



LERIA

UNIVERSITÉ D'ANGERS

UFR Sciences
Master 1 informatique

RAPPORT DE STAGE

Représentation d'inférences OWL2 en CHR++

Réalisé par : Arina Abed

Lieu de stage : Laboratoire d'Étude et de Recherche Informatique d'Angers LERIA

Encadrants :

Dr. Vincent Barichard

Dr. David Genest

Dr. Eric Monfroy

Date de soutenance : 17 juin 2025

Remerciements

Je tiens à exprimer ma gratitude à l'Université d'Angers pour avoir rendu possible la réalisation de ce stage au sein du Laboratoire d'Étude et de Recherche Informatique d'Angers (LERIA), qui se poursuit encore après la soutenance de ce rapport. Je remercie également mes professeurs dont les enseignements tout au long de cette année de Master ont été déterminants pour mener à bien ce projet.

Je suis particulièrement reconnaissante envers mes encadrants, **Dr. Vincent Barichard**, **Dr. David Genest** et **Dr. Eric Monfroy**, pour leur disponibilité, leur soutien constant et leurs conseils inspirants, non seulement tout au long de ce stage, mais aussi pendant mes études depuis ma licence jusqu'à cette année de Master. J'espère avoir l'opportunité de collaborer de nouveau avec eux à l'avenir.

Enfin, je remercie chaleureusement ma famille pour leur soutien inconditionnel, leur bienveillance et leurs encouragements tout au long de mon parcours académique.

Table des matières

1	Introduction	3
2	Cadre théorique	5
2.1	Le Web sémantique et les ontologies	5
2.1.1	RDF (Resource Description Framework)	5
2.1.2	RDFS (RDF Schema)	6
2.1.3	Ontologies : vocabulaire et schéma	7
2.2	OWL (Web Ontology Language)	7
2.2.1	Les datatypes et OWL 2	9
2.2.2	OWL 2 et ses profils	10
2.2.3	Syntaxes owl	11
2.2.4	Architecture du Web sémantique	11
2.3	Constraint Handling Rules (CHR)	11
2.4	CHR++ : CHR intégré à C++	13
3	Objectif du stage	14
4	Représentation en CHR d'une ontologie OWL	16
4.1	Prise en charge des axiomes OWL	16
4.2	Representation d'inférences OWL sous forme de règles CHR++	18
4.3	Vers une base de faits saturée	21
4.4	Interrogation de la base	21
5	Tests et résultats	22
5.1	Gestion avancée des <i>Datatype properties</i>	23
5.2	Exemples de requêtes	23
5.3	Test du prototype	24
5.4	Bilan des résultats	25
6	Conclusion	26
7	Bibliographie / Webographie	28

1 Introduction

Dans le cadre de mon Master 1 en Informatique à l'Université d'Angers, j'effectue un stage au sein du laboratoire LERIA, encadré par Vincent Barichard, David Genest et Éric Monfroy, portant sur l'inférence dans les ontologies à l'aide de CHR++ (Constraint Handling Rules).

Les ontologies, notamment celles exprimées en OWL 2 (Web Ontology Language), jouent un rôle fondamental dans la structuration, le partage et l'interrogation des connaissances. Cependant, les moteurs de raisonnement associés à OWL, tels que HermiT ou Pellet, fonctionnent comme des boîtes noires : leur logique interne est difficilement modifiable et leurs inférences sont recalculées à chaque requête plutôt que stockées, ce qui limite la transparence du raisonnement et la possibilité d'adapter le processus à des besoins spécifiques.

Ce stage propose une approche alternative, où les inférences du langage OWL 2 sont exprimées sous forme de règles CHR++, en transformant les représentations de connaissances en contraintes et en reproduisant le fonctionnement d'un moteur de raisonnement classique. L'objectif est de concevoir un système transparent et modulable, dans lequel chaque axiome OWL est modélisé comme une contrainte, et chaque mécanisme d'inférence (spécialisation, transitivité, cardinalités, détection d'incohérences, etc.) est formalisé sous forme de règle CHR++. Contrairement aux approches classiques, les faits déduits sont explicitement ajoutés à la base de connaissances, garantissant une persistance totale et une interrogation directe sans recalcul, comme illustré par la figure 1.

La mise en œuvre de cette idée a conduit au développement d'un prototype, capable d'analyser une ontologie OWL, de la traduire automatiquement en contraintes CHR++, d'appliquer les règles d'inférence et de permettre l'interrogation de la base résultante via des requêtes formulées sous forme de contraintes. Ce système ne se limite pas à l'exécution d'un raisonnement, mais le rend transparent, saturé et immédiatement exploitable. Bien qu'il ne prenne en charge qu'un sous-ensemble du langage OWL 2 à ce stade, il a démontré l'intérêt de cette approche, notamment en termes de lisibilité du raisonnement, modularité et performances.

Contrairement aux raisonneurs OWL classiques, qui utilisent SPARQL pour interroger des inférences calculées à la volée, notre système repose sur un raisonnement saturé et directement interrogeable via des contraintes-but. Ces contraintes jouent un rôle actif : elles déclenchent des règles conçues pour explorer la base de connaissances et en extraire les informations pertinentes.

L'introduction d'une contrainte-but ne vise donc pas à stocker de l'information, comme c'est le cas avec les contraintes représentant les axiomes OWL, mais à activer un raisonnement ciblé. Une fois introduite, elle initie un processus qui structure les résultats pour un affichage immédiat ou une exploitation ultérieure. Cette approche unifie la modélisation des axiomes, des inférences et des requêtes sous un même formalisme déclaratif, garantissant ainsi une cohérence d'ensemble et une transparence totale du fonctionnement du moteur.

Ce travail s'inscrit à l'intersection du web sémantique, de la représentation des connaissances et de la programmation par contraintes. En établissant un lien entre le formalisme d'OWL 2 et un système de règles exécutables comme CHR++, il facilite le développement de raisonneurs personnalisés et rend l'inférence directement exploitable, en intégrant explicitement les connaissances inférées à la base.

Ce rapport présente d'abord les fondements théoriques de cette approche, notamment

les concepts clés du Web sémantique, les spécificités du langage OWL 2 et le formalisme CHR++. Il détaille ensuite les objectifs du stage ainsi que la méthodologie adoptée, notamment la traduction d'une ontologie OWL en contraintes CHR++, la formalisation du raisonnement à travers des règles et la mise en place de requêtes ciblées. Enfin, il expose les règles développées, les résultats obtenus lors des tests, ainsi que les limites actuelles du prototype et ses perspectives d'évolution.

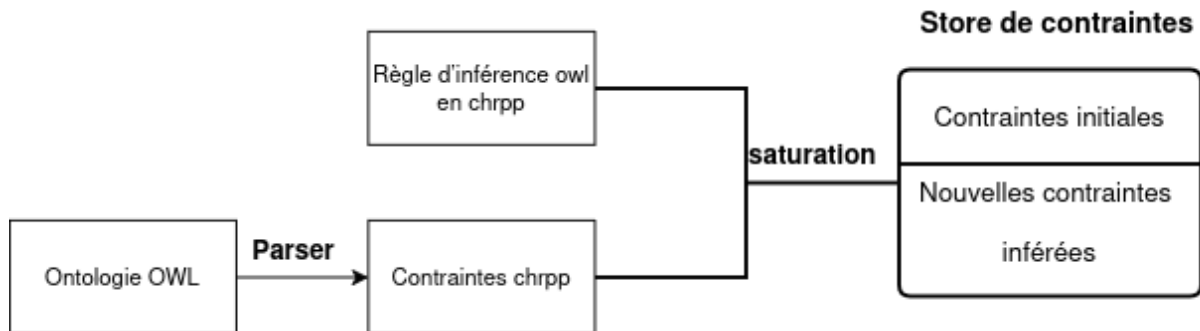


FIGURE 1 – Vue globale du système.

2 Cadre théorique

Avant de détailler notre approche, il est indispensable de comprendre les deux piliers sur lesquels elle s'appuie : le Web sémantique via les ontologies, et plus particulièrement le standard OWL 2 et la programmation par contraintes à l'aide des *Constraint Handling Rules* (CHR), dans leur version étendue CHR++. Nous allons donc passer en revue ces concepts clés.

2.1 Le Web sémantique et les ontologies

Le Web « classique » permet principalement de publier et naviguer dans des documents munis d'hyperliens : l'accent est mis sur l'affichage et la présentation. Le *Web sémantique*¹, en revanche, vise à doter chaque donnée d'une signification formelle ou *sémantique* pour que des agents logiciels puissent non seulement lire, mais *comprendre* et *raisonner* sur ces informations. Dans cette section, nous passerons en revue les briques fondamentales du Web sémantique, à savoir RDF, RDFS, les ontologies et OWL, qui sont essentielles pour la compréhension des travaux présentés dans la suite de ce rapport.

2.1.1 RDF (Resource Description Framework)

Le *Resource Description Framework* (RDF²) est la brique de base du Web sémantique. RDF organise l'information sous forme de *triplets* :

(sujet, prédicat, objet)

Le *sujet* et le *prédicat* sont des **URI** (Uniform Resource Identifier), garantissant l'unicité mondiale et l'*objet* peut être soit un URI, soit un littéral (chaîne de caractères, nombre, date).

RDF fournit notamment la propriété standard `rdf:type` pour déclarer qu'une ressource est instance d'une classe.

Un triplet RDF peut être représenté sous la forme d'un graphe orienté étiqueté comme l'illustre la figure 2.

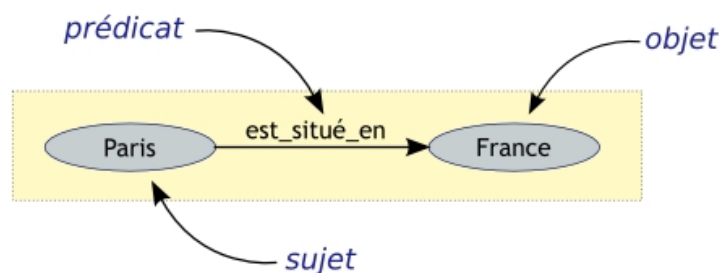


FIGURE 2 – Visualisation d'un triplet RDF : sujet Paris, prédicat `est_situé_en`, objet France.

1. https://fr.wikipedia.org/wiki/Web_s%C3%A9mantique/

2. <https://www.w3.org/RDF/>

Exemple minimal

```
<http://exemple.org#Alice> rdf:type      <http://exemple.org#Personne> .  
<http://exemple.org#Alice> ex:age       "42"^^xsd:integer      .
```

Ici, on affirme qu’Alice est une instance de la classe **Personne** et qu’Alice a pour âge la valeur entière 42.

Pour mieux comprendre l’organisation des triplets dans un contexte réel, la figure 3 illustre un graphe RDF annoté.

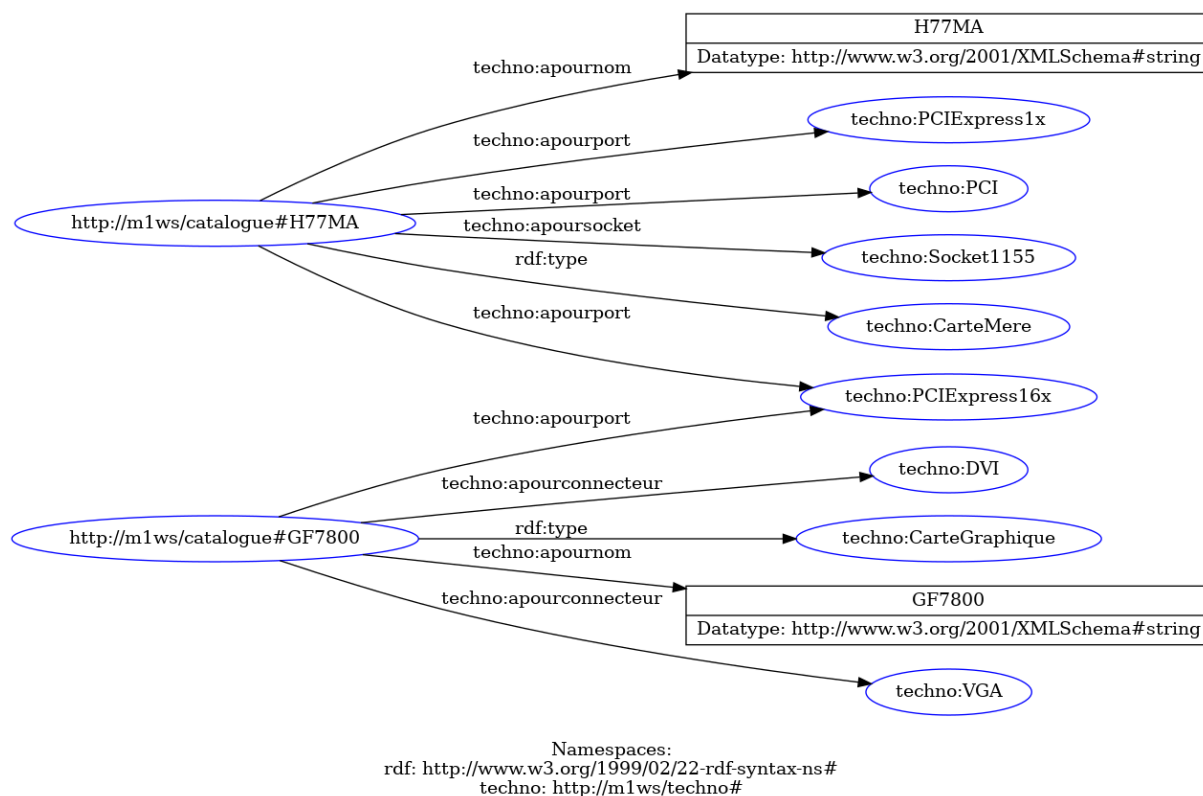


FIGURE 3 – Exemple de graphe RDF annoté : description de cartes mères et cartes graphiques.

2.1.2 RDFS (RDF Schema)

RDFS (RDF Schema³) étend RDF pour ajouter des mécanismes de structuration légère. Il permet de définir des classes via `rdfs:Class` et de les structurer dans une relation de spécialisation avec `rdfs:subClassOf`. De même, les propriétés sont définies avec `rdfs:Property`, et il est possible de leur associer un domaine (`rdfs:domain`) et une codomaine (`rdfs:range`). Grâce à ces mécanismes, certaines inférences peuvent être établies automatiquement. Par exemple, si `:Chien` est une sous-classe de `:Animal` et que Fido est un `:Chien`, on peut en déduire que Fido est aussi un `:Animal`.

Prenons un exemple de schéma en RDFS illustrant ces concepts :

3. <https://www.w3.org/TR/rdf-schema/>

```

:Personne    rdf:type    rdfs:Class .
:Etudiant    rdfs:subClassOf :Personne .
:age         rdf:type    rdf:Property .
:age         rdfs:domain  :Personne .
:age         rdfs:range   xsd:integer .

```

Nous avons une classe `:Personne`, représentant les individus, et une sous-classe `:Etudiant`, qui hérite des propriétés de `:Personne`. Une propriété `:age` est également définie, précisant que son domaine est `:Personne` et que sa valeur doit être un entier (`xsd:integer`). Cela signifie que toute instance de `:Personne` peut avoir un attribut `:age` dont la valeur sera un nombre.

Sur cette base formelle qu’offrent RDF et RDFS pour structurer et typer l’information, nous pouvons à présent aborder la notion d’ontologie, qui enrichit ce schéma léger par une modélisation plus expressive du domaine.

2.1.3 Ontologies : vocabulaire et schéma

Une ontologie est une **spécification formelle** d’un domaine de connaissance. Elle définit :

- **Classes (concepts)** : ce sont les types d’entités du domaine, par exemple `Personne` ou `Livre`.
- **Propriétés (relations)** : elles décrivent les liens entre entités, comme `auteurDe` ou `dateDeNaissance`.
- **Individus (instances)** : ce sont les objets concrets ou abstraits du domaine, par exemple `Alice` ou `Voldemort`.
- **Axiomes logiques** : ce sont des règles et des contraintes (sous-classes, cardinalités, domaines et portées, etc.) qui fixent la sémantique et organisent les relations entre classes et propriétés.

L’ontologie joue alors le rôle de **schéma partagé**, permettant à plusieurs applications de s’y référer pour échanger des données avec un vocabulaire commun, garantissant ainsi **l’interopérabilité** des systèmes.

Toutefois, pour une exploitation efficace des ontologies, il est nécessaire de s’appuyer sur des mécanismes avancés de formalisation et d’inférence. C’est précisément le rôle du Web Ontology Language (OWL), un langage conçu pour structurer, enrichir et exploiter les ontologies avec rigueur, facilitant ainsi leur interopérabilité et leur intégration dans des systèmes intelligents.

2.2 OWL (Web Ontology Language)

OWL⁴ enrichit RDFS en ajoutant des mécanismes avancés de représentation des connaissances, plus expressive et propice au *raisonnement automatisé*. Parmi ses principales caractéristiques :

- **Sousclasses, équivalences, disjonctions** de classes : une `:Voiture` peut être définie comme (équivalente à) un `:Véhicule` qui a 4 roues (liée par la propriété `:nombre_roue` à la valeur entière 4).

4. <https://www.w3.org/OWL/>

- **Restrictions de classes** (cardinalité minimale, maximale, exacte) : il est possible d'exiger qu'une classe possède un certain type de relation ou un nombre défini de propriétés. Ainsi, avec `owl:SomeValuesFrom`, on peut spécifier qu'un `:Véhicule` doit avoir au moins une `:roue`. À l'inverse, `owl:Cardinality(4)` impose que `:Voiture` possède exactement quatre `:roues`. Ces restrictions permettent de modéliser des concepts de manière plus précise.
- **Caractéristiques de propriétés** : transitivité (`owl:TransitiveProperty`), symétrie (`owl:SymmetricProperty`), réflexivité, etc.
- **ObjectProperties** (liens entre instances) et **DatatypeProperties** (liens vers des littéraux typés).

Grâce à OWL, il est possible de déduire automatiquement des relations implicites entre concepts, notamment en exploitant la spécialisation ou les propriétés transitives. De plus, ce langage permet de vérifier la cohérence d'une ontologie en identifiant les éventuelles contradictions, comme des instances qui ne pourraient satisfaire simultanément deux contraintes incompatibles.

Par ailleurs, OWL repose sur des **constructeurs**, qui sont les mécanismes permettant de définir des classes, des propriétés et des relations dans une ontologie. Ils renforcent l'expressivité du langage en structurant les connaissances sous une forme exploitable par les moteurs d'inférence. Parmi les principaux constructeurs :

- **Constructeurs de classes** : `owl:Class`, `owl:EquivalentClass`, `owl:DisjointClass`, permettant de définir, équivaloir ou exclure des classes.
- **Constructeurs de propriétés** : `owl:ObjectProperty`, `owl:DatatypeProperty`, `owl:InverseProperty`, facilitant la définition de relations entre individus et valeurs.
- **Restrictions et axiomes** : `owl:SomeValuesFrom`, `owl:AllValuesFrom`, `owl:hasValue`, qui permettent de préciser des contraintes sur les propriétés appliquées aux classes.

Ainsi, les constructeurs OWL permettent une modélisation explicite et formelle des connaissances, facilitant le raisonnement automatique et la validation de la cohérence des ontologies.

Exemple OWL simple (en syntaxe RDF/XML)

```
<owl:Class rdf:ID="Personne"/>
<owl:Class rdf:ID="Etudiant">
  <rdfs:subClassOf rdf:resource="#Personne"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="aPourEnfant">
  <rdf:type rdf:resource="owl:TransitiveProperty"/>
  <rdfs:domain rdf:resource="#Personne"/>
  <rdfs:range rdf:resource="#Personne"/>
</owl:ObjectProperty>

<owl:Restriction>
  <owl:onProperty rdf:resource="#aPourEnfant"/>
  <owl:cardinality rdf:datatype="xsd:nonNegativeInteger">2</owl:cardinality>
</owl:Restriction>
```

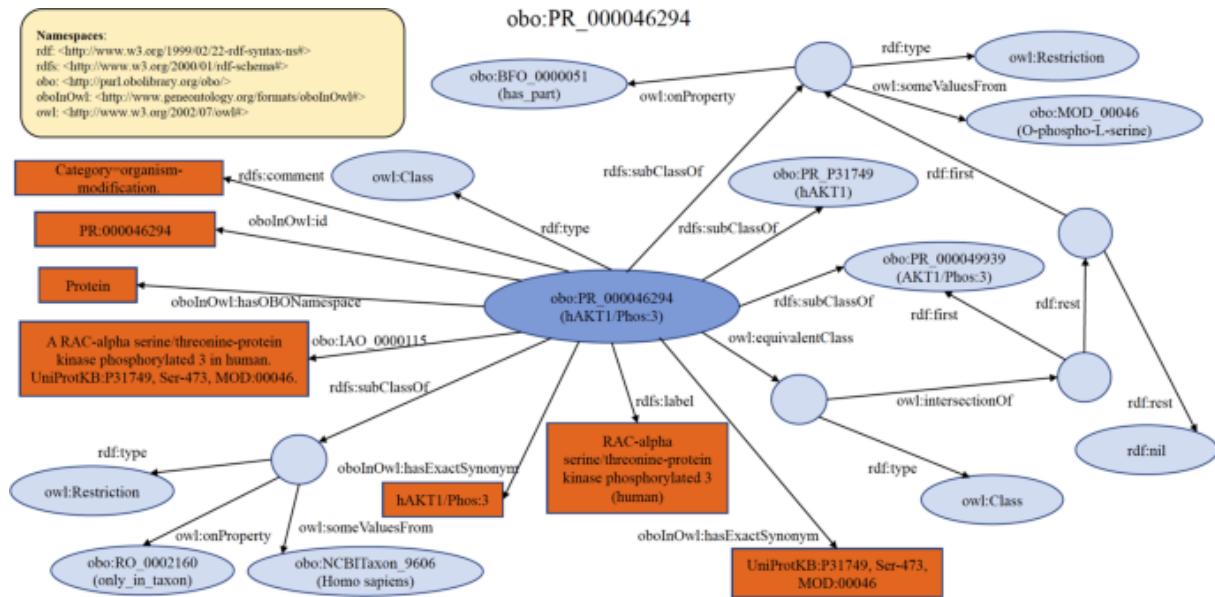


FIGURE 4 – Exemple d’ontologie OWL : Les formes elliptiques et circulaires sont des nuds RDF. Les formes rectangulaires représentent des littéraux RDF. Les arêtes dirigées correspondent aux propriétés RDF. Les formes circulaires symbolisent des classes anonymes ou des nuds blancs (Illustration tirée de Protein ontology on the semantic web for knowledge discovery)

Pour rendre plus concrète la structure d’une ontologie OWL, la figure 4 montre un schéma et ses instances.

2.2.1 Les datatypes et OWL 2

Nous avons vu la richesse d’expressivité d’OWL, notamment la possibilité d’utiliser des *data properties*, qui permettent d’associer des valeurs littérales aux individus. Dans cette continuité, il est essentiel de s’intéresser aux *datatypes*, qui jouent un rôle fondamental dans l’interprétation et la validation des données.

RDF et OWL reposent sur des *datatypes* définis par XML Schema (XSD⁵), offrant une base rigoureuse pour la gestion des valeurs littérales. Ces *datatypes* forment une hiérarchie structurée permettant de garantir la validité et la compatibilité des données manipulées. Ils définissent les types de valeurs possibles (entiers, chaînes de caractères, dates, booléens, etc.), facilitant ainsi l’intégration des données dans des raisonnements logiques et des inférences.

OWL 2, en particulier, étend la prise en charge des *datatypes* en introduisant des restrictions plus fines sur les valeurs et les expressions. Il permet par exemple de contraindre une propriété à un certain intervalle numérique (`xsd:int`, `xsd:float`) ou à un format de date spécifique (`xsd:dateTime`). Grâce à ces mécanismes, il devient possible d’exprimer des contraintes précises sur les données et d’améliorer la qualité des inférences.

La figure 5 présente l’arbre des *datatypes* XSD, illustrant la hiérarchie et les relations entre les types de données supportés. Cette structuration assure une compatibilité entre les systèmes et permet aux agents intelligents d’interpréter les données de manière cohérente et efficace.

5. <https://www.w3.org/XML/Schema/>

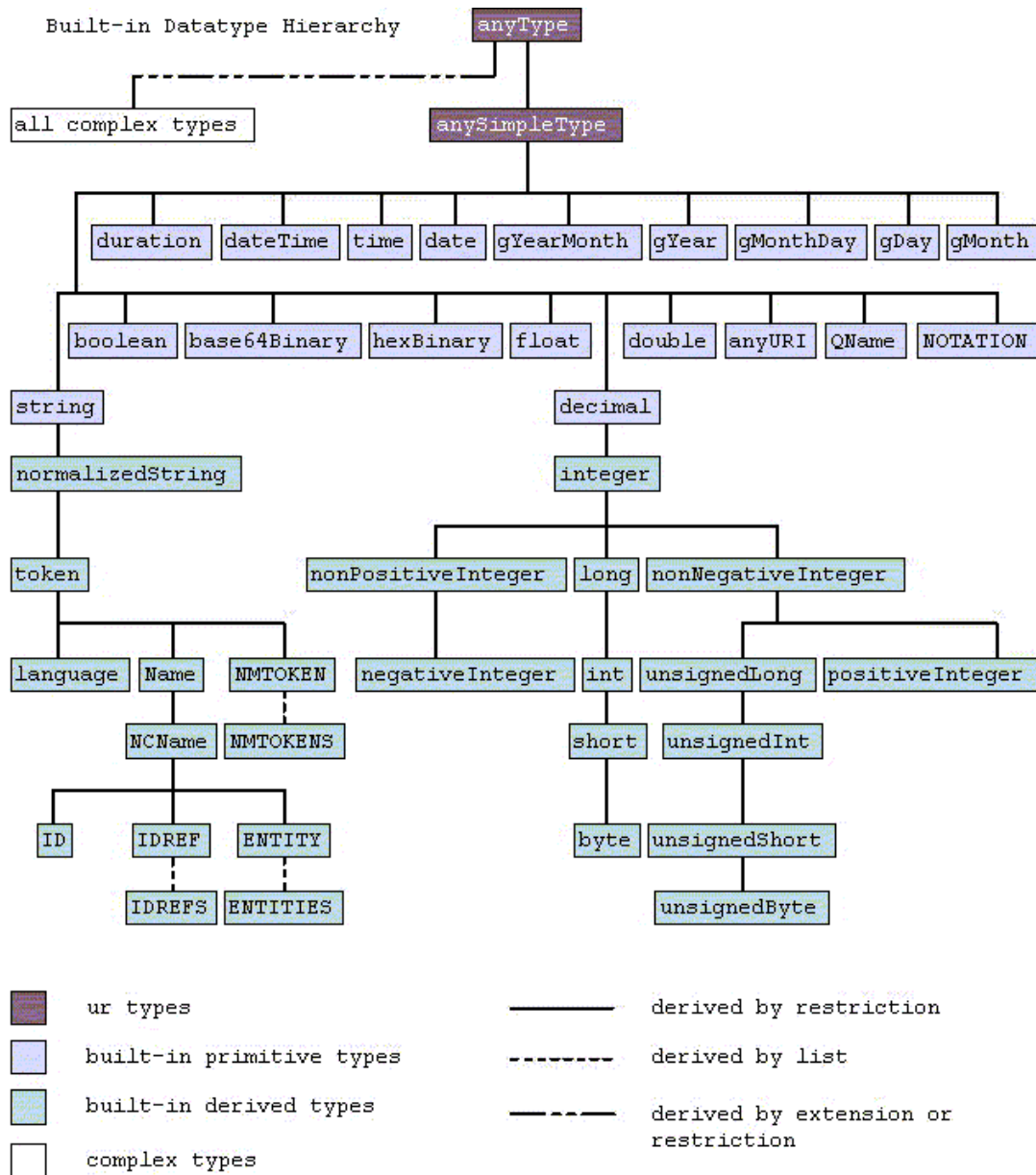


FIGURE 5 – Hiérarchie des datatypes XSD (XML Schema).

2.2.2 OWL 2 et ses profils

OWL 2⁶, normalisé en 2009, est la version actuelle d'OWL. Il offre une expressivité accrue, permettant de modéliser des concepts plus complexes et d'approfondir le raisonnement inférentiel. De plus, il introduit des *profils*, conçus pour garantir des performances déterministes adaptées à divers contextes d'utilisation :

OWL 2 EL Orienté *Large Ontologies* : expressivité modérée, mais raisonnement en temps polynomial (utile pour des ontologies très volumineuses).

OWL 2 QL Conçu pour *Querying Large Data* : optimisation des requêtes SPARQL sur de très grands ensembles d'instances, compatible avec bases de données relationnelles.

6. <https://www.w3.org/TR/owl2-overview/>

OWL 2 RL Ciblé *Rule-Based Reasoning* : restreint la logique pour correspondre à un raisonnement par règles (par exemple via un moteur de règles RIF), tout en permettant un raisonnement efficace.

Ces profils permettent à l'utilisateur de choisir un compromis adéquat entre expressivité et *scalabilité* du raisonnement :

- *EL* pour modéliser de larges taxonomies,
- *QL* pour interroger de grands volumes de données,
- *RL* pour intégrer des moteurs de règles existants.

2.2.3 Syntaxes owl

OWL permet d'exprimer les axiomes sous différentes syntaxes adaptées à divers usages et environnements. Voici les principales :

- *RDF/XML* : C'est la syntaxe standard basée sur XML, utilisée principalement pour l'interopérabilité avec les systèmes RDF. Elle est très répandue mais relativement verbeuse.
- *Turtle* : Une syntaxe plus compacte et lisible, qui facilite l'écriture et la compréhension des triplets RDF en comparaison avec RDF/XML. Elle est largement utilisée dans les bases de connaissances.
- *Syntaxe fonctionnelle* : Une notation proche de la logique formelle, adaptée aux outils de raisonnement et aux définitions précises d'ontologies. Elle permet une représentation claire des axiomes et des relations.

2.2.4 Architecture du Web sémantique

L'ensemble des technologies s'organise en couches, depuis l'identification des ressources jusqu'aux applications intelligentes, la figure 6 illustre l'architecture en couches du Web sémantique.

En synthèse, le Web sémantique repose sur un formalisme modulaire (RDF, RDFS, OWL/OWL 2) et des ontologies partageables, offrant un cadre structuré et formel pour la représentation des connaissances. Il permet non seulement de garantir la rigueur de ces représentations, mais aussi de les enrichir et d'en vérifier automatiquement la cohérence. Grâce aux ontologies, l'interopérabilité entre systèmes est facilitée, ouvrant la voie à des applications d'intelligence artificielle toujours plus performantes et autonomes.

2.3 Constraint Handling Rules (CHR)

Pour aborder la logique de raisonnement, découvrons CHR, Contrairement aux langages impératifs classiques, *Constraint Handling Rules* (CHR) est un langage déclaratif orienté réécriture de contraintes, conçu à l'origine pour Prolog. Un programme CHR manipule un *magasin de contraintes*, c'est-à-dire un multienemble de prédicats utilisateurs, en appliquant de manière itérative des règles de réécriture jusqu'à convergence. L'exécution suit un modèle *committedchoice* : une fois qu'une règle est appliquée, elle ne peut pas être annulée.

Types de règles CHR

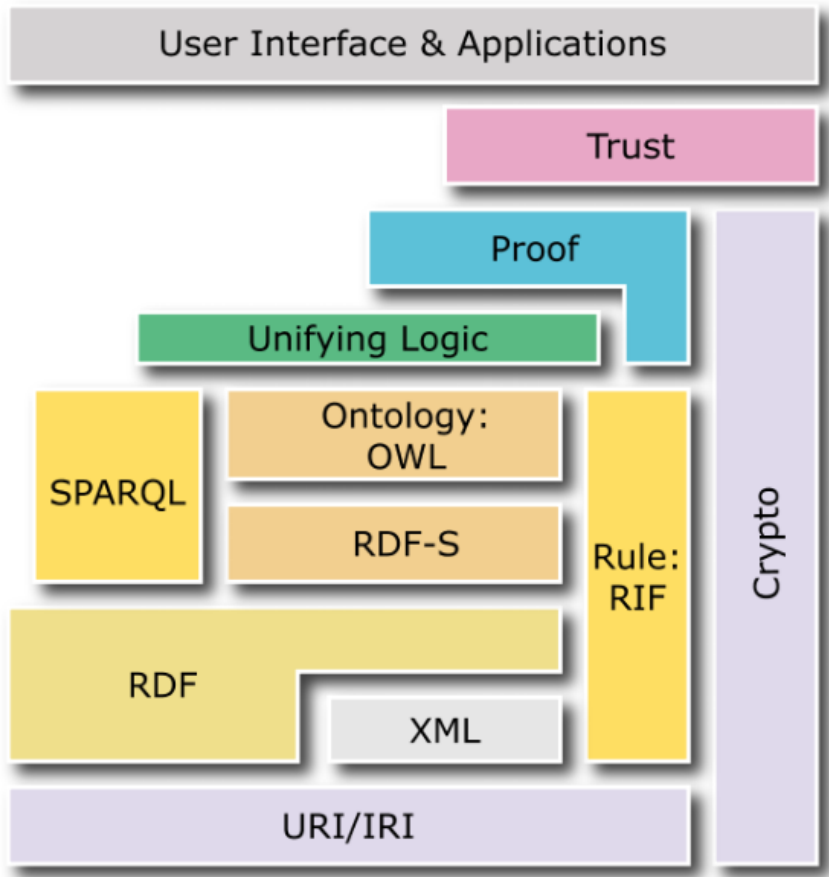


FIGURE 6 – architecture du Web sémantique (Illustration extraite de source.)

- **Simplification** ($\Leftarrow$) Remplace un ensemble de contraintes par un ensemble plus simple, préservant l'équivalence logique.
 $\text{nom} @ C1, C2 \Leftarrow \text{garde} \mid \text{corps};;$
- **Propagation** ($\Rightarrow$) Ajoute de nouvelles contraintes (redondantes) permettant de déclencher d'autres règles sans retirer les contraintes d'origine.
 $\text{nom} @ C1, C2 \Rightarrow \text{garde} \mid \text{corps};;$
- **Simpagation** ($\backslash \Leftarrow$) Supprime certaines contraintes et en conserve d'autres, combinant simplification et propagation.
 $\text{nom} @ C1 \backslash C2 \Leftarrow \text{garde} \mid \text{corps};;$

Exemple en CHR En CHR Prolog, le calcul du plus grand commun diviseur (PGCD) repose sur l'application de deux règles. La première, $\text{gcd}(0) \Leftarrow \text{success}()$, indique que lorsque la contrainte $\text{gcd}(0)$ est rencontrée, elle peut être supprimée et le calcul est terminé. La seconde règle, exploite une soustraction répétée pour réduire les valeurs jusqu'à obtenir une seule contrainte $\text{gcd}(D)$, où D représente le PGCD recherché. Cette approche fonctionne de manière itérative, chaque étape simplifiant le problème jusqu'à atteindre la solution finale.

```

gcd(0)  $\Leftarrow$  success()
gcd(N)  $\backslash$  gcd(M)  $\leq$  M  $\Rightarrow$  gcd(M - N)

```

Ces deux règles sont appliquées de façon itérative jusqu'à ce que l'espace de contraintes contienne une seule contrainte $\text{gcd}(D)$.

Exécution avec un exemple Prenons l'exemple du calcul de $\text{gcd}(48, 18)$. Nous initialisons l'espace des contraintes avec :

$\text{gcd}(48), \text{gcd}(18)$

Application successive des règles :

- $\text{gcd}(48) \setminus \text{gcd}(18) \Rightarrow \text{gcd}(48 - 18) = \text{gcd}(30)$
- $\text{gcd}(30) \setminus \text{gcd}(18) \Rightarrow \text{gcd}(30 - 18) = \text{gcd}(12)$
- $\text{gcd}(18) \setminus \text{gcd}(12) \Rightarrow \text{gcd}(18 - 12) = \text{gcd}(6)$
- $\text{gcd}(12) \setminus \text{gcd}(6) \Rightarrow \text{gcd}(12 - 6) = \text{gcd}(6)$
- $\text{gcd}(6) \setminus \text{gcd}(6) \Rightarrow \text{gcd}(0)$
- $\text{gcd}(0) \Rightarrow \text{success}()$

Conclusion À la fin de l'exécution, il ne reste qu'une seule contrainte active :

$\text{gcd}(6)$

Le raisonneur a donc trouvé que $\text{gcd}(48, 18) = 6$, comme attendu.

2.4 CHR++ : CHR intégré à C++

CHR++ est une réimplémentation moderne de CHR en C++, développée au LERIA de l'Université d'Angers. Son objectif est d'associer l'expressivité déclarative de CHR à la performance des langages compilés. Grâce à une API C++ légère, il peut être intégré facilement dans des projets existants sans nécessiter de dépendances complexes. Pour optimiser son exécution, il exploite intensivement les templates, gère finement l'allocation mémoire à l'aide de pools et de smart pointers, et utilise la généricité statique pour éliminer les surcoûts de calcul.

CHR++ prend en charge le nondéterminisme don't know, ce qui permet d'explorer différentes alternatives dans la résolution de problèmes NP. Sa syntaxe repose sur des blocs `<CHR name="">`, que le préprocesseur transforme en code C++ optimisé. Les variables logiques sont des objets templatisés, instanciés à l'aide de modèles génériques (`template<typename T>`), tandis que l'unification est réalisée via l'opérateur `%=`. Les gardes (`|`) et les corps de règles peuvent intégrer directement du code C++ hôte, offrant une grande flexibilité dans la définition des règles et des contraintes.

Exemple complet : calcul du PGCD

```
#include <chrpp.hh>

<CHR name="GCD">
  <chr_constraint> gcd(+unsigned long)

  // Simplification : suppression de gcd(0) succès
  stop @ gcd(0ul) <=> success();;

  // Simpagation : si N <= M, remplacer gcd(N), gcd(M) par gcd(M-N)
  compute @ gcd(N) \ gcd(M) <=> N <= M | gcd(M-N);;
```

</CHR>

```
int main() {  
    // Création d'un espace de contraintes pour le solveur "GCD"  
    auto space = GCD::create();  
    // Dépôt de deux contraintes dans l'espace  
    space->gcd(1000000000);  
    space->gcd(5);  
    return 0;  
}
```

- La règle **stop** élimine la contrainte `gcd(0u1)` et termine le processus.
- La règle **compute** effectue la soustraction répétée tant que les deux contraintes sont présentes.
- Le solveur s'arrête quand il ne reste qu'une seule contrainte `gcd(D)`, `D` étant le PGCD.

Bénéfices et perspectives Grâce à CHR++, on obtient :

- Un cadre modulaire pour développer rapidement de nouveaux solveurs de contraintes.
- Des performances comparables aux meilleures implémentations Java/C de CHR.
- Un moteur performant pour les problèmes de décision, d'optimisation (timetabling, planification) et les systèmes experts, avec gestion native du nondéterminisme.

Ainsi, CHR++ constitue une base solide pour la recherche et le développement de solveurs performants, alliant la clarté des règles CHR et l'efficacité du C++.

3 Objectif du stage

Dans la continuité du cadre théorique, nous abordons à présent les motivations et les objectifs spécifiques de ce stage, qui guideront les développements et expérimentations présentés dans la suite du rapport. Les moteurs de raisonnement classiques pour OWL, tels que HermiT, Pellet ou FaCT++, reposent sur la logique de description (Description Logics). Il s'agit d'une famille de logiques formelles conçues pour représenter les connaissances d'un domaine de manière structurée, tout en permettant de déduire automatiquement de nouvelles informations à partir des axiomes définis.

Ces raisonneurs utilisent des algorithmes complexes, tels que l'algorithme de tableau, pour inférer des faits, détecter des incohérences ou classer les concepts hiérarchiquement. Cependant, ils sont souvent perçus comme des « boîtes noires » difficiles à adapter, car leurs règles internes sont implicites et peu accessibles au développeur.

Par ailleurs, ces outils effectuent généralement un raisonnement à la volée (on-the-fly), ce qui signifie que les faits inférés ne sont pas ajoutés explicitement à l'ontologie : ils ne sont accessibles que durant l'exécution du raisonnement.

Cette approche présente certaines limites, la base de connaissance ne contient que les faits définis au préalable, sans mécanisme d'évolution ou d'intégration automatique de nouvelles informations. Ainsi, toute mise à jour ou enrichissement des connaissances requiert une intervention explicite. De plus, chaque interrogation implique un recalcul

des inférences, ce qui peut affecter la performance du système et la transparence du raisonnement dans certains contextes.

Dans ce stage, nous avons exploré une alternative à cette logique de raisonnement implicite, en traduisant une ontologie OWL en un ensemble de contraintes, et en exprimant le raisonnement d'inférence et de cohérence sous forme de règles déclaratives en CHR++.

L'objectif n'est pas de remplacer ni de rivaliser avec les moteurs de raisonnement existants, mais plutôt de proposer un cadre complémentaire, plus lisible et facilement extensible, où chaque règle peut être modifiée et où les contraintes peuvent être ajoutées de manière incrémentale. Ce choix ouvre la voie à une approche plus transparente et adaptable du raisonnement, facilitant l'expérimentation progressive sur des sous-ensembles d'OWL. Cette approche permet de saturer explicitement la base de faits, en y ajoutant directement les faits inférés produits. Les connaissances ainsi déduites deviennent alors persistantes dans le système et interrogeables de manière directe.

Le travail mené au cours de ce stage se structure autour de trois axes complémentaires :

1. Modélisation des ontologies OWL sous forme de contraintes CHR++

La première étape a consisté à réfléchir à la manière de représenter les concepts, les relations et les axiomes d'une ontologie OWL à l'aide de contraintes CHR++. L'objectif était de préserver la structure sémantique de l'ontologie tout en la rendant exploitable dans un système basé sur des règles déclaratives. Cette modélisation a nécessité l'élaboration de correspondances explicites entre les éléments du langage OWL (classes, sous-classes, propriétés, domaines, etc.) et des contraintes logiques manipulables dans CHR++.

2. Formalisation du raisonnement OWL avec des règles CHR++

Dans un second temps, le raisonnement a été formalisé sous forme de règles CHR++ exprimant explicitement les mécanismes d'inférence utilisés dans les raisonneurs OWL classiques. Cela inclut :

- l'inférence de nouveaux faits à partir des axiomes (tels que la transitivité, la spécialisation ou les contraintes de typage), en s'appuyant sur les spécifications du W3C relatives au comportement sémantique des constructions OWL, dans le but de saturer la base de faits.
- la détection d'incohérences dans les représentations.
- la mise en place de règles visant à rendre la base de connaissances interrogeable de façon structurée.

3. Implémentation d'un prototype de raisonneur basé sur CHR++

La dernière étape du stage a consisté à concrétiser l'approche proposée par le développement d'un prototype fonctionnel. Ce travail d'implémentation a nécessité :

- la définition d'un espace de contraintes regroupant l'ensemble des faits et des règles de raisonnement.
- l'analyse et le parsing d'ontologies OWL pour extraire les informations pertinentes à la génération des contraintes.
- l'implémentation concrète des règles modélisées dans le moteur CHR++ ;
- l'intégration d'un mécanisme de requête.

4 Représentation en CHR d'une ontologie OWL

Ayant défini le but du stage, décrivons la première brique concrète : la traduction des axiomes OWL en contraintes CHR++, puis l'élaboration des règles d'inférence correspondantes, que nous détaillerons dans cette partie.

Représentation des connaissances :

Les ontologies, formalisées en OWL, ont d'abord été analysées puis transformées en contraintes CHR++ : chaque axiome OWL est traduit en une ou plusieurs contraintes, tout en garantissant la validité et cohérence initiale. Cette étape a nécessité la définition d'une structure adaptée pour représenter les faits ; classes, relations, axiomes et assertions sont ainsi modélisés sous forme de contraintes déclaratives. Cette représentation constitue les fondations d'une base de connaissances structurée et formalisée, sur laquelle peuvent ensuite être appliquées des règles d'inférence.

Pour l'instant, seul un sous-ensemble d'OWL 2 est géré par notre système, mais celui-ci est conçu pour évoluer et prendre en charge progressivement l'intégralité du langage.

Mécanisme d'inférence :

Le raisonnement a ensuite été implémenté sous forme de règles CHR++ simulant le comportement attendu d'un raisonneur OWL, notamment dans le profil OWL 2 RL. Ces règles permettent, entre autres, d'inférer de nouveaux faits à partir d'axiomes comme la transitivité, la spécialisation, ou les contraintes de typage. Nous avons d'abord défini des règles de vérification qui assurent la cohérence et la validité des faits présents dans la base. En cas d'incohérence, le processus peut être interrompu afin d'éviter des inférences erronées, ensuite, des règles de nettoyage qui permettent de supprimer les doublons afin de maîtriser la croissance de la base de faits et d'optimiser les performances du système. Enfin, des règles d'inférence proprement dites enrichissent la base en ajoutant explicitement les faits déduits, rendant ainsi les connaissances persistantes et directement interrogeables.

4.1 Prise en charge des axiomes OWL

Les ontologies OWL 2 reposent sur plusieurs familles d'axiomes, chacune décrivant un aspect particulier du vocabulaire ou de la structure conceptuelle :

- **Structure de classes**
 - `SubClassOf(A,B)` : A est sous-classe de B.
 - `EquivalentClasses(A,B)` : A et B ont exactement les mêmes instances.
 - `DisjointClasses(A,A)` : ces classes n'ont aucun individu en commun.
- **Appartenance d'individus**
 - `ClassAssertion(a,C)` : l'individu a est instance de C.
 - `NegativeClassAssertion(a,C)` : l'individu a n'est pas instance de C.
- **Propriétés d'objets**
 - `ObjectPropertyAssertion(a,p,b)` : a est relié à b par la propriété p.
 - `SubObjectPropertyOf(p,q)`, `EquivalentObjectProperties(p,q)`,
 - `DisjointObjectProperties(p,q)` : hiérarchie et contraintes entre propriétés.
 - `ObjectPropertyDomain(p,D)`, `ObjectPropertyRange(p,R)` : ces axiomes précisent que pour toute assertion `p(a,b)`, l'individu *a* appartient à la classe *D*

(domaine de la propriété p) et l'individu b appartient à la classe R (codomaine de la propriété p).

- **Propriétés de données**
 - `DataPropertyAssertion(a,p,v)` : l'individu a possède, via p , la valeur littérale v de type T .
 - `SubDataPropertyOf`, `EquivalentDataProperties`, `DisjointDataProperties`, `DataPropertyDomain`, `DataPropertyRange` : fonctionnent de manière similaire aux propriétés d'objets, mais appliqués aux propriétés de données.
- **Restrictions de classes**
 - `ObjectSomeValuesFrom(p,C)`, `ObjectAllValuesFrom(p,C)`, `ObjectHasValue(p,b)` : définissent des classes par contraintes sur les relations d'objets.
 - `DataSomeValuesFrom`, `DataAllValuesFrom`, `DataHasValue` : contreparties pour les propriétés de données.
 - `ObjectMinCardinality`, `ObjectMaxCardinality`, `ObjectExactCardinality` (et équivalents `DataMin/Max/ExactCardinality`) : restrictions de cardinalité.
- **Caractéristiques de propriétés**
 - `FunctionalObjectProperty(p)`, `InverseFunctionalObjectProperty(p)`
 - `TransitiveObjectProperty(p)`, `SymmetricObjectProperty(p)`, `AsymmetricObjectProperty(p)`
 - `ReflexiveObjectProperty(p)`, `IrreflexiveObjectProperty(p)`

Les axiomes ont été analysés afin d'en extraire les éléments clés du triplet RDF (sujet, prédicat, objet) et de les modéliser sous forme de contraintes CHR++. Pour l'instant, l'attention se porte sur un sous-ensemble d'OWL 2, tandis que les axiomes non pris en charge sont identifiés et consignés en vue d'un traitement ultérieur.

Ces contraintes servent de base à la définition des règles d'inférence CHR++, qui permettent de reproduire la sémantique OWL, notamment la propagation des sous-classes, la validation de l'absence d'éléments communs entre classes disjointes et l'application des caractéristiques des propriétés.

Plusieurs modélisations ont été étudiées pour représenter un axiome. Pour l'illustrer, considérons l'opération « instance of », qui affirme qu'un individu x est membre d'une classe C . Trois schémas ont été envisagés :

1. *Niveau 0* : $C(x)$. Ce schéma entraîne une explosion du nombre de prédicats (un prédicat par classe C) et empêche la définition de règles génériques. Par exemple, pour propager une relation de sous-classe si $A \sqsubseteq B$ et $x \in A$, alors $x \in B$ il faudrait écrire autant de règles que de paires (A, B) :

```
// Pour A1 B1
A1(X) ==> B1(X).
// Pour A2 B2
A2(X) ==> B2(X).
```

2. *Niveau 1* : `owlClassAssertion(x,C)`. Schéma simple et homogène, directement inspiré de la syntaxe fonctionnelle d'OWL, où le même fait s'écrit `ClassAssertion(C,x)`.
3. *Niveau 2* : `owlTriple(x,ClassAssertion,C)`. Un triplet hétérogène permettant de représenter tous les axiomes, mais générant une surcharge (un espace de contraintes

unique) moins performant et complexifiant l'uniformisation des arguments (tous les sujets et objets devant partager le même type).

Le *niveau 1* (`ClassAssertion(x,C)`) a été retenu pour sa clarté, sa simplicité et sa correspondance directe avec la notation OWL. Nous l'étendrons de manière cohérente à l'ensemble des autres types d'axiomes, en conservant autant que possible les noms OWL pour nos contraintes.

4.2 Représentation d'inférences OWL sous forme de règles CHR++

Pour chaque axiome OWL, la formalisation en CHR++ suit deux étapes complémentaires :

1. **Définition de contraintes factuelles** Chaque axiome est d'abord traduit en une ou plusieurs contraintes élémentaires (p. ex. `owlSubclassOf(A,B)`, `owlClassAssertion(x,C)`, ...).
2. **Élaboration de règles d'inférence** Pour chaque type d'axiome, on définit une ou plusieurs règles CHR++ qui incarnent sa sémantique :
 - *Propagation* de faits (transitivité, spécialisation, domaines/codomains, etc.)
 - *Vérification* de contraintes (disjonctions, caractéristiques de propriétés, cardinalités)
 - *Nettoyage* des doublons et simplification des contraintes

Les contraintes constituent la base de connaissance initiale, et les règles d'inférence sont appliquées itérativement jusqu'à saturation de la base.

Chaque axiome OWL est ainsi traduit en une ou plusieurs règles CHR++ reproduisant précisément son comportement sémantique. Ci-dessous quelques exemples de ces traductions :

SubClassOf

- `owlClassAssertion(x,C)` signifie « l'individu `x` est instance de la classe `C` ».
- `owlSubclassOf(A,B)` représente l'axiome OWL `SubClassOf(A,B)`, c'est-à-dire $A \sqsubseteq B$.

OWL :

```
SubClassOf(:Vin :LiquidePotable)
```

Cet axiome signifie que la classe `Vin` est une sous-classe de `LiquidePotable` : tout individu appartenant à `Vin` appartient aussi à `LiquidePotable`. (c'est le même principe que l'héritage en programmation orienté objet)

CHR++ :

```
subclass @ owlSubclassOf(A, B), owlClassAssertion(x, A) ==>
  different(A,B) | owlClassAssertion(x, B);;
```

Cette règle traduit fidèlement la sémantique de l'axiome `SubClassOf`. Lorsqu'une classe `A` est définie comme sous-classe d'une classe `B` (`owlSubclassOf(A,B)`), et qu'un

individu x est instancié dans A ($\text{owlClassAssertion}(x,A)$), alors on déduit que x appartient également à B ($\text{owlClassAssertion}(x,B)$).

La condition de garde ($\text{different}(A,B)$) est ajoutée afin d'éviter l'application de la règle aux cas où une classe est sous-classe d'elle-même. En effet, dans OWL, toute classe est considérée comme une sous-classe d'elle-même, ce qui reflète le fonctionnement des langages ensemblistes

EquivalentClasses

OWL :

```
EquivalentClasses (:Person :Human)
```

Cet axiome indique que les classes **Person** et **Human** sont équivalentes, c'est-à-dire que tout individu appartenant à l'une appartient aussi à l'autre. On modélise cet axiome par la contrainte $\text{owlEquivalentClasses}(A,B)$, où A et B sont les deux classes équivalentes.

CHR++ :

```
equivalent @ owlEquivalentClasses(A, B), owlClassAssertion(x, A)
==> different(A,B) | owlClassAssertion(x, B);;
equivalentSym @ owlEquivalentClasses(A, B) ==> different(A,B) |
    owlEquivalentClasses(A, B) ;;
```

Ces règles traduisent la sémantique de l'axiome **EquivalentClasses**.

La règle « **equivalent** » stipule que si un individu appartient à la classe A , il appartient également à la classe équivalente B . Cependant, la condition de garde $\text{different}(A,B)$ est essentielle pour garantir que la règle ne s'applique pas lorsque A et B sont identiques. En effet, OWL considère que toute classe est équivalente à elle-même, ce qui, sans garde, pourrait entraîner des inférences redondantes ou inutiles.

La deuxième règle, « **equivalentSym** », assure la symétrie de l'axiome **EquivalentClasses**, en déclarant que si A est équivalent à B , alors B l'est aussi à A . Ici encore, la garde $\text{different}(A,B)$ évite la création de déclarations triviales comme $\text{owlEquivalentClasses}(A,A)$, qui n'apporteraient aucune information nouvelle et pourraient générer des boucles inutiles dans le processus d'inférence.

Grâce à ces précautions, le raisonnement reste structuré et efficace tout en préservant la cohérence des données dans l'ontologie.

ObjectPropertyDomain et ObjectPropertyRange

Souvent, le fait que deux individus soient liés par une certaine propriété permet de tirer d'autres conclusions à leur sujet. En particulier, il est possible d'en déduire leur appartenance à certaines classes. Par exemple, l'énoncé selon lequel B est l'épouse de A implique évidemment que B est une femme et A un homme. Ainsi, la déclaration selon laquelle deux individus sont reliés par une propriété donnée porte en elle une information implicite supplémentaire sur ces individus. Dans notre exemple, cette information additionnelle peut être exprimée par une appartenance à des classes. OWL offre un moyen d'exprimer cette correspondance.

OWL :

```
ObjectPropertyDomain(:hasWife :Man)
ObjectPropertyRange(:hasWife :Woman)
```

Cela signifie que si un individu x a la propriété `hasWife` avec un individu y , alors x est un `Man` et y une `Woman`.

Avant de présenter les règles, introduisons la signification des contraintes CHR++ utilisées :

- `owlObjectPropertyAssertion(x,p,y)` : l'individu x est lié à l'individu y par la propriété d'objet p .
- `owlObjectPropertyDomain(p,C)` : la propriété p a pour domaine la classe C , c'est-à-dire que tout sujet de p est instance de C .
- `owlObjectPropertyRange(p,C)` : la propriété p a pour codomaine la classe C , c'est-à-dire que tout objet de p est instance de C .
- `owlClassAssertion(x,C)` : l'individu x est instance de la classe C .

CHR++ :

```
domain @ owlObjectPropertyDomain(p, C), ObjectPropertyAssertion(x
, p, y) ==> owlClassAssertion(x, C);;
range @ owlObjectPropertyRange(p, C), ObjectPropertyAssertion(x,
p, y) ==> owlClassAssertion(y, C);;
```

Il est important de noter que le domaine et la codomaine d'une propriété n'interdisent pas les assertions, mais influencent leur interprétation. Autrement dit, une propriété peut être utilisée en dehors des classes spécifiées dans son domaine ou sa codomaine, sans que cela constitue une erreur formelle. Cependant, cette flexibilité peut affecter la cohérence du raisonnement et la structuration des connaissances au sein de l'ontologie.

Les règles formulées en CHR++ permettent de reproduire l'inférence des domaines et codomaines des propriétés OWL. Elles exploitent ces informations pour enrichir la structuration des connaissances en déduisant automatiquement que tout individu utilisé comme sujet ou objet d'une propriété est une instance de la classe définie comme domaine ou codomaine. Cette approche renforce la logique de classification et automatise l'organisation des données, limitant les déclarations explicites redondantes.

Propriétés de type de données (Datatype properties)

Dans de nombreux cas, il est nécessaire de décrire des individus à l'aide de **valeurs de données**. Pour représenter la date de naissance d'une personne par exemple, son âge, son adresse e-mail, etc.

Pour cela, OWL propose un autre type de propriétés, appelées *propriétés de type de données* (*Datatype properties*). Ces propriétés relient un individu à une valeur de donnée, contrairement aux *Object properties* qui relient deux individus. OWL permet ainsi l'utilisation de nombreux types de données standards, définis dans XML Schema, tels que `xsd:integer` ou `xsd:string` (voir schéma 5).

Voici un exemple utilisant une propriété de type de donnée, indiquant que l'âge de John est l'entier 51 .

OWL :

```
DataPropertyAssertion(:hasAge :John "51"^^xsd:integer)
```

CHR++ :

```
owlDataPropertyAssertion(John, hasAge, Value("51", integer)).
```

4.3 Vers une base de faits saturée

L'application répétée des règles CHR++ entraîne l'inférence de nouvelles contraintes et la saturation progressive de la base de faits. Ce processus permet d'explicitier des connaissances implicites qui ne sont pas formulées directement dans l'ontologie initiale.

4.4 Interrogation de la base

Grâce à la saturation préalable de la base, les connaissances peuvent être interrogées à l'aide de contraintes-but définies en CHR++. Lorsqu'une requête est formulée, comme `querySuperClassOf(x)`, le moteur CHR++ explore le magasin de contraintes et applique les règles associées à cette demande pour remonter l'arborescence des classes jusqu'à la racine et récupérer toutes les super-classes de la classe donnée `x`, pour ce faire nous avons défini les règles suivantes :

```
querySuperClassOf(X), owlSubclassOf(X, Y)==> different(X, Y) |
    res(Y), querySuperClassOf(Y);;
querySuperClassOf(X)==> res(X);;
res(X)==> afficher(X);;
```

Listing 1 – Règles pour `querySuperClassOf`

Dans ce cadre, les règles en CHR++ permettent de propager l'inférence en s'appuyant sur la relation `owlSubclassOf(X, Y)`, qui indique qu'une classe `X` est une sous-classe de `Y`. Lorsqu'une requête `querySuperClassOf(X)` est émise, si `X` est une sous-classe de `Y`, alors une nouvelle requête `querySuperClassOf(Y)` est déclenchée, garantissant l'exploration de la hiérarchie des classes et la récupération de toutes les super-classes pertinentes. Parallèlement, `res(Y)` est ajoutée pour signaler que `Y` doit être collectée comme super-classe de `X`. Afin d'éviter toute boucle indésirable, la condition `different(X, Y)` s'assure que `X` et `Y` sont bien distincts.

Une seconde règle veille à ce que `X` soit inclus comme résultat, en conformité avec la convention OWL, qui considère qu'une classe est toujours une sous-classe d'elle-même. Une fois les super-classes collectées via la contrainte `res(X)`, une dernière règle déclenche la fonction C++ `afficher(X)`, qui se charge d'afficher à l'écran la liste des classes trouvées.

Dans le même principe, nous avons défini d'autres requêtes-but qui se traduisent par des règles permettant de reproduire le comportement attendu lors de l'interrogation. Ces règles assurent l'exécution de la requête tout en stockant les résultats dans une contrainte unique `res`, qui centralise la ou les réponses obtenues.

Une fois les résultats enregistrés, CHR++ offre plusieurs possibilités de traitement. La contrainte `res` peut être exploitée pour l'affichage direct des réponses ou leur récupération en vue d'une utilisation ultérieure via une fonction C++.

La comparaison avec une approche classique met en évidence des différences majeures. Avec SPARQL combiné à un raisonneur OWL, l'inférence est généralement réalisée à la

volée, ce qui implique que le calcul des super-classes se produit au moment de la requête, sans stockage systématique des faits nouvellement inférés. Cette approche peut entraîner un coût élevé lorsque la même requête est répétée, car l'inférence doit être recalculée à chaque interrogation.

À l'opposé, CHR++ adopte une stratégie où la saturation est effectuée une seule fois, rendant les inférences persistantes. Les requêtes ultérieures peuvent ainsi exploiter directement le magasin saturé, offrant un retour immédiat et reproductible des résultats sans nécessiter de nouveaux calculs. Cette distinction, entre une exécution dynamique et une approche pré-calculée, influence la performance et la réactivité des requêtes. De plus, CHR++ sépare clairement la définition des connaissances via les contraintes et le mécanisme d'inférence structuré par les règles, ce qui améliore la transparence et la modularité du raisonnement.

5 Tests et résultats

Pour mettre en œuvre ce qui a été décrit dans la section précédente, nous avons conçu et expérimenté un prototype. Cette section décrit en détail les étapes d'implémentation, l'évaluation du prototype et son application à travers une série de tests, mettant en évidence les inférences obtenues. La figure 1 montre l'architecture simplifiée de notre système.

La conception du prototype s'est déroulée en plusieurs étapes. Tout d'abord, un *espace CHR++* (`auto space = OWL2::create();`) a été créé pour stocker les contraintes. Ensuite, un ensemble de contraintes a été déclaré dans le bloc `<chr_constraint>`, correspondant aux axiomes OWL, incluant classes, propriétés et assertions.

L'instanciation de l'espace a permis l'appel au parser *COWL*⁷, qui parcourt l'ontologie d'entrée au format fonctionnel OWL 2 (préalablement convertie depuis RDF/XML ou Turtle via *ROBOT CLI*⁸). À chaque axiome, le parser extrait les trois composantes du triplet RDF (*sujet*, *prédicat*, *objet*) et instancie immédiatement la contrainte CHR++ correspondante, évitant ainsi toute structure intermédiaire pour réduire la surcharge mémoire.

La phase d'inférence suit immédiatement cette instanciation. Les règles CHR++ (disponibles en ligne à l'adresse suivante : `owl.chrpp`) s'appliquent jusqu'à la *saturation* du magasin de contraintes, garantissant une structuration cohérente des connaissances. En cas d'incohérence, les règles de type `failure()` interrompent l'exécution pour éviter toute dérive.

Enfin, l'affichage des résultats repose sur les contraintes inférées, directement intégrées au store. Les règles de requête (`querySuperClassOf`, `queryInstancesOf`, etc.) génèrent des contraintes `res(...)` traitées immédiatement par la fonction C++ `afficher(...)`. Les résultats peuvent aussi être récupérés sous forme de liste ou de vecteur pour un post-traitement.

7. COWL est une bibliothèque C++ légère pour charger et parcourir des ontologies OWL 2 en syntaxe fonctionnelle.

8. ROBOT CLI est un outil en ligne de commande pour manipuler, valider, transformer et exporter des ontologies OWL.

5.1 Gestion avancée des *Datatype properties*

Initialement, les contraintes CHR++ utilisaient uniquement des arguments de type `std::string`, suffisants pour les axiomes de structure ou d'appartenance. Toutefois, pour les *Datatype properties*, associant un individu à une valeur littérale typée (âge, date de naissance, étiquette textuelle, etc.), il a été nécessaire d'enrichir la signature de la contrainte afin d'inclure non seulement la chaîne brute mais aussi son type.

Approche orientée polymorphisme La solution repose sur une conception basée sur le polymorphisme. Une classe abstraite `AnyType` a été définie pour représenter un *schéma de type* (exemple : `xsd:integer`, `xsd:string`). Chaque type XSD est encapsulé dans une sous-classe concrète (`XsdInteger`, `XsdString`, etc.), contenant notamment une fonction `verify(value)` qui valide, via une expression régulière, l'adéquation entre la chaîne fournie et le schéma attendu.

Grâce à cette hiérarchie, le *polymorphisme* permet d'utiliser une variable pointeur de type `std::unique_ptr<AnyType>` pour référencer n'importe quel schéma XSD, assurant ainsi la vérification automatique lors de l'injection des données, sans nécessiter de logique dédiée par type dans les règles CHR++.

Nouvelle signature de contrainte Avec cette amélioration, la contrainte CHR++ pour une assertion de donnée adopte la signature suivante :

```
owlDataPropertyAssertion(+std::string, +std::string, +Value)
```

où `Value` est une structure C++ regroupant la *chaîne brute* du littéral ("51" par exemple) et un pointeur `std::unique_ptr<AnyType>` vers l'objet de schéma (`XsdInteger`).

Avantages du polymorphisme en CHR++ Cette approche évite la duplication des règles de validation de chaque type dans CHR++, concentrant toute la complexité dans les classes C++. Elle marque également la première utilisation du polymorphisme dans notre prototype CHR++, permettant une gestion flexible et évolutive des types de données. De plus, le système reste ouvert : une nouvelle sous-classe `AnyType`, comme `XsdAdultAge` pour restreindre les entiers aux valeurs supérieures à 18, peut être ajoutée sans modifier le code des règles CHR++.

Cette structuration garantit un raisonnement cohérent et une gestion efficace des inférences tout en facilitant les futures extensions du modèle.

5.2 Exemples de requêtes

Toutes les contraintes-but sont intégrées dans des règles CHR++ qui permettent de récupérer les résultats dans une contrainte unique `res`. Cette contrainte centralise les réponses obtenues et peut ensuite être traitée pour affichage ou récupération via une fonction C++.

Les requêtes implémentées sont les suivantes :

- **Classification** La contrainte `querySuperClassOf("MasterStudent")` permet de récupérer toutes les super-classes de "MasterStudent", qu'elles soient directes ou indirectes, jusqu'à la classe racine `owl:Thing`. Cette requête repose sur l'exploitation de la propriété `rdfs:subClassOf`, permettant une navigation ascendante dans la

hiérarchie des classes. Son implémentation est récursive, puisqu'à chaque étape, `querySuperClassOf(X)` déclenche une nouvelle requête `querySuperClassOf(Y)`, où `Y` est la super-classe de `X`, garantissant ainsi une exploration complète de la hiérarchie.

- **Instances d'une classe** La contrainte `queryInstancesOf("C")` extrait directement l'ensemble des individus instanciés dans la classe `C`. Lorsqu'elle est déclenchée, elle génère des contraintes `res(x)` pour chaque individu `x` identifié comme instance de `C`.
- **Sous-classes d'une classe** La contrainte `querySubClassesOf("Employee")` fonctionne de manière similaire à `querySuperClassOf`, mais dans la direction opposée. Au lieu de rechercher les super-classes d'une classe donnée, elle explore les sous-classes en s'appuyant sur la propriété `owlSubclassOf(x, y)`, identifiant tous les `x` définis comme sous-classes de `y`.

Chaque requête est optimisée pour la gestion des connaissances en CHR++, permettant soit une exploration récursive, soit une récupération directe selon la nature de l'interrogation.

5.3 Test du prototype

Pour mettre en lumière les capacités de notre prototype, nous avons utilisé l'ontologie suivante :

```
Ontology(:Test
  Declaration(Class(:Person))
  Declaration(Class(:Student))
  Declaration(Class(:MasterStudent))
  Declaration(Class(:Employee))
  SubClassOf(:Student :Person)
  SubClassOf(:MasterStudent :Student)
  SubClassOf(:Employee :Person)
)
```

Listing 2 – Ontologie de test

1. Contraintes initiales Après parsing, l'espace CHR++ contient les contraintes suivantes (avant inférence) :

- `owlClass("Person"), owlClass("Student"), owlClass("MasterStudent"), owlClass("Employee")`
- `owlSubclassOf("Student", "Person"), owlSubclassOf("MasterStudent", "Student"), owlSubclassOf("Employee", "Person")`

Soit **7** contraintes dans le store.

2. Saturation et inférences En appliquant nos règles CHR++ jusqu'à saturation, le moteur a ajouté :

1. *Réflexivité de la sous-classe* Par règle $\forall C, C \sqsubseteq C$.

`owlSubclassOf("Person", "Person"), ..., ("Employee", "Employee")`

4 nouvelles contraintes (une par classe).

2. *Inférences de transitivité (spécialisation)* Si $A \sqsubseteq B$ et $B \sqsubseteq C$, alors $A \sqsubseteq C$.

```
owlSubclassOf("MasterStudent", "Person")
```

1 nouvelle contrainte.

3. *Inférences OWL standard (racine owl:Thing)* Par convention OWL, chaque classe est sous-classe de `owl:Thing`.

```
owlSubclassOf("Person", "owl:Thing"), ..., ("Employee", "owl:Thing")
```

4 nouvelles contraintes (une par classe).

4. *Vérification et suppression des doublons* Aucune incohérence détectée. Grâce aux règles de nettoyage, le magasin ne contient aucun doublon.

Au total, le store contient 16 contraintes après saturation, toutes distinctes.

3. Requête des super-classes

Nous lançons ensuite la contrainte-but :

```
space->querySuperClassOf("MasterStudent");
```

qui extrait récursivement toutes les classes Y telles que $\text{MasterStudent} \sqsubseteq Y$.

4. Résultat affiché

La liste $\{\text{"Student"}, \text{"Person"}, \text{"owl:Thing"}\}$ est imprimée dans l'ordre de découverte.

Cet exemple montre clairement que, dès la première exécution :

- la *transitivité* des sous-classes est correctement appliquée ;
- la relation universelle avec `owl:Thing` est automatiquement ajoutée ;
- le magasin de contraintes passe de 7 à 16 entrées, toutes directement interrogeables.

5.4 Bilan des résultats

- **Exactitude** : tous les axiomes testés (sous-classes, équivalences, domaines/codomains, propriétés de données, caractéristiques de propriétés) ont donné lieu aux inférences attendues, validées sur des petits jeux de test.
- **Performance** : pour des ontologies de quelques centaines d'axiomes, la saturation se fait en quelques millisecondes. La suppression des doublons et la détection précoce des incohérences garantissent que le store ne gonfle pas inutilement.
- **Extensibilité** : grâce à la journalisation des axiomes non pris en charge, il est facile d'ajouter de nouveaux types de contraintes ou de règles CHR++ à mesure que le sous-ensemble d'OWL 2 s'étend.

En somme, ce prototype prouve la faisabilité d'un raisonneur OWL 2 basé sur CHR++, offrant transparence, modularité et performances prometteuses pour des applications nécessitant une inférence persistante et interrogeable, tout en étant adapté à toutes les syntaxes OWL (fonctionnelle, RDF/XML, Turtle).

6 Conclusion

Ce rapport se termine par une analyse des résultats obtenus et une réflexion sur les perspectives et limites de l'approche CHR++ appliquée au raisonnement OWL 2. Le prototype développé a démontré la faisabilité de cette approche et posé les bases d'un système extensible.

Perspectives

L'implémentation actuelle prend en charge un sous-ensemble représentatif d'OWL 2 et a été conçue pour faciliter son extension. Les axiomes non pris en charge sont automatiquement journalisés, notamment ceux liés aux relations avancées entre classes, aux restrictions sur les propriétés, aux contraintes de cardinalité ou à l'énumération des individus. De même, l'utilisation avancée des propriétés, les chaînes de propriétés, les clés et les types de données complexes restent à intégrer. Cette démarche progressive permet d'assurer une évolution maîtrisée du raisonneur, et le travail se poursuivra au-delà de la période de stage afin d'élargir la couverture des constructeurs OWL.

À ce jour, seules quelques contraintes-but ont été implémentées. L'enrichissement des capacités d'interrogation passera par l'ajout de nouvelles contraintes correspondant aux constructions OWL prises en charge dans le prototype.

Enfin, l'interrogation de la base se limite actuellement à une seule variable à la fois. Il serait pertinent de permettre la sélection simultanée de plusieurs variables, à l'image du fonctionnement de SPARQL, où une requête peut récupérer plusieurs éléments en une seule interrogation. Par exemple, en SPARQL, la requête suivante :

```
SELECT ?x ?y WHERE { ?x :hasFriend ?y . }
```

permet d'obtenir toutes les paires (x,y). Un équivalent en CHR++ pourrait produire une liste de couples via une contrainte `res(x,y)`.

Enseignements personnels

Ce stage a été une expérience enrichissante, tant sur le plan technique que méthodologique. Il m'a offert l'opportunité d'explorer en profondeur des concepts complexes liés aux ontologies et au raisonnement automatisé, tout en développant des compétences essentielles en programmation par contraintes et en modélisation. Au-delà des aspects purement techniques, ce travail m'a aussi permis d'adopter une démarche rigoureuse et analytique, indispensable à la recherche appliquée.

Voici les principaux apprentissages que j'ai pu en tirer :

- J'ai approfondi ma compréhension des ontologies et élargi mes connaissances sur OWL 2.
- J'ai découvert le langage CHR++ et assimilé les principes de la programmation par contraintes, notamment la définition de gardes, la gestion des doublons et la stratégie de saturation d'une base de faits.
- J'ai développé une méthodologie pour traduire une spécification formelle (OWL) en un ensemble cohérent de contraintes et de règles.

- J’ai expérimenté la conception de requêtes par contraintes et analysé les différences fondamentales entre un raisonnement persistant (CHR++) et un raisonnement *à la volée* *z* (raisonneurs OWL classiques).
- J’ai appris à documenter et structurer un travail de recherche appliquée, en justifiant mes choix de modélisation et en identifiant les limites et perspectives du raisonneur développé.
- J’ai découvert la réalité d’un projet de recherche et pris plaisir à mener une réflexion approfondie sur un sujet, ce qui m’encourage à poursuivre dans cette voie.
- J’ai acquis des compétences en rédaction scientifique grâce à l’apprentissage et à l’utilisation de LaTeX pour la mise en forme de mon rapport.

Cette immersion m’a conforté dans mon intérêt pour la recherche et la représentation des connaissances, ouvrant ainsi de nouvelles perspectives pour mes travaux futurs.

7 Bibliographie / Webographie

Références

- [1] V. Barichard, *CHR++ : An efficient CHR system in C++ with don't know non-determinism*, Expert Systems With Applications, vol. 238, 2024. <https://doi.org/10.1016/j.eswa.2023.121810>
- [2] T. Berners-Lee, J. Hendler, O. Lassila, *The Semantic Web*, Scientific American, May 2001.
- [3] W3C, *RDF Schema 1.1*, <https://www.w3.org/TR/rdf-schema/>, 2014.
- [4] W3C, *OWL 2 Web Ontology Language Primer (Second Edition)*, <https://www.w3.org/TR/owl2-primer/>, 2012.
- [5] W3C, *XML Schema Part 2 : Datatypes Second Edition*, <https://www.w3.org/TR/xmlschema-2/>, 2012.
- [6] W3C, *OWL 2 Profiles : EL, QL, and RL*, <https://www.w3.org/TR/owl2-profiles/>, 2012.
- [7] F. Guidi, R. M. Martin, S. Montanelli, P. Ciancarini, *COWL : A C++ library for OWL 2 ontologies*, Proceedings of the 12th International Workshop on OWL : Experiences and Directions (OWLED), 2015.
- [8] T. Frühwirth, *Welcome to Constraint Handling Rules*, in : T. Frühwirth, T. Schrijvers (Eds.), *Constraint Handling Rules*, LNAI 5388, Springer, 2008. :contentReference[oaicite :0]index=0
- [9] T. Frühwirth, *Constraint Handling Rules What Else ?*, arXiv :1701.02668 [cs.PL], 2017. :contentReference[oaicite :1]index=1
- [10] K. Dentler, R. Cornet, A. ten Teije, N. de Keizer, *Comparison of Reasoners for Large Ontologies in the OWL 2 EL Profile*, Semantic Web, vol. 1, 2011, pp. 15.
- [11] M. Krötzsch, *OWL 2 Profiles : An Introduction to Lightweight Ontology Languages*, in : *OWL 2 Profiles : EL, QL, and RL*, W3C Working Group Note, 2012.
- [12] Document de travail, *Comparaison et définition simplifiée des profils OWL 2 (ELtiny, RLtiny, QLtiny)*, 2011.
- [13] I. Bilenchi et al., *COWL : Pushing OWL 2 over the Edge*, Internet of Things, vol. 29, 2025, article 101439.
- [14] Université Grenoble Alpes, *Web sémantique et Web de données*, FUN-MOOC, <https://www.fun-mooc.fr/fr/cours/web-semantique-et-web-de-donnees/>, 2024.