N° d'ordre : ……

UNIVERSITE DE M'SILA

FACULTE DES MATHEMATIQUES ET DE L'INFORMATIQUE

Département d'Informatique

MEMOIRE

Présenté pour l'obtention du diplôme de Master

Domaine : Mathématiques et Informatique

Filière : Informatique

Spécialité : Système d'information avancée

Par :

HAMDANI Khadidja

SUJET

TRANSFORMATION AUTOMATIQUE DES ONTOLOGIES DE OWL VERS PROLOG

Soutenu publiquement le : 28/06/2012 devant le jury composé de :

Mr. S. Kadri	**Université de M'sila**	**Président**
Dr. M. BOURAHLA	**Université de M'sila**	**Rapporteur**
Mr. M.Boubakir	**Université de M'sila**	**Examinateur**

Promotion : 2011 /20 12


بسم الله الرحمن الرحيم

Remerciements

Je remercie le bon Dieu qui m' a orienté et éclairé le bon chemin de la recherche et du savoir.

- *Je tiens à exprimer mon reconnaissance à **Dr. BOURAHLA Mustapha**, je le remercie pour son implication, ses enseignements et ses conseils.*
- *A tout les enseignants et les administratifs de l'institut de l'informatique.*
- *Je remercie tout ma famille, qui m'a toujours soutenu dans mes études.*

Khadidja HAMDANI

Dédicaces

A mes parents, mon fiancé Tayeb et sa famille

A ma sœur JIJI, Mes frères abd el-kader et abd el-djalil

A mes grands-parents

A mes oncles, mes antes

A mes cousins et mes cousines

HAMDANI Khadidja.

Table des matières :

Table des Figures :

Table de tableaux :

Introduction:

Introduction générale

Le Web tel que nous le connaissons aujourd'hui est encore conforme à la vision qu'en avait Tim Berners-Lee il y a quinze ans : il s'agit d'un Web de documents. Ceux-ci sont écrits en HTML, identifiés de manière unique par des URLs (Uniform Resource Locator) et reliés entre eux par des liens hypertextes. L'utilisateur surfe manuellement de page en page et peut depuis quelques années interagir avec le Web grâce aux technologies du Web 2.0 .

Les technologies du Web sémantique complètent le Web actuel avec des outils sémantiques. Il ne s'agit donc pas de créer un nouveau Web ou un Web séparé de l'existant : ce Web de données repose entièrement sur les technologies et concepts qui ont fait le succès du Web tel que nous le connaissons aujourd'hui.

Les ontologies sont alors centrales pour le Web sémantique qui, d'une part, cherche à s'appuyer sur des modélisations de ressources du Web à partir de représentations conceptuelles des domaines concernés et, d'autre part, a pour objectif de permettre à des programmes de faire des inférences dessus.

Ces dernières années le développement des ontologies - spécifications formelles explicites de termes d'un domaine et de relations entre elles (Gruber 1993) – a quitté les laboratoires d'Intelligence Artificielle pour gagner les postes informatiques des experts de domaines. Les ontologies sont devenues très courantes dans le World-Wide Web. Le champ des ontologies dans le web varie de taxonomies larges servant à catégoriser les sites Web (tels que dans Yahoo!) aux catégorisations de produits destinés à la vente et de leurs caractéristiques (tel que dans Amazon.com).

Pour représenter les ontologies le W3C propose un standard qui est le OWL (Ontology Web Language).

OWL est tout comme RDF un langage XML profitant de l'universalité syntaxique du XML. OWL offre la possibilité d'écrire des ontologies Web.

Le langage prolog un langage à part, différant radicalement des langages impératifs (type C++, Java) et des langages fonctionnels (type Lisp). Il s'agit d'un langage *déclaratif*. Idéalement, le programmeur se contente de donner des connaissances à la machines, et celle-

ci fait le reste ! Prolog est une occasion idéale pour découvrir et maîtriser certaines notions fondamentales de la science informatique, parmi lesquelles :

- la récursivité
- la déclarativité
- l'unification
- le *backtracking*

Il existe plusieurs travaux dans ce domaine. Comme transformation des ontologies vers UML, vérification des ontologies avec prolog, notre projet se situé sur le profit de l'automatisation de la transformation avec xslt pour transformer les ontologies créés avec protégé 4.2_alpha. Le but de notre travail est de faciliter la transformation des ontologies pour faciliter leur vérification avec prolog.

Donc l'objectif de notre travail est la proposition d'un programme de transformation automatique avec xslt. ce programme sera testé sur un exemple.

Ce mémoire est divisé en quatre chapitres :

❖ Le premier chapitre : web sémantique et ontologie

Nous allons faire une partie de l'état de l'art de web sémantique et des ontologies, exactement, les notions de base, des langages de représentation des ontologies, et leurs éditeurs.

❖ Le deuxième chapitre : la programmation logique

Nous allons faire des petites définitions sur la programmation logique (prolog) et nous avons expliqué comment pouvons-nous de créer des bases de données et des bases des connaissances.

❖ Le troisième chapitre : eXtensible Stylesheet Language

Nous définissons le langage xsl et nous avons parlé sur les fonctions de xslt et xpath

❖ Le quatrième chapitre : implémentation

Permet d'envisager la transformation des ontologies avec xslt. Pour réaliser l'objectif à partir des étapes suivantes :

1. Représentation de notre ontologie avec UML et avec protégé
2. Transformer cette ontologie avec notre programme xsl
3. Representer l'interface global de notre projet.

Enfin, nous concluons ce mémoire par une conclusion générale et présentation de quelques perspectives.

Chapitre1:

web sémantique et ontologie

1. Web Sémantique :

1.1. Définition :

Le web sémantique est un web « plus intelligent » où les documents sont compréhensibles par les machines et peuvent donc être traités automatiquement de manière beaucoup plus pertinente. Plus précisément, c'est une évolution progressive du Web portant à la fois sur la production du contenu et sur son analyse. [1]

1.2. Origine:

L'idée d'un web sémantique est ancienne et remonte au philosophe Leibniz, dont le rêve était de concevoir une *lingua characteristica* (une langue dans laquelle toutes les connaissances peuvent être exprimées sans aucune équivoque) et un *calculus ratiocinator* (un calcul sur le raisonnement, c'est-à-dire principalement un moteur d'inférence sémantique) afin d'améliorer la communication et de résoudre plus facilement les désaccords. Cette idée a été réintroduire il y a quinze ans par Tim Berners-Lee, l'inventeur du World Wide Web, et développée dans le cadre du World Wide Web Consortium(W3C). Alors que le Web s'est développé comme un médium de documents reliés entre eux par de l'hypertexte et destinés aux êtres humains, non compris par la machine et dont les possibilités de traitement de l'information sont plutôt limitées, le Web sémantique est associé à deux éléments : le premier porte sur l'intégration et la combinaison de données issues de diverses sources; le second a trait à un langage permettant d'enregistrer la manière dont les données portent sur les objets du monde réel. Cela signifie qu'un individu est en mesure de démarrer une recherche dans une base de données et d'aller ensuite dans un ensemble non limité de bases de données reliées entre elles par leur intérêt pour le même objet.

Des programmes informatiques « pensants » :

Comme les individus souhaitent donner du sens aux résultats de recherche, l'émergence du Web sémantique vise à permettre l'interaction des ordinateurs pour réunir des données satisfaisant des requêtes très spécifiques. A titre illustratif, une requête du type « quel est le meilleur restaurant italien de Paris ? » sur le Web aujourd'hui conduirait à des centaines d'informations –voire davantage-, contraignant le demandeur à faire le tri dans ce qui est collecté pour en extraire le matériau souhaité, sans que le résultat soit pour autant intéressant. Ce médiocre résultat est dû à l'inaptitude des moteurs de recherche à comprendre le langage naturel. Dès lors, le défi du Web sémantique consiste à extraire des données de milliers de

bases de données et à les relier de façon significative afin de faciliter la recherche et l'organisation de connaissances sur le réseau.

1.3.Objectifs du web sémantique :

Deux objectifs majeurs sont recherchés par le Web sémantique: partager et enrichir tous les types de données de manière structurée en leur donnant du sens afin d'améliorer la pertinence des contenus ; constituer une base de données exhaustives standardisée à l'échelle du Web.

L'impératif de la structuration du contenu :

Permettant aux machines de comprendre les documents et l'information –et non pas des discours et des écrits humains-, le web sémantique est considéré comme susceptible de structurer le contenu des pages web en leur donnant du sens, via la création d'un environnement dans lequel les logiciels, se déplaçant de page en page, pourtant aisément entreprendre des taches complexes pour les utilisateurs, sans recourir obligatoirement à l'intelligence artificielle ou à l'utilisation d'algorithmes de programmation puissante. En effet, comme les informations sont supposées posséder un sens bien défini, elles peuvent être traitées et comprises par les ordinateurs. Il s'agit dès lors de trouver un dispositif qui exprime de façon concomitante des données et des règles de raisonnement sur les données. Ce dispositif passe par des métadonnées, des ontologies et des méthodes de raisonnement.

Les métadonnées : sont des informations associées à une ressource du web, permettant d'en favoriser l'utilisation par un agent humain, du fait de son exploitation par un agent logiciel ; jouent un rôle central dans le web sémantique et vont au-delà du catalogage et de l'indexation.

Les fonctions des métadonnées dans le web sémantique dépassent les dimensions signalétiques et thématiques qu'on leur connaissait jusqu'à présent. Selon le contexte et les applications, elles servent aussi de support à la gestion des droits, au recueil d'annotations diverses telles que commentaires et recommandations, à la qualification des hyperliens, à la définition de parcours de lecture ou d'assemblage de documents à la carte, etc. [2]

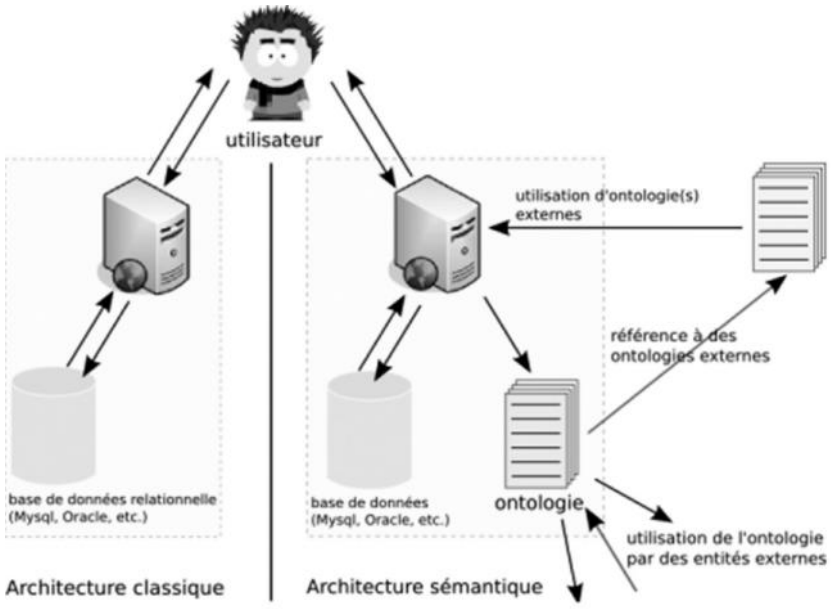


Fig1.3 la déférence entre le web classique et sémantique

*1.4.**Architecture du Web sémantique :***

La vision courante du Web sémantique peut être représentée dans une architecture en plusieurs couches différentes.

Les couches les plus bases assurent l'interopérabilité syntaxique : la notion d'URI fournit un adressage standard universel permettant d'identifier les ressources tandis que Unicode est un encodage textuel universel pour échanger des symboles. Rappelons que l'URL, comme l'URI, est une chaîne courte de caractères qui est aussi utilisée pour identifier des ressources (physiques) par leur localisation.

XML fournit une syntaxe pour décrire la structure du document, créer et manipuler des instances des documents. Il utilise l'espace de nommage (namespace) afin d'identifier les noms des balises (tags) utilisées dans les documents XML. Le schéma XML permet de définir les vocabulaires pour des documents XML valides. Cependant, XML n'impose aucune contrainte sémantique à la signification de ces documents, l'interopérabilité syntaxique n'est pas suffisante pour qu'un logiciel puisse "comprendre" le contenu des données et les manipuler d'une manière significative.

Les couches RDF M&S (RDF Model and Syntax) et RDF Schéma sont considérées comme les premières fondations de l'interopérabilité sémantique. Elles permettent de décrire les taxonomies des concepts et des propriétés (avec leurs signatures). RDF fournit un moyen d'insérer de la sémantique dans un document, l'information est conservée principalement sous

forme de déclarations RDF. Le schéma RDFS décrit les hiérarchies des concepts et des relations entre les concepts, les propriétés et les restrictions domaine/Co-domaine pour les propriétés. Les trois langages XML, RDF et RDFS seront présentés plus en détail dans la partie suivante.

La couche suivante Ontologie décrit des sources d'information hétérogènes, distribuées et semi-structurées en définissant le consensus du domaine commun et partagé par plusieurs personnes et communautés. Les ontologies aident la machine et l'humain à communiquer avec concision en utilisant l'échange de sémantique plutôt que de syntaxe seulement.

Les règles sont aussi un élément clé de la vision du Web sémantique, la couche Règles offre la possibilité et les moyens de l'intégration, de la dérivation, et de la transformation de données provenant de sources multiples, etc.

La couche Logique se trouve au-dessus de la couche Ontologie. Certains considèrent ces deux couches comme étant au même niveau, comme des ontologies basées sur la logique et permettant des axiomes logiques. En appliquant la déduction logique, on peut inférer de nouvelles connaissances à partir d'une information explicitement représentée.

Les couches Preuve (Proof) et Confiance (Trust) sont les couches restantes qui fournissent la capacité de vérification des déclarations effectuées dans le Web Sémantique. On s'oriente vers un environnement du Web sémantique fiable et sécurisé dans lequel nous pouvons effectuer des tâches complexes en sûreté.

D'autre part, la provenance des connaissances, des données, des ontologies ou des déductions est authentifiée et assurée par des signatures numériques, dans le cas où la sécurité est importante ou le secret nécessaire, le chiffrement est utilisé.[3]

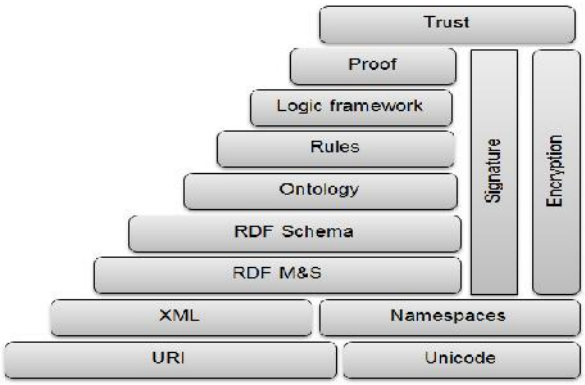


Figure 1.4 *Architecture du Web sémantique*

2. ontologie :

2.1. Définitions :

Les ontologies, à l'origine d'une branche de la philosophie qui s'intéresse à la nature et l'organisation de la réalité, correspondent à ce qu'Aristote appelait la Philosophie première, c'est-à-dire la partie de la métaphysique qui s'intéresse à l'être en tant qu'être, par opposition aux philosophies secondes qui s'intéressent à l'étude des manifestations de l'être. En informatique, la littérature fournit un tas de définitions du mot ontologie. Ces définitions, dans leur diversité, offrent des points de vues à la fois différents et complémentaires. Voici quelques définitions :

En 1991 Neches et ses collègues[4] ont été les premiers à en proposer une définition :
« Une ontologie définit les termes et les relations de base du vocabulaire d'un domaine ainsi que les règles qui indiquent comment combiner les termes et les relations de façon à pouvoir étendre le vocabulaire ». Cette définition descriptive donne un premier aperçu sur la manière de construire une ontologie, à savoir l'identification des termes de bases d'un domaine et les relations entre ces termes ainsi que les règles pouvant s'appliquer sur ces derniers.
En 1993, Gruber[5] a proposé une définition à cette notion qui est la plus célèbre couramment citée dans la littérature :

« Une ontologie est une spécification explicite d'une conceptualisation ».

En 1997, Borst [6]a modifié légèrement la définition de Gruber en citant qu'une ontologie est définie comme étant :

« Une ontologie est une spécification formelle d'une conceptualisation partagée ».

En 1998, Studer[7] et ses collègues ont rassemblé ces deux définitions (celles de Gruber et Borst) dans une seule qui est :
« Une ontologie est une spécification formelle et explicite d'une conceptualisation partagée ».
Ils l'expliquent comme suit :

- ***Spécification explicite*** signifie que les concepts, les propriétés, les relations, les fonctions, les restrictions et les axiomes de l'ontologie sont définis de façon déclarative ;
- ***Formelle*** réfère au fait qu'une ontologie doit être traduite dans un langage interprétable par une machine (*machine-readable*) ;

- ***Conceptualisation*** réfère à un modèle abstrait d'un phénomène du monde en identifiant les concepts appropriés à ce domaine ;
- ***Partagé*** réfère au fait qu'une ontologie capture la connaissance consensuelle c'est-à-dire non réservée à quelque individus, mais partagée par un groupe ou une communauté.

En 1995, Guarino et Giaretti [8] proposent leur définition :

« Une ontologie est une théorie logique proposant une vue explicite et partielle d'une conceptualisation ».

En 2000 Aussenac-Gilles[9] et ses collègues énonce que :

« Une ontologie organise dans un réseau des concepts représentant un domaine. Son contenu et son degré de formalisation sont choisis en fonction d'une application ».

Il existe un autre groupe de définitions basées sur le processus de construction d'une ontologie, alors que celles précitées sont indépendamment de ce dernier. Ce type de définition souligne la relation entre une ontologie et une base de connaissances.

En 1996 Bernaras et ses collègues[10] au sein du projet KAKTUS propose :

« Une ontologie fournit le moyen pour décrire d'une manière explicite la conceptualisation des connaissances représentées dans les bases de connaissances ».

Dans le même ordre d'idée, en 1997 Swartout et ses collègues [11]au sein du projet SENSUS

« Une ontologie est un ensemble de termes hiérarchiquement structurés, conçu afin de décrire un domaine qui peut être utilisé comme un squelette de base pour les bases de connaissances ».

Selon Gomez et ses collègues[12] « les ontologies visent à capturer les connaissances ***consensuelles*** de façon générique ainsi que la façon de leur réutilisation et leur partage en travers des applications et des groupes de personnes ».

2.2.Historique :

"Ontologie" est un mot que l'informatique a emprunté à la philosophie au début des années 90.

Depuis la Création du **Web en 1990,** par Tim Berners Lee, Il était exclusivement destiné à partager des informations sous forme de pages html, affichables par un logiciel «navigateur web», et généralement destinées à être lues par un utilisateur humain.

Ainsi L'arrivée de **XML en 1998**, a donné un cadre à la structuration des connaissances, rendant ainsi possible la création de nouveaux langages web destinés non plus à un rendu graphique à l'écran pour un utilisateur humain, mais à un réel partage et à une manipulation des savoirs.

En 2004 W3C publie RDF et OWL deux technologies clés du Web sémantique : Ces deux technologies RDF et OWL, fournissent un cadre de travail pour la gestion des ressources, l'intégration, le partage et la réutilisation des données sur le Web.

2.3. *Pourquoi vouloir développer une ontologie?*

Certaines de ces raisons sont :

- partager connaissance commune de la structure d'information entre personnes ou agents logiciels
- permettre la réutilisation de connaissances du domaine
- faire domaine hypothèses explicites
- séparer connaissance du domaine des connaissances opérationnelles
- analyser connaissances du domaine [13]

2.4. Composantes de l'ontologie :

Les ontologies rassemblent les connaissances propres à un domaine particulier. Ces connaissances sont formalisées en mettant en jeu les composants suivants: concepts ; relations ; axiomes et instances.

2.4.1. Concepts :

Un concept peut représenter un objet, une idée, ou bien une notion abstraite. Ils sont appelés aussi classes de l'ontologie dans certain travaux. Un concept peut être divisé en trois parties : ***un terme (***ou plusieurs***)***, ***une notion*** et un ***ensemble d'objets***.

- Le terme (ou bien label) d'un concept est l'expression linguistique utilisée couramment pour y faire référence.

❖ La notion désigne ce qui est appelé, au sens de la représentation des connaissances, ***l'intention*** du concept. Elle contient sa sémantique qui est définie à l'aide de propriétés (relations et attributs), de règles et de contraintes.

L'ensemble d'objets définis par le concept forme ce qui est appelé ***l'extension***

2.4.2. Relations :

Les relations sémantiques unissent les concepts du segment analysé de la réalité entre eux et traduisent les associations existantes.

Les ontologies généralement contiennent que des relations binaires. Le premier argument d'une relation binaire est dit domaine, alors que le deuxième argument est dit co-domaine. Cela permet de designer la façon dont la relation doit être lue. Ces relations sont caractérisées par un terme (voire plusieurs), une signature qui précise le nombre d'instances de concepts que la relation lie et leurs types. Elles englobent les associations suivantes : sous-classe de (généralisation spécialisation) ; partie de (agrégation ou composition) ; associée-à ; instance de ; est un ; etc.

On distingue alors, les relations taxonomiques (dite aussi de subsomption) et les relations associatives.

a)Relation de subsomption :

La relation de subsomption « est-un » (is-a) a un statut particulier car elle structure **la hiérarchie ontologique**.

La relation de subsomption n'est pas la seule relation qui permette de structurer la hiérarchie ontologique, la relation de méronymie, « partie-tout » (part-of) est souvent utilisée.

b)Relation associative :

Les relations « associatives » sont des relations d'interaction entre deux concepts qui ne sont pas la relation de subsomption. Elles correspondent à la notion de rôle en Logique de Description et permettent de typer les concepts reliés.

Les relations binaires sont parfois utilisées pour exprimer les attributs d'un concept. Les attributs sont distingués des relations par leur co-domaine qui est un type de donné tel que: nombre, chaîne de caractères….etc. Tandis que le co-domaine d'une relation est d'un genre plus complexe puisqu'il s'agit d'un autre concept présent dans l'ontologie.

2.4.3. Axiomes :

Les axiomes permettent de modéliser des assertions acceptées comme vraies, à propos des abstractions du domaine traduites par l'ontologie. Leur inclusion dans une ontologie peut avoir plusieurs objectifs: interviennent dans la définition des significations des composants d'ontologie, les contraintes sur les valeurs des attributs, les arguments de relations et dans l'inférence de nouvelles informations, etc. Les axiomes formels sont utilisés pour vérifier la consistance de l'ontologie.

2.4.4. Instances (ou individus)

Elles constituent la définition extensionnelle de l'ontologie. elles représentent les éléments singuliers véhiculant les connaissances à propos du domaine. [14]

Représentation des connaissances :

Une représentation des connaissances (ou Base de connaissances, ou ontologie) est la somme des TBox et des ABox :

Les ABox: sont les assertions concernant les instances . Elles contiennent des connaissances quelconques sur les instances.

Les TBox: sont les connaissances terminologiques concernant les classes. Les T-Box contiennent des formules logiques qui caractérisent les classes , c'est le lieu même des logiques descriptives, que l'on appelle aussi parfois les logiques terminologiques.

2.5. Classification des ontologies

En 2003 Gómez-Pérez [15]a classifie les ontologies selon la richesse de leur structure interne comme le travail de Lassila en 2001[16], et le sujet de conceptualisation (le but de leur utilisation)qui est une extension du travaux de Van Heijst [17]et ce du Guarino [18]en 1998:

2.5.1. Types d'ontologies bases sur la richesse de leurs structures :

a) **Vocabulaires contrôles :** c'est-à-dire un ensemble fini de termes, par exemple les catalogues ;

b) **Glossaires :** des listes de termes avec leurs significations, présenter en langage naturel.

c) **Thesaurus :** il fournit plus de la sémantique entre termes, il présente les informations comme des relations entre synonymes, mais il ne fournit aucune hiérarchie explicite ;

d) **Hiérarchies informelles :** la structure de ce type d'hiérarchie est basée non pas sur des relations de généralisation mais sur la proximité des concepts.

e) **Hiérarchies formelles :** hiérarchie dont la structure est déterminée par des relations de généralisation.

f) **Hiérarchie formelle avec instances du domaine**: similaire a la catégorie précédente mais incluant des instances.

g) **Frame** : ontologies incluant des classes avec propriétés pouvant être héritées.

h) **Ontologies avec restrictions de valeur**: ontologies pouvant contenir des restrictions sur les valeurs des propriétés.

i) **Ontologies avec contraintes logiques**: ontologies pouvant contenir des contraintes entre constituants (exemple relations) définies dans un langage logique.

2.5.2. .Types d'ontologies bases sur le sujet de conceptualisation :

a) **Ontologie pour la représentation de connaissances :** d'après Van Heijst, elle capture les primitifs de représentation employés pour formalisée la connaissance, l'exemple le plus représentatif et celui de Gruber, ils ont fournissent des définitions formelles des primitives de représentations utilises principalement dans les langages de Frame.

b) **Ontologie générale ou commune :** utilise pour représenter un sens commun de la connaissance réutilisable à travers des domaines, ces ontologies incluent le vocabulaire lié aux : choses, évènements, espace,…

c) **Les ontologies supérieures (Upper ou Top-level ontologie) :** modélisent les concepts très généraux auxquels les racines des ontologies de plus bas niveaux devraient être liées. Cependant, il existe plusieurs ontologies de niveau supérieur et qui sont divergentes. Afin de résoudre ce problème, l'organisation de standardisation IEEE tente de développer une ontologie de niveau supérieur qui soit standard.

d) **Les ontologies de domaine :** ce sont des ontologies qui sont construites sur un domaine particulier de la connaissance. Elles fournissent le vocabulaire des concepts du domaine de connaissance et les relations entre ces derniers, les activités de ce domaine ainsi que les théories et les principes de base de ce domaine. Les ontologies de domaine constituent donc des méta-descriptions d'une représentation de connaissances du domaine.

e) **Les ontologies de tache :** Ces ontologies sont utilisées pour gérer des taches spécifiques liées a la résolution de problèmes dans les systèmes, telles que les taches de diagnostic, de planification, de configuration, etc. Elles fournissent un ensemble de termes au moyen desquels on peut décrire au niveau générique comment résoudre un type de problème.

f) **Les ontologies d'application :** ce sont les ontologies les plus spécifiques. Elles permettent de décrire des concepts dépendants à la fois d'un domaine et d'une tache. Dans cette classification, la notion d'ontologie d'application définit le contexte d'une application qui décrit la sémantique des informations et des services manipules par une ou un ensemble d'applications sur un même domaine.[19]

2.6. Langages de représentation de l'ontologie

Une des principales décisions à prendre dans le procédé de développement d'ontologies consiste à choisir le langage dans lequel l'ontologie sera exprimée et utilisée.

Au début des années 1990, des langages fondés sur des techniques de représentation de l'Intelligence Artificielle (IA) ont été conçus pour la spécification des ontologies. Ils sont basés principalement, sur les formalismes de représentation de connaissances suivants :

- La logique du premier ordre tel que ***KIF*** ;
- Les frames combinés avec la logique du premier ordre tel qu'Ontolingua, OCML et ***Flogic*** ;
- La logique de description telle que ***Loom***.

Ces langages sont considérés comme étant traditionnels (appelés aussi classiques) par rapport à ceux présentés dans la section suivante.

Le boom d'Internet a mené à la création des langages d'implémentation des ontologies exploitant les caractéristiques du Web. Ils sont connus sous le nom des langages de balisage (*markup languages*) ou des langages d'ontologie dans le contexte du Web sémantique (web-based ontology language). Certains d'entre eux sont basés sur la syntaxe de XML tels que : ***XOL*** (Ontology Exchange Language), ***SHOE*** (Simple HTML Ontology Extension) qui a été précédemment basé sur le HTML, ***RDF*** (Resource Description Framework) et ***RDF Schéma*** qui est une extension de RDF.

La combinaison de RDF et RDF *Schéma* est connue sous le nom de ***RDF(S)***. Par la suite, trois autres langages ont été développés comme une extension de RDF(S) qui sont basés sur la logique de description : ***OIL*** (Ontology Inference Layer), ***DAML+OIL*** et ***OWL*** qui est le

successeur de DAML+OIL. OWL est fractionné en trois langages distincts : ***OWL LITE***, ***OWL DL*** et ***OWL FULL***.

2.6.1. RDF

RDF (Ressource Description Framework) développé et recommandé par le W3C, permet de décrire les ressources du web sémantique qui sont l'élément de base de RDF. Chaque ressource est pourvue d'un identifiant URI (Uniform Resource Identifier). Tout document RDF est composé d'un ensemble de triplets (sujet, prédicat, objet) ou encore (ressource, propriété, valeur). Un ensemble de tels triplets est appelé un graphe RDF. Ceci peut être illustré par un diagramme composé de nœuds et d'arcs orientés, dans lequel chaque triplet est représenté par un lien nœud-arc-nœud (d'où le terme de "graphe").
A ce modèle est associée une syntaxe écrite en XML et basée sur les triplets :

- Ressource (Sujet) : une entité d'informations pouvant être référencée par un identificateur. Cet identificateur doit être une URI.
- Propriété (prédicat) : l'attribut ou la relation utilisée (e) pour décrire une ressource.
- Valeur (objet) : la valeur d'une propriété associée à une ressource spécifique.

2.6.2. RDFS

Afin de renforcer ce langage, RDF Schéma a été construit par W3C comme extension de RDF comportant des primitives basées sur des frames. RDF Schéma permet notamment de déclarer les propriétés des ressources ainsi que le type des ressources. Bien que relativement limités dans la mesure où ils ne sont pas très expressifs, les langages RDF(S) peuvent cependant spécifier des concepts, des taxonomies et des relations binaires.

2.6.3. DAML-OIL

DAML-OIL a été proposé par le W3C pour représenter des méta-données et des ontologies.
DAML a été transformé en DAML+OIL en intégrant certaines propriétés d'OIL. Il repose sur RDF et RDF schéma et fournit en plus des primitives plus riches issues de la logique de description.

2.6.4.OWL :

2.6.4.1.Objectifs et motivation

OWL est, tout comme RDF, un langage XML profitant de l'universalité syntaxique de XML. Fondé sur la syntaxe de RDF/XML, OWL offre un moyen d'écrire des ontologies web.

OWL se différencie du couple RDF/RDFS en ceci que, contrairement à RDF, il est justement un langage d'ontologies. SI RDF et RDFS apportent à l'utilisateur la capacité de décrire des classes (i.e. avec des constructeurs) et des propriétés, OWL intègre, en plus, des outils de comparaison des propriétés et des classes : identité, équivalence, contraire, cardinalité, symétrie, transitivité, disjonction, etc.

Ainsi, OWL offre aux machines une plus grande capacité d'interprétation du contenu web que RDF et RDFS, grâce à un vocabulaire plus large et à une vraie sémantique formelle.

2.6.4.2.Différentes déclinaisons de OWL

Plus un outil est complet, plus il est, en général, complexe. C'est cet écueil qu'a voulu éviter le groupe de travail WebOnt du W3C en dotant OWL de trois sous-langages offrant des capacités d'expression croissantes et, naturellement, destinés à des communautés différentes d'utilisateurs :

- **OWL Lite** est le sous langage de OWL le plus simple. Il est destiné aux utilisateurs qui ont besoin d'une hiérarchie de concepts simple. OWL Lite est adapté, par exemple, aux migrations rapides depuis d'anciens thésaurus.

- **OWL DL** est plus complexe que OWL Lite, permettant une expressivité bien plus importante. OWL DL est fondé sur la logique descriptive (d'où son nom, OWL Description

Logics). Malgré sa complexité relative face à OWL Lite, OWL-DL garantit la complétude des raisonnements (toutes les inférences sont calculables) et leur décidabilité (leur calcul se fait en une durée finie).

- **OWL Full** est la version la plus complexe d'OWL, mais également celle qui permet le plus haut niveau d'expressivité. OWL Full est destiné aux situations où il est plus important d'avoir un haut niveau de capacité de description, quitte à ne pas pouvoir garantir la complétude et la décidabilité des calculs liés à l'ontologie. OWL Full offre cependant des mécanismes intéressants, comme par exemple la possibilité d'étende le vocabulaire par défaut de OWL.

Il existe entre ces trois sous langage une dépendance de nature hiérarchique : toute ontologie OWL Lite valide est également une ontologie OWL DL valide, et toute ontologie OWL DL valide est également une ontologie OWL Full valide.

2.6.4.3.Structure d'une ontologie OWL

Cette partie indique les principaux éléments constituant une ontologie OWL. Avant de poursuivre, il est bien nécessaire de comprendre que OWL n'a pas été conçu dans un esprit fermé permettant de borner les frontières d'une ontologie. Au contraire, la conception de OWL a pris en compte la nature distribuée du web sémantique et, de ce fait, a intégré la possibilité d'étendre des ontologies existantes, ou d'employer diverses ontologies existantes pour compléter la définition d'une nouvelle ontologie.

a. Espaces de nommage

Afin de pouvoir employer des termes dans une ontologie, il est nécessaire d'indiquer avec précision de quels vocabulaires ces termes proviennent. C'est la raison pour laquelle, comme tout autre document XML, une ontologie commence par une déclaration d'espace de nom (parfois appelée « de nommage ») contenue dans une balise rdf:RDF. Supposons que nous souhaitons écrire une ontologie sur une population de personnes ou, d'une manière plus générale, sur l'humanite. Voici la déclaration d'espace de nom qui pourrait être employée :

xmlns = "http://domain.tld/path/humanite#"

xmlns:humanite= "http://domain.tld/path/humanite#"

Ces déclarations identifient l'espace de nommage propre à l'ontologie que nous sommes en train d'écrire. La première déclaration d'espace de nom indique à quelle ontologie se rapporter en cas d'utilisation de noms sans préfixe dans la suite de l'ontologie.

La déclaration xmlns:base = "http://domain.tld/path/humanite#" identifie l'URI de base de l'ontologie courante.

La déclaration xmlns:vivant = "http://otherdomain.tld/otherpath/vivant#" signifie simplement que, au cours de la rédaction de l'ontologie humanité, on va employer des concepts développés dans une ontologie vivant, qui décrit ce qu'est un être vivant.

Les déclarations :

xmlns:owl = "http://www.w3.org/2002/07/owl#"

xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

xmlns:rdfs = "http://www.w3.org/2000/01/rdf-schema#"

xmlns:xsd = "http://www.w3.org/2001/XMLSchema#"

Introduisent le vocabulaire d'OWL et les objets définis dans l'espace de nommage de RDF, du schéma RDF et des types de données du Schéma XML.

Afin de simplifier l'écriture des URI dans la déclaration d'espace de nom et, surtout, dans les valeurs des attributs de l'ontologie, il est conseillé de définir des abréviations au moyen d'entités de type de document :

```
<!DOCTYPE rdf:RDF [
<!ENTITY humanite "http://domain.tld/path/humanite#" >
<!ENTITY vivant "http://otherdomain.tld/otherpath/vivant#" >
]>
```

Ainsi, la déclaration d'espace de nom initiale devient :

```
<rdf:RDF
xmlns = "&humanite;"
xmlns:humanite= "&humanite;"
xmlns:base = "&humanite;"
xmlns:vivant = "&vivant;"
xmlns:owl = "http://www.w3.org/2002/07/owl#"
xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs = "http://www.w3.org/2000/01/rdf-schema#"
```

xmlns:xsd = "http://www.w3.org/2001/XMLSchema#">

b.En-têtes d'une ontologie

Tout comme il existe une section d'en-tête <head>..</head> en haut de tout document XHTML bien formé, on peut écrire, à la suite de la déclaration d'espaces de nom, un en-tête décrivant le contenu de l'ontologie courante. C'est la balise owl:Ontology qui permet d'indiquer ces informations :

<owl:Ontology rdf:about="">

<rdfs:comment>Ontologie décrivant l'humanité</rdfs:comment>

<owl:imports

rdf:resource="http://otherdomain.tld/otherpath/vivant"/>

<rdfs:label>Ontologie sur l'humanité</rdfs:label>

...

Le contenu et la signification des différents éléments de cette section d'en-tête sont parfaitement décrits dans « OWL Web Ontology Language Reference », et il est recommandé de s'y reporter pour plus de détails. [20]

2.7.Outils de développement des ontologies

Les outils de développement d'ontologies qui existent sur le marché aujourd'hui sont divers et variés à bien des égards. Cet état de choses suscite beaucoup d'interrogations lorsque vient le moment d'en choisir un pour construire une nouvelle ontologie :
L'outil offre-t-il une assistance au développement ? dispose-t-il d'un moteur d'inférence ?
Quels langages d'ontologies supporte-t-il ? permet-il d'importer/exporter des ontologies ? offre-t-il un support à la réutilisation d'ontologies existantes ? permet-il de documenter les ontologies construites ? offre-t-il un support graphique à la construction des ontologies ? Les réponses à toutes ces questions pourraient aider à prendre une décision dans le choix de l'un ou l'autre outil.
Dans cette section, nous allons présenter quelques outils d'ingénierie ontologique. Ils permettent à l'utilisateur de créer des ontologies de manière indépendante des langages de

représentation et de prendre en charge la phase d'opérationnalisation de l'ontologie en l'exportant dans des langages informatisés standards.

2.7.1. OILEd [21]

L'éditeur OILEd**10** a été développé en 1991 sous la responsabilité de l'université de Manchester pour éditer des ontologies dans les langages de représentation OIL, puis DAML+OIL. Il est orienté vers la représentation en logique de description expressive et, à ce titre, fournit tous les éléments d'interface permettant de spécifier des hiérarchies de concepts et de rôles, les restrictions sur les rôles et les instances. Il peut être connecté à un raisonneur de logique des descriptions tel que FaCT et RACER, capable de tester la satisfiabilité des ontologies construites ou d'expliciter de nouvelles relations de subsomption entre concepts complexes.

2.7.2. WebODE [22]

WebODE11 a été développé par le groupe Ontological Engineering du département d'Intelligence artificielle de la faculté d'Informatique de l'université polytechnique de Madrid. Un éditeur qui assurait le support de METHONTOLOGY, la méthodologie proposée par ce laboratoire. WebODE est composé de plusieurs modules : un éditeur d'ontologie qui intègre la plupart des services nécessaires à la construction d'ontologies (édition, navigation, comparaison, fusion, raisonnement…), un système de gestion des connaissances à base ontologique, un outil pour annoter les ressources du web et un éditeur de services pour le Web sémantique.

2.7.3. ONTOEDIT [23]

ONTOEDIT est un environnement d'ingénierie ontologique mis au point par l'institut AIFB de l'université de Karlsruhe et qui est maintenant commercialisé par la société Ontoprise GmbH. Cet outil est fondé sur un processus de développement d'ontologies suivant les différentes étapes de la méthode de construction ON-TO-KNOWLEDGE et met à disposition de l'utilisateur plusieurs vues graphiques correspondant aux différentes phases de conception de l'ontologie.

2.7.4. PROTÉGÉ :

La plate-forme de Protégé prend en charge deux façons principales de modélisation d'ontologies via les éditeurs Protégé-Frames et Protégé-OWL. Les Ontologies de Protégé peuvent être exportées dans plusieurs formats, y compris RDF (S), OWL, et XML Schéma.

2.8.Moteurs d'inférences

La plupart des moteurs d'inférences existants sont conçus pour raisonner sur les logiques de descriptions, mais acceptent en entrée des fichiers OWL. Une fois l'ontologie chargée, ces moteurs effectuent les inférences sur la TBox et la ABox. Le tableau suivant contient des exemples des moteurs d'inferences :

tableau 2.8 exemple des moteurs d'inférences existants

Moteur	RACER	Pellet	FaCT	FaCT++
DL	*SHIQ*	*SHIN(D)* *SHON*	*SHIQ* *SHF*	*SHIF*
Implantation	C++	Java	Common Lisp	C++
Inférence	TBox/ABox	TBox/ABox	TBox	TBox
API Java	Oui	Natif	Oui	Oui
OWL	OWL DL	OWL DL	OWL DL	OWL Lite
Décidabilité	Oui (OWL Lite	Oui (OWL Lite)	Oui	Oui

Pellet et Racer sont à l'heure actuelle les deux seuls moteurs d'inférence, permettant le raisonnement sur la ABox et la TBox et exploitent des ontologies possédant un niveau d'expressivité en logique de description et acceptent en entrée des fichiers OWL. [14]

chapitre 2:
la programmation logique

La programmation logique

1.Un mode de programmation à part

Il est important avant d'expliquer ce qu'est PROLOG et d'étudier son utilisation de bien voir ce qu'il n'est pas ; c'est à dire de comprendre sa 'philosophie'. Pour cela, il faut décrire ce qu'est la programmation logique et la comparer aux autres modes de programmation.

1.1. Les autres modes de programmation

1.1.1. La programmation impérative

Cette programmation s'inscrit dans une démarche algorithmique qui décrit la façon de traiter les données pour atteindre un résultat par une série d'actions. Celles-ci sont toujours de trois types : le test, l'ordre (chaque instruction par laquelle le programme passe est impérative) et l'itération (obtenue grâce aux branchements).

Le déroulement du programme est parfaitement déterministe. Par exemple Fortran, Pascal, C….

1.1.2. La programmation fonctionnelle

Le résultat est ici comme la composition de fonctions. Pratiquement, elle est proche de la programmation impérative ; cependant ses fondements (λ-calcul) ainsi que l'absence de branchement et d'affectation (du moins dans sa forme théorique) en fait un mode de programmation à part. Dans la programmation fonctionnelle, on distingue deux grandes familles : - les langages fortement typés : ML (ex : Caml) - les langages non typés : LISP (ex : Schème)

1.2. La programmation logique

Les modes de programmation décrits juste au-dessus sont dits procéduraux car ils cherchent à obtenir le résultat grâce à une procédure (qui peut être une suite d'actions ou une composition de fonctions). A cela on oppose la programmation logique qui est dite déclarative. En effet, ici on ne s'occupe pas de la manière d'obtenir le résultat ; par contre, le programmeur doit faire la description du problème à résoudre en listant les objets concernés, les propriétés et les relations qu'ils vérifient.

Ensuite, le mécanisme de résolution (pris entièrement en charge par le langage) est général et universel. Il parcourt de façon non déterministe toutes les possibilités du problème et peut donc retourner plusieurs solutions.

1.3. La programmation orientée objet

Ce mode de programmation a été mis à part car il regroupe en fait tous les modes précédemment vus en utilisant à la fois des techniques déclaratives et d'autres procédurale. Par exemple : C++, Java…..

2. Constitution d'un programme PROLOG

Nous avons vu que le principe de la programmation logique est de décrire le problème à résoudre. Dans le cas de PROLOG, cela est formalisé grâce à un langage dérivé du calcul des prédicats (ou calcul du premier ordre). Les prédicats servent à qualifier (donner les caractéristiques de) les objets du problème et à décrire les relations dans lesquelles ils sont impliqués.

2.1. Les faits

Les faits sont des données élémentaires qu'on considère vraies. Ce sont des formules atomiques constituées d'un nom suivi entre parenthèse d'une liste ordonnée d'arguments qui sont les objets du problème principal ou d'un sous-problème.

Un programme PROLOG est au moins constitué d'un ou plusieurs fait(s) car c'est à partir de lui (d'eux) que PROLOG va pouvoir rechercher des preuves pour répondre aux requêtes de l'utilisateur. Généralement, on place toutes les déclarations de faits au début du programme même si ce n'est pas obligatoire.

2.2. Les règles :

Un programme PROLOG contient presque toujours des règles, cependant ce n'est pas une obligation. Si les faits sont les hypothèses de travail de PROLOG, les règles sont des relations qui permettent à partir de ces hypothèses d'établir de nouveaux faits par déduction (si on a démontré F1 et F1⇒F2 alors on a démontré F2).

Les relations qui se traduisent en règle sont de la forme : H1/H2/ … Hn ⇒ C

Où :

- / peut être soit une disjonction (∨) soit une conjonction (∧)
- H1, H2,… Hn sont des formules atomiques ou des directives de contrôle
- C'est une formule atomique.

Les variables utilisées dans une règle sont universellement quantifiées par PROLOG (avec quantificateur ∀). Ainsi, il interprète

H1/H2/ … Hn ⇒ C par ∀ X1, X2, … Xm (H1/H2/ … Hn ⇒ C) si X1, X2, … Xm sont les variables des Hk.

Ex : grand_père(X,Y):-père(X,Z),mère(X,Y). est quantifié

- soit par : ∀X ∀Y ∀Z (père(X,Z) & mère(X,Y))⇒grand_père(X,Y)

- soit par : ∀X ∀Y ,∃Z / (père(X,Z) & mère(X,Y))⇒grand_père(X,Y)

3. Utilisations de PROLOG

3.1. Interrogation de bases de données relationnelles

Il est très facile d'utiliser PROLOG pour programmer une base de données et l'interroger. En effet, il suffit de déclarer en tant que faits toutes les données de la base.

- Il existe en fait une extension de PROLOG appelée DATALOG qui est justement orientée base de données. Grâce aux règles, on peut obtenir des sous-tables ou établir des relations entre les tables.

- C'est ce qui fait la puissance d'un langage comme PROLOG ou DATALOG par rapport à SQL où toutes les données de la base doivent être explicitement énoncées.

- Les bases de données sont un exemple où le programme peut ne contenir que des faits sans aucune règle même si cela fait perdre une partie de l'intérêt de PROLOG.

3.1.1. Programmation de la base de données

Pour créer une table bio avec les attributs nom, sexe, annee_naissance, père, mère ; nous écrivons avec chaque tuple l'instruction suivant bio (nom, sexe, annee_naissance, père, mère).

par exemple : bio (mohammed, homme,1985 , ahmed, fatma).

Grâce aux règles nous pouvons définir un sous-table : 'enfant' qui ont comme attributs enfant (NomEnfant, NomParent) par les deux règles suivantes :

enfant(X,Y):-bio(X,_,_,Y,_).

enfant(X,Y):-bio(X,_,_,_,Y).

3.1.2. Interrogation de la base de données

La base de données étant en place, nous allons l'interroger grâce à des requêtes PROLOG. Il est possible de l'interroger de nombreuses manières. En voici quelques exemples.

3.1.2.1. Vérification de la présence d'une donnée dans la table :

Il est bien sûr possible de savoir si une donnée existe bien dans la base.

ex : ?- bio(ahmed, homme, 1986, mohammed, fatma).

la résultat est « Yes » si il existe sinon « No »

3.1.2.2. Recherche d'une liste simple

On peut extraire de la table 'bio' une sous-table à un seul attribut.

Cela correspond à l'opération : Πx (σcondition (bio)) où

- X est un (et un seul) attribut de la table 'bio'
- Conditions est un ensemble de conjonctions et/ou de disjonctions

Ex : Quelles sont les femmes ?

C'est à dire que contient ΠNom (σSexe=femme (bio)) ?

?- bio(Qui,f,_,_,_).

Qui = fatma ;

Qui = maria ;

No

3.1.2.3. Recherche d'une liste multiple

On peut aussi extraire de la table 'bio' une sous-table ayant plus d'un seul attribut.

Cela peut correspondre à l'opération : ΠX1,X2,…Xn (σConditions (bio)) où

- X1, X2, …Xn sont des attributs de la table 'bio'
- Conditions est un ensemble de conjonctions et/ou de disjonctions

ex: Qui sont les parents de mohammed ?

?- bio(mohammed,_,_,Papa,Maman).

Papa = ahmed

Maman = fatma ;

No

3.1.2.4. Recherche dans un intervalle de valeurs

PROLOG peut gérer un intervalle de valeurs.

Ex : Quels sont les personnages nés entre 1982 et 1985 ?

?- bio(Qui,_,N,_,_),1982=<N,N=<1985.

Qui = maria

N = 1982 ;

Qui = ahmed

N = 1984 ;

No

La recherche de solutions dans un intervalle de valeur fait apparaître quelques caractéristiques intéressantes de PROLOG :

• L'implémentation de PROLOG ne respecte pas complètement la logique du premier ordre : une requête n'est pas traitée comme une formule complète mais elle est divisée en formules atomiques (les buts) reliées par des disjonctions et/ou des conjonctions et ces sous-formules sont traitées les unes après les autres dans l'ordre où elles se trouvent sur la ligne (de gauche à droite).

• PROLOG considère à priori toute variable numérique comme faisant partie des réels.

Ex :Quels sont les personnages nés entre 1982 et 1985 ?

?- 1982=<N,N=<1985,bio(Qui,_,N,_,_).

ERROR: Arguments are not sufficiently instantiated

Par rapport au premier exemple, nous n'avons fait qu'inverser les formules de la requête ce qui en logique ne change rien puisque l'opérateur ∧ est commutatif. Mais comme nous l'avons dit juste au-dessus, PROLOG découpe les requêtes en formules atomiques. Il va donc d'abord s'attacher à résoudre le but 1982=<N. Or N étant à priori un réel, il y a une infinité de solution à ce but. Il arrive à le détecter et alors il s'arrête.

Remarque : Nous venons de voir dans cet exemple un des rares cas où PROLOG effectue un contrôle. Sans cela, il chercherait tous les N (réels) ≤ 1985 avant de passer à la formule atomique suivante. Donc il ne terminerait jamais.

Pour contourner ces deux difficultés (les règles sont découpées en formules atomiques et les variables numériques sont considérées comme des réels) dans le cas d'une recherche dans un intervalle de valeurs, PROLOG introduit le mot clé between(debut,fin,X) où X est une variable considérée comme un entier et comprise entre 'début' et 'fin'.

Ex:?- between(1982,1985,N),bio(Qui,_,N,_,_).

N = 1982

Qui = maria;

N = 1984

Qui = ahmed ;

No

Cette fois PROLOG veut bien s'exécuter car il y a un nombre fini d'entiers compris entre 1982 et 1985. On retrouve bien les même résultats qu'avec le premier exemple

PROLOG permet également de façon tout à fait naturelle de rechercher des solutions n'appartenant pas à un ensemble de valeurs.

Ex : Quels sont les personnages qui ne sont pas nés entre 1982 et 1985 ?

?- bio(Qui,_,N,_,_),(1982>N;N>1985).

Qui = mohammed

N = 1980 ;

No

3.1.2.5. Utilisation des sous-tables

Lorsque nous avions programmé la base de données, en plus de la table ‘bio’, nous avions également déclaré trois autres sous tables : ‘enfant’, ‘ptenfant’ et ‘descendant’ par l’intermédiaire de règles. Il est tout à fait possible d’interroger ces dernières comme nous l’avons fait avec ‘bio’.

3.2. Formalisation de systèmes experts

3.2.1. PROLOG et les systèmes experts

PROLOG est parfaitement adapté pour formaliser des systèmes experts. En effet, un système expert est un programme informatique simulant l'intelligence humaine dans un champ particulier de la connaissance ou relativement à une problématique déterminée. Or PROLOG a justement été conçu dans cette optique là puisqu’il a été fait par des chercheurs en intelligence artificielle.

Un système expert a trois composantes essentielles :

- une base de connaissances, formée des énoncés relatifs aux faits de tous ordres constitutifs du domaine
- un ensemble de règles de décision, consignant les méthodes, procédures et schémas de raisonnement utilisés dans le domaine

Or ces trois points sont extrêmement simples à implémenter dans programme PROLOG:

- la base de connaissances est constituée par les faits et quelques règles pour éviter l’énumération exhaustive de tous les faits
- les règles de décision sont des règles (au sens de PROLOG)
- le moteur d’interface est l’interpréteur PROLOG lui-même.
- un moteur d'inférence, sous-système qui permet d'appliquer les règles de décision à la base de connaissances.

3.2.2. Constitution de la base de connaissance

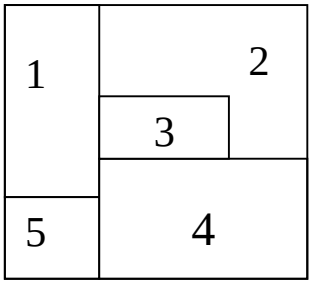
En guise d’exemple de système expert, nous allons formaliser le problème de coloriage de région. Les règles de ce problème sont les suivantes :

• Une surface est découpée en un certain nombre de régions de surfaces et de formes variables

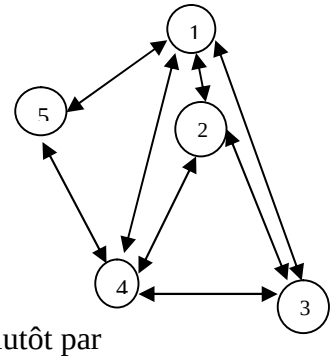
• Chaque région doit être coloriée

• Deux régions adjacentes doivent avoir deux couleurs différentes

Nous essaierons de colorier les régions suivantes :



que nous représenterons plutôt par

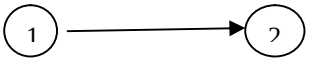


La deuxième représentation permet de s'abstraire de toute géométrie :

• Un sommet représente une région

• Une arête reliant deux sommets signifie que les régions équivalentes sont adjacentes

C'est cela qu'on va traduire dans le programme PROLOG par le prédicat adjacent

Ex : adjacent(1,2). Signifie 

ce qui n'est pas exactement « La région 1 est adjacente à la région 2. »

3.2.2.1. Enumération exhaustive

La première solution pour constituer la base de connaissance est de faire une énumération exhaustive des régions adjacentes.

exemple: Base de connaissances exhaustive

adjacent(1,2). adjacent(2,1).

adjacent(1,3). adjacent(3,1).

adjacent(1,4). adjacent(4,1).

adjacent(1,5). adjacent(5,1).

adjacent(2,3). adjacent(3,2).

adjacent(2,4). adjacent(4,2).

adjacent(3,4). adjacent(4,3).

adjacent(4,5). adjacent(5,4).

Cette base de connaissance fonctionne parfaitement.

Ex : Les régions 1 et 2 sont adjacentes :

?- adjacent(1,2) ,adjacent(2,1).

Yes

La région 2 n'est pas adjacente à la région 11 (qui n'existe pas)

?- adjacent(2,11).

No

Cependant, on voit très bien que ce genre de base de connaissances est limité : si le problème devient trop complexe, on ne peut pas saisir à la main toutes les possibilités. De plus cette écriture sous-exploite le potentiel de PROLOG qui grâce aux règles peut grandement alléger cette écriture.

3.2.2.2. Ecritures condensées

Nous allons condenser l'écriture exhaustive en remplaçant certains faits par des règles ce qui comme nous le verrons n'est pas sans danger.

a). Règle de commutation

Essayons tout d'abord d'introduire une règle de commutativité. En effet, cette règle si naturelle pour nous quand on parle de coté adjacent fait cruellement défaut à PROLOG et permettrait de réduire de moitié le nombre de faits.

ex : Base de connaissances commutative (erronée)

```
%Liste des régions adjacentes 1 fois (sans commutation)
adjacent(1,2).
adjacent(1,3).
adjacent(1,4).
adjacent(1,5).
adjacent(2,3).
adjacent(2,4).
adjacent(3,4).
adjacent(4,5).
/*La règle de commutation */
adjacent(X,Y):-adjacent(Y,X).
```

Cette règle de commutation fonctionne très bien dans certains cas.

Le programme détecte parfaitement les régions adjacentes :

?- adjacent(2,4).

Yes

?- adjacent(4,2).

Yes

2 et 4 sont bien deux régions adjacentes

Malheureusement, dans d'autres cas elle engendre des boucles infinies.

Ex : Le programme boucle systématiquement pour détecter des régions non adjacentes :

?- adjacent(2,5).

Action (h for help) ? abort

Execution Aborted

Ex : Enumérer toutes les régions adjacentes à 1

?- adjacent(1,X).

X = 2 ;

X = 3 ;

X = 4 ;

X = 5 ;

X = 2 ;

X = 3 ;

X = 4 ;

X = 5 ;

X = 2 ;

X = 3 ;

X = 4 ;

X = 5

…

PROLOG ne boucle pas tout seul mais il retourne toujours la même série de solutions tant que l'utilisateur lui en demande encore : il ne s'arrête pas tout seul pour signifier qu'il n'y en a plus d'autres.

En fait, la règle de commutativité remplie parfaitement son rôle pour déterminer que deux régions sont adjacentes car soit elle n'est jamais invoquée soit elle est utilisée une seule fois avant de trouver le fait permettant d'arrêter la recherche.

Ex : [debug] ?- adjacent(2,4).

T Call: (8) adjacent(2, 4)

T Exit: (8) adjacent(2, 4)

PROLOG trouve la preuve directement dans les faits.

[debug] ?- adjacent(4,2).

T Call: (8) adjacent(4, 2)

T Call: (9) adjacent(2, 4)

T Exit: (9) adjacent(2, 4)

T Exit: (8) adjacent(4, 2)

Cette fois il est obligé d'appliquer la règle de commutativité 1 fois (2ème call) puis il trouve un fait pour elle, il en sort et il répond à la requête.

Par contre, si la requête porte sur deux régions non adjacentes, PROLOG va appeler une infinité de fois la règle de commutativité.

Ex : [debug] ?- adjacent(2,5).

T Call: (8) adjacent(2, 5)

T Call: (9) adjacent(5, 2)

T Call: (10) adjacent(2, 5)

T Call: (11) adjacent(5, 2)

…….

Les zone 2 et 5 n'étant pas adjacentes, ni adjacent(2,5), ni adjacent(5,2) ne fait partis des faits.

La première action de PROLOG est de regarder si la requête est un fait. Puisque ce n'est pas le cas, il applique la règle de commutation en espérant que cela lui permet de conclure. Il reparcourt donc les faits mais il ne trouve encore aucune preuve pour répondre à la requête. Donc il applique une nouvelle fois la règle de commutation.

Dans le cas d'une énumération de zones adjacentes à une autre, la règle de commutation crée en quelque sorte une infinité de faits en répétant les faits une fois à l'endroit, une fois en commutant les paramètres et ceci une infinité de fois.

Ex : [debug] ?- adjacent(1,X).

T Call: (7) adjacent(1, _G304)

T Exit: (7) adjacent(1, 2)

X = 2 ;

T Exit: (7) adjacent(1, 3)

X = 3 ;

T Exit: (7) adjacent(1, 4)

X = 4 ;

T Exit: (7) adjacent(1, 5)

X = 5 ;

T Redo: (7) adjacent(1, _G304)

T Call: (8) adjacent(_G304, 1)

T Redo: (8) adjacent(_G304, 1)

T Redo: (8) adjacent(_G304, 1)

T Call: (9) adjacent(1, _G304)

T Exit: (9) adjacent(1, 2)

T Exit: (8) adjacent(2, 1)

T Exit: (7) adjacent(1, 2)

X = 2

…

Remarque : Il existe des méthodes pour prévenir ces boucles, cependant pour des raisons d'efficacité, elle ne sont pas utilisées dans PROLOG.

c). Enumération par variables

Une autre façon de condenser l'écriture est de faire l'énumération des quatre premiers faits (et leurs images commutées) par l'intermédiaire d'une variable. En effet, on remarque que ces faits sont tous de la forme adjacent(1,x) avec $2 \leq x \leq 5$.

Ex :Base de connaissances avec énumération par variable (erronée)

```
% Liste des régions adjacentes autres que la 1
adjacent(2,3).
adjacent(2,4).
adjacent(3,4).
adjacent(4,5).
adjacent(3,2).
adjacent(4,2).
adjacent(4,3).
adjacent(5,4).
% Règles concernant la région 1
adjacent(1,X):-X>=2,X=<5.
adjacent(X,1):-X>=2,X=<5.
```

Ce programme marche très bien avec des requêtes simples

Ex : On peut savoir que les régions 3 et 1 sont adjacentes :

```
?- adjacent(1,3),adjacent(3,1).
Yes
```

ou que les régions 5 et 2 ne le sont pas :

?- adjacent(5,2).

No

Par contre, une erreur peut apparaître lorsqu'on emploie des requêtes un peu plus complexes.

Ex : Enumérer toutes les régions adjacentes à 2

?- adjacent(2,X).

X = 3 ;

X = 4 ;

X = 1 ;

No

La région 2 est bien adjacente aux régions 1, 3 et 4.

Ex : Enumérer toutes les régions adjacentes à 1

?- adjacent(1,X).

ERROR: Arguments are not sufficiently instantiated

^ Exception: (7) _G158>=2 ? abort

% Execution Aborted

PROLOG refuse d'exécuter la requête. En effet, il s'agit d'un des rares contrôle de PROLOG. Sans ce contrôle, il regarderait adjacent(1,X) pour tous les $X \geq 2$ puis pour tous les $X \leq 5$; or il y en a une infinité dans les deux cas et il entrerait donc dans une boucle infinie.

Il existe une solution à ce problème. PROLOG fournit un système pour borner une variable entière : between.

Ainsi, il faut remplacer les deux règles concernant la région 1 par :

```
% Règles concernant la région 1
adjacent(1,X):-between(2,5,X).
adjacent(X,1):-between(2,5,X).
```

3.2.2.3. Relations entre paramètres

On remarque que concernant les régions (2,3), (3,4) et (4,5) les deux paramètres X et Y du prédicat adjacent sont liés par la relation Y=X+1. On pourrait donc écrire explicitement cette relation.

Ex :Base de connaissances avec relation entre paramètres (erronée)

```
% Régions adjacentes explicitement déclarées
adjacent(2,4).
```

adjacent(4,2).

% Règles concernant la région 1

adjacent(1,X):-between(2,5,X).

adjacent(X,1):-between(2,5,X).

% Règles concernant les couples (2,3), (3,4) et (4,5)

adjacent(X,X+1).

adjacent(X+1,X).

Ce programme présente plusieurs défauts :

• D'abord, il est syntaxiquement incorrect. En effet, PROLOG ne reconnaît pas « X+1 » comme étant la valeur de X augmenté de 1. Pour cela, il dispose du mot clé is :

adjacent(X,Y):-Y is X+1.

adjacent(X,Y):-Y is X+1.

• Ensuite, comme nous l'avons vu au chapitre précédent, il faut spécifier que X est un entier borné en utilisant between

adjacent(X,Y):-between(2,4,X), Y is X+1.

adjacent(X,Y):-between(2,4,X), Y is X+1.

Ex :Finalement, nous avons pu condenser la base de connaissances en :

Base de connaissances condensée (finale)

% Régions adjacentes explicitement déclarées

adjacent(2,4).

adjacent(4,2).

% Règles concernant la région 1

adjacent(1,X):-between(2,5,X).

adjacent(X,1):-between(2,5,X).

% Règles concernant les couples (2,3), (3,4) et (4,5)

adjacent(X,Y):-between(2,4,X),Y is X+1.

adjacent(Y,X):-between(2,4,X),Y is X+1.

Cette écriture condensée est exactement équivalente au programme exhaustif

Test sur deux régions adjacentes

?- adjacent(1,2),adjacent(2,1).

Yes

1 et 2 sont bien adjacentes

Enumération de régions adjacentes

?- adjacent(1,X).

X = 2 ;

X = 3 ;

X = 4 ;

X = 5 ;

No

1 est adjacente exactement à 2, 3, 4 et 5

3.3. Mise en place des règles de décision

Pour les règles de décision, on introduit deux nouveaux prédicats :

• conflit(Coloriage) qui permet de voir si un coloriage des régions respecte les contraintes que nous nous sommes fixées

• conflit(X,Y,Coloriage) qui permet de savoir quelles régions adjacentes ont la même couleur

Un coloriage des régions est une fonction qui à chaque région associe une couleur. Elle est réalisée par le prédicat :

color(Region,Couleur,Coloriage)

Formalisation d'un système expert (complète)

```
% Description des zones
adjacent(2,4).
adjacent(4,2).
adjacent(1,X):-between(2,5,X).
adjacent(X,1):-between(2,5,X).
adjacent(X,Y):-between(2,4,X),Y is X+1.
adjacent(Y,X):-between(2,4,X),Y is X+1.
% Règles de décisions
conflit(Coloriage):-adjacent(X,Y),color(X, Couleur, Coloriage),color(Y, Couleur, Coloriage).
conflit(X,Y,Coloriage):-adjacent(X,Y),color(X, Couleur, Coloriage),color(Y, Couleur, Coloriage).
% Exemples de coloriages possibles
/* Coloriage sans conflit */
color(1,bleu,coloriage1).
```

color(2,rouge,coloriage1).

color(3,vert,coloriage1).

color(4,jaune,coloriage1).

color(5,rouge,coloriage1).

/* Coloriage avec conflit */

color(1,vert,coloriage2).

color(2,rouge,coloriage2).

color(3,vert,coloriage2).

color(4,jaune,coloriage2).

Nous pouvons maintenant vérifier si un coloriage est valide et dans le cas contraire, connaître les régions adjacentes qui ont la même couleur.

Ex : Un coloriage sans conflit

?- conflit(coloriage1).

No

Le coloriage 1 ne possède effectivement pas de régions adjacentes de même couleur.

Ex : Un coloriage avec conflit

?- conflit(coloriage2).

Yes

Les régions adjacentes de même couleur sont :

?- conflit(X,Y,coloriage2),between(X,5,Y).

X = 1

Y = 3 ;

X = 1

Y = 5 ;

No

La région 1 a la même couleur que la région 3 et la région 5.

ex: Toutes les régions de la même couleur

?- conflit(coloriage3).

Yes

Toutes les régions adjacentes entrent en conflit

?- conflit(X,Y,coloriage3),between(X,5,Y).

X = 2

Y = 4 ;

X = 1

Y = 2 ;

X = 1

Y = 3 ;

X = 1

Y = 4 ;

X = 1

Y = 5 ;

X = 2

Y = 3 ;

X = 3

Y = 4 ;

X = 4

Y = 5 ;

No

Effectivement, il s'agit bien là de toutes les régions adjacentes.

4. Exécution d'un programme PROLOG

4.1. L'interactivité par les requêtes

Un programme PROLOG est interactif. En effet, dans un langage comme le C, une fois le programme compilé l'utilisateur ne peut utiliser celui-ci que de la manière dont le programmeur l'a voulue. En PROLOG, les faits et les règles servent à définir le comportement général des outils mis à la disposition de l'utilisateur pour travailler mais ensuite ce dernier peut faire ce qu'il lui plait. Cette interactivité est obtenue par l'intermédiaire de requête que l'utilisateur soumet à l'interpréteur PROLOG.

4.1.1. Des questions et des réponses

Le déroulement d'un programme PROLOG est un jeu de questions / réponses : l'utilisateur interroge l'interpréteur PROLOG qui lui répond toujours au moins par oui ou non. Parfois, il instancie également les variables d'une requête s'il a trouvé une preuve de validité de cette instanciation.

Ex : Avec le programme de base de données précédant, nous avons demandé à PROLOG si Louis XIII a vécu entre 1601 et

1643 et est le fils de Henri IV et Marie de Médicis :

?- bio(louis13, h, 1601, 1643, henri4, marie_medicis).

Nous lui avons aussi demandé d'énumérer tous les personnages nés entre 1750 et 1800 :

?- bio(Qui,_,N,_,_,_),1750=<N,N=<1800.

Il faut noter que l'utilisateur peut poser toutes les requêtes qu'il désire en se servant des règles comme il l'entend.

Cependant, cette interactivité n'est pas sans risque. De fait, pour des questions de performances, PROLOG fait très peu de contrôles. Ainsi, si l'utilisateur ne sait pas ce qu'il fait, il peut facilement mettre l'interpréteur PROLOG en défaut notamment en rentrant dans une boucle infinie.

4.1.2. Un programme évolutif

Mais surtout là où l'interactivité de PROLOG est remarquable c'est qu'il est possible de modifier le programme pendant son exécution. En effet, PROLOG a été conçu à l'origine pour l'intelligence artificielle, un programme PROLOG doit donc être capable d'évoluer durant son existence. Cette évolution peut être automatisée ou forcée par l'utilisateur.

4.2. La recherche de preuves par l'unification

4.2. 1. Fonctionnement général

a). La logique

Pour répondre à une requête, PROLOG cherche à prouver celle-ci par la logique.

En effet, nous avons vu au chapitre I.2.2 que chaque règle PROLOG C:-H. est interprétable par « H ⇒ C ». Donc pour démontrer C (la conclusion) PROLOG va essayer de démontrer H (les hypothèses). Si H est démontrable alors C l'est aussi, sinon il essaie avec une autre règle et si aucune règle ou fait ne permet de prouver C alors C'est faux. Ce procédé est itératif.

b). L'implémentation

Remarque : On appelle but une conclusion que PROLOG cherche à démonter.

Le premier but qu'une requête est toujours la requête elle-même.

Pour trouver une preuve et afficher la solution associée, PROLOG utilise deux piles de piles :

La pile des buts dans laquelle il stocke tous les niveaux de buts qu'il a cherché à atteindre. Un niveau de buts est luimême représenté par une pile de buts qui contient toujours au moins un but sauf lorsqu'on a atteint la fin de la démonstration (on peut alors

retourner le résultat en cas de succès ou no sinon). Une preuve est donc l'ensemble des buts par lesquels on est passé avant d'obtenir une pile de niveau de buts vide.

• La pile d'environnement est elle-aussi une pile de piles car à chaque niveau de buts est associée une pile d'environnement de niveau de buts. PROLOG stocke dans chacune de ces sous-piles les variables intervenues dans l'unification qui l'a amené à ce niveau. Dans la pile d'environnement, il conserve donc toutes les variables qui ont été utilisées pendant la démonstration. Elle lui sert ainsi à retrouver le résultat de la preuve en cas de succès.

Le changement de niveau de but correspond à un pas d'itération dans la démonstration logique (cf. chapitre précédent) et comme nous l'avons vu juste au-dessus, il est validé par une unification. En effet, pour prouver un but PROLOG :

• Soit trouve un fait pour lequel il existe une unification avec le but. Dans ce cas le but courant est démontrable et itérativement la requête est démontrable.

• Soit essaie d'appliquer la règle dont le membre gauche (la conclusion) est unifiable avec le but. S'il en existe une, alors il se fixe comme nouveau(x) but(s) la (ou les) formule(s) atomique(s) du membre droit de cette règle. Il passe ainsi à un autre niveau de but. Si plusieurs règles ont un membre gauche unifiable, PROLOG choisit la première dans le code du programme. Si celle-ci mène à une démonstration (succès), l'utilisateur peut demander une autre solution. Alors PROLOG remonte au dernier choix qu'il a du faire et appliqué la règle suivante s'il y en a encore une, sinon il remonte au choix précédent et s'il n'y a plus aucun choix possible il retourne no.

Ce fonctionnement en niveaux avec parfois plusieurs choix possibles à certains niveaux permet de qualifier l'exécution d'un programme PROLOG de parcours préfixe d'arbre. En effet, il est tout à fait possible, comme nous allons le voir, de représenter une telle exécution par un arbre.

4.2.2. Exemples avec des listes

4.2.2.1. Propriétés des listes

En PROLOG, les listes sont prédéfinies. Elles sont notées entre [et]. Leurs éléments sont séparés par des ,.

Ex : [] est la liste vide

[a,b,c] est une liste non vide

[[a,b,c],12 ,X,[f,g]] est une autre liste contenant des sous-listes et une variable.

Elles sont toujours constituées d'une tête et d'une queue. La tête est toujours un et un seul élément (qui peut éventuellement être une autre liste). La queue est toujours une liste (éventuellement vide). Le symbole | permet d'unir ou séparer ces deux parties.

Ex : Concaténation d'un élément en tête d'une liste :

[a|[c,d,e]] is [a,b,c,d,e]

Ex : Extraction de la tête ou de la queue d'une liste :

?- [a,b,c,d,e]=[H|T].

H = a

Cas d'une liste où la tête est une autre liste :

?- [[a,b,c,d],e]=[H|T].

H = [a, b, c, d]

T = [e]

Yes

Cas d'une queue vide :

?- [a]=[H|T].

H = a

T = []

Yes

Il est possible grâce à l'unification d'extraire d'une liste n'importe quel élément. Cependant, toutes les unifications ne sont pas forcément valides.

Ex : Cas d'unifications impossibles :

?- []=[H|T].

No

La liste vide n'a pas de tête.

?- [[a,b],[c,d,e]]=[[H1|T1]|[H2|[H3|T3]]].

No

Essayons de faire l'unification à la main pour comprendre où est l'erreur.

1) {[a,b]=[H1|T1]

[[c, d, e]]= [H2|[H3|T3]]}

2) H1= a

T1= [b]

H2=[c,d,e]

[]= [H3|T3]

3) Nous avons vu juste au-dessus que la liste vide n'avait pas de tête donc l'unification []=[H3|T3]n'a pas de solution.

Ex : Recherche d'éléments dans une liste

La liste contenant la liste vide possède une tête et une queue :

?- [[]]=[Tete|Queue].

Tete = []

Queue = []

Yes

La queue est toujours incorporée dans une liste :

?- [0,[a,b,c]]=[Tete|Queue].

Tete = 0

Queue = [[a, b, c]]

Yes

La queue est donc ici une liste ayant pour seul élément une autre liste à trois éléments. On peut récupérer cette sous-liste par la requête suivante :

?- [0,[a,b,c]]=[Tete|[SousListe|Fin]].

Tete = 0

SousListe = [a, b, c]

Fin = []

Yes

Récupération de tous les éléments d'une queue :

?- [[a,b,c],[d,e,f],g]=[Tete|[[D|[E|[F|T1]]]|[G|T2]]].

Tete = [a, b, c]

D = d

E = e

F = f

T1 = []

G = g

T2 = []

Yes [25].

Chapitre 3:

eXtensible Stylesheet Language

(XSL)

Introduction :

Le langage XSL(eXtensible Style Language) permet de présenter visuellement des éléments définis dans un document XML alors que le langage XML (eXtended Markup Language) définit plutôt la sémantique (le sens) des données.

Le langage XSL se divise en trois parties principales :

- Le formatage : application de règles de style sur des éléments XML à l'instar du langage CSS.(Cascading Style Sheet)
- La transformation : substitution d'un marquage XML en un balisage HTML ou un autre marquage XML.

La partie formatage du langage XSL (eXtensible Stylesheet Language) a une fonction semblable à celle du langage CSS (Cascading StyleSheet).Un ensemble de propriétés de style permet de contrôler la présentation des données à l'écran.

Ainsi, un document XML pourra être mis en forme pour un affichage sur un moniteur informatique, un écran de télévision ou pour l'impression ou encore pour une version auditive.

Le langage XSL, par une série de règles de transformation, remplace les éléments XML et leurs attributs en balisage HTML (HyperText Markup Language) ou en d'autres marqueurs XML. Cette section du langage XSL s'appelle XSLT soit Langage des feuilles de Style de Transformation dont les spécifications sont mises au point par le W3C (Worl Wide Web Consortium).

Par des règles de transformation, les données textuelles contenues dans les éléments XML ou dans leurs attributs sont présentés selon le résultat de la génération d'un balisage HTML.

Une feuille de style XSL peut entièrement réorganiser les éléments XML en sélectionnant des éléments et leurs attributs puis en les transformant en d'autres éléments.

Cette fonctionnalité peut être utilisée lorsqu'un document XML doit être intégré dans un ensemble d'autres documents XML. Si les éléments ne correspondent pas, alors il est nécessaire de les transformer afin de les rendre compatibles.

Le fonctionnement du langage XSL s'effectue selon des règles de style applicables à différent motif d'un document XML.

Ainsi, tous les motifs énoncés par la feuille de style et trouvés dans le document XML par un processeur XSL, sont soit formatés, soit transformés en une combinaison textuelle spécifique.

1. Le processus de transformation :

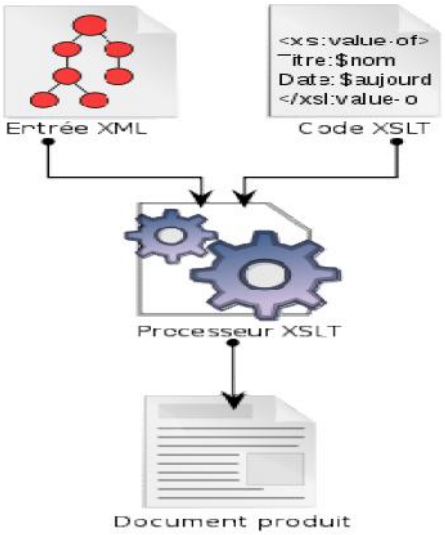


Fig. 1. Processus de transformation

La transformation ou le formatage permet d'afficher sur le navigateur Internet d'un ordinateur client, les données d'un document XML structurées en conformité avec les règles de style d'un document XSL. Selon la méthode de sortie de la feuille de style, le document résultant peut être généré en HTML, en texte brut ou en une autre arborescence XML.

Le processus de transformation ou de formatage d'un document XML peut s'effectuer par trois moyens distincts :

- Un processeur XSL installé sur l'ordinateur client (en général dans le navigateur) se charge de transformer localement la source XML en conjonction avec la feuille de style XSL.
- Un processeur XSL installé sur le serveur envoie, après traitement, le document résultant au client.
- Un logiciel spécifique installé sur le serveur effectue un traitement préalable des documents XML, puis stocke les documents résultants sur le serveur lui-même.

Ces trois moyens nécessitent diverses ressources logiciels, mais tous font appel à un langage XSL identique sans injection d'instructions spécifiques.

La première solution demande à l'ordinateur client de posséder impérativement un navigateur compatible avec les technologies XML à l'image d'Internet Explorer 6.0 ou Netscape 6.1.

La seconde solution demande un aménagement logiciel du serveur web avec par exemple, un programme XML Enabler d'IBM, afin de le rendre compatible aux technologies XML/XSL.

Enfin, la dernière solution consiste à installer un moteur de transformation XML nommé XT (XML Transformer) associé à un analyseur (ou parser) conforme à SAX (Simple API for XML) ou DOM (Document Object Model).

De nombreux programmes permettent de mettre à niveau son serveur :

- JAXP édité par Sun Microsystems.
- Xalan & Xerces édités par l'Organisation Apache
- XP & XT édités par l'auteur des spécifications XSL, James Clark.
- MSXML 4.0 édité par Microsoft.
- XML Parser édité par IBM.

L'analyse d'un document XML peut être accomplie selon deux méthodes :

- **Le DOM** (Document Object Model) autorise la gestion de l'ensemble d'un document. Dans ce cas, la source XML est entièrement analysée par rapport au DTD (Document Type Définition) ou au schéma (XSchema) associé. Si le document est conforme aux règles structurelles énoncées alors le parser le valide, subséquemment peut commencer la manipulation de l'arborescence XML par un programme.

Le DOM possède un inconvénient du fait de l'analyse complète d'un document XML avant traitement. En effet, si la source XML est volumineuse, l'analyse risquerait d'affecter les performances d'un serveur.

- **Le SAX** (Simple API for XML) permet de traiter événementiellement un document XML. Les éléments contenus dans le document XML sont analysés au fur et à mesure de leur exploitation par un programme de formatage. Ainsi, seuls les éléments utilisés sont soumis à une validation par le parser.

En fait, lorsque l'analyseur SAX rencontre un nouveau nœud (racine, élément, attribut, données textuelles analysées, instruction de traitement, espace de noms ou commentaire), est créé un événement que réceptionne le programme de transformation.

En fonction de ces prérogatives, le programme déclenchera l'analyse puis traitera le nœud concerné.

Contrairement au DOM, SAX consomme moins de ressources machines et en conséquence s'adapte idéalement au traitement de document XML volumineux. DOM sera préféré pour des sources XML plus légères.

Par ailleurs, un processeur XML peut être validant ou non-validant ou les deux. Ce type doit être pris en considération, en fonction du genre d'activité souhaitée. Un parseur non-validant est forcément plus performant.

Cependant, l'arborescence du document XML ne sera pas analysée en conformité avec son DTD ou son schéma. Autrement dit le document XML sera obligatoirement bien formé mais risquera d'être non-valide.

En outre, les entités analysées ne pourront faire l'objet de substitutions textuelles dans le document résultant puisque le DTD ne sera pas pris en compte dans le traitement de la source XML.

En raison de ce genre de problèmes, il est préférable d'utiliser un analyseur XML validant.

2. Les espaces de noms ;

Afin de reconnaître les éléments et les attributs des espaces de noms XSL, le processeur XSL a besoin de les identifier précisément. Les éléments des feuilles de style XSLT utilisent le préfixe xsl: pour référencer les éléments de l'espace de noms XSLT (XSL Transformation). Quant aux feuilles de style de formatage, le préfixe utilisé dans les marqueurs XSL, est fo.

Cependant, les feuilles de style XSL sont libres d'utiliser n'importe quel préfixe, à condition que l'espace de noms soit déclaré au préalable. La déclaration de l'espace de noms s'effectue en associant un préfixe à un URI (Uniform Resource Identifier) dans l'élément racine de feuille de style.

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"> ...

</xsl:stylesheet>

L'URI associé au préfixe xsl: est : http://www.w3.org/1999/XSL/Transform

L'URI associé au préfixe fo: est :http://www.w3.org/1999/XSL/Format/1.0

3.les patterns (paths):

Les patterns sont des expressions destinées à la sélection de nœuds dans l'arborescence d'un document XML. En général, les patterns se trouvent dans l'attribut select ou match des éléments XSL-Transformation suivants :

xsl:template

xsl:apply-templates

xsl:value-of

xsl:copy-of

xsl:sort

xsl:for-each

<xsl:template match="pattern">

<xsl:apply-templates select="pattern">

Le tableau suivant contient des exemples des patterns et leur descriptions

tableau 3. Les paths

Pattern	Description
paragraphe	n'importe quel élément *paragraphe* du nœud courant.
*/paragraphe	n'importe quel élément *paragraphe* petit-fils du nœud courant.
Chapitre/paragraphe	n'importe quel élément *chapitre* et n'importe quel élément *paragraphe*.
chapitre/paragraphe	tout élément *paragraphe* avec un élément parent *chapitre*.
chapitre//paragraphe	tout élément *paragraphe* descendant d'un élément *chapitre* à quelque niveau de profondeur que ce soit.
/	le nœud racine.
./	tous les éléments fils du nœud courant.
../chapitre	tout les éléments *chapitre* fils du père du nœud courant.
id("Identificateur")	l'élément avec l'identificateur unique *Identificateur*.

tableau 3. Les paths(suite).

Pattern	Description
items/item[position()>1]	tout élément *item* qui a un *item* parent et qui n'est pas le premier *item* enfant de son parent.
item[position() mod 2 = 1]	n'importe quel élément *item* qui est un élément enfant *item* impair de son parent.
div[@class="appendice"]//para	tout élément *p* avec un élément *div* ancêtre qui a un attribut *class* avec la valeur *appendice*.
paragraphe/@class	tout attribut *class* des éléments *paragraphe* mais pas tout élément *paragraphe* qui a un attribut *class*.
@*	tout attribut.
para[1]	n'importe quel élément *para* qui est le premier élément enfant *para* de son parent
*[position()=1 and self::para]	n'importe quel élément *para* qui est le premier élément enfant *para* de son parent

3.1. Le langage xpath:

3.1.1. Les types de nœuds:

Un document XML possède sept types de nœuds différents. Ces nœuds permettent de naviguer dans l'arborescence d'un document et surtout de sélectionner des éléments, attributs, ou tout autre constituant. Suite à la sélection de ces derniers, des valeurs, en l'occurrence toujours des chaînes de caractères, peuvent être extraites de ces nœuds. Le tableau suivant représente les types des nœuds existants :

tableau. 3.1.1. les types de nœuds

Type de nœuds	Description
Racine	représente la valeur de l'instruction de traitement <xml-stylesheet> et l'élément racine.
Elément	représente la valeur de la concaténation de toutes les données caractères analysables trouvées dans l'élément lui-même et tous ces descendants.

tableau. 3.1.1. les types de nœuds(suite)

Type de nœuds	Description
Texte	représente le texte contenu dans le nœud courant.
Attribut	représente la valeur de l'attribut du nœud courant.
Espace de noms	représente l'URI (Uniform Resource Identifier) désignant l'espace de noms.
Instruction de traitement	représente la valeur des attributs contenue dans l'instruction.
Commentaire	représente le texte contenu dans le commentaire.

3.1.2. Les tests nœuds:

Les tests de nœuds permettent de préciser le type de nœuds à sélectionner par l'intermédiaire d'un pattern d'un élément XSL-T. Combiné avec des axes nodaux, des prédicats ou encore des fonctions nodales, les tests de nœuds améliorent la puissance de sélection des nœuds d'une arborescence d'un document XML.le tableau suivant représente les tests nœuds :

tableau. 3.1.2. Les tests nœuds

Test	Description
comment()	renvoie *true* si un nœud commentaire est trouvé.
text()	renvoie *true* si un nœud textuel est trouvé.
node()	renvoie *true* si un nœud autre qu'un nœud attribut ou un nœud racine est trouvé.
processing-instruction()	renvoie *true* si une instruction de traitement est trouvée.

3.1.3.Les axes nodaux :

Les axes nodaux permettent de sélectionner des parties de l'arborescence d'un document XML à partir du nœud courant. En fait, les axes nodaux ouvrent des directions de recherche indiquées par des préfixes situé avant le pattern et séparé par un double deux-points. préfixe::pattern. L'expression suivante sélectionne l'attribut nom des nœuds éléments fils directs de l'élément service.

following-sibling::service/@nom

le tableau suivant représente les axes nodaux :

tableau 3.1.3.Les axes nodaux

Axe nodal	Description
ancestor::	représente les nœuds parents du nœud courant jusqu'au nœud racine.
ancestor-or-self::	représente le nœud lui-même avec les mêmes caractéristiques que *ancestor*.
descendant::	représente les nœuds fils du nœud courant jusqu'au nœud terminal.
descendant-or-self::	représente le nœud lui-même avec les mêmes caractéristiques que *descendant*.
self::	représente le nœud lui-même.
parent::	représente les nœuds parents directs.
child::	représente les nœuds fils directs.
attribute::	représente les nœuds attributs du nœud courant.
following::	représente tous les nœuds suivant le nœud courant hormis les nœuds attributs et espaces de noms.
following-sibling::	représente la même chose que *following* mais essentiellement les nœuds qui ont le même parent que le nœud courant.
preceding::	représente tous les nœuds précédant le nœud courant hormis les nœuds attributs et espaces de noms.
preceding-sibling::	représente la même chose que *preceding* mais essentiellement les nœuds qui ont le même parent que le nœud courant.

3.1.3.1.les axes nodaux abrégés :

Les axes nodaux possèdent des synonymes abrégés permettant de naviguer dans l'arborescence d'un document à l'image des commandes de système de fichiers telles que connaissent UNIX, le DOS ou encore comme les liens relatifs sur Internet. Par exemple, l'expression suivante est équivalente à la seconde.

parent::personnel → ../personnel

le tableau suivant représente les axes nodaux abrégés :

tableau 3.1.3.1.les axes nodaux abrégés

Opérateur	Description
élément	sélectionne tous les éléments *élément* fils du nœud courant (*child::élément*).
*	sélectionne tous les éléments fils du nœud courant.
/	représente l'élément racine.
//	représente n'importe quel descendant de l'élément racine, donc tous les éléments (*descendant-or-self::node()*).
.	représente l'élément courant (*self::node()*).
..	permet de remonter d'un niveau dans l'arborescence du document par rapport à l'élément courant (*parent::node()*).
/élément	sélectionne tous les éléments *élément* sous l'élément racine ().
./élément	sélectionne tous les éléments *élément* sous l'élément courant (*following::élément*).
../élément	sélectionne tous les éléments *élément* sous l'élément parent du nœud courant (*preceding::élément*).
//élément	sélectionne tous les éléments *élément* descendants du nœud courant à quelque niveau de profondeur que ce soit.
@attribut	sélectionne tous les attributs *attribut* du nœud courant (*attribute::attribut*).
|	correspond à un *ou*.

3.1.4.Les opérateurs :

Les opérateurs sont utilisables dans des expressions XPath permettant de sélectionner des nœuds dans l'arborescence d'un document XML. Il existe deux sortes d'opérateurs :les opérateurs logiques, les opérateurs de calcul. Les premiers renvoient une valeur booléenne true ou false après avoir effectué un test entre les deux opérandes. Les seconds permettent d'accomplir un calcul entre deux nombres et de renvoyer le résultat.

3.1.4.1. Les opérateurs booléens :

Dans le tableau suivant (*tableau 3.1.4.1.)* nous avons présenté les opérateurs booléens :

tableau 3.1.4.1. Les opérateurs booléens

Opérateur	Description
or	représente un OU logique.
and	représente un ET logique.
not()	signifie la négation ou NON logique.
|	permet la sélection de plusieurs motifs.
=	représente l'égalité.
!=	signifie différent de...
<	signifie inférieur à...
<=	signifie inférieur ou égal à...
>	signifie supérieur à...
>=	signifie supérieur ou égal à...

3.1.4.2. Les opérateurs de calculs :

Dans le tableau suivant nous avons présenté les opérateurs de calculs :

tableau. 3.1.4.2. Les opérateurs de calculs

Opérateur	Description
+	effectue une addition.
-	effectue une soustraction.
div()	effectue une division.
mod()	effectue un modulo.

3.1.5. Les prédicats :

Les prédicats sont des expressions entre crochets permettant de cibler plus précisément une portion de l'arborescence d'un document. En coopération avec les axes nodaux, les types de nœud ou encore les fonctions nodales, les prédicats assurent un moyen puissant de sélection de nœuds. L'expression suivante est équivalente à la seconde.

3.1.6. Les fonctions xpath :

Le langage XPath comporte quatre types de fonctions distinctes permettant de travailler sur les nœuds d'une arborescence, sur des chaînes de caractères, sur des valeurs booléennes et enfin sur des nombres. Ces fonctions peuvent effectuer diverses opérations telles que de la concaténation sur des chaînes de caractères, des calculs sur des nombres, des évaluations d'expressions, etc.. Les fonctions sont utilisables non

seulement dans une expression XPath mais aussi, directement dans un pattern d'un élément XSL.le tableau suivant représente les fonctions xpath :

tableau 3.1.6.les fonctions xpath

Prédicat	Description
élément	sélectionne tous les éléments *élément* fils du nœud courant.
élément[n]	sélectionne le n^{ième} élément *élément* dans le nœud courant.
élément[elt]	sélectionne dans le nœud courant, l'élément *élément* qui a comme élément fils *elt*.
[elt="valeur"]	sélectionne dans le nœud courant, l'élément ayant pour fils un nœud *elt* qui a une valeur égale à *valeur*.
élément[@attribut]	sélectionne dans le nœud courant, l'élément *élément* qui possède un attribut *attribut*.
[@attribut='valeur']	sélectionne dans le nœud courant, l'élément dont l'attribut *attribut* a une valeur égale à *valeur*.

3.1.6.1. Les fonctions nodales :

Les fonctions nodales renvoyant différentes caractéristiques sur des ensembles de nœuds en général passés en argument. Ces fonctions ont divers rôles dans l'arborescence d'un document XML comme localiser, positionner, identifier, compter, ou encore dénommer des ensembles (ou jeux) de nœuds. Ces ensembles de nœuds sont des groupes de nœuds sélectionnés par l'intermédiaire des axes nodaux dans l'arborescence d'un document XML.

tableau 3.1.6.1. Les fonctions nodales

Fonction	Description
count(ensemble_noeud)	retourne le nombre de nœud dans l'ensemble de nœuds passé en argument.
current()	retourne le nœud courant.
document(objet, ensemble_noeud)	fournit un chemin pour retrouver d'autres ressources XML à l'intérieur d'une feuille de style de transformation au-delà des données fournies par l'entrée courante.

tableau 3.1.6.1. Les fonctions nodales(suite)

Fonction	Description
generate-id(ensemble_noeuds)	retourne une chaîne qui identifie individuellement le premier nœud dans un ensemble de nœuds passé en argument.
id("identifiant")	sélectionne l'élément dans le nœud courant par son identifiant (W3C ou Microsoft).
Key(nom, valeur)	retrouve les éléments précédemment marqués par une instruction *xsl:key*.
last()	retourne un nombre égal à la dimension contextuelle de provenant du contexte d'évaluation de l'expression.
local-name(ensemble_noeud)	retourne la partie locale du nom étendu du nœud dans l'ensemble de nœuds passé en argument qui est le premier dans l'ordre du document.
name(nœuds)	retourne une chaîne de caractères contenant un nom qualifié représentant le nom étendu du nœud dans l'ensemble de nœuds passé en argument qui est le premier dans l'ordre du document.
namespace-uri(ensemble_noeud)	retourne l'URI de l'espace de noms du nom étendu du nœud de l'ensemble de nœuds passé en argument qui est le premier dans l'ordre du document.
node-set(chaîne)	convertit une arborescence à l'intérieur d'un ensemble de nœuds. Le nœud résultant contient toujours un unique nœud et le chemin du nœud de l'arbre.
position()	retourne un nombre représentant la position du nœud courant à l'intérieur du nœud parent.

3.1.6.2.Les fonctions booléennes :

Le langage XPath utilise cinq fonctions booléennes. Ces fonctions permettent de travailler sur les valeurs logiques true ou false afin de créer des conditions dans les expressions XPath. On présente des fonctions booléennes dans le tableau suivant :

tableau 3.1.6.2.Les fonctions booléennes

Opérateur	Description
boolean(objet)	convertit l'argument en valeur booléenne.
élément-available(chaîne)	retourne la valeur booléenne *true* si et seulement si le nom étendu est le nom d'une instruction.
false()	retourne la valeur logique *false*.
function-available()	retourne *true* si la fonction est disponible dans la librairie de fonctions.
lang(chaîne)	retourne la valeur logique *true* si l'attribut *xml:lang* du nœud contextuel est le même que l'argument.
not(valeur)	retourne le contraire de la valeur booléenne passée en argument.
true()	retourne la valeur logique *true*.

3.1.6.3.Les fonctions chaines de caractères :

Les fonctions sur les chaînes de caractères permettent d'effectuer différentes opérations sur des chaînes de caractères. Ces fonctions peuvent effectuer des concaténations, des remplacements, des comparaisons sur des chaînes de caractères ainsi que la suppression des espaces blancs superflus ou encore de renvoyer le nombre de caractères.

tableau. 3.1.6.3.Les fonctions chaines de caractères

Opérateur	Description
concat(chaîne1,chaîne2,...)	concatène les chaînes de caractères passées en argument.
contains(contenant,contenu)	retourne *true* si la première chaîne de caractères contient la seconde, sinon renvoie *false*.
normalize-space(chaîne)	retourne l'argument chaîne après suppression des espaces superflus.

tableau. 3.1.6.3.Les fonctions chaines de caractères(suite)

Opérateur	Description
starts-with(chaîne,chaîne)	retourne *true* si la première chaîne commence avec les mêmes caractères que la seconde, sinon, elle renvoie *false*.
string(ensemble_noeud)	convertit son argument en chaîne de caractères.
string-length(chaîne)	retourne le nombre de caractères dans la chaîne.
substring(chaîne,position,longueur)	retourne la sous-chaîne du premier argument démarrant à la position et à la longueur spécifiées.
substring-after(chaîne,marqueur)	retourne la sous-chaîne du premier argument résultant de la suppression de tous les caractères précédant le marqueur localisé dans la chaîne.
substring-before(chaîne,marqueur)	retourne la sous-chaîne du premier argument résultant de la suppression de tous les caractères suivant le marqueur localisé dans la chaîne.
system-property(chaîne)	retourne un objet représentant la valeur de la propriété système identifié par un nom.
translate(chaîne,chaîne,chaîne)	retourne le premier argument chaîne dans lequel les occurences des caractères de la deuxième chaîne sont remplacées par les caractères correspondant aux mêmes positions de la troisième chaîne.
unparsed-entity-uri(chaîne)	retourne les déclarations d'entités non-analysées dans la Définition de Type de Document (DTD) de la source du document.

3.1.6.4. Les fonctions numériques :

Les fonctions numériques permettent d'effectuer différentes opérations sur des nombres. Ces fonctions peuvent calculer une somme, arrondir ou encore convertir des valeurs passées en argument. Cela permet de construire des expressions XPath complexes et dynamiques. Le tableau suivant représente les fonctions numériques :

tableau. 3.1.6.4.Les fonctions numériques

Fonction	Description
ceiling(nombre)	retourne le plus petit entier qui n'est pas inférieur au nombre passé en argument.
floor(nombre)	retourne le plus grand entier qui n'est pas supérieur au nombre passé en argument.
format-number(nombre,format age,décimal)	convertit le premier argument en une chaîne de caractères utilisant le masque de formatage sur le nombre et le cas échéant le format décimal spécifié.
number(objet)	convertit l'argument en un nombre.
round(nombre)	retourne la valeur la plus proche du nombre passé en argument.
sum(ensemble_noeud)	retourne la somme de tous les nœuds composant l'ensemble de nœuds après que chacun des nœuds ait subit en une valeur numérique.

3.1.7.Les fonctions xslt :

La syntaxe du langage XPath supporte des fonctions qui fournissent des informations relatives à des nœuds dans une collection. Ces fonctions retournent des chaînes de caractères ou des nombres et peuvent être utilisées avec des opérateurs de comparaisons dans un pattern. Ces fonctions ne sont pas disponibles à partir des méthodes de sélection de nœuds dans le DOM (Document Object Model). Le tableau suivant représente les fonctions XSLT :[26]

tableau 3.1.7.Les fonctions xslt

Fonction	Description
current()	retourne un jeu de nœud qui possède le nœud courant comme son seul membre.
élément-available()	retourne la valeur *true* si et seulement si le nom étendu eest le nom d'une instruction.
format-number()	convertit le premier argument d'une chaîne de caractères en une chaîne utilisant le *pattern* de formatage spécifié par le second argument.
function-available()	retourne la valeur *true* si la fonction est disponible dans la librairie de fonctions.

tableau 3.1.7.Les fonctions xslt(suite)

Fonction	Description
generate-id(jeu_noeuds)	retourne une chaîne de caractères unique qui identifie le nœud cible dans un jeu de nœuds passé en argument.
node-set(jeu_noeuds)	convertit une arborescence à l'intérieur d'un jeu de nœuds. Le nœud résultant contient toujours un unique nœud et la racine de l'arborescence.
system-property()	retourne un objet représentant la valeur de la propriété système identifié par un nom.
unparsed-entity-uri()	retourne les déclarations d'entités non-analysées dans la DTD (document type définition) du document XML source.

4. Structure d'un document XSL :

Un document XSL étant un document XML, il commence obligatoirement par la balise suivante :

<?xml version="1.0" encoding="ISO-8859-1"?>

D'autre part, toute feuille de style XSL est comprise entre les balises *<xsl:stylesheet ...>* et*</xsl:stylesheet>*.

La balise *<xsl:stylesheet* >encapsule des balises *<xsl:template>* définissant les transformations à faire subir à certains éléments du document XML.

<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet

xmlns:xsl="http://www.w3.org/TR/WD-xsl"

xmlns="http://www.w3.org/TR/REC-html40"

result-ns="">

<xsl:template ... >

<!-- traitements à effectuer -->

```
    </xsl:template >
</xsl:stylesheet>
```

5. Association d'une feuille XSL à un document XML :

Une feuille de style XSL (enregistré dans un fichier dont l'extension est *.xsl*) peut être liée à un document XML (de telle manière à ce que le document XML utilise la feuille XSL) en insérant la balise suivante au début du document XML :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet href="fichier.xsl" type="text/xsl"?>
```

6. Les template rules (règles de gabarit) :

Les template rules sont des balises XSL permettant de définir des opérations à réaliser sur certains éléments du document XML utilisant la page XSL, c'est-à-dire généralement de transformer un tag XML en au moins un tag HTML (généralement plusieurs).

Ainsi le tag XML suivant :

```
<personne>
<nom>Pillou</nom>
<prénom>Jean-François</prénom>
</personne>
<personne>
<nom>VanHaute</nom>
<prénom>Nico</prénom>
</personne>
<personne>
<nom>Andrieu</nom>
<prénom>Seb</prénom>
</personne>
```

pourra être transformé en les tags HTML suivants :

```
<ul>
<li>Pillou - Jean-François</li>
<li>VanHaute - Nico</li>
```

Andrieu - Seb

L'attribut "match" de la balise <xsl:template> permet de définir (grâce à la notation XPath) le ou les éléments du document XML sur lesquels s'applique la transformation. [27]

7. Les éléments de transformation :

Les éléments de transformations permettent de sélectionner et effectuer des opérations sur les éléments du document XML. Leur syntaxe est la suivante :

<xsl:nom [attributs]/>

Remarquez la présence du / indiquant que la balise ne possède pas de balise fermante.Voici une petite liste des éléments de transformation : [28]

tableau 7. Les éléments de transformation

Elément	**Utilité**
xsl:apply-imports	Importe une feuille de style importée
xsl:apply-templates	Indique au processeur XSL de traiter les éléments enfants directs en leur appliquant les template rules définies dans la feuille XSL. L'attribut *select* permet de spécifier certains éléments enfants auxquels la transformation doit être appliquée
xsl:attribute-set	Permet de créer un attribut à associer à un élément
xsl:attribute-set	Permet de nommer une liste d'attributs, pouvant être appliqués à un élément particulier
xsl:template	Permet de charger un canevas (*template*) grâce à son nom.
xsl:choose	Structure conditionnelle de type "case" (utilisé en combinaison avec *xsl:when* et/ou*xsl:otherwise*)
xsl:comment	Crée un commentaire dans l'arbre résultat

tableau 7. Les éléments de transformation(suite)

Elément	**Utilité**
xsl:copy	Copie le nœud courant dans l'arbre résultat
xsl:copy-of	Copie le nœud sélectionné par le modèle dans l'arbre résultat
xsl:decimal-format	Déclare un format de nombre décimal
xsl:element	Permet de créer un élément avec le nom spécifié
xsl:for-each	Permet d'appliquer un canevas à chaque nœud correspondant au modèle
xsl:if	Permet d'effectuer un test conditionnel sur le modèle indiqué

Chapitre 4 :

Implémentation de Transformation

Introduction :

Dans ce chapitre, nous allons expliquer notre travail et cette explication sera avec un exemple d'ontologies. Par exemple l'ontologie de description d'une population qui décrit ensemble des personnes, des pays, des villes et leurs relations entre eux .

1.Explication du travail :

Notre travail est basé sur xsl, nous avons utilisé l'éditeur oxygenXML : est le meilleur éditeur XML disponibles, avec un grand nombre d'utilisateurs allant des débutants aux experts de XML. Il est le seul outil XML qui supporte tous les langages de schéma XML. Le support XSLT et XQuery est renforcée avec les débogueurs et les profileurs de performance puissants. L'éditeur XML <oXygen/> peut être utilisé pour travailler avec toutes les technologies basées sur XML, y compris des bases de données XML et des services web. l'image suivante représente l'interface de l'éditeur oxygen XML :

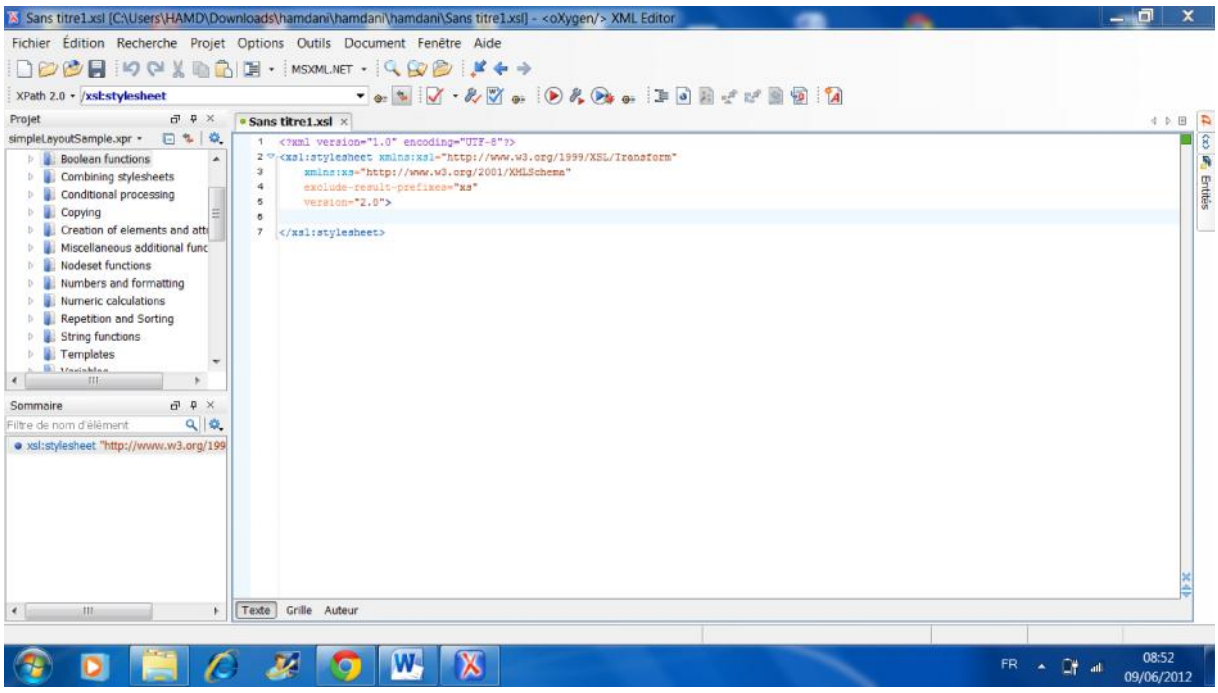


Fig1.l'interface de l'éditeur <oxygen/>XML

2.Explication de l'idée de transformation :

L'idée principale de travail est d'appliquer une fonction sur un nœud de l'arbre OWL pour obtenir le résultat de prolog selon les règles des transformations(voir annexes).

3.Explication de l'exemple étudié:

Notre population est composé d'une classe Humains divisé en deux sous-classes Femme et Homme ,qui habitent dans une ville qui est a sa tour situe dans un pays.les Humains peut

avoir une relation de fraternité entre eux , un Humain surement a un lien de parenté avec un autre, l'Homme a une relation de mariage avec une Femme, une ville situe forcement dans un pays. n'oublier pas les attributs :pour l'Humain on a nom, prénom et date de naissance en plus pour femme il y a nom de jeune fille, pour pays nom pays et pour ville nom ville.

3.1.Définition de TBox et ABox :

3.1.1.TBox :

- ✓ **Définition des classes :**Humain, Homme, Femme, Pays, Ville.
- ✓ **Définition des propriétés :**
  a. **Les propriétés d'objet :** habitent_a, a_pour_pere, se_trouve _en, a_un_lien_de_fraternité et est_marie_a .les deux derniers sont définis comme propriétés symétriques les autres propriétés fonctionnelles .
  b. **Les properieté de type de donnée :** Nom, Prénom, Date_de_naissance pour Humain, Nom_de_jeune_fille pour Femme, Nom_ville pour Ville et Nom_pays pour Pays.

3.1.2.ABox :

Nous définirons l'ontologie d' une famille appelé « la famille SALEM ».

Nous avons définit des personnes avec ses propriétés soit propriétés d'objets soit propriétés de type de donnée par exemple :

- ✓ Homme :

(SALEM, Ahmed.25-12-1946).

- ✓ Femme :

(SALEM, Fatima, Fati,17-12-1978).

- ✓ Pays : Algérie .
- ✓ Ville :Alger, Msila.
- ✓ Les propriétés d'objets : A_pour_pere (Ahmed, Omar).

3.2.Diagramme de classes de l'ontologie :

Pour mieux présenter notre ontologie on décrit le diagramme de classes qui la représente :

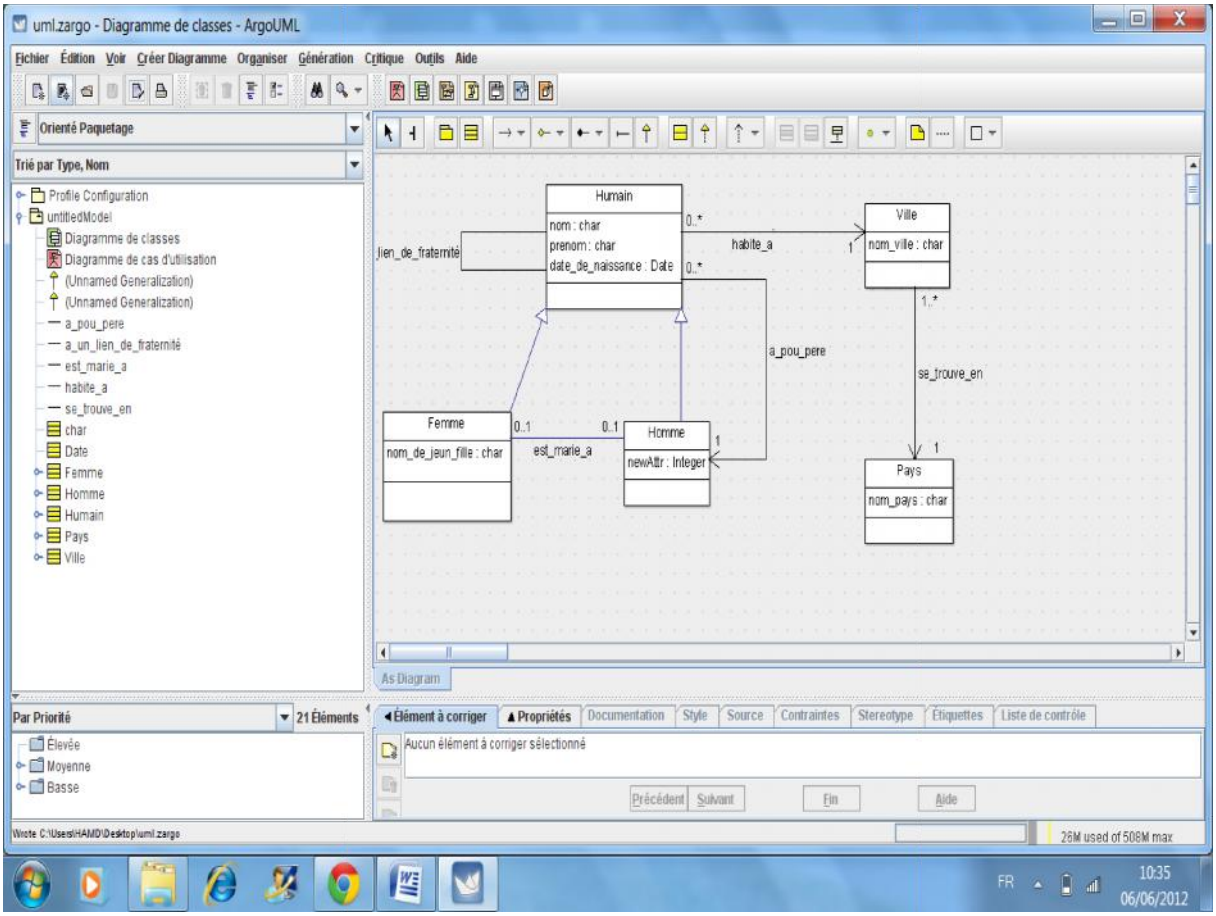


Fig. 3.2.Diagramme de classes de l'ontologie

3.3. Presentation de l'ontologie avec protégé :

Nous définissons protégé dans le 1e r chapitre nous utilisons dans notre projet la version 4.2_alpha

3.3.1.présentation de TBox :

L'image suivante représente les classes avec Protégé :

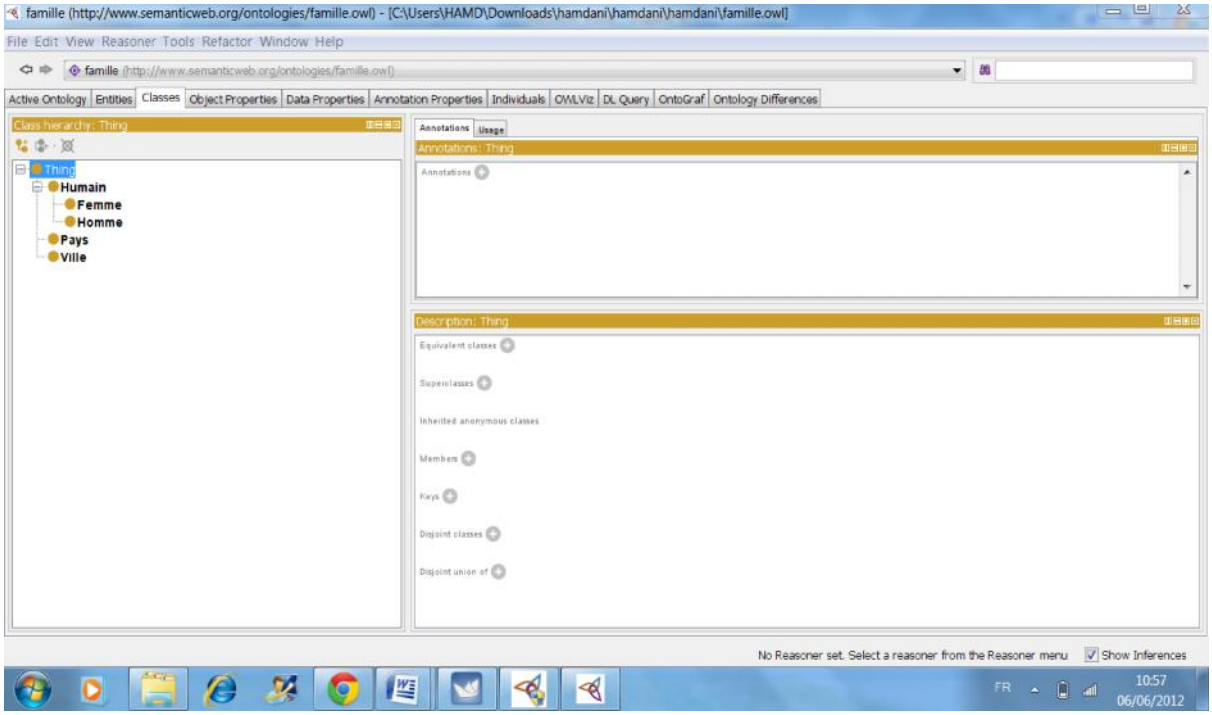


Fig. 3.3.1.1.présentation des classes de l'ontologie

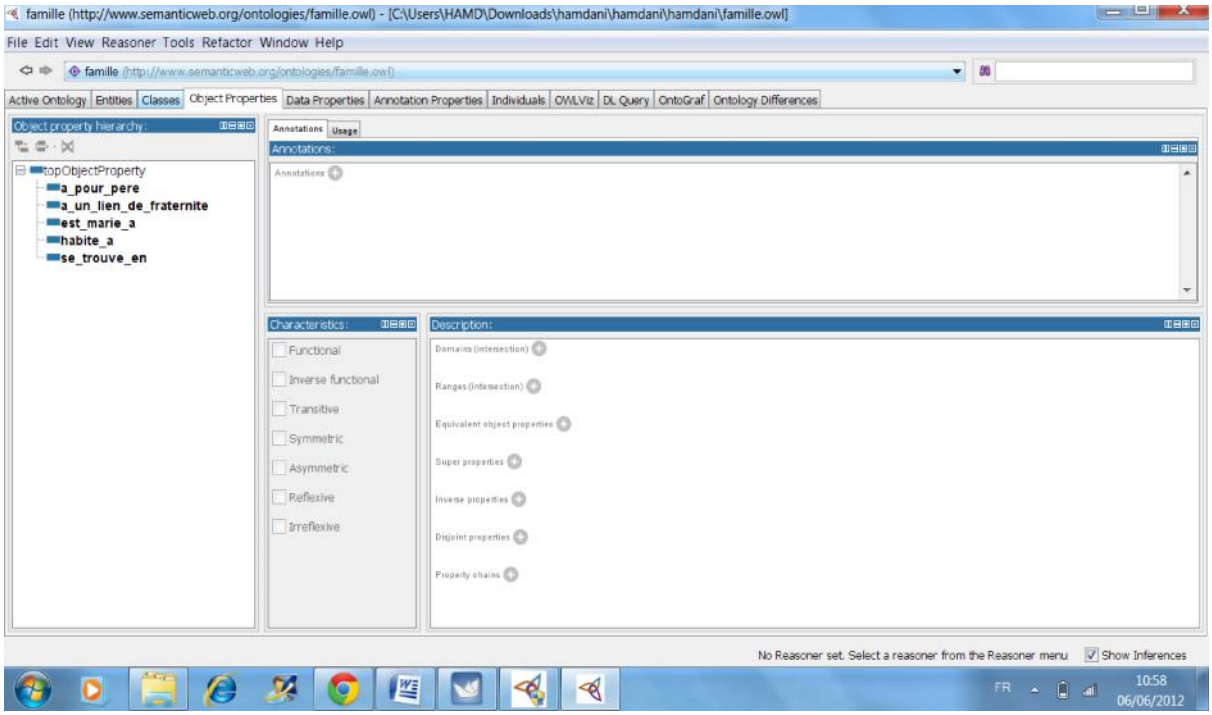


fig. 3.3.1.2.presentation des propriétés d'objet

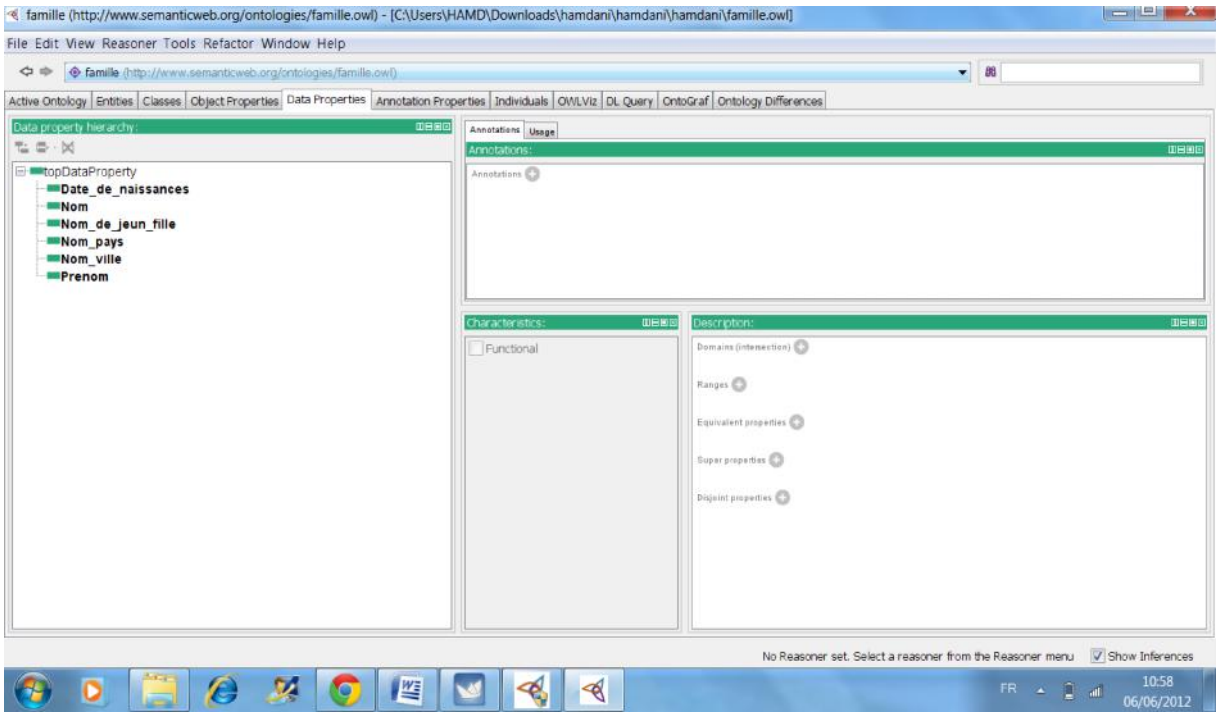


Fig. 3.3.1.3.presentation des propriétés de type de donnée

3.3.2.présentation de ABox :

ABox est définie dans Protégé avec le bouton « individuals » l'image suivante représente les individuels de la classe « Pays » :

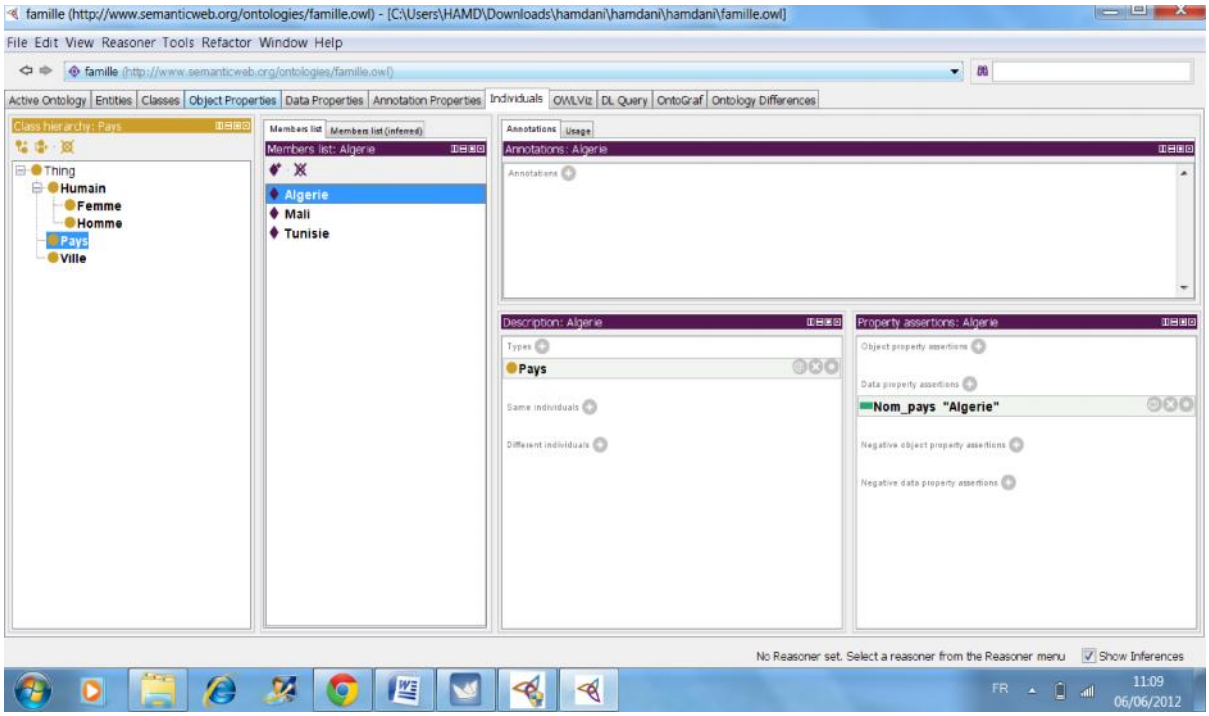


Fig. 3.3.2.1.présentation des des individuels de la classe « pays »

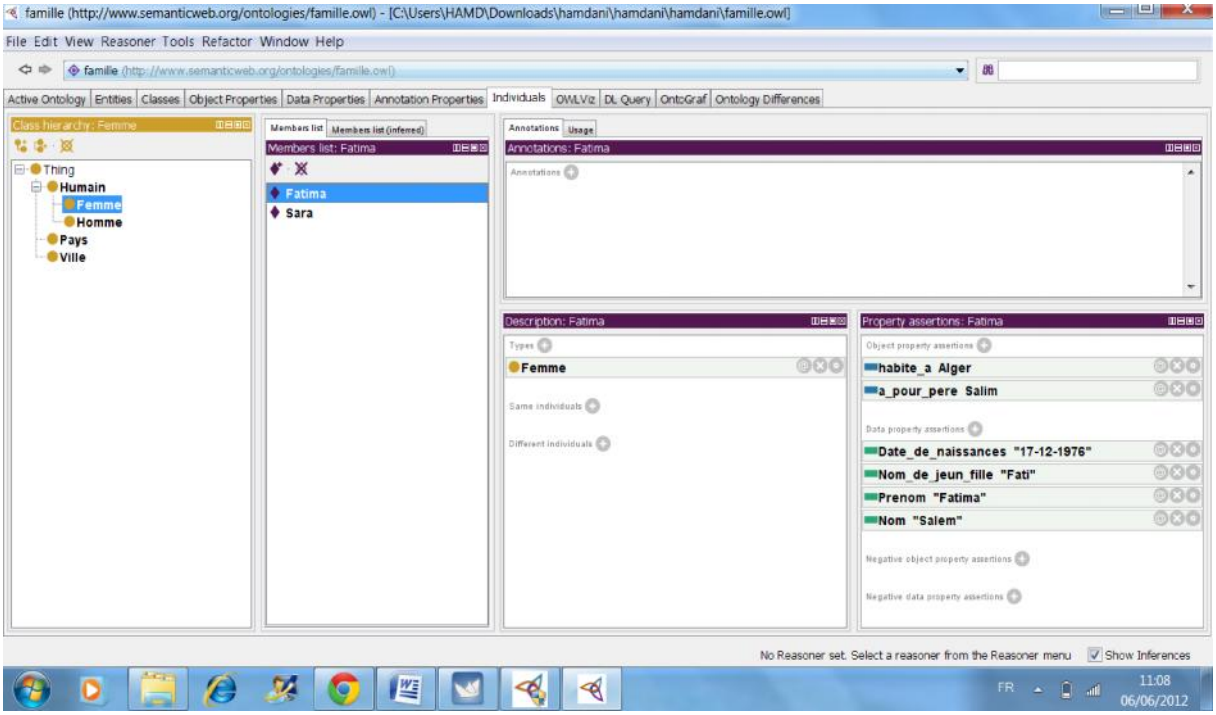


fig. 3.3.2.2.presentation des individuels de la classe « femmes »

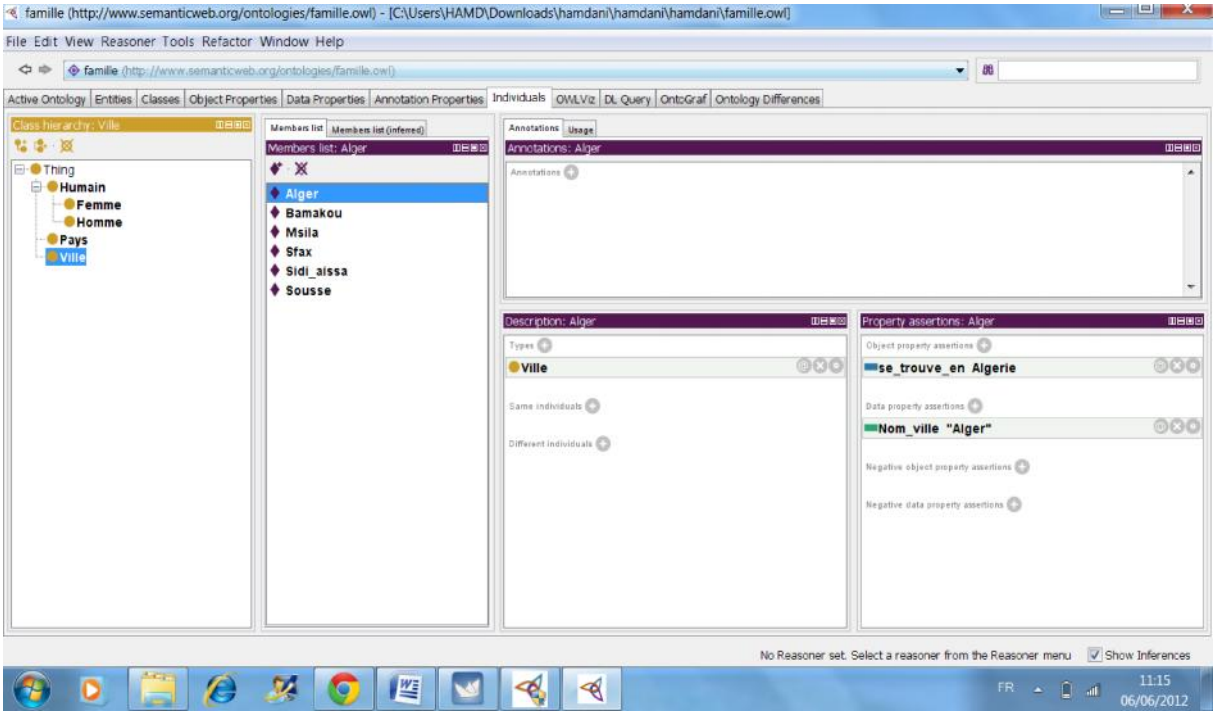


fig. 3.3.2.3.presentation des individuels de la classe « villes »

l'image suivante représente le graphe dessiné avec « ontoGraph » de protégé :

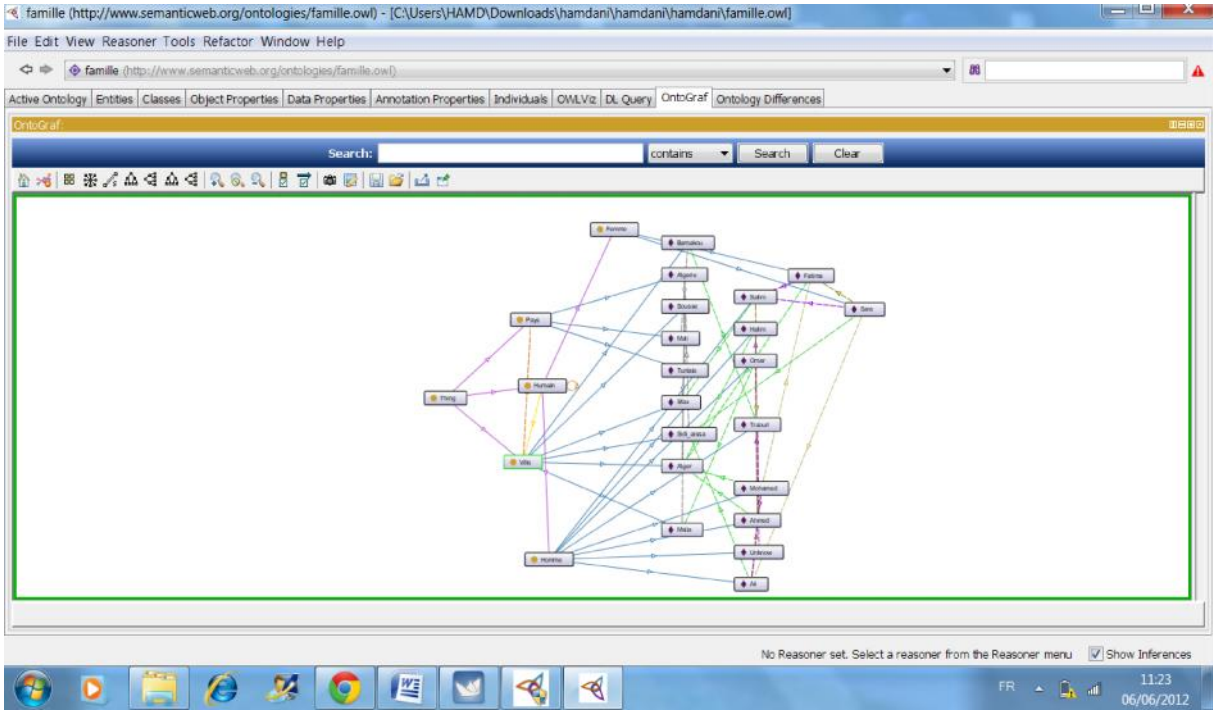


Fig. 3.3. Présentation de l'ontologie avec ontoGraf

4. Présentation des interface du mon travail:

Notre travail est présenté avec une interface Delphi

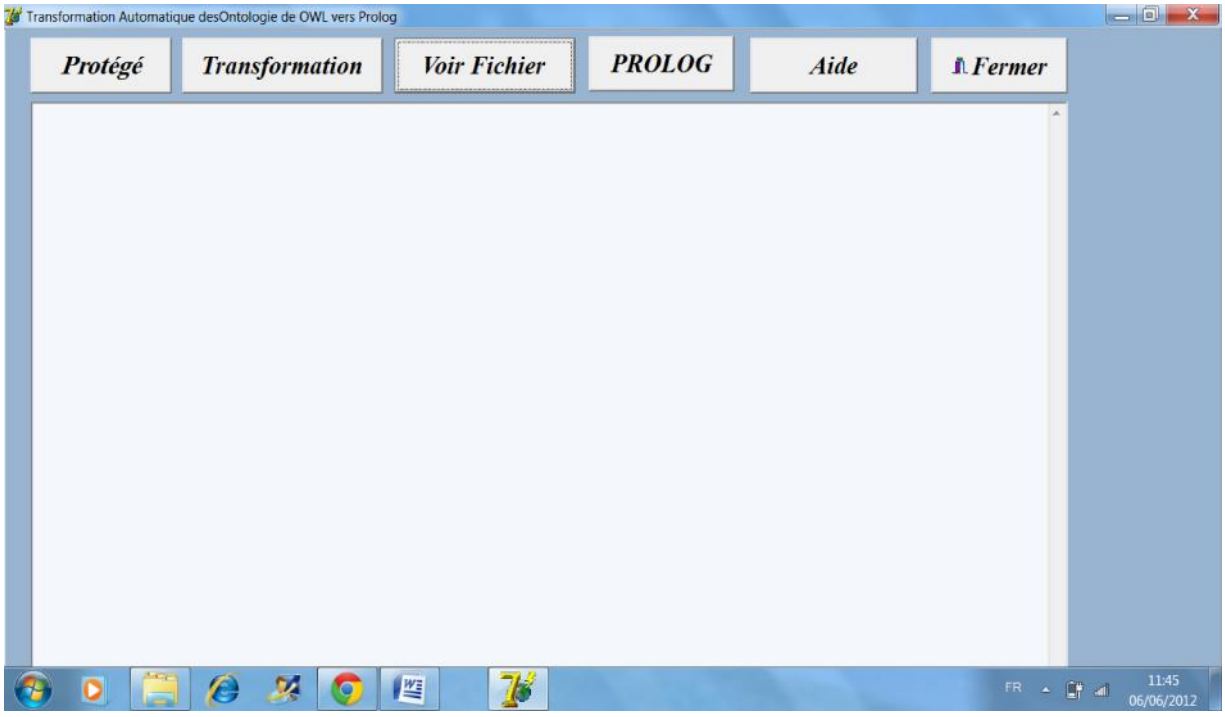


Fig. 4. L'interface globale

Ici nous voulons expliquer les fonctions de chaque bouton

4.1. Bouton Protégé : permet d'ouvrir l'éditeur pour créer une nouvelle ontologie ou ouvrir une ontologie existante.

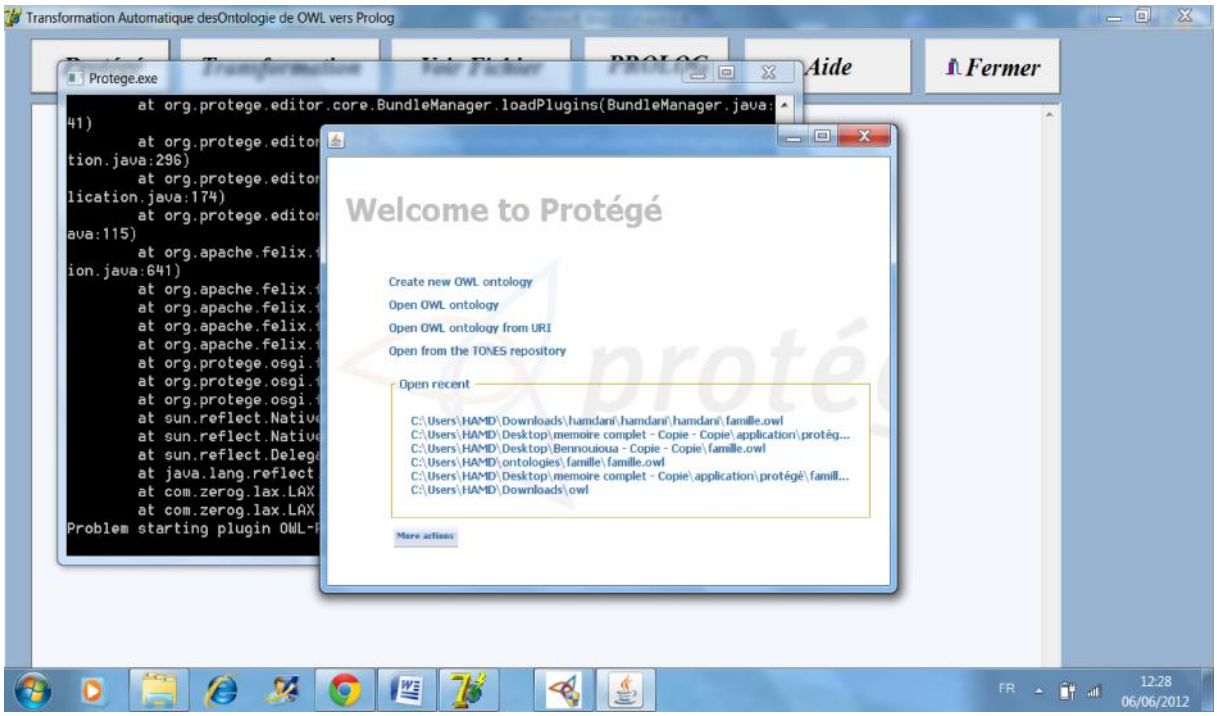


Fig. 4.1. L'interface de Protégé

4.2. Bouton Transformation :ouvrit une boite de dialogue pour choisir l'ontologie OWL à transformer

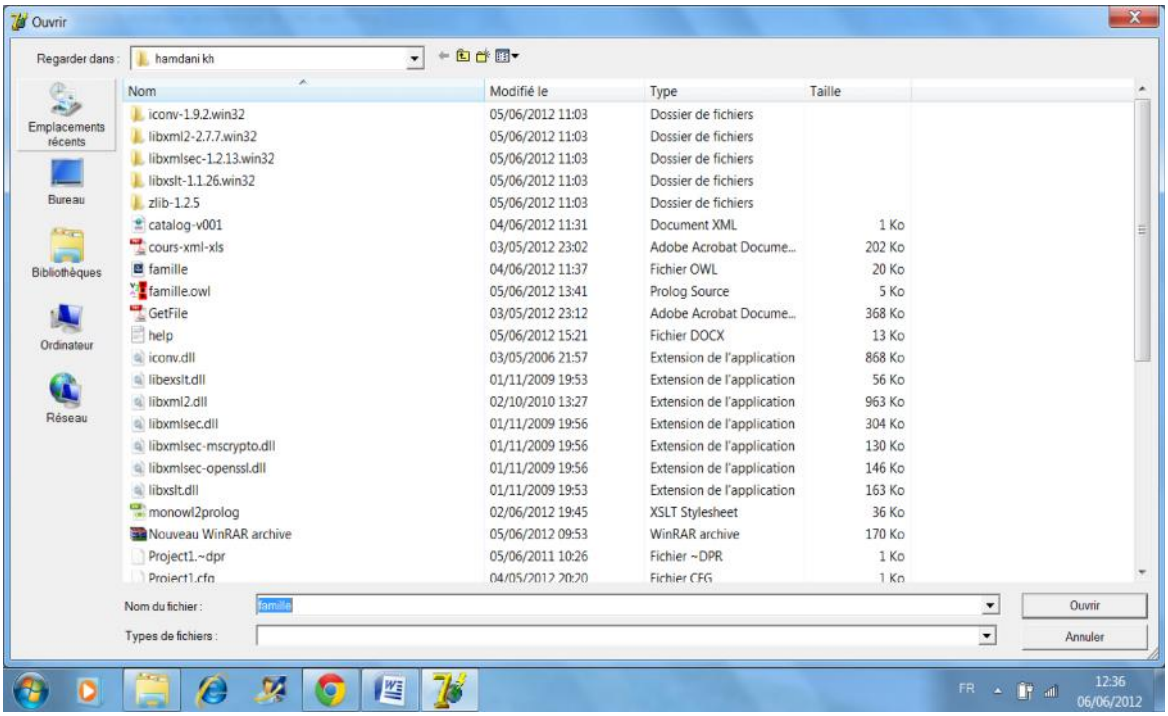


Fig. 4.2. boite de dialogue

4.3.Bouton voir fichier : la même fonction avec Transformer mais pour voir le contenu de fichier selectionné.

fig. 4.3.la fenêtre de voir fichier

4.4. Bouton PROLOG :ouvrir l'interface de PROLOG.

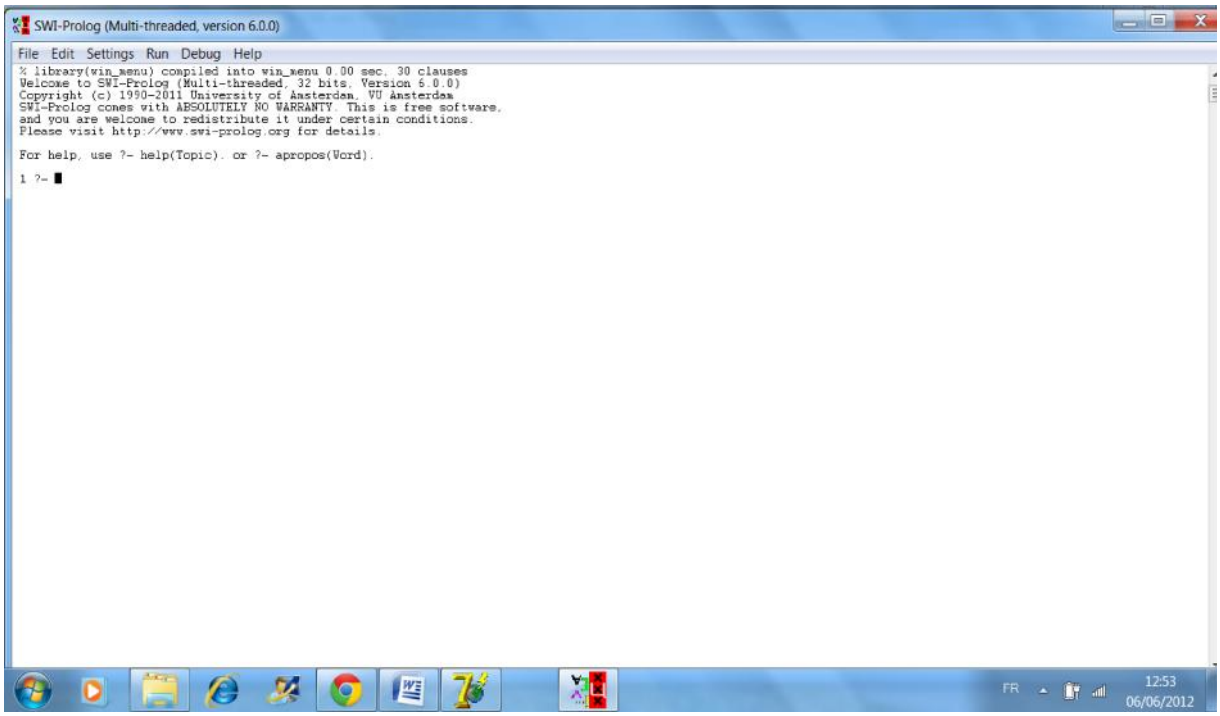


Fig. 4.4.l'interface de PROLOG

4.5.Bouton Aide : ouvrir un fichier WordPad expliquer comment utiliser notre programme

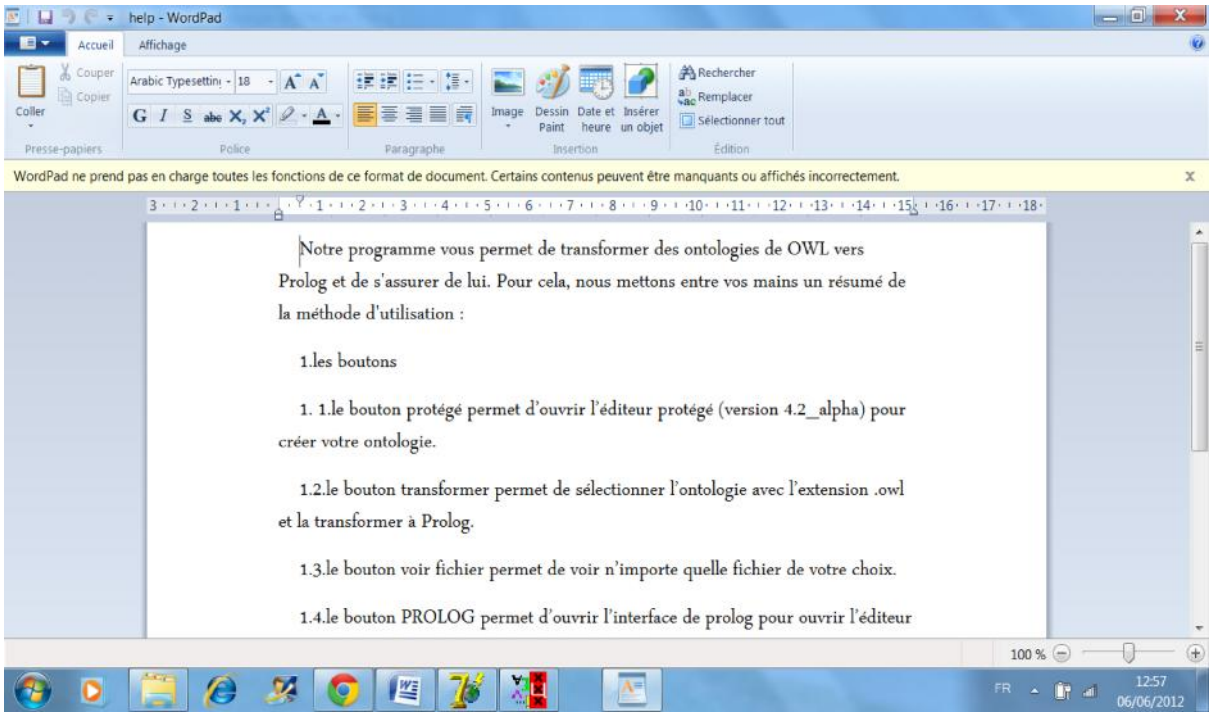


Fig. 4.5.la fenêtre d'Aide

4.6. Bouton Fermer :fermer le programme.

Dans ce chapitre nous avons présenté le travail et le testé .

Conclusion :

Conclusion générale

Dans ce mémoire nous avons présenté un travail pour la transformation automatique des ontologies décrites avec OWL vers le langage de programmation logique Prolog.

Nous avons codé des règles de transformation avec le langage XSL (eXtensible Stylesheet Language). Le processus de transformation utilise l'outil XSLTProc, c'est un outil de compilation et transformation.

Nous avons développé un outil d'interface avec Delphi pour intégrer tous les outils nécessaires pour notre application de transformation. Cette transformation est testée sur un exemple d'ontologie avec un programme Prolog nommé raisonneur pour la vérification de certaines propriétés.

Dans ce mémoire nous avons proposé un programme pour transformer les ontologies décrites avec la version 4.2_alpha de protégé et nous souhaitons de créer un moteur d'inférence avec PROLOG et le généraliser pour d'autre versions.

Bibliographie :

[1] J. CHAUMOND, Social commerce : Quand le e-commerce rencontre le Web d'aujourd'hui, Digital Mammouth Editions, Paris, France, 2012 .

[2] D. AUTISSIER, F. BENSBAA et F. BOUDIER, L'atlas du management: l'encyclopédie du management en 100 dossier-clés, Groupe Eyrolles, Paris, France, 2010.

[3] P. LUONG, Gestion de l'évolution d'un Web sémantique d'entreprise, l'Ecole des Mines ED 84 : Sciences et Technologies de l'Information et de la Communication, Paris, France, 2007.

[4]R. Neches, RE. Fikes, T. Finin, TR. Gruber, T. Senator, WR. Swartout, Enabling technology for knowledge sharing. AI Magazine 12(3) , 1991.

[5]. T. R. Gruber, A translation approach to portable ontology specification, Knowledge Acquisition ,5 (2), 1993.

[6]. WN. Borst.: Construction of Engineering Ontologies. Centre for Telematica and Information Technology, University of Tweenty. Enschede, The Netherlands, 1997.

[7]. R. Studer, VR. Benjamins, D. Fensel, Knowledge Engineering: Principles and Methods ,25(1-2), 1998.

[8]. N. Guarino, P. Giaretta.: Ontologies and Knowledge Bases: Towards a Terminological Clarification, Mars N edition, University of Twente, Enschede,The Netherlands, IOS Press, Amsterdam, The Netherlands, 1995.

[9] N. Aussenac-Gilles, B. Biébow, N. Szulman, Revisiting Ontology Design: a method based on corpus analysis, R. Dieng, & O. Corby edition, 12th International Conference in Knowledge Engineering and Knowledge Management (EKAW'00), Berlin, Germany, 2000.

[10] A. Bernaras, I. Laresgoiti, J. Corera, Building and reusing ontologies for electrical network applications, W. Wahlster edition, European Conference on Artificial Intelligence , Chichester, United Kingdom: John Wiley and Sons, 1996.

[11]B. Swartout, P. Ramesh, K. Knight, T. Russ, Toward Distributed Use of Large- Scale Ontologies, A. Farquhar, M. Gruninger, A. Gómez-Pérez, M. Uschold, V. van der et P edition AAAI'97 Spring Symposium on Ontological Engineering. Stanford University, California, 1997.

[13] Natalya F. Noy et Deborah L. McGuinness, `Ontology Development 101: A Guide to Creating Your First Ontology". Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical, 2001.

[12] A. Gómez-Pérez, M. Fernández-López, O. Corcho, Ontological Engineering (with

examples from the areas of Knowledge Management, e-Commerce and the Semantic Web), Springer, 2004.

[14]S. KHALFI, Construction d'une ontologie pour la prise en charge des patients à domicile, Université Mentouri Constantine Faculté des Sciences de l'Ingénieur Département d'Informatique, 2009.

[15] A. Gómez-Pérez, M. Fernández-López, O. Corcho, Ontological Engineering, Springer-Verlag London, 2004.

[16] O. Lassila, D. McGuinness, The Role of Frame-Based Representation on the Semantic Web, Knowledge Systems Laboratory, Stanford University, Stanford, California, 2001.

[17] G. Van Heijst, Ath. Schreiber, BJ. Wielinga, Using explicit ontologies in KBS development, International Journal of Human-Computer Studies, 1997.

[18] N. Guarino, Formal Ontology in Information Systems, Proceedings of FOIS'98, Trento, Italy, 1998.

[19] N. CHERGUI, Une approche de mapping pour l'intégration des ontologies, Ecole Doctorale de l'Est en Informatique Département d'Informatique Université Mentouri de Constantine, 2008.

[20] X. LACOT, Introduction à OWL un langage XML d'ontologies Web, 2005.

[21] S. Bechhofer, I. Horrocks, C. Goble, R. Stevens, OilEd: a reasonable ontology editor for the Semantic Web, Joint German/Austrian conference on Artificial Intelligence (KI'01) Springer-Verlag, 2001.

[22] J. C. Arpírez, O. Corcho, M. Fernández-López, A. Gómez-Pérez, WebODE in a nutshell, 2003.

[23] Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, OntoEdit: Collaborative Ontology Engineering for the Semantic Web, I. Horrocks, & J. A. Hendler (Éd.), First International Semantic Web Conference (ISWC'02) , Springer-Verlag, 2002.

[24] M. LUDOVIC, Programmation fonctionnelle et logique Programmation (Dossier sur PROLOG), 2002 .

[25]http://www.laltruiste.com/document.php?url=http://www.laltruiste.com/coursxsl(18/05/2012.)

[26] http://www.commentcamarche.net/contents/xml/xmlxslt.php3. (18/05/2012.)

[27] http://www.commentcamarche.net/contents/xml/xmltransform.php3. (18/05/2012.)

Annexe 1 :

Les règles de transformation

Soient, C, D des concepts, r, g des relations et ont le nom de l'ontologie.

Définition d'une classe

*owl:Class***.** en OWL permet de définir les classes, pour transformer ce code vers le prolog on peut définir le prédicat suivante: *owl_Class*

OWL	**Prolog**
<owl:Class rdf:ID="C" />	owl_Class(C).

Définition d'une relation, rôle, propriété

Owl:ObjectProperty. En OWL permet de définir les rôles entre les concepts (entre les individus de concepts), pour transformer ce code vers le prolog on peut définir les règles suivantes:

Ont_C(X):- Ont_r(X, _).le domaine et

Ont_D(X):- Ont_r(_, X).le range.

OWL	**Prolog**
<owl:ObjectProperty rdf:ID="#r"> <rdfs:domain rdf:resource="#C"/> <rdfs:range rdf:resource="#D"/> </owl:ObjectProperty>	Ont_C(X):- Ont_r(X, _). Ont_D(X):- Ont_r(_, X).

Définition d'une propriété de données

owl:DatatypeProperty. En OWL permet de relier des individus à des valeurs de données. Pour transformer ce code vers le prolog on peut définir les règles suivantes :

Ont_C(X):- Ont_r(X, _). le domaine et xsd_positiveInteger(X) :- Ont_r(_, X). le range.

OWL	**Prolog**

<owl:DatatypeProperty rdf:ID="#r"> <rdfs:domain rdf:resource="#C"/> <rdfs:range rdf:resource= "&xsd;positiveInteger"/> </owl:DatatypeProperty>	Ont_C(X):- Ont_r(X, _). xsd_positiveInteger(X):- Ont_r(_, X).

Rôles inverses

owl:inverseof. En OWL permet de définir nouvelle rôle l'inverse de rôle existe, pour transformer ce code vers le prolog on peut définir les règles suivante:

Soit g un rôle inverse de r alors

ont_r(X, Y) :- ont_g(Y, X).

ont_g(X, Y) :- ont_r(Y, X).

OWL	**Prolog**
<owl:ObjectProperty rdf:ID="#g"> <owl:inverseOf rdf:resource="#r"/>	ont_r(X, Y):-ont_g(Y, X). ont_g(X, Y):-ont_r(Y, X).

La subsomption entre les rôles

owl:subpropriétéof. En *OWL* permet de définir des sous propriétés d'autre propriété (hiérarchie entre les propriétés), pour transformer ce code vers le prolog on peut définir la règle suivante:

Soit g sous propriété de r :

ont_r(X, Y) :- ont_g(X, Y).

OWL	**Prolog**
<owl:ObjectProperty rdf:ID="#g"> <rdfs:subPropertyOf rdf:resource="#r"/>	ont_r(X, Y) :- ont_g(X, Y).

La subsomption entre les classes

owl:suclassof. En OWL permet de définir des sous classes d'autre classe (hiérarchie entre les classes), pour transformer ce code vers le prolog on peut définir la règle suivante:

soit C sous propriété de D alors

Owl_Class(C):- Owl_Class(D).

OWL	**Prolog**
<owl:ObjectProperty rdf:ID="#C"> <rdfs:subPropertyOf rdf:resource="#D"/>	ont_D(X) :- ont_C(X).

Quantification universelle

owl:Restriction avec *owl:allValuesFrom.* En OWL permet de définir une classe anonyme composée de toutes les instances de classe D qui satisfont tous les propriétés avec les instances de la classe C. pour transformer ce code vers le prolog on peut définir la règle suivante:

ont_D(X) :- ont_r(X,_),forall(ont_r(X,Y) , ont_C(Y)).

OWL	**Prolog**
<owl:Class rdf:about="#D"> <owl:Restriction> <owl:onProperty rdf:resource="#r"/> <owl:allValuesFrom rdf:resource="#C"/> </owl:Restriction> </owl:Class>	ont_D(X) :- ont_r(X,_), forall(ont_r(X,Y) , ont_C(Y)).

Quantification existentielle

owl:Restriction avec *owl:sameValuesFrom.* En OWL permet de définir une classe anonyme composée de toutes les instances de classe D qui satisfont au moins une propriété avec les instances de la classe C. pour transformer ce code vers le prolog on peut définir la règle suivante:

ont_D(X) :- ont_r(X,_), not(forall(ont_r(X,Y), not(ont_C(Y)))).

OWL	**Prolog**
<owl:Class rdf:about="#D"> <owl:Restriction> <owl:onProperty rdf:resource="#r"/> <owl: someValuesFrom rdf:resource="#C"/> </owl:Restriction> </owl:Class>	ont_D(X) :- ont_r(X,_), forall(ont_r(X,Y) , ont_C(Y)).

Propriété symétrique

rdf:type rdf:resource="&owl;SymmetricProperty". En OWL permet de définir les propriété symetrics. si une telle relation lie un individu c à un individu d, alors la même relation lie également l'individu d à l'individu c, pour transformer ce code vers le prolog on peut définir la règles suivante:

ont_r(X,Y) :- call_with_depth_limit(ont_r(Y,X),1,R), R==2.

OWL	**Prolog**
<owl:ObjectProperty rdf:ID="r"> <rdf:type rdf:resource="&owl;SymmetricProperty" /> <rdfs:domain rdf:resource="#C" /> <rdfs:range rdf:resource="#D" /> </owl:ObjectProperty>	ont_r(X, Y) :- call_with_depth_limit(ont_r(Y,X),1,R), R==2.

Propriété transitive

rdf:type rdf:resource="&owl;transivitéProperty". En OWL permet de définir les propriété transitives. si une telle relation lie un individu c à un individu d et une relation lie un individu

e, alors la même relation lie également l'individu c à l'individu e, pour transformer ce code vers le prolog on peut définir la règles suivante:

ont_r(X,Z) :- call_with_depth_limit((ont_r(X,Y), ont_r(Y,Z)),1,R), R==2.

OWL	**Prolog**
<owl:ObjectProperty rdf:ID="r"> <rdf:type rdf:resource="&owl;TransitiveProperty" /> <rdfs:domain rdf:resource="#C" /> <rdfs:range rdf:resource="#D" /> </owl:ObjectProperty>	ont_r(X,Z) :- call_with_depth_limit((ont_r(X,Y), ont_r(Y,Z)),1,R), R==2.

Classes disjointes

owl:disjointWith. En OWL permet d'assure que tous les individus n'appartient pas à deux classes aux même temps, pour transformer ce code vers le prolog on peut définir la règle suivante:

owl_NoThing(X):- ont_C(X);ont_D(X).

OWL	**Prolog**
<owl:Class rdf:about="#C"> <owl:disjointWith rdf:resource="#D"/> </owl:Class>	owl_NoThing(X):- ont_C(X);ont_D(X).

Classes équivalentes

owl:equivalentClass. En OWL permet de déclarer que deux classes désignent la même classe, pour transformer ce code vers le prolog n peut définir les règles suivantes:

ont_C(X) :- ont_D(X).

ont_D(X) :- ont_C(X).

OWL	**Prolog**

<owl:Class rdf:ID="C"> <owl:equivalentClass rdf:resource="#D"/> </owl:Class>	ont_C(X) :- ont_D(X). ont_D(X) :- ont_C(X).

Complément de

Soit C,D des concepts

owl:complementOf . en OWL permet que C fait partie du complément de l'ensemble D. Ainsi, une entité ne peut être à la fois un C et D pour transformer ce code on eu définir les règles suivantes: ont_C(X) :- not(ont_D(X)).

ont_D(X) :- not(ont_C(X)).

OWL	**Prolog**
<owl:Class rdf:ID="C"> <owl:complementOf rdf:resource="#D" /> </owl:Class>	ont_C(X) :- not(ont_D(X)). ont_D(X) :- not(ont_C(X)).

SameAs, differentFrom

*owl:sameAs***.** En OWL permet de déclarer que les noms des individus désignant la même personne, aussi pour *owl:differentFrom* OWL permet de déclarer que les noms des individus désignant la différente personne, pour transformer ces codes on peut definir les ègles suivantes:

sameIndividuals(X,Y).

differentIndividuals(X,Y)

OWL	**Prolog**
<owl:sameAs rdf:resource="#i" /> <owl:differentFrom rdf:resource="#i"/>	sameIndividuals(X,Y). differentIndividuals(X,Y)

Cardinalité minimale

owl:minCardinality. En OWL permet de définir au moins n éléments dans le Co-domaine de r, pour transformer ce code vers prolog on peut définir la règle suivante:

ont_E(X) :- findall(X, ont_r(_,X), B), sort(B,L),length(L,N), N >= 1,ont_r(_,X).

OWL	**Prolog**
<owl:Class rdf:ID="E"> <rdfs:subClassOf> <owl:Restriction> <owl:onProperty rdf:resource="#r" /> <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger"> 1</owl:minCardinality> </owl:Restriction> </rdfs:subClassOf> </owl:Class>	ont_E(X) :- findall(X, ont_r(_,X), B), sort(B,L), length(L,N), N >= 1,ont_r(_,X).

Cardinalité maximale

owl:maxCardinality. *En* OWL permet de définir au plus n éléments dans le Co-domaine de r, pour transformer ce code vers l prolog on peut définir la règle suivante:

ont_E(X) :- findall(X, ont_r(_,X), B), sort(B,L),length(L,N), N <= 1,ont_r(_,X).

OWL	**Prolog**
<owl:Class rdf:ID="E"> <rdfs:subClassOf> <owl:Restriction> <owl:onProperty rdf:resource="#r" /> <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger"> 1</owl:maxCardinality>	ont_E(X) :- findall(X, ont_r(_,X), B), sort(B,L), length(L,N), N <= 1,ont_r(_,X).

</owl:Restriction> </rdfs:subClassOf> </owl:Class>	

Propriété fonctionnelle

owl;FunctionalProperty, En OWL permet de définir les propriétés fonctionnels, pour transformer ce code vers l prolog on peut définir la règle suivante:

sameIndividuals(Y,Z):- r(X,Y), r(X,Z).

OWL	**Prolog**
<rdf:type rdf:resource="&owl;FunctionalProperty"/>	sameIndividuals(Y,Z):- r(X,Y), r(X,Z).

Annexe 2 :

Vous cherchez un Prolog ? Je vous conseille SWI Prolog (sous Linux ou Windows), disponible gratuitement (licence GPL) au département informatique de l'Université de Psychologie d'Amsterdam (http://www.swi-prolog.org/)

Mais il en existe d'autres, en particulier GNU Prolog (par l'INRIA). Prolog cherche à prouver que le but (goal) demandé est vrai. Pour cela, il analyse les règles, et considère comme faux tout ce qu'il n'a pas pu prouver. Quand le but contient une variable libre, Prolog la liera à toutes les possibilités :

?- est_un_homme(X).

X=marc

X=jean

?- est_le_pere_de(Pere,jean).

Pere=marc

?- est_le_mari(Pere,Mere), est_le_pere_de(Pere,jean).

Pere=marc, Mere=anne

On peut combiner les buts à l'aide des opérations logiques et (,), ou (;) et non (not).

Le but défini ci-dessus nous permet de trouver la mère de jean. Mais on peut également créer une règle en rajoutant :

```
est_la_mere_de(Mere,Enfant) :-
    est_le_mari(Pere,Mere),
    est_le_pere_de(Pere,Enfant).
```

Désormais le but : est_la_mere_de(Mere,jean) (réponse : Mere=anne) permet de rechercher la mère d'un individu, mais la même règle permet de répondre également à une recherche d'enfants : est_la_mere_de(anne,X) (réponse : X=jean). On peut aussi vérifier une affirmation (est_la_mere_de(anne,jean)) ou afficher tous les couples de solutions à la requête : est_la_mere_de(M,X). On ne trouvera ici qu'une solution, pour véritablement tester ceci il faut augmenter notre base de connaissances.

Annexe 3 :

Tim Berners-Lee : de son nom complet Sir Timothy John Berners-Lee né le 8 juin 1955 à Londres est un citoyen britannique surtout connu comme le principal inventeur du World Wide Web.Depuis 1994, il préside le World Wide Web Consortium (W3C), organisme qu'il a fondé.

W3C : (world wide web consortium) est une organisation non lucrative permettant définir des standards pour les technologies liées aux web.

RDF : Resource Description Framework. C'est une norme W3C pour décrire des syntaxes.

الملخص:

ان التحقق اليدوي من الانتولوجيا OWL **بلستخدام** PROLOG **صعب** لأنه ينبغي ان نحولها **أولا إلى** PROLOG, و لتسهيل هذا التحقق يجب ان نجعله هو و التحويل آليين .لهذا الغرض اقترحنا برنامج من اجل التحويل الأوتوماتيكي للأنتولوجيات من لغة OWL **الى لغة** PROLOG.

الكلمات المفتاح: PROLOG OWLProlog, xslt, ontologie, OWL

RESUME :

la verification manuel des ontologies OWL avec PROLOG est tres difficile parceque nous devons avant tout les transformer vers PROLOG, et pour faciliter cette verification il faut la automatiser et automatiser la transformation. nous proposons un programme pour la transformation automatique des ontologie de OWL vers PROLOG .

Mot clé : Prolog, xslt, ontologie, OWL

ABSTRACT :

the manual verification of OWL ontologies with PROLOG is very difficult because we must first transform them into PROLOG, and to facilitate this verification must make it and the transformation automatic. we propose a program for the automatic transformation of ontologies OWL into PROLOG.

Key word: Prolog, XSLT,ontology,OWL.