupe peut être vide, une définition de groupe peut être inutilisée dans la déclaration de la structure, une structure peut imbriquer des $\langle many \rangle$ basés sur le même $\langle attribut \rangle$, une structure peut ne pas contenir d'arc ou juste un seul, etc. Une solution serait d'appliquer automatiquement des règles d'optimisation et de réduction.

Complexité Les propriétés de "Conformité" et d'"Unicité de la propagation" garantissent le maintien de la complexité de l'algorithme de propagation. Cependant, la déclaration de moteurs de propagation mal formés, ou l'utilisation excessive de critères dynamiques peut avoir un impact néfaste sur les performances du moteur de propagation. Les utilisateurs doivent être en conscients durant le prototypage d'un moteur de propagation et son évaluation. Par contre, le DSL garantit que la propagation se termine.

3.3 Implémentation

Étendre un langage de modélisation est une manière simple d'implanter notre DSL : l'accès aux objets du modèle y est assuré, sans requérir un solveur concret.

FIGURE 6 – Modèle MiniZinc `magic sequence` étendu avec notre DSL .

```
1 :   include "globals.mzn";
2 :   annotation name(string: s);
3 :   annotation engine(string: s);
4 :   int: n;
5 :   array [0..n - 1] of var 0..n: x;
6 :   constraint count(x,0,x[0])::name("c0");
7 :   constraint forall (i in 1..n - 1) (count(x, i, x[i]));
8 :   solve
9 :   :: engine("
10 :  G1: in("c0");
11 :  All: true;
12 :  list(wone) of {queue(wone) of{G1},
13 :        All as queue(wone) of {each cstr as list(wfor)}};
14 :  ")
15 :  satisfy;
```

Pour cela, nous avons choisi MiniZinc [19]. Il s'agit d'un langage de modélisation simple mais expressif. Il couvre un large éventail de solveurs. Nous avons également choisi ANTLR [2] pour interpréter la grammaire. Cet outil permet la séparation claire entre l'analyseur syntaxique, (le *lexeur*), la reconnaissance syntaxique (le *parseur*) et la transposition en instructions solveur. Cela simplifie la réutilisation du lexeur et du parseur, seule la transposition doit être adaptée aux solveurs.

En pratique, nous devons ajouter la description du moteur de propagation au sein du modèle MiniZinc. Ceci est effectué à l'aide d'annotations : la première sert à intégrer la description du moteur de propagation, la seconde sert à nommer une contrainte. La Figure 6 montre l'exemple d'un modèle MiniZinc dans lequel une déclaration de moteur de propagation a été ajoutée. Les lignes 1, 4-8 and 15 détaillent le modèle du problème de `magic sequence` (prob019, [18]) de taille 5. Les annotations requises pour nommer les contraintes (ligne 2) et décrire le moteur de propagation doivent être déclarées dans l'en-tête du fichier. La première contrainte (ligne 6) est annotée avec `::name("c0")` pour la nommer. Les lignes 9-14 décrivent le moteur de propagation : deux groupes sont définis. Le premier groupe `G1` (ligne 10) contient les arcs de la contrainte nommée `c0` ; le second groupe `All` (ligne 11) contient les arcs restants, grâce à l'évaluation haut-bas gauche-droite. La structure du moteur est composé d'une liste elle-même composée de deux files (lignes 12-13). La première file stocke les arcs du groupe `G1` (ligne 12) ; la seconde est composée de listes d'arcs, réunis autour de contraintes (ligne 13). Ce moteur oblige à toujours traiter les événements de la contraintes `c0` avant les autres événements.

3.4 Discussion

MiniZinc est distribué avec un langage bas niveau appelé FlatZinc. FlatZinc est le langage cible pour les modèles contraintes dans lequel les modèles MiniZinc sont traduits, à destination des solveurs. Cependant, la

2. http ://www.antlr.org/

traduction de MiniZinc vers FlatZinc peut induire une reformulation des contraintes optionnelles grâce aux contraintes obligatoires. Pour se concentrer sur le DSL et sa mise en application, nous interdisons ce processus de reformulation. Même si la déclaration originale pouvait être transformée comme le sont les contraintes, il n'y a aucune garantie que la déclaration résultante respecte la sémantique d'origine. En effet, si on considère une déclaration retardant autant que possible la programmation d'une contrainte `AllDifferent` qui, après reformulation, soit transformée en une clique de différences : cela serait contreproductif et trompeur. De plu, l'impact de la reformulation sur l'efficacité de la propagation n'a pas encore été évaluée formellement, et représente en soi un intéressant sujet de recherche.

D'autre part, un solveur peut ne pas disposer des pré-requis listés en Section 3.1, et donc, ne pas supporter les moteurs de propagation basés sur le DSL. Une refonte du solveur est alors nécessaire pour remplir ces conditions. Une telle refonte, vraisemblablement à destination des développeurs de solveurs, est une tâche critique. Cependant, il est fort probable que les solveurs basés sur les événements sont mieux préparés aux algorithmes de filtrage orientés arcs. De plus, cette refonte apporte non seulement plus de flexibilité dans les solveurs de contraintes mais ouvre également des perspectives pour la résolution. Naturellement, notre DSL ne remet pas en cause les moteurs natifs, il vient en complément de ceux-ci. Il peut également simplifier la conception des groupes de propagateurs.

4 Evaluation expérimentale

La plupart du temps, les structures de données utilisées dans un moteur de propagation ont des complexités très faibles. Par exemple, une file ou un bitset définit les opération `add()` and `remove()` en temps constant, un tas binaire définit les opérations `add()`, `extractMin()` et `update()` en $\mathcal{O}(log(n))$. Même si l'efficacité de ces opérations est critique, l'efficacité d'une résolution est plus liée à l'ordre dans lequel les propagateurs sont exécutés. Il y aura sûrement un surcoût lié au parsing du DSL, dû à la fois au chargement des données et à la vérification des garanties. Mais le surcoût lié à l'utilisation d'un moteur interprété devra être limité aux indirections induites par les compositions de groupes et aux évaluation des critères.

La configuration. Toutes les expérimentations ont été faites sur un Macbook Pro avec un 6-core Intel Xeon cadencé à 2.93Ghz sur MacOS 10.6.8, et Java 1.6.0_35. Les résultats sont basés sur 20 exécutions (l'écart-type est toujours inférieur à 5%) et les tableaux reportent le temps de parsing en millisecondes et le temps de résolution en secondes.

Solver. L'évaluation de notre DSL et son interprétation ont été implantées en Java avec Choco [5], une bibliothèque Java pour la programmation par contraintes. Les sources ont été modifiés pour répondre aux pré-requis listés dans la Section 3.1. De plus, trois moteurs de propagation natifs (*i.e.*, non basés sur le DSL) ont été implantés dans la bibliothèque : un moteur orienté propagateurs `prop`, un autre orienté variables `var`, le dernier est un moteur orienté propagateurs, composé de sept files et évaluant la priorité des propagateurs dynamiquement `7qd` [15].

Expérimentations. La première évaluation a été faite sur la base de 39 instances extraites de 21 problèmes [3] de la distribution MiniZinc [19]. Plus de détails sur les instances sont disponibles sur demande. Ensuite, nous présenterons un cas d'utilisation, basé sur le problème du `Golomb ruler`. Il permettra de démontrer combien le prototypage de moteurs de propagation est simplifié avec le DSL.

4.1 Descriptions des moteurs

Cette section est dédiée à la description des moteurs natifs(`prop`, `var` et `7qd`) à l'aide du DSL, ainsi qu'à la comparaison du traitement des fichiers MiniZinc avec les moteurs natifs et avec les moteurs interprétés. Pour chaque moteur natif, il est garanti que sa version DSL construise le même arbre de recherche et préserve l'ordre ainsi que le nombre d'exécutions des propagateurs. Cela nous permet d'illustrer l'expressivité du DSL.

Version DSL des moteurs de propagation Tout d'abord, nous présentons comment déclarer `prop`, un moteur de propagation orienté propagateurs, à l'aide du DSL (Figure 7(a)). La structure prend tous les arcs du problème en entrée. Puis, les arcs sont regroupés par propagateurs dans des listes. Chaque liste définit une traversée incomplète (`for`) mais la file haut niveau assure la complétude de la propagation (`wone`). Ensuite, nous présentons la déclaration de `var`, un moteur de propagation orienté variables, à l'aide du DSL (Figure 7(b)). Il est très proche de la description précédente, seulement, les arcs sont regroupés par variables. Enfin, nous présentons comment déclarer `7qd` à l'aide de notre DSL, un moteur orienté propagateur, composé de sept files et évaluant la priorité des propagateurs dynamiquement (Figure 7(c)). Comme les précédents, il prend tous les arcs du problème en entrée. Ces arcs sont distribués dans les différentes files en fonction de la priorité dynamique de leurs propagateurs. Pour chaque priorité disponible, les arcs sont regroupés autours des propagateurs dans les listes. Comme

3. Les problèmes sélectionnés sont : `bacp, bibd, black-hole, CostasArrays, debruijn-binary, fastfood, fillomino, ghoulomb, golomb, jobshop, langford, latin-squares, magicseq, market_split, nonogram, p1f, pento-minoes, radiation, rcpsp, slow_convergence, still_life`.

FIGURE 7 – Trois moteurs de propagation, version DSL

```
1 : All :true;
2 : All as queue(wone) of {
3 :     each prop as list(for)
4 : }
```

(a) Propagator-oriented.

```
1 : All :true;
2 : All as queue(wone) of {
3 :     each var as list(for)
4 : }
```

(b) Variable-oriented.

```
1 : All : true;
2 : All as list(wone) of {
3 :      each prop.prioDyn as queue(one) of {
4 :           each prop as list(for)
5 :      }key any.any.prop.prioDyn
6 : }
```

(c) Priority-oriented.

l'évaluation des priorités est dynamique, le choix de la file dans laquelle mettre une liste est fait grâce à l'évaluation du propagateur d'un arc de la liste. La liste de haut niveau garantit à la fois la complétude de la propagation (`wone`) et que les événements associés aux propagateurs de faible priorité sont traités avant ceux de plus forte priorité. Notez que le mot-clé `any` est doublé. Cela est nécessaire pour déléguer l'évaluation de la `priority` de la file à un élément évaluable, un arc parmi ceux présents dans ces listes. Le premier `any` délègue d'un niveau, *i.e.*, une liste, le second pointe vers un élément de cette liste, *i.e.*, un arc. Cet arc est donc le représentant de la file, et son propagateur est utilisé pour évaluer la file.

4.2 Evaluation du parsing et de la résolution

L'intérêt de la phase de prototypage est de construire et d'évaluer rapidement, mais sûrement, des moteurs de propagation. D'un côté, la phase de parsing n'est pas critique dans une démarche de prototypage, mais elle ne doit tout de même pas trop pénaliser les tests. La phase de résolution, quant à elle, est critique. Bien que les versions interprétées des moteurs de propagation ne puissent être compétitives, en terme d'efficacité, avec les approches natives, leur coût d'utilisation doit être aussi faible que possible pour valider notre approche en tant qu'outil utilisable en pratique. La Table 1 reporte le surcoût du parsing des fichiers MiniZinc sans et avec la description des moteurs de propagations. La phase de parsing inclut le chargement du fichier, l'analyse lexicographique, la reconnaissance syntaxique et la traduction en instructions du solveur. La Table 1 reporte également le surcoût de résolution des instances avec les moteurs interprétés en comparaison avec les versions natives. Dans ce cas, le temps affiché exclut le temps de parsing. Enfin, la colone "ranking" indique le pourcentage d'instances qui préserve, avec les versions interprétées, l'ordre impliqué par le temps de calcul entre les moteurs natifs. Sans surprise, la phase de parsing souffre de l'ajout du DSL, et globalement cette phase requiert 28,46% plus de temps, en moyenne. Les détails ne sont pas donnés ici, mais il y a un lien direct entre le temps de parsing

TABLE 1 – Surcoûts moyens induit par l'introduction du DSL, sur 39 instances.

Moteur	Parsing		Solving		ranking
	moy.	ec.typ.	moy.	ec.typ.	
`prop`	28.56%	19.34	9.33%	7.29	
`var`	28.75%	20.15	7.01%	5.29	97.4%
`7qd`	28.07%	17.64	13.20%	7.63	

et le nombre d'arcs créés : plus il y aura d'arcs, plus le parsing prendra de temps. Cela vient du besoin de représentation des arcs en mémoire et de vérification des garanties inhérentes au DSL mais également de la manière dont sont regroupés les arcs. Par exemple, dans les instances où le nombre de variables est petit comparé au nombre de propagateurs, parser la version DSL de `var` se fait plus rapidement que les deux autres. Enfin, on observe que parser `prop` et `7qd` est comparable. Regrouper les arcs autour des propagateurs est une opération commune aux deux déclarations.

Concernant la phase de résolution, nous pouvons observer que le surcoût moyen induit par l'utilisation d'un moteur interprété est de 9,46%. De tels surcoûts proviennent vraisemblablement du manque d'optimisation dont souffre les versions interprétées. Dans ces versions, lorsqu'un arc est planifié, la liste à laquelle il appartient doit être également planifiée, si nécessaire. Ces opérations sont améliorables dans un langage de planification supportant l'optimisation directe.

Les surcoûts d'utilisation liés à `var` et `prop` sont très proches de ceux des versions natives. Ce qui n'est pas le cas de `7qd`. Dans sa version native, l'évaluation de `prioDyn` est faite de manière efficace en interrogeant directement le propagateur. Dans sa version interprétée, chaque arc doit être évalué à chaque planification. Donc, même si la version native de `7qd` est habituellement très performante, sa version interprétée est pénalisée. Sans surprise une fois de plus, les moteurs basés sur le DSL induisent un surcoût. Cependant, il est intéressant de remarquer que seule une instances sur les 39 ne préserve pas l'ordre entre les moteurs. Plus généralement, l'introduction du DSL souffre, d'une part, des évaluations multiples de critères et d'autre part, des structures de données imbriquées. Une description basée sur l'un de ces aspects sera nécessairement moins compétitive pendant la résolution. Rappelons que les moteurs de propagation dans leur version native ont fait l'objet de multiples optimisations. Bien que les version DSL introduisent des surcoûts mémoire et temporel, ils assurent toutes les propriétés naturellement. La prochaine étape d'évaluation concerne la mise en œuvre, sur un cas d'utilisation, du DSL afin de démontrer la pertinence de son utilisation.

4.3 `Golomb ruler` : un cas d'utilisation

Nous montrons maintenant, à l'aide d'un cas d'utilisation, combien il est facile de prototyper des moteurs

FIGURE 8 – Moteurs pour `Golomb ruler`.

```
1 : All : true;
2 : All as heap(wone) of {
3 :      each var as list(for) key any.var.card
4 : };
```

(a) Moteur basé sur un tas.

```
1 : All : true;
2 : All as heap(wone) of {
3 :      each var as list(for) of {
4 :           each prop.priority as list(for)
6 :           key any.prop.priority
5 : } key any.any.var.card};
```

(b) Une variante du moteur basé sur un tas.

```
1 : M : in(mark);
2 : A : true;
3 : list(wfor) of {
4 :    list(wfor) of {M key var.name},
5 :    queue(wone) of {A}
6 : };
```

(c) Moteur à deux collections.

de propagation grâce à notre langage dédié. Nous allons déclarer les moteurs de l'état de l'art et essayer deux autres moteurs ad hoc.

Le problème du `Golomb ruler` (prob006, [18]) est défini comme "un ensemble de m entiers $0 = a_1 < a_2 < \cdots < a_m$ tel que les $\frac{m(m-1)}{2}$ différences $a_j - a_i, 1 \leq i < j \leq m$ soient distinctes. Une telle règle contient m marques et est de taille a_m. L'objectif est de trouver une règle de taille minimum." [8] a montré que propager les événements en traitant d'abord ceux associés à la variable de plus petite cardinalité est une bonne stratégie pour améliorer la résolution de ce problème. Nous construisons une instance de taille $m = 10$ en utilisant MiniZinc, et instrumentons le code pour déclarer différents moteurs de propagation. En utilisant les contraintes globales disponibles dans Choco, le problème est alors modélisé grâce à 55 variables, une contrainte `allDifferent`, 11 inégalités binaires et 45 équations ternaires ; la stratégie de recherche est basée sur a, elle préserve l'ordre lexicographique et choisit la plus petite valeur de chaque variable choisie.

Tout d'abord, nous présentons les moteurs de propagation utilisés pour cette évaluation. En plus de `prop`, `var` et `7qd` présentés auparavant, nous déclarons `heap-var`, `heap-var-prio` et `2-coll`. `heap-var` (Figure 8(a)) est un moteur orienté variables basé sur un tas, comme décrit dans [8]. Les arcs sont regroupés autour des variables dans des listes, ces listes sont ensuite ajoutées dans un tas. La liste qui contient la variable de plus petite cardinalité est alors sélectionnée pour être propagée. Cette sélection est répétée tant qu'il reste des listes planifiées. Puis, nous déclarons `heap-var-prio` (Figure 8(b)), une variante de `heap-var` où les arcs, au sein de chaque liste, sont triés en fonction de la priorité de leur propagateur. L'idée ici est d'exécuter

TABLE 2 – Moteurs pour `Golomb ruler`.

Engine	$m = 10$		$m = 11$	
	`time` (sec.)	`pex` (10^6)	`time` (sec.)	`pex` (10^6)
`prop`	3.83	10.085	76.70	191.894
`var`	2.99	10.305	55.50	192.034
`7qd`	3.19	10.860	60.67	200.108
`heap-var`	2.79	10.717	53.09	196.525
`heap-var-prio`	3.31	11.242	59.84	207.973
`2-coll`	**2.69**	**9.975**	**52.48**	**185.921**

d'abord, pour une variable, les propagateurs les plus rapides. Cela devrait permettre de discriminer les propagateurs de la contrainte `AllDifferent` des autres. Enfin, nous déclarons `2-coll` (Figure 8(c)), un moteur composé de deux collections. La première collection est basée sur `M`, l'ensemble des arcs associés aux variables a. Ils sont placés dans une liste et triés par ordre lexicographique du nom de leur variable. Les événements intervenant sur la variable a_i seront traités avant ceux intervenant sur la variable a_{i+1}. Le mot-clé `wfor` garantit, avec une propagation par *balayage*, d'atteindre un point fixe local (tous les événements des arcs de `M` auront été traités). La deuxième collection est une file qui stockent les arcs restants (ceux impliqués dans les variables de différences), elle garantit un second point fixe local (`wone`). La Table 2 montre les résultats pour les différents moteurs déclarés. Le temps de résolution (`time`, en secondes) et le nombre de propagations (`per`) pour $m = 10$ et $m = 11$ y sont reportés.

Même si il n'y a pas de relation direct entre le nombre de propagations et le temps de résolution (par exemple,`7qd` demande 5.9% plus de propagations que `prop` mais prend 23.2% moins de temps), les moteurs qui exécutent moins les propagateurs sont souvent les plus rapides. Nous observons les mêmes résultats que [8] : la sélection de la variable de plus petite cardinalité permet de résoudre plus efficacement le problème. De plus, distinguer les propagateurs d'une variable grâce à la priorité, dans `heap-var-prio`, est contre-productif : cela retarde trop l'exécution des propagateurs de la contrainte `AllDifferent`. Enfin, le moteur `2-coll` est le plus efficace : son comportement est proche de `heap-var`. Par contre il ne nécessite pas d'évaluation dynamique, et passe donc mieux à l'échelle. Il est basé sur une liste et une file dont les opérations se font en temps constant. Le nombre d'exécutions de propagateurs est également le plus faible.

Ce qu'il est important que de remarquer à ce stade, c'est la facilité avec laquelle le prototypage a été effectué avec le DSL. Par trois fois, il a été possible, en quelques lignes, de définir une heuristique de propagation spécifique, reposant sur les propriétés des variables et des contraintes, et de produire un moteur de propagation sûr, interprétable et utilisable en pratique. Cela simplifie grandement le processus d'évaluation et promeut l'étude de l'ordre de révision des contraintes au sein de CSP.

5 Conclusion et Travaux Futurs

Notre première motivation, dans cet article, était de fournir un outil qui simplifie la configuration du moteur de propagation dans un solveur de contraintes. Après avoir rappelé les différents variantes de l'algorithme de propagation (orienté arcs, variables et propagateurs), ainsi que les techniques modernes pour les rendre plus efficaces, nous avons présenté un langage dédié ouvert et extensible qui permet de décrire, simplement mais efficacement, des moteurs de propagation sans connaissance spécifique d'un langage de programmation. Nous avons listé les pré-requis nécessaires pour bénéficier totalement du langage ainsi que les propriétés et garanties qu'il fournit. Nous avons discuté des faiblesses de notre approche. Enfin, nous avons évalué les surcoûts liés à l'utilisation du langage lors des phases de parsing et de résolution. Nous avons montré l'expressivité de notre proposition et la flexibilité qu'il permet d'apporter aux solveurs. En plus des moteurs communément utilisés, il permet la déclaration de schémas de propagation très précis.

Il existe également des limites à la configuration des moteurs de propagation par notre approche, traçant ainsi le chemin des travaux futurs. Tout d'abord, il est important d'empêcher la création de moteur mal formés en proposant des règles de transformation et d'optimisation. La reformulation des contraintes et son impact sur les moteurs de propagation devra également être étudiée. Notre DSL pourrait alors faire partie intégrante des standards pour la programmation par contraintes. Il convient enfin d'étudier notre approche sur des problèmes réels, et d'analyser plus finement son interaction avec la stratégie de recherche.

Références

[1] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1) :99–118, 1977.

[2] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inf. Sci.*, 19(3) :229–250, 1979.

[3] P. Van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *AI*, 57(2-3) :291–321, 1992.

[4] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *PLILP '97*, pages 191–206, 1997.

[5] F. Laburthe. Choco : Implementing a CP kernel. In *TRICS*, pages 71–85, 2000.

[6] A. van Deursen, P. Klint and, J. Visser Domain-specific languages : an annotated bibliography In *SIGPLAN Not.*, pages 26–36, 2000.

[7] A. López-Ortiz, CG. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *IJCAI*, pages 245–250, 2003.

[8] F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the constraint satisfaction problem. In *CP'04*, pages 9–43, 2004.

[9] C. Bessiere. Constraint propagation. Technical report 06020, LIRMM, 2006.

[10] I. P. Gent, C. Jefferson, and I. Miguel. Minion : A fast scalable constraint solver. In *Proceedings of ECAI 2006*, pages 98–102. IOS Press, 2006.

[11] I. P. Gent and C. Jefferson, and I. Miguel. Watched literals for constraint propagation in minion. In *Procedings CP 2006*, pages 182–197, 2006.

[12] I. Katriel Expected-Case Analysis for Delayed Filtering In *Lecture Notes in Computer Science*, pages 119–125, Springer Berlin Heidelberg, 2006.

[13] M. Z. Lagerkvist and C. Schulte. Advisors for incremental propagation. In *CP 2007*, pages 409–422, 2007.

[14] I. P. Gent, I. Miguel, and P. Nightingale. Generalized arc consistency for the alldifferent constraint : An empirical survey. *Artificial Intelligence*, 172(18) :1973–2000, 2008.

[15] C. Schulte and P. J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 2008.

[16] M. Z. Lagerkvist and C. Schulte. Propagator groups. In Ian Gent, editor, *CP'09*, pages 524–538, 2009.

[17] C. Schulte and G. Tack. Implementing efficient propagation control. In Christopher Jefferson, Peter Nightingale, and Guido Tack, editors, *TRICS 2010*, 2010.

[18] I. P. Gent and T. Walsh. Csplib : a benchmark library for constraints. Technical report, APES-09-1999, 1999.

[19] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck and G. Tack MiniZinc : Towards a standard CP modelling language. In *CP'07*, pages 529–543, 2007.

[20] IBM ILOG CPLEX Optimizer. http ://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/, 2009.

[21] Gecode : Generic Constraint Development Environment. http ://www.gecode.org, 2012.

[22] JaCoP : Java Constraint Solver. http ://www.jacop.eu/, 2012.

[23] or-Tools : Operations Research Tools developed at Google. http ://code.google.com/p/or-tools/, 2012.