Modélisation déclarative du raisonnement OWL 2 RL avec Constraint Handling Rules

Arina Abed, Vincent Barichard, David Genest, Éric Monfroy

Univ Angers, LERIA, SFR MATHSTIC, 49000 Angers, France

arina.abed@etud.univ-angers.fr vincent.barichard@univ-angers.fr
david.genest@univ-angers.fr eric.monfroy@univ-angers.fr

Résumé

*Cet article présente **owl2chr**, un moteur de raisonnement déclaratif pour OWL 2 RL basé sur CHR++. Les axiomes OWL sont traduits en contraintes, et les inférences par un ensemble de règles CHR appliquées jusqu'à saturation. Cette approche offre une transparence et une modularité supérieures à celles des raisonneurs classiques. Les tests avec OWL2Bench confirment la validité de l'approche, avec des performances du même ordre de grandeur. Une fois la base saturée, les requêtes deviennent quasi instantanées, faisant d'**owl2chr** une alternative extensible et explicable.*

Mots-clés

Ontologies, OWL, programmation par règles, Constraint Handling Rules.

Abstract

*The article presents **owl2chr**, a declarative reasoning engine for OWL 2 RL based on CHR++. The OWL axioms are translated into constraints, and the inferences are produced by a set of CHR rules applied until saturation. This approach offers greater transparency and modularity than classical reasoners. Tests with OWL2Bench confirm the validity of the approach, with performance in the same order of magnitude. Once the knowledge base is saturated, queries become almost instantaneous, making **owl2chr** an extensible and explainable alternative.*

Keywords

Ontologies, OWL, rule-based programming, Constraint Handling Rules.

1 Introduction

Le Web sémantique [6] repose sur la formalisation des connaissances à l'aide d'ontologies. Pour exploiter pleinement ces structures, l'usage d'un raisonneur est indispensable afin de déduire des informations implicites à partir des faits explicitement déclarés. Cependant, les outils de référence tels que *Pellet* [22], *HermiT* [17] ou *Konklude* [24] fonctionnent souvent comme des « boîtes noires ». Cette opacité découle d'algorithmes de recherche complexes (comme les méthodes de Tableaux [3]) et d'architectures internes sophistiquées. En privilégiant l'efficacité du calcul, ces mécanismes masquent le lien logique entre les faits initiaux et la conclusion produite, le rendant difficilement exploitable par l'utilisateur.

Cette absence de transparence constitue aujourd'hui un frein majeur dans les contextes où l'explicabilité est essentielle, notamment dans les domaines médical, juridique ou industriel. Dans ces environnements critiques, produire une réponse correcte ne suffit pas : il est indispensable de pouvoir retracer précisément le raisonnement qui y conduit. De plus, la rigidité des moteurs existants limite leur adaptabilité. L'intégration de règles métier spécifiques ou l'ajustement de certains mécanismes d'inférence nécessitent généralement d'intervenir sur des architectures fermées qui ne sont pas conçues pour la modularité ni pour l'extension.

Dans ce travail, nous défendons une approche alternative fondée sur la transparence, la traçabilité et la modularité du raisonnement. Nous proposons d'exploiter le langage déclaratif **CHR** (*Constraint Handling Rules*) [15], un formalisme à base de règles de réécriture particulièrement adapté à la formalisation explicite des mécanismes d'inférence basés sur des règles. CHR repose sur un *store* de contraintes représentant l'état courant des connaissances, et sur un ensemble fini de règles gardées qui transforment progressivement cet état jusqu'à l'atteinte d'un point fixe. Ce mode de calcul déclaratif rend chaque étape du raisonnement observable et contrôlable.

Nous appliquons cette approche au profil **OWL 2 RL** [19], un sous-ensemble standardisé d'OWL 2 [25] conçu pour un raisonnement efficace par règles. Chaque axiome de l'ontologie est traduit en contraintes CHR, tandis que les mécanismes d'inférence du profil sont formalisés sous forme de règles stables et lisibles. Le système résultant, **owl2chr**, implémenté en **CHR++** [4, 5], un système CHR performant et extensible basé sur C++, applique ces règles de manière récursive jusqu'à atteindre un point fixe, où plus aucune transformation n'est possible. Une fois la saturation complète de la base de faits, l'ensemble des connaissances implicites est explicitement disponible dans le *store*, permettant un requêtage quasi instantané car, pour la majorité des requêtes, il ne nécessite plus aucun calcul d'inférence supplémentaire : il s'agit d'une simple lecture des contraintes présentes dans le *store*.

Au-delà de l'efficacité obtenue dans les scénarios d'interrogation intensive, l'apport principal de notre approche réside dans son explicabilité native : chaque déduction peut être retracée, analysée et, si nécessaire, adaptée. Contrairement aux raisonneurs industriels optimisés comme des systèmes clos, **owl2chr** offre un cadre ouvert où l'utilisateur peut enrichir, modifier ou spécialiser les règles d'inférence selon les besoins d'un domaine particulier. L'intégration dans CHR++ permet en outre de coupler directement les règles à des actions C++, ouvrant la voie à des extensions applicatives fines sans altérer le cœur du moteur.

Cet article est organisé de la manière suivante. Nous commencerons par décrire, en section 2, les concepts de CHR et du Web Sémantique. Dans la section 3, nous présenterons la traduction des axiomes en contraintes ainsi que la modélisation des inférences OWL 2 RL sous forme de règles CHR. La section 4 expose l'architecture de **owl2chr** ainsi que les résultats des tests de conformité et de performance. Enfin, une discussion en section 5 précède la conclusion et les perspectives en section 6.

2 Cadre théorique

2.1 Constraint Handling Rules (CHR) et CHR++

CHR (Constraint Handling Rules) [15, 9, 10, 11, 12, 13, 14] est un langage de programmation déclaratif de haut niveau, orienté vers la réécriture de contraintes. Initialement conçu comme une extension pour Prolog, il est aujourd'hui implémenté au-dessus de divers langages hôtes tels que Java, C++ ou Haskell. Il convient de distinguer le *langage* CHR, qui définit la syntaxe et la sémantique formelle, d'un *système* CHR qui constitue le moteur d'exécution intégré au langage hôte.

Un programme CHR manipule un *store* de contraintes, qui est un multi-ensemble de prédicats définis par l'utilisateur. Contrairement à une base de données classique, l'ajout d'une contrainte dans le *store* déclenche un mécanisme actif de recherche de règles applicables. L'exécution procède par chaînage avant (*forward chaining*) de manière incrémentale. Le processus suit un principe de saturation : une règle est déclenchée dès que ses têtes correspondent à des contraintes du *store* et que sa garde est vérifiée. Une fois appliquée, le système garantit qu'une règle ne peut pas se redéclencher sur la même combinaison de contraintes. Cette gestion de la redondance favorise la convergence du système vers un point fixe (ou état de stabilité) où plus aucune règle ne peut être déclenchée (ou lorsqu'une contradiction est détectée).

Variable logique. À l'instar des implémentations classiques de CHR (en particulier en Prolog), CHR++ intègre les variables logiques. Contrairement aux variables classiques, une variable logique dans CHR++ appartient à une classe d'équivalence. Initialement, chaque variable X réside dans sa propre classe X.

L'opérateur d'unification $\doteq$ permet de fusionner ces classes d'équivalence. Par exemple, l'unification de deux variables X et Y les regroupe dans une classe unique $\{X, Y\}$, les rendant sémantiquement indiscernables pour le moteur de règles. Ce processus peut se poursuivre par étapes successives : une unification ultérieure avec la constante 2 produira la classe $\{X, Y, 2\}$.

La classe est dite instanciée (ou *ground*) dès qu'elle contient une constante. Une fois cette valeur fixée, la classe d'équivalence ne peut plus être unifiée avec une constante différente. Cette contrainte garantit l'intégrité logique du système en empêchant toute contradiction au sein d'une même classe d'équivalence.

Structure et fonctionnement des règles. Une règle CHR est composée de plusieurs éléments et obéit à une forme syntaxique générale. Elle repose principalement sur une tête, constituée d'un multiensemble de contraintes, et un corps, formé d'une séquence de contraintes. La règle peut également comporter une garde, constituée d'expressions du langage hôte, qui permet de vérifier certaines préconditions (par exemple des tests arithmétiques ou logiques) avant d'autoriser son application. Enfin, la règle peut être associée à un nom optionnel, correspondant à une étiquette qui permet de l'identifier ; cette nomination facilite sa documentation ainsi que son repérage dans les traces d'exécution.

Le mode de sélection et d'application des règles dépend de la sémantique adoptée. En particulier, dans la sémantique raffinée, les règles sont considérées selon leur ordre d'apparition dans le programme (de haut en bas). Une règle est déclenchée dès que ses têtes correspondent (matching) à des contraintes présentes dans le *store* et que sa garde est satisfaite. Une propriété fondamentale de CHR est le ***committed choice*** : contrairement au retour en arrière (*backtracking*) classique de Prolog, une fois qu'une règle est sélectionnée et déclenchée, elle est appliquée définitivement et le système ne revient pas sur ce choix.

Types de règles. Il existe trois formes de règles régissant la modification du *store* :

- **Propagation** ($Nom \ @ \ H_k \implies G \mid B$) : cette règle ajoute les contraintes B au *store* si les contraintes de tête H_k y sont présentes et que la garde G est vérifiée. Les contraintes H_k sont conservées. Cette règle permet de faire de l'inférence.
- **Simplification** ($Nom \ @ \ H_r \iff G \mid B$) : elle remplace les contraintes H_r par B dans le *store* si les contraintes de tête H_r y sont présentes et que la garde G est vérifiée. Cette transformation permet de simplifier l'état du *store*.
- **Simpagation** ($Nom \ @ \ H_k \setminus H_r \iff G \mid B$) : généralise les deux précédentes, elle préserve H_k mais supprime H_r du *store*, tout en ajoutant B, si la garde G est vérifiée et que les contraintes de tête H_k et H_r sont présentes.

Exemple illustratif. Pour illustrer ces mécanismes, nous définissons une contrainte d'ordre total "inférieur ou égal" (leq) et nous allons en calculer la fermeture transitive. Considérons le programme CHR suivant composé de quatre

règles :

$$\begin{aligned} \textit{idempotence} \ @\ \text{leq}(X,Y)\backslash\text{leq}(X,Y) &\iff \top \\ \textit{réflexivité} \ @\ \text{leq}(X,X) &\iff \top \\ \textit{antisymétrie} \ @\ \text{leq}(X,Y),\ \text{leq}(Y,X) &\iff X \doteq Y \\ \textit{transitivité} \ @\ \text{leq}(X,Y),\ \text{leq}(Y,Z) &\implies \text{leq}(X,Z) \end{aligned}$$

Ces règles permettent de calculer la fermeture transitive de l'opérateur inférieur ou égal. Elles sont toutes dépourvues de garde, la présence de la tête dans le *store* suffit donc à déclencher leur application :

1. *Idempotence* : si le *store* contient deux occurrences de la même contrainte leq(X, Y), l'une d'elles est supprimée. Cela évite la redondance.
2. *Réflexivité* : s'il existe une contrainte leq(X, X) dans le *store* elle est simplifiée en $\top$ (elle est retirée du *store* de contraintes).
3. *Antisymétrie* : si le *store* contient leq(X, Y) et leq(Y, X), ces deux contraintes sont consommées et remplacées par l'unification des deux variables $X \doteq Y$.
4. *Transitivité* : la présence de leq(X, Y) et leq(Y, Z) entraîne l'ajout d'une nouvelle contrainte leq(X, Z) sans supprimer les originales.

Déroulement de l'exécution. Voici le déroulement de l'exécution suite à l'insertion des contraintes leq(A, B), leq(C, A) et leq(B, C) dans le *store* de contraintes : l'exécution débute avec la contrainte leq(A, B), qui est activée et ajoutée au *store*. À cet instant, aucune règle ne se déclenche ; elle est donc inactivée et conservée. La contrainte leq(C, A) est alors ajoutée et activée. La règle de *transitivité* est déclenchée avec pour têtes leq(C, A) et leq(A, B), permettant l'ajout d'une nouvelle contrainte leq(C, B). Celle-ci devient active, puis est stockée dans le *store* car aucune règle ne s'applique.
Enfin, la contrainte leq(B, C) est ajoutée. Son activation déclenche la règle d'antisymétrie avec leq(C, B), qui provoque la suppression de leq(B, C) et leq(C, B) du *store* ainsi que l'exécution de l'unification logique $B \doteq C$. Cette unification modifie le *store* : {leq(A, B), leq$(B, A), B \doteq C$} et réactive les contraintes leq(A, B) et leq(C, A), car l'unification modifie les classes d'équivalence des variables B et C. La sémantique ne définissant pas d'ordre de réactivation prioritaire, supposons que $leq(A, B)$ soit réactivée en premier. Ce choix déclenche à nouveau la règle d'antisymétrie, ce qui provoque la suppression des contraintes $leq(A, B)$ et $leq(B, A)$ ainsi que l'ajout de l'unification $A \doteq B$. Le *store* devient alors : $\{B \doteq C, A \doteq B\}$. Comme plus aucune contrainte n'est susceptible d'être activée, l'exécution du programme prend fin.

Le système CHR++ [4, 5] est une implémentation moderne et performante du langage CHR pour le langage C++, développée au LERIA (Université d'Angers). Son objectif principal est de marier l'expressivité déclarative du langage CHR avec la puissance de calcul des langages compilés.
Sur le plan pratique, l'intégration se fait via une API légère. Le développeur définit ses contraintes et ses règles dans des blocs spécifiques balisés par `<CHR>`. Un préprocesseur analyse ces blocs pour générer les structures C++ nécessaires, permettant au programme hôte de poster des contraintes et de récupérer l'état du *store* de manière transparente. Enfin, CHR++ se distingue par son support du non-déterminisme (*don't know non-determinism*), offrant ainsi la flexibilité nécessaire pour résoudre des problèmes combinatoires complexes ou explorer plusieurs alternatives de résolution.

2.2 Ontologies OWL 2 et le profil RL

Ontologie. Une ontologie est une spécification formelle d'un domaine de connaissance. Elle s'articule autour de quatre piliers : les **classes** (concepts comme Book), les **propriétés** (relations comme hasStudent), les **individus** (instances concrètes) et les **axiomes** (comme SubClassOf). Ces derniers fixent la sémantique du domaine en définissant des contraintes, telles qu'une relation de subsomption entre classes ou des contraintes de cardinalité, qui organisent les relations entre les entités.

OWL 2 (Web Ontology Language) [25] est le standard du W3C pour la représentation des ontologies du Web sémantique. Il repose sur une sémantique formelle basée sur les logiques de description. Il offre une grande expressivité en permettant la définition de classes complexes ainsi que la spécification précise des propriétés et de leur sémantique :

- **Restrictions de classes** : SubClassOf permet, par exemple, de définir une relation de subsomption entre les classes Book et Publication, voir l'exemple 1. Des constructeurs permettent de combiner des classes pour créer des classes complexes, entre autres, le constructeur ObjectAllValuesFrom permet d'imposer que toute valeur prise par une propriété donnée appartienne à une classe cible (exemple : un WomanCollege ne doit avoir que des étudiants qui ne soient pas des Man, comme défini par le deuxième axiome de l'exemple 1).
- **Caractéristiques de propriétés** : elles permettent de définir qu'une relation est transitive, symétrique ou l'inverse d'une autre relation (exemple : parentOf est l'inverse de childOf).

Exemple OWL 2. Une ontologie OWL 2 peut être exprimée sous un des formats classiques du web sémantique comme RDF/XML ou Turtle. OWL 2 propose aussi une syntaxe fonctionnelle [18] qui est celle que nous utilisons ici. Elle met en avant la structure fonctionnelle des axiomes et s'avère très proche de la spécification abstraite d'OWL, comme l'illustrent les déclarations de la figure 1

Profils d'OWL 2. Les profils d'OWL 2 [19] constituent des sous-ensembles du langage complet, conçus pour offrir un compromis entre expressivité logique et efficacité algorithmique. Parmi les trois profils standards (EL, QL et RL), nous nous intéressons plus particulièrement à OWL 2 RL (Rule Language), qui est au cœur de notre travail. Il est conçu pour les applications nécessitant un raisonnement efficace. Sa structure syntaxique est restreinte de manière à ce que les axiomes puissent être traduits directement en règles

```
SubClassOf( :Book :Publication)
SubClassOf ( :WomanCollege
   ObjectIntersectionOf (
      :College
      ObjectAllValuesFrom(
         :hasStudent ObjectComplementOf( :Man )
      )
   )
)
```

FIGURE 1 – Exemple d'axiomes OWL 2 en syntaxe fonctionnelle

d'inférence logiques. Cette caractéristique rend OWL 2 RL particulièrement adapté à une exécution par chaînage avant, garantissant une complexité polynomiale pour les tâches de raisonnement.

Raisonneur. Un raisonneur est un composant logiciel chargé d'inférer des connaissances implicites à partir de faits explicitement déclarés dans une base de connaissances. Il assure les fonctionnalités fondamentales du raisonnement : la **vérification de la cohérence** (s'assurer qu'aucun axiome ne mène à une contradiction logique), la **classification** (calculer la hiérarchie complète subsomption entre toutes les classes) et la **réalisation** (déterminer, pour chaque individu, les classes les plus spécifiques auxquelles il appartient). Parmi les plus connus, on trouve *Pellet* [22], *HermiT* [17], *Openllet* [16], *JFact* [23] ou Konklude [24]. Par exemple, à partir des axiomes :

SubClassOf(:Book :Publication)
ClassAssertion(:Book :P10)

Le raisonneur déduit l'assertion suivante :

ClassAssertion(:Publication :P10)

3 Modélisation des inférences OWL 2 RL en CHR

Le principe de la représentation d'une ontologie OWL 2 RL en CHR est le suivant : pour chaque axiome OWL 2 RL, nous avons défini en CHR une contrainte équivalente, par exemple pour SubClassOf nous définissons la contrainte owlSubClassOf. Ainsi qu'un ensemble de règles d'inférence qui en incarnent la sémantique reproduisant le comportement attendu d'un raisonneur. L'analyseur syntaxique alimente le *store* de contraintes en transformant les axiomes lus dans l'ontologie en contraintes correspondantes. Lorsqu'une contrainte est ajoutée au *store*, elle déclenche immédiatement l'application de règles CHR. Ces règles peuvent inférer de nouvelles contraintes qui correspondent à de nouveaux faits, ces derniers sont immédiatement intégrés au *store*, déclenchant potentiellement d'autres règles, et ainsi de suite, jusqu'à ce que plus aucun déclenchement ne soit possible pour cette contrainte. La contrainte est alors considérée comme inactive. Ce processus se répète pour chaque axiome de l'ontologie, jusqu'à ce qu'un échec soit détecté, signalant une incohérence, ou que tous les axiomes de l'ontologie aient été traités. Les faits déduits sont alors explicitement ajoutés au *store*, rendant les connaissances persistantes et directement interrogeables.

3.1 Prise en charge des axiomes

Le profil OWL 2 RL (Rule Language) couvre une large partie des constructeurs d'OWL 2. Le travail que nous avons mené prend en charge l'intégralité d'OWL 2 RL, mais dans la suite de cet article, pour des raisons de place, nous ne décrivons qu'une partie de ce profil. Toutefois, l'implémentation complète des règles du profil est consultable sur Github [1]. Voici quatre contraintes représentatives de notre modélisation d'OWL 2 RL en CHR :

- owlSubClassOf(A, B) : cette contrainte est générée lors de la lecture d'un axiome SubClassOf(A B) dans l'ontologie. Elle définit une relation de subsomption entre les classes A et B.
- owlClassAssertion(x, C, n) : lors de la lecture d'un axiome ClassAssertion(C x) dans l'ontologie, une contrainte owlClassAssertion(x, C, true) est ajoutée au *store*. L'argument booléen n permet de qualifier la nature de cette assertion : la valeur true, utilisée par défaut, indique que l'assertion est définitive (fait avéré), tandis que la valeur false signale une assertion provisoire. Cette dernière est employée spécifiquement dans le cadre du complément (voir section 3.2.3) pour marquer des faits susceptibles d'évoluer au cours du processus d'inférence.
- owlDisjointClasses(A, B) : est ajoutée au *store* dès lors qu'un axiome DisjointClasses(A B) est rencontré, spécifie que les classes A et B ne peuvent pas avoir d'instances communes.
- owlComplementOf(C, NotC) : correspondant à l'axiome ComplementOf, spécifie que la classe NotC est le complément de la classe C.

3.2 Représentation d'inférences OWL sous forme de règles CHR

La sémantique de chaque axiome OWL 2 RL est capturée par la définition d'une ou plusieurs règles CHR. L'utilisation de règles de propagation permet d'expliciter des connaissances implicites en enrichissant le *store* de nouveaux faits. Dans ce qui suit, nous présentons certaines des règles associées à la contrainte owlSubClassOf, puis nous présentons celles relatives à owlDisjointClasses pour finir avec owlComplementOf .

3.2.1 SubClassOf

Cet axiome se décline en plusieurs règles CHR qui traduisent sa sémantique. Ces règles implémentent trois propriétés de cet axiome.

Héritage : si A est une sous-classe de B, et qu'un individu X est une instance de A, alors il est également une instance de B (règle *sub-herit*, Figure 2) .

sub-herit @ owlSubClassOf(A, B), owlClassAssertion$(X, A, \text{true}) \Longrightarrow$ owlClassAssertion(X, B, true)

sub-trans @ owlSubClassOf(A, B), owlSubClassOf$(B, C) \Longrightarrow$ owlSubClassOf(A, C)

sub-idem @ owlSubClassOf$(A, B) \backslash$owlSubClassOf$(A, B) \iff \top$

disj-verif @ owlDisjointClasses(A, B), owlClassAssertion(X, A, true), owlClassAssertion$(X, B, \text{true}) \Longrightarrow \bot$

comp-init @ owlComplementOf$(A, NOTA)$, owlNamedIndividual$(X) \Longrightarrow$ owlClassAssertion$(X, NOTA, \text{false})$

comp-valid @ owlClassAssertion$(X, NOTA, \text{true}) \backslash$owlClassAssertion$(X, NOTA, \text{false}) \iff \top$

comp-elim @ owlComplementOf$(A, NOTA)$, owlClassAssertion$(X, A, \text{true}) \backslash$owlClassAssertion$(X, NOTA, \text{false}) \iff \top$

comp-verif @ owlComplementOf$(A, NOTA)$, owlClassAssertion(X, A, true), owlClassAssertion$(X, NOTA, \text{true}) \iff \bot$

FIGURE 2 – Règles CHR

Par exemple si le *store* contient :

owlSubClassOf(Book,Publication)
owlClassAssertion(P10,Book,true)

La règle sub-herit déduira, et ajoutera au *store* la contrainte :

owlClassAssertion(P10,Publication,true)

Transitivité de la relation de subsomption : si A est une sous-classe de B et B est une sous-classe de C, alors A est une sous-classe de C. Cette propriété est capturée par la règle de propagation *sub-trans* présentée en Figure 2.
Idempotence : afin de gérer la taille du *store* et d'optimiser les performances, la règle d'idempotence *sub-idem* (Figure 2) est mise en place et supprime des doublons de contraintes owlSubClassOf. De façon plus générale, des règles d'idempotence sont mises en œuvre pour supprimer les doublons de toutes les contraintes.

3.2.2 DisjointClasses

Règle de vérification : cette règle permet de détecter une violation de la disjonction de classes et de déclencher un échec provoquant un arrêt immédiat du raisonnement (règle *disj-verif*, Figure 2).
Si deux classes sont disjointes et qu'un individu est déclaré comme appartenant aux deux, alors il y a une incohérence dans la base.

3.2.3 ComplementOf

Contrairement aux raisonneurs classiques qui traitent le complément de manière indirecte via une preuve par contradiction en réduisant chaque inférence à un test de consistance, notre approche repose sur une gestion explicite du complément dans le *store* de contraintes.
Là où un raisonneur basé sur les Tableaux [3] ajoute la négation d'une proposition pour vérifier si elle mène à une contradiction, nous introduisons la notion d'assertion provisoire avec la contrainte owlClassAssertion(X, C, false).
Les règles CHR suivantes orchestrent ce mécanisme.
Initialisation du complément : pour toute classe A, tout individu nommé est provisoirement considéré comme appartenant à la classe complémentaire $NOTA$ (règle *comp-init*, Figure 2).
Validation de l'assertion : dès qu'une appartenance définitive à la classe complémentaire est présente dans le *store*, owlClassAssertion(X, $NOTA$, true), elle rend l'assertion provisoire owlClassAssertion(X, $NOTA$, false) obsolète et la supprime pour ne conserver que l'information certaine. (règle *comp-valid*, Figure 2).
Élimination par l'original : si au cours de la saturation, l'individu X est identifié comme appartenant à la classe d'origine A avec certitude, alors l'hypothèse selon laquelle il appartient à son complément $NOTA$ devient logiquement impossible. La règle élimine alors l'assertion provisoire false du *store* (règle *comp-elim*, Figure 2).
Vérification de la cohérence : si un individu est marqué à la fois dans une classe et son complément de façon définitive, une erreur est levée (règle *comp-verif*, Figure 2)

4 Validation expérimentale

Pour mettre en œuvre les mécanismes décrits précédemment, nous avons développé une implémentation fonctionnelle de l'outil **owl2chr**, dont le code source est disponible sur GitHub [1]. Nous avons implémenté nos règles CHR au sein du système CHR++ [4, 5], un système intégrant les Constraint Handling Rules (CHR) dans le langage C++. Les autres composants de l'outil comme l'analyseur syntaxique sont écrits directement en C++ à l'aide de bibliothèques dédiées.

4.1 Architecture et fonctionnement de owl2chr

Conformément aux usages en CHR++, un *espace* a été créé et contient le *store de contraintes* et les règles d'inférence.
Un analyseur syntaxique basé sur COWL [7, 8], une bibliothèque C++ légère conçue pour charger et parcourir des ontologies OWL 2, a pour rôle d'analyser un fichier OWL et de peupler l'espace. Cet analyseur parcourt l'ontologie lue au format fonctionnel [18]. Pour importer une ontologie aux formats RDF/XML ou Turtle, il faudra préalablement utiliser un convertisseur vers le format fonctionnel, tel que ROBOT CLI [2], un outil en ligne de commande permettant de manipuler, transformer et exporter des ontologies OWL. Pour chaque axiome l'analyseur en extrait son type et les URIs manipulées.
Plutôt que d'utiliser directement ces URIs au sein des contraintes, nous introduisons une contrainte logicalName faisant office de table de correspondance. Celle-ci asso-

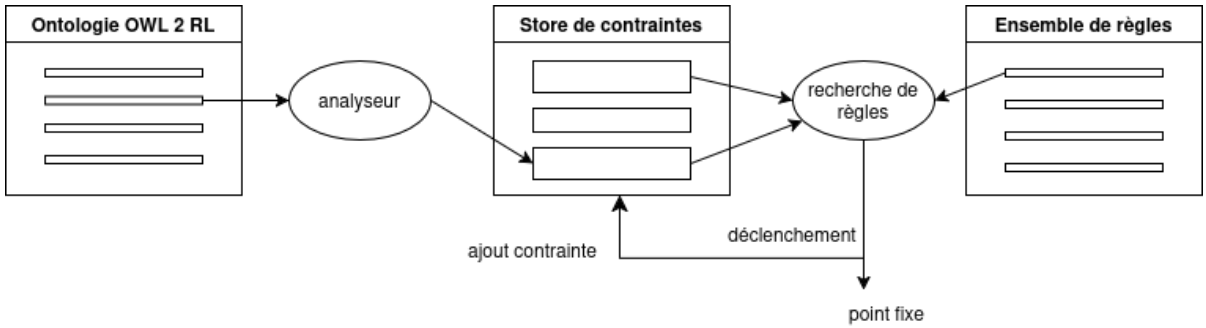

FIGURE 3 – Processus de saturation à partir d'un axiome

$$\textit{classification-rule} \ @ \ \text{classification}(), \text{owlSubClassOf}(X, Y) \Rightarrow \text{stockerClassification}(X, Y)$$
$$\textit{realisation-rule} \ @ \ \text{realisation}(), \text{owlClassAssertion}(X, C, true) \Rightarrow \text{stockerRealisation}(X, C)$$

FIGURE 4 – Règles CHR d'interrogation

cie chaque URI à une variable logique (*logical var*). La contrainte représentant l'axiome est ajoutée au *store* en utilisant ces variables logiques comme arguments. Cette approche permet au moteur CHR++ de réaliser des inférences par simple unification et d'éviter les comparaisons coûteuses de chaînes de caractères. Afin d'illustrer ce mécanisme de traduction, nous présentons ci-après un exemple concret de traitement de l'axiome suivant :

SubClassOf(:LeisureStudent :Student)

Pour chaque URI rencontrée, une contrainte logicalName est ajoutée au *store* et lie l'URI à une variable logique. Celle-ci est soit créée pour l'occasion, soit récupérée si l'URI a déjà été rencontrée. Enfin, la contrainte owlSubClassOf est instanciée en utilisant ces variables logiques.

```
logicalName("LeisureStudent", logicalVar1)
logicalName("Student", logicalVar2)
owlSubClassOf(logicalVar1, logicalVar2)
```

Le comportement de owl2chr repose sur un traitement séquentiel des axiomes de l'ontologie. Comme l'illustre la Figure 3, pour un axiome donné, l'analyseur syntaxique génère une contrainte dans le *store* qui déclenche des règles CHR par propagation. Ce processus récursif se poursuit jusqu'à l'obtention d'un point fixe (plus aucune règle ne peut se déclencher), permettant alors de passer à l'axiome suivant jusqu'à saturation complète du *store* de contraintes.
Une fois le chargement terminé et la base stabilisée, les connaissances peuvent être interrogées via des « contraintes but ». Contrairement aux contraintes de structure, ces dernières ne stockent pas d'information mais agissent comme des déclencheurs de règles pour extraire des faits. Il est important de souligner que le comportement de ces contraintes but suit rigoureusement la même logique que celle des axiomes : leur introduction active une chaîne de déductions qui se poursuit jusqu'à l'obtention d'un nouveau point fixe.

Comme l'illustre la Figure 4, les mécanismes de classification et de réalisation reposent sur l'activation de règles d'interrogation par des contraintes but (classification() et realisation()). Ces règles identifient les contraintes d'intérêt dans le *store* : respectivement owlSubClassOf et owlClassAssertion, afin d'en extraire les données.
Il est important de noter qu'en plus de manipuler des contraintes CHR, le système permet de solliciter des *built-ins* implémentés dans le langage hôte (C++). Ces fonctions, directement appelables depuis le corps des règles, permettent d'exécuter des traitements impératifs et de déléguer certaines tâches au code natif. Dans notre approche, les règles *classification-rule* et *realisation-rule* exploitent ce mécanisme pour appeler les fonctions C++ stockerClassification et stockerRealisation, lesquelles assurent l'exportation des connaissances extraites vers des fichiers de sortie dédiés.

4.2 Test de owl2chr

Afin de valider cette implémentation, nous nous appuyons sur OWL2Bench [21], un benchmark de référence conçu pour l'évaluation des performances et de la scalabilité des raisonneurs d'ontologies et des moteurs de requêtes SPARQL.

OWL2Bench fournit des ontologies synthétiques combinant une taille importante et une expressivité logique élevée. Il s'appuie sur une extension du University Ontology Benchmark (UOBM) et permet d'évaluer les systèmes de raisonnement dans des conditions réalistes et exigeantes. Il se présente sous la forme de deux outils : le premier permet de générer des bases de connaissances représentant des universités. Tandis que le second mesure les performances des raisonneurs sur ces bases.
Le générateur permet de produire des ontologies de tailles arbitraires en faisant varier le nombre d'universités modélisées. Cela augmente le volume d'assertions d'individus tout en conservant une hiérarchie de classes stable et complexe. Le générateur permet aussi de choisir un profil du langage

OWL 2 (EL, QL, RL ou DL), l'utilisateur peut isoler les constructeurs propres au profil choisi et éviter les inférences hors périmètre.

L'outil d'évaluation permet, pour un raisonneur donné, d'exécuter individuellement la classification ou la réalisation. Il fournit également une suite de requêtes SPARQL complexes, nécessitant un raisonnement sémantique approfondi pour produire des résultats complets, afin d'étudier la montée en charge des systèmes. OWL2Bench utilise l'OWL API pour piloter des raisonneurs tels que HermiT, JFact, Pellet et Openllet. Il mesure le temps de calcul propre à chaque tâche, en intégrant explicitement le temps de chargement de l'ontologie ainsi que le coût d'initialisation du raisonneur.

Nous avons utilisé OWL2Bench en nous basant sur le profil OWL 2 RL, puisque c'est celui que nous prenons en compte dans notre approche.

Validation des résultats. Nous cherchons avant tout à nous appuyer sur un cadre expérimental solide afin de démontrer l'exactitude et la complétude des inférences produites par notre implémentation CHR++ sur trois tâches distinctes que sont la classification, la réalisation et l'évaluation de requêtes SPARQL. Les différentes tâches ont été exécutées à partir des fichiers fournis par le benchmark pour le raisonneur Pellet, puis reproduites avec owl2chr en introduisant les contraintes but classification et realisation, ainsi que des contraintes spécifiques traduisant les requêtes SPARQL. Les expérimentations ont été menées sur une ontologie de taille correspondant à une université, soit un volume de 50120 assertions. Dans les trois cas considérés, les résultats obtenus sont strictement identiques, ce qui démontre la cohérence de notre implémentation avec les résultats de référence.

Mesures de performance. Dans un second temps, et bien que l'objectif de owl2chr ne soit ni la prise en charge de volumes massifs de faits ni l'optimisation des performances sur de très larges bases de connaissances, nous avons néanmoins mesuré, à titre indicatif, les temps de calcul nécessaires à la classification (CC) et à la réalisation (CR) pour les raisonneurs Pellet, JFact, HermiT, Openllet, ainsi que owl2chr, afin d'obtenir un ordre de grandeur des coûts de raisonnement. Les expérimentations ont été réalisées sur un ordinateur équipé d'un processeur Intel Core i5-1335U cadencé jusqu'à 4,6 GHz, avec 32 Go de mémoire vive, sous Debian GNU/Linux 12. Afin de respecter le protocole du benchmark de référence, nous avons opté pour une moyenne de cinq lancements indépendants avec un timeout (t.o) de 45 minutes, un échantillonnage jugé amplement suffisant au regard de la faible dispersion des résultats entre les différentes exécutions. Les résultats présentés dans le tableau 1 montrent que, malgré l'absence d'optimisations spécifiques, les temps d'exécution d'owl2chr sont comparables à ceux de raisonneurs de référence sur les bases de connaissances du benchmark. En conséquence, la traçabilité des inférences offerte par notre approche ne s'accompagne pas de performances insuffisantes empêchant son usage dans des cas réels.

TABLE 1 – Temps de calcul (en secondes) pour la classification (CC) et la réalisation (RT) selon le nombre d'universités et le raisonneur utilisé.

Tâche	**Raisonneur**	**Nb. d'universités**	
		1	2
CC	Pellet	3.55	8.50
	JFact	1392.00	t.o
	HermiT	58.16	355.50
	Openllet	6.33	14.75
	owl2chr	38.26	174.60
RT	Pellet	3.42	8.69
	JFact	1182.00	t.o
	HermiT	111.40	712.50
	Openllet	15.00	14.51
	owl2chr	38.29	172.84

Il est important de noter qu'une part importante du temps d'exécution de owl2chr est consacrée à la saturation de la base. Par conséquent, une utilisation limitée à une seule requête par appel ne permet pas de mettre en valeur les capacités de notre outil, celui-ci n'ayant pas été conçu pour cet usage atomique.

L'intérêt de notre approche réside dans la réutilisation de la base saturée : une fois cette étape accomplie, le temps nécessaire pour la réalisation et la classification devient négligeable. Lors d'un lancement de owl2chr avec les deux contraintes but permettant la classification et la réalisation, sur la même ontologie utilisée précédemment, le temps consacré à la saturation représente 99,7 % du temps d'exécution total qui est de 38.28 secondes. Le temps pris par l'ajout des contraintes buts classification et réalisation, les déclenchements des règles associées, et l'exécution des fonctions C++ de sauvegarde des résultats est négligeable.

5 Discussion

Bien que notre modélisation soit actuellement restreinte au profil **OWL 2 RL**, elle pose les bases d'une extension possible vers la totalité de la spécification **OWL 2**. Les résultats expérimentaux montrent que notre implémentation se situe dans le même ordre de grandeur de temps de calcul que la plupart des raisonneurs de référence auxquels elle a été comparée. Des optimisations ultérieures sur les règles permettraient de réduire les temps de calcul lors de la phase de saturation.

Contrairement aux raisonneurs classiques souvent perçus comme des « boîtes noires », notre approche offre plusieurs avantages structurels comme la modularité et la flexibilité. En effet l'ajout, le retrait ou la modification de règles s'effectuent sans remettre en cause l'architecture globale du système. Il est ainsi possible d'adapter le comportement du raisonneur à des domaines spécifiques ou d'ajuster la granularité des inférences en activant ou désactivant sélectivement certains ensembles de règles, voire en modifiant le comportement interne de règles existantes. De plus, l'interaction avec le langage hôte permet d'ajouter des règles déclenchant des actions supplémentaires à chaque

fois qu'une contrainte est postée, transformant ainsi le raisonneur statique en un système réactif capable d'interagir dynamiquement. En outre, notre approche est extensible et compacte. À la différence des outils comme *Pellet* qui reposent sur des milliers de lignes de code impératif complexe, owl2chr se distingue par sa concision. L'intégralité de la logique d'inférence de OWL 2 RL est couverte par seulement **133 règles** (auxquelles s'ajoutent 86 règles dédiées à la suppression des doublons) qui demeurent totalement lisibles. Chaque étape du raisonnement est traçable, facilitant ainsi la maintenance et l'évolution du moteur d'inférence par rapport aux implémentations traditionnelles en C++ ou Java. Cette traçabilité permet d'expliquer chaque inférence mais aussi de localiser précisément l'origine des incohérences détectées.

Cette transparence s'accompagne d'une forte robustesse technique. En effet, la sémantique de CHR garantit qu'une règle ne peut jamais être déclenchée plusieurs fois pour une même combinaison de contraintes CHR. Ce mécanisme, renforcé par des règles d'idempotence, empêche l'introduction de contraintes redondantes. L'ensemble des faits déductibles étant fini, l'algorithme procède par saturation jusqu'à atteindre un point fixe, ce qui en garantit la terminaison. De la même manière, en cas d'inconsistance, des règles de détection d'incohérence sont déclenchées, produisant une contradiction et entraînant l'arrêt immédiat de la saturation. Il est possible d'associer à ces règles un comportement spécifique, permettant d'informer l'utilisateur de l'inconsistance et de lui expliquer les causes ayant conduit à l'arrêt des inférences.

En résumé, si la performance pure n'a pas été un objectif premier lors de la conception de owl2chr, sa transparence et sa facilité d'adaptation en font une alternative robuste pour des environnements nécessitant un raisonnement sur mesure et évolutif.

6 Conclusion et perspectives

Dans cet article, nous avons présenté une approche de raisonnement pour les ontologies **OWL 2 RL** fondée sur le langage **CHR**. Les axiomes d'une ontologie sont traduits en contraintes, tandis que le moteur d'inférence repose sur un ensemble stable de règles déclaratives. Dans ce cadre, nous avons proposé owl2chr, une implémentation concrète de notre approche développée avec le système CHR++. Nous avons montré qu'un tel formalisme permet de reproduire fidèlement le comportement d'un raisonneur.

La validation expérimentale sur le benchmark **OWL2Bench** confirme la validité de notre implémentation, avec des résultats identiques à ceux des outils de référence et des temps d'exécution du même ordre de grandeur qu'une grande partie des raisonneurs actuels. Une fois la phase de saturation effectuée, l'interrogation de la base devient quasi-immédiate, ce qui rend l'approche particulièrement adaptée aux scénarios nécessitant de multiples requêtes.

Ainsi, **owl2chr** se positionne comme un moteur de raisonnement explicable, extensible et adaptable, destiné aux environnements où la maîtrise fine du processus d'inférence constitue un enjeu central.

La perspective principale de ce travail est d'étendre owl2chr à d'autres profils voire la totalité du langage OWL 2. Une fois ce développement achevé, un cas d'usage concret serait l'utilisation de l'outil pour prototyper des extensions d'OWL 2 ou concevoir des sous-ensembles du langage sur mesure, pour des applications réelles. Il suffirait alors de désactiver, modifier ou ajouter des règles CHR pour adapter les types de raisonnements effectués.

Par ailleurs, la structure déclarative facilite également l'intégration d'extensions telles que la manipulation de variables numériques ou l'ajout de règles métier personnalisées, ce qui permet d'envisager une ouverture vers **SWRL** [20].

Références

[1] OWL to CHR++. https://github.com/arigraphitech/owlChrpp, 2025.

[2] ROBOT : A command-line tool for ontology management. http://robot.obolibrary.org, 2026. Version 1.8.3, consulté le 9 février 2026.

[3] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. In *Handbook of Modal Logic*. Elsevier, 2007.

[4] V. Barichard. CHR++ : An efficient CHR system in C++ with don't know non-determinism. *Expert Systems with Applications*, 221 :119548, 2023.

[5] V. Barichard. Solveurs et systèmes pour la résolution de problèmes sous contraintes, 2025. HDR, Université d'Angers.

[6] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5) :34–43, May 2001.

[7] Ivano Bilenchi, Filippo Gramegna, Giuseppe Loseto, Saverio Ieva, Floriano Scioscia, and Michele Ruta. Cowl : pushing OWL 2 over the edge. *Elsevier Internet of Things Journal*, 29, January 2025. [core publication].

[8] Ivano Bilenchi, Floriano Scioscia, and Michele Ruta. Cowl : a lightweight OWL library for the semantic web of everything. In *First International Workshop on the Semantic Web of EveryThing (SWEET 2022), co-located with the 22nd International Conference on Web Engineering (ICWE 2022)*, 2022. https://swot.sisinflab.poliba.it/cowl/.

[9] T. Frühwirth. Constraint handling rules. Technical report, ECRC, 1992.

[10] T. Frühwirth. Constraint handling rules. In *Constraint Programming : Basics and Trends*, pages 90–107, 1994.

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

[12] T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[13] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer-Verlag, 2003.

[14] T. Frühwirth, A. Herold, V. Küchenhoff, T. L. Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint logic programming – an informal introduction. In *Logic Programming in Action*, volume 636 of *LNCS*, pages 3–35. Springer-Verlag, 1993.

[15] T. Frühwirth and F. Raiser. *Constraint Handling Rules : Compilation, Execution, and Analysis*. Cambridge University Press, 2011.

[16] A. S. Galarraga. Openllet : An open source OWL 2 reasoner, 2024. https://github.com/Galigator/openllet.

[17] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang. HermiT : An OWL 2 reasoner. *Journal of Automated Reasoning*, 53(3) :245–269, 2014.

[18] W3C OWL Working Group. OWL 2 web ontology language : Structural specification and functional-style syntax, dec 2012. https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/.

[19] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph. OWL 2 web ontology language profiles (second edition), 2012. https://www.w3.org/TR/owl2-profiles/.

[20] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Michael Dean. SWRL : A semantic web rule language combining OWL and RuleML. W3c member submission, W3C, 2004. https://www.w3.org/Submission/SWRL/.

[21] Gunjan Singh, Sumit Bhatia, and Raghava Mutharaju. OWL2Bench : A benchmark for OWL 2 reasoners. In *The Semantic Web – ISWC 2020 : 19th International Semantic Web Conference, Athens, Greece, November 2–6, 2020, Proceedings, Part II*, page 81–96, Berlin, Heidelberg, 2020. Springer-Verlag.

[22] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet : A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2) :51–53, 2007.

[23] E. Sirinschi and D. Tsarkov. JFact : An OWL 2 reasoner. http://jfact.sourceforge.net/.

[24] Andreas Steigmiller, Thorsten Liebig, and Birte Glimm. Konclude : System description. *Journal of Web Semantics (JWS)*, 27 :78–85, 2014. https://www.derivo.de/en/products/konclude/.

[25] W3C. OWL 2 web ontology language : Document overview, 2012. https://www.w3.org/TR/owl2-overview/.