Activation de règles CHR pour la composition et l'orchestration de solveurs par contraintes

Victor Allard, Vincent Barichard, Eric Monfroy, Jean-Michel Richer

LERIA, Université d'Angers

{victor.allard,vincent.barichard,eric.monfroy,jean-michel.richer}@univ-angers.fr

Résumé

CHR est un langage à base de règles et de contraintes pour modéliser et résoudre des problèmes combinatoires. Contrairement aux solveurs dédiés à un langage de modélisation tel que CSP, l'utilisateur écrit les règles qui simplifieront le problème jusqu'à sa résolution. Nous étudions ici différents algorithmes de résolution (méthodes de filtrage et d'énumération), ainsi que leur composition par activation et désactivation des composants. Il est alors possible de définir des stratégies de résolution flexibles et adaptatives, évoluant durant la résolution. Ce travail préliminaire en CHR++*, ouvre la voie à la conception de solveurs performants pouvant ajuster leur stratégie dynamiquement durant la résolution.*

Mots-clés

stratégie, composition de solveurs, Constraint handling rules, Constraint (logic) programming, CSP, CHR++

Abstract

CHR is a rule- and constraint-based language for modeling and solving combinatorial problems. Unlike solvers dedicated to a modeling language such as CSP, the user writes the rules that simplify the problem until it is solved. Here, we examine various solving algorithms (filtering and enumeration methods), as well as their composition through the activation and deactivation of components. It is then possible to define flexible and adaptive solving strategies that evolve during resolution. This preliminary work in CHR++ *paves the way for the design of high-performance solvers capable of dynamically adjusting their strategy during resolution.*

Keywords

strategy, solvers composition, Constraint handling rules, Constraint logic programming systems, CSP, CHR++

1 Introduction

La résolution de problèmes combinatoires sous contraintes est un domaine largement étudié, offrant de nombreuses applications concrètes, notamment en logistique [11], planification [5], allocation de ressources [13], ... Pour traiter ces problèmes, un large éventail de méthodes a été développé, qu'elles soient exactes ou approchées.

Dans le cadre des approches exactes, plusieurs méthodes ont été développées parmi lesquelles : méthode analytique de réécriture (méthode de Gauss), méthode de calcul numérique (Simplex), méthode d'exploration arborescente (élagage alpha-bêta [8], CSP [10], ...). Certaines de ces approches, notamment les CSPs, reposent sur un langage de modélisation de haut niveau associé à un solveur chargé de résoudre le problème. Ces solveurs sont construits autour d'un ensemble de composants, chacun étant dédié à un aspect clé de la méthode. Le formalisme CSP, en particulier, permet de modéliser précisément les propriétés du problème avant de les soumettre à un solveur. Celui-ci est généralement conçu avec une architecture statique, où les composants et algorithmes sont sélectionnés dès l'implémentation, en fonction des performances moyennes observées. Dans ce travail préliminaire, nous nous proposons de définir un langage de stratégie permettant d'organiser dynamiquement les composants d'un solveur. Nous concentrerons d'abord notre étude sur les composants des solveurs CSP afin d'analyser les opérateurs existants, d'enrichir leur palette par l'ajout de nouveaux opérateurs et d'élaborer des stratégies simples. Nous présentons les CSPs ainsi que les différents composants que nous étudierons en section 2. La section 3 présente CHR, notre langage de représentation et d'implémentation. La section 4 est consacrée à l'étude expérimentale des différents composants et l'analyse des premiers résultats. Nous concluons en section 5, en proposant des perspectives d'amélioration.

2 CSP

Un problème modélisé par un CSP est fourni en entrée d'un solveur en charge de le résoudre.

Définition 1 (CSP) *Un CSP (Constraint Satisfaction Problem) est un problème modélisé sous forme de contraintes. X est l'ensemble des variables $\{X_1, \ldots, X_n\}$ qui prennent leurs valeurs dans un ensemble de domaines $D = \{D_1, \ldots, D_n\}$, avec $X_i \in D_i$ $\forall i \in [1, n]$. Enfin, $C = \{C_1, \ldots, C_m\}$ est l'ensemble des contraintes exprimées sur les variables. Résoudre un CSP revient à trouver une assignation, pour chaque*

variable de X, qui satisfait toutes les contraintes de C.

Les solveurs CSP, tels que Gecode [12], offrent la possibilité de paramétrer la recherche de solutions en sélectionnant un ensemble d'heuristiques pour guider les processus de branchement et de filtrage. Gecode permet notamment de configurer les stratégies de branchement, de contrôler les mécanismes de redémarrage de la recherche et de sélectionner la puissance de certains opérateurs de filtrage [1].
Cependant, ces paramètres restent statiques, car les choix de configuration sont fixés avant le démarrage de la recherche et ne peuvent pas être adaptés dynamiquement en cours d'exécution. Pour étudier, réorganiser et combiner les différents composants d'un solveur, il est essentiel de les implémenter dans un cadre flexible qui permette leur réutilisation sans nécessiter un redéveloppement complet. Pour cela, l'utilisation d'un langage de haut niveau spécialement adapté à la conception de solveurs est indispensable.

3 CHR

CHR (*Constraint Handling Rules*) est un langage à base de règles, offrant un haut niveau d'abstraction. Il permet de décrire les propriétés d'un problème, mais aussi d'étendre ses capacités en ajoutant de nouvelles règles. À la fois langage de modélisation et de programmation, il constitue un cadre très adapté au développement de solveurs. CHR est utilisé pour implémenter et étudier les différents composants d'un solveur CSP. Ces composants sont implémentés sous forme de règles, pouvant être activées ou désactivées dynamiquement. Cette approche permet de composer et d'orchestrer efficacement les différents éléments du solveur.

3.1 Présentation

CHR est un langage de programmation à chaînage avant à base de règles et de contraintes [6], qui nécessite un langage hôte pour être implémenté. CHR permet de déclarer des règles qui réécrivent des contraintes (des relations entre variables logiques), sous certaines conditions, jusqu'à la rencontre d'un critère de terminaison. Nous utilisons ici CHR$^{\veebar}$ [3], une extension de CHR intégrant la disjonction. Cette extension est particulièrement adaptée à la modélisation de problèmes combinatoires et à la conception d'algorithmes de résolution efficaces reposant sur une exploration arborescente.

3.1.1 Règles CHR

Un programme CHR est un ensemble fini de règles. Dès lors qu'une contrainte est activée, le programme recherche les règles qui peuvent être déclenchées. Chaque règle se compose d'une tête, d'une garde optionnelle et d'un corps. La tête est une conjonction de contraintes qui doivent être présentes dans le store de contraintes pour que la règle puisse s'appliquer. La garde est une condition qui doit être satisfaite pour autoriser l'activation de la règle. Le corps définit une séquence de contraintes qui sont ajoutées au but lors du déclenchement de la règle. Il existe trois types de règles :

- Propagation : $H \Rightarrow G \mid B$
- Simplification : $H \Leftrightarrow G \mid B$
- Simpagation : $H_k \setminus H_r \Leftrightarrow G \mid B$

Où H, H_k et H_r représentent des multiensembles de contraintes, G une garde et B le corps d'une règle. Dans une règle de propagation, les contraintes H utilisées pour le déclenchement sont conservées dans le store de contraintes. Dans une règle de simplification, les contraintes H sont supprimées du store de contraintes au moment de son activation. Lors de l'activation d'une règle de simpagation, le multiensemble de contraintes H_k est conservé, tandis que le multiensemble de contraintes H_r est supprimé du store de contraintes.

3.1.2 Exemple de programme CHR

Le programme CHR présenté à la figure 1 permet de calculer le plus grand commun diviseur (PGCD) de deux nombres. Initialement, deux contraintes représentant les conditions de départ du problème (les deux nombres dont on souhaite déterminer le PGCD) sont ajoutées. Ensuite, la règle 2 réécrit les contraintes de la tête en d'autres plus simples jusqu'à ce que la règle 1 soit activée, mettant ainsi fin au calcul.

$$gcd(0) \qquad \Leftrightarrow \top \qquad (1)$$
$$gcd(N) \setminus gcd(M) \qquad \Leftrightarrow N \leq M \mid gcd(M - N) \qquad (2)$$

FIGURE 1 – Programme CHR du PGCD

3.2 Solveur CSP en CHR

Nous proposons d'étudier les composants d'un solveur CSP, tels que les opérateurs de filtrage et la fonction d'énumération. Chacun de ces composants est implémenté sous forme d'un ensemble de règles CHR.
Certaines heuristiques ont démontré leur efficacité comme l'ordonnancement dynamiquement de variables (DVO) [1]. Cependant, il existe plusieurs heuristiques DVO concurrentes et la représentation de ces heuristiques dans des composants interchangeables offre la possibilité de choisir l'heuristique DVO avant la recherche et d'en changer pendant la recherche.
La figure 2 illustre deux règles implémentant des stratégies d'énumération distinctes. La règle dom explore successivement toutes les valeurs du domaine d'une variable, tandis que bin applique un branchement binaire : elle assigne une valeur à la variable sur une branche et l'exclut sur l'autre. Pour enchaîner les points de choix, les contraintes CHR $n_{var}()$ et $n_{val}()$ sont postées. Bien que non détaillées ici, ces contraintes permettent d'imbriquer récursivement les points de choix en ajoutant les contraintes CHR nécessaires pour déclencher l'une des deux règles présentées dans la figure 2. Dans la

1. Voir Chapitres 8, 9 et Partie P : https://www.gecode.org/doc/6.2.0/MPG.pdf

règle dom, la variable hôte d permet de propager le domaine modifié à chaque opérande de la disjonction et la contrainte $n_{dom}(d, A)$ fixe la variable logique A à la prochaine valeur branchée sur le domaine d.

$$v(X, D_x) \setminus b_{var}(X), b_{val}(V) \Leftrightarrow \\ d \leftarrow D_x, \\ \bigvee_{y \in d} (n_{dom}(d, A), d \leftarrow d/\{A\}, D_x \leftarrow \{x\}, n_{var}(), n_{val}()) \quad \text{(dom)}$$

$$v(X, D_x) \setminus b_{var}(X), b_{val}(V) \Leftrightarrow \\ (D_x \leftarrow \{V\}, n_{var}(), n_{val}() \quad \text{(bin)} \\ \vee (D_x \leftarrow D_x/\{V\}, b_{var}(X), n_{val}())$$

FIGURE 2 – Énumérations en CHR$^{\forall}$

Un solveur CSP sélectionne un type d'énumération à priori, avant le début de la résolution. Pour permettre un changement dynamique de stratégie d'énumération, nous activons ou désactivons certaines règles en cours d'exécution. Cette approche impacte directement le comportement du programme et modifie ainsi le fonctionnement initial du solveur. Dans notre étude expérimentale, nous avons réimplémenté plusieurs composants d'un solveur CSP. Implémentés sous forme de règles CHR, ces composants peuvent être activés ou désactivés dynamiquement selon les besoins.

4 Expérimentations

L'étude expérimentale présentée ici vise à combiner et évaluer différentes versions des composants d'un solveur CSP en les appliquant à un problème bien connu : le problème des N-Reines.

4.1 Le problème des N-Reines.

Le problème des N-Reines est un problème classique de la programmation par contraintes. Il consiste à placer n reines sur un échiquier de taille $n \times n$ de manière à ce qu'aucune d'elles ne puisse en attaquer une autre, que ce soit sur une même ligne, colonne ou diagonale. La figure 3 propose un modèle CSP permettant de calculer une solution à ce problème, si elle existe. La taille de l'échiquier $n \in \mathcal{N}$ est un paramètre du modèle qui permet de générer l'instance à résoudre. La contrainte *all_diff(S)* est satisfaite si toutes les variables assignées de S prennent des valeurs distinctes.

$$X = \{X_1, \ldots, X_n\} \\ D = (D_i = 1..n \mid i \in 1..n) \\ C = \{all_diff(X), \\ \quad all_diff(\{X_i + i \mid i \in 1..n\}), \\ \quad all_diff(\{X_i - i \mid i \in 1..n\})\}$$

FIGURE 3 – Modèle CSP des N-Reines

Nous avons traduit ce modèle en CHR, où la contrainte *all_diff()* a été convertie en un ensemble de règles assurant le test de satisfiabilité, le filtrage de domaine (forward checking [7]) et le maintien de l'arc-consistance.

4.2 Résultats

Ce travail est validé expérimentalement par l'utilisation du solveur `CHR++` qui est une implémentation CHR efficace en `C++` qui supporte le non-déterminisme *don't know* [3].
Nous avons choisi les composants suivants pour implémenter le solveur CHR : branchement sur une variable $Vr \in \{first, min, deg\}$, branchement sur une valeur $Vl \in \{first\}$, contrainte $Ctr \in \{diff\}$, propagateur $Pr \in \{none, diff\}$, consistance $Cs \in \{none, ac_1\}$, énumération $Lb \in \{dom, bin\}$.

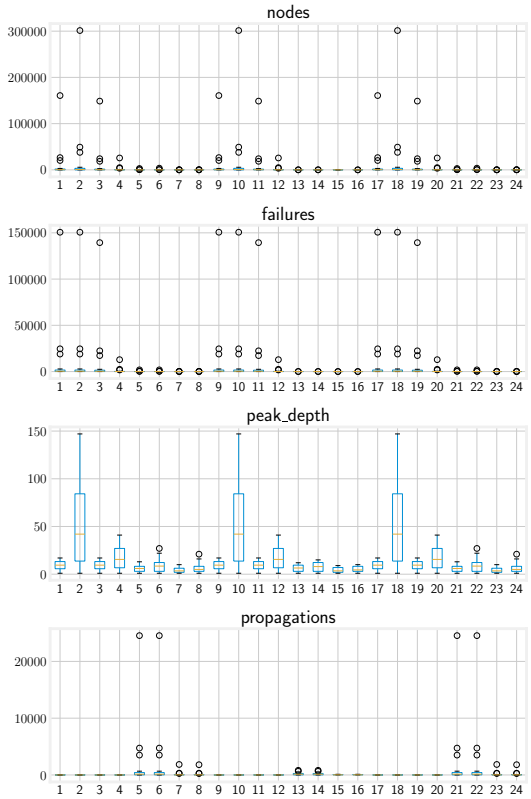

FIGURE 4 – Distribution des indicateurs de recherche sur les N-Reines, N variant de 1 à 16.

Nous avons testé différentes configurations de solveurs CSP en cherchant à déterminer la satisfiabilité des instances $1 \leq N \leq 16$ du modèle des N-Reines. La figure 4 montre la distribution des indicateurs de recherche : nombre de nœuds de l'arbre de recherche (nodes), nombre d'échecs après avoir fixé une variable CSP (failures), profondeur de l'arbre de recherche (peak_depth), nombre de propagations effectuées (propagations). Les 24 compositions différentes ($Vr \times Vl \times Ctr \times Pr \times Cs \times Lb$) ont été testées. L'analyse qui suit se concentre sur la composition numéro 15 de la figure 4, car elle a une des moyennes et des écarts

type la plus basse sur les indicateurs node, failure et peak_depth.

La figure 5 montre l'évolution des indicateurs de recherche pour la composition suivante : $Vr = min$ qui branche en priorité les variables qui ont le plus petit domaine, $Vl = first$ qui branche les valeurs dans l'ordre de définition, $Ctr = diff$ qui représente la contrainte binaire $\neq_2$, $Pr = diff$ qui représente le propagateur de la contrainte binaire $\neq_2$, $Cs = AC_1$ qui représente une implémentation naïve de l'arc consistance [9], $Lb = dom$ qui correspond à la règle dom de la figure 2. On remarque sur cette instance que le nombre de nœuds est proportionnel au nombre d'échecs, ce qui est prévisible, car chaque échec produit de nouveaux points de choix. Les instances $N \in \{5, 7, 9, 16\}$ se démarquent des autres, car elles sont satisfiables et car le maintien de l'arc consistance a permis d'éviter les échecs lors de la recherche. L'étude de ce type d'instance pourrait s'avérer intéressante pour détecter quand l'activation de l'arc consistance est utile.

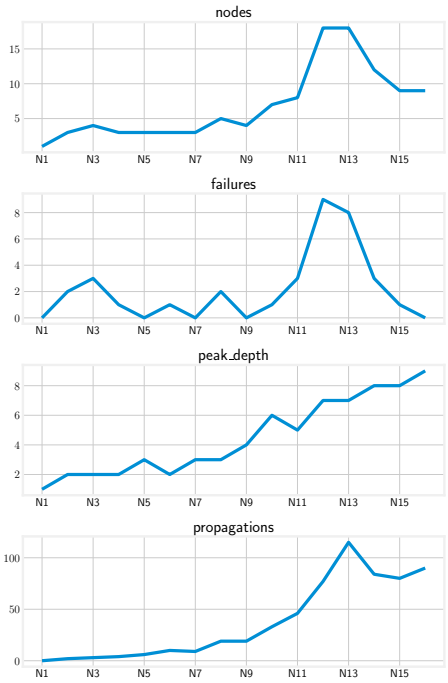

FIGURE 5 – Indicateurs pour la composition 15

5 Conclusion

L'objectif de ce travail est de présenter un système de stratégies au-dessus de CHR qui permet à la manière d'`ELAN` [4] et de `Tom` [2], de définir la sémantique à appliquer pour exécuter un ensemble de règles CHR. L'élaboration d'un langage déclaratif d'application et de combinaison d'opérateurs de stratégies est en cours d'élaboration avec pour objectif de l'intégrer dans le solveur `CHR++`. La prochaine étape de ce travail est donc de trouver des stratégies efficaces qui font varier les composants du solveur CSP pendant la recherche.

Remerciements Nous tenons à remercier l'université d'Angers (UA) et la région Pays de la Loire qui ont financé ce travail.

Références

[1] Fahiem Bacchus and Paul van Run. Dynamic variable ordering in CSPs. In Ugo Montanari and Francesca Rossi, editors, *Principles and Practice of Constraint Programming — CP '95*, pages 258–275, Berlin, Heidelberg, 1995. Springer.

[2] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom : Piggybacking rewriting on java. In Franz Baader, editor, *Term Rewriting and Applications*, pages 36–47. Springer Berlin Heidelberg, 2007.

[3] Vincent Barichard. CHR++ : An efficient CHR system in C++ with don't know non-determinism. *Expert Systems with Applications*, 238 :121810, 2024.

[4] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN : A logical framework based on computational systems. *Electronic Notes in Theoretical Computer Science*, 4 :35–50, 1996. RWLW96, First International Workshop on Rewriting Logic and its Applications.

[5] Jeremy Frank and Ari Jónsson. Constraint-based attribute and interval planning. *Constraints*, 8(4) :339–364, Oct 2003.

[6] Thom Frühwirth. *Constraint handling rules*. Cambridge University Press, New York, 2009.

[7] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *AI*, 14(3) :263–313, 1980.

[8] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *AI*, 6(4) :293–326, 1975.

[9] Alan K. Mackworth. Consistency in networks of relations. *AI*, 8(1) :99–118, 1977.

[10] Stuart Russell and Peter Norvig. *Artificial Intelligence, Global Edition A Modern Approach*, chapter Chapter 5 : Constraint Satisfaction Problems. Pearson Deutschland, 2021.

[11] Abdelkader Sbihi and Richard W. Eglese. Combinatorial optimization and Green Logistics. *4OR : A Quarterly Journal of Operations Research*, 5(2) :99–116, June 2007. 18 pages.

[12] Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 619–633, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[13] G. Tychogiorgos and K.K. Leung. Optimization-based resource allocation in communication networks. *Computer Networks*, 66 :32–45, 2014. Leonard Kleinrock Tribute Issue : A Collection of Papers by his Students.