Rapport Stage CHR++

Tom Colombu

Sujet :

Transformation XCSP vers CHR++

Encadré par :
Vincent Barichard,
Marc Legeay.

université
angers

Sommaire

I. Introduction

I-1 Sujet du stage

CHR (Constraint Handling Rules) est un langage de règles à chaînage avant qui permet de définir des contraintes déclaratives au sens de la programmation logique par contraintes [3]. C'est une extension de langage qui permet d'introduire des contraintes définies par l'utilisateur, c'est-à-dire des prédicats du premier ordre, dans un langage hôte donné comme Prolog, Lisp, Java ou C/C++. CHR++, une implémentation de CHR au-dessus de C++, est aujourd'hui utilisé dans plusieurs projets de recherche au LERIA. Les CSP (Constraint Satisfaction Problems) permettent de modéliser des problèmes de satisfaction de contraintes. Un CSP est défini par un ensemble de variables, chacune à valoir dans un domaine, liées entre elles par des contraintes. Une solution à un CSP est une affectation de toutes les variables satisfaisant la totalité des contraintes. Plusieurs formalismes existent pour décrire des CSP, notamment XCSP3, un langage de description d'un CSP basé sur XML. XCSP fournit à l'utilisateur un ensemble d'éléments de modélisation dont un large catalogue de contraintes prédéfinies prêtes à l'usage. Ce catalogue intègre des contraintes arithmétiques, ensemblistes, et un ensemble de contraintes globales comme linear et alldiff. Transformer un CSP en programme CHR, exécutable avec CHR++ implique d'écrire toutes les règles CHR garantissant qu'une solution du programme CHR sera également une solution du CSP. Une façon d'aboutir à ce résultat serait de traduire les contraintes du CSP en entrée en un ensemble de règles CHR réalisant des actions de filtrage analogues. Il serait alors possible de traduire intégralement un modèle CSP en programme CHR qui calculerait la solution au problème posé.

Des chercheurs du LERIA ont initié le développement d'un solveur de contraintes arithmétiques au-dessus de CHR++. Le but du stage est de continuer le développement de ce solveur en réalisant un parseur permettant de traduire une instance au format XCSP vers un programme CHR++. Ce parseur facilitera l'adoption du nouveau solveur, car il rendra possible la réutilisation directe d'un ensemble de benchmarks existants mis à disposition par la communauté travaillant à l'élaboration du format XCSP. Il permettra également d'élargir le catalogue de contraintes mises à disposition par CHR++, lui permettant de modéliser plus facilement de nouveaux problèmes.

I-2 La programmation par contraintes

La programmation par contraintes (Constraint Programming, ou CP) est une approche déclarative qui permet de modéliser et résoudre efficacement des problèmes combinatoires. Elle repose sur la définition de variables, de domaines de valeurs possibles et d'un ensemble de contraintes exprimant les relations ou restrictions que doivent satisfaire ces variables.

Contrairement à la programmation impérative, qui décrit la suite d'actions à effectuer pour résoudre un problème, la programmation par contraintes exprime les conditions que doivent remplir les solutions. Le moteur de résolution se charge ensuite de trouver une ou plusieurs affectations des variables qui respectent toutes les contraintes.

Structure générale d'un problème de contraintes :

- Variables : $x_1, x_2, \ldots, x_n$
- Domaines : chaque variable prend ses valeurs dans un domaine qui lui est propre.
- Contraintes : restrictions sur les valeurs pouvant être prises simultanément par les variables avec par exemple : alldifferent, neq, linear.

Considérons un problème de sudoku :

- Les variables sont les cases de la grille.
- Les domaines sont **{1,2,…,9}** pour chaque case.
- Les contraintes expriment que chaque ligne, chaque colonne, et chaque sous-grille 3x3 contient des valeurs toutes différentes (contraintes alldiff).

Intérêts de ce type de modélisation :

- Lisibilité : le modèle reflète directement la structure logique du problème.
- Modularité : ajout ou modification de contraintes faciles.
- Abstraction : séparation entre la description du problème et l'algorithme de résolution.
- Richesse de modélisation : possibilité de modéliser des problèmes complexes issus de domaines variés.

Résolution

Une fois le problème modélisé, un solveur de contraintes explore les différentes affectations possibles des variables, souvent à l'aide de techniques de propagation et de backtracking, afin de trouver une ou plusieurs solutions satisfaisant l'ensemble des contraintes.

Positionnement dans ce stage

Dans le cadre de ce stage, la programmation par contraintes est au cœur du projet. Elle permet d'exploiter des spécifications de problèmes combinatoires exprimées au format XCSP3. Ces modèles sont ensuite traduits automatiquement en CHR++, un langage fondé sur les règles de réécriture de contraintes.

Cette traduction vise à établir une passerelle entre la richesse du format de modélisation XCSP3 et la puissance de raisonnement offerte par CHR++, facilitant ainsi l'expérimentation, la création de solveurs spécialisés et la constitution d'un benchmark.

I-3 Le filtrage de domaines

Dans la programmation par contraintes, le filtrage est un mécanisme fondamental qui consiste à réduire les domaines des variables en éliminant les valeurs qui ne peuvent pas participer à une solution valide, compte tenu des contraintes. Ce processus est effectué automatiquement par le solveur avant ou pendant l'exploration de l'espace des solutions.

L'objectif du filtrage est d'élaguer autant que possible l'espace de recherche, ce qui permet d'accélérer considérablement la résolution. Par exemple, si une contrainte interdit que deux variables aient la même valeur, et qu'une d'elles est fixée, le filtrage peut immédiatement retirer cette valeur du domaine de l'autre.

Dans le cadre de ce stage, le filtrage joue un rôle crucial, notamment parce qu'il doit être reconstitué dans la traduction vers CHR++. En effet :

- Le format XCSP3 décrit les contraintes mais ne spécifie pas les mécanismes de filtrage associés.

- Dans CHR++, ce filtrage doit être explicitement programmé sous forme de règles : il faut donc traduire les contraintes XCSP en règles CHR++ capables de propager les effets des contraintes sur les domaines.

Cela demande une bonne compréhension de la sémantique des contraintes et de leur comportement algorithmique, afin de produire des règles capables de reproduire un filtrage efficace, comparable à celui d'un solveur classique. Ce travail de traduction est essentiel pour garantir la qualité et la performance du solveur CHR++.

II. Présentation des langages

Les problèmes de satisfaction de contraintes (CSP) peuvent être modélisés de nombreuses façons différentes, selon le langage utilisé et le moteur de résolution visé. Chaque environnement propose généralement ses propres structures, notations et syntaxes adaptées à son fonctionnement interne.

Pour pallier cette diversité et faciliter l'échange, la comparaison ou la réutilisation des instances de problèmes, un format universel a été défini : XCSP, un langage neutre basé sur XML, permettant une représentation standardisée et indépendante des outils de résolution.

II-1 Le format XCSP3

Le format XCSP3 (pour XML Constraint Satisfaction Problems, version 3) est un langage standardisé, déclaratif et extensible conçu pour représenter une grande variété de problèmes de satisfaction et d'optimisation sous forme de fichiers XML.

1 - Objectifs et avantages

XCSP3 vise à répondre à plusieurs objectifs :

- Fournir une représentation claire, lisible et structurée des modèles de contraintes ;
- Permettre la modélisation de familles variées de problèmes : satisfaction (CSP), optimisation (COP), satisfaction booléenne (SAT), et extensions (....)
- Assurer une compatibilité avec les solveurs modernes en facilitant le parsing et la transformation automatique
- Servir de base à l'analyse, la visualisation ou l'apprentissage automatique sur des instances de problèmes.

Parmi ses atouts figurent son extensibilité, son support riche de types de variables et contraintes, et sa compatibilité avec des outils d'analyse syntaxique et sémantique.

2 - Structure d'un fichier XCSP3

Un fichier XCSP3 est organisé en balises XML, structurées selon le modèle suivant :

- <instance format="XCSP3" type="CSP"> : déclaration d'une instance de type CSP
- <variables> : déclaration des variables et de leurs domaines (finis ou énumérés)
- <constraints> : ensemble des contraintes entre les variables, exprimées sous différentes formes

Les contraintes peuvent être exprimées de plusieurs manières :

- Intentionnelles : sous forme d'expressions booléennes ou arithmétiques, par exemple eq(x,y), lt(x,5)).
- Extensionnelles : par énumération explicite des tuples autorisés ou interdits.
- Globales : à l'aide de contraintes prédéfinies comme alldifferent par exemple.

3 - Le modèle du Sudoku en XCSP3

Le format XCSP3 permet de décrire des problèmes de manière structurée en XML comme celui du sudoku par exemple. Ce problème classique de satisfaction de contraintes où l'on doit remplir une grille 9×9 avec les chiffres de 1 à 9 tout en respectant certaines contraintes d'unicité.

Dans les grandes lignes une instance de Sudoku exprimée en XCSP3 représenterait :

- La déclaration des **variables** et de leurs **domaines** :

  Ici on crée une variable correspondant à une matrice de taille 9 par 9 et où chaque position de la matrice à un domaine initial dans l'intervalle [1,9].

```
<variables>
      <array id="x" size="[9][9]"> 1..9 </array>
</variables>
```

- La déclaration d'un bloc de **contraintes** avec la balise : <constraints>

- Les contraintes **allDifferent** exprimant les règles du jeu de Sudoku sur les lignes, colonnes et blocs internes. Par exemple, on peut modéliser ainsi la règle du jeu du Sudoku sur les lignes tel que :

```
<group class="rows">
      <allDifferent> %... </allDifferent>
            <args> x[0][] </args>
             <args> x[1][] </args>
             ...
             <args> x[8][] </args>
</group>
```

- Les contraintes **d'instanciation** pour les valeurs initiales de la grille :

```
<instantiation class="clues">
        <list> x[0][1] x[0][6..8] x[1][2] ... x[8][7] </list>
        <values> 4 1 7 9 2 8 5 4 6 5 ... </values>
</instantiation>
```

4 - Limites du langage

Malgré sa richesse et sa large adoption dans la communauté des CSP, le langage XCSP3 présente certaines limites qui peuvent freiner son exploitation dans des environnements comme CHR++. Conçu comme un langage de description de haut niveau, XCSP3 se concentre sur la modélisation déclarative de problèmes, en séparant clairement cette étape de celle de la résolution. Il fournit un catalogue important de contraintes prédéfinies, utilisables dans une syntaxe XML structurée. Toutefois, cette approche descriptive implique que l'utilisateur doit entièrement spécifier les faits et contraintes du problème, ce qui rend la modélisation parfois fastidieuse et rigide, notamment lorsque les contraintes sont nombreuses ou évolutives.

En effet, la syntaxe XML, bien que normalisée, peut devenir lourde et peu lisible, surtout pour des modèles complexes avec de nombreuses variables et relations. Cette lourdeur complique à la fois la lecture manuelle et le traitement automatique. De plus, le format ne propose pas de mécanisme natif pour l'induction ou la génération dynamique de faits ou de règles : toute la structure du problème doit être décrite explicitement, ce qui limite l'adaptabilité à des contextes dynamiques.

Par ailleurs, XCSP3 ne permet pas la définition directe de contraintes personnalisées : le langage repose sur un ensemble fermé de contraintes, ce qui réduit la flexibilité pour des problèmes spécifiques ou non standards. Enfin, bien qu'il soit excellent pour la modélisation, XCSP3 ne fournit aucun support pour la résolution elle-même : il n'est pas un langage de programmation, et ne permet pas de décrire des stratégies ou des algorithmes. Il est donc dépendant d'un solveur externe, et son intégration dans un environnement comme CHR++, plus orienté vers l'exécution et l'adaptabilité, exige une phase de traduction non triviale.

II-2 Le langage CHR et CHR++

1- CHR

CHR (Constraint Handling Rules) est un langage déclaratif introduit en 1991 par Thom Frühwirth, dans le but d'implémenter des solveurs de contraintes de manière concise, modulaire et réutilisable. CHR se concentre sur la réécriture dirigée par contraintes, en manipulant directement les contraintes présentes dans un environnement appelé store.

Un programme CHR est constitué d'un ensemble de règles, chacune agissant sur certaines contraintes présentes dans le store pour les simplifier, en ajouter d'autres, ou les transformer. L'exécution est entièrement pilotée par un moteur de règles qui applique de manière automatique toutes les règles dont les conditions sont satisfaites.

CHR permet non seulement de modéliser mais aussi d'exécuter directement les contraintes via des règles personnalisées, offrant ainsi une flexibilité de résolution que XCSP3, limité à la description, ne fournit pas.

2- CHR++

CHR++ (Constraint Handling Rules++) est une extension du langage CHR, intégrée dans l'environnement C++, conçue pour écrire des programmes déclaratifs fondés sur la réécriture de contraintes. Contrairement à XCSP3, qui se limite à une description statique du problème, CHR++ est un langage de programmation à part entière, positionné à un niveau plus opérationnel dans la hiérarchie de modélisation.

Un des avantages fondamentaux de CHR++ réside dans sa capacité à induire dynamiquement de nouveaux faits à partir de règles existantes. Là où, comme mentionné précédemment, XCSP exige la définition explicite de toutes les relations du problème, CHR++ permet de raisonner à partir d'un nombre réduit d'axiomes ou de contraintes initiales, les autres étant déduites par propagation ou réécriture. Cette capacité rend le langage particulièrement puissant pour les problèmes évolutifs, incomplets ou ouverts.

CHR++ offre également une grande extensibilité : les utilisateurs peuvent définir des contraintes sur mesure, intégrer des stratégies de recherche spécifiques, et exploiter l'environnement C++. Ces qualités viennent toutefois avec un coût en complexité : modéliser en CHR++ demande en effet un effort plus important qu'en XCSP3, mais en contrepartie, il permet un contrôle total sur la résolution et une flexibilité bien supérieure.

Principes de base de CHR++

Un programme CHR++ est un ensemble de règles qui s'appliquent sur un store de contraintes. Ces règles peuvent être simultanément réactives et déclaratives, et permettent d'exprimer facilement des relations complexes entre les variables.

Un fichier CHR++ est constitué de plusieurs éléments principaux :

- Une déclaration de nom de programme (<CHR name="nom">)
- Une section de déclaration des contraintes (<chr_constraint>)
- Un corpus de règles (simples, simplification, propagation, propagation retardée…)
- Éventuellement, des fonctions C++ externes appelées depuis les règles
- Une fonction main qui peut permettre de déclarer d'intéragir avec le space / store

1 - Particularités syntaxiques CHR++

- Variables logiques : typées via ?int, +int, -interval, etc.
- Appels de fonctions C++ autorisés dans les gardes ou corps : Solvint::lt(...), (*Dom).empty().
- Règles anonymes : le nom est facultatif.
- Support de structures personnalisées : comme des listes, intervalles.

2 - Les différents types de règles

CHR++ reprend les trois formes classiques de CHR, avec quelques variantes syntaxiques :

1. Règle de simplification

Supprime les contraintes de la tête et les remplace par d'autres au sein du store.

Syntaxe :

```
nom @ contrainte1, contrainte2 <=> garde | nouvelle_contrainte1, nouvelle_contrainte2 ;;
```

Exemple :

```
CspVarInt(X, Dom), eq(X, V) <=> V.ground() |CspVarInt(X, V) ;;
```

Dans cette situation, nous avons un cas où lorsque V est fixé alors on remplace le domaine de X (Dom) par V en substituant CspVarInt(X,Dom) dans le store par CspVarInt(X,V) et on retire simplement eq(X,V).

2. Règle de propagation

Ajouter de nouvelles contraintes sans supprimer celles de la tête.

Syntaxe :

```
nom @ contrainte1, contrainte2 ==> garde | nouvelle_contrainte1, nouvelle_contrainte2 ;;
```

Exemple :

```
neq_var_var @ CspVarInt(X, DX), CspVarInt(Y, DY), neq(X, Y) ==> Solvint::ne(DX, DY) ;;
```

Ici, la contrainte neq(X, Y) est propagée vers une mise à jour des domaines DX et DY via la fonction de propagation.

3. Règle de simpagation (simplification + propagation)

Supprimer une partie des contraintes tout en conservant les autres et en ajoutant de nouvelles contraintes.

Syntaxe :

```
nom @ contrainte1 \ contrainte2 <=> garde | nouvelle_contrainte1 ;;
```

Exemple :

```
CspVarInt(X, XDom), CspVarInt(Y, YDom) \ lt(X,Y) <=> (*X).max() < (*Y).min() | success() ;;
```

La contrainte lt(X,Y) n'a plus lieu d'être si les domaines de X et Y respectent la contrainte, donc on l'enlève.
success() est une fonction fournie par CHR++ signifiant que l'application de la règle s'est bien déroulée. Cette fonction est généralement utilisée lorsqu'on ne souhaite pas poster de nouvelles contraintes à l'exécution de la règle.

4. Règle de propagation sans historique

En CHR, une règle de propagation ne peut être déclenchée qu'une seule fois. Lorsque les contraintes en tête ont déclenché une propagation, elles ne pourront pas les redéclencher si une variable est réveillée par la suite.
La propagation sans historique est une extension proposée par CHR++ qui permet de déclencher une règle de propagation même si les contraintes en tête ont déjà déclenché une première fois la règle.
Comme une règle de propagation classique, elle ajoute de nouvelles contraintes sans supprimer celles de la tête et sans mémoriser l'historique de son application.
Syntaxe :

```
nom @ contrainte1, contrainte2 =>> garde | nouvelle_contrainte1, nouvelle_contrainte2 ;;
```

Une règle de propagation (==>) mémorise son application (qui a été déclenchée avec quelles contraintes), pour ne pas être réappliquée plusieurs fois inutilement.

Cependant :

- Cette mémorisation de l'historique peut ralentir l'exécution,
- Il est parfois souhaitable de réappliquer plusieurs fois la même règle (par exemple si une contrainte est modifiée mais pas supprimée),
- Ou bien on sait que la suppression future d'une contrainte rendra une nouvelle application nécessaire.

Dans ces cas, CHR++ permet d'utiliser une règle sans historique avec =>>.
Certaines variables peuvent être mutables, c'est-à-dire que leur valeur peut changer au cours de l'exécution sans qu'elles soient supprimées ou recréées. Cela signifie que la même contrainte contenant une variable mutable peut évoluer et dans ce cas, la règle devrait être réappliquée même si elle l'a déjà été dans le passé.
C'est ici que =>> devient utile en permettant de surveiller les changements d'état d'une contrainte mutable sans être bloquée par une mémoire d'application précédente.

3 - Les traces, un outil de compréhension et de débogage

Ces traces sont essentielles à la fois pour le débogage, la compréhension du comportement du programme, et l'analyse de performances.

3.1 – Les différents types de traces en CHR++

Le système de traces de CHR++ permet de suivre en détail l'évolution de l'exécution d'un programme basé sur des règles de contraintes. Il repose sur l'enregistrement de messages structurés correspondant aux différentes étapes de traitement des contraintes dans le store. Ce système est configurable à l'aide de flags, activés typiquement par l'appel :

```
TRACE(chr::Log::register_flags(chr::Log::ALL));
```

Cependant, l'usage du flag ALL génère un volume considérable d'informations, souvent verbeuses et peu lisibles dans des cas complexes comme la résolution d'un Sudoku. Il est donc préférable de n'activer que certains types de traces, selon le besoin d'analyse.

Les traces générées peuvent être regroupées en catégories selon les étapes du cycle de vie d'une contrainte :

- Ajout et activation de contraintes : [CALL] signale l'introduction d'une contrainte dans le store, tandis que [WAKE] indique sa réactivation en réponse à une mise à jour.

- Exploration des règles : [TRY] représente une tentative d'application d'une règle pour une contrainte donnée, souvent indexée par le numéro d'occurrence dans le programme.

- Matching multi-contrainte : [PARTNER] identifie d'autres contraintes nécessaires pour compléter le matching d'une règle à plusieurs têtes.

- Application de règle : [COMMIT] marque l'application effective d'une règle avec les contraintes correspondantes.

- Historique : [HISTORY] permet de vérifier si un matching a déjà été effectué, évitant les redondances.

- Sortie du store : [EXIT] indique qu'une contrainte a été retirée (car résolue ou non applicable).

- Objectifs de résolution : [GOAL] désigne des contraintes cibles pour la propagation.

Ce système de traces constitue un outil essentiel pour mieux comprendre les mécanismes internes de propagation, de matching et d'application des règles dans un programme CHR++. Il permet non seulement de déboguer plus facilement, mais aussi d'acquérir une vision fine de la manière dont le store évolue, comment les contraintes interagissent, et pourquoi certaines règles sont (ou ne sont pas) appliquées.

3.2 – Un outil puissant… mais verbeux

Les traces permettent de nombreuses choses en somme notamment :

- Comprendre les mécanismes de matching, de déclenchement, et d'instanciation des règles CHR.
- Observer en temps réel l'évolution du store et l'application successive des règles.
- Déboguer des erreurs de modélisation, telles qu'une règle non déclenchée ou une contrainte mal déclarée.

Elles sont également précieuses pour mesurer l'impact d'une nouvelle règle ajoutée dans le système, en évaluant sa fréquence de déclenchement, ou son interaction avec d'autres.

L'un des inconvénients majeurs de l'utilisation du tracing en CHR++ est la taille exponentielle des traces générées en fonction du nombre de règles et de contraintes.
En effet :

- Chaque contrainte ajoutée peut potentiellement activer plusieurs règles.
- Chaque règle déclenchée peut à son tour créer de nouvelles contraintes, alimentant une cascade de déclenchements.
- Sur des programmes réalistes ou fortement combinatoires, comme un solveur de Sudoku, on peut obtenir des centaines de milliers de lignes très rapidement.

Par conséquent :

- Il est fortement recommandé de désactiver ou filtrer les traces pour se concentrer sur l'essentiel.
- Lors du développement, il est utile de limiter le nombre de règles testées simultanément, ou de n'afficher que certains types d'information (chr::Log::RULE, chr::Log::FAILURE…).
- Il peut être utile de rediriger la sortie dans un fichier pour faciliter l'analyse hors ligne ou l'archivage.

4 - Modélisation d'un Sudoku en CHR++ avec Solvint

La modélisation du Sudoku présentée ici s'inscrit dans le contexte du développement de Solvint, un solveur arithmétique en CHR++ actuellement en cours de conception.
Solvint est développé en parallèle du déroulement du stage, et cette modélisation constitue à la fois un cas d'usage concret et un terrain d'expérimentation pour faire évoluer et définir la syntaxe, les règles et le comportement du solveur selon les besoins identifiés.

En ce sens, nous avons défini et utilisé plusieurs constructions spécifiques, comme mentionné ci-dessous.

Ces choix syntaxiques reflètent une volonté d'adapter progressivement le parser de Solvint en fonction des contraintes rencontrées dans les problèmes concrets comme le Sudoku. De plus, certaines règles CHR++ font appel à des fonctions de Solvint via l'espace de noms Solvint:: pour exécuter les parties droites de règle avec le solveur arithmétique.

1. Déclaration des contraintes

Le fichier CHR++ définit un ensemble de contraintes nécessaires à la résolution du Sudoku. Ces contraintes incluent :

```
CspVarIntDec(+int,+string,?int,-interval),
CspVarInt(+int,?int,-interval),
neq(+int,?int,+int,?int),
put_alldiff(?list_var,?list_int),
eq(+int, ?int, +int, ?int),
labelling(+int)
```

- CspVarIntDec(+Id, +Name, ?Var, -Domain) permet de déclarer une variable avec un domaine. Avec donc un id et un nom pour identifier la variable, une variable logique et un domaine de type mutable qui évolueront grâce aux règles

- CspVarInt(+Id, ?Var, -Domain) sera la forme des variables qui sera manipulée par l'ensemble des règles. On reprend les mêmes arguments que pour CspVarIntDec.

- neq(+IdX ?X, +IdY, ?Y), eq(+IdX, ?X, IdY, ?Y) modélise les inégalités et égalité entre variables. On prendra pour arguments un couple de variables logiques et les IDs correspondants à ces variables.

- put_alldiff(?ListVars, ?ListIds) permet de spécifier qu'un ensemble de variables doit être toutes différentes. Avec ?ListVars : liste de variables et ?ListIds : liste d'identifiants correspondants.

- Enfin labelling(+Id) à un usage de résolution (section 4-6).

2. Règles de réécriture des contraintes

Les règles CHR assurent la propagation des contraintes et simplifient les domaines des variables. Par exemple, les règles suivantes réduisent les domaines lorsque deux variables sont différentes :

```
neq_var_var @ CspVarInt(Id1, X, XDom), CspVarInt(Id2, Y, YDom), neq(Id1, X, Id2, Y) ==> Solvint::ne(XDom, YDom);
```

Une règle spéciale gère également la propagation des contraintes alldiff :

```
put_alldiff(L, Ids) <=> alldiff_fun(*this, *L, *Ids);
```

La fonction alldiff_fun est définie en C++ pour ajouter toutes les inégalités binaires ≠ entre les variables de la liste.
Ainsi pour 3 variables A,B,C on aura put_alldiff([A, B, C], [IdA, IdB, IdC]) qui permet d'établir : neq(A, B) , neq(A, C), et neq(B, C)

3. Initialisation dans le `main`

L'initialisation des variables de l'instance du Sudoku se fait dans la fonction main, où chaque case est représentée par une variable logique de domaine [1..9] :

```
for (int i = 0; i < 81; ++i) {
        Dom[i] = interval(1, 9);
        space->CspVarIntDec(i, "x[i][j]", X[i], Dom[i]);
}
```

Les cases connues du Sudoku sont ensuite initialisées via :

```
set_eq(Dom[1], 4);
set_eq(Dom[6], 1);
[...]
```

4. Les Contraintes du Sudoku

Les contraintes de type alldiff sont également initialisés dans le main et on ajoute pour chaque ligne, colonne et sous-grille (blocs 3x3) :

```
// première ligne
space->put_alldiff(make_list_var(X[0], ..., X[8]), make_list_int(0, ..., 8));
// première colonne
space->put_alldiff(make_list_var(X[0], X[9], ..., X[72]), make_list_int(...));
// premier bloc
space->put_alldiff(make_list_var(X[0], X[1], X[2], X[9], ..., X[20]), make_list_int(...));
```

5. Fonctions auxiliaires en C++

Des fonctions ont été ajoutées pour la gestion de certaines contraintes tel que :

```
inline chr::ES_CHR set_eq(chr::Logical_var_mutable<interval> &x, int v) {
        return x.update_mutable(...);
}
```

De plus, on ajoute des fonctions d'affichage notamment qui permettent le débogage.

6. Résolution

La stratégie de labellisation est activée via une contrainte labelling(Id) qui tente successivement toutes les valeurs possibles pour chaque variable :

```
labelling(Id) <=> exists_it(i, *Dom, (set_eq(Dom,i), labelling(Id+1)));
```

La fonction exists_it(..) joue ici un rôle essentiel : elle permet d'explorer l'espace de recherche en créant un nœud pour chaque valeur possible dans le domaine de la variable concernée.
Pour chaque valeur i, la contrainte set_eq(Dom, i) fixe la variable à cette valeur, puis relance récursivement la labellisation sur la variable suivante.

Le processus s'arrête lorsque toutes les variables sont instanciées et que la grille complète est cohérente.

III - Solution développée

III-1 Architecture choisie

L'architecture du projet repose sur les composants suivants :

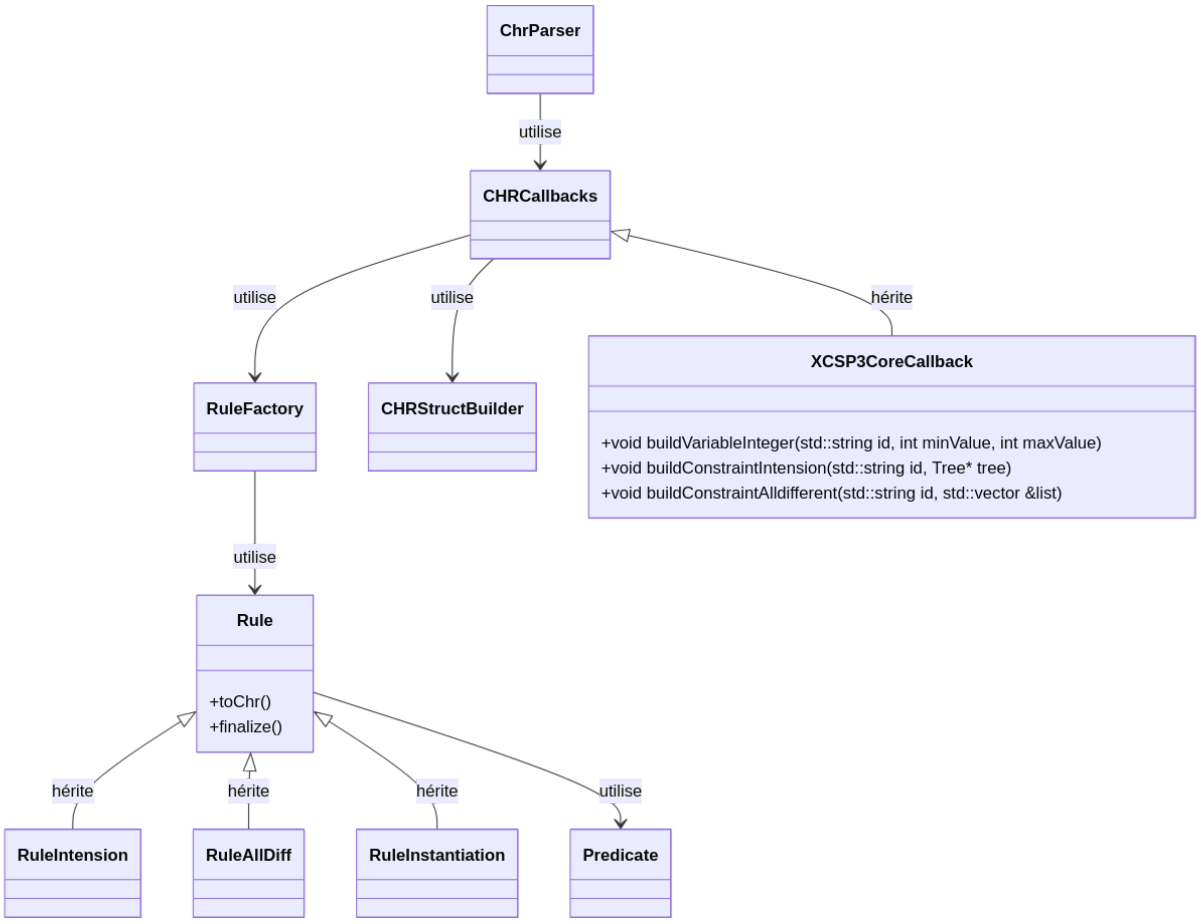
- CHRParser : point d'entrée du programme. Il configure le parseur, les callbacks, et lance le parsing du fichier XCSP3.

- CHRCallbacks : héritée de XCSP3CoreCallbacks, elle surcharge les méthodes de construction de contraintes pour produire des objets Predicate et Rule, qui sont ensuite transformés en code CHR.

- CHRStructBuilder : responsable de générer le squelette du programme C++ (includes, fonctions utilitaires, main, etc.) et d'intégrer les règles produites par les callbacks.

- Classes de représentation des règles et contraintes:
    - Predicate : représente une contrainte (avec ses types et ses éléments).
    - Rule, RuleIntension, RuleAllDifferent : Représentation des règles CHR construites à partir des contraintes XCSP3.
    - RuleFactory : fabrique de règles à partir de prédicats.

<u>Diagramme de classe (non exhaustif) :</u>



III-2. Le parser XCSP3

Le projet repose sur XCSP3CoreParser, la bibliothèque officielle de parsing du format XCSP3, développée par l'équipe du Laboratoire CRIL.

Fonctionnement général

- XCSP3CoreParser implémente un parser SAX .
- Il appelle automatiquement des méthodes comme buildVariableInteger, buildConstraintIntension, etc. lors de la lecture du fichier XML.
- Pour personnaliser le traitement, il nous faut étendre la classe XCSP3CoreCallbacks et de surcharger les méthodes appropriées.

Design pattern utilisé

La bibliothèque repose fortement sur le Pattern Visiteur. Cela se manifeste par :

- Le fait que XCSP3CoreParser parcourt l'arbre du document XML et délègue le traitement à des méthodes callbacks, une par type d'élément.
- L'utilisateur fournit sa propre classe dérivée de XCSP3CoreCallbacks pour gérer ces événements, en définissant (override) toutes fonctions de la forme build…()

Exemple :

```
void buildConstraintIntension(std::string id, Tree* tree) override;
```

Cela permet une séparation propre entre le parsing et l'application que l'on veut effectuer ensuite.

III-3 L’intégration d’un nouveau type de contrainte

Le projet vise à offrir un traducteur de contraintes depuis le format XCSP3 vers un système CHR++ permettant leur exécution. Cependant, le format XCSP3 contient un très grand nombre de contraintes prédéfinies, dont certaines très spécifiques ou rarement utilisées. Une prise en charge complète serait difficilement réaliste dans la durée du stage

C’est pourquoi l’architecture du projet a été pensée pour faciliter l’extensibilité : il doit être simple d’ajouter la prise en charge d’un nouveau type de contrainte sans avoir à modifier en profondeur les classes déjà existantes. Dans l’idéal, l’ajout repose uniquement sur l’implémentation d’un module dédié, tout en utilisant les outils de génération et d’intégration déjà présents.

Ce qui suit décrit les principales étapes nécessaires à l’ajout d’une nouvelle contrainte au système.

Étapes d’ajout d’une contrainte

1. Identification de la contrainte dans le parseur XCSP3

L’ajout d’une nouvelle contrainte commence par son identification dans un fichier .xml conforme au format XCSP3. Il faut notamment :

- Identifier le nom XML de la contrainte, tel qu’il apparaît dans le fichier <constraints>.

- Déterminer la fonction de rappel (callback) du parseur qui est appelée lorsqu’une contrainte de ce type est rencontrée. Il peut s’agir de fonctions de la forme BuildConstraintX(..)
- Surcharger cette méthode dans CHRCallbacks, et y invoquer la RuleFactory pour créer une règle adaptée.

2. Création d’une classe dérivée de Rule

Chaque contrainte est modélisée par une classe dédiée héritant de la classe Rule. Cette classe doit redéfinir deux méthodes essentielles :

toCHR()

Cette méthode est appelée lors de la traduction d’une règle. Elle sert à :

- Créer les prédicats nécessaires à la contrainte pour les variables spécifiques à l’instance (ex. : X[i], interval, make_list_var(...)).

- Générer les appels personnalisés via le CHRStructBuilder (ex : builder->addPredicateCallToMain(...)).

finalize()

Cette méthode est utile une seule fois par type de contrainte. Elle sert à :

- Ajouter les règles générales CHR++ liées à cette contrainte (classe RuleRaw).
- Ajouter les prédicats nécessaires à l'exécution (classe Predicate).
- Déclarer les appels à inclure dans le bloc CHR.

3. Ajout dans la RuleFactory

La RuleFactory permet d'instancier dynamiquement la classe correspondant à chaque type de contrainte.

Pour cela, il suffit d'ajouter une méthode dans la RuleFactory, comme :

```
std::shared_ptr<Rule> createAllDiffRule(std::string id, std::vector<XVariable*>& vars,
std::map<std::string, int>& name_id_map,CHRCallbacks* cb) const {return
std::make_shared<RuleAllDiff>(id, vars, name_id_map, cb);}
```

Cela permet de conserver un code propre dans CHRCallbacks, et de centraliser les règles disponibles au sein de la fabrique.

4. Utilisation de CHRStructBuilder

Lorsqu'une règle est générée, la classe CHRStructBuilder permet de formuler des instructions spécifiques qui seront insérées dans le main généré (section d'iinstanciation). Une fonction est principalement utilisée :

```
builder->addPredicateCallToMain("put_alldiff(make_list_var(...), make_list_int(...))");
```

Cela permet de gérer l'ajout au store via des appels par exemples pour put_alldiff, eq, neq, etc., directement dans le main().

Le fichier main() final, généré ou modifié, contient uniquement les éléments propres à l'instance :

- Initialisation des domaines (Dom[i] = interval(...))
- Assignation de valeurs fixes (set_eq(...))
- Appels put_alldiff(...) sur les lignes, colonnes, blocs.
- Appel à labelling(...) en l'ajoutant au store

À noter que l'on a implémenté la possibilité pour l'utilisateur de récupérer uniquement le bloc CHR en désactivant l'usage de la partie CHRStructBuilder.

5. Intégration dans le bloc CHR

Les règles génériques et prédicats nécessaires à une contrainte sont ajoutés dans le bloc CHR final généré, par les fonctions addRule(...) et add_constraint(...). Le résultat ressemble à ceci :

```
<CHR name="sudoku_1a">
<chr_constraint> CspVarIntDec(+int,+string,?int,-interval), ...
CspVarIntDec(Id,_,X,Dom) ==> CspVarInt(Id,X,Dom);;
put_alldiff(L, Ids) <=> alldiff_fun(*this, *L, *Ids);;
eq(X, Y) ==> X %= Y ;;
...
</CHR>
```

Ce bloc est commun à toutes les instances utilisant la même contrainte. Par conséquent, toutes les instances de Sudoku basées sur ce format XCSP3 partagent normalement un bloc CHR identique.

III-4 Différentes approches envisagées et choix

Dans le cadre du projet, plusieurs décisions d'architecture ont été explorées concernant la manière de générer le code CHR++ à partir d'une instance XCSP3. L'un des choix majeurs concerne le positionnement des éléments propres à l'instance, notamment les déclarations de variables et les affectations initiales.

Deux approches ont été mises en œuvre pour gérer ces éléments :

- Intégration complète dans le bloc <CHR> via une règle spéciale de type CspInit.
- Externalisation dans le main() du programme C++ généré.

Ces deux méthodes ont été comparées à l'aide d'un Sudoku de taille 4×4, un exemple suffisamment petit pour rester observable, mais révélateur des impacts de chaque stratégie.

Approche 1 – Traduction complète dans le bloc CHR

Dans cette première version, toute l'information liée à l'instanciation des variables, aux valeurs données initialement et aux groupes de contraintes (par exemple les alldiff) est directement intégrée dans une règle CspInit() du bloc CHR. Cette règle encode :

- toutes les déclarations de variables,
- toutes les affectations (eq(...)),
- tous les appels put_alldiff(...).

Extrait :

```
CspInit() <=>
  CspVarIntDec(0, string("x[0][0]"), X0, interval(1,4)),   ...,
  put_alldiff(make_list_var(X0, X1, X2, X3)), ...   ;;
```

Le principal facteur de ralentissement était alors la manière dont les contraintes alldiff et instanciation étaient traduites dans le bloc CHR.

En effet, chaque contrainte alldiff(...) était postée en 36 paires neq(Xi,Xj) (pour 9 variables), dans des règles longues et verbeuses du type :

```
CspVarIntDec(0,_,X0,_), CspVarInt(X0,Interval_0), ...,
CspVarIntDec(8,_,X8,_),CspVarInt(X8,Interval_8) ==> neq(X0,X1), neq(X0,X2), ..., neq(X7,X8).
```

Ce schéma est problématique et amène à une explosion combinatoire, chaque alldiff génère 36 règles de disjonction binaire, multipliées par le nombre de lignes, colonnes et blocs. De plus, on a un coût d'activation initial très élevé : le solveur CHR++ doit activer toutes ces contraintes dès le départ.

En parallèle, l'instanciation des cases initiales était aussi codée au sein du bloc CHR, avec des règles du type :

```
CspVarIntDec(4,_,X4,_), CspVarInt(X4,Interval_4) ==> eq(X4,1).
```

Multiplier ces règles dans le même bloc CHR ajoute encore de la charge inutilement.

Approche 2 – Séparation stricte : règles générales et données d'instance

Dans la version actuelle retenue, seule la définition des règles générales est présente dans le bloc CHR :

- Déclaration des prédicats (eq, neq, put_alldiff, etc.)
- Définition des règles générales de propagation

Les données propres à chaque instance (grille, indices fixés, put_alldiff sur les lignes, colonnes, blocs) sont entièrement générées dans la fonction main(). Cela inclut :

- Déclaration de variables X[i]
- Initialisation des domaines : Dom[i] = interval(...)
- Affectation de valeurs initiales : set_eq(Dom[i], val)
- Appels put_alldiff(...) explicites, sur les groupes spécifiques

	Approche 1	**Approche 2**
Temps d'exécution totale	17,4 s	4,8 s
Nombre de règles déclenchées	76 600	293

Bilan :
Dans la première stratégie, l'ensemble des déclarations de variables, des instanciations et des contraintes (telles que put_alldiff) sont insérées dans une unique règle déclenchée dès le démarrage : CspInit().

Cependant, cette méthode entraîne une explosion du nombre de règles déclenchées au moment de l'exécution. Chaque appel CspVarIntDec(...) génère une nouvelle occurence de contrainte, qui est ensuite testée pour toutes les règles dont le motif head inclut CspVarIntDec. Ce phénomène s'amplifie exponentiellement avec le nombre de variables distinctes, car les moteurs CHR++ tentent des appariements pour chaque combinaison possible de prédicats en présence.

Evolutions des prédicats arithmétiques :

Dans les premières versions de notre modélisation, les contraintes arithmétiques étaient exprimées sous la forme :

op(X, Y), où X et Y étaient des variables logiques.

Cette forme simple s'est révélée problématique dans certains cas : lorsque plusieurs variables partagent un même domaine, le moteur CHR++ peut ne plus distinguer correctement les contraintes une fois les variables instanciées.
Par exemple, les contraintes neq(X, Y) et neq(U, W) peuvent se retrouver toutes deux réduites à neq(2, 3) une fois les variables instanciées, ce qui fait perdre le lien entre les identifiants des variables d'origine. Cette perte d'information entraîne des incohérences dans le store, et peut perturber la propagation.

Pour résoudre cela, nous avons introduit une nouvelle signature des contraintes :

op(IdX, X, IdY, Y)

En associant à chaque variable logique son identifiant (IdX, IdY), on garantit la traçabilité des contraintes, même après instanciation. Ainsi, neq(IdX, 2, IdY, 3) et neq(IdU, 2, IdW, 3) restent distinguables, assurant une cohérence du store et une propagation fiable.

Contraintes déjà traitées :

Le système mis en place est déjà capable de prendre en charge la traduction de plusieurs types de contraintes issues du format XCSP3. Pour chacune de ces contraintes, une classe dérivée spécifique de la classe abstraite Rule a été implémentée, rendant le système modulaire, évolutif et maintenable.

1. Intension

Les contraintes d'intension sont parmi les plus générales et les plus expressives dans XCSP3. Elles permettent de définir des relations booléennes ou arithmétiques entre des variables via un arbre syntaxique (Tree*).

Dans notre système, elles sont prises en charge par la classe RuleIntension, qui se charge de :

- Traduire récursivement l'arbre d'expression en une suite de prédicats CHR++ (comme eq, lt, plus, etc.).

- Créer si nécessaire des variables temporaires pour les sous-expressions intermédiaires.

- Estimer les intervalles de domaines associés à ces temporaires.

- Ajouter dynamiquement les contraintes correspondantes dans le bloc CHR via des appels à add_constraint() et add_builtin_rule().

La méthode toCHR() construit la partie spécifique à l'instance (en analysant les variables présentes), tandis que la méthode finalize() ajoute les règles génériques nécessaires à l'exécution des opérateurs utilisés.

2. AllDifferent

Les contraintes alldifferent, très fréquentes en modélisation (comme dans le Sudoku), imposent que toutes les variables d'un ensemble prennent des valeurs distinctes.
Ces contraintes sont gérées par la classe RuleAllDiff, qui :

- Traduit les contraintes alldifferent(...) en un appel à un prédicat spécial put_alldiff(...).

- Génère des règles CHR++ génériques pour traiter la contrainte par appariement binaire via la contrainte neq(...).

- Ajoute dans le bloc CHR les règles nécessaires pour faire respecter la dissociation des valeurs, à l'aide d'une fonction externe alldiff_fun(...) définie en C++.

Le prédicat put_alldiff est instancié dans le main avec les listes concrètes de variables de l'instance. Le système est capable de traiter toutes les variantes classiques : par lignes, colonnes ou blocs dans une grille.

3. Instantiation

Les contraintes d'instanciation permettent d'affecter une valeur fixe à une ou plusieurs variables.
Elles sont traduites par la classe RuleInstantiation, qui :

- Génère une règle contenant une queue de type eq(...) pour exprimer l'affectation.

- Utilise les métadonnées des variables pour injecter les bonnes correspondances entre noms et identifiants.

- Établit les prédicats CspVarInt(...), CspVarIntDec(...) nécessaires pour la déclaration de la variable dans le CHR++.

Ici aussi, l'instanciation propre à l'instance est faite dans le main via des appels à set_eq(...).

IV - Perspectives et améliorations futures

1. Évolution de la structure et unification de l'appel solveur

Actuellement, l'appel à la labellisation se fait via space->labelling(0). Une première amélioration envisagée est l'introduction d'une fonction centralisée csp_solve(), destinée à encapsuler toute la logique de résolution. Cela rendra l'architecture plus modulaire et facilitera l'intégration future de stratégies de recherche plus complexes.

2. Traitement des contraintes en cours

Le traitement des contraintes de type ConstraintLinear, notamment dans le cas du problème du carré magique, nécessite encore des ajustements. Il s'agira notamment de corriger les cas où certaines règles ne se déclenchent pas comme attendu, en identifiant les éventuels conflits ou imprécisions dans le format généré.

3. Extension à de nouveaux types de contraintes

La suite du stage sera orientée vers l'ajout progressif de nouveaux types de contraintes issus du format XCSP3. Cela permettra d'étendre le champ d'application du traducteur à une gamme plus large de problèmes combinatoires.

4. Adaptation continue à Solvint

Le projet étant étroitement lié au développement de Solvint, il sera essentiel d'adapter en continu le générateur de code aux évolutions de ce solveur arithmétique. Cela inclut l'intégration de nouveaux prédicats, l'évolution de la syntaxe des contraintes en fonction des corrections et améliorations apportées à Solvint.

5. Ajout d'options de configuration

Afin de mieux contrôler le niveau de détail et le comportement du code généré, plusieurs options devraient être introduites dont :

- Des modes de génération : un mode minimal (centré uniquement sur la résolution) et un mode verbose (incluant commentaires, affichages intermédiaires, traçage).

- La sélection des règles : permettre à l'utilisateur de choisir les règles CHR qu'il souhaite générer ou ignorer, selon les types de contraintes présentes dans l'instance.

Annexes

Exemple d'une instance de Sudoku
[XCSP3]

```
<instance format="XCSP3" type="CSP">
 <variables>

  <array id="x" size="[9][9]"> 1..9 </array>            //déclaration variables et domaines
 </variables>
 <constraints>
  <group class="rows">                                  //lignes
   <allDifferent> %... </allDifferent>
   <args> x[0][] </args>
   <args> x[1][] </args>
   <args> x[2][] </args>
   <args> x[3][] </args>
   <args> x[4][] </args>
   <args> x[5][] </args>
   <args> x[6][] </args>
   <args> x[7][] </args>
   <args> x[8][] </args>
  </group>
  <group class="columns">                               //colonnes
   <allDifferent> %... </allDifferent>
   <args> x[][0] </args>
   <args> x[][1] </args>
   <args> x[][2] </args>
   <args> x[][3] </args>
   <args> x[][4] </args>
   <args> x[][5] </args>
   <args> x[][6] </args>
   <args> x[][7] </args>
   <args> x[][8] </args>
  </group>
  <group class="blocks">                                //grilles 3x3
   <allDifferent> %... </allDifferent>
   <args> x[0..2][0..2] </args>
   <args> x[0..2][3..5] </args>
   <args> x[0..2][6..8] </args>
   <args> x[3..5][0..2] </args>
   <args> x[3..5][3..5] </args>
   <args> x[3..5][6..8] </args>
   <args> x[6..8][0..2] </args>
   <args> x[6..8][3..5] </args>
   <args> x[6..8][6..8] </args>
  </group>
```

```
  <instantiation class="clues">
    <list> x[0][0] x[0][2] x[0][7..8] x[1][1..2] x[1][5] x[2][0] x[2][3] x[2][8] x[3][3..4] x[3][6] x[4][0]
x[4][2] x[4][6] x[4][8] x[5][2] x[5][4..5] x[6][0] x[6][5] x[6][8] x[7][3] x[7][6..7] x[8][0..1] x[8][6]
x[8][8] </list>
    <values> 2 6 4 9 3 7 9 1 7 6 5 8 9 7 5 8 4 9 6 2 9 4 1 3 4 9 4 1 2 8 </values>
  </instantiation>
 </constraints>
</instance>
```

[TRADUCTION CHR++]

[INCLUDE / FUNCTIONS]

```
#include <iostream>
#include <string>
#include <map>
#include <regex>
#include <chrpp.hh>
#include <bt_interval.hh>
#include <solvint.cpp>

using namespace chr;
using interval = chr::Interval<int, false>;
int global_id = 0;

// Fonction pour imprimer la grille en sortie
template <typename T>
void print_sudoku_grid(T& pb) {
        int grid[9][9] = {0};
        auto it = pb.chr_store_begin();
        std::regex re(R"(x\[(\d+)\]\[(\d+)\],\s*(\d+),)");
        while (!it.at_end()) {
        std::string fact = it.to_string();
        if (fact.find("CspVarIntDec") == std::string::npos) { ++it; continue; }
        std::smatch match;
        if (std::regex_search(fact, match, re)) {
        int i = std::stoi(match[1].str());
        int j = std::stoi(match[2].str());
        int val = std::stoi(match[3].str());
        if (i >= 0 && i < 9 && j >= 0 && j < 9) grid[i][j] = val;
        }
        ++it;
        }
        for (int i = 0; i < 9; ++i) {
        if (i % 3 == 0) std::cout << "-------------------------\n";
        for (int j = 0; j < 9; ++j) {
        if (j % 3 == 0) std::cout << "| ";
```

```
        std::cout << (grid[i][j] ? std::to_string(grid[i][j]) : ".") << ' ';
        }
        std::cout << "|\n";
        }
        std::cout << "-------------------------\n";
}

// Fonction utilitaire : définir l'égalité d'un domaine à une constante
inline chr::ES_CHR set_eq(chr::Logical_var_mutable<interval> &x, int v) {
        bool modified = false;
        return x.update_mutable(modified, [&modified, v](auto& intvl){
        modified = intvl.eq(v);
        });
}

// Fonction qui impose une contrainte allDifferent (traduction du XCSP3)
template<class T>
bool alldiff_fun(T& space, const std::list<chr::Logical_var<int>>& vars, const std::list<int>&
ids) {
        auto it1 = vars.begin();
        auto id1 = ids.begin();
        for (; it1 != vars.end(); ++it1, ++id1) {
        auto it2 = it1;
        auto id2 = id1;
        ++it2; ++id2;
        for (; it2 != vars.end(); ++it2, ++id2) {
        CHECK_ES(space.neq(chr::Logical_var_ground<int>(*id1), *it1,
chr::Logical_var_ground<int>(*id2), *it2));
        }
        }
        return true;
}
```

[Section CHR]

```
<CHR name="sudoku">
<chr_constraint>
        CspVarIntDec(+int,+string,?int,-interval),
        CspVarInt(+int,?int,-interval),
        neq(+int,?int,+int,?int),
        put_alldiff(?list_var,?list_int),
        eq(?int,?int),
        labelling(+int)

CspVarInt(_,_, Dom) ==> (*Dom).empty() | failure();;
CspVarInt(_,X, XDom) ==> (*XDom).singleton() | X %= (*XDom).val();;
CspVarInt(_,Val, Dom) ==> Val.ground() | set_eq(Dom, *Val);;
CspVarIntDec(Id,_,X,Dom) ==> CspVarInt(Id,X,Dom);;
```

```
neq_var_var @ CspVarInt(Id1, X, XDom), CspVarInt(Id2, Y, YDom), neq(Id1, X, Id2, Y) ==>
Solvint::ne(XDom, YDom);;
neq_cste_var @ CspVarInt(Id, Y, YDom), neq(_,X,Id,Y) ==> X.ground() | Solvint::ne(YDom, *X) ;;
neq_var_cste @ CspVarInt(Id, X, XDom), neq(Id,X,_,Y) ==> Y.ground() | Solvint::ne(XDom, *Y) ;;

put_alldiff(L, Ids) <=> (*L).empty() | success;;
put_alldiff(L, Ids) <=> alldiff_fun(*this, *L, *Ids);;

eq_cste_cste @ eq(X, Y) ==> X %= Y ;;
eq_var_var @ CspVarInt(_,X, XDom), CspVarInt(_,Y, YDom), eq(X, Y) ==> XDom %= YDom ;;

CspVarIntDec(Id, _, _, Dom) \ labelling(Id) <=> exists_it(i, *Dom, ( set_eq(Dom,i) , labelling(Id+1)));;
labelling(Id) <=> print_sudoku_grid(*this);;
</CHR>
```

[Main]

```
                                int main(int argc, const char *argv[]) {
        auto space = sudoku_2a::create();
//Création des variables logiques et domaines associés à chaque variable à représenter.
        chr::Logical_var<int> X[81];
        chr::Logical_var_mutable<interval> Dom[81];

        for (int i = 0; i < 81; ++i) {
        Dom[i] = interval(1, 9);
        space->CspVarIntDec(i, "x[" + std::to_string(i / 9) + "][" + std::to_string(i % 9) + "]",
X[i], Dom[i]);}
//Contrainte d'instanciation correspondant aux valeurs de la grille initiale du Sudoku
        set_eq(Dom[0], 2);  set_eq(Dom[2], 6); set_eq(Dom[7], 4);
        set_eq(Dom[8], 9); set_eq(Dom[10], 3); set_eq(Dom[11], 7);
        set_eq(Dom[14], 9); set_eq(Dom[18], 1); set_eq(Dom[21], 7);
        set_eq(Dom[26], 6); set_eq(Dom[30], 5); set_eq(Dom[31], 8);
        set_eq(Dom[33], 9); set_eq(Dom[36], 7); set_eq(Dom[38], 5);
        set_eq(Dom[42], 8); set_eq(Dom[44], 4); set_eq(Dom[47], 9);
        set_eq(Dom[49], 6); set_eq(Dom[50], 2); set_eq(Dom[54], 9);
        set_eq(Dom[59], 4); set_eq(Dom[62], 1); set_eq(Dom[66], 3);
        set_eq(Dom[69], 4); set_eq(Dom[70], 9); set_eq(Dom[72], 4);
        set_eq(Dom[73], 1); set_eq(Dom[78], 2); set_eq(Dom[80], 8);

//Déclaration des contraintes AllDifferent
//lignes
CHR_RUN(        space->put_alldiff(make_list_var(X[0], X[1], X[2], X[3], X[4], X[5], X[6], X[7], X[8]),
make_list_int(0, 1, 2, 3, 4, 5, 6, 7, 8));
        space->put_alldiff(make_list_var(X[9], X[10], X[11], X[12], X[13], X[14], X[15], X[16], X[17]),
make_list_int(9, 10, 11, 12, 13, 14, 15, 16, 17));
        space->put_alldiff(make_list_var(X[18], X[19], X[20], X[21], X[22], X[23], X[24], X[25], X[26]),
make_list_int(18, 19, 20, 21, 22, 23, 24, 25, 26));
        space->put_alldiff(make_list_var(X[27], X[28], X[29], X[30], X[31], X[32], X[33], X[34], X[35]),
make_list_int(27, 28, 29, 30, 31, 32, 33, 34, 35));
```

```
        space->put_alldiff(make_list_var(X[36], X[37], X[38], X[39], X[40], X[41], X[42], X[43], X[44]),
make_list_int(36, 37, 38, 39, 40, 41, 42, 43, 44));
        space->put_alldiff(make_list_var(X[45], X[46], X[47], X[48], X[49], X[50], X[51], X[52], X[53]),
make_list_int(45, 46, 47, 48, 49, 50, 51, 52, 53));
        space->put_alldiff(make_list_var(X[54], X[55], X[56], X[57], X[58], X[59], X[60], X[61], X[62]),
make_list_int(54, 55, 56, 57, 58, 59, 60, 61, 62));
        space->put_alldiff(make_list_var(X[63], X[64], X[65], X[66], X[67], X[68], X[69], X[70], X[71]),
make_list_int(63, 64, 65, 66, 67, 68, 69, 70, 71));
        space->put_alldiff(make_list_var(X[72], X[73], X[74], X[75], X[76], X[77], X[78], X[79], X[80]),
make_list_int(72, 73, 74, 75, 76, 77, 78, 79, 80));

//colonnes
        space->put_alldiff(make_list_var(X[0], X[9], X[18], X[27], X[36], X[45], X[54], X[63], X[72]),
make_list_int(0, 9, 18, 27, 36, 45, 54, 63, 72));
        space->put_alldiff(make_list_var(X[1], X[10], X[19], X[28], X[37], X[46], X[55], X[64], X[73]),
make_list_int(1, 10, 19, 28, 37, 46, 55, 64, 73));
        space->put_alldiff(make_list_var(X[2], X[11], X[20], X[29], X[38], X[47], X[56], X[65], X[74]),
make_list_int(2, 11, 20, 29, 38, 47, 56, 65, 74));
        space->put_alldiff(make_list_var(X[3], X[12], X[21], X[30], X[39], X[48], X[57], X[66], X[75]),
make_list_int(3, 12, 21, 30, 39, 48, 57, 66, 75));
        space->put_alldiff(make_list_var(X[4], X[13], X[22], X[31], X[40], X[49], X[58], X[67], X[76]),
make_list_int(4, 13, 22, 31, 40, 49, 58, 67, 76));
        space->put_alldiff(make_list_var(X[5], X[14], X[23], X[32], X[41], X[50], X[59], X[68], X[77]),
make_list_int(5, 14, 23, 32, 41, 50, 59, 68, 77));
        space->put_alldiff(make_list_var(X[6], X[15], X[24], X[33], X[42], X[51], X[60], X[69], X[78]),
make_list_int(6, 15, 24, 33, 42, 51, 60, 69, 78));
        space->put_alldiff(make_list_var(X[7], X[16], X[25], X[34], X[43], X[52], X[61], X[70], X[79]),
make_list_int(7, 16, 25, 34, 43, 52, 61, 70, 79));
        space->put_alldiff(make_list_var(X[8], X[17], X[26], X[35], X[44], X[53], X[62], X[71], X[80]),
make_list_int(8, 17, 26, 35, 44, 53, 62, 71, 80));


//grilles 3x3
        space->put_alldiff(make_list_var(X[0], X[1], X[2], X[9], X[10], X[11], X[18], X[19], X[20]), make_list_int(0,
1, 2, 9, 10, 11, 18, 19, 20));
        space->put_alldiff(make_list_var(X[3], X[4], X[5], X[12], X[13], X[14], X[21], X[22], X[23]),
make_list_int(3, 4, 5, 12, 13, 14, 21, 22, 23));
        space->put_alldiff(make_list_var(X[6], X[7], X[8], X[15], X[16], X[17], X[24], X[25], X[26]),
make_list_int(6, 7, 8, 15, 16, 17, 24, 25, 26));
        space->put_alldiff(make_list_var(X[27], X[28], X[29], X[36], X[37], X[38], X[45], X[46], X[47]),
make_list_int(27, 28, 29, 36, 37, 38, 45, 46, 47));
        space->put_alldiff(make_list_var(X[30], X[31], X[32], X[39], X[40], X[41], X[48], X[49], X[50]),
make_list_int(30, 31, 32, 39, 40, 41, 48, 49, 50));
        space->put_alldiff(make_list_var(X[33], X[34], X[35], X[42], X[43], X[44], X[51], X[52], X[53]),
make_list_int(33, 34, 35, 42, 43, 44, 51, 52, 53));
        space->put_alldiff(make_list_var(X[54], X[55], X[56], X[63], X[64], X[65], X[72], X[73], X[74]),
make_list_int(54, 55, 56, 63, 64, 65, 72, 73, 74));
        space->put_alldiff(make_list_var(X[57], X[58], X[59], X[66], X[67], X[68], X[75], X[76], X[77]),
make_list_int(57, 58, 59, 66, 67, 68, 75, 76, 77));
        space->put_alldiff(make_list_var(X[60], X[61], X[62], X[69], X[70], X[71], X[78], X[79], X[80]),
make_list_int(60, 61, 62, 69, 70, 71, 78, 79, 80)););
//Ajout de labelling(0) pour lancer la résolution
CHR_RUN(space->labelling(0););
//Affichage des statistiques
chr::Statistics::print(std::cout);
return 0;}
```

Références

- [1] Vincent Barichard. CHR++: An efficient CHR system in C++ with don't know non-determinism. Expert Systems with Applications, 238:121810, 2024.
- [2] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. Journal of Logic Programming, 19/20:503–581, 1994.
- [3] Thom Frühwirth, Constraint Handling Rules (Cambridge Univ. Press, 2009)
- [4] https://vynce.gitlab.io/chrpp/modeling.html#modeling_constraint_declarations
- [5] https://xcsp.org/