REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE MOHAMED BOUDIAF - M'SILA

N° d'ordre : ….....

N° de Série : .....

Faculté des Mathématiques et de l'Informatique

Département d'Informatique

THÈSE

Présentée pour obtenir le diplôme de

DOCTORAT EN SCIENCES EN INFORMATIQUE

Option : Informatique

Par

BOUDIA Malika

THÈME

UNE APPROCHE MDA (Model Driven Architecture) POUR LA TRANSFORMATION DES ONTOLOGIES

Soutenue publiquement le : 19 Janvier 2023

Devant le jury composé de :

AKHROUF.Samir	Professeur	Université de M'sila	Président
BOURAHLA Mustapha	Professeur	Université de M'sila	Rapporteur
BOUAMAMA Salim	Professeur	Université de Sétif	Examinateur
MOUHOUB Nassereddine	MCA	Université de M'sila	Examinateur
NOUIOUA Farid	MCA	Université de BBA	Examinateur
ZOUACHE Djaafar	MCA	Université de BBA	Examinateur

الإهداء

الأم وطن .....الى روح أمي وذاكرة أمي ورائحة أمي ونظرة أمي وصوت أمي ودفء أمي وكل أمي

سلام علي روحك الطاهرة وأسكنك الله الفردوس الأعلى من الجنة يا أماه.....

الى أبي المبجل..... أطال الله في عمره، وأمدّه بالصحة والعافية.

الى روح أخي التوأم "العربي صالح" ...

الى أمي بعد أمي أختي الكبرى فتيحة وعائلتها...

الى روح أبي الثاني" بوجملين عبد المجيد" رحمه الله

الى من يحملون دمي ولقبي الى إخوتي وأخواتي وأسرهم ...

الى كل من علمني حرفا فصرت له عبدا ....

الى أرواح شهداء ثورة نوفمبر المجيدة والجزائر" وَلَا تَحْسَبَنَّ ٱلَّذِينَ قُتِلُوا۟ فِى سَبِيلِ ٱللَّهِ أَمْوَٰتًۢا ۚ بَلْ أَحْيَآءٌ عِندَ رَبِّهِمْ يُرْزَقُونَ" سورة آل عمران - الآية 169.

الى من جمعتنا بهم المواقف الإنسانية والصداقات الحقيقية ..... الصداقة ليست شيئاً كبيراً، الصداقة ملايين الأشياء الصغيرة"باولو كويلو".

الى وطني المفدى الجزائر بلد المليون ونصف المليون شهيد .....

بوديعة وديعة مليكة

Remerciements

Les anges baissent leurs ailes vers le chercheur de connaissance, en signe d'approbation de ce qu'il fait. Je remercie Dieu, le Tout Miséricordieux, le Très Miséricordieux, qui m'a donné la patience, la certitude et le courage d'assister à ce jour, le jour d'achèvement de la rédaction de cette thèse, et de la soutenir dans les meilleures conditions.

Comme toute thèse cette recherche a été ponctuée de nombreux moments d'enthousiasme et de joie, mais également de nombreuses périodes de doute et de découragement. L'achèvement de ce travail n'aurait pas été possible sans la précieuse contribution de nombreuses personnes que je veux remercier ici.

Je souhaiterais tout d'abord adresser mes sincères remerciements à Monsieur ***Mustapha Bourahla****, Professeur à l'Université de Mohamed Boudiaf M'sila pour la confiance qu'il m'a accordée en acceptant de diriger cette thèse, puis pour m'avoir guidée, encouragée et conseillée tout au long de ce travail de recherche. Je le remercie infiniment pour m'avoir fait bénéficier de sa grande compétence, de sa rigueur intellectuelle et de ses précieux conseils.*

Je le remercie sincèrement pour sa disponibilité, sa persévérance, et ses encouragements continus, ainsi que pour ses qualités humaines d'écoute et son soutien moral pendant les moments difficiles.

Je remercie également Mr ***Akhrouf Samir*** *Professeur à l'Université de Mohamed Boudiaf M'sila, d'avoir accepté de présider le jury de cette thèse.*

Je remercie Mr ***Salim Bouamama****, Professeur à l'Université Ferhat Abbas Sétif 1, et Mr* ***Mouhoub Nassereddine****, Maître de conférences à l'Université de Mohamed Boudiaf M'sila, et Mr* ***Farid Nouioua****, Maître de conférences Université Mohamed El Bachir El Ibrahimi de Bordj Bou Arréridj, ainsi que Mr* ***Djaafar Zouache*** *Maître de conférences Université Mohamed El Bachir El Ibrahimi de Bordj Bou Arréridj, qui ont eu la gentillesse d'accepter d'examiner et d'évaluer cette thèse. Et pour cela, je leur en suis très reconnaissante.*

Je tiens également à exprimer ma plus vive gratitude à Monsieur ***Messaoud Bendiaf****, Maître de conférences Université Mohamed El Bachir El Ibrahimi de Bordj Bou Arreridj. Il m'a encouragée par ses orientations sans cesser d'être une grande source de motivation et d'enthousiasme. Je le remercie sincèrement pour ses encouragements fraternels.*

Je garde pour la fin un remerciement particulier pour ma sœur, Aicha (Fatiha). Je la remercie du fond du cœur de m'avoir soutenue, encouragée et surtout fait preuve de son affectation.

Pour finir, je souhaiterais dédier ce travail à ma mère. Paix à son âme. Que Dieu lui fasse miséricorde. ***Boudia Malika Wadia***

Never Stop Learning

Because Life

Never Stops Teaching

Emmily Vara.

الملخص

يـتم وضـع النمـاذج فـي مركـز عمليـات التطـوير، إن UML2 هـي اللغـة المتوافقـة مـع OMG التـي تصـف هـذه الأنمـاط والتـي أصـبحت ضـرورية، بالإضـافة إلـى الأنطولوجيـا التـي أصـبحت أساسـية لتطـوير الويـب الـدلالي. ونجـد OWL2 لغـة الوصـف التـي تبناهـا W3C كاللغـة الرئيسـية لتمثيـل المعرفـة ويجـري تطـوير هـذه التكنولوجيـات بشكل مستقل ولكن مع نفس الهدف المتمثل في تصوير العالم الحقيقي.

أصـبحت التسـوية والمقاربـة بـين UML2 وOWL2 مهمـة مشـجعة، ويشـكل هـذا العمـل جـزءا مـن إطـار الهندسـة القائمـة علـى النمـاذج. نحـن نقتـرح أداة لانشـاء نمـاذج لأنمـاط الوجـود وإضـفاء الطـابع الرسـمي عليهـا، وقـد تـم تطـوير محـرر رسـومي لوضـع نمـاذج للأنطولوجيـا فيمـا يتعلـق بـالنموذج التعريفـي لمعـرف بمعيـار Ecore، باسـتخدام إشـارات UML.

تسـمح هـذه الأداة بالتحويـل ثنـائي الاتجـاه للنمـاذج (UML2؛OWL2) التـي تعتبـر واحـدة مـن التقنيـات الواعـدة فـي هـذا الـنهج .والهـدف هـو توليـد رمـز OWL2 مـن نمـاذج مفاهيميـة لأنـواع معينـة مـن المجـالاتUML . لهـذا الغـرض، نسـتخدم التحويـل النموذج نموذج وتحويل النموذج نص (توليد الرمز).

الكلمـات الرئيسـية: الويـب الـدلالي، أنطولوجيـا، OWL2، UML2، تحويـل النمـاذج، تحويـل النمـوذج إلـى نمـوذج، تحويـل النمـوذج إلـى نـص، تحويـل الرسـوم البيانيـة الثلاثيـة، مترجم TGG، توليد رمز.

Résumé :

Les modèles sont situés au centre des processus de développement UML2, le langage standardisé par l'OMG. Ce langage, UML2, décrit ces modèles et ces derniers sont devenus essentiels. En outre, le paradigme de l'ontologie de l'ingénierie est devenu fondamental pour le développement du Web sémantique. Dans ce paradigme, nous trouvons OWL2, le langage de description adopté par le W3C comme le langage principal de la représentation de la connaissance. Les ontologies sont développées indépendamment mais avec le même objectif qui est la conceptualisation du monde réel. La correspondance entre UML 2 et OWL 2 est devenue une tâche satisfaisante. Ces travaux font partie du cadre de l'ingénierie pilotée par les modèles (MOE), et nous tenterons de proposer un outil de modélisation et de formalisation des ontologies. Un éditeur graphique est développé pour modéliser la conceptualisation des ontologies par rapport au méta-modèle défini avec Ecore, en utilisant des notations UML.

Cet outil permet la transformation bidirectionnelle des modèles (UML2, OWL2), qui est considérée comme l'une des techniques prometteuses dans cette approche. L'objectif est de générer du code OWL2 à partir de modèles conceptuels d'ontologies de domaine spécifiques. Pour ce faire, nous utiliserons des transformations modèle-modèle et modèle-texte combinées (génération de code).

Mots-clés : web sémantique, Ontologie, OWL2, UML 2, transformation de modèles, transformation modèle-à-modèle, transformation modèle-à-texte, grammaires de triples graphes, Interpréteur TGG, Génération de code.

Abstract

Models are at the center of development processes. UML2, the language standardized by the OMG, describes these models and has become essential. In addition, the paradigm of engineering ontology has become fundamental for the development of semantic Web, and in this paradigm we find OWL2 which is the language of description adopted by W3C as the main language of the representation of knowledge.

Ontologies are developed independently, but with the same objective which is the conceptualization of the real world. Correspondence between UML 2 and OWL 2 has become a satisfying task. This work is part of the Model Driven Engineering (MOE) framework; and we will propose a tool for modeling and formalizing ontologies. A graphical editor is developed to model the conceptualization of ontologies related to the meta-model defined with Ecore, using UML notations.

This tool allows bidirectional transformation of models (UML2, OWL2), which is considered one of the promising techniques in this approach. The goal is to generate OWL2 code from conceptual models of specific domain ontologies. To do so, we will use combined model-model and model-text transformations (code generation).

Keywords: semantic web, Ontology, OWL2, UML 2, model transformation, model-to-model-transformation, model-to-text-transformation, triple graph grammars, TGG Interpreter, Code generation.

Liste des abréviations

AGG : Attributed Graph Grammar

AGL : Atelier de Génie Logiciel

API : Application Programming Inteface

ATOM3 : A Tool for Multi-formalism and Meta-Modelling

CIM : Computation Independent Model

CSS : Cascading Style Sheets

DOE : Differential Ontology Editor

DSL : Domain Specific Language

DSML : Domain Specific Modeling Languages

EMF : Eclipse Modeling Framework

ETP : Eclipse Tools Project

Fujaba : From UML to Java And Back Again

GMF : Graphical Modeling Framework

GROOVE : GRaph of Object Oriented VErification

GOPRR : Graph, Objects, Property, Role and Relationship

KIF : Knowledge Interchange Format

LDs : Les Logiques de Descriptions

M2M : Model to Model

M2T : Model to Text

MBT : Tests basés sur les modèles

MDA : Model-Driven Architecture

MDD : Développement piloté par les modèles

MDE : Model-Driven Engineering

OAW : Open Architecture Ware

OCL : Object Constraint Language

OIL : Ontology Inference Layer

OMG : Object Management Group

OMT : Object Modeling Technique

OWL : Web Ontology Language

PIM : Platform Independent Model

PDM : Platform Description Model

PSM : Platform Specific Model

RDF : Resource Description Framework

RDFS : Resource Description Framework Schema

RHS : Right Hand Side

SWT : Standard Widget Toolkit

SWRL : Semantic Web Rule Language

TGG : Triple Graph Grammars

UML : Unified Modeling Language

UMLS : Unified Medical Language System

URIs : Universal Resources Identifiers

XMI : XML Metadata Interchange

XML : Extensible Markup Language

XPDL : XML Process Definition Language

W3C : World Wide Web Consortium

Table des matières

Liste des figures

Liste des tableaux

CHAPITRE 1

Introduction générale

Sommaire

1.1 Contexte

L'Ingénierie Dirigée par les Modèles (IDM), ou Model Driven Engineering (MDE) en anglais, s'inscrit dans l'évolution des techniques pour le développement de systèmes informatiques afin d'en maîtriser leurs complexités, en se concentrant sur une préoccupation plus abstraite que la programmation classique. En s'appuyant sur des approches génératives, il s'agit de générer tout ou partie d'une application à partir de modèles. Un modèle est une abstraction, une simplification d'un système qui est nécessaire et suffisante pour comprendre un aspect particulier du système modélisé et permet de répondre aux questions que soulève cet aspect du système. Un système peut être décrit par différents modèles liés les uns aux autres, et exprimé chacun à l'aide d'un langage de modélisation dédié (Domain Specific Modeling Languages - DSML). Le principe est d'utiliser autant de langages de modélisation différents que les aspects chronologiques ou technologiques du développement du système le nécessitent. L'activité consistant à définir ces DSML (la syntaxe et la sémantique), appelée méta-modélisation, est donc une problématique clé de l'IDM. En outre, les autres problématiques clés de l'IDM consistent à rendre les modèles construits opérationnels (pour la simulation, la génération de code, de documentation ou de test, la validation, la vérification, l'exécution, etc.) à l'aide de composition et de transformation de modèle.

Les modèles sont placés par la modélisation au centre de processus de développement. Ces modèles sont décrits par des langages, à l'instar d'UML qui est le langage standardisé par l'OMG, devenu incontournable. En plus, le paradigme de l'ingénierie d'ontologies est devenu nécessaire pour le développement du Web sémantique, et dans ce paradigme on trouve OWL, qui est 2 le langage de description adopté par W3C comme le principal langage de représentation de connaissances. Les modèles et les ontologies sont développés indépendamment mais dans le même objectif : la conceptualisation du monde réel.

Les ontologies décrivent les ressources Web sous la forme d'un modèle basé sur des graphes et est formalisée dans le langage OWL (Web Ontology Language), qui est basé sur RDF (Resource Description Framework). Le modèle basé sur les graphes peut être sérialisé en un ensemble de triplets en utilisant l'une des différentes syntaxes comme XML, Turtle, ou les formats fonctionnels. Un triplet est composé d'un sujet, d'un prédicat et d'un objet. Le sujet est une ressource Web (document), l'objet peut être une ressource Web (document) ou un littéral, et le predicat. Le prédicat est une propriété (relation) OWL ou RDF (RDFS) qui relie le sujet à l'objet. Le modèle OWL/RDF contient la terminologie et les assertions de l'ontologie sous la forme d'un ensemble de triplets RDF. Ainsi, avec ce modèle OWL/RDF, nous pouvons déclarer des axiomes, comme les axiomes d'inclusion de concepts, et les axiomes d'appartenance. Une ontologie est généralement composée de :

- l'ensemble terminologique (TBOX) : nous définirons des axiomes par les concepts généraux et les inclusions de rôles. Les rôles (relations) peuvent être entre paires de concepts (appelés propriétés d'objet) ou entre concepts et type de données (appelés propriétés de type de données).

- l'ensemble des assertions (ABOX) qui sont soit des instances de concepts (axiomes d'appartenance aux concepts) pour affirmer qu'un individu (objet) appartient à un concept, soit des instances de propriétés d'objets (axiomes d'appartenance aux propriétés d'objets) pour affirmer qu'un individu a une relation avec un autre individu, soit enfin des instances de propriétés de types de données (axiomes d'appartenance aux propriétés de types de données) pour affirmer qu'un individu a une propriété de données.

Le principal domaine d'application des ontologies est le Web sémantique, où les documents Web sont annotés avec des informations (méta-données) issues de la terminologie ontologique.

Les modèles UML et les ontologies sont deux représentations de la connaissance avec des forces et des faiblesses différentes. Jusqu'à récemment, ils étaient considérés comme des domaines de recherche sans lien entre eux. Cependant, des études portant sur leurs paradigmes sous-jacents, et les approches combinant les deux émergent de plus en plus.

UML2 est un langage de modélisation qui permet de spécifier, visualiser, construire, et documenter les artefacts des systèmes logiciels, de même que pour la modélisation d'entreprise et des systèmes non logiciels, selon l'OMG (OMG 2003). Au niveau de Unifed Modeling Language, deux éléments importants sont à noter. Le terme "unifed" et le terme "langage". Le premier terme signifie que les auteurs ont essayé de regrouper les éléments importants des concepts objets, alors

que la deuxième montre qu'il s'agit d'un langage de modélisation et non pas d'une méthode. UML est un langage qui permet de modéliser non seulement des applications informatiques ou des structures de données, mais également les activités d'un domaine : mécanique, biologie, processus métier (AUDIBERT 12 janvier 2009) (Xavier et Mounier 28 September,2006) (Barbier 07/11/2005). Ce langage de modélisation unifié repose sur deux concepts essentiels :

- La modélisation du monde réel au moyen de l'approche orientée objet.
- L'élaboration d'une série de diagrammes facilitant l'analyse et la conception des systèmes, et permettant de représenter les aspects statiques et dynamiques du domaine à modéliser ou à informatiser.

Le rapprochement et la correspondance entre UML et OWL est apparu à plusieurs égards tant pour les classes que pour les associations. Ce travail est inscrit dans le cadre de l'ingénierie dirigée par les modèles (IDM), en utilisant la transformation de modèles qui est considérée comme l'une des techniques prometteuses dans cette approche.

Dans (Kleppe, Warmer et Bast : 2003) on trouve pour les transformations de modèles les définitions suivantes :

- « Une transformation est la génération automatique d'un modèle cible à partir d'un modèle source, suivant une définition de la transformation » ;
- « Une définition de transformation est un ensemble de règles de transformation qui, réunies, décrivent comment un modèle dans un langage source peut être transformé dans un langage cible » ;
- « Une règle de transformation est une description de la manière dont une ou plusieurs structures du langage source peuvent être transformées en une ou plusieurs structures du langage cible ». Mens et Gorp généralisent cette définition en considérant qu'une transformation peut avoir plusieurs modèles source et cible. Ils présentent dans leur état de l'art (Mens et Van Gorp March 2006) une classification des différentes transformations de modèles selon les critères suivants :

  - **Similarité des espaces techniques** : l'espace technique correspond aux langages de représentations supportés pour représenter les modèles en machine. On peut citer par exemple XMI (OMG, XML Metadata Interchange 2007) défini par l'OMG.

- **Transformation endogène versus transformation exogène** : une transformation endogène est une transformation dont les modèles source et cible ont le même métamodèle tandis qu'ils sont différents pour une transformation exogène.

- **Transformation horizontale versus transformation verticale** : une transformation horizontale est une transformation où les modèles source et cible sont au même niveau d'abstraction, tandis qu'ils sont différents dans les transformations verticales. Le niveau d'abstraction dépend du niveau de détails dans le modèle, mais ne dépend pas forcément du méta-modèle, ainsi une transformation endogène, telle qu'un raffinement de modèles, peut être considérée comme verticale.

- **Transformation syntaxique versus transformation sémantique** : une transformation syntaxique se contente de modifier la syntaxe de représentation des modèles tandis qu'une transformation plus complexe est considérée comme sémantique. On considère notamment le passage d'une syntaxe concrète vers une syntaxe abstraite comme une transformation syntaxique. Les transformations peuvent être écrites dans différents langages.

- **Des langages généralistes** auxquels on ajoute un framework ou des librairies permettant la manipulation des modèles, par exemple EMF (Steinberg, et al. 2008)Avec JAVA ou FAME (Kuhn et Verwaest 2008) qui est disponible dans plusieurs langages.

- **Des langages spécialisés** tels que KERMETA (Muller, Fleurey et Jézéquel 2005) qui est un langage orienté objet, ou ATL (Bézivin, et al. January 2003) qui est un langage déclaratif ou pour finir (Csertan, et al. 2002) qui est un langage basé sur les transformations de graphes.

1.2 Problématique

Le modèle se concentre sur le processus de développement par la modélisation. Ces modèles sont écrits dans des langages comme UML, un langage standardisé par OMG, ce qui est essentiel. De plus, le développement du Web sémantique nécessite le paradigme de l'ingénierie des ontologies. Ce paradigme inclut OWL, un langage descriptif adopté par le W3C comme principal langage de représentation des connaissances. Bien que les modèles et les ontologies soient développés indépendamment, leur objectif est le même, celui de conceptualiser le monde réel. La relation entre UML et OWL se présente de plusieurs manières, incluant les classes et les associations. Ce travail fait partie d'un cadre d'ingénierie pilotée par les modèles (MDE) qui utilise

des transformations de modèles. Ceci est considéré comme l'une des techniques prometteuses proposées par MDA (Model Driven Architecture).

Les modèles UML et les ontologies sont deux représentations de la connaissance avec des forces et des faiblesses différentes. Jusqu'à récemment, ils étaient considérés comme des domaines de recherche sans lien entre eux. Cependant, des études portant sur leurs paradigmes sous-jacents et les approches combinant les deux émergent de plus en plus. Néanmoins, l'état de la recherche sur la relation entre les deux est toujours en cours d'exploration.

Les transformations de modèles basées sur TGG (Triple Graph Grammar) relèvent d'un mécanisme standard qui propose des règles, pour transformer respectivement les modèles de diagramme de classes vers l'ensemble terminologique (TBOX), et les diagrammes objet vers l'ensemble des assertions (ABOX). Nous générerons ainsi le code OWL2 équivalent de manière automatique.

Les quatre axes qui sont : l'ingénierie dirigée par les modèles, le langage de modélisation graphique UML2, la sémantisation du web, et les ontologies, sont ici tous réunis pour générer la problématique principale de cette thèse :

- **Comment assurer la correspondance et la cohérence entre les diagrammes UML et Ontologies afin de se convertir (se transformer) entre elles et surtout converger vers les ontologies OWL2 (le web Sémantique) afin de générer le code OWL 2 équivalent.**

L'utilisation de l'IDM pour effectuer les transformations bidirectionnelles nécessite la définition d'un ensemble cohérent et complet d'artéfacts (règles méthodologiques, transformations de modèles, génération automatique de code) préalablement testés, outillés et évalués. Il reste néanmoins que l'ingénierie dirigée par les modèles présente un inconvénient mineur qui consiste à produire un code généré moins performant qu'un code optimisé directement pour une plateforme cible. L'objectif de tâches présentées dans ce manuscrit est de répondre principalement aux questions suivantes :

- Comment formaliser les diagrammes UML (méta-modélisation)
- Comment formaliser les ontologies (méta-modélisation)

- Comment effectuer une transformation bidirectionnelle entre les diagrammes UML et les ontologies de façon d'encapsuler les connaissances (Transformation bidirectionnelle).

Comment effectuer la correspondance entre les diagrammes UML (diagrammes de classe et les diagrammes objet) et les ontologies (ABOX, TBOX) (Transformation modèle 2 modèle) (les règles de transformation (TGG).

- Comment générer le code OWL2 à partir des ontologies formalisées. (Transformation modèle 2 texte : génération de code)

1.3 Contributions

Dans cette thèse, nous proposons un outil de modélisation et de formalisation d'ontologies (Boudia et Bourahla 2022). Un éditeur graphique est développé pour modéliser la conceptualisation des ontologies par rapport au méta-modèle défini avec le standard Ecore, en utilisant les notations UML. Ecore est le méta-modèle central d'EMF. Il permet d'exprimer d'autres modèles en s'appuyant sur ses constructions. Ecore est défini en fonction de lui-même (son propre méta-modèle). Le résultat de la modélisation sera transformé, en utilisant des grammaires de graphes triples en un modèle d'ontologie par rapport à un méta-modèle d'ontologie OWL2 défini développé avec Ecore. Cette transformation bidirectionnelle peut être réalisée en exécutant un ensemble de règles définies par rapport à un méta-modèle de correspondance entre le méta-modèle ontologique-conceptuel et le méta-modèle ontologique basé sur OWL2.

L'objectif est de générer du code OWL2 à partir de modèles conceptuels d'ontologies de domaine spécifiques. Pour cela, nous utilisons des transformations combinées modèle-modèle et modèle-texte (code) (Boudia et Bourahla 2022). L'ingénierie dirigée par les modèles (MDE), l'architecture dirigée par les modèles (MDA) et les tests basés sur les modèles (MBT) sont des approches dirigées par les modèles pour l'analyse et la transformation des modèles, qui ont été adoptées par divers domaines (par exemple, l'ingénierie d'entreprise et l'ingénierie logicielle).

Selon le contenu expédié dans les modèles, les transformations de modèles peuvent être divisées en deux classes : modèle-à-modèle et modèle-à-texte (code). Il existe deux types de transformations. Le premier type de transformation est la transformation horizontale, qui est une transformation où les modèles source et cible résident au même niveau d'abstraction. La transformation horizontale est divisée en deux types : les transformations endogènes et exogènes. Les transformations endogènes sont des transformations entre des modèles exprimés dans le même langage de modélisation, par exemple, le refactoring. Les transformations exogènes sont des transformations entre des modèles exprimés dans des langages de modélisation différents, par exemple, la migration.

Le deuxième type de transformation est la transformation verticale, qui est une transformation où les modèles source et cible résident à des niveaux d'abstraction différents. Le raffinement est un exemple typique, où une spécification initiale peut être progressivement raffinée par rapport à des règles de raffinement qui ajoutent des détails plus concrets aux raffinements successifs jusqu'à arriver à une implémentation complète.

Pour cela, la génération automatique de descriptions formelles de modèles conceptuels spécifiques à une ontologie, nous a amené à développer deux processus de transformation de modèles.

Le premier processus est la transformation de modèle à modèle (Boudia et Bourahla 2022), qui est une transformation horizontale et endogène, où la méthode TGG est utilisée pour la transformation du modèle. Pour cela, nous définissons trois méta-modèles de base pour les modèles conceptuels spécifiques à l'ontologie (les modèles sources), les modèles OWL2 (les modèles cibles), et les modèles de correspondance pour le mappage des éléments des modèles sources et cibles.

Le deuxième processus est la transformation de modèle en texte (code) (Boudia et Bourahla 2022), qui est une transformation verticale et exogène, où la méthode Xpand (Xpand Documentation, 2020), qui est un langage spécialisé dans la génération de code basé sur des modèles EMF est utilisée pour la transformation de modèle. Pour cela, nous utilisons les modèles OWL2 produits par le premier processus par rapport à son méta-modèle Ecore comme source de cette transformation de modèle. La cible est le code OWL2 (texte) par rapport aux modèles Xpand (représentant son méta-modèle Xpand).

Les résultats de ce processus de recherche sont présentés dans l'article intitulé "**Formalization of Ontology Conceptualizations using Model Transformation**", publié dans le journal international "**Journal international de modélisation et de conception de systèmes d'information (IJISMD)** (Boudia et Bourahla 2022).

BOUDIA, M., & BOURAHLA, M. (2022). Formalization of Ontology Conceptualizations using Model Transformation. International Journal of Information System Modeling and Design (IJISMD)"., Volume 13, issue1.

1.4 Organisation du manuscrit

Ce document est organisé selon le plan suivant :

Chapitre 1 : intitulé « Introduction générale », nous proposerons une introduction générale au thème étudié.

Chapitre 2 : intitulé « MDA et transformation de modèle », nous examinerons plus en détail les principes de l'ingénierie pilotée par les modèles (MDE) et les concepts de base liés à la MDE ; tels que : les systèmes ; les modèles ; les méta-modèles ; les méta-méta-modèles. Plus précisément, nous aborderons les deux activités de base du Model Driven Engineering (MDE) que sont la méta modélisation et la transformation de modèles.

Chapitre 3 : Intitulé « Le Web sémantique et l'ontologie », nous enseignerons les concepts du Web sémantique, son architecture et ses principaux composants. Nous exposerons également la définition, les caractéristiques, les composants des concepts d'ontologie, les étapes de construction d'ontologie, les différentes classifications d'ontologie et leurs méthodes de construction.

Chapitre 4 : intitulé « OWL2 », nous développerons la nouvelle version OWL2, exposerons les éléments de bases de la structure OWL 2 et préciserons leurs profiles. De plus ; nous découvrons les des différentes syntaxes offertes par OWL2.

Chapitre 5 : intitulé « Contributions », nous décrivons l'approche MDA pour la transformation des diagrammes UML vers OWL2 en utilisant le TGG (Triple Graph Grammar).

Chapitre 6 : Intitulé « Étude de cas et évaluation », dans ce chapitre, nous appliquerons l'approche de formalisation d'ontologies pour générer du code OWL 2 équivalent. À cette fin, une description formelle d'un modèle conceptuel spécifique à une ontologie est automatiquement générée (code OWL2). Enfin, nous proposerons une évaluation de cette approche et soulignerons de nouvelles perspectives pour enrichir la contribution de cette thèse.

CHAPITRE 2

MDA et Transformations des modèles

Sommaire

2.1 Introduction

L'ingénierie dirigée par les modèles (IDM), ou Model Driven Engineering (MDE) en anglais, a permis plusieurs améliorations significatives dans le développement de systèmes complexes en permettant de se concentrer sur une préoccupation plus abstraite que la programmation classique.

L'ingénierie Dirigée par les modèles, terme proposé par Kent Stuart (Stuart 2002), est une forme d'ingénierie générative (Kent 2002), qui se caractérise par une démarche rigoureuse par laquelle tout est généré à partir d'un modèle, ce qui fait passer les modèles du statut de contemplatifs à celui de productifs, afin de faire face à la complexité croissante de la conception et de la production d'un logiciel.

L'ingénierie dirigée par les modèles (IDM) est une approche de développement logiciel dont le but est d'élever le niveau d'abstraction et d'augmenter le degré d'automatisation dans le

développement logiciel. Cette forme d'ingénierie générative propose que tout ou partie d'une application et est engendrée à partir de modèles.

Un modèle est une abstraction, une simplification d'un système qui est suffisante pour comprendre le système modélisé et répondre aux questions que l'on se pose sur lui. Un système peut être décrit par différents modèles liés les uns aux autres.

Un méta-modèle est un modèle qui définit le langage pour exprimer un modèle » (KLEPPE , WARMER et BAST 2003). Les modèles écrits dans ce langage sont alors dits conformes au méta-modèle. Un méta-modèle étant aussi un modèle, il est donc conforme à un méta-modèle : le méta-méta-modèle.

Un méta-méta-modèle est le modèle d'un langage permettant d'écrire des langages de modélisation. Il est conforme à lui-même, devenant ainsi le sommet de la pile de méta-modélisation. Ainsi, chaque plateforme de modélisation est basée sur un méta-méta-modèle. On peut citer par exemple MOF et sa version simplifiée EMOF définis par l'OMG (OMG, OMG : Meta object facility (MOF) core specification. 2006), Ecore qui est le méta-méta-modèle d'Eclipse (Budinsky, et al. 2003), ou encore FM3 qui est le méta-méta-modèle du projet FAME (KUHN et VERWAEST 2008) .

L'ingénierie dirigée par les modèles consiste en deux activités : la méta-modélisation et la transformation des modèles.

- La méta-modélisation est l'activité qui consiste à définir le méta-modèle d'un langage de modélisation. Elle vise donc à bien modéliser un langage, qui joue alors le rôle de système à modéliser.
- La transformation de modèles constitue l'activité primordiale de la démarche IDM. En effet, cette activité considère l'automatisation de l'opération de transformation pendant le cycle de développement, qui peut avoir des sémantiques différentes en fonction des utilisations, raffinement, optimisation, génération de code, etc. La transformation de modèles est un processus qui consiste à générer un ou plusieurs modèles cibles conformément à leur méta-modèle à partir d'un ou de plusieurs modèles sources conformément à leur méta-modèle. Elle est qualifiée d'endogène si les modèles sources et cibles sont conformes au même méta-modèle (sinon elle est dite exogène et elle se fait entre deux méta-modèles différents).

Dans ce chapitre nous allons étudier les principes et les concepts de base de l'ingénierie dirigée par les modèles. Nous présenterons les concepts modèle ; méta- modèle et méta-méta-modèle ou le préfix « méta » désigne un niveau plus élevé d'abstraction.

Nous clarifierons l'activité de la transformation des modèles, leurs classifications ainsi que la suite des outils désignés à la transformation des modèles. Nous inclurons également les travaux liés aux transformations des modèles.

2.2 Les principes généraux de l'IDM

Suite à l'approche objet des années 80 et de son principe du « tout est objet », l'ingénierie du logiciel s'oriente aujourd'hui vers l'ingénierie dirigée par les modèles (IDM) et le principe du « tout est modèle ». Cette nouvelle approche peut être considérée à la fois en continuité et en rupture avec les précédents travaux (Bézivin, 2004) (Bézivin, 2005). Tout d'abord en continuité car c'est la technologie objet qui a déclenché l'évolution vers les modèles. En effet, une fois acquise la conception des systèmes informatiques sous la forme d'objets communicant entre eux, Bezivin s'est posé la question de les classifier en fonction de leurs différentes origines (objets métiers, techniques, etc.), de manière plus radicale que pouvaient l'être les approches des patterns (Gamma, et al., 1995) et des aspects (Kiczales, et al., 1997). L'IDM vise donc à fournir un grand nombre de modèles pour exprimer séparément chacune des préoccupations des utilisateurs, des concepteurs, des architectes, etc. C'est par ce principe de base fondamentalement différent que l'IDM peut être considérée en rupture par rapport aux travaux de l'approche objet.

Alors que l'approche objet est fondée sur deux relations essentielles, « InstanceDe »et « Hérite De », l'IDM est basée sur un autre jeu de concepts et de relations. Le concept central de l'IDM est la notion de modèle pour laquelle il n'existe pas à ce jour de définition universelle.

2.3 Approches de l'ingénierie dirigée par les modèles

Si l'ingénierie dirigée par les modèles peut être considérée comme un domaine qui a émergé avec les technologies liées à l'informatisation des modèles, il existe différentes approches concrétisant différentes façons d'utiliser les modèles dans le cadre de l'ingénierie système. L'approche la plus connue, et peut-être la plus développée, est l'approche appelée Model Driven Architecture (MDA).

2.3.1 Model Driven Architecture (MDA)

L'Architecture Dirigée par les Modèles (Model Driven Architecture – MDA (OMG, 2014) est une approche fournie et soutenue par l'Object Management Group (OMG). Cette approche peut être considérée comme une variante de l'IDM pour le génie logiciel. Son objectif est de faire évoluer les pratiques de conception du logiciel vers une approche centrée sur le modèle et non plus sur le code. Voir figure 2.1

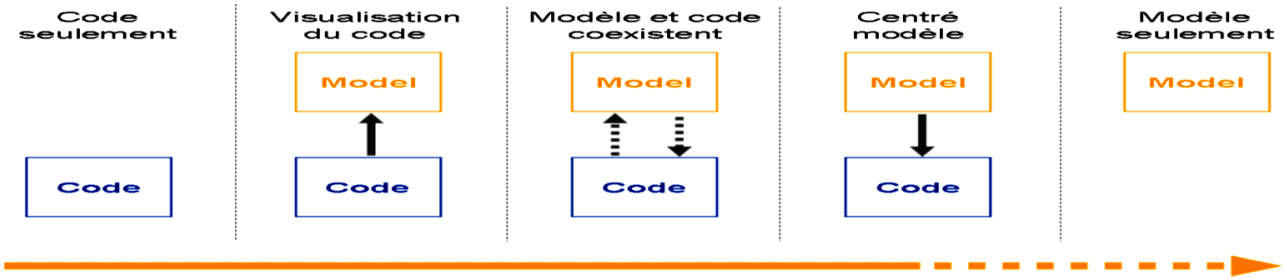


Figure 2. 1 : Spectre des pratiques de conception en génie logiciel (Hardebolle, 2009)**.**

2.3.2 Les standards de l'Architecture Dirigée par les Modèles

L'approche MDA s'appuie sur un ensemble de standards de l'OMG (voir la figure 2.2), dont notamment UML (OMG, 2017) , le MOF (OMG, 2019) et le CWM (OMG, 2003).

- **L'Unified Modeling Language (UML)**, qui a largement inspiré l'approche MDA, est un langage de modélisation à vocation généraliste. De par son historique, c'est un langage orienté objet. Il comprend un ensemble de notations graphiques permettant de représenter un système sous différents points de vue.
- **Le Meta Object Facility (MOF)** définit un langage abstrait et extensible, permettant de décrire, définir et manipuler des méta-modèles. Il a la particularité de s'auto définir. C'est aujourd'hui la pierre angulaire de l'approche MDA.
- **Le Common Warehouse Metamodel (CWM)** définit un framework permettant de décrire des méta-données concernant des sources de données, des transformations de données ainsi que des processus de gestion d'entrepôts de données. Il a pour objectif de faciliter les échanges de méta-données dans le cadre d'environnements distribués et hétérogènes.
- **Le standard XML Metadata Interchange (XMI)** (OMG, 2007) vient compléter ces standards en définissant un format d'échange de méta-données basé sur XML. C'est sur XMI que reposent les techniques de sérialisation de modèles.

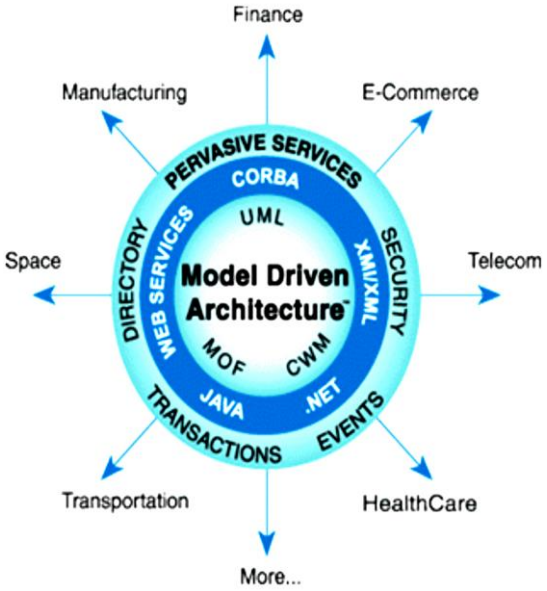


Figure 2. 2 : Les standards de l'Architecture Dirigée par les Modèles (MDA).

Cette sphère de standards n'est pas fermée : au fur et à mesure que l'approche MDA évolue, l'OMG publie de nouveaux standards pour la soutenir.

2.4 Concepts de bases de l'IDM

Dans cette section nous donnerons quelques notions de base sur les Modèles, Méta-modèles, Langages de modélisation, méta-modélisation et transformation de modèles.

2.4.1 Un modèle

Plusieurs définitions sont proposées, et il n'existe pas à ce jour une définition universelle. Nous citerons ci-dessous quelques définitions significatives.

- **"A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system."** (Bézivin, et al., 2001)
- **"A model is a set of statements about some system under study (SUS)."** (Seidwitz, 2003)
- **"A model is an abstraction of a (real or langague based) system allowing predictions or inferences to be made."** (KuhÜne, 2006)
- **Un modèle** est une abstraction d'un système qui devrait être plus simple que celui-ci. Cette simplification se traduit, en général, par la disparition de détails d'ordre technique. Un modèle représente le système qu'il décrit et doit pouvoir être utilisée à sa place pour répondre à un certain nombre de questions sur celui-ci. (Béziven , et al., 2001).
- Pour (Erik Erikson , et al., 2004), **un modèle** est une description abstraite des systèmes exprimés par des diagrammes.

- **Un modèle** : Un modèle est une abstraction de la réalité, qui permet de mieux comprendre le processus de développement d'un système. Il met l'accent sur certains aspects du système et ignore d'autres (c'est un point de vue simplicité du système). Son but est de modéliser et faciliter le traitement du système à utiliser. (Kadima, 2005) (Blanc, 2005).
- **Un modèle** est une description abstraite d'un système ou d'un processus, une représentation simplifiée qui permet de comprendre et simuler. La modélisation n'est pas un problème à solution unique, bien souvent, le même problème analysé par des personnes différentes conduit à des modèles différents. De ce fait, il n'y a donc pas de mauvais modèles, mais en revanche des modèles plus élégants que d'autre. Un modèle capture la sémantique d'un problème et contient des données exploitées par les outils pour l'échange d'information, la génération de code, la navigation, etc. (Muller, et al., 2000)

Malgré les différences de formulation, ces définitions se rejoignent sur certaines propriétés (Vega, 2005) (Kûhne, 2006) (Nguyen, 2008)**.**

- **Représentation** : Il existe un système original qui est l'objet à étudier et que l'on veut représenter par un modèle. Cette caractéristique est fondamentale ; un modèle ne peut exister sans le système qu'il représente.
- **Simplification** : Le modèle donne une vue simplificatrice du système. Certaines caractéristiques du système sont représentées, d'autres non ; tout dépend du but et de l'utilisation du modèle. Un modèle ne représente pas toutes les propriétés du système. Il n'est donc pas une copie du système
- **Pragmatisme** : Un modèle peut remplacer un système pour un propos donné. Les informations obtenues en consultant le modèle devraient être conformes à celles qu'on aurait obtenues en consultant le système. Cette caractéristique assure l'utilité du modèle car elle permet d'obtenir les informations souhaitées plus rapidement et plus simplement qu'en interrogeant le système.

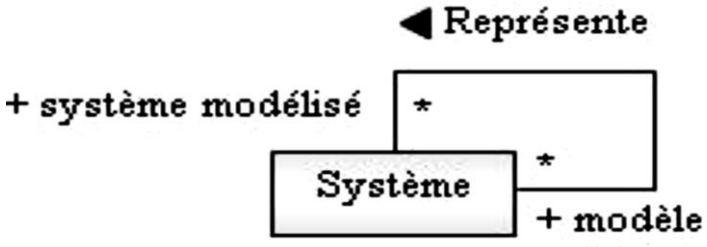


Figure 2. 3 : Modèle et Représente De

Le fait qu'un modèle représente un système introduit une relation Représente entre eux. Il est important de noter que les notions de système original et de modèle sont relatives. (Nguyen, 2008).

Un modèle peut représenter un système, mais aussi et en même temps être un système représenté par un autre modèle. Le modèle et le système étudié ont donc deux rôles complémentaires. Favre a synthétisé les concepts de modèle, système étudié et représente en un diagramme de classe. Consulter la figure 2.3

2.4.1.1 La classification des modèles (Nguyen, 2008)

Les modèles, dans le sens le plus général, représentent un spectre très vaste de systèmes, comme énoncé par Bézivin "**tout est modèle**" (Bézivin, 2005) . Ce slogan vient après le célèbre slogan "tout est objet" de la programmation orienté objet (POO). Dans la POO, le concept de classe est utilisé pour classifier des objets. De la même manière, dans le monde IDM, les modèles sont classifiés selon les langages de modélisation dans lesquels les modèles sont écrits.
Cette classification est ainsi basée sur les formalismes dans lesquels sont exprimés les modèles.

Néanmoins, un modèle peut être classifié en se basant sur d'autres critères, par exemple :

- **La nature du modèle : Physique, abstrait ou numérique** : Favre (Favre, 2004) distingue les systèmes physiques, abstraits ou numériques. Puisqu'un modèle est lui-même est un système, il peut également être physique, abstrait ou numérique. Cette classification, comme l'indique Favre, est relative. Le but est simplement de distinguer les modèles numériques utilisés en informatique, par rapport aux autres.

- **La motivation de modélisation : descriptive ou spécification** : Ce critère de classification est un raffinement de la relation de représentation. Un modèle peut être utilisé pour décrire un système existant ou pour spécifier un système à construire. Dans « What Models Mean » (Seidwitz, 2003) E., Seidewitz a proposé de distinguer les modèles descriptifs et les spécifications.

La différence entre ces deux types est que dans le premier cas, le modèle est construit par l'observation d'un système existant alors que dans le deuxième cas, le système est construit par l'observation d'un modèle (Bézivin, 2005). Une analyse du domaine peut être considérée comme un modèle descriptif, par contre, un cahier de charges des besoins de l'utilisateur est une spécification.

- **La nature de l'évolution du modèle : statique ou dynamique** : Bézivin catégorise les modèles en statique et dynamique selon le changement d'état du modèle au cours du temps. Un modèle est dit statique si son état est constant ; il est dit dynamique si son état évolue. Il utilise cette distinction pour remarquer l'usage répandu en informatique de modèles statiques et de systèmes dynamiques : le modèle en soi ne change pas, mais il représente l'évolution du système modélisé dans le temps (Vega, 2005). A titre d'exemple, le code source d'un programme peut être considéré comme un

modèle statique, représentant le comportement d'un système dynamique réel tel qu'une transaction de virement bancaire.

- **L'usage du modèle : contemplatif ou productif** : Cette distinction différencie les modèles produits par des méthodes de modélisation à la Merise, qui sont en général contemplatives, interprétées par les humains, par opposition aux modèles productifs, outillables, interprétables par des machines dans l'IDM. Un modèle dans l'IDM est non seulement une documentation mais un artéfact central participant au processus de développement logiciel.

- **Leur rôle : produit ou processus :** L'idée de séparer les données à manipuler (produit) de la manipulation (processus) de ces données est très ancienne dans l'ingénierie de logiciel. On peut voir cette idée à plusieurs endroits, par exemple, dans la programmation procédurale où la partie des donnés et des contrôles sont séparés en variables globales et procédures. Cette notion est également adoptée dans CBSE[1] (Composant-Based Software Engineering) où plusieurs systèmes sont développés en procédé/composant. Produit et processus sont donc deux faces d'un système. De ce point de vue, Bézivin (Bézivin, 2005) a fait la différence entre les modèles de produit et les modèles de processus. Dans le développement, le mariage entre ces deux modèles est une des façons de construire un système. De ce fait, on retrouve à nouveau le problème de la composition de modèles.

- **Le niveau d'abstraction** : PIM (Platform Independent Model). Ou PSM (Platform Specific Model)

Les modèles peuvent représenter les systèmes à des degrés d'abstraction différents. Ils peuvent être très abstraits ou très détaillés ; très conceptuels ou très techniques. Pour distinguer les modèles selon les différences de niveaux d'abstraction, le standard MDA introduit les concepts de modèles indépendants de la plate-forme (PIM) et dépendants de la plateforme (PSM).

Cette distinction se fonde sur une l'idée de base du MDA selon laquelle le cycle de développement du logiciel est un processus de transformation progressive des modèles métiers, au niveau d'abstraction haut, qui ne dépendent d'aucune technologie d'implémentation, vers des modèles plus spécifiques aux plates-formes techniques. Le code exécutable est le résultat final de ce processus de raffinement.

[1] Depuis plusieurs années, l'ingénierie du logiciel tend à s'inspirer du développement du hardware en utilisant l'assemblage de composants logiciels, cette approche est une avancée majeure pour la onstruction de systèmes complexes. Elle vise la construction de systèmes de grande taille par l'assemblage de composants logiciels préfabriqués considérés omme des boîtes noires qui communiquent avec leur environnement par le biais d'interfaces. La construction basée sur l'approche CBSE présente des avantages en termes de modularité, de réutilisation, de coût et de sûreté.

Bien que le principe de MDA soit facile à comprendre, sa réalisation n'est pas évidente.

Cela vient de l'ambiguïté de la définition du concept de plate-forme. Cette notion est assez vague et très dépendante du contexte, ce qui rend les notions de PIM et PSM controversées.

Malgré cela, l'idée de raffinement des modèles jusqu'au code exécutable du MDA est intéressante, et offre une nouvelle conception du développement logiciel.

D'autres critères de classification peuvent être trouvés dans (Bézivin, et al., 2003). Notons que, parmi les critères déjà abordés, les trois premiers concernent tous les modèles, alors que les trois derniers sont plutôt destinés aux modèles informatiques.

2.4.1.2 Typologies des modèles (Hardebolle, 2009)

Typologie des modèles dans l'approche MDA

L'OMG a défini une typologie de modèles, ainsi qu'un ensemble de relations de transformation qui permettent de passer de l'un à l'autre. Les quatre principaux types de modèles définis dans l'approche MDA sont les suivants : (Bézivin , et al., 2002).

- **CIM (Computation Independant Model)** Aussi appelé modèle de domaine ou modèle métier, le CIM capture les exigences en termes de besoins et décrit la situation dans laquelle le système sera utilisé. Son but est d'aider à la compréhension du problème mais aussi de fixer un vocabulaire commun pour un domaine particulier. Dans la pratique, l'appellation « CIM » est très peu utilisée.

- **PIM (Platform Independant Model)** Le PIM décrit le système indépendamment de la plate-forme cible sur laquelle il s'exécutera. Il présente donc une vue fonctionnelle détaillée du système, sans détails techniques. Il peut être raffiné progressivement jusqu'à intégrer des détails d'architecture spécifiques à un type de plate-forme (machine virtuelle, système d'exploitation, etc.) mais il doit rester technologiquement neutre.

- **PDM (Platform Description Model)** Le PDM est le modèle qui décrit une plate-forme d'exécution. Il fournit un ensemble de concepts techniques représentant les différentes parties de la plate-forme et/ou les services qu'elle fournit. Un PDM peut représenter, par exemple, des plates-formes à base de composants comme CCM (OMG, 2006) ou EJB.

- **PSM (Platform Specific Model)** Le PSM est le résultat de la combinaison du PIM et du PDM. Il représente une vue technique détaillée du système. Il peut exister avec différents niveaux de détails. Dans sa forme la plus détaillée, il sert de base à la génération de l'implémentation.

2.4.1.3 Passages entre les modèles (Bézivin, et al., 2002)

Le principe de l'approche MDA consiste à passer des PIMs aux PSMs pour préparer et faciliter la génération de code vers une plate-forme technique choisie. Ce passage des PIMs aux PSMs est une transformation de modèles. Le MDA identifie plusieurs types de transformations (voir figure 2.4) :

- **PIM vers PIM** : Ces transformations s'effectuent pour ajouter ou soustraire des informations aux modèles. Le fait de masquer par exemple quelques éléments afin de s'abstraire de détails fonctionnels est typiquement une transformation PIM vers PIM. Dans l'autre sens, le passage du problème à la solution est la plus naturelle des transformations PIM vers PIM. Il est important de noter que ces transformations ne sont pas toujours automatisables.

- **PIM vers PSM** : Ces transformations s'effectuent lorsque les PIMs sont suffisamment complets pour pouvoir être "immergés" dans une plate-forme technique. L'opération, qui consiste à ajouter des informations propres à une plate-forme technique pour permettre la génération de code, est une transformation PIM vers PSM. A l'heure actuelle, les plates-formes techniques visées sont DotNet, J2EE, XMLet CORBA. Il apparaît clairement que ce sont les règles qui permettent ces transformations qui sont importantes, et qui doivent être généralisées et capitalisées. Ces transformations ont donc pour but d'être fortement automatisées.

- **PSM vers PSM** : Ces transformations s'effectuent lors des phases de déploiement, d'optimisation ou de reconfiguration. Notons par ailleurs qu'une unique transformation PIM vers PSM n'est pas toujours suffisante pour permettre la génération de code, et qu' il faudra alors parfois transformer les PSM en PSM en utilisant des formalismes intermédiaires (exemple de passage d'UML à SDL puis de SDL à C++).

- **PSM vers PIM** : Ces transformations s'effectuent pour construire des PIMs à partir d'un existant. Même quand le patrimoine applicatif peut être représenté sous forme de PSM, ces transformations sont actuellement assez difficiles à établir. Elles sont pourtant nécessaires à considérer dans toute stratégie de migration.

Il est important de noter que la génération de code n'est pas toujours considérée comme une transformation de modèles même si certains font la distinction entre PSM exécutables (le code) et PSM non exécutables.

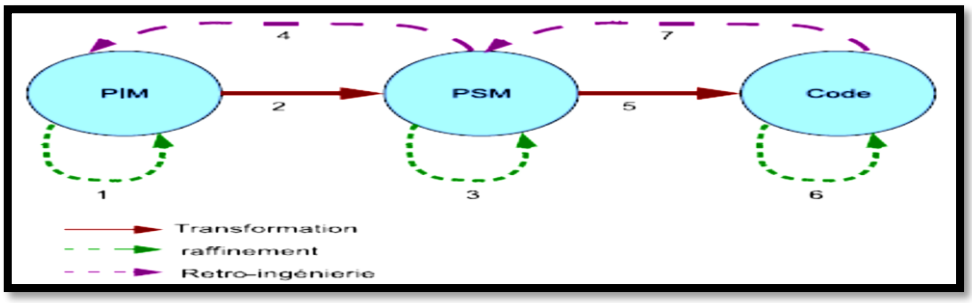


Figure 2. 4 : Les différentes opérations sur les modèles dans le MDA

Il est important de noter que la génération de code n'est pas toujours considérée comme une transformation de modèles même si certains font la distinction entre PSM exécutables (le code) et PSM non exécutables. En résumé, il est possible de classer les passages de modèles possibles dans le MDA dans quatre catégories comme l'indique la figure 2.4 :

- Les transformations (2) : décrivent le processus de conversion d'un PIM en un PSM,
- Les raffinements (1), (3) et (6) : introduisent ou suppriment des informations dans un modèle,
- Les retro-ingénieries (4) et (7) : convertissent un modèle vers un niveau d'abstraction plus élevé,
- La génération de code (5) : transforme un PSM non exécutable en un code exécutable. Nous avons déjà mentionné que la génération de code n'est pas toujours considérée comme une transformation de modèles dans la pratique (Bendiaf, Spécification et vérification des systèmesembarqués temps réel en utilisant la logique de réécriture 2017).

2.4.2 Un méta modèle

Un méta-modèle : Un méta-modèle est un modèle qui permet de définir le langage d'expression ou la structure d'un modèle. Autrement dit, la méta-modélisation modélise les entités d'un système, le lien existant entre un modèle et le système qu'il représente avec les contraintes existantes, c'est-à-dire qu'un méta-modèle est une spécification de la syntaxe et la sémantique d'un système. Pour illustrer la notion de méta-modèle on peut citer l'exemple suivant : un programme source Java est un modèle pour toutes ses exécutions possibles. Le méta-modèle d'un programme source Java est la grammaire de Java. La grammaire définit un ensemble de programmes syntaxiquement valides. Un programme source conforme à la grammaire appartient à cet ensemble (Blanc, 2005) (Kerkouche , 2011).

Selon (Piel, 2007), un métamodèle est un langage de modélisation. Il définit les concepts ainsi que les relations entre concepts nécessaires à l'expression de modèles. La notion de métamodèle conduit à l'identification d'une seconde relation, liant le modèle et le langage utilisé

pour le construire, appelée « conforme A ». Ces deux relations permettent ainsi de bien distinguer le langage qui joue le rôle de système, du (ou des) métamodèle(s) qui jouent le rôle de modèle(s) de ce langage.

Selon Beugnard (Beugnard, et al., 2015), pour décrire les situations de modélisation, nous avons deux types d'artéfacts à considérer : les modèles et les méta-modèles. Les situations varient selon l'ordre dans lequel ces artéfacts apparaissent, ou sont reliés dans la démarche. Nous simplifions la description en ne considérant qu'un seul acteur dans chaque situation. Une analyse plus fine de ces situations pourrait être intéressante en prenant en compte différents acteurs et donc différentes intentions dans le processus. Les figures 2.5, 2.6 et 2. 7 présentent les 12 situations considérées, une flèche indique le ou les éléments qui apparaissent. Nous utiliserons aussi les mots 'instance' pour décrire ce qui se trouve au niveau modèle et 'concept' pour ce qui est au niveau méta-modèle. Chacune des situations est décrite en détail et illustrée dans une des sous-sections suivantes par un exemple concret.

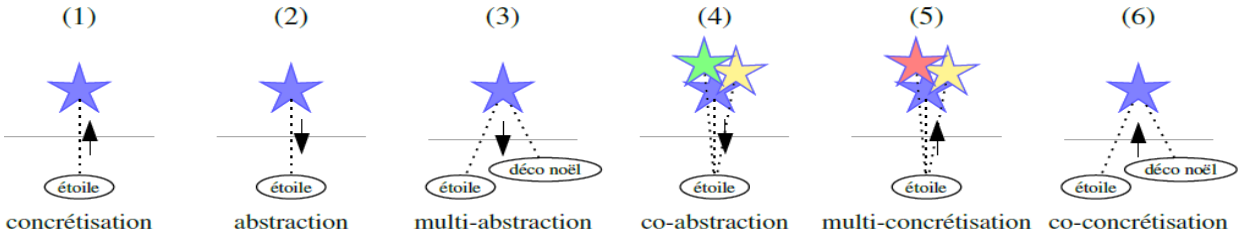


Figure 2. 5 : Des situations de modélisation (haut : niveau modèle, et bas : niveau méta-modèle)

(1) Un méta-modèle existe, le travail consiste à produire un modèle [concrétisation].
(2) Un modèle existe, le travail consiste à trouver un méta-modèle [abstraction].
(3) Un modèle existe, il faut trouver plusieurs méta-modèles [multi-abstraction].
(4) Des modèles existent, le travail consiste à trouver un méta-modèle [coabstraction].
(5) Un méta-modèle existe, le travail consiste à construire plusieurs modèles [multi-concrétisation].
(6) Des méta-modèles existent, le travail consiste à construire un modèle [coconcrétisation].
(7) Un modèle et un méta-modèle existent, il faut les relier [interprétation].

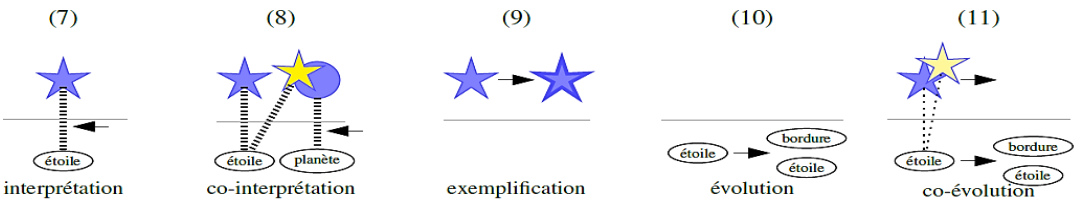


Figure 2. 6 : Des situations de modélisation (haut : niveau modèle, et bas : niveau méta-modèle).

(8) Des modèles existent, des méta-modèles existent, le travail consiste à les relier [co-interprétation].
(9) Un modèle existe, le travail consiste à construire un autre modèle (sans aucun méta-modèle) [exemplification/extension].
(10) Un méta-modèle existe, le travail consiste à construire un autre méta-modèle (sans aucun modèle) [évolution/extension].
(11) Un méta-modèle existe avec plusieurs de ses modèles conformes, le travail consiste à faire évoluer le méta-modèle (cas précédent) en adaptant (ou non) ses modèles [co-évolution].

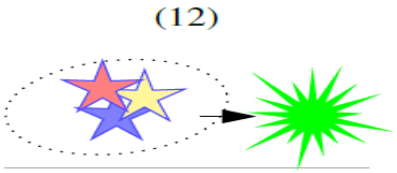


Figure 2. 7: Une situation de regroupement (niveau modèle)

(12) Un ensemble de modèles est regroupé pour former un tout [regroupement].

2.4.3 Un Méta-Méta-modèle :

Un méta-méta modèle est un langage de méta modélisation utilisé pour exprimer le méta modèle. C'est le cas par exemple pour la proposition de MDA (Model Driven Architecture) (FAVRE , et al., 2006) (Diaw, et al., 2010).

Dans l'IDM, formalisme de modélisation et méta formalisme sont appelés respectivement méta modèle et méta-méta modèle. Ainsi chaque modèle est exprimé dans le formalisme de son méta modèle, qui lui-même est définit par son méta modèle, autrement dit le méta-méta modèle. Afin de limiter le nombre de niveaux d'abstraction, un méta-méta modèle est conçu avec la propriété de méta-circularité, c'est-à-dire avec la capacité de se décrire lui-même. L'OMG (OMG, OMG : Meta object facility (MOF) core specification. 2006) à défini le méta-méta modèle comme suit :

Définition : Un méta-méta modèle est un modèle qui permet de décrire un langage de méta modélisation. Un méta-méta modèle doit être réflexif pour limiter le nombre de niveaux d'abstraction. Les concepts de système, modèle et méta modèle ainsi que les relations qui les lient, représentent les principes de base sur lesquels s'appuie l'IDM. A cet égard, l'OMG a introduit l'architecture à quatre niveaux illustrés dans la Figure 2.8 appelée également l'architecture 3+1 ou bien la pile de modélisation (Pyramide de modélisation à quatre niveaux). Le monde réel est représenté à la base de la pyramide (niveau M0). Les modèles représentant cette réalité constituent

le niveau M1. Les méta modèles permettant la définition de ces modèles (UML, par exemple) constituent le niveau M2. Enfin, le méta modèle, unique et méta-circulaire, est représenté au sommet de la pyramide (niveau M3) (Combemale, July 2008).

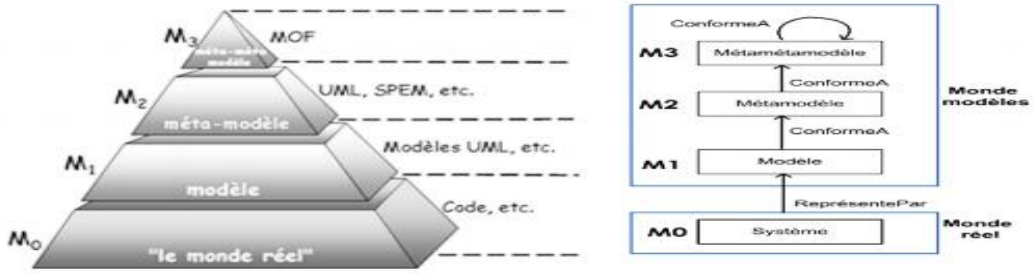


Figure 2. 8 : *Pyramide de modélisation à quatre niveaux*

Un méta-méta-modèle est un modèle d'un langage de modélisation permettant de décrire des méta- modèles. Dans la figure 2.9, l'objectif est la modélisation de figures géométriques imbriquées (le système à modéliser). Une de ces figures est présentée en partie droite. L'espace technique est donc celui des formes qui peuvent être spécialisées et composées. C'est le méta-méta-modèle. Dans cet espace, un méta-modèle permet de décrire des rectangles, dont certains comportent des carrés et d'autres des ronds. Un carré peut posséder un rectangle (par héritage un carré) ou un rond. Ce méta-modèle peut être utilisé pour décrire de nombreuses formes et en particulier celles décrites en partie droite. Un modèle de cette figure est par exemple constitué de deux carrés. Un carré est imbriqué dans le premier carré puis un rond est imbriqué dans le second carré. Dans ce modèle, les caractéristiques de couleur ne sont pas décrites puisque le méta-modèle ne les capture pas.

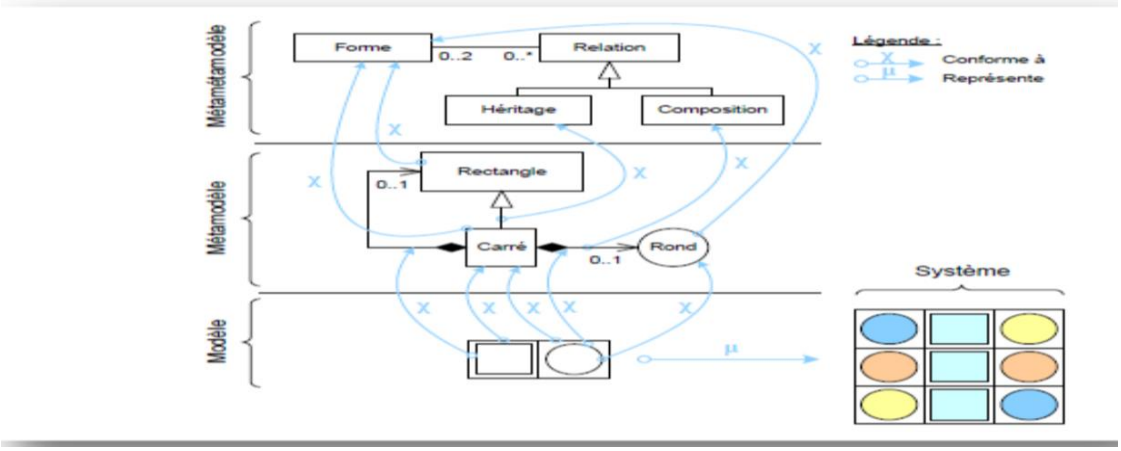


Figure 2. 9: *Illustration des concepts modèle, méta-modèle et méta-méta-modèle.*

2.4.5 Transformation des modèles

Le principe de base du MDA est l'élaboration de modèles indépendants de plate-forme (Platform Independent Model, PIM) et la transformation de ceux-ci en modèles dépendants de plates-formes (Platform Specific Model, PSM) pour l'implémentation concrète du système. Les techniques employées sont donc principalement des techniques de méta- modélisation et des techniques de transformation de modèles.

Aujourd'hui, il est largement reconnu qu'un des points clés de MDA est la transformation de modèles (Sendall, et al., 2003). La définition suivante de la transformation de modèles, largement consensuelle, est proposée dans l'ouvrage « MDA Explained : The Model-Driven Architecture : Practice and Promise », de Kleppeet al.. (Kleppe , et al., 2003).

- "A Transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language".
- La définition transformation des modèles : Une transformation de modèle est une fonction, t : S ⟶ T, qui à partir d'un ensemble de modèles sources dans S, crée un ensemble de modèles cibles dans T. Les ensembles S et T sont des ensembles de modèles conformes à deux ensembles de méta-modèles. Si ces deux ensembles de méta-modèles sont identiques, alors la fonction est endogène, sinon elle est exogène.
- Dans l'article « Feature –based survey of model transformation approaches » rédigé par Czarnecki, K. Helsen.S (Czarnecki, et al., 2006), la transformation des modèles est définie comme étant le processus de conversion d'un modèle d'un système vers un autre modèle. Une transformation de modèles peut également avoir plusieurs modèles source et plusieurs modèles cibles. Une caractéristique de transformation de modèles est qu'elle est un modèle puisqu'elle doit être conforme à un méta-modèle donné.

La figure 2.10 extraite de (Czarnecki, et al., 2006) décrit l'ensemble des concepts de la transformation des modèles.

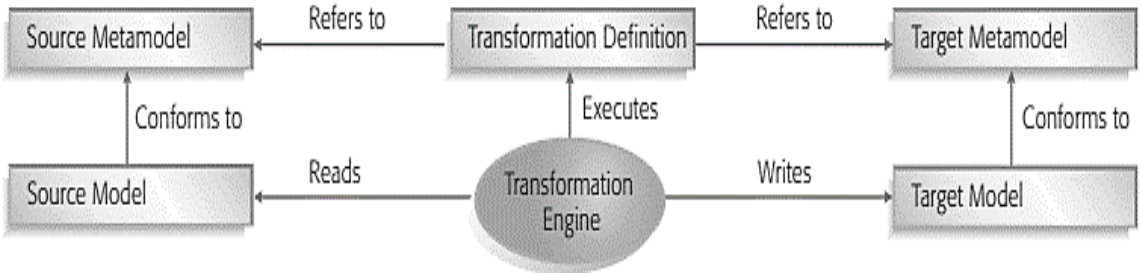


Figure 2. 10 *:: Concepts de base de la transformation de modèles (Czarnecki, et al., 2006)*

- Dans l'ouvrage « MDA Explained : The Model-Driven Architecture : Practice and Promise» (Kleppe , et al., 2003) et dans l'article « A Taxonomy of Model Transformations » (Mens, et al., 2006), les auteurs proposent la définition suivante pour la transformation de modèles : la transformation est une génération automatique d'un modèle cible (Target model) à partir d'un modèle source (source Model). D'après la définition de la transformation, La transformation est un ensemble de règles de transformation, qui décrivent ensemble comment un modèle dans un langage source, est transformé en modèle dans un langage cible. Une règle de transformation est une description de comment un concept ou plus dans un langage source peut être transformé en un concept ou plus d'un langage cible. Une transformation de modèles est une opération qui crée automatiquement un ensemble de modèles cible à partir d'un ensemble de modèles source. La figure 2.11 représente le contexte opérationnel de la transformation de modèles (Jouault, 2006). Le modèle MB, conforme au méta-modèle MMB, est obtenu par l'application de la transformation MMA à MMB au modèle MA, conforme au méta-modèle MMA. De plus, en suivant le principe « tout est modèle ». La transformation elle-même est un modèle dont le méta-modèle est MMT. On dit alors que MMA à MMB est un modèle de transformation. Le langage de transformation, ou plus précisément ses concepts et leurs relations, est donc capturé par ce méta-modèle. Les trois méta-modèles MMA, MMB et MMT sont conformes au méta-métamodèle qui est conforme à lui-même.

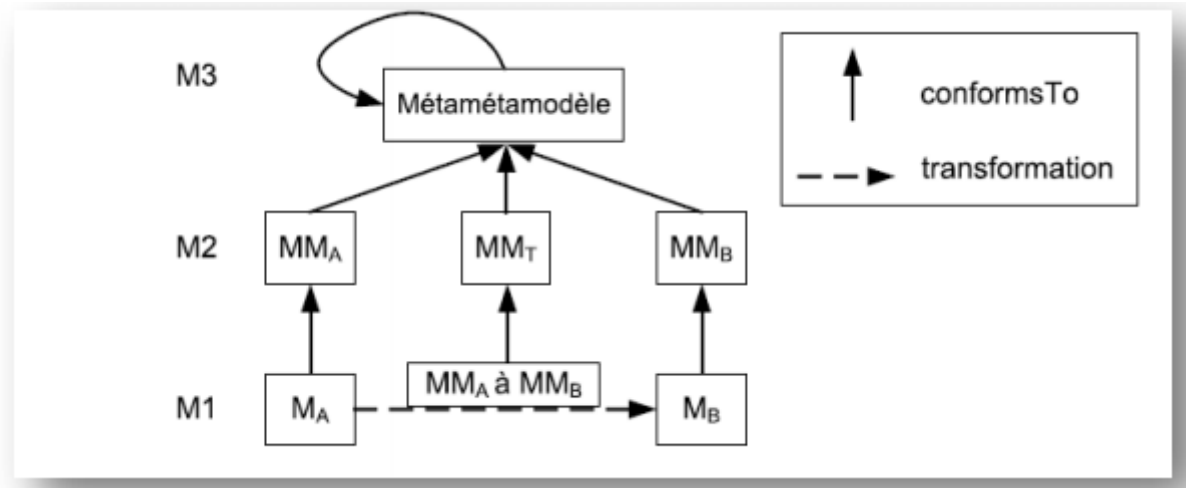


***Figure 2. 11** : La transformation de modèles (Jouault, 2006).*

2.4.5.1 La classification des transformations des modèles

- **La transformation verticale/La transformation horizontale/ La transformation oblique** : (Lara , et al., 2005) (Azaiez, 2007)

  - La transformation verticale : La source et la cible d'une transformation verticale sont définies à différents niveaux d'abstraction. Une transformation qui baisse le niveau d'abstraction est appelée un raffinement. Une transformation qui élève le niveau est appelée une abstraction. (Exemple : génération de code).

  - La transformation horizontale : Une transformation horizontale modifie la représentation source tout en conservant le même niveau d'abstraction. La transformation peut être l'ajout, la modification, la suppression ou la restructuration d'informations (exemple : la refactorisation).

  - La transformation oblique : Une transformation oblique combine une transformation horizontale et une verticale. Ce type de transformation est notamment utilisé par les compilateurs, qui effectuent des optimisations du code source avant de générer le code exécutable.

- **La transformation endogène / La transformation exogène** : transformer un modèle Ma en un modèle Mb, si les méta-modèles respectifs MMa et MMb sont identiques (transformation endogène), ou différents (transformation exogène) (Bézivin, 2004).

  - Les transformations endogènes mettent en jeu des modèles exprimés dans un même langage (possédant donc le même méta-modèle). Il peut ainsi s'agir de :

- L'optimisation : la transformation a pour but d'améliorer par exemple la qualité d'exécution (en termes de performance) du modèle, tout en préservant la sémantique du modèle,
- La restructuration : la transformation entraine un changement de la structure interne du modèle afin d'améliorer la qualité de certaines caractéristiques comme la modularité et la réutilisation, sans modifier le comportement du modèle,
- La simplification : la transformation va permettre de simplifier la complexité de la syntaxe du modèle.

▪ La transformation exogène, quant à elle, met en jeu des modèles exprimés dans des langages différents. De même que pour la transformation endogène, on peut citer plusieurs exemples de transformation exogène :

- La migration : la transformation d'un modèle écrit dans un certain langage en un modèle écrit dans un autre langage, tout en gardant le même niveau d'abstraction.
- La synthèse : la transformation d'un modèle de haut niveau, très abstrait (spécification, modèle fonctionnel) vers un modèle de bas niveau plus concret (code généré, exécutable).
- Le « reverse engineering » : il s'agit d'une transformation de synthèse inverse qui permet d'extraire des spécifications de haut niveau d'un modèle de bas niveau (Guihal, Mai 2007). La figure 2.12 extraite de la thèse de doctorat intitulée «Approche de méta-modélisation pour la simulation et la vérification de modèle » (COMBEMALE juillet 2008), Combemale Benoît récapitule les types de transformation et leurs principales utilisations.

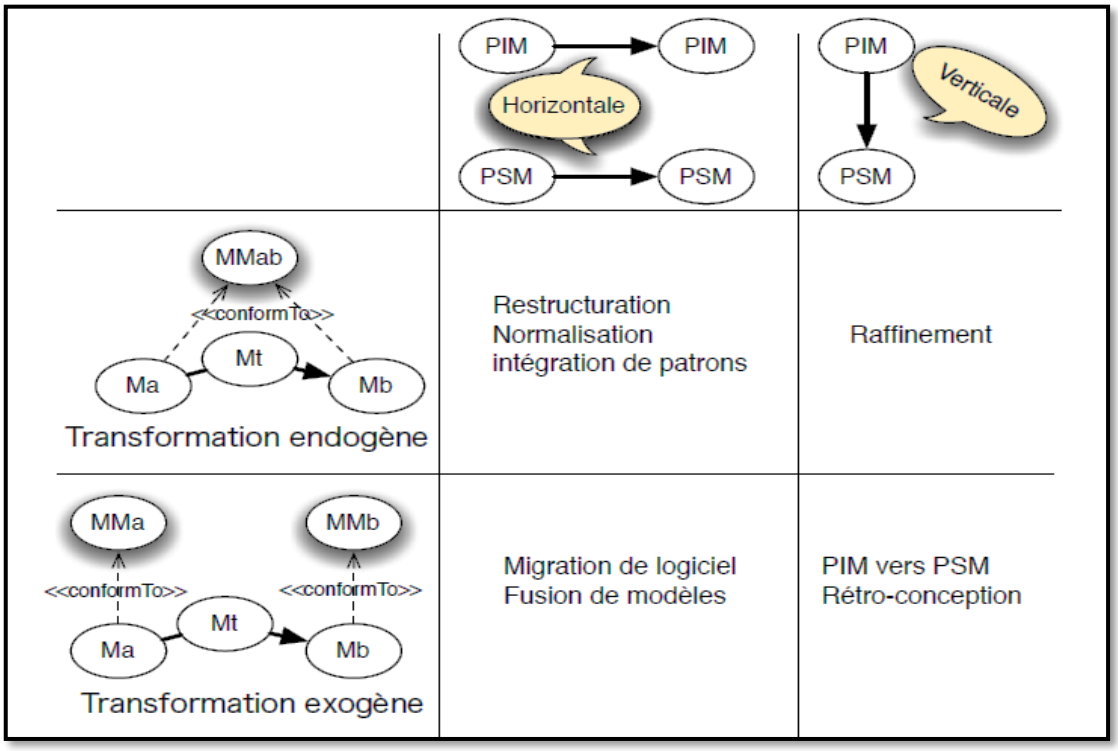


Figure 2. 12 : Types de transformation et leurs principales utilisations **(COMBEMALE juillet 2008).**

-Transformations PIM- PIM et PSM -PSM / Transformation PIM-PSM/ Transformation PSM–code/ Transformations inverses PIM-PSM et PSM-code. Rappel sur : Typologie des modèles dans l'approche MDA L'OMG [2] a défini une typologie de modèles, ainsi qu'un ensemble de relations de transformation qui permettent de passer de l'un à l'autre. Les quatre principaux types de modèles définis dans l'approche MDA (Bézivin , et al., 2002) sont les suivants :

- CIM (Computation Independent Model) : Aussi appelé modèle de domaine ou modèle métier, le CIM capture les exigences en termes de besoins et décrit la situation dans laquelle le système sera utilisé. Son but est d'aider à la compréhension du problème mais aussi de fixer un vocabulaire commun pour un domaine particulier. Dans la pratique, l'appellation « CIM » est très peu utilisée.

- PIM (Platform Independant Model) : Le PIM décrit le système indépendamment de la plate-forme cible sur laquelle il s'exécutera. Il présente donc une vue fonctionnelle détaillée du système, sans détails techniques. Il peut être raffiné progressivement jusqu'à intégrer des détails

[2] Object Management Group est une association américaine à but non-lucratif créée en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes. L'OMG est notamment à la base des standards UML (Unified Modeling Language), MOF (Meta-Object Facility), CORBA (Common Object Request Broker Architecture) et IDL (Interface Definition Language).

d'architecture spécifiques à un type de plate-forme (machine virtuelle, système d'exploitation, etc.), mais il doit rester technologiquement neutre.

- PDM (Platform Description Model) : Le PDM est le modèle qui décrit une plate-forme d'exécution. Il fournit un ensemble de concepts techniques représentant les différentes parties de la plate-forme et/ou les services qu'elle fournit. Un PDM peut représenter, par exemple, des plates-formes à base de composants comme CCM (OMG, CORBA Component Model Specification 2006) ou EJB.
- PSM (Platform Specific Model) : Le PSM est le résultat de la combinaison du PIM et du PDM. Il représente une vue technique détaillée du système. Il peut exister avec différents niveaux de détails. Dans sa forme la plus détaillée, il sert de base à la génération de l'implémentation.

Plusieurs types de transformations de modèle sont définis dans la MDA

- Transformations PIM ⟶ PIM : PIM vers PIM. Ce type de transformation est utilisé pour étendre ou spécialiser un modèle sans ajout d'information dépendant de la plateforme. Typiquement, cette projection est mise en œuvre pour l'analyse et la conception de modèles. Ce type de transformation est en règle générale liée au raffinement de modèles. Elle est elle- même exprimée sous la forme d'un modèle de transformation. (Marvie décembre 2002).
- Transformations PIM ⟶ PSM : PIM vers PSM. Ce type de transformation est utilisé dès lors qu'un PIM est suffisamment raffiné pour être projeté vers la plate-forme d'exécution. Les caractéristiques de la plate-forme servent de base à la projection et doivent être décrites dans un formalisme de modélisation, comme UML par exemple. Le passage d'un modèle abstrait de composants à un modèle technologique comme le CCM représente une projection de type PIM vers PSM (Marvie décembre 2002). Les transformations de type PIM vers PIM ou PSM vers PSM tentent d'enrichir, filtrer ou spécialiser le modèle. Il s'agit de transformations de modèle à modèle (Czarnecki, et al., 2006). Elles sont automatisables (ou partiellement automatisables) dans certains cas comme la traduction vers un autre langage mais les transformations de type raffinement ne le sont généralement pas.
- Transformation PIM ⟶ PSM : La transformation de PIM vers PSM permet de spécialiser le PIM en fonction de la plate-forme cible choisie. Elle n'est effectuée qu'une fois le PIM suffisamment raffiné. Cette transformation de modèle à modèle est réalisée en s'appuyant sur les informations fournies par le PDM.
- Transformation PSM ⟶ code : La transformation de PSM vers l'implémentation (le code) est une transformation de type modèle à texte (Czarnecki, et al., 2006). Le code est parfois

assimilé par certains à un PSM exécutable. Dans la pratique, il n'est généralement pas possible d'obtenir la totalité du code à partir du modèle, et il est alors nécessaire de le compléter manuellement.

- Transformations inverses PIM ⟶ PSM et PSM ⟶ code : Ces transformations sont des opérations de rétro-ingénierie (reverse engineering). Ce type de transformation pose de nombreuses difficultés mais il est essentiel pour la réutilisation de l'existant dans le cadre de l'approche MDA. La figure 2.13 réunit les modèles ainsi que les transformations dans l'approche MDA.

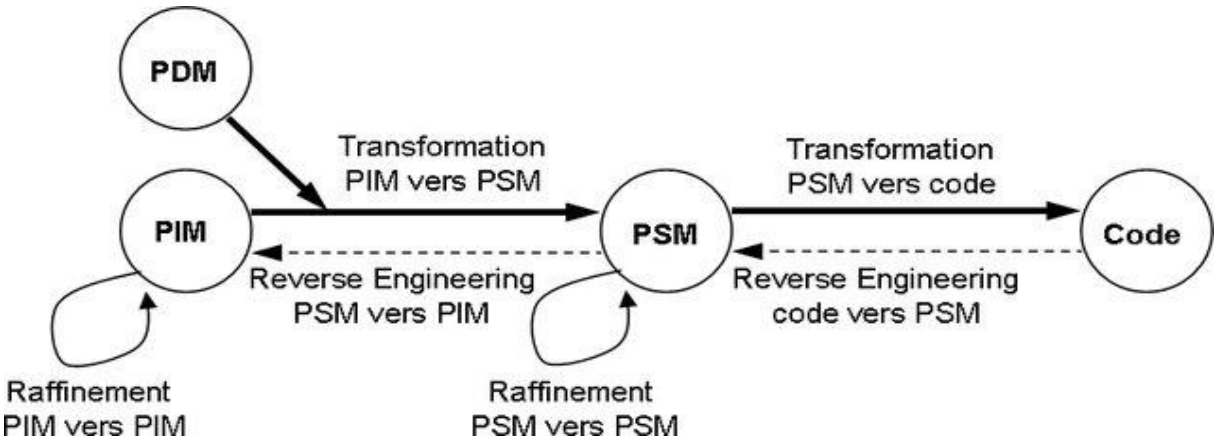


Figure 2. 13: Modèles et transformations dans l'approche MDA.

-**Transformation modèle vers modèle / modèle vers code** : (Czarnecki, et al., January 2003) De manière perpendiculaire à cette classification, on peut distinguer les transformations de type modèle vers modèle et celles de type modèle vers code. En fait, même si les secondes peuvent être considérées comme un cas particulier des premières, il suffit de fournir un métamodèle pour le langage de programmation cible. La distinction est néanmoins justifiée car le code est le plus souvent généré comme du texte simple.

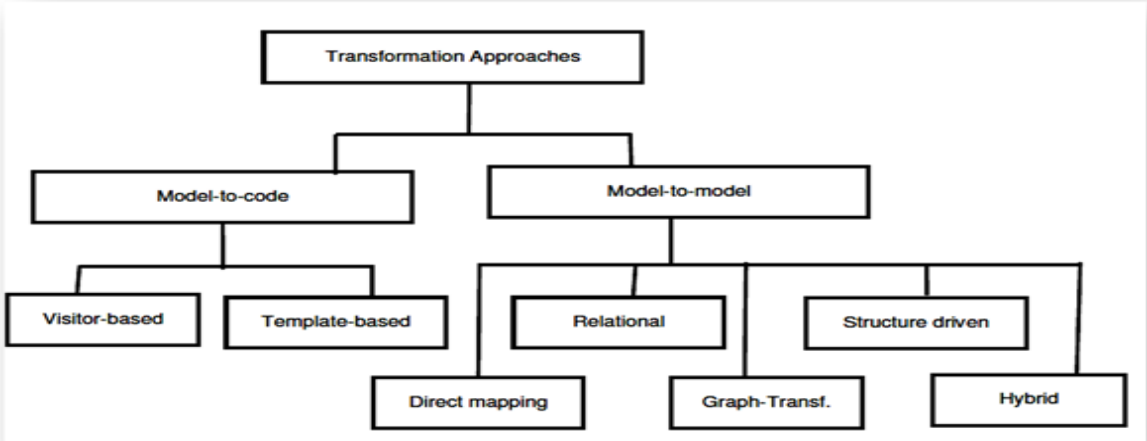


Figure 2. 14 : de transformations de modèles (modèle vers modèle / modèle vers code)

- Dans «Classification of model transformation approaches» (Czarnecki, et al., January 2003), Krzystof Czarnecki et Simon Helsen proposent la classification schématisée dans la figure 2.14.
- **Transformations de type modèle vers code** : Dans cette catégorie, on distingue deux approches, basées sur les principes visiteur[3] ou patron[4].

  - Les transformations basées sur le principe du visiteur : consistent à traverser le modèle en lui ajoutant des éléments qui réduisent la différence de sémantique entre le modèle et le langage de programmation cible. Le code est obtenu en parcourant le modèle enrichi pour créer un flux de texte. Le projet Jamda[5] destiné à la génération de code Java fournit un bon exemple de cette approche.
  - Les transformations basées sur le principe des patrons : elles sont actuellement les plus courantes. Le code cible contient des morceaux de méta-code utilisés pour accéder aux informations du modèle source. Parmi les outils basés sur cette approche, on peut citer AndroMDA[6] un générateur de code qui repose notamment sur Velocity[7] pour l'écriture des patrons.

- **Transformations de type modèle vers modèle** : Dans cette catégorie, on distingue :

  - Les transformations basées sur la manipulation directe : ces transformations se basent sur une représentation interne des modèles source et cible, et sur un ensemble d'APIs : une API permet de manipuler la représentation interne des modèles. Elles sont en général implémentées comme un framework orienté objet qui fournit une infrastructure pour organiser les transformations (par exemple, la définition de classes abstraites pour les transformations). Les utilisateurs doivent eux-mêmes réaliser les règles de transformations au moyen d'un langage de programmation comme Java. La combinaison JMI (Java Metadata Interface) et Java est souvent utilisée dans la mise en œuvre de cette approche. JMI est une API basée sur le MOF et permet de créer, de sérialiser, et d'accéder aux éléments d'un modèle défini à l'aide du MOF ou Ecore.

  - **Les transformations relationnelles** : cette transformation consiste à établir une relation entre les éléments des modèles sources et cibles. Ces relations seront spécifiées à l'aide de contraintes. Elles sont purement déclaratives et leur spécification n'est pas exécutable. La

[3] Visitor- based approach dans le vocabulaire anglo-saxon.
[4] Template-based approach dans le vocabulaire anglo-saxon
[5] Le projet Jamda. Disponible à : http://jamda.sourceforge.net/
[6] AndroMDA, site Internet. Disponible à : http://www.andromda.org
[7] Velocity 1.4, The Apache Jakarta Project, site Internet. Disponible à http://velocity.apache.org/

programmation logique avec les principes d'unification, de recherche et de backtracking est notamment adaptée dans cette transformation. Les transformations produites généralement sont bidirectionnelles.

- **Les transformations des graphes :** Les modèles et méta-modèles possèdent souvent une représentation graphique apparentée à un graphe. Un modèle, dans ce cas, peut être considéré comme un graphe étiqueté, contraint par des règles de cohérence essentiellement définies comme un méta-méta-modèle MOF. Les techniques de réécriture de graphes et de transformation de graphes peuvent être appliquées pour des transformations de modèles. D'une manière générale, les systèmes de réécriture de graphes combinent une notation graphique et une notation textuelle afin d'exprimer ces transformations. Un programme de transformation essentiellement composé de règles de réécriture va d'abord sélectionner un fragment d'un graphe source, identifié grâce à un langage de navigation. L'application d'un filtre sur ce fragment sélectionné permet de le modifier avant de le recopier dans le graphe cible.
- **Les transformations basées sur la structure :** Ces transformations spécifient deux phases. La première consiste à créer la structure hiérarchique du modèle cible. La seconde consiste à ajuster les attributs et références dans le modèle cible. On peut citer comme exemple : OptimalJ, Interactive Objects and Project Technology (IOPT).
- **Les transformations hybrides :** Elles représentent une combinaison entre plusieurs transformations précédentes. Le langage de transformation de règles est une combinaison d'approches déclarative et impérative. Une combinaison se traduit par des :

  -Mapping rules qui spécifient les relations entre les éléments sources et cibles,

  -Operational rules qui décrivent les règles exécutables (réalisation de la transformation).

  On peut citer aussi les transformations incrémentales de modèle et les transformations de modèle bidirectionnelle (Le Calvar 2019).

-Transformation incrémentale de modèles (T. Le Calvar 2019)

Les transformations classiques traitent l'intégralité du modèle source pour créer un modèle cible conforme à la spécification de transformation. Cette opération peut être coûteuse lorsque le modèle source est composée d'un nombre important d'éléments, ou lorsque la transformation requiert des opérations coûteuses en temps de calcul. Ce coût de calcul n'est pas problématique dans le cas de modèles qui changent peu. Cependant, si de nombreuses modifications sont appliquées aux éléments du modèle source, alors il peut être intéressant d'utiliser une approche de transformation incrémentale.

L'idée au cœur de la transformation de modèles incrémentale est de n'appliquer la transformation qu'aux éléments sources ayant été modifiés depuis la dernière exécution. Cela réduit ainsi le temps nécessaire à l'application de la transformation au modèle mis à jour. De plus, mettre à jour le modèle cible lorsque des modifications sont détectées sur le modèle source permet de conserver l'identité des éléments du modèle cible.

C'est notamment le cas lorsque le modèle cible est un modèle graphique présenté à l'utilisateur.

Dans ce cas, il est important que les modifications apportées au modèle source soient appliquées rapidement au modèle cible. De plus, ici, il est préférable de mettre à jour les éléments visuels présentés à l'utilisateur plutôt que de devoir les remplacer complètement à chaque changement.

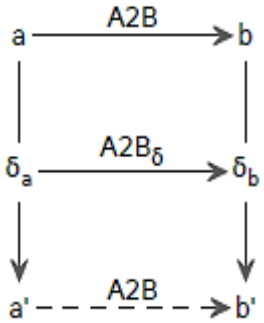


Figure 2. 15: Schéma d'une transformation incrémentale

La Figure 2.15 reprend l'idée de la transformation incrémentale. En haut à gauche, le modèle source a est utilisé par la transformation A2B, pour générer le modèle b. Ensuite, une modification *δa* est appliquée à a qui devient à. Une transformation classique repartirait de a' pour calculer b'. Une transformation incrémentale utilisera *δa* pour calculer *δb*, l'ensemble de modifications à appliquer a b pour qu'il devienne b'. Cela évite ainsi la transformation A2B, en pointillés, généralement plus couteuse que A2B*δ* qui ne transforme que la mise à jour.

Plusieurs approches parmi celles présentées précédemment supportent l'exécution de transformations incrémentales. Par exemple, on peut citer Viatra à partir de sa version 3 (Horváth , Ráth et Ujhelyi 2016), des approches basées sur les TGGs (Giese et Wagner 2006) (Leblebici , et al. 2014), plusieurs extensions d'ATL (Jouault et Tisi, Towards Incremental Execution of ATL Transformations 2010) (Le Calvar, et al. 2019) (Martínez Pérez, Tisi et Douence 2017), YAMTL (Boronat 2018), ou les opérations actives (Beaudoux, et al. 2010). Dans «A Survey on incremental model transformation approaches» (Kusel, et al. 2013), les auteurs présentent en détail de nombreuses approches de transformations incrémentales.

-Transformation de modèles bidirectionnelle (Le Calvar, et al. 2019)

Lors d'une transformation de modèles, le modèle cible est calculé à partir du modèle source. Idéalement un seul modèle est modifié (la source), ainsi, recalculer une nouvelle cible (ou calculer un ensemble de modifications à appliquer) est possible à l'aide des approches classiques ou incrémentales présentées précédemment. Un exemple classique de transformation bidirectionnelle est la génération de code à partir d'un modèle. Il n'est pas rare que le code généré doive être manuellement modifié pour ajouter des fonctionnalités manquantes, ou pour optimiser le code. Cependant, si le modèle ayant servi à générer le code est modifié, il est nécessaire de régénérer le code. Une approche classique ne prendrait pas en compte les changements sur le code et les écraserait avec la nouvelle version.

Ce comportement n'est pas satisfaisant.

Généralement, lorsque deux modèles peuvent être modifiés, il est intéressant d'être capable de calculer des modifications à appliquer à la source lorsque la cible est modifiée. Dans ces cas, la transformation de modèles peut être vue comme une relation de cohérence entre deux modèles existants. Cette relation assure la cohérence entre les deux modèles, et répare les modèles lorsque l'un des deux est modifié. On peut définir cette relation comme deux fonctions de réparation prenant un modèle source $s \in S$ et un modèle cible $c \in C$:

$$\overrightarrow{R} : S \times C \rightarrow C$$

$$\overleftarrow{R} : S \times C \rightarrow S$$

$\overrightarrow{\mathbf{R}}$ correspond à la transformation avant (ou forward), et $\overleftarrow{\mathbf{R}}$ correspond à la transformation inverse (ou backward). Soit $s' \in S$ une mise a jour de s, alors :

$\overrightarrow{\mathbf{R}}$ peut être utilisée pour calculer $c' \in C$, $c' = \overrightarrow{\mathbf{R}}(s', c)$.De même $\overleftarrow{\mathbf{R}}$ peut être calculé s' à partir de s et c', $s' = \overleftarrow{\mathbf{R}}(s,c')$.

Il est important de s'assurer que $\overleftarrow{\mathbf{R}}$ et $\overrightarrow{\mathbf{R}}$ soient cohérentes l'une avec l'autre. Une façon de s'assurer que ces deux fonctions soient cohérentes est de les combiner. C'est le cas par exemple de la syntaxe graphique des TGGs. Dans ce formalisme, si un motif est détecté sur le LG (respectivement RG), alors les motifs du CG et RG (respectivement LG) seront ajoutés aux graphes existants.

C’est aussi le cas des approches à base de lentilles (*lenses*) apparues dans « Symmetric lenses », ridigé par Hofmann et al. (Hofmann, Pierce et Wagner 1January 2011), et dans « Boomerang : Resourceful Lenses for String Data », rédigé par Bohannon et al.. (Bohannon, et al. 2008). Cependant, il est parfois plus simple et moins restrictif de définir séparément les deux fonctions $\overleftarrow{R}$ et $\overrightarrow{R}$. Dans «A Landscape of Bidirectional Model Transformations » de Stevens (Stevens 2008) et dans « Feature-based classification of bidirectional transformation approaches »de Hidaka (Hidaka, et al. 2016), les auteurs présentent un aperçu des techniques et problématiques de la transformation bidirectionnelle de modèles.

2. 5 Les outils de transformations

- **AGG** "Algebraic Graph Grammar" (Taentzer, Springer, 2000.) est un environnement à usage général pour le développement des systèmes de transformation de graphes attribués. Il est basé sur l'approche algébrique pour la transformation de graphes. Il vise à la spéciation et le prototypage rapide d'applications complexes tel que le graphe de données structurées. Puisque la transformation de graphe peut être appliquée à des niveaux d'abstraction très différents, elle peut être attribuée, ou non, par des calculs simples ou par des processus complexes, selon le niveau d'abstraction. En raison de sa base formelle, AGG offre un support de validation et de vérification de la cohérence des graphes, en fonction des contraintes graphiques.

- **ATOM3**[8] "Domain Specific Visual Languages" (De Lara , et al., 2002) est un outil pour la conception de Domain Specific Visual Languages (DSML). Il permet de définir la syntaxe abstraite et concrète d'un langage visuel au moyen de la méta-modélisation et d'exprimer la manipulation du modèle en utilisant la transformation de graphe. Avec les informations de métamodèle, ATOM3 génère un environnement de modélisation personnalisé pour le langage décrit. Récemment, ATOM3 a été étendu avec des fonctionnalités pour générer des environnements avec des vues multiples de langages visuels (tel que UML) et Triple Graph Grammar (TGG) (Schürr, January 1994). Ce dernier est utile pour exprimer l'évolution de deux modèles différents, liés par un modèle intermédiaire.

- **GROOVE** "GRaph of Object Oriented VErification"[9] (GROOVE) est un outil d'usage général basé sur la transformation de graphes pour modéliser la conception, la compilation et la structure d'exécution des systèmes orientés objet. La transformation de graphes avec GROOVE se base sur la transformation de modèles et la sémantique opérationnelle. Cela implique l'usage d'un fondement formel de transformation de modèles et de sémantique dynamique, et aussi la capacité de vérifier la transformation de modèle en utilisant le model checking. L'outil GROOVE a un éditeur pour la création des règles de production graphiques, un simulateur pour calculer visuellement les transformations de graphe, induit par un ensemble de règles de production graphique (Ghamarian, et al., 2012).

- **GrGen**[10] est un outil de transformation de graphes utilisé dans différents domaines tels que la transformation de données graphiques complexes, la linguistique informatique, ou la construction de compilateur moderne. GrGen permet de travailler à un niveau très abstrait en utilisant le modèle

[8] **AToM3**. Homepage : http://atom3.cs.mcgill.ca
[9] **GROOVE**. Homepage : https://groove.ewi.utwente.nl/about
[10] **GrGen**. Homepage http://www.info.uni-karlsruhe.de/software/grgen/

déclaratif. Le code généré par GrGen s'exécute très rapidement. En termes de performance par rapport aux autres outils, GrGen permet la manipulation et la transformation des systèmes complexes. Le système GrGen est écrit en Java et C. Son noyau est un générateur de grammaires de graphe.

- **Fujaba** L'outil FUJABA (From UML to Java And Back Again) utilise les classes UML et les diagrammes pour produire du code Java, afin de fournir une conception de systèmes formels et de langages spécifiques. Les fonctionnalités de la rétro-ingénierie (reverse engineering) y compris la conception de reconnaissance de formes sont également disponibles. Le métamodèle du FUJABA peut être étendu par héritage à la place de l'instanciation (Levendovszky, et al., 2005).

- **EMF Tiger** (Transformation generation) est un environnement pour la transformation de modèles EMF. Les transformations sont définies et exécutées directement sur des modèles EMF soit à l'aide d'un code généré ou soit avec un interpréteur afin d'assurer la sécurité et l'efficacité de types d'artifacts.

Les règles basées sur le langage de transformation EMF Tiger sont inspirées des concepts de transformation de graphes et combinent les deux aspects déclaratif et procédural. En particulier, les règles de transformation sont déclaratives, dans le sens où un modèle structurel est utilisé pour définir les préconditions d'une règle. Le concept procédural est représenté par la structure de contrôle de flux tels que les couches et les boucles, et peuvent être utilisés pour appliquer un ensemble de règles de transformation d'une manière contrôlée (Biermann, et al., 2010).

- **Viatra** est un outil basé sur Eclipse pour un usage général de l'ingénierie de transformation de modèles. C'est un framework qui supporte un cycle de vie complet pour la spécification, la conception, l'exécution, la validation et la maintenance des transformations entre les déférentes langages et les domaines de modélisation. VIATRA2 est apte à coopérer avec un système externe arbitraire et à exécuter la transformation avec une transformation de modèle natif (plugin), qui est généré par VIATRA2. Le langage de spécification des règles combine la transformation de graphes et les machines d'état abstraites en un seul paradigme. Essentiellement, les étapes de transformation élémentaires sont capturées par des règles de transformation graphiques, tandis que des transformations complexes sont assemblées à partir des étapes élémentaires en utilisant les règles de machine d'états abstraites pour la spécification du flux de contrôle (Taentzer, et al., January 2013).

Viatra (Varro et Balogh 2007) est une approche combinant les techniques de réécriture de graphes et de machine à états abstraits (ASM). Les techniques de réécriture de graphes servent notamment à la détection de motifs. Ces motifs (positifs ou négatifs) sont similaires aux contextes présentes dans les TGGs, ce sont des sous-graphes devant être présents (ou absents dans le cas de motifs négatifs). Ces motifs sont utilisés dans les

différentes parties des règles Viatra pour extraire les parties des modèles d'entrée ou de sortie impliques dans les règles.

La définition des règles Viatra se fait en trois parties. La première, la précondition, est constituée d'un ensemble d'appels à des motifs (positifs ou négatifs) devant être trouvés pour pouvoir appliquer la règle. La seconde, la post-condition, est-elle aussi constituée de motifs qui devront être présents après l'application de la règle. La partie action contient du code pouvant être exécuté par l'ASM dans le contexte de la règle après que la post-condition ait été vérifiée. Enfin, l'application des différentes règles se fait grâce a du code ASM appelant les différentes règles de transformation. Viatra utilise un algorithme basé sur l'algorithme Rete (Forgy 1989). L'algorithme Rete est un algorithme adapté à la détection de nombreuses occurrences de motifs sur un nombre important d'éléments. Les différents motifs à détecter sont compilés en un réseau qui sera utilisé pour détecter leurs occurrences motifs dans le modèle source.

- **ATLAS Transformation Language** ATL (Jouault, et al. June 2008) (Jouault et Kurtev , Transforming Models with ATL s.d.) est un autre langage de transformation hybride basé sur le concept de règles. Cependant, contrairement aux TGGs et à Viatra, les règles ATL ne se basent pas sur les motifs de graphes pour l'activation et les effets d'une règle. Les règles ATL déclarent un ensemble d'éléments d'entrée ainsi que des gardes optionnelles sur ces éléments d'entrée.

Ce mécanisme permet la sélection fine des éléments d'entrée comme le permettent les AC ou NAC des TGGs et de Viatra. Les éléments de sortie sont déclarés dans une seconde section de la règle. C'est à cet endroit que sont déclarées les relations (ou *bindings*). Ces *bindings* sont fortement inspirés des expressions OCL. Les *bindings* permettent de calculer les valeurs des propriétés du modèle cible à partir des propriétés du modèle source. Sauf cas particulier, le modèle source n'est accessible qu'en lecture seule et le modèle cible n'est accessible qu'en écriture seule.

En ATL il existe deux grands types de règles. Les règles standard sont automatiquement exécutées pour toutes les combinaisons d'entrée présentes dans le modèle source. Les règles paresseuses (*lazy*) ne sont exécutées que sur demande d'une autre règle. Il existe deux types de règles paresseuses, les règles avec cache (*unique*) ou sans. Le cache d'une règle paresseuse permet de retourner le même élément de sortie si la même règle est appelée avec les mêmes éléments d'entrée plusieurs fois.

En plus de ce système de règles déclaratives, ATL permet l'exécution de blocs de code impératif. Ces blocs impératifs facilitent la spécification de transformations difficilement exprimables en ATL déclaratif pur.

L'exécution d'une transformation se fait par l'appel optionnel à une règle spéciale, le point d'entrée (ou *entry point*). L'algorithme cherche ensuite les règles applicables aux éléments d'entrée fournis. Pour cela il compare les éléments d'entrée des règles ainsi que les gardes éventuelles aux éléments du modèle, et

enregistre les combinaisons valides. Pour chacune de ces combinaisons, l'algorithme crée les liens de traçabilité, servant à associer à chaque élément de sortie le ou les éléments d'entrée correspondants. Le concept de lien de traçabilité est très proche de la notion de graphe de correspondance des TGGs. Ensuite, l'algorithme crée les éléments de sortie correspondant et exécute les *bindings* pour calculer la valeur de leurs propriétés. On notera que l'ordre d'application des règles n'est pas garanti.

- TGG [11]Triple Graph Grammars (TGGs) est un formalisme pour la spécification basée sur des règles de correspondances entre différents types de modèles. La suite d'outils d'interprétation TGG fournit des outils pour la spécification et l'exécution de transformations de modèle à modèle (M2M) basées sur TGG[12] et la synchronisation de modèle au sein de la plate-forme Eclipse.

Il existe de nombreuses techniques de transformation utilisant des approches différentes. On peut citer les approches basées sur la réécriture de graphe, tel que les Triple Graph Grammars (TGGs) (Schürr , Specification of graph translators with triple graph grammars 1995) . De nombreux outils sont basés sur cette approche, on peut notamment citer MoTe [13], TGG Interpreter ou eMoflon[14] . Pour plus de détails sur les différents outils de transformation de graphes ainsi que leurs capacités peuvent être trouvés dans l'article « A Survey of Triple Graph Grammar Tools » (Hildebrandt, Lambers, et al., A Survey of Triple Graph Grammar Tools 2013), ridigé par Hildebrandt et al.. Les modèles étant des graphes typés il est facile d'exprimer des transformations de modèles comme des réécritures de graphes. Trois graphes sont mis en relations, le graphe "source" ou "gauche" (LG ou Left Graph), le graphe "cible" ou "droit" (RG ou Right Graph) et le graphe de "correspondance" (CG ou Correspondance Graph). Les graphes LG et RG font référence aux modèles source et cible de la transformation, tandis que le CG correspond à la traçabilité permettant de retracer les correspondances entre les éléments du graphe source et leurs équivalents dans le graphe cible. (OMG, OMG : Meta object facility (MOF) core specification. 2006).

Les transformations basées sur les TGGs sont composées d'un ensemble de règles. Ces règles permettent de faire évoluer les trois graphes en assurant qu'ils soient toujours cohérents les uns par rapport aux autres. Ces règles sont composées de deux parties. La première partie, le contexte ou précondition, correspond à l'ensemble des éléments qui doivent être présents pour pouvoir appliquer la règle (application condition ou AC). Certains moteurs d'exécution proposent, en plus des éléments qui doivent être présents dans le contexte, d'ajouter des éléments qui ne doivent pas être présents (negative application condition ou NAC). La seconde partie, ou post-

[11] http://jgreen.de/tools/tgg-interpreter/
[12] http://www.mdetools.com/detail.php?toolId=39
[13] http://www.mdelab.de/mote/
[14] https://emoflon.org/

condition est constituée des éléments créés, c'est à dire la partie des graphes qui est générée par l'application de la règle. Ensuite, cet ensemble de règles de construction permet de générer le graphe de correspondance ainsi que le graphe de droite (respectivement gauche) à partir du graphe de gauche (respectivement droite). L'exécution d'une transformation consiste alors à trouver un ordre d'application des règles tel que l'application de ces règles qui génère le graphe de droite (respectivement gauche) en partant d'un modèle vide. Il est ainsi possible de construire en même temps les graphes de correspondance et de gauche (respectivement droite). Le tableau 2.1présente une comparaison entre différents outils de transformation.

Critère	AGG	ATOM3	GROOVE	GrGen	Fujaba	EMF Tiger	Viatra
Nombre de source et cible	One to one	One to one	One to one	One to one	Many to many	One to one	Many to many
Type de transformation	Endogène	Endogène	Endogène	Exogène	Endogène	Exogène	Endogène et exogène
Approche utilisé	SPO et NAC	Grammaire de graphe	SPO et NAC	SPO et DPO	TGG	SPO	TGG
Intégrité de contrainte	Non supporté Présenté par NAC ou Java	Non supporté Présenté par OCL ou Python	Non supporté Présenté par NAC	Non supporté Présenté par NAC	Non supporté Présenté par OCL	Non supporté Présenté par NAC	Supporté Graph pattern
Editeur	Graphique et textuel	Graphique	Graphique	Textuel	Graphique	Graphique	Graphique pour les modèles et textuel pour les règles
Degré d'automatisation	Grammaire de graphe	Grammaire de graphe	Grammaire de graphe	Optimisation automatique de règles de transformation	Histoire axée modélisation	Simulation automatique	Transformation entraînées par la machines d'état abstraites
Processus de transformation	Automatique	Automatique	Automatique	Automatique	Semi-automatique	Automatique	Automatique
Réutilisabilité	Transformation paramétrées				Transformation paramétrées/hérita -ge de transformations	Transformation configurable	Réutilisation des motifs prédéfinis
Vérification	Exprimé par les couches de règles	Priorité de règles	Priorité de règles	(GRS) pour le contrôle des règles de transformation	Analyse automatique et une vérification cohérente	Contrôles de fin de contrat et l'unicité des résultats de la transformation.	Vérification à part entière
Validation	L'analyse des paires critiques, la résiliation, la préservation de contraintes graphiques	Typage correct du modèle cible	Valider avec le model checking de groove	Contrôle les règles de réécriture contenant les tests arbitraires	Techniques de tests unitaires sont utilisées	Confluence, terminaison, analyse des paires critiques de règles	″Typage correct+ Préservation des modèles de graphes″ valider les transformations avec des techniques de model checking

Composition	Niveaux de priorités (layers) règle	Priorité de règle	Priorité de règle	Graphe Rewrite Séquence (GRS)	Appel aux méthodes de modélisation de l'histoire conduit ¨story diagrams¨	Niveaux de priorité Procédural	Composition non récursive de modèle dans les règles 'machine d'états'
Complexité	Niveaux de grammaires	Des couches avec priorités, le séquençage par priorité. L'exécution en parallèle des matches non chevauchés			Transformation de graphes contrôlés	Respecter les contraintes induites par la sémantique d'agrégation	D'ordre supérieur et méta transformations
Standardisation	GXL, GTXL	XMI	GXL	XMI	UML, Java, XMI, MOF	ECORE	XMI, MOF, UML
Type d'utilisation	Usage général	Transformation de modèle	Usage général	Haute performance	Haute performance	Transformation de modèle	Transformation de modèle
Typage	Requis	Requis	Optionnel	Requis	Requis	Requis	Requis
Contrôle	Priorité	Priorité	Impératif/Priorité	Impératif	Impératif	Priorité	Impératif
Exploration	Linéaire	Linéaire	Multiple	Linéaire	Linéaire	Linéaire	Linéaire

Tableau 2.1: comparaison entre les différents outils de transformation

Note : pour étudier les critères deTGG (Triple Graph Grammar), onsultez:http://www.mdetools.com/detail.php?toolId=39

2. 6 Travaux connexes

- Les auteurs de «A Model Driven Approach for Building OWL DL and OWL Full Ontologies» ont présenté un méta-modèle basé sur MOF (Meta Object Facility) et un profil UML pour OWL DL et OWL Full. Ils ont également développé une transformation entre les ontologies OWL et les modèles UML et vice versa, en utilisant une approche dirigée par le modèle. Ils ont mis en œuvre et validé leur approche avec le Visual Ontology Modeler et le Integrated Ontology Development Toolkit. Dans le travail intitulé « A Model Driven Approach for Building OWL DL and OWL Full Ontologies » (Brockmans, 2006b), les auteurs ont présenté un méta-modèle conforme au MOF et un profil UML pour le Semantic Web Rule Language (SWRL) comme extension de leur travail précédent. La validité des instances de ce méta-modèle est assurée par des contraintes OCL.

- L'article « Tool Integration with Triple Graph Grammars » (Königs, et al., February 2006), Königs, et al., propose une approche basée sur les règles qui permet la spécification déclarative des règles d'intégration des données. Ils sont basés sur le formalisme des grammaires de graphes triples et utilisent des graphes orientés pour représenter les (méta) modèles conformes au MOF, ce qui est conforme aux propositions de l'OMG appelées QVT pour Query, Views, and Transformation du domaine de développement MDA (Model Driven Application).

- Dans « Applying Triple Graph Grammars For Pattern-Based Workflow Model Transformations » (Lohmann, et al., 2007), les auteurs ont proposé une approche dirigée par les modèles pour transformer les modèles de flux de travail donnés comme des diagrammes d'activité UML en descriptions de compositions de services. Ils ont montré comment réaliser une transformation de UML vers BPEL (Business Process Execution Language) et XPDL (XML Process Definition Language) avec une technologie basée sur les Triple Graph Grammars (TGGs). Ils ontnt montré, en particulier, comment les modèles de flux de travail communément connus et récurrents dans les différents langages de processus métier peuvent servir de guide pour la conception des règles de transformation.

- Dans « Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations» (Arendt, et al., October 2010), les auteurs ont montré que l'Eclipse Modeling Framework (EMF) fournit des facilités de modélisation et de génération de code pour les applications Java basées sur des modèles de données structurés. Ils ont étudié "Henshin", qui est un nouveau langage et un ensemble d'outils associés pour les transformations en place des modèles EMF, qui est construit sur des concepts de transformation de graphe.

- Les auteurs de «Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations» (Hildebrandt, et al., October 2011) ont présenté une approche de test de conformité automatique pour les implémentations TGGs, où l'approche Triple Graph Grammars (TGGs) est un représentant important des approches de transformation de modèles relationnels. Ils ont expliqué la commodité de l'utilisation des TGG pour la génération automatique de cas de test de conformité. Ils ont montré comment mesurer la qualité des suites de tests en utilisant leur propre implémentation de TGGs.

- Dans «UML Class Diagrams to OWL Ontologies: A Graph Transformation based Approach » (Belghiat, et al., 2012), les auteurs ont discuté de l'implémentation d'une application, qui effectue une transformation d'un diagramme de classes UML vers une ontologie OWL basée sur la transformation de graphes en utilisant l'outil AToM3. Pour cette tâche, ils ont développé un méta-modèle pour les diagrammes de classes UML, et une grammaire de graphes composée de plusieurs règles, qui leur permettent de transformer tout ce qui est modélisé dans leur environnement généré par AToM3 en ontologies OWL.

- Les auteurs de «Bidirectional Model Transformation with Precedence Triple Graph Grammars» (Lauder, et al., July 2012) ont quant à eux, convenu que les grammaires de graphes triples (TGG) sont une technique basée sur des règles avec un contexte formel pour spécifier la transformation bidirectionnelle des modèles. Ils ont présenté des scénarios pratiques, qui ont montré que les règles

unidirectionnelles nécessaires pour les transformations vers l'avant et vers l'arrière sont automatiquement dérivées d'une nouvelle classe de règles TGGs avec un algorithme de contrôle qui abandonne un certain nombre de restrictions pratiques pour les règles TGGs et qui a toujours une complexité d'exécution polynomiale.

- Les auteurs (Bernaschina, 2017) ont illustré que le développement piloté par les modèles (MDD) nécessite des transformations de modèle à modèle et/ou de modèle à texte (code) pour produire du code d'application à partir de descriptions de haut niveau. Ils ont supposé que la création de ces transformations soit ainsi une tâche complexe, qui nécessite de maîtriser la méta-modélisation, les langages de transformation ad hoc et les outils de développement personnalisés.

- Dans l'article intitulé « A Model Transformation Approach for Specifying Real-Time Systems and Its Verification Using RT-Maude» (Bendiaf, Bourahla, et al. 2017) les auteurs ont proposé une approche basée sur le TGG (Triple Graph Grammar) pour l'intégration des méthodes formelles aux processus de développement suivant les bases de l'ingénierie dirigée par les modèles, pour la spécification et la vérification formelle de SETR(Systèmes Embarqués Temps-Réel).

- Dans l'article intitulé «Transformation of UML class diagram into OWL Ontology» (Minh , et al., 2019), Minh , et al. ont anoncé qu'un ensemble de règles est ajouté pour la transformation des diagrammes de classe UML en ontologies, pour la transformation des attributs avec des types de données comme classes, pour des attributs définis comme structure d'autres attributs, egalement pour des associations avec des types comme classe d'association, et enfin pour l'association récursive, et la transformation des agrégations avec qualification.

- Les auteurs de l'article titré «A Configurable Semantic-Based Transformation Method towards Conceptual Models» de (Tiexin , et al., 2020) ont proposé une méthodologie (semi-) automatique configurable de transformation de modèles conceptuels basée sur la sémantique qui tente de réutiliser des modèles conceptuels existants pour générer de nouveaux modèles. Ils essaient d'améliorer l'efficacité du processus de construction.

- Dans l'article «Transformation of SysML Requirement Diagram into OWL Ontologies» (Wardhana, et al., 2020), Wardhana. et al. proposent un modèle qui peut automatiquement transformer un SysML Requirement Diagram en un fichier OWL, pour extraire les connaissances contenues dans les diagrammes précédents. Le processus de transformation utilise des règles de transformation et un algorithme qui peuvent être impliqués pour changer un diagramme d'exigences SysML en un fichier d'ontologie OWL, dans lequel la sérialisation XML Metadata Interchange (XMI) est utilisée pour représenter la transformation.

- L'article intitulé «Transformation of BPMN Model into an OWL2 Ontology» de (Kchaou, et al., January 2021), les auteurs proposent d'ajouter des règles pour transformer les modèles annotés de gestion des processus métier (BPMN) en ontologies, prenant en compte la sémantique du modèle BPMN, et fournissant tous les objets et activités de l'entreprise. Les transformations génèrent également la représentation graphique OntOWL2.

- Dans l'article intitulé « A generic metamodel for data extraction and generic ontology population » (Chasseray, et al., 2021), les auteurs ont présenté un méta-modèle générique pour l'extraction de données hétérogènes. Le méta-modèle a été conçu avec deux objectifs, le besoin de généricité concernant la source des éléments de connaissance collectés, et l'intention de coller à une structure proche d'une structure ontologique.

2.7 Conclusion

Nous avons récapitulé dans ce chapitre les principes de l'ingénierie dirigée par les modèles (IDM) et les concepts de base liées à IDM. L'objectif primordial de l'ingénierie dirigée par les modèles est l'élaboration des modèles pérennes, indépendants des détails techniques d'implémentation ; afin de permettre la génération automatique de la totalité de code des applications, et d'obtenir un gain significatif de productivité. Nous avons également étudié les deux activités essentielles de l'ingénierie dirigée par les modèles (IDM) qui sont la méta-modélisation et la transformation des modèles.

La méta-modélisation un processus de méta-modélisation qui consiste à capitaliser un domaine de connaissance au niveau d'un DSML (Domain Specific Modeling Language) et de son environnement.

La transformation est la génération automatique d'un modèle cible à partir d'un modèle source, suivant une définition de la transformation, où la définition de transformation est un ensemble de règles de transformation qui, réunies, décrivent comment un modèle dans un langage source peut être transformé dans un langage cible.

Les langages de modélisation permettent d'exprimer les méta-modèles et d'effectuer les transformations des modèles.

MOF (Meta Object Facility) a été adopté par OMG. La spécification de MOF définit un langage abstrait et un Framework pour la spécification, la construction et la gestion des méta-modèles génériques. De plus, MOF définit une plateforme pour l'implémentation de modèles décrits par les méta-modèles.

ECORE est un langage de méta-modélisation qui fait partie d'EMF (Eclipse Modeling Framework) et qui est le résultat des efforts du projet ETP (Eclipse Tools Project). EMF est un Framework de modélisation et génération de code pour supporter la création d'outils et d'applications dirigés par les modèles.

L'ingénierie dirigée par les modèles (IDM) est prometteuse et créatrice, plusieurs approches et travaux connexes ont été exposés dans ce chapitre.

CHAPITRE 3

Web sémantique et Ontologies

Sommaire

3.1 Introduction

Le Web sémantique fournit dorénavant un saut qualitatif par rapport au Web actuel. Il permet de connecter différentes ressources du Web (documents et données, au sens large) par des liens sémantiques en construisant ainsi des graphes de connaissances structurés. Ceux-ci sont ensuite exploitables pour différentes tâches, en déployant ces structures et la sémantique ainsi créée. Le modèle de représentation RDF (Resource Description Framework) aide à réaliser aisément cette approche dans le Web distribué. La standardisation de ce modèle, mais aussi celle des vocabulaires décrivant les notions utilisées, permettent d'obtenir une interopérabilité généralisée. Les ontologies, au sens de l'ingénierie des connaissances, jouent alors un rôle majeur. Ces notions de Web sémantique et d'ontologie sont ainsi étroitement liées.

L'initiative du Web sémantique, impulsée par le W3C, propose un saut qualitatif à partir du Web actuel qui, d'une certaine manière, reprend la vision originelle de Berners-Lee, pour reprendre Berners-Lee et al.. Dans The Semantic Web, « le Web sémantique est une extension du Web actuel dans laquelle l'information est munie d'une signification bien définie, permettant aux ordinateurs et aux personnes de mieux travailler en coopération ».

Dans ce chapitre nous developperons le concept du Web sémantique, son architecture et ces principaux composants.

Les deux composants fondamentaux et primordiaux du Web sémantique sont l'ontologie et les métadonnées afin de rendre le web « sémantique ». Par « sémantique », il ne s'agit pas d'envisager que la machine « comprenne » au sens humain le contenu de l'information de chacune de ces bases. Par contre, ces informations (données) peuvent faire l'objet d'un langage structuré décrivant ces données, et suffisamment standardisé pour être partageable par des machines. Ce langage est appelé « métadonnées » (des données décrivant des données, ou *metadata*).

Nous enoncerons également les définitions du concept ontologie, ses caractéristiques, ses composants : les étapes de la construction des ontologies, les différentes classifications des ontologies et leurs méthodes de construction.

Une ontologie consiste en une représentation formelle d'un domaine de connaissance sous la forme de terminologies dotées de relations sémantiques. Les ontologies jouent un rôle important dans différents domaines, et spécialement dans le cadre du Web Sémantique. Elles fournissent en effet les structures conceptuelles utilisées pour la description des ressources du Web, et permettent ainsi un traitement automatique de l'information.

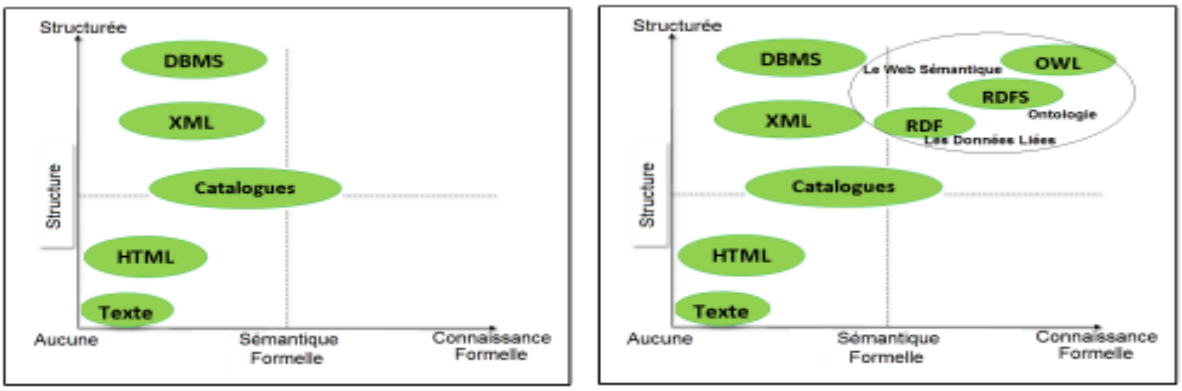
Dans ce chapitre, nous préciserons les outils de développement des ontologies et leurs langages. En particulier le langage OWL le langage OWL 2.

Nous decrirons par la suite la logique de descriptions, qui constitue une famille de langages de représentation de connaissances. Les logiques de descriptions permettent de représenter les connaissances d'un domaine de référence à l'aide de concepts (classes d'individus), de rôles (relations entre classes) et d'individus. Une sémantique est associée aux concepts, aux rôles et aux individus par l'intermédiaire d'une interprétation. Les concepts et les rôles sont organisés en hiérarchies sur lesquelles opèrent les processus de classification et d'instanciation, qui sont à la base du raisonnement terminologique. Nous discuterons en particulier sur la syntaxe et la sémantique de la logique de description.

3.2 Définitions du web sémantique

Le Web actuel est essentiellement syntaxique, dans le sens que la structure des documents (ou ressources au sens large) est bien définie, mais que son contenu reste quasi inaccessible aux traitements machines. Seuls les humains peuvent interpréter leurs contenus. La nouvelle génération de Web – Le Web sémantique – a pour ambition de lever cette difficulté. Les ressources du Web seront plus aisément accessibles aussi bien par l'homme que par la machine, grâce à la représentation sémantique de leurs contenus.

Le Web sémantique, concrètement, est d'abord une infrastructure pour permettre l'utilisation de connaissances formalisées en plus du contenu informel actuel du Web, même si aucun consensus n'existe sur les limites de cette formalisation. Cette infrastructure doit permettre d'abord de localiser, d'identifier et de transformer des ressources de manière robuste et saine tout en renforçant l'esprit d'ouverture du Web avec sa diversité d'utilisateurs. Elle doit s'appuyer sur un certain niveau de consensus portant, par exemple, sur les langages de représentation ou sur les ontologies utilisées. Elle doit contribuer à assurer, le plus automatiquement possible, l'interopérabilité et les transformations entre les différents formalismes et les différentes ontologies. Elle doit faciliter la mise en œuvre de calculs et de raisonnements complexes tout en offrant des garanties supérieures sur leur validité. Elle doit offrir des mécanismes de protection (droits d'accès, d'utilisation et de reproduction), ainsi que des mécanismes permettant de qualifier les connaissances afin d'augmenter le niveau de confiance des utilisateurs. La figure 3.1 exprime la vision du web sémantique



(a) La structure du web actuel (b) la structure et la sémantique du web sémantique (Dimitrov 2010)

Figure 3. 1 : la vision du web sémantique.

Le web sémantique permet de présenter d'une façon explicite le sens des informations, et la granulité de ces informations s'appuie sur les ressources et des faits plutôt que sur des documents. Contrairement au web actuel, les ressource du web sémantique ont des identificateurs uniques et, à travers la sémantique explicite de l'information, la machine peut traiter les données de façon automatique, et relier l'ensemble. Cela permet de régler le problème de l'hétérogénéité et d'assurer l'interopérabilité sémantique.

Généralement, les informations du web actuel sont représentées par les langages comme XML, HTML, etc (voir la Figure 3.1.a). Cependant, ces langages sont concernés par la structure et la présentation des données au lieu de leur sémantique.

À l'inverse du web actuel, les informations du web sémantique (la Figure 3.1.b) sont présentées par des langages comme RDF, OWL, afin de structurer et sémantiser les données.

Afin de clarifier l'expression « web sémantique » ; plusieurs définitions ont lui été attribuées ; citons :

L'expression Web sémantique, donnée par Tim Berners-Lee au sein du W3C qui définit le web sémantique: « **The semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation**» (Berners Lee, Hendler et Lassila 2001).

" Le Web sémantique est une extension du Web actuel (prolongation du Web actuel), dans laquelle l'information reçoit une signification bien définie, améliorant les possibilités de travail collaboratif entre les ordinateurs et les personnes."

The Semantic Web can be defined as "**an extension of the World Wide Web using new technologies and standards dealing with interpretation of exchanged data and also with automatically inferring the useful information from these data**". (Berners Lee, Hendler et Lassila 2001).

Selon W3C (W3C 2009), le web sémantique est un groupe de méthodes et de technologies qui permettent aux machines de comprendre la signification (ou sémantique) des informations contenues sur le World Wide Web. Comme déjà indiqué, l'expression web sémantique a été inventée par Berners-Lee et al.. (Berners Lee, Hendler et Lassila 2001), l'inventeur du World Wide Web, et actuel directeur du World Wide Web Consortium (W3C). Ainsi, on peut considérer que la définition du web sémantique donnée par le W3C est la définition la plus juste.

Le Web sémantique (Berners Lee, Hendler et Lassila 2001) désigne un ensemble de technologies visant à rendre le contenu des ressources du World Wide Web accessible et utilisable par les programmes et agents logiciels, grâce à un système de métadonnées formelles utilisant notamment la famille de langages.

L'objectif du Web sémantique est de rendre explicite le contenu sémantique des ressources dans le Web (documents, pages web, services, etc.). Les machines et les agents logiciels pourraient "comprendre" les contenus décrits dans les ressources et faciliter les tâches de traitement des informations de façon plus automatique et plus efficace.

3.3 Les composants du web sémantique (LEHIRECHE 2021)

Les deux composants fondamentaux et primordiaux du Web sémantique sont l'ontologie et l'annotation sémantique. Les ontologies sont la technologie dorsale pour le Web sémantique et pour le management des connaissances formalisées décrivant les ressources du Web. Elles fournissent la "sémantique" exploitable par machine des données et des sources d'informations qui peuvent être communiquées entre différents agents (logiciel et humaines), tandis que les annotations sémantiques décrivent les ressources en utilisant la "sémantique" définie dans l'ontologie. Les ressources annotées par les méta-données faciliteront la recherche, l'extraction, l'interprétation et le traitement de l'information d'une manière plus efficace.

- **Ontologie** : L'importance des ontologies est reconnue dans divers domaines de recherche comme la représentation des connaissances, l'ingénierie des connaissances, la conception de bases de données, l'intégration des données, les systèmes d'information. Les ontologies sont aussi essentielles 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.

- **Méta-données et Annotations sémantiques** : L'information du Web actuel devient plus fortement distribuée, extrêmement volumineuse, évolutive, très hétérogène et souvent très peu structurée. Afin de mieux utiliser l'information et les ressources du Web, il est nécessaire de proposer des méthodes et des outils pour représenter, manipuler et exploiter des ressources. Nous discuterons dans cette section certains moyens et outils qui visent à améliorer la communication et l'interopérabilité des applications, et à partager et exploiter l'information sur le Web. Une annotation peut être considérée d'une manière simple comme une information graphique ou textuelle attachée à un document et le plus souvent placée dans ce document. En ce qui concerne le Web, les annotations les plus courantes sont des annotations en langage naturel (informelles), des symboliques (surlignée, soulignage, italique), des notes textuelles en marge, des images, des sons, etc. Une annotation est toujours associée à l'objet qui a été annoté. Dans ce sens, les annotations sont considérées comme des méta-données. Les méta-données peuvent être définies comme étant des données relatives à d'autres données : données sur des données. La méta-donnée est une information interprétable par machine sur des ressources d'information du Web ou d'autres sources de données. Si une méta-donnée est une donnée sur une donnée, une annotation constitue un cas particulier d'une métadonnée puisqu'elle représente une nouvelle donnée attachée à une ressource documentaire, d'un point de vue plus lié à la pratique de l'annotation et de métadonnées. Puisque les données actuelles du Web sont destinées essentiellement aux humains, elles ne sont pas très bien structurées et n'ont pas de sémantique formelle. Par conséquent, un des objectifs, dans l'environnement du Web Sémantique, est de décrire le contenu des ressources en les annotant avec des informations non ambigües afin de favoriser l'exploitation de ces ressources par des agents logiciels. Cet objectif est considéré comme la tâche d'annotation consistant donc à prendre en entrée une ressource documentaire et fournir en sortie le même contenu enrichi par des annotations sémantiques basées sur des représentations de la connaissance plus ou moins formelles.

3.4 Architecture du Web sémantique (Gueffaz, 2012)

La vision courante du Web sémantique proposée dans l'article «The Semantic Web" (Berners Lee, et al., 2001) peut être représentée dans une architecture en plusieurs couches différentes (voir Figure 3.2).

Les couches les plus basses assurent l'interopérabilité syntaxique : la notion d'URI [15] 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[16] , comme l'URI, est une chaîne courte de caractères qui est aussi utilisée pour identifier des ressources (physiques) par leur localisation.

- **XML** (Extensible Markup Language) 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 et RDF-Schema** (Brickley, et al., 2014) sont considérées comme les premières fondations de l'interopérabilité sémantique. Elles permettent de décrire les taxonomies des concepts et des propriétés (avec leur signature). RDF fournit un moyen d'insérer la sémantique dans un document. L'information est conservée principalement sous forme de déclarations RDF. Le schéma RDF (RDFS) décrit les hiérarchies des concepts et leurs relations, les propriétés et les restrictions domaine/co-domaine pour les propriétés. RDF, acronyme de *Resource Description Framework,* est un modèle de données. Par abus de langage, il est commun de dire que c'est un « langage » d'assertion et d'annotation. RDF est un outil fondamental du web sémantique : il permet de définir des métadonnées afin de préciser les caractéristiques d'une information. Il établit des relations entre ressources. Il est donc particulièrement adapté aux annotations associées aux ressources du Web. RDFS (RDF Schéma) définit un vocabulaire utilisé dans les modèles de données RDF. Un document RDFS précise également les propriétés des différents objets modélisés, les domaines de valeurs possibles et décrit les relations entre ces différents objets. Les mécanismes de raisonnement mis en œuvre dans RDFS trouvent leurs fondements dans les premiers langages à base de logique pour données orientées objet, données. Par abus de langage, il est commun de dire que c'est un « langage » d'assertion et d'annotation. RDF est un outil fondamental du web sémantique : il permet de définir des métadonnées afin de préciser les caractéristiques d'une information. Il établit des relations entre ressources. Il est donc particulièrement adapté aux annotations associées aux ressources du Web. RDFS (RDF Schéma) définit un vocabulaire utilisé dans les modèles de données RDF. Un document RDFS précise

[15] Uniform Resource Identifier
[16] Uniform Resource Locator

également les propriétés des différents objets modélisés, les domaines de valeurs possibles et décrit les relations entre ces différents objets. Les mécanismes de raisonnement mis en œuvre dans RDFS trouvent leurs fondements dans les premiers langages à base de logique pour données orientées objet, tels que F-logique (Kifer, et al., 1995) et la logique de description (Baader, et al., January 2007). Une formalisation du modèle théorique de RDFS sémantique a été fournie dans le rapport technique «RDF Formalization» par Marin Draltan (Marin 2004). Les fondations et les aspects avancés de bases de données RDFS ont été étudiés dans «Foundations of Semantic Web Databases» (Gutierrez, et al., 2004). Ce dernier travail couvre des problèmes tels que le raisonnement sur RDFS et présente un ensemble de règles d'inférence qui mettent en œuvre le cœur de la sémantique du modèle théorique RDF. En outre, il étudie les problèmes tels que la complexité de l'implication (si un graphe RDFS implique logiquement un autre graphique), les formes normales de RDFS, et les requêtes réalisées sur les bases de données RDFS. Dans un style similaire, un langage de requête d'enregistrement de données de style RDF, appelé RDFLog, a été proposé par Bry, et al. dans «RDFLog : It's like datalog for RDF» (Bry, et al., January 2008). Dans «Minimal Deductive Systems for RDF» (Muñoz, et al., June 2007), Muñoz, et al. ont présenté un système minimal déductif pour RDFS. Ce travail donne des indications précieuses sur le processus de raisonnement sur RDFS.

- La couche suivante est **l'*ontologie OWL***. Elle décrit des sources d'information hétérogènes, distribuées et semi-structurées en définissant les consensus du domaine commun et partagées par plusieurs personnes et communautés (la signification de la connaissance). OWL, acronyme de *Web Ontology Language*, permet d'étendre le vocabulaire et les propriétés de RDF. Il offre une grande souplesse dans la définition des relations. Par exemple, OWL permet de préciser qu'une propriété est l'inverse de l'autre, ce qui permet d'inférer des relations non explicites. « À est le père de B», nous dit également que B est fils de A. Ceci paraît trivial, mais en informatique, il faut établir ce genre de raisonnement de base afin d'avoir des informations traitées « intelligemment ». OWL permet de construire des relations de disjonctions ou de faire des unions. Les ontologies aident la machine et l'humain à communiquer avec concision en utilisant l'échange sémantique plutôt que syntaxique. Au même niveau, on trouve la couche du langage SPARQL. Ce langage est désigné par le W3C comme le standard pour l'interrogation des graphes RDF et OWL. SPARQL est l'équivalent de SQL (Structured Query Language), (Chebotko, et al., 2006) pour le Web des données. La syntaxe SPARQL varie quelque peu de celle du SQL et il nécessaire de déclarer les espaces de nom utilisé lors de la requête. SPARQL se base directement sur les métadonnées RDF. Cela permet aux machines ou aux humains d'interroger des bases de données sur le Web, sans forcément en connaitre le schéma au préalable, ce qui permettrait un accès aux données sans

intermédiaire. Le langage SPARQL a été standardisé par le DAWG (*RDF Data Access Working Group*) le 15 janvier 2008 (Herman, 2007). Ce langage a été inventé pour interroger une base de données RDF, et signifie littéralement « protocole d'interrogation en langage RDF ».

- **La couche de règles** a pour objectif de normaliser la représentation des règles RDF. Elle comporte deux langages de règles : SWRL (*Semantic Web Rule Langage*) et RIF (*Rule Interchange Format*). SWRL est une extension d'OWL. RIF ne repose pas directement sur RDF, mais sur XML et il permet de faciliter l'utilisation et l'échange de règles entre les formats déjà existants.

- **La couche *logique*** se trouve au-dessus de la couche ontologie. Certains considèrent ces deux couches comme étant au même niveau : c'est-à-dire comme des ontologies basées sur la logique et permettant des axiomes logiques. En appliquant la logique déductive9, on peut inférer de nouvelles connaissances à partir d'une information explicitement représentée.

- **Les couches *preuve* et *confiance*** fournissent des éléments pour réaliser la 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 est nécessaire, le chiffrement est utilisé.

- **La couche *Cryptographie*** a pour but de s'assurer et de vérifier que les déclarations issues du web sémantique proviennent d'une source sure, ce qui peut être réalisé par la signature numérique des déclarations RDF.

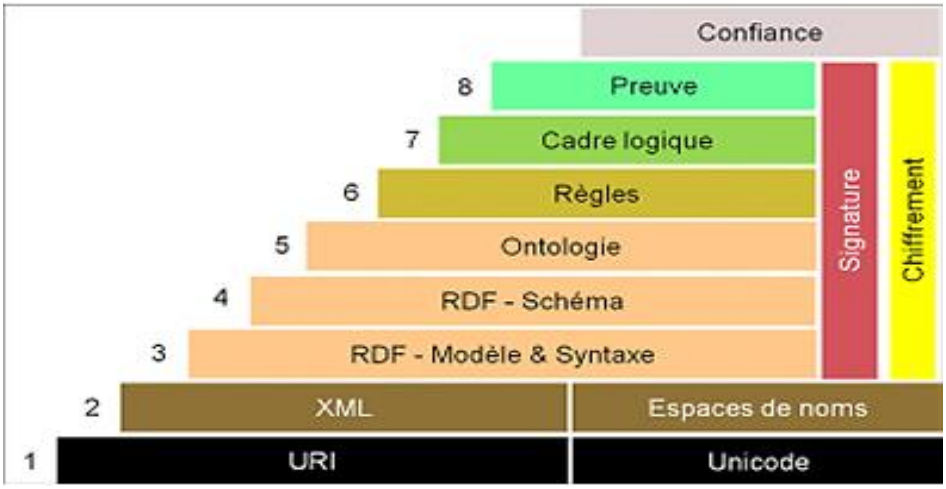


Figure 3. 2 : Architecture du Web sémantique (Berners Lee, et al., 2001).

3.5 Ontologies

3.5.1 Définitions

Le terme *ontologie* est initialement emprunté à la philosophie signifiant “explication systématique de l’existence”. Une ontologie est similaire à un dictionnaire ou un glossaire mais avec une structure détaillée et grande qui permet aux machines de traiter son contenu. Le mot *ontologie* semble susciter beaucoup de débats ainsi que de définitions dans la communauté de l’Intelligence Artificielle.

Définition de Neches et al. 1991 (Neches, et al., 1991): « une ontologie définit les termes et les relations de bases qui compose le vocabulaire d’un domaine, bien que les règles de combinaison des termes et les relations pour définir l’extension du vocabulaire ». Cette définition indique en quelque sorte ce qui est mis en œuvre pour construire une ontologie, elle identifie les termes de base et les relations entre termes, et les règles pour combiner les termes. Selon cette définition, une ontologie inclut les termes qui sont définis d’une manière explicite, en plus des connaissances qui peuvent être portées par les termes. Des années plus tard, apparaît la définition qui nous semble être la plus célèbre et la plus citée, celle de Gruber.

Définition de Gruber : Gruber a fourni la définition la plus référencée dans le domaine d'Intelligence Artificielle (Gruber, 1993) : « An ontology is a formal, explicit specification of a shared conceptualization of a domain of interestune **»,** “une ontologie est une spécification explicite d’une conceptualisation”. Une *conceptualisation* est une abstraction du monde que nous souhaitons représenter dans un certain but. La conceptualisation est le résultat d’une analyse ontologique du domaine étudié. L’ontologie est une *spécification* parce qu’elle représente la conceptualisation dans une forme concrète. Elle est *explicite* parce que tous les concepts et les contraintes utilisés sont explicitement définis. Une ontologie exprime la conceptualisation explicitement dans un langage formel. Une définition explicite et formelle permet aux agents de raisonner et d’inférer de nouvelles connaissances.

Basées sur la définition du Gruber, plusieurs définitions d’ontologies ont été proposées :

Définition de Borst 1997 (Borst, 1997) : « une ontologie est une spécification formelle d’une conceptualisation partagée ». Cette définition précise d'une part, le fait que l'ontologie doit être formelle, c'est à dire exprimée sous forme d'une logique pouvant être manipulée sur machine, et d'autre part, le fait qu'elle doit être ‘partagée’ dans la mesure où elle doit référer à la notion de groupe qui impose ainsi la mise en place d'un partage de connaissances entre ses différents individus.

Les deux dernières définitions sont fusionnées par Studer pour avoir la définition suivante :

Définition de Studer et al. 1998 (Studer, et al., 1998): « une ontologie est une spécification formelle et explicite d'une conceptualisation partagée ». La conceptualisation désigne un modèle abstrait de certains phénomènes réels par l'identification des concepts importants caractérisant un domaine. 'Explicite' signifie que le type du concept et les contraintes sur leurs usages sont explicitement définies. 'Formelle' se rapporte au fait que l'ontologie devrait être compréhensible par la machine, et enfin 'partagé' reflète le fait qu'une ontologie capture la connaissance consensuelle, c'est-à-dire qu'elle n'est pas privée d'un certain individu, mais acceptée par un groupe.

Définition de Guarino et al. : Guarino a introduit la notion d'ontologie formelle, qui est définie en tant que modélisation conceptuelle, ou une représentation de cette modélisation. "Une ontologie est un accord sur une conceptualisation partagée et éventuellement partielle" (GUARINO , et al., January 1995), où le terme conceptualisation désigne l'idée de la réalité qu'un individu ou un groupe d'individus peuvent avoir.

Il existe un autre type de définition basé sur le processus de construction des ontologies, à ce titre nous citerons :

- **Définition de Shreiber et al. 1995** (Schreiber, et al., 1995): au sein du projet KAKTUS, « une ontologie fournie le moyen pour décrire d'une manière explicite la conceptualisation des connaissances représentées dans les bases de connaissances ». Cette définition propose l'extraction d'ontologie à partir d'une base de connaissances, ce qui reflète l'approche utilisée par les auteurs pour construire une ontologie.

- **La définition de Swartout et al. 1997, extraite** de « Toward Distributed Use of Large- Scale Ontologies (Swartout, et al., 1997) Projet SENSUS, Swartout et al. définit l'ontologie comme suit « 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 ». donc une ontologie peut servir à construire plusieurs bases de connaissances qui peuvent partager la même taxonomie.
- **Définition d'Uschold et al.** : De même, Uschold définit une ontologie comme une description formelle d'entités et de leurs propriétés, relations, contraintes et comportements. De plus, les auteurs ont introduit, dans «Ontologies : Principles, methods and applications » Uschold et al., la notion de l'ontologie explicite "An explicit ontology may take a variety of forms, but necessarily

it will include a vocabulary of terms and some specification of their meaning" (Uschold, et al., January 1996).

« Une ontologie peut prendre différentes formes, mais elle inclura nécessairement un vocabulaire de termes et une spécification de leur signification, ceci inclut à la fois des définitions et une indication de la façon dont des concepts sont reliés entre eux, les liens imposent collectivement une structure sur le domaine et contraignant les interprétations possibles du terme » (Uschold, et al., 1999).

- **Définition F. Sowa 2000** : Plus tard, John F. Sowa a spécifié de façon plus précise cette notion. Dans sa définition, l'ontologie est vue comme un catalogue de types issus de l'étude des catégories d'entités abstraites et concrètes qui existent ou peuvent exister dans un domaine. Elle est définit comme suit: "The subject of ontology is the study of the categories of things that exist or may exist in some domain. The product of such a study, called an ontology, is a catalogue of the types of things that are assumed to exist in a domain of interest D from the perspective of a person who uses a language L for the purpose of talking about D. The types in the ontology represent the predicates, word senses, or concept and relation types of the language L when used to discuss topics in the domain D" (SOWA, August 2000.).

- **Définition Christophe Roche 2005** : Enfin, Christophe Roche (ROCHE, 2005)a donné une définition générique et simple "Une ontologie est une conceptualisation d'un domaine à laquelle sont associés un ou plusieurs vocabulaires de termes. Les concepts se structurent en un système et participent à la signification des termes. Une ontologie est définie pour un objectif donné et exprime un point de vue partagé par une communauté. Une ontologie s'exprime dans un langage (représentation) qui repose sur une théorie (sémantique) qui garantit des propriétés de l'ontologie en termes de consensus, cohérence, réutilisation et partage".

- **La définition formelle des ontologies**

Une formalisation claire et standardisée des ontologies en informatique est nécessaire. Nous utiliserons la définition suivante, basée sur la formalisation donnée par Ehrig, et al., dans «QOM – Quick ontology mapping» (Ehrig, et al., 2004), et complétée par Essaid, et al. (Essaid, et al., 2011) dans «Gestion du conflit dans l'appariement des ontologiesExtraction et Gestion des Connaissances».

$O = \{C, H_C, I, R, H_R, R_C, \ , A, N\}$. Une ontologie O est définie comme un 8-uplet composé :

- D'un ensemble de concepts C organisés en hiérarchie de classes et sous-classes
- D'un ensemble de relations de subsomption H_C reliant les classes et les sous-classes

- D'un ensemble I des instances des concepts, souvent appelés individus
- De relations logiques binaires R
- D'instances de ces relations entre des concepts, R_C
- D'instances de ces relations entre des instances des concepts, R_I
- Un ensemble de relations de subsomption entre les relations elles-mêmes H_R
- Un ensemble d'axiomes A permettant d'inférer de nouvelles connaissances
- D'un ensemble d'annotations N enrichissant la description des concepts, des relations ou de leurs instances.

3.5.2 Les composants d'une ontologie.

Dans «Le langage d'ontologie Web OWL Guide (Smith, Welty et Mcguinness 2004)», les auteurs mentionnent les composants les plus communs des ontologies :

- **Les concepts ou les classes** : représentent et regroupent un ensemble d'objets (d'instances) et leurs propriétés communes. Par exemple : le concept Pays est constitué de Algérie, Italie, etc. Les classes sont construites à partir de descriptions qui contraignent les conditions d'appartenance d'une instance à une classe, et la relation d'ordre entre les classes construit une hiérarchie des classes.

- **Les instances de classes (individus)** : ce sont les objets du domaine que nous souhaitons modéliser. Par exemple : les instances de la classe "personne" sont Mohammed, Houda, etc.

- **Les propriétés ou relations (properties)** : les propriétés peuvent être a nées en sous-propriétés (super-propriétés). Le sens des propriétés est généralement défini par le domaine, co-domaine, les caractéristiques et les restrictions. Les propriétés peuvent aussi être définies par d'autres propriétés, ce qui construit une hiérarchie des propriétés. Les propriétés peuvent être divisées en deux types : les propriétés de type objet et les propriétés de type donnée.

    1. **Les propriétés de type objet** : définissent les relations entre deux individus. Le domaine et co-domaine de ces propriétés sont des classes d'individus. Ces propriétés peuvent être inversées, c'est-à-dire si une certaine propriété relie un individu A à un individu B, alors sa propriété inverse reliera l'individu B à individu A. En plus de la caractéristique inversée, d'autres caractéristiques peuvent être appliquées comme : fonctionnelle, fonctionnelle inversée, transitif, symétrique, antisymétrique, réflexive, irréflexive.

    2. **Les propriétés de type donné** : définissent les relations entre un individu et une valeur de donnée. Ces propriétés ne peuvent pas avoir des propriétés inversées, la seule caractéristique qui peut être appliquée est seulement fonctionnelle.

- **Les assertions de propriété de type objet** : spécifient des relations sur une propriété donnée de type objet.

- **Les assertions de propriété de type donnée** : spécifient des relations sur une propriété donnée de type donnée.
- **Les restrictions** : permettent de décrire des classes ou des propriétés (de type objet et de type donnée) anonymes (non nommés), autrement dit l'ensemble des sous-classes, sous-propriétés et les individus qui satisfont une restriction donnée. Il y'a trois types de restrictions :

1. **Les restrictions quantificateurs** : mettent efficacement les contraintes sur les relations dans lesquelles l'individu est impliqué en (1) précisant qu'au moins un type (existentiel) de relation doit exister, ou spécifiant les seuls genres (universelles) de relations qui peuvent exister (si elles existent).

2. **Les restrictions de cardinalité** : indiquent le nombre de relations auxquelles un individu peut participer à une propriété donnée : Min, Max, etc.

3. **Les restrictions de valeur** : spécifient une relation de propriété de type objet à un individu en particulier.

- **Les axiomes** : les axiomes constituent la "connaissances de base" qui doit être vraie lors de raisonnements, c'est-à-dire, la connaissance donnée en entrée. Les "axiomes" sont souvent utilisés pour désigner les états cohérents qui peuvent être faits dans RDFS / OWL. La figure 3.3 illustre les composants de l'ontologie « Ontology Emotions ».

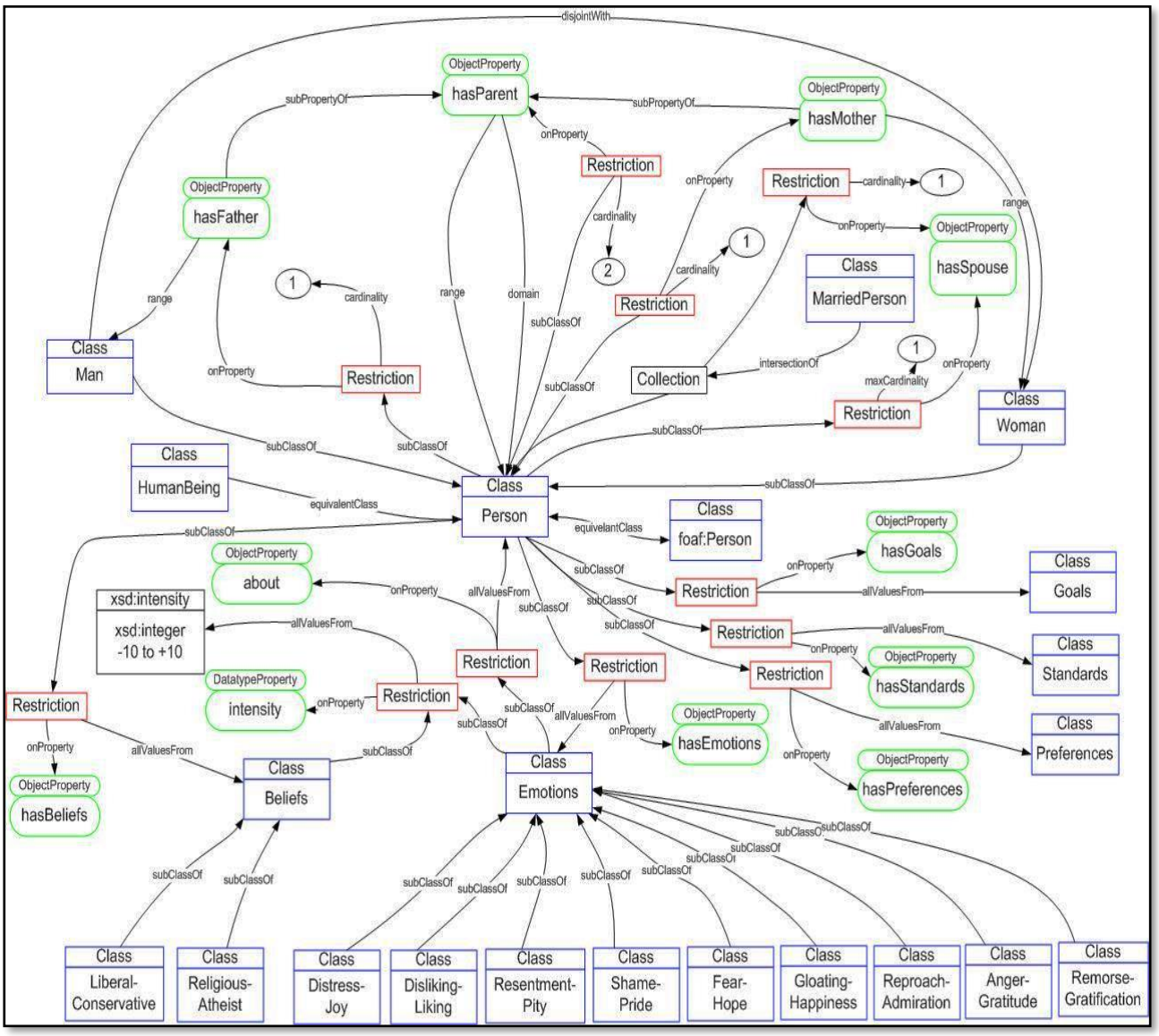


Figure 3. 3: Les composants de l'ontologie « Ontology Emotions ».

3.5.3 Les Caractéristiques des ontologies

Une ontologie est une bibliothèque de concepts bien définie. Cette bibliothèque décrit la structure de l'information pour un domaine particulier. Les ontologies possèdent les caractéristiques fondamentales suivantes (Guber, 1993) :

- **La clarté et l'objectivité** : elles doivent fournir le sens des termes définis en offrant des définitions objectives en langage naturel. La formalisation, les concepts et les relations sont exprimés dans une syntaxe claire et cohérente sur une base mathématique qui peut être traitée par des programmes informatiques.
- **La lisibilité** : Les concepts et les relations doivent être lisibles par les programmeurs, les communautés d'experts de domaine, et aussi par les utilisateurs potentiels.

- **L'exhaustivité** : Toutes les définitions des concepts ou des relations doivent être exprimées par des conditions nécessaires et suffisantes.
- **Partageabilité** : Ils sont utilisables à travers de multiples domaines d'application pour faciliter la combinaison entre les ontologies développées.

3.5.4 Les étapes de la construction des ontologies

Dans la littérature, il n'y a pas un consensus sur une méthodologie fixe pour la construction des ontologies. Plusieurs méthodologies ont été proposées dont la plus connue est celle proposée et fondée sur l'expérience de la construction de l'Entreprise Ontology par Uschold et Grüninger (Uschold, et al., January 1996). Comme la méthode est générique, ses étapes sont considérées comme la base d'un processus standard de construction. Ce processus opère selon quatre étapes fondamentales (Voir la figure 3.4) :

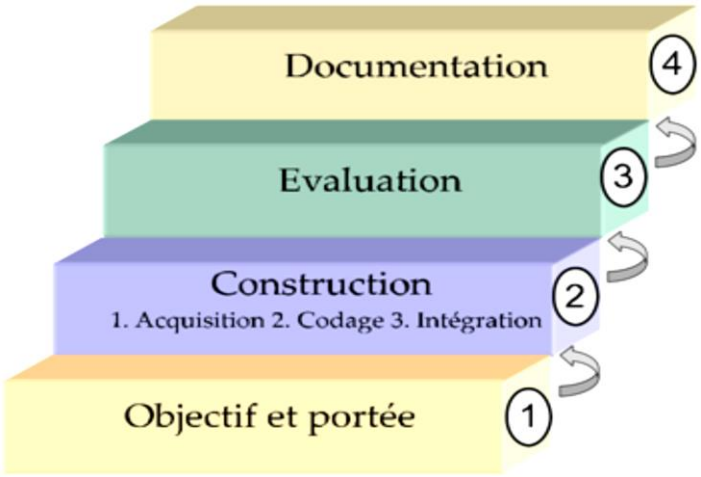


Figure 3. 4: les étapes pour la construction des ontologies.

- **L'identification de l'objectif** permet d'identifier, en termes généraux, l'objectif, la portée et les limitations de l'ontologie à construire.
- **La création de l'ontologie**, étape la plus longue et la plus difficile, contient elle-même trois sous-étapes :

1. L'acquisition des connaissances sert à définir les concepts dans un domaine donné et les relations entre eux, de manière à ne pas être ambiguës. Différentes techniques permettent de faire l'acquisition des connaissances. Elles peuvent se matérialiser par des entretiens informels avec des experts ou des entretiens structurés en vue de collecter des connaissances spécifiques et détaillées sur les concepts, leurs instances et leurs relations. Elles peuvent également être obtenues sous forme d'analyse informelle de texte pour définir les concepts

fondamentaux, ou bien sous forme d'une analyse formelle afin de définir les structures des connaissances (Fernández-López, et al., 1997).

2. Le codage, une fois les concepts et leurs relations acquises, permet de représenter l'ontologie dans un langage formel. La formalisation de l'ontologie peut être de différents degrés (Brisson, 2004) :
   - Très informel : l'ontologie s'exprime dans le langage naturel,
   - Semi-informel: l'ontologie s'exprime dans une forme structurée du langage naturel,
   - Semi-formel: l'ontologie est exprimée dans un langage artificiel défini formellement,
   - formel : l'ontologie est exprimée dans un langage formel utilisant une sémantique formelle avec des théorèmes et preuves.
3. L'intégration des ontologies existantes est l'étape qui permet de réutiliser les concepts déjà définis dans des ontologies existantes.

- **L'évaluation de l'ontologie** a été définie en s'inspirant des travaux de Gomez-Perez (GÒMEZ-PÉREZ, et al., 1995) :"to make a technical judjment of the ontologies, their associated software environment, and documentation with respect to a frame of reference. The frame of reference may be requirements specifications, competency questions, and/or the real world ". D'un autre côté, Gruber a proposé quelques critères pour l'évaluation d'une ontologie :
  - **La clarté** : les concepts de l'ontologie doivent présenter le sens voulu des termes,
  - **La cohérence** : les raisonnements construits à partir des axiomes d'une ontologie ne doivent pas aboutir à des contradictions,
  - **L'extensibilité** : l'ontologie doit être conçue de manière à ce qu'une nouvelle utilisation se fasse sans remettre en cause ce qui a été précédemment conçu,
  - **Le biais d'encodage minimum** : la spécification de l'ontologie doit être aussi indépendante que possible d'un méta-langage particulier de représentation,
  - **L'engagement ontologique minimal** : l'objectif est de permettre la spécialisation des spécifications d'une ontologie donnée selon des besoins réels,
- **La documentation** permet de renseigner les ontologies, leurs concepts importants ainsi que leurs objectifs.

3.5.6 Les typologies des d'ontologies

Nous classerons différents types d'ontologies selon qu'elles soient caractérisées par leur granularité, formalité, généralité ou calculabilité (Hadzic, et al., 2009). Une ontologie peut être classifiée en granularité importante si elle facilite la conceptualisation du domaine au niveau macro et est représentée dans un langage d'une expressivité minimale, ou en granularité fine permettant la conceptualisation du domaine au niveau micro et représentée dans ce cas à l'aide d'un langage d'une expressivité signifiante (Hadzic, et al., 2009).

Les ontologies peuvent être classifiées selon plusieurs critères. Ces derniers sont énoncés dans l'article intitulé «Apport de l'ingénierie ontologique aux environnements de formation à distance» (Psyché, et al., 2004), dans lequel Psyché, et al. ont déterminé quatre typologies d'ontologies.

- **Le niveau de formalisme de représentation**
- **Le niveau de complétude**
- **Le niveau de détail**
- **L'objet de conceptualisation**

La Fig. 3.5 éclaircit les typologies des ontologies selon les quatre dimensions.

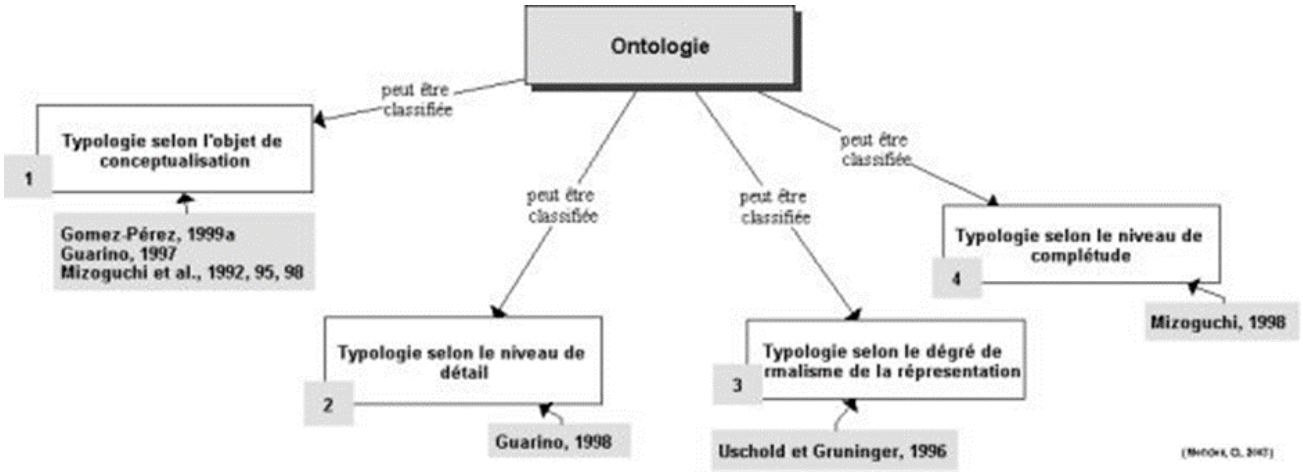


Figure 3. 5 : typologies des ontologies selon quatre dimensions.

3.5.6.1 Typologie selon le niveau du formalisme de représentation

En termes **de formalité**, la classification se fait en quatre catégories (Uschold, et al., 1996):

- **Ontologies fortement informelles** : exprimées librement dans un langage naturel,
- **Ontologies semi-informelles** : exprimées dans une forme restreinte et structurée d'un langage naturel, réduisant l'ambiguïté,
- **Ontologies semi-formelles** : exprimées dans un langage formellement artificiel,

- **Ontologies rigoureusement formelles** : composées de termes méticuleusement définis avec des sémantiques formelles, des théorèmes et preuves de certaines propriétés comme la stabilité et la complétude.

La figure 3.6 développe la typologie selon le formalisme de représentation

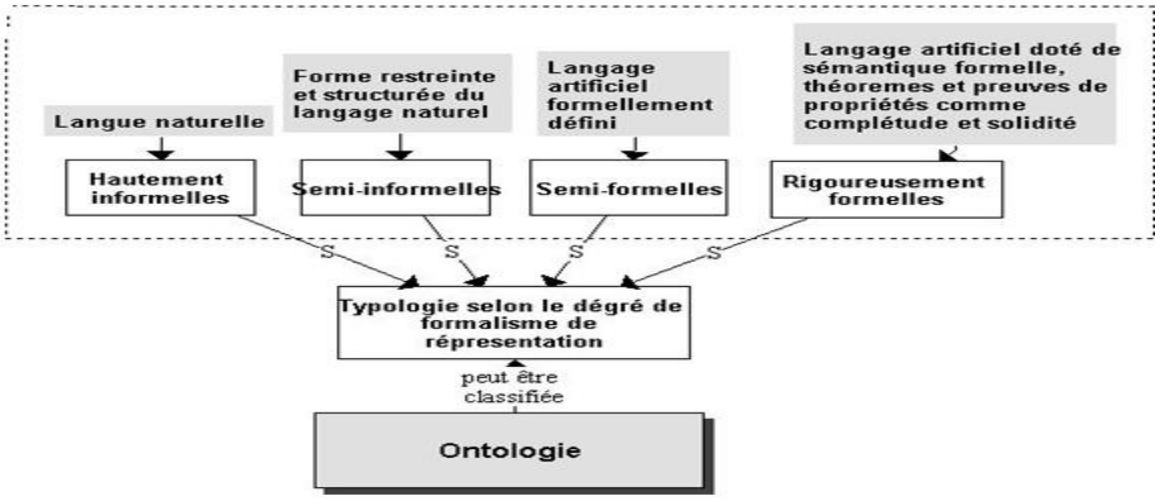


Figure 3. 6: Typologie selon le formalisme de représentation (Psyché, et al., 2004).

3.5.6.2 Typologie selon le niveau de complétude

Cette typologie a été introduite par Psyché qui s'est inspiré des trois engagements considérés par Bachimont pour la définition d'ontologie. En effet, Bachimont (BACHIMONT, 2000) a identifié trois engagements correspondant aux étapes de la modélisation des connaissances :

- Un engagement sémantique est vu comme un arbre de concepts sémantiques définis par un libellé linguistique.
- Un engagement ontologique qui décrit un ensemble des concepts référentiels (ou formels) qui se caractérisent par un terme/libellé dont la sémantique est définie par une extension d'objets.
- Un engagement computationnel traite des concepts computationnels qui sont caractérisés par les opérations qu'il est possible de leur appliquer pour générer des inférences. La figure 3.7 enrichit la typologie selon le niveau de complétude.

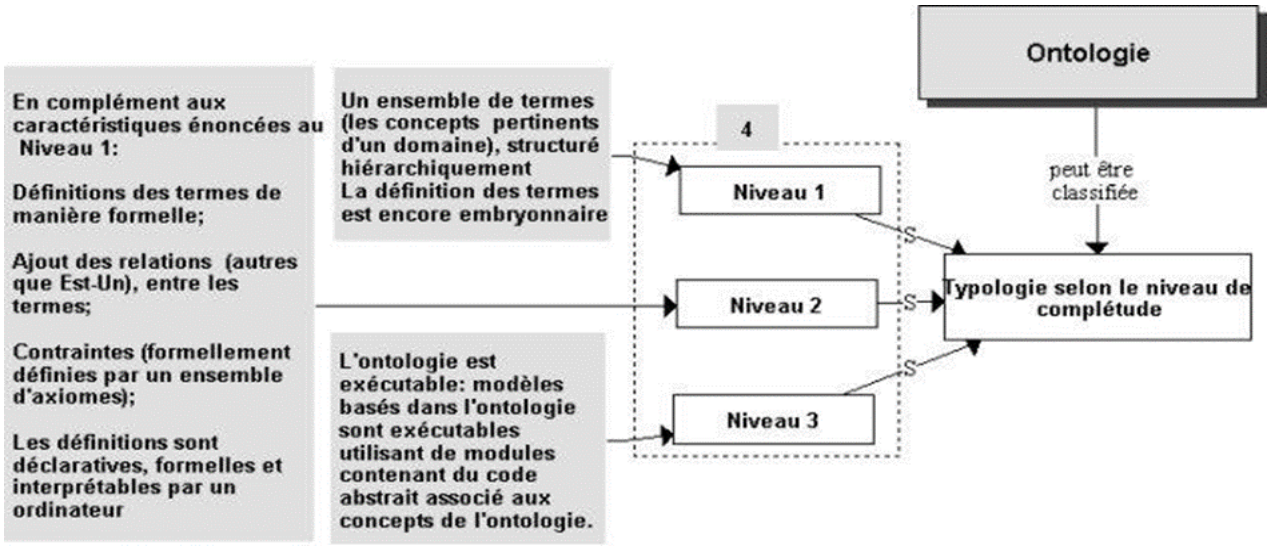


Figure 3. 7: Typologie selon le niveau de complétude (Psyché, et al., 2004).

Pour chaque engagement, un type d'ontologie est défini comme suit :

- **L'ontologie régionale** est vue comme un arbre de concepts sémantiques. Dans «Engagement Sémantique et Engagement Ontologique» (BACHIMONT, 2000), cela représente un concept sémantique est défini par un libellé linguistique. Celui-ci est emprunté à la langue du domaine. Son interprétation est contrainte par les principes différentiels : ceux qui lui sont directement associés et ceux de ses ancêtres dans l'arbre.
- **L'ontologie référentielle** décrit un ensemble des concepts référentiels (ou formels) qui se caractérisent par un terme/libellé dont la sémantique est définie par une extension d'objets. Les concepts formels sont soit des concepts sémantiques dont on reprend le libellé et auquel on associe des référents conformément à l'engagement sémantique, soit de nouveaux concepts définis formellement par intersection de concepts formels déjà définis.
- **L'ontologie computationnelle** traite des concepts computationnels qui sont caractérisés par les opérations qu'il est possible de leur appliquer pour générer des inférences.

3.5.6.3 Typologie selon le niveau de détail ou de granulité

Dans cette typologie, deux types de granularité ont été distingués par Fürst (FÜRST, 2002).

- **La granulite fine** : Quand les ontologies sont très détaillées au niveau du vocabulaire utilisé, qui est plus riche, on parle de granularité fine. Ce vocabulaire doit assurer la pertinence des concepts d'une tâche spécifique, dans un domaine particulier. Souvent, les ontologies de domaine, les ontologies de tâches et les ontologies d'applications représentent des ontologies à granularité fine.
- **La granularité large :** concerne le cas où les ontologies sont moins détaillées. Un exemple est celui des ontologies de haut niveau, car elles disposent de concepts génériques qui peuvent

être raffinés dans d'autres types d'ontologies (ontologie de domaine, de tâches et d'application). La figure 3.8 illustre la typologie selon le niveau de granulité.

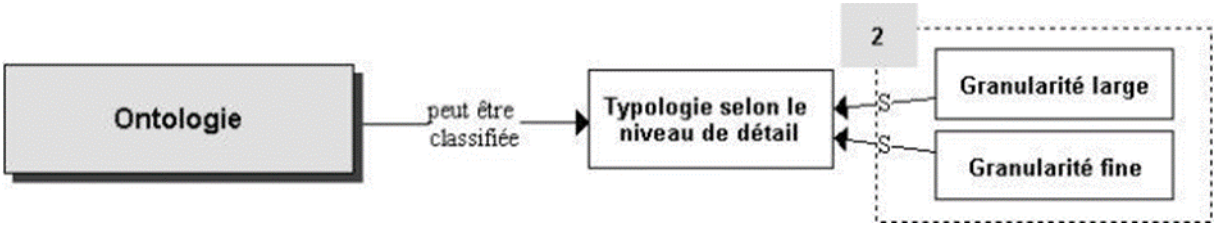


Figure 3. 8: Typologie selon le niveau de granulité (Psyché, et al., 2004)

3.5.6.4 Typologie selon l'objet de conceptualisation

Les ontologies classifiées selon leur **objet de conceptualisation** dans «Ontological Engineering : A state of the art» dirigé par (Gómez-Pérez; 1999), et dans « Some organizing principles for a unified top-level ontology», rédigé par Guarino. N. (Guarino, 1997), puis dans « A Step Towards Ontological Engineering» rédigé par Mizoguchi, R (R. Mizoguchi 1998) sont illustrés dans la Figure3.9.

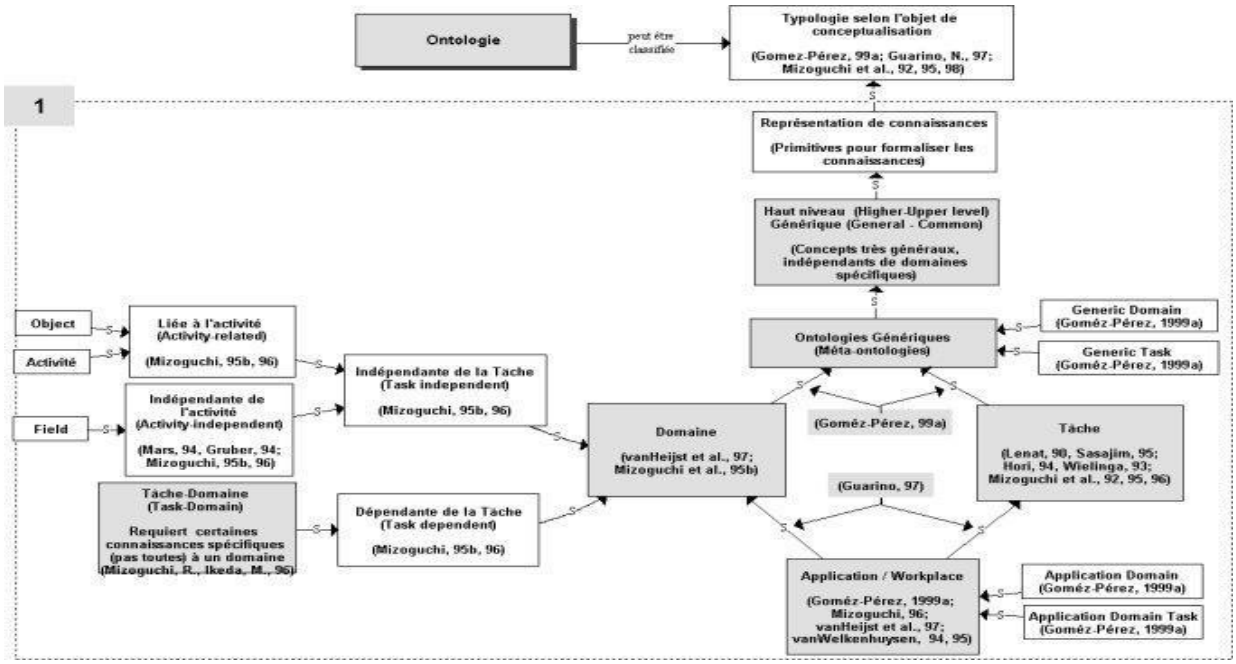


Figure 3. 9: Typologie selon l'objet de conceptualisation (Psyché, et al., 2004)

- **Les ontologies de représentation des connaissances (Knowledge Representation Ontologies)** permettent d'expliquer la conceptualisation sous-jacente aux formalismes de représentation (Davis, et al., 1993). Elles regroupent les concepts impliqués dans la formalisation des connaissances. On les désigne également comme ontologies abstraites ou de haut niveau parce qu'elles permettent de définir des concepts abstraits, et peuvent être réutilisées pour définir des

concepts spécifiques. Un exemple d'ontologie de ce type est la Frame Ontology utilisée dans Ontolingua (Gruber, 1993), de même que l'ontologie de Sowa (SOWA, August 2000.) Knowledge Representation (KR), qui est une ontologie générique à une visée universelle.

- **Les ontologies de haut niveau ou ontologies fondamentales (Top-Level Ontologies)** sont des ontologies indépendantes du domaine. La principale utilisation de ces types d'ontologies est pour supporter l'interopérabilité sémantique des ontologies de domaines spécifiques, même si elles peuvent aussi être utilisées pour l'alignement d'ontologies. Nous citerons dans cette catégorie DOLCE[17] (Descriptive Ontology for Linguistic and Cognitive Engineering), une ontologie développée pour la linguistique et les sciences cognitives, et SUMO[18] (Suggested Upper Merged Ontology) qui nait d'une fusion d'autres ontologies de haut niveau. Parmi ces ontologies on citera souvent " Upper Cyc ". Elle contient presque 3000 termes relatifs aux concepts les plus généraux sur les connaissances humaines " Every concept one can imagine can be correctly linked into the Upper Cyc Ontology ". Elle a été construite en une douzaine d'années. Les ontologies supérieures ou de Haut niveau visent à étudier les catégories des choses qui existent dans le monde, comme les concepts de haut niveau d'abstraction tels que les entités, les événements, les états, les processus, les actions, le temps, l'espace, les relations et les propriétés. L'ontologie de haut niveau est fondée sur la théorie de l'identité, la méréologie (theory of whole part), et la théorie de la dépendance. Ces concepts sont indépendants d'un domaine ou d'un problème particulier.

- **Les ontologies génériques sont appelées**, également, **des méta-ontologies ou-Core" ontologies**". Elles décrivent des concepts génériques moins abstraits que ceux décrits par des ontologies supérieures. Dans cette classe, citons SUMO (Suggested Upper Merged Ontology) développée dans le cadre du projet IEEE SUO (Standar Upper Ontology). L'objectif assigné à SUMO est de constituer un standard pour permettre l'interopérabilité sémantique entre les systèmes d'information. Actuellement, SUMO comporte plusieurs centaines de concepts et de relations généraux (Inaba, et al., 2000).

- **Les ontologies de niveau intermédiaire ou ontologies utilitaires** servent de pont entre les ontologies de domaine et les ontologies de haut niveau.

- **Les ontologies de domaine (Domaine Ontologies)** contiennent des connaissances propres à un domaine de connaissances. Elles décrivent le vocabulaire lié à des domaines particuliers comme la physique, la mécanique, la chimie, la médecine et la modélisation d'entreprise. Elles sont

[17] http ://www.ontology4.us/Ontologies/Upper-Ontologies/Dolce%20Ontology/index.html

[18] http ://www.adampease.org/OP/

réutilisables pour plusieurs applications sur ce domaine. L'ontologie Ménélas (Zweigenbaum, 1993) est un exemple d'ontologie de domaine, celui des maladies coronariennes, rassemblant des concepts et leurs relations structurés à partir de la relation « sorte de ». Ménélas comprend également des lexiques sémantiques et morphosyntaxiques des mots simples et composés. Cette ontologie est dédiée à l'analyse automatique de compte-rendu d'hospitalisation. Selon Guarino (GUARINO , et al., January 1995), les ontologies de domaine sont définies de deux manières :

- **Des ontologies spécifiques à un domaine particulier**, contenant le vocabulaire spécifique à un domaine précis,
- **Des ontologies de tâches (Task Ontologies)** qui contiennent l'ensemble des tâches réalisées dans un domaine particulier. Elles décrivent un vocabulaire en relation avec une tâche ou une activité générique. Elles fournissent un ensemble de termes au moyen desquels on peut décrire, au niveau générique, comment résoudre un type de problème. D'après Mizoguchi (Mizoguchi, et al., 2001), l'ontologie de tâche caractérise l'architecture computationnelle d'un système à base de connaissances qui réalise une tâche. Les ontologies de tâches peuvent dépendre des ontologies de haut niveau. Par exemple, dans le domaine de la modélisation et du manufacturing des entreprises, on trouve l'ontologie TOVE (TOronto Virtual Entreprise) dans laquelle des modèles d'entreprise peuvent être représentés pour que le système réponde à certaines questions[19]. En effet, la tendance actuelle des entreprises est d'identifier, de décrire les types de problèmes, de sélectionner de nouveaux processus pouvant y apporter des solutions, les évaluer, etc. Toutefois, cette tâche requiert nombre d'acteurs, à tout instant, et leur coopération, à tous les niveaux de la hiérarchie. C'est pourquoi il serait avantageux de recourir à une modélisation des processus d'activités avec une représentation des processus, ressources, produits, qualités, organisation, ensuite, de disposer d'un outil d'aide à la décision.

▪ **Les ontologies d'applications (Application Ontologies)** dépendent, selon Guarino, à la fois d'un domaine particulier et d'une tâche spécifique (voir la figure 3.9). Elles contiennent des connaissances du domaine nécessaires à une application donnée, elles sont spécifiques et non réutilisables. Généralement, les ontologies d'application combinent des éléments d'ontologies de domaine et d'ontologies génériques choisies en fonction des méthodes spécifiques pour réaliser la tâche visée. Elles sont rarement réutilisables pour une autre application. On peut citer, par exemple, **PhysSys** (PIM, et al., 1997) qui exploite l'ontologie Eng-Math, couvrant tous les aspects liés à la

[19] http ://www.eil.utoronto.ca/entreprise-modelling/tove/

modélisation mathématique en ingénierie (Gruber, et al., 1994). D'autres exemples d'ontologies d'application sont CO et GO. **CO** (**C**himical **O**ntology) est une ontologie dans le domaine de la chimie qui permet d'identifier les groupes fonctionnels chimiques trouvés dans des inter-acteurs de petites molécules (Feldman, et al., 2005). **GO** (**G**ene **O**ntology) est une ontologie qui vise à établir un vocabulaire structuré et contrôlé pour décrire certains domaines de la biologie moléculaire et cellulaire[20]. Dans le domaine juridique, l'ontologie LKIF Core Legal Ontology est employée pour organiser et représenter des concepts juridiques[21].

La figure. 3.10 récapitule les types d'ontologie selon Guarino.

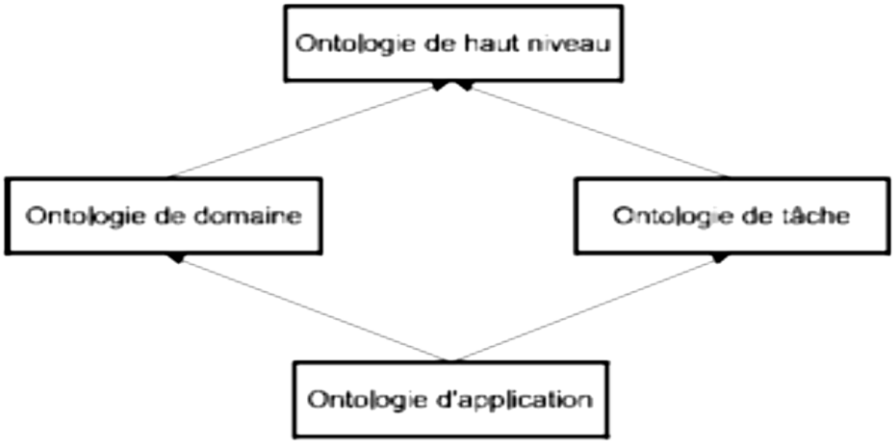


Figure 3. 10: Types d'ontologie selon Guarino (GUARINO , et al., January 1995).

- **Les ontologies d'information** : elles spécifient la structure des enregistrements d'une base de données. Les schémas de base de données en sont un exemple. Elles proposent un cadre de représentation de la connaissance stockée mais ne spécifient pas de détails sur la sémantique des champs.

- **Les ontologies terminologiques ou linguistiques** spécifient les termes utilisés pour représenter la connaissance d'un domaine. Un exemple de ce type d'ontologie est le réseau sémantique UMLS (Unified Medical Language System) (Lindberg , et al., 1993). D'autres typologies ont été adoptées pour classifier les ontologies.

3.5.6.5 Typologie selon la richesse de la structure interne

Lassila et McGuinness (Mcguinness, et al., 2000) proposent une classification d'ontologies en fonction des données que l'ontologie décrit et en fonction de la richesse de sa structure interne à travers les catégories suivantes (Zghal, 2010).

[20] http ://www.geneontology.org/
[21] http ://www.estrellaproject.org/lkif-core/

Le vocabulaire contrôlé : est un ensemble de termes définis par un groupe de personnes ou une communauté (par exemple les catalogues).

- Le glossaire : représente un ensemble de termes avec leur signification.
- Le thésaurus : est défini par un ensemble de termes organisés suivant un nombre restreint de relations. Ces relations peuvent être entre termes synonymes ou entre termes préférés.
- La hiérarchie informelle : organise des catégories à partir de la notion générale de généralisation/spécification (par exemple la hiérarchie proposée par Yahoo qui représente une catégorisation des thèmes de recherches).
- La hiérarchie formelle : est une hiérarchie dont la structure est déterminée par des relations de généralisation.
- La hiérarchie formelle avec instances : est similaire à la catégorie précédente mais elle inclut des instances.
- Le frame : représente une ontologie contenant des classes avec des propriétés pouvant être héritées.
- L'ontologie avec restrictions de valeurs : est une ontologie contenant des restrictions sur les valeurs des propriétés.
- L'ontologie avec contraintes logiques : est une ontologie pouvant contenir des contraintes entre les constituants, définies dans un langage logique. La figure 3.11 illustre la **typologie** selon la richesse de la structure interne.

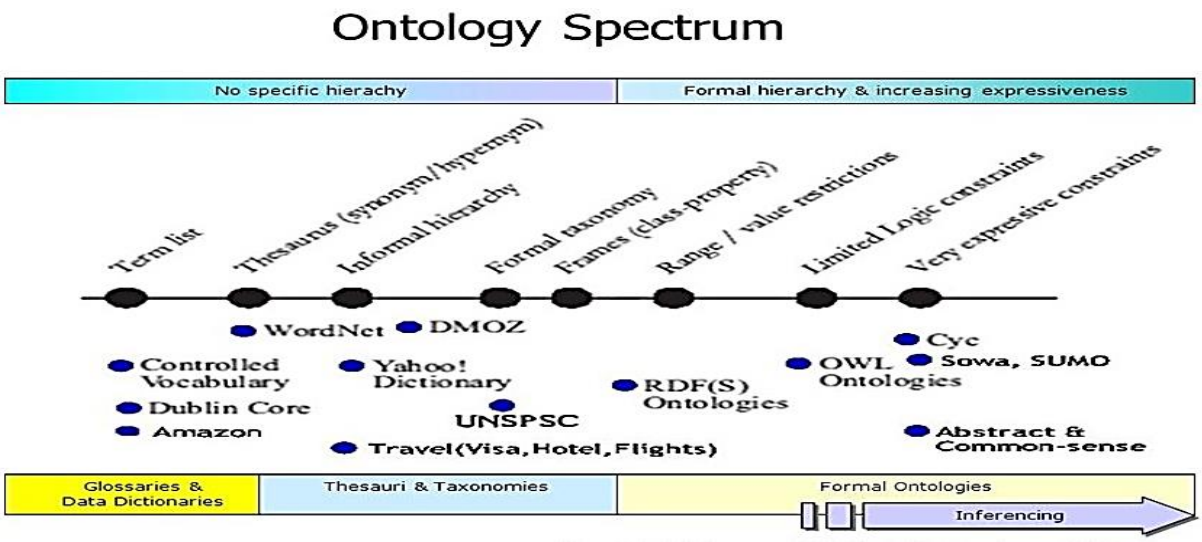


Figure 3. 11: Typologie selon la richesse de la structure interne.

3.5.6.6 Typologie selon le niveau de complexité :

- **Les ontologies légères** : sont des structures taxonomiques simples, composées de primitives ou de termes et les définitions qui leur sont associées. Les ontologies légères ne sont pas ou peu axiomatisées, et ne garantissent pas un engagement ontologique. Elles sont réduites à des structures taxonomiques entre des termes considérés comme appropriés.
- **Les ontologies lourdes** *:* ces ontologies ont recours à une axiomatisation riche, à la différence des ontologies légères, les ontologies lourdes représentent explicitement l'engagement ontologique. Le but de l'axiomatisation est d'exclure des ambiguïtés terminologiques et conceptuelles, due aux interprétations fortuites.

3.5.6.7 Typologie selon leur propos

Selon leur propos, les ontologies peuvent être classées enselon les catégories suivantes :

- **Les ontologies d'application** : l'ontologie d'application est la plus spécifique, dont les concepts correspondent souvent aux rôles joués par les entités du domaine, tout en exécutant une certaine activité (Maedche, et al., 2002). Elle contient toutes les définitions nécessaires pour décrire la connaissance requise pour une application particulière.
- **Les ontologies de référence** : elles sont définies dans « The Blackwell Guide to the Philosophy of Computing and Information » (Smith, 2004), d'après Smith, ces ontologies sont conçues pour décrire un domaine correctement. Les ontologies de référence sont utilisées durant le processus de développement d'applications pour une compréhension mutuelle entre des agents, appartenant à des communautés différentes, pour établir un consensus dans une communauté, qui a besoin d'adopter de nouveaux termes, ou tout simplement pour expliques le sens des termes aux nouveaux arrivants dans la communauté.

3.5.6.8 Typologie selon le nombre de points de vue de concepteurs

Yiling Lu dans sa thèse (Yiling, 2003), a classé les ontologies en deux types :

- **Les ontologies inspirationnelles** : les ontologies inspirationnelles sont conçues selon un seul point de vue du concepteur de domaine
- **Les ontologies collaboratives** : à la différence des ontologies inspirationnelles, les ontologies collaboratives sont conçues selon multiples points de vue de différents concepteurs et acteurs de domaines.

3.4.6.9 Typologie selon le nombre de points de vue des futurs utilisateurs

A travers ce que nous avons présenté dans les sections précédentes, il ressort que la notion d'ontologie constitue l'une des approches les plus efficaces pour représenter les connaissances d'un domaine, mais l'existence de plusieurs façons d'appréhender ces connaissances selon différents points de vue donne lieu à la naissance d'une autre notion d'ontologie qui est l'ontologie multipoints de vue. Dans nos travaux, nous nous sommes rendu compte que nous pouvons aussi classer les ontologies selon le nombre de points de vue de futurs utilisateurs de l'ontologie. Nous distinguons alors :

- **Les ontologies mono-point de vue** *:* elles sont conçues selon un point de vue global des futurs utilisateurs.
- **Les ontologies multi-points de vue** *:* Une ontologie étant une représentation subjective de la réalité, il semble difficile (voire impossible) d'imaginer que les concepteurs d'ontologies puissent se mettre d'accord sur une représentation commune de la réalité, et que celle-ci soit adapté aux besoins de tous les utilisateurs (DJAKHDJAKHA, 2014).

3.4.7 Les méthodes de construction des ontologies (Corcho, et al., 2003)

Les méthodologies peuvent porter sur l'ensemble du processus et guider l'ontologiste dans toutes les étapes de la construction, bien qu'aucune méthodologie générale n'ait pour l'instant réussi à s'imposer. Une méthodologie comprend l'ensemble de procédures, de techniques de processus d'activités, et de directives qui assistent le développement de l'ontologie durant son cycle de vie et suivant une approche donnée (bottom-up, -top -down, et middle-out) (Casellas, June 2011).

Il existe trois types de méthodes pour la construction d'ontologie : des méthodes manuelles, automatiques et mixtes. Dans la première, les experts créent une nouvelle ontologie d'un domaine ou étendent une ontologie existante, comme par exemple l'ontologie Wordnet.

Dans la méthode automatique, l'ontologie est construite par des techniques d'extraction des connaissances : les concepts et leurs relations sont extraits de bases de connaissances et ensuite vérifiés par les inférences. Enfin, la méthode mixte, les techniques automatiques permettent d'étendre des ontologies qui ont été construites manuellement, tout comme la base des connaissances Cyc[22] .

[22] . http ://www.opencyc.org/

Il existe une multitude de méthodologies et de nombreux critères de construction d'ontologies ont été proposés parmi lesquelles nous citerons :

- **METHONTOLOGY** a été définie par Gomez-Perez à l'université Polytechnique de Madrid (Fernández-López, Gomez-Perez et Juristo 1997). La figure 3.12 montre le cycle de vie d'une ontologie constitué de plusieurs étapes : la spécification, la conceptualisation, la formalisation, l'intégration, l'implémentation et la maintenance. Chaque étape peut intégrer une phase d'acquisition des connaissances, documentation et évaluation. Ce procédé cyclique rend cette méthode très attractive pour construire des ontologies dynamiques. METHONTOLOGY recommande la suite d'outils Protégé 2000 et web DE.METHONTOLOGY, et est implémentée dans l'environnement Web ODE[23].

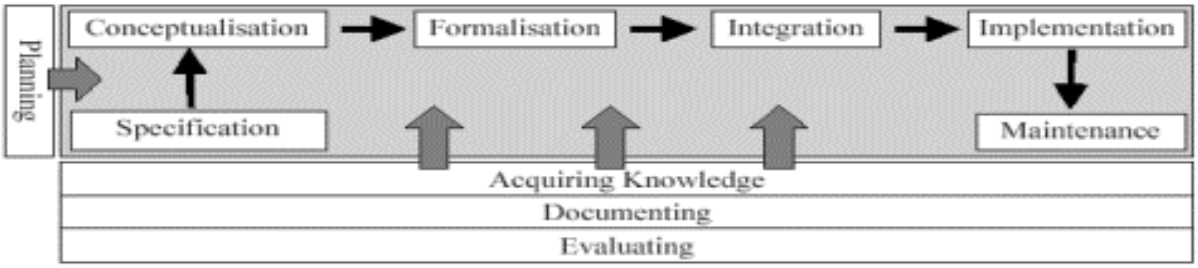


Figure 3. 12: Cycle de vie METHONTOLOGY.

- **OnToKnowledge** : OntoKnowledge est une méthodologie qui recommande un procédé itératif de développement, comme nous montre la figure 3, elle comporte quatre phases principales : une phase de spécification de condition, une phase d'amélioration, une phase d'évaluation et une phase d'application et d'évolution. OnToKnowledge propose l'acquisition des connaissances en spécialisant une ontologie générique, et finalement recommande la suite d'outils OntoStudio comme système de développement des ontologies. Le principal auteur est GRÜNINGER (Gómez-Pérez, Fernández-López et Oscar 2004) . La fig. 3.13 récapitule le cycle de vie OnToKnowledge.

[23] . WebODE est une plate-forme de conception d'ontologies fonctionnant en ligne. Elle fait suite à ODE, un éditeur qui assurait fidèlement le support de la méthodologie maison METHONTOLOGY.

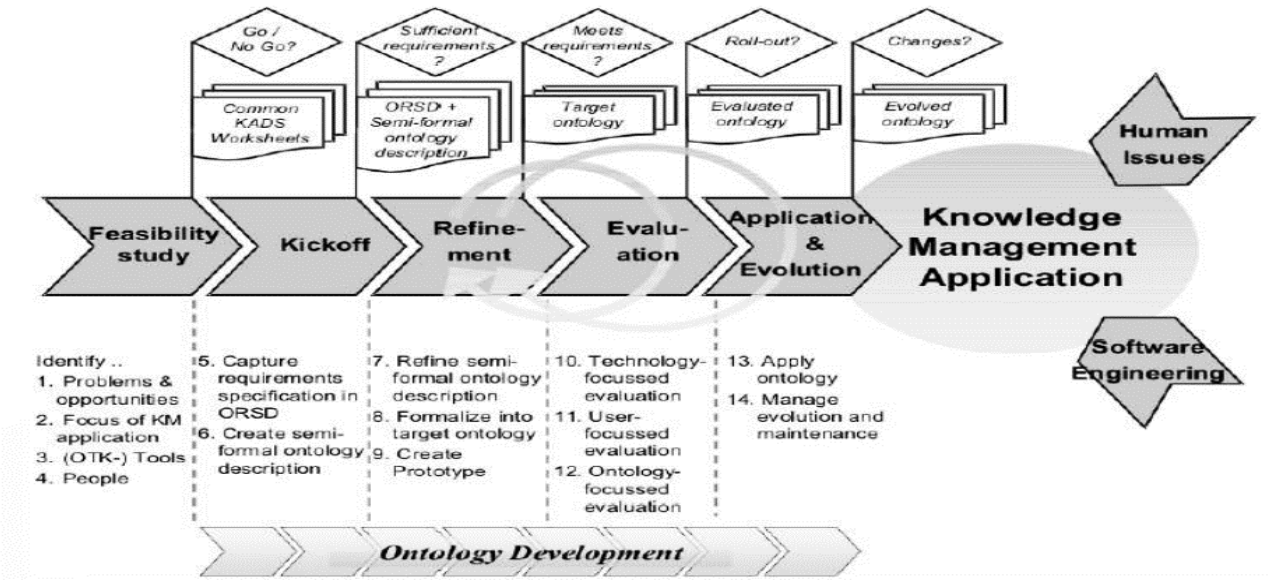


Figure 3. 13: Cycle de vie OnToKnowledge.

- **Ontology Development 101** : Cette méthode a été développée à l'université de Stanford, elle cherche à construire des ontologies formelles par la reprise et l'adaptation des ontologies déjà existantes, et propose de suivre les démarches ci-après

    1. Déterminer le domaine et la portée de l'ontologie
    2. Considérer la réutilisation des ontologies existantes
    3. Énumérer les termes les plus importants dans l'ontologie
    4. Définir les classes et hiérarchie des classes
    5. Définir les propriétés des classes
    6. Définir les facettes des Attributs
    7.Construire les instances. Elle base son développement sur les outils : Protégé 2000 et Ontolingua (N. NOY et D. L. MCGUINNESS, Noy et McGuinness Mars 2001.)

3.4.8 Les langages des ontologies

Plusieurs langages de représentation d'ontologie ont été développés afin de mettre en œuvre des raisonnements sur une ontologie. Parmi ces langages développés au niveau conceptuel pour la modélisation d'ontologie, trois grands modèles sont distingués :

- **Les langages basés sur les graphes tel que :** les réseaux sémantiques et les graphes conceptuels. D'ailleurs, les diagrammes de classe UML (Unified Modeling Language) sont utilisés comme un langage pour modéliser les ontologies (Chein, et al., 2009) (Sowa, 1987).

- **Les langages basés sur les frames** comme : OKBC[24] (Open Knowledge Base Connectivity) et KM[25] (Knowledge Machine).
- **Les langages basés sur la logique** telle que KIF (Knowledge Interchange Format)[26], CycL[27] qui sont fondés sur la logique de premier ordre, et sur la logique de description (Franz, et al., 2010).

La logique de description est l'une des logiques d'ontologies les plus importantes recommandées par W3C[28] (World Wide Web Consortium) pour le Web sémantique. Elle permet aux machines de chercher, de combiner et de traiter intelligemment les contenus du Web en fonction de leurs significations (Nigel, et al., 2006).

Dans notre approche, nous nous intéresserons aux ontologies décrites en langage de représentation des connaissances OWL 2 (Web Ontology Language). Ce langage est utilisé pour obtenir une représentation sémantiquement riche d'un domaine d'intérêt, afin de pouvoir utiliser ce dernier pour d'effectuer des tâches de raisonnement. Les langages d'ontologie amplement utilisés et basés sur la logique de description sont :

- **eXtensible Markup Language (XML)** (Michard, 1999) : La syntaxe de XML issu de SGML (Standard Generalized Markup Language) est défini par le consortium Web. Le langage XML est un langage de balisage, il va permettre de structurer les informations contenues dans un document grâce à leurs balises. Contrairement à HTML, XML offre la possibilité de créer ses propres balises. Effectivement, le but de XML n'est pas le développement d'ontologie (il ne permet pas d'imposer des contraintes sémantiques) mais il est considéré comme un langage de description et d'échange de documents structurés. En d'autres termes, XML traite la structure syntaxique des documents sans aucune spécification sémantique de cette structuration. Le langage RDF (Ressource Description Framework) est un langage qui a été défini pour cette raison.

- **RDF (Resource Description Framework)** : RDF[29] est un standard utilisé pour l'échange des ressources sur le web. Le W3C a adopté le langage RDF comme une recommandation en 1999. Les ressources dans le RDF sont identifiées par les URIs (Universal Resources Identifiers). Un document RDF est à la base exprimé sous la forme d'un graphe orienté, mais on peut le décrire par la syntaxe XML. Il fournit une sémantique simple pour des données sans faire des contraintes sur la structure du document (Baget , et al., 2003). Cependant, il existe certaines limites pour RDF ;

[24] http ://www.ai.sri.com/ okbc/
[25] http ://www.cs.utexas.edu/users/mfkb/RKF/km.html
[26] http ://www.ksl.stanford.edu/knowledge-sharing/kif/
[27] http ://www.cyc.com/documentation/syntax-cycl
[28] https://www.w3.org/
[29] http ://www.w3.org/RDF/

par exemple, nous avons besoin de connaître des informations sur les ressources identifiées. C'est pour cela que le W3C a associé à RDF le standard RDF Schéma (RDFS).

- **RDFS[30] (Resource Description Framework Schema)** est un langage extensible qui permet la représentation des connaissances. Il appartient à la famille des langages du Web sémantique publiés par le W3C. Il fournit des éléments de base pour la définition d'ontologies ou de vocabulaires destinés à structurer des ressources RDF. Construit par le W3C en 2000 comme une extension sémantique, il fournit un vocabulaire de modélisation RDF. RDFS permet de définir des mécanismes d'héritage pour la description des groupes de ressources similaires (classes) et des relations entre ces ressources (propriétés). Comme son nom l'indique, le RDFS est un langage permettant de définir des propriétés sémantiques pour les ressources par un schéma. Dans un schéma on peut définir de nouvelles ressources comme des spécialisations d'autres ressources. Les schémas contraignent aussi le contexte d'utilisation des ressources.

Cependant, RDF et RDFS souffrent de limites, tout comme l'impossibilité de raisonner, et de mener des raisonnements automatisés sur les modèles de connaissances établis à l'aide de ces langages. En conséquence, un nouveau langage, OWL (Web Ontology Language), est apparu.

OIL (Ontology Inference Layer) est un langage dédié à la spécification et à l'échange des ontologies sur le Web[31]. Il permet la représentation et l'inférence d'ontologies, en combinant des primitives de modélisation des langages de frame, avec la sémantique formelle et les modes de raisonnement des logiques descriptives. Ainsi, il représente une ontologie par un conteneur (ontology container) et des définitions ontologiques (ontology definition). Pour cela, il se base sur des formalismes tels que RDF/RDFS et XML, ce qui garantit sa totale compatibilité avec ces formalismes standards, ou des formalismes en cours de standardisation. Les liens existants entre la structure d'un document et la modélisation du domaine couvert par ce document sont étudiés dans « The Transformation of Pricing Models on the Web: Examples from the Airline Industry» (Klein, et al., 2000), Klein et al. ont effectué une comparaison entre OIL et les schémas XML.

- **OWL**[32]Plus tard OWL (Web Ontology Language) est apparu. C'est un dialecte XML fondé sur une syntaxe RDF. Il fournit les moyens pour définir des ontologies Web structurées. Il se différencie du couple RDF / RDFS par le fait que c'est un langage d'ontologies, contrairement à RDF. Si RDF et RDFS apportent à l'utilisateur la capacité de décrire des classes et des propriétés, OWL intègre, en plus, des constructeurs de comparaison des propriétés et des classes : identité,

[30] http ://www.w3.org/TR/rdf-schema/
[31] http ://www.ontoknowledge.org/oil/
[32] https://www.w3.org/TR/owl-ref/

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. Notons que de nombreux éditeurs d'ontologies sont apparus. Protégé est l'un des éditeurs d'ontologie les plus utilisés[33]. Il peut lire et sauvegarder des ontologies dans la plupart des formats d'ontologies : RDF, RDFS, OWL.

- **LOOM** est une plate-forme pour la représentation des connaissances. Son objectif principal est de construire des applications intelligentes. Les connaissances déclaratives dans LOOM sont composées de définitions, de règles, de faits, etc. Pour compiler les connaissances déclaratives, LOOM utilise un moteur déductif. Ce dernier est un classifieur qui utilise le chaînage-avant, l'unification sémantique et des technologies orientées objet. SUMO est l'une des ontologies utilisées dans LOOM par l'intermédiaire d'un outil SUMO2LOOM (Flater, 2003). Motta a, lui aussi, montré qu'il est plus facile de compléter une ontologie existante que de partir ex-nihilo. Il utilise le langage OCML. Le projet, appelé WebOnto, consiste en une application Java couplée à un serveur Web qui permet de naviguer et d'éditer des modèles de connaissance (Motta, et al., 2000).

- **ONTOLINGUA** Ontolingua est un mécanisme qui permet aux utilisateurs de créer et manipuler des ontologies[34]. Il supporte les ontologies portables pour qu'elles soient traduites dans différents systèmes. Ontolingua est basé sur le langage d'interchange KIF (Knowledge Interchange Format). Celui-ci est conçu pour l'échange de connaissances entre des systèmes informatiques répartis. Thomas Gruber a introduit la syntaxe et la sémantique utilisées dans KIF dans (Gruber, 1993). Ontolingua permet aussi de traduire des ontologies génériques en LOOM, KIF, etc.

- **SHOE** LOOM, (Simple HTML Ontology Extensions) est une extension du langage HTML qui permet aux auteurs de pages Web de générer une annotation de leurs documents, compréhensible par la machine[35]. Ce langage peut être utilisé par des agents pour la gestion des pages Web (Luke, et al., 1997). En effet, autant le langage HTML est utilisé pour rendre la connaissance facilement lisible par un humain, autant il n'est pas adapté pour permettre cette lisibilité à un système informatique. Un agent chargé d'extraire la sémantique d'un document a beaucoup de difficulté à le faire, car les données et leur présentation sont entremêlées. SHOE évite ce problème grâce à sa propriété d'inclusion des données directement lisibles et exploitables dans les pages Web. La figure 3.14 récapitule les langages basés web.

[33] http ://protege.stanford.edu/
[34] http ://www.ksl.stanford.edu/software/ontolingua/
[35] http ://www.cs.umd.edu/projects/plus/SHOE/

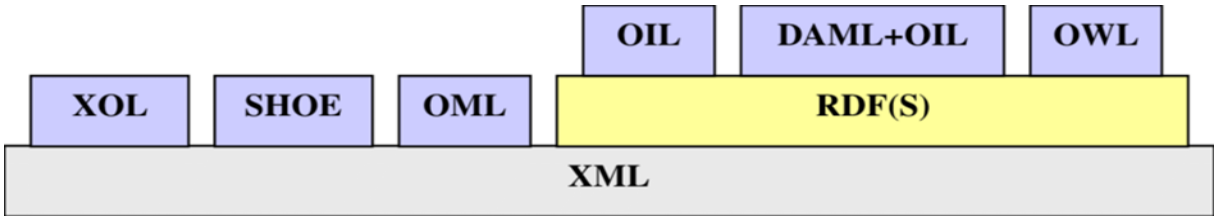


Figure 3. 14: La pyramide des langages basés Web

3.4.9 Outils de développement des ontologies

Il existe plusieurs outils pour développer et maintenir des ontologies. Dans «A survey on ontology tools. OntoWeb -Ontology-based information exchange for knowledge management and electronic commerces» (Gomez-Perez, et al., 2002), Gomez-Perez, et al. ont résumé des nombreux outils de construction d'ontologies qui ont été développés ces dernières années. Nous introduirons par la suite certains de ces outils qui sont largement utilisés par la communauté des ontologistes.

- **OILEd**[36] est un éditeur d'ontologie graphique développé par l'Université de Manchester qui permet à l'utilisateur de construire des ontologies en langage DAML+OIL. Le modèle de connaissances de l'OILEd est basé sur DAML+OIL mais il supporte aussi un paradigme des formalismes *frames* pour modéliser les connaissances (BechhoferIan , et al., 2001). Les classes sont définies en termes de leurs super-classes et les restrictions de propriété avec des axiomes additionnels. Un aspect principal de l'outil OILEd est l'utilisation du raisonneur FaCT pour classifier des ontologies et vérifier leur consistance par la traduction de DAML+OIL à la logique de description en SHIQ.

OntoEdit[37] est un projet développé à l'Université de Karlsruhe qui fournit les méthodologies et processus d'intégration d'ontologie ainsi que l'éditeur d'ontologie Selon Sure, et al.. Dans « OntoEdit: Collaborative Ontology Development for the Semantic Web ». (Sure, et al., 2002). Cet outil permet à l'utilisateur de modifier la hiérarchie de concepts (ou de classes), il implémente aussi les interfaces graphiques facilitant le processus de construction de l'ontologie (e.g. les fonctions de copie/coller, de réorganisation de la hiérarchie…). OntoEdit supporte également une interface de *plugs-in* qui permet le test de la cohérence d'une ontologie.

- **Protégé**[38] (Noy, et al., July 2002) est un outil d'édition de l'ontologie développé à l'Université de Stanford, qui est utilisé largement aujourd'hui pour élaborer des ontologies en RDF(S) et OWL. Protégé fournit un environnement de développement graphique et interactif pour aider les ingénieurs et les experts du domaine à réaliser des tâches plus facilement. Le modèle de

[36] http://oiled.man.ac.uk/
[37] http://www.ontoknowledge.org/tools/ontoedit.shtml
[38] http://protege.stanford.edu/

connaissances de Protégé est compatible avec l'OKBC (Open Knowledge Base Connectivity). Un des avantages de Protégé est son architecture ouverte et modulaire, il facilite le développement de nouvelles fonctionnalités à travers des *plugs-in* pour effectuer des opérations différentes.

- **Web ODE 16** [39], développé à l'Université Technique de Madrid, est un *benchmark* supportant l'ingénierie de l'ontologie. Il est construit sur un serveur d'application, qui fournit à la fois une haute extensibilité et rentabilité, en permettant d'ajouter facilement de nouveaux services et fonctionnalités. Les ontologies de WebODE sont représentées en utilisant un modèle de connaissances expressif basé sur la méthodologie de Methontology (Corcho, et al., 13 September 2002). Ce modèle contient des concepts (avec des instances et des attributs du concept) des relations, des axiomes et des règles.

- **DOE** : Le dernier outil présenté ici est DOE pour Differential Ontology Editor (Bachimont, et al., 2002) .Cet outil n'a pas pour ambition de concurrencer les grands environnements existants, mais plutôt de fournir un début d'implémentation à la méthodologie de structuration différentielle proposée par B. Bachimont. A l'instar des autres éditeurs, il offre une représentation graphique des arbres de concepts et des relations de l'ontologie et permet d'interagir avec les hiérarchies. L'outil assiste également la saisie des principes différentiels issus de la méthodologie en automatisant partiellement cette tâche. Le modèle de représentation de l'ontologie est finalement proche de celui du langage RDFS, à ceci près qu'il autorise la modélisation de relations n-aires. Au niveau formel, l'éditeur est capable de faire quelques inférences en vérifiant la consistance de 1'ontologie (propagation de l'arité le long de la hiérarchie des relations et héritage des domaines par exemple).

3.6 La logique de description (AIT YAKOUB 2018) (DURIF 2014)

Les logiques de descriptions (LDs) ont été introduites par Brachman (Baader, et al., 2010). C'est une famille de formalismes conçus pour représenter la connaissance d'un domaine particulier, puis la raison en dérivant de nouvelles connaissances (Napoli, 1997).

Le but de la logique de description dans l'Intelligence Artificielle est de développer des langages pour représenter formellement les connaissances d'un domaine et de rendre cette représentation disponible pour le raisonnement (Huth , et al., 2004). Comme son nom l'indique, les LDs ont une sémantique formelle, la description de concepts utilisée pour décrire un domaine, et la sémantique basée sur la logique, qui peut être donnée par une transcription en logique des prédicats ou en logique du premier ordre.

[39] http://kw.dia.fi.upm.es/wpbs/

3.6.1 Définition la logique de description

La logique de description utilise uniquement des prédicats unaires et binaires, appelés : concepts et rôles. Les concepts représentent l'ensemble des individus, et les rôles représentent les relations binaires entre les individus (ou entre les concepts). Les concepts et les rôles peuvent être définis ou primitifs. Un concept (ou un rôle) est défini s'il est structuré à partir des constructeurs. Un concept (ou un rôle) est primitif s'il est comparable à des atomes et sert de base à la construction des concepts (ou des rôles) définis. Une LD est composée de deux niveaux (Voir Figure 3.15) : (ELLI 2018)

- Un niveau terminologique TBox : dédié à la description de concepts et de rôles
- Un niveau assertionnel ABox : décrit la description de faits

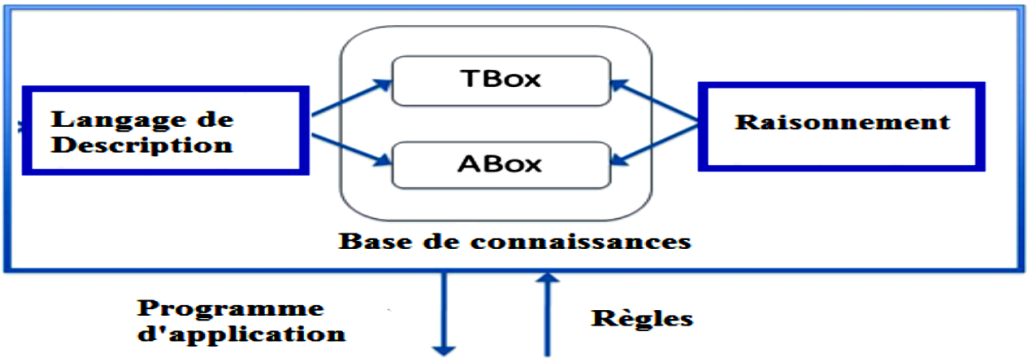


Figure 3. 15: La logique de description.

3.6.2 Syntaxe d'une logique de description (AIT YAKOUB 2018)

Une logique de description LD divise la connaissance en deux parties :

- T-box. Les informations terminologiques : un ensemble de formules relatives aux informations sur les notions basiques ou dérivées et comment elles sont reliées entre elles. Ces informations sont génériques ou globales, vraies dans tous les modèles et pour tous les individus.
- A-box. Les informations sur les individus : un ensemble de formules relatives aux informations spécifiques ou locales, vraies pour certains individus particuliers. Les logiques de description utilisent les notions de concept, de rôle et d'individu.

Pour définir des concepts dans une base de connaissances en LD, on commence par un ensemble N_C de prédicats unaires nommés 'noms de concepts' et un ensemble N_R de prédicats binaires appelés 'noms de rôles' et construit des descriptions de concepts plus complexes en utilisant les constructeurs fournis par le langage de description utilisé.

Concept primitif : les concepts primitifs servent de base à la construction de concepts définis. Ce type de concept représente le niveau le plus atomique de la connaissance en logique de description.

Concept défini : ils sont construits à partir des rôles qu'ils entretiennent avec des concepts primitifs ou avec d'autres concepts définis. Ce type de concept représente les niveaux de connaissances les plus complexes. Contrairement aux concepts primitifs, les concepts définis possèdent une définition complète et suffisante pour en déduire l'appartenance certaine d'une instance. Un rôle représente une relation binaire entre deux concepts. C'est une relation qui a pour fonction de caractériser le premier concept en précisant sa relation avec le second. Le concept non caractérisé par le rôle est appelé le co-domaine. Des restrictions de cardinalité peuvent être attribuées aux rôles. Par exemple, le rôle enfant, caractérisant le concept Parent, devrait être défini avec une cardinalité $n \geq 1$.

Les concepts et les rôles atomiques peuvent être combinés au moyen de constructeurs pour former respectivement des concepts et des rôles composés. Les différentes LDs se distinguent par les constructeurs qu'elles proposent. Dans le Tableau 3.1, nous illustrons certaines familles de langage de logiques de description. Dans ce tableau, r représente un nom de rôle, A un nom de concept et C, D des descriptions de concepts. Les noms de concept spéciaux TOP (>) et BOTTOM (⊥) dénotent respectivement le plus général et le plus spécifique. Intuitivement, l'extension de TOP inclut tous les individus possibles tandis que celle de BOTTOM est vide. A u D permet de faire la conjonction de deux concepts composés, A t D permet de faire la disjonction de deux concepts composés, ¬A est utilisé pour évoquer la négation d'un concept atomique. La restriction universelle et existentielle sont respectivement des expressions de la forme ∃r.C et ∀r.C. Les concepts de la forme ∃r.> sont généralement appelées restrictions existentielles non typées (non qualifiées), écrit ∃r.

LD	$\top$	$\bot$	$\neg A$	$\neg C$	$C \sqcap D$	$C \sqcup D$	$\exists r.C$	$\forall r.C$
$\mathcal{EL}$	×				×		×	
$\mathcal{ELU}$	×				×	×	×	
$\mathcal{ALE}$	×	×	×		×		×	×
$\mathcal{ALC}$	×	×	×	×	×	×	×	×

Tableau 3. 1 : Langages de LD.

La section suivante sera consacrée à la suite des définitions liées à la syntaxe de la logique de description (LEFRANCOIS 2016).

- **Définition 3.1** (Concept). Un concept d'une logique de description LD correspond à une classe d'éléments et est interprété comme un ensemble dans un univers donné. Les concepts

peuvent être atomiques (correspondant à un symbole de typé concept), ou complexes obtenus en utilisant les constructeurs de concepts autorisés par le langage de LD.

- **Définition 3.2** (Rôle). Un rôle d'une logique de description LD correspond à un lien entre deux éléments et est interprété comme une relation binaire sur un univers donné. Les rôles peuvent être atomiques, ou complexes, obtenus en utilisant les constructeurs de rôles autorisés par le langage de LD.
- **Définition 3.3** (Individu). Un individu d'une logique de description LD correspond à un élément d'un univers donné.
- **Définition 3.4 (**Définition de concepts et GCI) Une définition de concepts est une expression de la forme A ≡ C, où A est un nom de concept, et C'est une description de concepts complexes. Il attribue le nom de concept A à la description de concepts C. Les noms de concepts apparaissant sur le côté gauche d'une définition de concept sont appelés concepts définis et les autres sont appelés concepts primitifs. Une inclusion générale de concepts (GCI) est une expression de la forme C v D, où C et D sont deux descriptions de concepts éventuellement complexes. Il énonce une relation de sous-concept/super-concept entre les deux descriptions de concepts.

Généralement, une base de connaissances de LD se compose de deux parties appelées partie terminologique (TBox en abrégé) et partie assertionnelles (factuelle) ou (ABox en abrégé).

- **Définition 3.5. (TBox)** Une TBox est un ensemble fini non ambigu de définitions de concepts, c'est-à-dire, chaque nom a au plus une définition. Elle est appelée TBox acyclique si aucune des définitions de concepts dans la TBox ne fait référence directement ou indirectement au nom qu'il définit. Sinon, elle est appelée TBox cyclique. Un ensemble fini de GCIs est appelé TBox général. La Figure 3.16 présente un exemple de TBox

TBox
$Femelle \sqsubseteq \top \sqcap \neg Mâle$
$Mâle \sqsubseteq \top \sqcap \neg Femelle$
$Animal \equiv Mâle \sqcup Femelle$
$Humain \sqsubseteq Animal$
$Femme \equiv Humain \sqcap Femelle$
$Homme \equiv Humain \sqcap \neg Femelle$
$Mère \equiv Femme \sqcap \exists relationParentEnfant$
$Père \equiv Homme \sqcap \exists relationParentEnfant$
$MèreSansFille \equiv Mère \sqcap$ $\forall relationParentEnfant.\neg Femme$
$relationParentEnfant \sqsubseteq \top_R$

Figure 3. 16 : Exemple de TBox

- **Définition 3.6. (ABox)** Une ABox est un ensemble fini d'assertions de concept et de rôle. Une assertion de concept est une expression de la forme C(a) où C'est une description de concepts et a est un nom d'individu. Une assertion de rôle est une expression de la forme r(a, b) où r est un nom

de rôle, et a et b sont des noms d'individus. Nous représentons une base de connaissance comme K = (T, A) où T est une TBox et A est une ABox. La Figure 3.17 présente un exemple de ABox.

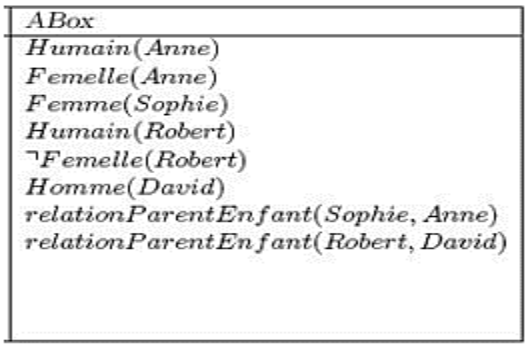


Figure 3. 17 : Exemple d'ABox

- **Définition 3.7 (Signature).** Soit LD une logique de description, C = {C_1, C2,...} un ensemble fini de concepts atomiques, R = {R_1, R2,...} un ensemble fini de rôles atomiques et U = {a_1, a2,...} un ensemble fini d'individus. Pour C, R, U disjoints deux à deux, le triplet Σ = ⟨C, R, U⟩ est une signature de LD.

- **Définition 3.8. (Formule).** Soit LD une logique de description, Une formule de LD pour une signature Σ est l'un des axiomes autorisés par le langage de LD.

- **Définition 3.9 (Ontologie)**. Soit LD une logique de description, une ontologie de LD est une paire ⟨Σ,F⟩ tel que Σ est une signature et F = ⟨T, A⟩ contient des axiomes T de la T-box, et des assertions A de la A-box.

- **Définition 3.10** (**Elément d'une ontologie**). Un élément d'une ontologie o = ⟨Σ, F⟩ est soit une entité atomique de l'ontologie (e.g., un élément de la signature Σ), soit une entité complexe construite en utilisant des constructeurs du langage de l'ontologie. Dans la suite, on utilisera la notation Q_{LD} (o) pour désigner l'ensemble des éléments d'une ontologie.

Le tableau 3.2 donne en exemple les syntaxes de chaque terme atomique, ainsi que différents constructeurs de concept, constructeur de rôle, axiomes de la T-box et axiomes de la A-box. Le nom de la logique minimale qui les autorise est noté dans la colonne de droite. Le tableau 3.1 récapitule la syntaxe et la sémantique de différents constructeurs et connecteurs de logiques de description.

Terme atomique	Syntaxe	Sémantique	
individu	a	$I(a) \in \Delta$	
concept atomique	A	$I(A) \subseteq \Delta$	
rôle atomique	R	$I(R) \subseteq \Delta \times \Delta$	$(\mathcal{AL})$
Constructeur de concepts	**Syntaxe**	**Sémantique**	**nom**
concept universel	$\top$	$I(\top) = \Delta$	$(\mathcal{AL})$
concept vide	$\bot$	$I(\bot) = \varnothing$	$(\mathcal{AL})$
conjonction	$C \sqcap D$	$I(C \sqcap D) = I(C) \cap I(D)$	$(\mathcal{AL})$
disjonction	$C \sqcup D$	$I(C \sqcup D) = I(C) \cup I(D)$	$\mathcal{U}$
négation	$\neg C$	$\Delta \backslash I(C)$	$\mathcal{C}$
restriction existentielle	$\exists R.C$	$\{x \mid \exists y, \langle x, y\rangle \in I(R) \wedge y \in I(C)\}$	$(\mathcal{AL})$
restriction de valeur	$\forall R.C$	$\{x \mid \forall y, \langle x, y\rangle \in I(R) \Rightarrow y \in I(C)\}$	$(\mathcal{AL})$
restriction numérique	$\leq nR$	$\{x \mid card(\{y, \langle x, y\rangle \in I(R)\}) \leq n\}$	$\mathcal{N}$
restriction numérique	$\geq nR$	$\{x \mid card(\{y, \langle x, y\rangle \in I(R)\}) \geq n\}$	$\mathcal{N}$
nominaux	$\{a_1, \ldots, a_n\}$	$\{I(a_1), \ldots, I(a_n)\}$	$\mathcal{O}$
Constructeur de rôles	**Syntaxe**	**Sémantique**	**nom**
conjonction de rôles	$R \sqcap S$	$I(R \sqcap S) = I(R) \cap I(S)$	$.^{\sqcap}$
disjonction de rôles	$R \sqcup S$	$I(R \sqcup S) = I(R) \cup I(S)$	$.^{\sqcup}$
complément de rôle	$\neg R$	$(\Delta \times \Delta) \backslash I(R)$	$.^{\neg}$
clôture transitive	R^+	$clôturetransitivedeI(R)$	$.^{+}$
rôle inverse	R^-	$\{\langle y, x\rangle \mid \langle x, y\rangle \in I(R)\}$	$.^{-}$
composition de rôles	$R \circ S$	$\{\langle x, z\rangle \mid \exists y, \langle x, y\rangle \in I(R) \wedge \langle y, z\rangle \in I(S)\}$	$.^{\circ}$
Axiomes de la T-box	**Syntaxe**	**Contrainte d'interprétation**	**nom**
subsomption	$C \sqsubseteq D$	$I(C) \subseteq I(D)$	$(\mathcal{AL})$
inclusion de rôles	$R \sqsubseteq S$	$I(R) \subseteq I(S)$	$\mathcal{H}$
transitivité de rôles	$\mathrm{Trans}(R)$	$I(R) = I(R^+)$	$\mathcal{S}$
Axiomes de la A-box	**Syntaxe**	**Contrainte d'interprétation**	**nom**
appartenance à un concept	$C(a)$	$I(a) \in I(C)$	$\mathcal{AL}$
appartenance à un rôle	$R(a_1, a_2)$	$\langle I(a_1), I(a_2)\rangle \in I(R)$	$\mathcal{AL}$
identité	$a_1 = a_2$	$I(a_1) = I(a_2)$	

Tableau 3. 2: Sémantique des logiques de description.

3.6.3 La sémantique des logiques de description est définie comme suit :

- **Définition 3.11 (Interprétation).** Soient LD une logique de description et Σ = ⟨C,R,U⟩ une signature de LD. Une interprétation I de Σ est une paire ⟨Δ,I⟩, où le domaine d'interprétation Δ est un ensemble non-vide, et I est un triplet de fonctions : $I_C : C \rightarrow 2^{\Delta}$, $I_R : R \rightarrow 2^{\Delta \times \Delta}$, et $I_U : U \rightarrow \Delta$.

- **Définition 3.12 (Relation de satisfaction).** Soit LD une logique de description. A chaque axiome autorisé par le langage de LD, est associé une contrainte d'interprétation. On dit qu'une interprétation I satisfait une formule f si et seulement si la contrainte d'interprétation associée à l'axiome de f est satisfaite. On note cette relation I |= f.

Le tableau 3. 3 donne en exemple les relations de satisfaction de chaque terme atomique, constructeur de concept, constructeur de rôle, axiomes de la T-box et axiomes de la A-box.

- **Définition 3.13 (Modèle).** Soient LD une logique de description et o = ⟨Σ,F⟩ une ontologie de LD. Une interprétation de Σ est un modèle de o si et seulement si elle satisfait l'ensemble des formules de F. On note cette relation I |= F. On note également M(o) l'ensemble des modèles de o.

Les deux notions essentielles de raisonnement logique sont la consistance d'une ontologie et l'opérateur de conséquence sémantique (ou de manière équivalente, une relation d'inférence).

- **Définition 3.14 (Consistance).** Soit LD une logique de description, une ontologie o de LD est consistante si et seulement s'il existe au moins un modèle de o.

Une autre notion, celle de cohérence, est parfois présente dans la littérature, mais elle est associée à des sens différents. A priori, une ontologie sera dite incohérente si elle ne répond pas à certains critères que l'on peut associer à des règles de bon usage. Ces règles peuvent évidemment varier d'un groupe d'ingénieurs d'ontologie à un autre. Un critère classique consiste à dire qu'une ontologie est incohérente si une de ses classes est nécessairement vide d'individus, sous peine de rendre l'ontologie inconsistante.

- **Définition 3.15 (Conséquence sémantique, clôture).** Soit LD une logique de description, o une ontologie de LD et f une formule de LD. On dit que f est une conséquence sémantique de o lorsque tous les modèles de o satisfont f . Dans ce cas on note o |= f . L'ensemble des conséquences sémantiques de l'ontologie o est la clôture de o et est noté Cn(o).

- **Définition 3.16 (Décidabilité).** Soit LD une logique de description, on dit que LD est décidable si et seulement si pour toute ontologie o de LD et pour toute formule f de LD, il existe un algorithme permettant de déterminer si f est une conséquence sémantique de o ou non.

- **Définition 3.17 (Correction et complétude).** Soit LD une logique de description, o une ontologie de LD et f une formule de LD. Un algorithme A tel que A(o, f) ∈ {V R AI,F AU X} est dit :

  - correct si et seulement si pour toute ontologie o et pour toute formule f , A(o, f) = V R AI ⇒ o |= f ;
  - complet si et seulement si pour toute ontologie o et pour toute formule f , o |= f ⇒ A(o, f) = V R AI ;

3.7 Conclusion

Le Web sémantique (plus techniquement appelé « le Web de données ») permet aux machines de comprendre la sémantique, la signification de l'information sur le Web. Il déploie le réseau des hyperliens entre des pages Web classiques par un réseau de lien entre données structurées, permettant ainsi aux agents automatisés d'accéder d'une manière intelligente, précise et concises aux différentes sources de données contenues sur le Web, mais aussi d'exécuter des tâches (recherche, apprentissage, etc.) plus précises et plus exactes pour les utilisateurs, pour enfin renvoyer les résultats les plus pertinents et judicieux. L'objectif du Web sémantique est, tout simplement, d'apporter la sémantique formelle nécessaire pour que des machines, elles aussi, puissent consulter et interpréter les informations présentées sur le Web.

Les ontologies et les métadonnées sont les composantes primordiales du web sémantique, et sont particulièrement responsables de sa sémantisation.

Ce chapitre a tenté de presenter ainsi un survol sur le web sémantique et les ontologies.

Le prochain chapitre sera consacré au langage OWL dans ses deux versions OWL et OWL2, qui propose un langage pour représenter des ontologies dans le Web sémantique.

CHAPITRE 4

OWL 2

Sommaire

4.1 Introduction

Le langage d'ontologie Web (OWL) est une famille de langage de représentation des connaissances pour la création d'ontologies. Les ontologies sont

une manière formelle de décrire les taxonomies et les réseaux de classification, définissant essentiellement la structure des connaissances pour divers domaines.Les langages OWL sont caractérisés par une sémantique formelle. Ils sont fondés sur la norme XML du World Wide Web Consortium (W3C) pour les objets appelés Resource Description Framework (RDF).

Le W3C a annoncé la nouvelle version d'OWL le 27 octobre 2009. Cette nouvelle version, appelée OWL 2, a rapidement trouvé sa place dans les éditeurs sémantiques tels que Protégé et les raisonneurs sémantiques tels que Pellet, RacerPro, [FaCT++ et HermiT]
La famille OWL contient de nombreuses espèces, sérialisations, syntaxes et spécifications avec des noms similaires. OWL et OWL2 sont utilisés pour se référer respectivement aux spécifications

2004 et 2009. Les noms complets des espèces seront utilisés, y compris la version de la spécification (par exemple, OWL2 EL).

OWL 2 a une structure globale très similaire à OWL 1. OWL 2 ajoute de nouvelles fonctionnalités par rapport à OWL 1. Certaines de ces nouvelles fonctionnalités sont du sucre syntaxique (par exemple, l'union disjointe de classes), tandis que d'autres offrent une nouvelle expressivité, notamment : des clés, des chaînes de propriétés, des types de données plus riches, des plages de données, des restrictions de cardinalité qualifiées, des propriétés asymétriques, réflexives et disjointes, et des capacités d'annotation améliorées.

Dans ce chapitre nous développerons la nouvelle version OWL2, exposerons les éléments de bases de la structure OWL 2 et énoncerons leurs profiles.

Aussi, nous dévoilerons les différentes syntaxes offertes par OWL2 pour exprimer les :

- Axiomes : les énoncés de base qu'une ontologie OWL exprime.
- Entités : éléments utilisés pour faire référence aux objets du monde réel.
- Expressions : combinaisons d'entités pour former des descriptions complexes à partir de descriptions de base.

4.2 OWL1 : Web Ontology Language

Le langage OWL (Schreiber et Dean 2004) est apparu en 2004, en même temps que RDF et RDFS, il a ensuite été étendu en 2009 dans la norme OWL 2 (OWL Working Group 2009). OWL est un langage qui étend RDFS, il introduit dans les documents RDF une sémantique logique tirée des logiques de description (Grau, Patel-Schneider et Motik 2012). OWL apporte un langage plus riche pour la description de ressources en permettant, en plus d'une hiérarchie verticale, la définition de relations entre les classes et entre les propriétés, comme l'équivalence, la différence, l'union, le complément, etc., ainsi que des raisonnements suivant les logiques de description.

Les logiques de description sont une famille de modèles formels de représentation des connaissances. Dans ces modèles on sépare d'une part la terminologie dans une T-Box – l'ontologie – qui décrit les taxonomies et les relations. D'autre part, sont définis les faits, ou les assertions, dans une A-Box, contenant les connaissances factuelles. T-Box et A-Box forment une base de connaissances sur laquelle on peut effectuer des raisonnements grâce à une sémantique adéquate associée. Les bases de connaissances en logiques de description reposent sur l'hypothèse d'un monde ouvert dans lequel l'absence d'une connaissance n'implique pas qu'elle soit fausse (à l'opposé d'un monde clos où tout ce que l'on ignore est considéré comme faux). Cette hypothèse suppose qu'une base de connaissances ne contient pas toutes les connaissances disponibles. En

suivant cette hypothèse, il est possible de définir des informations de manière dispersée, à la manière des bases RDF du Web des données. Les raisonnements en logique de description sont basés sur la classification et l'instanciation des connaissances. La classification des connaissances correspond à l'organisation des classes et propriétés en hiérarchies. L'instanciation des connaissances correspond au typage des individus par les différentes classes du document (notamment les plus spécialisées). Ainsi, en pratique, les raisonnements sur les documents OWL permettent de déduire de nouvelles connaissances sur les hiérarchies de classes et de propriétés, et sur les types des individus du document. Nous donnons ici quelques exemples des éléments définis par OWL :

- Toutes les classes définies en OWL sont instances de OWL : Class, elle-même sous classe de RDFS : Class, pour des raisons de compatibilité avec RDFS. De plus, toutes les classes sont par défaut sous-classes de OWL : Thing.
- Les disjonctions de classes sont définies grâce à la relation OWL : disjoint With, par exemple la classe ex: Homme est disjointe avec la classe ex:Femme par le triplet (ex:Homme, OWL:- disjointWith, ex:Femme).
- Les classes OWL peuvent être définies selon des opérations ensemblistes, telle que les unions, par OWL : union Of, les intersections, par OWL : inter section Of et les compléments, par OWL : complément Of.

OWL est une extension de RDF Schémas qui est basé sur RDF. Il définit un vocabulaire riche pour décrire les ontologies. Ce langage est recommandé par le W3C comme un standard pour manipuler les ontologies.

4.2.1 Définitions

OWL est une extension de RDF Schémas qui est basé sur RDF. Il définit un vocabulaire riche pour décrire les ontologies. Ce langage est recommandé par le W3C comme un standard pour le Web sémantique depuis 2004 et constitue un pilier pour le Web sémantique, selon Tim Berners-Lee (Berners-Lee, Hendler et Lassila 2001). OWL a pour principale caractéristique de décrire les concepts et les relations entre ces concepts, pour permettre un raisonnement logique au sein des systèmes d'information, appelé inférences.

4.2.2 Les sous langages d'OWL1

Le langage OWL peut être défini en trois sous langages (La figure 4.1), suivant le niveau d'expressivité qu'on cherche à exprimer.

– **OWL Lite :** Il s'agit d'un sous-ensemble d'OWL qui permet d'exprimer la classification, et les relations simples entre les classes. Ce sous langage ne permet pas d'exprimer des contraintes complexes sur les classes ou sur les associations. OWL Lite permet uniquement des hiérarchies et contraintes simples, e.g. il supporte uniquement les contraintes de cardinalité 0 ou 1. Les limitations de l'expressivité de OWL Lite ne réduisent pas la complexité des raisonnements par rapport à OWL DL, leur but est de faciliter l'implémentation d'un raisonneur. C'est le moins complexe des 3 sous-langages OWL 1.

– **OWL DL :** Ce sous-ensemble permet un meilleur niveau d'expressivité tout en maintenant la complétude et la décidabilité (tous les calculs doivent être achevés en un temps fini). Ce sous-ensemble repose sur les caractéristiques de la logique de description pour inclure des propriétés utiles aux systèmes de raisonnement. OWL DL : ce sous-langage a été créé pour permettre une expressivité maximale tout en garantissant la décidabilité et un temps de calcul fini pour le raisonnement en logiques de description. «DL» signifie Description Logics.

– **OWL Full :** Ce sous-ensemble offre un maximum d'expressivité, mais sans aucune garantie de calcul. OWL permet donc à une ontologie d'augmenter le sens du vocabulaire prédéfini.

OWL Full permet l'utilisation de tous les éléments d'OWL sans contraintes, mais il n'offre aucune garantie de décidabilité ou de complexité pour les raisonnements.

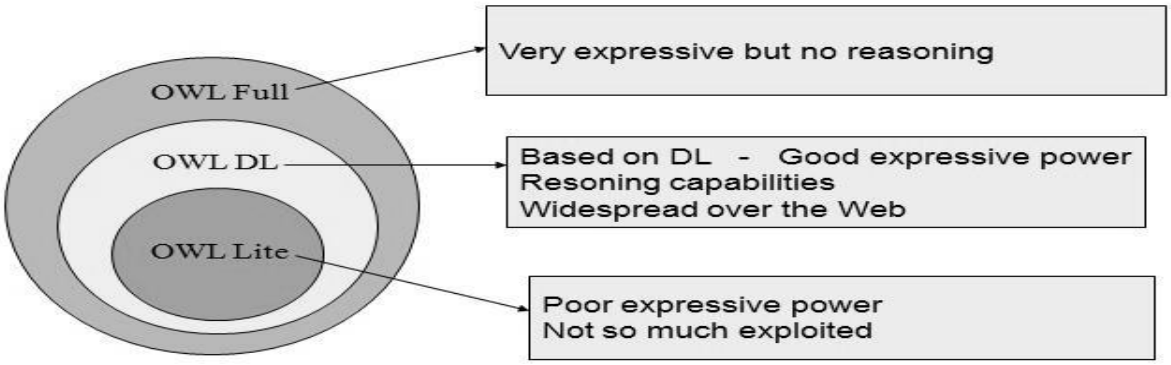


Figure 4. 1 : Les sous langages OWL

Un document OWL Lite est un document OWL DL valide, et un document OWL DL est un document OWL Full valide. Les trois sous-langages d'OWL illustrent le dilemme des ontologies OWL pour lesquelles il faut choisir entre expressivité et décidabilité du raisonnement.

4.3 OWL 2: Web Ontology Language2

4.3.1 Définitions

Dans«OWL 2 Web Ontology Language Document Overview (Second Edition) » (W3C OWL Working Group 2012) et «Langage d'ontologies Web OWL 2. Vue d'ensemble (Deuxième édition) »(Yohan 2014), les auteurs définissent OWL 2 comme est un langage ontologique pour le Web Sémantique possédant une signification formellement définie. OWL 2 est une extension et une révision du langage d'ontologies Web OWL. OWL 2 est conçu pour faciliter le développement d'ontologies et leur partage via le Web, avec pour objectif final la plus grande accessibilité des contenus du Web pour les machines. Les ontologies OWL 2 intègrent des classes, des propriétés, des individus et des valeurs de données, et sont stockées dans des documents Web Sémantique.

La Figure 4.2 fournit une vue d'ensemble du langage OWL 2 en montrant ses principaux composants et la manière dont ils sont associés. L'ellipse au centre représente la notion abstraite d'une ontologie qui peut être vue soit comme une structure abstraite, soit comme un graphe RDF. Dans la partie supérieure sont représentées des syntaxes concrètes qui peuvent être utilisées pour sérialiser et échanger des ontologies.

Le langage OWL 2 reprend les fonctionnalités de sa version précédente en ajoutant quelques éléments supplémentaires. Le principal apport d'OWL 2 est de définir des profils, des sous-langages d'OWL 2 qui limitent et contraignent les expressions utilisables selon l'usage voulu de l'ontologie. Contrairement aux sous-ensembles OWL Lite, DL et Full, les profils d'OWL 2 sont indépendants.

4.3.2 Les éléments de bases de la structure OWL 2 (Yohan 2014) (W3C OWL Working Group, OWL 2 Web Ontology Language Document Overview (Second Edition) 2012)

- **Les ontologies** : les ontologies OWL 2 peuvent être utilisées de concert avec des informations décrites à l'aide de RDF et les ontologies OWL 2 sont elles-mêmes principalement échangées sous la forme de documents RDF.
- **La syntaxe** : une syntaxe concrète est nécessaire pour stocker des ontologies OWL 2 et les échanger entre les outils et les applications. La syntaxe d'échange principale pour OWL 2 est RDF/XML. D'autres syntaxes concrètes peuvent également être utilisées. Parmi ces sérialisations se trouve :

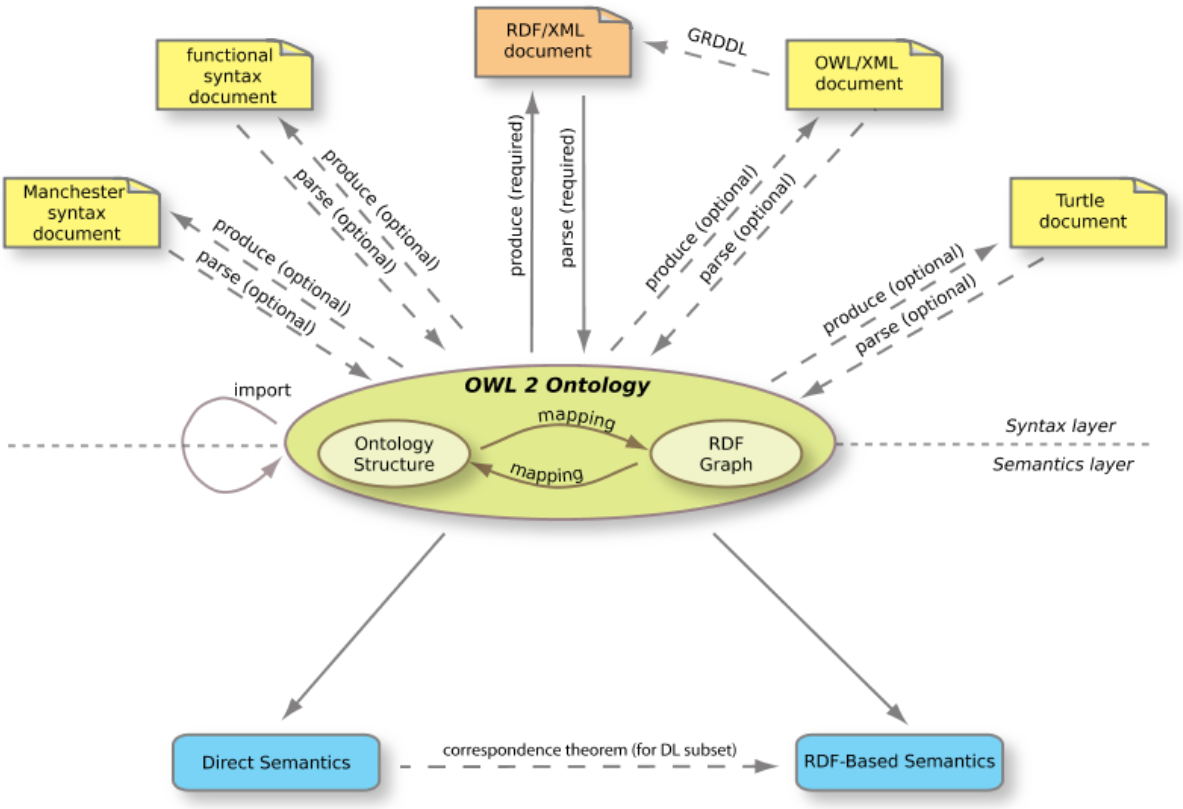


Figure 4. 2 : la structure de OWL 2 (W3C OWL Working Group, OWL 2 Web Ontology Language Document Overview (Second Edition) 2012).

–**Turtle** : permettant de lire/écrire plus facilement des triplets RDF.

–**OWL/XML** : plus simple à traiter en utilisant des outils XML.

–**Syntaxe fonctionnelle** : permettant de mieux voir la structure formelle des ontologies.

–**Syntaxe Manchester** : permettant de lire/écrire plus facilement des ontologies DL.

–**La sémantique** : Le document de spécification structurelle de OWL 2 définit la structure abstraite des ontologies OWL 2 mais il ne définit pas leur signification. Deux alternatives pour associer une signification à des ontologies OWL 2 :

- La sémantique directe OWL 2 : cette sémantique assigne une signification directement dans les structures ontologiques. Elle est compatible avec le modèle sémantique théorique de la logique de description SROIQ et ainsi elle ne peut être appliquée qu'a des ontologies satisfaisant certaines conditions syntaxiques, ces dernières sont appelées ontologies "OWL 2 DL".
- La sémantique OWL 2 basée sur RDF : cette sémantique tend vers les conditions sémantiques définies pour RDF. Elle attribue une signification directement sur les graphes RDF et donc indirectement sur la structure ontologique via la correspondance vers les graphes RDF. La sémantique basée sur RDF peut être appliquée à toute ontologie

OWL 2, sans restrictions, comme toute ontologie OWL 2 peut avoir une correspondance vers RDF.

4.3.3 Les profils OWL 2 : (Antoniou, et al. 2012)

Les profils OWL 2 sont des sous-langages ou sous-ensembles syntaxiques de OWL 2 qui offrent d'importants avantages dans certains scénarios d'applications. Ils sont définis comme une restriction syntaxique de la spécification structurelle de OWL 2.

Le langage OWL 2 reprend les fonctionnalités de sa version précédente en ajoutant quelques éléments supplémentaires. Le principal apport d'OWL 2 est de définir des profils, des sous-langages d'OWL 2 qui limitent et contraignent les expressions utilisables selon l'usage voulu de l'ontologie. Contrairement aux sous-ensembles OWL Lite, DL et Full, les profils d'OWL 2 sont indépendants. La norme OWL 2 définit trois profils différents :

- **OWL EL** : OWL EL est adapté pour la définition d'ontologie contenant un très grand nombre de classes et de propriétés. Ce sous-langage limite l'expressivité des ontologies OWL 2 pour garantir des raisonnements en logiques de description de complexité au pire des cas polynomiale (PTIME-complet) selon la taille de l'ontologie. Il garantit également une interrogation de complexité au pire exponentielle (ou polynomiale sous certaines conditions). Le nom « EL » est tiré du sous-ensemble des logiques de description sur lequel ce profil se base : les logiques EL (Baader, Brandt et Lutz 2005) permettent uniquement la quantification existentielle.

- **OWL RL** : OWL RL est adapté pour permettre les raisonnements, issus de la logique classique, à grande échelle et sans sacrifier trop d'expressivité. Les raisonnements appliqués sur OWL RL peuvent être implémentés par des systèmes à base de règles («RL» vient de Rule Langage). Ils sont au pire des cas Co-NP-complets si ontologie et données sont mélangées, et PTIME-complets s'ils sont considérés séparément. Ce profil est parfois comparé aux différentes propositions de langages de règles pour le Web des données, telles que SWRL (Horrocks, et al. 2004), pour la description de règles.

- **OWL QL** : OWL QL est adapté pour les bases où on souhaite effectuer des raisonnements sur un grand nombre d'individus avec une ontologie de taille réduite. Ce sous-langage a été conçu de façon à ce que l'interrogation conjonctive des bases OWL QL puisse être effectuée par des systèmes performants de base de données relationnelle. OWL QL permet des raisonnements de complexité N-LOGSPACE et des interrogations de complexité NP-complète.

  Il est possible de définir d'autres profils en définissant des sous-langages de OWL 2, par

exemple, OWL Lite et OWL DL peuvent être vus comme d'autres profils OWL 2. Les relations entre les sous-langages et profiles de OWL sont résumés en figure 3.4.

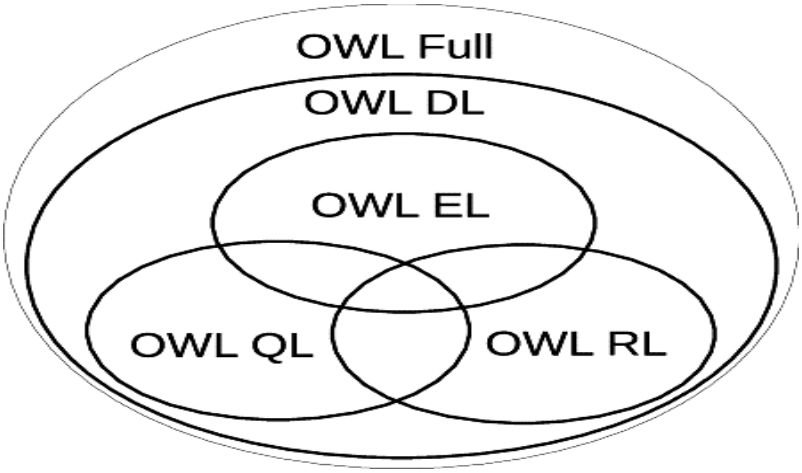


Figure 4. 3 : Diagramme de Venn des sous-langages de OWL

Après que OWL 1 ait proposé différents niveaux d'expressivité en fonction de leurs complexités de raisonnement, OWL 2 propose donc de nouveaux sous-langages dans lesquels l'expressivité est réglée en fonction de l'usage voulu, de la complexité et du niveau de raisonnements recherchés.

Pour résumer la structure OWL 2 est très similaire à celle de OWL. Les nouvelles fonctionnalités OWL 2 concernent principalement des simplifications syntaxiques (la classe union de classes disjointes), et l'augmentation de l'expressivité (les clés, les chaînes de propriétés, des types de données plus riches, plage de données, les restrictions de cardinalités qualifiées, les propriétés disjointes asymétriques et réflexives, et des possibilités d'annotations améliorées). OWL 2 a également défini trois nouveaux profils (OWL 2 EL, OWL 2 QL, et OWL 2 RL) et une nouvelle syntaxe (Syntaxe Manchester).

4.4 Raisonneurs

Les sous langages d'OWL ont été implémentés dans plusieurs raisonneurs prenant en charge différents niveaux d'expressivité. Nous citons ici quelques exemples des raisonneurs récemment mis à jour. ELK Reasoner (Kazakov et Klinov 2014) est un raisonneur sur les ontologies OWL EL. Apache Jena2 est un framework Java répandu pour l'utilisation de données RDF qui intègre un raisonneur OWL prenant en compte un sous-langage d'OWL 2 proche de OWL EL. Pellet (Sirin, et al. 2007) est un raisonneur OWL 2 prenant en charge les ontologies OWL EL et OWL DL. Mastro (Civili, et al. 2013) est un système de gestion de bases RDF basé sur une base de données relationnelles classiques et intégrant un raisonneur prenant en charge OWL Lite et OWL

2 QL. Racer (Haarslev, et al. 2012) est un raisonneur prenant en charge les ontologies OWL DL et proposant son propre langage de requête d'interrogation nommé nRQL.

Pour résumer la structure OWL 2 est très similaire à celle de OWL. Les nouvelles fonctionnalités OWL 2 concernent principalement des simplifications syntaxiques (la classe union de classes disjointes), et l'augmentation de l'expressivité (les clés, les chaînes de propriétés, des types de données plus riches, plage de données, les restrictions de cardinalités qualifiées, les propriétés disjointes asymétriques et réflexives, et des possibilités d'annotations améliorées). OWL 2 a également défini trois nouveaux profils (OWL 2 EL, OWL 2 QL, et OWL 2 RL) et une nouvelle syntaxe (Syntaxe Manchester) donnés directement via des requêtes relationnelles.

4.5 Syntaxe OWL2 (Hitzler, et al. 2012) (Horridge et Patel-Schneider 2009) (Bock, et al. 2012) (Smith, Welty et McGuinness 2004) (Recommendation 2004) (Recommendation, OWL 2 Web Ontology Language XML Serialization (Second Edition) 2012) (Horridge et Patel-Schneider, OWL 2 Web Ontology Language Manchester Syntax (Second Edition) 2012)

OWL 2 est un langage de représentation des connaissances, conçu pour formuler, échanger et raisonner sur des connaissances relatives à un domaine d'intérêt. Certaines notions fondamentales doivent d'abord être expliquées pour comprendre comment les connaissances sont représentées dans OWL 2. Ces notions de base sont les suivantes :

- Axiomes : les énoncés de base qu'une ontologie OWL exprime.
- Entités : éléments utilisés pour faire référence aux objets du monde réel.
- Expressions : combinaisons d'entités pour former des descriptions complexes à partir de descriptions de base.

Si OWL 2 vise à capturer la connaissance, le type de "connaissance" qui peut être représenté par OWL ne reflète évidemment pas tous les aspects de la connaissance humaine. OWL peut être considéré comme un puissant langage de modélisation à usage général pour certaines parties de la connaissance humaine. Les résultats des processus de modélisation sont appelés ontologies - une terminologie qui permet également d'éviter toute confusion puisque le terme "modèle" est souvent utilisé dans un sens assez différent dans la représentation des connaissances.

Pour formuler les connaissances de manière explicite, il est utile de supposer qu'elles sont constituées d'éléments élémentaires, souvent appelés énoncés ou propositions. Des énoncés comme "il pleut" ou "tout homme est mortel" sont des exemples typiques de ces propositions élémentaires. En effet, chaque ontologie OWL 2 est essentiellement une collection de ces "éléments de connaissance" de base. Les déclarations qui sont faites dans une ontologie sont appelées axiomes dans OWL 2, et l'ontologie affirme que ses axiomes sont vrais. En général, les

déclarations OWL peuvent être soit vraies, soit fausses, compte tenu d'un certain état de fait. Cela les distingue des entités et des expressions décrites plus loin. OWL 2 est caractérisée par :

- Une caractéristique importante d'OWL est qu'il capture cet aspect de l'intelligence humaine pour les formes de connaissances qu'il peut représenter. Mais qu'est-ce que cela signifie, en général, qu'un énoncé est une conséquence d'autres énoncés ? Essentiellement, cela signifie que cette déclaration est vraie chaque fois que les autres déclarations le sont. En termes OWL, nous disons qu'un ensemble d'énoncés A entraîne un énoncé a si, dans tout état de choses où tous les énoncés de A sont vrais, a est également vrai. De plus, un ensemble d'énoncés peut être cohérent (c'est-à-dire qu'il existe un état de choses possible dans lequel tous les énoncés de l'ensemble sont conjointement vrais) ou incohérent (il n'existe pas d'état de choses de ce type). La sémantique formelle de OWL spécifie, en substance, pour quels "états de choses" possibles un ensemble particulier d'énoncés OWL est vrai.
- Il existe des outils OWL - raisonneurs - qui peuvent calculer automatiquement les conséquences. La façon dont les axiomes ontologiques interagissent peut-être très subtile et difficile à comprendre pour les gens. C'est à la fois une force et une faiblesse d'OWL 2. C'est une force car les outils OWL 2 peuvent découvrir des informations qu'une personne n'aurait pas repérées. Cela permet aux ingénieurs des connaissances de modéliser plus directement et au système de fournir un retour d'information et une critique utile de la modélisation. C'est une faiblesse car il est comparativement difficile pour les humains de prévoir immédiatement l'effet réel de diverses constructions dans diverses combinaisons. Le support des outils améliore la situation, mais une ingénierie des connaissances réussie nécessite encore souvent une certaine formation et une certaine expérience.
- En examinant de plus près les énoncés dans OWL, nous constatons qu'ils sont rarement "monolithiques" mais qu'ils ont plus souvent une structure interne qui peut être représentée explicitement. Ils font normalement référence à des objets du monde et les décrivent, par exemple en les classant dans des catégories (comme "Marie est une femme") ou en disant quelque chose sur leur relation ("Jean et Marie sont mariés"). Tous les constituants atomiques des déclarations, qu'il s'agisse d'objets (Jean, Marie), de catégories (femme) ou de relations (marié) sont appelés entités. Dans OWL 2, nous désignons les objets comme des individus, les catégories comme des classes et les relations comme des propriétés. Les propriétés dans OWL 2 sont encore subdivisées. Les propriétés d'objet relient les objets entre eux (comme une personne à son conjoint), tandis que les propriétés de type de données attribuent des valeurs de données aux objets (comme un âge à une personne). Les propriétés d'annotation sont utilisées pour encoder des informations sur (des

parties de) l'ontologie elle-même (comme l'auteur et la date de création d'un axiome) au lieu du domaine d'intérêt.

- L'une des principales caractéristiques de OWL est que les noms des entités peuvent être combinés en expressions à l'aide de **constructeurs**. À titre d'exemple, les classes atomiques "femme" et "professeur" peuvent être combinées de manière conjonctive pour décrire la classe des femmes professeurs. Cette dernière serait décrite par une expression de classe OWL, qui pourrait être utilisée dans des déclarations ou dans d'autres expressions. En ce sens, les expressions peuvent être considérées comme de nouvelles entités définies par leur structure. Dans OWL, les constructeurs pour chaque type d'entité varient considérablement. Le langage d'expression des classes est très riche et sophistiqué, alors que le langage d'expression des propriétés l'est beaucoup moins. Ces différences ont des raisons aussi bien historiques que techniques.

4.5.1 Syntaxe des classes

Une classe définit un groupe d'individus possédant des caractéristiques similaires. L'ensemble des individus d'une classe est désigné par le terme extension de classe, chacun de ces individus étant alors une instance de la classe. Chaque individu du monde OWL est un membre de la classe owl: Thing. Chaque classe définie par l'utilisateur est donc implicitement une sous-classe de owl:Thing. Une assertion de classe ClassAssertion(CE a) déclare que l'individu a est une instance de l'expression de classe CE. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Voir le tableau 4.1.

Functional-Style Syntax	**ClassAssertion(:Person :Mary)**
RDF/XML Syntax	<Person rdf:about="Mary"/>
Turtle Syntax	:Mary rdf:type :Person .
Manchester Syntax	Individual: Mary Types: Person
OWL/XML Syntax	<ClassAssertion> <Class IRI="Person"/> <NamedIndividual IRI="Mary"/> </ClassAssertion>

Tableau 4. 1 : Exemple des différentes syntaxes OWL2 pour exprimer les classes.

Cet énoncé parle d'un individu nommé Marie et affirme que cet individu est une personne. Plus techniquement, le fait d'être une personne est exprimé en déclarant que Marie appartient à (ou "est un membre de" ou, encore plus techniquement, "est une instance de") la classe de toutes les personnes. En général, les classes sont utilisées pour regrouper des individus qui ont quelque chose

en commun afin de s'y référer. Par conséquent, les classes représentent essentiellement des ensembles d'individus. En modélisation, les classes sont souvent utilisées pour désigner l'ensemble des objets compris par un concept de la pensée humaine, comme le concept « Person » ou le concept « Woman» . Par conséquent, on peut utiliser le même type d'énoncé pour indiquer que Marie est une femme en exprimant qu'elle est une instance de la classe des femmes. Voir le tableau 4.2.

Functional-Style Syntax	ClassAssertion(:Woman :Mary)
RDF/XML Syntax	<Woman rdf:about="Mary"/>
Turtle Syntax	:Mary rdf:type :Woman
Manchester Syntax	Individual: Mary Types: Woman
OWL/XML Syntax	<ClassAssertion> <Class IRI="Woman"/> <NamedIndividual IRI="Mary"/> </ClassAssertion>

Tableau 4. 2 : Exemple des différentes syntaxes OWL2 pour exprimer les individus et les classes auxquels ils appartiennent.

4.5.1.2 Hiérarchies de classes

Dans la section précédente, nous parlions de deux classes : la classe de toutes les personnes et celle de toutes les femmes. Pour le lecteur humain, il est clair que ces deux classes ont une relation particulière : Personne n’est plus général que Femme, ce qui signifie que chaque fois que nous savons qu'un individu est une femme, cet individu doit être une personne. Toutefois, cette correspondance ne peut être déduite des étiquettes "Person" et "Woman", mais fait partie des connaissances de base de l'homme sur le monde et de l'usage que nous faisons de ces termes. Par conséquent, pour qu'un système puisse tirer les conclusions souhaitées, il doit être informé de cette correspondance. Dans OWL 2, Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.3.

Functional-Style Syntax	SubClassOf(:Woman :Person)
RDF/XML Syntax	<owl:Class rdf:about="Woman"> <rdfs:subClassOf rdf:resource="Person"/> </owl:Class>
Turtle Syntax	:Woman rdfs:subClassOf :Person .
Manchester Syntax	Class: Woman SubClassOf: Person
OWL/XML Syntax	<SubClassOf> <Class IRI="Woman"/> <Class IRI="Person"/> </SubClassOf>

Tableau 4. 3 : Exemple des différentes syntaxes OWL2 pour exprimer la hiérarchie des classes

4.5.1.3 Equivalence des classes

Les classes de notre vocabulaire peuvent effectivement faire référence aux mêmes ensembles, et OWL fournit un mécanisme par lequel elles sont considérées comme sémantiquement équivalentes. Par exemple, nous utilisons indifféremment les termes Person et Humain, ce qui signifie que toute instance de la classe Personne est également une instance de la classe Humain, et vice versa. Deux classes sont considérées comme équivalentes si elles contiennent exactement les mêmes individus. L'exemple suivant indique que la classe Person est équivalent à la classe Humain. Affirmer que la Person et Humain sont équivalents revient exactement au même que d'affirmer que la Person est une sous-classe de Humain et que Humain est une sous-classe de la Person. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.4

Functional-Style Syntax	EquivalentClasses(:Person :Human)
RDF/XML Syntax	<owl:Class rdf:about="Person"> <owl:equivalentClass rdf:resource="Human"/> </owl:Class>
Turtle Syntax	:Person owl:equivalentClass :Human .
Manchester Syntax	Class: Person EquivalentTo: Human
OWL/XML Syntax	<EquivalentClasses> <Class IRI="Person"/> <Class IRI="Human"/> </EquivalentClasses>

Tableau 4. 4: Exemple des différentes syntaxes OWL2 pour exprimer l'équivalence des classes.

4.5.1.4 Disjonction des classes

Un individu peut être une instance de plusieurs classes. Cependant, dans certains cas, l'appartenance à une classe exclut spécifiquement l'appartenance à une autre. Par exemple, si nous considérons les classes Man et Woman, nous savons qu'aucun individu ne peut être une instance des deux classes (pour les besoins de l'exemple, nous ne tenons pas compte des cas limites biologiques). Cette "relation d'incompatibilité" entre les classes est appelée disjonction (de classe). Là encore, l'information selon laquelle deux classes sont disjointes fait partie de nos connaissances de base et doit être explicitement énoncée pour qu'un système de raisonnement puisse l'utiliser. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.5.

Functional-Style Syntax	DisjointClasses(:Woman :Man)
RDF/XML Syntax	<owl:AllDisjointClasses> <owl:members rdf:parseType="Collection"> <owl:Class rdf:about="Woman"/> <owl:Class rdf:about="Man"/> </owl:members> </owl:AllDisjointClasses>
Turtle Syntax	[] rdf:type owl:AllDisjointClasses ; owl:members (:Woman :Man) .
Manchester Syntax	DisjointClasses: Woman, Man
OWL/XML Syntax	<DisjointClasses> <Class IRI="Woman"/> <Class IRI="Man"/> </DisjointClasses>

Tableau 4. 5: Exemple des différentes syntaxes OWL2 pour exprimer disjonction des classes**.**

4.5.2 Les propriétés des objets

Les Propriétés de l'objet sont des relations entre les instances de deux classes. Une assertion positive de propriété d'objet **ObjectPropertyAssertion**(OPE a1 a2) déclare que l'individu a1 est relié par l'expression de propriété d'objet OPE à l'individu a2. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.5.

Functional-Style Syntax	ObjectPropertyAssertion(:hasWife :John :Mary)
RDF/XML Syntax	<rdf:Description rdf:about="John"> <hasWife rdf:resource="Mary"/> </rdf:Description>
Turtle Syntax	:John :hasWife :Mary .
Manchester Syntax	Individual: John Facts: hasWife Mary
OWL/XML Syntax	<ObjectPropertyAssertion> <ObjectProperty IRI="hasWife"/> <NamedIndividual IRI="John"/> <NamedIndividual IRI="Mary"/> </ObjectPropertyAssertion>

Tableau 4. 6 : Exemple des différentes syntaxes OWL2 pour exprimer Les propriétés des objets.

Une assertion négative de propriété d'objet **NegativeObjectPropertyAssertion**(OPE a1 a2) déclare que l'individu a1 n'est pas relié par l'expression de propriété d'objet OPE à l'individu a2.On peut aussi affirmer que deux individus ne sont pas liés par une propriété en utilisant les assertions de propriétés négatives 'Negative property assertions'. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.7.

Functional-Style Syntax	NegativeObjectPropertyAssertion(:hasWife :Bill :Mary)
RDF/XML Syntax	<owl:NegativePropertyAssertion> <owl:sourceIndividual rdf:resource="Bill"/> <owl:assertionProperty rdf:resource="hasWife"/> <owl:targetIndividual rdf:resource="Mary"/> </owl:NegativePropertyAssertion>
Turtle Syntax	[] rdf:type owl:NegativePropertyAssertion ; owl:sourceIndividual :Bill ; owl:assertionProperty :hasWife ; owl:targetIndividual :Mary .
Manchester Syntax	[] rdf:type owl:NegativePropertyAssertion ; owl:sourceIndividual :Bill ; owl:assertionProperty :hasWife ; owl:targetIndividual :Mary .

OWL/XML Syntax	<NegativeObjectPropertyAssertion> <ObjectProperty IRI="hasWife"/> <NamedIndividual IRI="Bill"/> <NamedIndividual IRI="Mary"/> </NegativeObjectPropertyAssertion>

Tableau 4. 7: Exemple des différentes syntaxes OWL2 pour exprimer Les propriétés des objets négatives.

4.5.2.1 Hiérarchies des propriétés des objets

On peut exprimer la hiérarchie des propriétés des objets. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.8.

Functional-Style Syntax	SubObjectPropertyOf(:hasWife :hasSpouse)
RDF/XML Syntax	<owl:ObjectProperty rdf:about="hasWife"> <rdfs:subPropertyOf rdf:resource="hasSpouse"/> </owl:ObjectProperty>
Turtle Syntax	:hasWife rdfs:subPropertyOf :hasSpouse .
Manchester Syntax	ObjectProperty: hasWife SubPropertyOf: hasSpouse
OWL/XML Syntax	<SubObjectPropertyOf> <ObjectProperty IRI="hasWife"/> <ObjectProperty IRI="hasSpouse"/> </SubObjectPropertyOf>

Tableau 4. 8 : Exemple des différentes syntaxes OWL2 pour exprimer hiérarchies des propriétés des objets.

On peut également exprimer l'axiome axiome de propriétés d'objet équivalentes : **EquivalentObjectProperties(** OPE1 ... OPEn) déclare que toutes les expressions de propriétés d'objet OPEi, $1 \leq i \leq n$, sont sémantiquement équivalentes les unes aux autres. Cet axiome permet d'utiliser chaque OPEi comme synonyme de chaque OPEj - c'est-à-dire que dans toute expression de l'ontologie contenant un tel axiome, on peut remplacer OPEi par OPEj sans affecter le sens de l'ontologie.

L'axiome **EquivalentObjectProperties (OPE1 OPE2)** est équivalent aux deux axiomes suivants :

SubObjectPropertyOf(OPE1 OPE2)

SubObjectPropertyOf(OPE2 OPE1)

On peut également exprimer l'axiome axiome de propriétés d'objet disjointes :

Un axiome de propriétés d'objet disjointes **DisjointObjectProperties(OPE_1 ... OPE_n)** stipule que toutes les expressions de propriétés d'objet OPEi, $1 \leq i \leq n$, sont disjointes par paire ; c'est-à-dire qu'aucun individu x ne peut être relié à un individu y à la fois par OPEi et OPEj pour $i \neq j$.

On peut également exprimer l'axiome de propriétés d'objet inverses :

L'axiome de propriétés d'objet inverses InverseObjectProperties(OPE1 OPE2) indique que l'expression de propriété d'objet OPE1 est l'inverse de l'expression de propriété d'objet OPE2. Ainsi, si un individu x est relié par OPE1 à un individu y, alors y est également relié par OPE2 à x, et vice versa. Chacun de ces axiomes peut être considéré comme un raccourci syntaxique pour l'axiome suivant :

InverseObjectProperties(OPE1 ObjectInverseOf(OPE2))

4.5.2.2 Restrictions de domaine et range 'Domain and Range Restrictions'

- **Domaine de propriétés d'un objet**

Un axiome de domaine de propriété d'objet **ObjectPropertyDomain**(OPE CE) déclare que le domaine de l'expression de propriété d'objet OPE est l'expression de classe CE - c'est-à-dire que si un individu x est connecté par OPE avec un autre individu, alors x est une instance de CE. Chacun de ces axiomes peut être vu comme un raccourci syntaxique pour l'axiome suivant : **SubClassOf(ObjectSomeValuesFrom**(OPE owl:Thing)CE)).

- **Range de propriétés d'un objet**

Un axiome d'étendue de propriété d'objet **ObjectPropertyRange**(OPE CE) déclare que l'étendue de l'expression de propriété d'objet OPE est l'expression de classe CE - c'est-à-dire que si un individu est connecté par OPE avec un individu x, alors x est une instance de CE. Chacun de ces axiomes peut être considéré comme un raccourci syntaxique pour l'axiome suivant : **SubClassOf (owl:Thing ObjectAllValuesFrom**(OPE CE))

Il existe plusieurs façons de restreindre la relation, lorsque nous définissons une propriété des objets. Nous pouvons définir un domaine et une image (range). Comme il est formulé dans l'exemple ci-dessous. Consulter le tableau 4.9.

Functional-Style Syntax	ObjectPropertyDomain(:hasWife :Man) ObjectPropertyRange(:hasWife :Woman)
RDF/XML Syntax	<owl:ObjectProperty rdf:about="hasWife"> <rdfs:domain rdf:resource="Man"/> <rdfs:range rdf:resource="Woman"/> </owl:ObjectProperty>
Turtle Syntax	:hasWife rdfs:domain :Man ; rdfs:range :Woman .
Manchester Syntax	ObjectProperty: hasWife Domain: Man Range: Woman
OWL/XML Syntax	<ObjectPropertyDomain> <ObjectProperty IRI="hasWife"/> <Class IRI="Man"/> </ObjectPropertyDomain> <ObjectPropertyRange> <ObjectProperty IRI="hasWife"/> <Class IRI="Woman"/> </ObjectPropertyRange>

Tableau 4. 9: Exemple des différentes syntaxes OWL2 pour exprimer les restrictions de domaine et range des propriétés des objets.

4.5.3 Égalité, inégalité et énumération des Individus

- **Égalité des individus**

Un axiome d'égalité individuelle SameIndividual(a_1 ... a_n) affirme que tous les individus a_i, $1 \leq i \leq n$, sont égaux entre eux. Cet axiome permet d'utiliser chaque ai comme synonyme de chaque a_j - c'est-à-dire que dans toute expression de l'ontologie contenant un tel axiome, ai peut être remplacé par a_j sans affecter le sens de l'ontologie. On peut exprimer légalité des individus, comme il est formulé dans l'exemple ci-dessous. Consulter le tableau 4.10.

Functional-Style Syntax	SameIndividual(:James :Jim)
RDF/XML Syntax	<rdf:Description rdf:about="James"> <owl:sameAs rdf:resource="Jim"/> </rdf:Description>
Turtle Syntax	:James owl:sameAs :Jim.
Manchester Syntax	Individual: James SameAs: Jim

OWL/XML Syntax	<SameIndividual> <NamedIndividual IRI="James"/> <NamedIndividual IRI="Jim"/> </SameIndividual>

Tableau 4. 10: Exemple des différentes syntaxes OWL2 pour exprimer l'égalité des individus.

- **Inégalité des individus**

Un axiome d'inégalité individuelle **DifferentIndividuals**(a_1 ... a_n) stipule que tous les individus ai, $1 \leq i \leq n$, sont différents les uns des autres ; c'est-à-dire qu'aucun individu ai et a_j avec $i \neq j$ ne peut être dérivé pour être égal. Cet axiome peut être utilisé pour axiomatiser l'hypothèse du nom unique - l'hypothèse selon laquelle tous les noms d'individus différents dénotent des individus différents. Et on peut exprimer l'inégalité des individus, comme il est formulé dans l'exemple ci-dessous. Consulter le tableau 4.11.

Functional-Style Syntax	DifferentIndividuals(:John :Bill)
RDF/XML Syntax	<rdf:Description rdf:about="John"> <owl:differentFrom rdf:resource="Bill"/> </rdf:Description>
Turtle Syntax	:John owl:differentFrom :Bill .
Manchester Syntax	Individual: John DifferentFrom: Bill
OWL/XML Syntax	<DifferentIndividuals> <NamedIndividual IRI="John"/> <NamedIndividual IRI="Bill"/> </DifferentIndividuals>

Tableau 4. 11: Exemple des différentes syntaxes OWL2 pour exprimer l'inégalité des individus.

- **Enumération des Individus**

Les individus dans la syntaxe OWL 2 représentent des objets réels du domaine. Il existe deux types d'individus dans la syntaxe d'OWL 2. Les individus nommés (**Named Individuals**) reçoivent un nom explicite qui peut être utilisé dans n'importe quelle ontologie pour faire référence au même objet. Les individus anonymes (**Anonymous Individuals**) n'ont pas de nom global et sont donc locaux à l'ontologie dans laquelle ils sont contenus.

Une façon très simple de décrire une classe est d'énumérer toutes ses instances. OWL offre cette possibilité, cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.12 :

Functional-Style Syntax	DifferentIndividuals(:John :Bill)
RDF/XML Syntax	<rdf:Description rdf:about="John"> <owl:differentFrom rdf:resource="Bill"/> </rdf:Description>
Turtle Syntax	:John owl:differentFrom :Bill .
Manchester Syntax	Individual: John DifferentFrom: Bill
OWL/XML Syntax	<DifferentIndividuals> <NamedIndividual IRI="John"/> <NamedIndividual IRI="Bill"/> </DifferentIndividuals>

Tableau 4. 12: Exemple des différentes syntaxes OWL2 pour exprimer l'énumération des individus.

4.5.4 Les types de données (Datatypes)

Les types de données sont des entités qui font référence à des ensembles de valeurs de données. Ainsi, les types de données sont analogues aux classes, la principale différence étant que les premiers contiennent des valeurs de données telles que des chaînes de caractères et des nombres, plutôt que des individus. Les types de données sont une sorte de plage de données, ce qui leur permet d'être utilisés dans des restrictions. Chaque plage de données est associée à une arité. Pour les types de données, l'arité est toujours un. Le type de données intégré rdfs :Literal désigne tout ensemble de valeurs de données qui contient l'union des espaces de valeurs de tous les types de données. Un IRI utilisé pour identifier un type de données dans une ontologie OWL 2 DL doit

- Être rdfs:Literal, ou
- Identifier un type de données dans la carte des types de données OWL 2 ou
- Ne pas être dans le vocabulaire réservé de OWL 2

Les propriétés de données relient les individus aux littéraux. Dans certains systèmes de représentation des connaissances, les propriétés fonctionnelles des données sont appelées attributs. Voici un exemple d'utilisation d'une propriété de type de données. Consulter le tableau 4.13.

Functional-Style Syntax	DataPropertyAssertion(:hasAge :John "51"^^xsd:integer)
RDF/XML Syntax	<Person rdf:about="John"> <hasAge rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">51</hasAge> </Person>
Turtle Syntax	:John :hasAge 51 .
Manchester Syntax	Individual: John Facts: hasAge "51"^^xsd:integer
OWL/XML Syntax	<DataPropertyAssertion> <DataProperty IRI="hasAge"/> <NamedIndividual IRI="John"/> <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#integer">51</Literal> </DataPropertyAssertion>

Tableau 4. 13: Exemple des différentes syntaxes OWL2 pour exprimer la propriété de type de données.

On peut également exprimer la négation des propriétés de type de données, Cela se fait de la manière suivante. Consulter le tableau 4.14.

Functional-Style Syntax	NegativeDataPropertyAssertion(:hasAge :Jack "53"^^xsd:integer)
RDF/XML Syntax	<owl:NegativePropertyAssertion> <owl:sourceIndividual rdf:resource="Jack"/> <owl:assertionProperty rdf:resource="hasAge"/> <owl:targetValue rdf:datatype="http://www.w3.org/2001/XMLSchema#integer"> 53 </owl:targetValue> </owl:NegativePropertyAssertion>
Turtle Syntax	[] rdf:type owl:NegativePropertyAssertion ; owl:sourceIndividual :Jack ; owl:assertionProperty :hasAge ; owl:targetValue 53 .
Manchester Syntax	Individual: Jack Facts: not hasAge "53"^^xsd:integer
OWL/XML Syntax	<NegativeDataPropertyAssertion> <DataProperty IRI="hasAge"/> <NamedIndividual IRI="Jack"/> <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#integer">53</Literal> </NegativeDataPropertyAssertion>

Tableau 4. 14 : Exemple des différentes syntaxes OWL2 pour exprimer la négation d'une propriété de type de données.

4.5.4.1 Le domaine et l'étendue propriété de type de données

Le domaine et l'étendue (range) peuvent également être indiqués pour les propriétés de type de données, comme c'est le cas pour les propriétés d'objet. Dans ce cas, cependant, l'intervalle sera un type de données au lieu d'une classe. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.15

Functional-Style Syntax	DataPropertyDomain(:hasAge :Person) DataPropertyRange(:hasAge xsd:nonNegativeInteger)
RDF/XML Syntax	<owl:DatatypeProperty rdf:about="hasAge"> <rdfs:domain rdf:resource="Person"/> <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#nonNegativeInteger"/> </owl:DatatypeProperty>
Turtle Syntax	:hasAge rdfs:domain :Person ; rdfs:range xsd:nonNegativeInteger .
Manchester Syntax	DataProperty: hasAge Domain: Person Range: xsd:nonNegativeInteger
OWL/XML Syntax	<DataPropertyDomain> <DataProperty IRI="hasAge"/> <Class IRI="Person"/> </DataPropertyDomain> <DataPropertyRange> <DataProperty IRI="hasAge"/> <Datatype IRI="http://www.w3.org/2001/XMLSchema#nonNegativeInteger"/> </DataPropertyRange>

Tableau 4. 15: Exemple des différentes syntaxes OWL2 pour exprimer le domaine et le range des propriétés de type de données.

4.5.5 Relations avancées entre les classes

Au moyen des éléments de langage décrits jusqu'à présent, des ontologies simples peuvent être modélisées. Afin d'exprimer des connaissances plus complexes, OWL fournit des constructeurs de classes logiques. En particulier, OWL fournit des éléments de langage pour les termes logiques et, ou, et non. Les termes OWL correspondants sont empruntés à la théorie des ensembles : intersection (de classe), union et complément. Ces constructeurs combinent des classes atomiques - c'est-à-dire des classes avec des noms - en classes complexes.

4.5.5.1 Expression de classe d'intersection

Une expression de classe d'intersection ObjectIntersectionOf(CE_1 ... CE_n) contient tous les individus qui sont des instances de toutes les expressions de classe CE_i pour $1 \leq i \leq n$.

L'intersection de deux classes se compose exactement des individus qui sont des instances des deux classes. L'exemple suivant indique que la classe Mère est constituée exactement des objets qui sont des instances à la fois de Woman et de Parent : Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.16

Functional-Style Syntax	EquivalentClasses(:Mother ObjectIntersectionOf(:Woman :Parent))
RDF/XML Syntax	<owl:Class rdf:about="Mother"> <owl:equivalentClass> <owl:Class> <owl:intersectionOf rdf:parseType="Collection"> <owl:Class rdf:about="Woman"/> <owl:Class rdf:about="Parent"/> </owl:intersectionOf> </owl:Class> </owl:equivalentClass> </owl:Class>
Turtle Syntax	:Mother owl:equivalentClass [rdf:type owl:Class ; owl:intersectionOf (:Woman :Parent)] .
Manchester Syntax	Class: Mother EquivalentTo: Woman and Parent
OWL/XML Syntax	<EquivalentClasses> <Class IRI="Mother"/> <ObjectIntersectionOf> <Class IRI="Woman"/> <Class IRI="Parent"/> </ObjectIntersectionOf> </EquivalentClasses>

Tableau 4. 16: Exemple des différentes syntaxes OWL2 pour exprimer l'intersection des classes

4.5.5.2 Expression de classe d'union

Une expression de classe d'union ObjectUnionOf(CE1 ... CEn) contient tous les individus qui sont des instances d'au moins une expression de classe CE_i pour $1 \leq i \leq n$.

L'union de deux classes contient tout individu qui est contenu dans au moins une de ces classes. On peut donc caractériser la classe de tous les parents comme l'union des classes Mother et Father : cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.17.

Functional-Style Syntax	EquivalentClasses(:Parent ObjectUnionOf(:Mother :Father))
RDF/XML Syntax	<owl:Class rdf:about="Parent"> <owl:equivalentClass> <owl:Class> <owl:unionOf rdf:parseType="Collection"> <owl:Class rdf:about="Mother"/> <owl:Class rdf:about="Father"/> </owl:unionOf> </owl:Class> </owl:equivalentClass> </owl:Class>
Turtle Syntax	:Parent owl:equivalentClass [rdf:type owl:Class ; owl:unionOf (:Mother :Father)] .
Manchester Syntax	Class: Parent EquivalentTo: Mother or Father
OWL/XML Syntax	<EquivalentClasses> <Class IRI="Parent"/> <ObjectUnionOf> <Class IRI="Mother"/> <Class IRI="Father"/> </ObjectUnionOf> </EquivalentClasses>

Tableau 4. 17: Exemple des différentes syntaxes OWL2 pour exprimer l'union des classes.

4.5.5.3 Complément d'expressions de classe

Le complément d'une classe correspond à la négation logique : il se compose exactement des objets qui ne sont pas membres de la classe elle-même. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.18.

Functional-Style Syntax	EquivalentClasses(:ChildlessPerson ObjectIntersectionOf(:Person ObjectComplementOf(:Parent)))
RDF/XML Syntax	<owl:Class rdf:about="ChildlessPerson"> <owl:equivalentClass> <owl:Class> <owl:intersectionOf rdf:parseType="Collection"> <owl:Class rdf:about="Person"/> <owl:Class> <owl:complementOf rdf:resource="Parent"/> </owl:Class> </owl:intersectionOf> </owl:Class> </owl:equivalentClass> </owl:Class>
Turtle Syntax	:ChildlessPerson owl:equivalentClass [rdf:type owl:Class ; owl:intersectionOf (:Person [rdf:type owl:Class ; owl:complementOf :Parent])] .
Manchester Syntax	Class: ChildlessPerson EquivalentTo: Person and not Parent
OWL/XML Syntax	<EquivalentClasses> <Class IRI="ChildlessPerson"/> <ObjectIntersectionOf> <Class IRI="Person"/> <ObjectComplementOf> <Class IRI="Parent"/> </ObjectComplementOf> </ObjectIntersectionOf> </EquivalentClasses>

Tableau 4. 18 : Exemple des différentes syntaxes OWL2 pour exprimer le complément des classes.

Une expression de classe complémentaire **ObjectComplementOf**(CE) contient tous les individus qui ne sont pas des instances de l'expression de classe CE.

En général, les classes complexes peuvent être utilisées partout où des classes nommées peuvent apparaître, donc également dans les assertions de classe. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.19.

Functional-Style Syntax	ClassAssertion(ObjectIntersectionOf(:Person ObjectComplementOf(:Parent)) :Jack)
RDF/XML Syntax	<rdf:Description rdf:about="Jack"> <rdf:type> <owl:Class> <owl:intersectionOf rdf:parseType="Collection"> <owl:Class rdf:about="Person"/> <owl:Class> <owl:complementOf rdf:resource="Parent"/> </owl:Class> </owl:intersectionOf> </owl:Class> </rdf:type> </rdf:Description>
Turtle Syntax	:Jack rdf:type [rdf:type owl:Class ; owl:intersectionOf (:Person [rdf:type owl:Class ; owl:complementOf :Parent])] .
Manchester Syntax	Individual: Jack Types: Person and not Parent

OWL/XML Syntax	<ClassAssertion> <ObjectIntersectionOf> <Class IRI="Person"/> <ObjectComplementOf> <Class IRI="Parent"/> </ObjectComplementOf> </ObjectIntersectionOf> <NamedIndividual IRI="Jack"/> </ClassAssertion>

Tableau 4. 19 : Exemple des différentes syntaxes OWL2 pour exprimer les classes complexes.

4.5.6 Restrictions des propriétés des objets

Les expressions de classe dans OWL 2 peuvent être formées en plaçant des restrictions sur les expressions de propriété d'objet, comme le montre la figure 4.5

L'expression de classe **ObjectSomeValuesFrom** permet une quantification existentielle sur une expression de propriété d'objet, et contient les individus qui sont connectés par une expression de propriété d'objet à au moins une instance d'une expression de classe donnée. L'expression de classe **ObjectAllValuesFrom** permet une quantification universelle sur une expression de propriété d'objet, et contient les individus qui sont connectés par une expression de propriété d'objet uniquement à des instances d'une expression de classe donnée. L'expression de classe **ObjectHasValue** contient les individus qui sont reliés par une expression de propriété d'objet à un individu particulier. Enfin, l'expression de classe **ObjectHasSelf** contient les individus qui sont reliés par une expression de propriété d'objet à eux-mêmes.

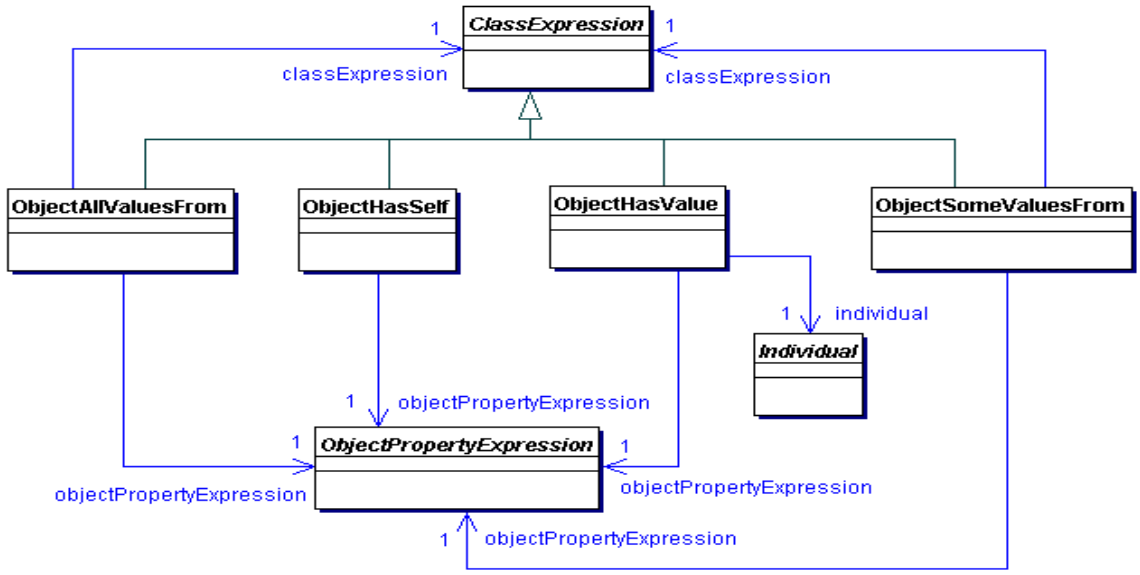


Figure 4. 4 : Expressions des restrictions de propriétés d'objets dans OWL 2

Les restrictions de propriété fournissent un autre type de constructeurs basés sur la logique pour les classes complexes. Comme leur nom l'indique, les restrictions de propriété utilisent des constructeurs impliquant des propriétés.

4.5.6.1 Quantification existentielle ObjectSomeValuesFrom

Une expression de classe existentielle **ObjectSomeValuesFrom(** OPE CE) se compose d'une expression de propriété d'objet OPE et d'une expression de classe CE, et elle contient tous les individus qui sont reliés par OPE à un individu qui est une instance de CE. Une telle expression de classe peut être considérée comme un raccourci syntaxique pour l'expression de classe ObjectMinCardinality(1 OPE CE).

ObjectSomeValuesFrom := 'ObjectSomeValuesFrom'
'(' **ObjectPropertyExpression ClassExpression** ')'

Une restriction de propriété appelée quantification existentielle définit une classe comme l'ensemble de tous les individus qui sont reliés par une propriété particulière à un autre individu qui est une instance d'une certaine classe. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.20.

Functional-Style Syntax	EquivalentClasses(:Parent ObjectSomeValuesFrom(:hasChild :Person))
RDF/XML Syntax	<owl:Class rdf:about="Parent"> <owl:equivalentClass> <owl:Restriction> <owl:onProperty rdf:resource="hasChild"/> <owl:someValuesFrom rdf:resource="Person"/> </owl:Restriction> </owl:equivalentClass> </owl:Class>
Turtle Syntax	<owl:Class rdf:about="Parent"> <owl:equivalentClass> <owl:Restriction> <owl:onProperty rdf:resource="hasChild"/> <owl:someValuesFrom rdf:resource="Person"/> </owl:Restriction> </owl:equivalentClass> </owl:Class>
Manchester Syntax	Class: Parent

	EquivalentTo: hasChild some Person
OWL/XML Syntax	<EquivalentClasses> <Class IRI="Parent"/> <ObjectSomeValuesFrom> <ObjectProperty IRI="hasChild"/> <Class IRI="Person"/> </ObjectSomeValuesFrom> </EquivalentClasses>

Tableau 4. 20 : Exemple des différentes syntaxes OWL2 pour exprimer ObjectSomeValuesFrom.

4.5.6.2. Quantification universelle ObjectAllValuesFrom

Une expression de classe universelle **ObjectAllValuesFrom**(OPE CE) se compose d'une expression de propriété d'objet OPE et d'une expression de classe CE, et elle contient tous les individus qui sont connectés par OPE uniquement à des individus qui sont des instances de CE. Une telle expression de classe peut être considérée comme un raccourci syntaxique de l'expression de classe ObjectMaxCardinality(0 OPE ObjectComplementOf(CE)).

ObjectAllValuesFrom := 'ObjectAllValuesFrom'
'(' **ObjectPropertyExpression ClassExpression** ')'

Le quantificateur universel est utilisé pour décrire une classe d'individus pour laquelle tous les individus apparentés doivent être des instances d'une classe donnée. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.21.

Functional-Style Syntax	EquivalentClasses(:HappyPerson ObjectAllValuesFrom(:hasChild :HappyPerson))
RDF/XML Syntax	<owl:Class> <owl:Class rdf:about="HappyPerson"/> <owl:equivalentClass> <owl:Restriction> <owl:onProperty rdf:resource="hasChild"/> <owl:allValuesFrom rdf:resource="HappyPerson"/> </owl:Restriction> </owl:equivalentClass> </owl:Class>
Turtle Syntax	:HappyPerson rdf:type owl:Class ; owl:equivalentClass [rdf:type owl:Restriction ;

	owl:onProperty :hasChild ; owl:allValuesFrom :Happy] .
Manchester Syntax	Class: HappyPerson EquivalentTo: hasChild only HappyPerson
OWL/XML Syntax	<EquivalentClasses> <Class IRI="HappyPerson"/> <ObjectAllValuesFrom> <ObjectProperty IRI="hasChild"/> <Class IRI="Happy"/> </ObjectAllValuesFrom> </EquivalentClasses>

Tableau 4. 21 : Exemple des différentes syntaxes OWL2 pour exprimer ObjectAllValuesFrom.

4.5.6.3 Restriction de valeur individuelle ObjectHasValue

Une expression de classe has-value **ObjectHasValue**(OPE a) se compose d'une expression de propriété d'objet OPE et d'un individu a, et contient tous les individus qui sont reliés par OPE à a. Chacune de ces expressions de classe peut être considérée comme un raccourci syntaxique de l'expression de classe ObjectSomeValuesFrom(OPE ObjectOneOf(a)).

ObjectHasValue := 'ObjectHasValue' '(' ObjectPropertyExpression Individual ')'

Les restrictions de propriété peuvent également être utilisées pour décrire des classes d'individus qui sont liées à un individu particulier. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.22.

Functional-Style Syntax	EquivalentClasses(:JohnsChildren ObjectHasValue(:hasParent :John))
RDF/XML Syntax	<owl:Class rdf:about="JohnsChildren"> <owl:equivalentClass> <owl:Restriction> <owl:onProperty rdf:resource="hasParent"/> <owl:hasValue rdf:resource="John"/> </owl:Restriction> </owl:equivalentClass> </owl:Class>
Turtle Syntax	:JohnsChildren owl:equivalentClass [rdf:type owl:Restriction ; owl:onProperty :hasParent ;

	owl:hasValue :John] .
Manchester Syntax	Class: JohnsChildren EquivalentTo: hasParent value John
OWL/XML Syntax	

Tableau 4. 22 : Exemple des différentes syntaxes OWL2 pour exprimer ObjectHasValue.

4.5.6.4 Auto-restriction ObjectHasSelf

Une auto-restriction **ObjectHasSelf**(OPE) consiste en une expression de propriété d'objet OPE, et elle contient tous les individus qui sont reliés par OPE à eux-mêmes.

ObjectHasSelf := 'ObjectHasSelf' '(' **ObjectPropertyExpression** ')'

Dans un cas particulier où les individus sont liés par des propriétés, un individu peut être lié à lui-même. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.23.

Functional-Style Syntax	EquivalentClasses(:NarcisticPerson ObjectHasSelf(:loves))
RDF/XML Syntax	<owl:Class rdf:about="NarcisticPerson"> <owl:equivalentClass> <owl:Restriction> <owl:onProperty rdf:resource="loves"/> <owl:hasSelf rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean"> true </owl:hasSelf> </owl:Restriction> </owl:equivalentClass> </owl:Class>
Turtle Syntax	:NarcisticPerson owl:equivalentClass [rdf:type owl:Restriction ; owl:onProperty :loves ; owl:hasSelf "true"^^xsd:boolean .] .
Manchester Syntax	Class: NarcisticPerson EquivalentTo: loves Self
OWL/XML Syntax	<EquivalentClasses> <Class IRI="NarcisticPerson"/>

	<ObjectHasSelf> <ObjectProperty IRI="loves"/> </ObjectHasSelf> </EquivalentClasses> 5.3 Property Cardinality

Tableau 4. 23: Exemple des différentes syntaxes OWL2 pour exprimer ObjectHasSelf**.**

4.5.7 Restrictions de cardinalité des propriétés d'objet

Les expressions de classe dans OWL 2 peuvent être formées en plaçant des restrictions sur la cardinalité des expressions de propriété d'objet, comme le montre la figure 4.5. Toutes les restrictions de cardinalité peuvent être qualifiées ou non : dans le premier cas, la restriction de cardinalité ne s'applique qu'aux individus qui sont connectés par l'expression de propriété d'objet et qui sont des instances de l'expression de classe qualifiante, dans le second cas, la restriction s'applique à tous les individus qui sont connectés par l'expression de propriété d'objet (ceci est équivalent au cas qualifié avec l'expression de classe qualifiante égale à owl:Thing). Les expressions de classe **ObjectMinCardinality**, **ObjectMaxCardinality** et **ObjectExactCardinality** contiennent les individus qui sont reliés par une expression de propriété d'objet à au moins, au plus et exactement un nombre donné d'instances d'une expression de classe spécifiée, respectivement.

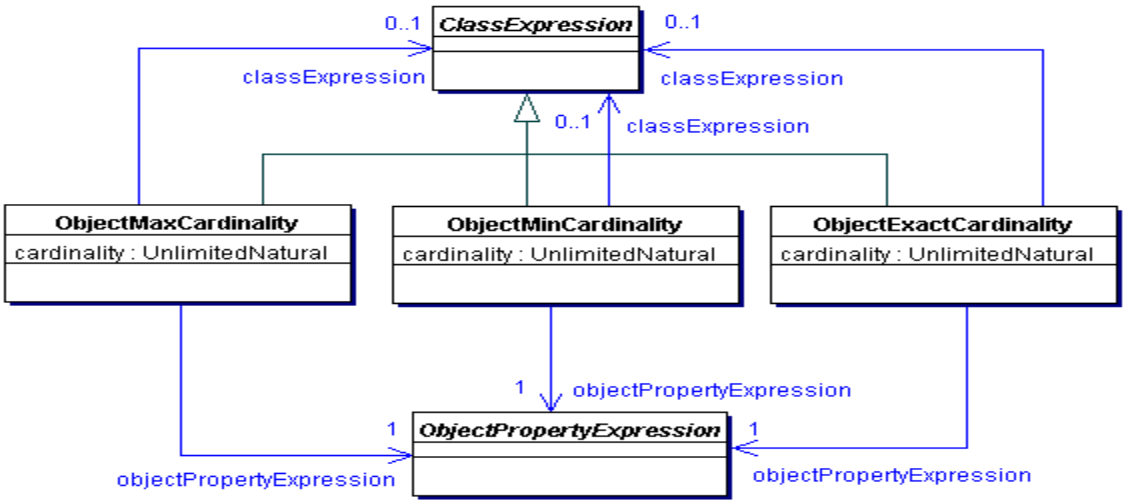


Figure 4. 5 : Expressions des restrictions de la cardinalité de propriétés d'objets dans OWL 2

4.5.7.1 Cardinalité maximale (ObjectMaxCardinality)

Une expression de cardinalité maximale **ObjectMaxCardinality** (n OPE CE) est constituée d'un entier non-négatif, d'une expression de propriété d'objet OPE, et d'une expression de classe CE, et contient tous les individus qui sont connectés par OPE à au plus n individus différents qui

sont des instances de CE. Si CE est manquante, elle est considérée comme étant owl:Thing. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.24.

Functional-Style Syntax	ClassAssertion(ObjectMaxCardinality(4 :hasChild :Parent) :John)
RDF/XML Syntax	<rdf:Description rdf:about="John"> <rdf:type> <owl:Restriction> <owl:maxQualifiedCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#nonNegativeInteger"> 4 </owl:maxQualifiedCardinality> <owl:onProperty rdf:resource="hasChild"/> <owl:onClass rdf:resource="Parent"/> </owl:Restriction> </rdf:type> </rdf:Description>
Turtle Syntax	:John rdf:type [rdf:type owl:Restriction ; owl:maxQualifiedCardinality "4"^^xsd:nonNegativeInteger ; owl:onProperty :hasChild ; owl:onClass :Parent] .
Manchester Syntax	Individual: John Types: hasChild max 4 Parent
OWL/XML Syntax	<ClassAssertion> <ObjectMaxCardinality cardinality="4"> <ObjectProperty IRI="hasChild"/> <Class IRI="Parent"/> </ObjectMaxCardinality> <NamedIndividual IRI="John"/> </ClassAssertion>

Tableau 4. 24 : Exemple des différentes syntaxes OWL2 pour exprimer ObjectMaxCardinality.

4.5.7.2 Cardinalité minimale ObjectMinCardinality

Une expression de cardinalité minimale **ObjectMinCardinality**(n OPE CE) est constituée d'un nombre entier non négatif n, d'une expression de propriété d'objet OPE et d'une expression de classe CE. Elle contient tous les individus qui sont reliés par OPE à au moins n individus différents qui sont des instances de CE. Si CE est manquante, elle est considérée comme étant owl:Thing.

ObjectMinCardinality := 'ObjectMinCardinality' '(' nonNegativeInteger
ObjectPropertyExpression [ClassExpression] ')'

Cela peut être exprimé en OWL 2 sous les différentes syntaxes .Consulter le tableau 4.25

Functional-Style Syntax	ClassAssertion(ObjectMinCardinality(2 :hasChild :Parent) :John)
RDF/XML Syntax	<rdf:Description rdf:about="John"> <rdf:type> <owl:Restriction> <owl:minQualifiedCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#nonNegativeInteger"> 2 </owl:minQualifiedCardinality> <owl:onProperty rdf:resource="hasChild"/> <owl:onClass rdf:resource="Parent"/> </owl:Restriction> </rdf:type> </rdf:Description>
Turtle Syntax	:John rdf:type [rdf:type owl:Restriction ; owl:minQualifiedCardinality "2"^^xsd:nonNegativeInteger ; owl:onProperty :hasChild ; owl:onClass :Parent] .
Manchester Syntax	Individual: John Types: hasChild min 2 Parent
OWL/XML Syntax	<ClassAssertion> <ObjectMinCardinality cardinality="2"> <ObjectProperty IRI="hasChild"/> <Class IRI="Parent"/> </ObjectMinCardinality> <NamedIndividual IRI="John"/> </ClassAssertion>

Tableau 4. 25: Exemple des différentes syntaxes OWL2 pour exprimer ObjectMinCardinality.

4.5.7.3 Cardinalité exacte ObjectExactCardinality

Une expression de cardinalité exacte **ObjectExactCardinality(** n OPE CE) est constituée d'un nombre entier non négatif n, d'une expression de propriété d'objet OPE et d'une expression de

classe CE. Elle contient tous les individus qui sont reliés par OPE à exactement n individus différents qui sont des instances de CE. Si CE est manquante, elle est considérée comme étant owl:Thing. Une telle expression est en fait équivalente à l'expression

ObjectIntersectionOf(ObjectMinCardinality(n OPE CE) ObjectMaxCardinality(n OPE CE)).

Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.26.

Functional-Style Syntax	ClassAssertion(ObjectExactCardinality(3 :hasChild :Parent) :John)
RDF/XML Syntax	<rdf:Description rdf:about="John"> <rdf:type> <owl:Restriction> <owl:qualifiedCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#nonNegativeInteger"> 3 </owl:qualifiedCardinality> <owl:onProperty rdf:resource="hasChild"/> <owl:onClass rdf:resource="Parent"/> </owl:Restriction> </rdf:type> </rdf:Description>
Turtle Syntax	:John rdf:type [rdf:type owl:Restriction ; owl:qualifiedCardinality "3"^^xsd:nonNegativeInteger ; owl:onProperty :hasChild ; owl:onClass :Parent] .
Manchester Syntax	:John rdf:type [rdf:type owl:Restriction ; owl:qualifiedCardinality "3"^^xsd:nonNegativeInteger ; owl:onProperty :hasChild ; owl:onClass :Parent] .
Functional-Style Syntax	ClassAssertion(ObjectExactCardinality(3 :hasChild :Parent) :John)

Tableau 4. 26: Exemple des différentes syntaxes OWL2 pour exprimer ObjectExactCardinality.

Dans une restriction de cardinalité, la fourniture de la classe est facultative, si nous voulons simplement parler du nombre. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.27.

Functional-Style Syntax	ClassAssertion(ObjectExactCardinality(5 :hasChild) :John)
RDF/XML Syntax	ClassAssertion(ObjectExactCardinality(5 :hasChild) :John)
Turtle Syntax	:John rdf:type [rdf:type owl:Restriction ; owl:cardinality "5"^^xsd:nonNegativeInteger ; owl:onProperty :hasChild] .
Manchester Syntax	Individual: John Types: hasChild exactly 5
OWL/XML Syntax	<ClassAssertion> <ObjectExactCardinality cardinality="5"> <ObjectProperty IRI="hasChild"/> </ObjectExactCardinality> <NamedIndividual IRI="John"/> </ClassAssertion>

Tableau 4. 27: Exemple des différentes syntaxes OWL2 pour exprimer les restrictions des cardinalités sans préciser les classes.

4.5.8 Caractéristiques des propriétés

4.5.8.1 La propriété inverse :

Si une propriété P1 est marquée comme étant l'inverse de P2, alors pour tout x, y :

P1(x,y) if P2(y,x)

Un axiome de propriétés d'objet inverses **InverseObjectProperties(** OPE_1 OPE_2) exprime que l'expression de propriété d'objet OPE1 est l'inverse de l'expression de propriété d'objet OPE2. Ainsi, si un individu x est relié par OPE1 à un individu y, alors y est également relié par OPE2 à x, et vice versa.

Chacun de ces axiomes peut être considéré comme un raccourci syntaxique pour l'axiome suivant :

EquivalentObjectProperties(OPE_1 ObjectInverseOf(OPE_2))

Remarquez que la syntaxe de la propriété **owl:inverseOf** impose un nom de propriété comme argument. La notation A if B signifie que (A implique B) et (B implique A). Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.28.

Functional-Style Syntax	InverseObjectProperties(:hasParent :hasChild)
RDF/XML Syntax	<owl:ObjectProperty rdf:about="hasParent"> <owl:inverseOf rdf:resource="hasChild"/> </owl:ObjectProperty>
Turtle Syntax	:hasParent owl:inverseOf :hasChild .
Manchester Syntax	ObjectProperty: hasParent InverseOf: hasChild
OWL/XML Syntax	<InverseObjectProperties> <ObjectProperty IRI="hasParent"/> <ObjectProperty IRI="hasChild"/> </InverseObjectProperties>

Tableau 4. 28: Exemple des différentes syntaxes OWL2 pour exprimer la propriété inverse**.**

4.5.8.2 La propriété symétrique :

Si une propriété P est symétrique, alors pour tout x, y :

P(x,y) if P(y,x).

Un axiome de symétrie de propriété d'objet **SymmetricObjectProperty**(OPE) stipule que l'expression de propriété d'objet OPE est symétrique - c'est-à-dire que si un individu x est relié par OPE à un individu y, alors y est également relié par OPE à x. Chacun de ces axiomes peut être considéré comme un raccourci syntaxique pour l'axiome suivant :

SubObjectPropertyOf(OPE ObjectInverseOf(OPE))

Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.29.

Functional-Style Syntax	SymmetricObjectProperty(:hasSpouse)
RDF/XML Syntax	<owl:SymmetricProperty rdf:about="hasSpouse"/>
Turtle Syntax	:hasSpouse rdf:type owl:SymmetricProperty .
Manchester Syntax	ObjectProperty: hasSpouse Characteristics: Symmetric
OWL/XML Syntax	<SymmetricObjectProperty> <ObjectProperty IRI="hasSpouse"/> </SymmetricObjectProperty>

Tableau 4. 29: Exemple des différentes syntaxes OWL2 pour exprimer la propriété symétrique.

4.5.8.3 La propriété asymétrique

L'axiome d'asymétrie de la propriété d'un objet **AsymmetricObjectProperty**(OPE) stipule que l'expression de la propriété d'un objet OPE est asymétrique, c'est-à-dire que si un individu x est relié par OPE à un individu y, alors y ne peut pas être relié par OPE à x.

D'autre part, une propriété peut également être asymétrique, c'est-à-dire que si elle relie A à B, elle ne relie jamais B à A. Cela peut être exprimé en OWL 2 sous les différentes syntaxe. Consulter le tableau 4.30.

Functional-Style Syntax	AsymmetricObjectProperty(:hasChild)
RDF/XML Syntax	<owl:AsymmetricProperty rdf:about="hasChild"/>
Turtle Syntax	:hasChild rdf:type owl:AsymmetricProperty .
Manchester Syntax	ObjectProperty: hasChild Characteristics: Asymmetric
OWL/XML Syntax	<AsymmetricObjectProperty> <ObjectProperty IRI="hasChild"/> </AsymmetricObjectProperty>

Tableau 4. 30: Exemple des différentes syntaxes OWL2 pour exprimer la propriété asymétrique.

4.5.8.4 La propriété réflexive :

Un axiome de réflexivité de propriété d'objet **ReflexiveObjectProperty**(OPE) affirme que l'expression de propriété d'objet OPE est réflexive - c'est-à-dire que chaque individu est connecté par OPE à lui-même. Chacun de ces axiomes peut être considéré comme un raccourci syntaxique pour l'axiome suivant :

SubClassOf(owl:Thing ObjectHasSelf(OPE))

Les propriétés peuvent également être réflexives ; une telle propriété rapporte tout à elle-même. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.31.

Functional-Style Syntax	ReflexiveObjectProperty(:hasRelative)
RDF/XML Syntax	<owl:ReflexiveProperty rdf:about="hasRelative"/>
Turtle Syntax	:hasRelative rdf:type owl:ReflexiveProperty .
Manchester Syntax	ObjectProperty: hasRelative Characteristics: Reflexive
OWL/XML Syntax	<ReflexiveObjectProperty> <ObjectProperty IRI="hasRelative"/> </ReflexiveObjectProperty>

Tableau 4. 31: Exemple des différentes syntaxes OWL2 pour exprimer la propriété réflexive.

Notez que cela ne signifie pas nécessairement que deux individus qui sont liés par une propriété réflexive sont identiques**.**

4.5.8.5 La propriété irréflexive :

Un axiome d'irréflexivité de propriété d'objet **IrreflexiveObjectProperty**(OPE) déclare que l'expression de propriété d'objet OPE est irréflexive - c'est-à-dire qu'aucun individu n'est connecté par OPE à lui-même. Chacun de ces axiomes peut être considéré comme un raccourci syntaxique pour l'axiome suivant :

SubClassOf(ObjectHasSelf(OPE) owl:Nothing)

Les propriétés peuvent en outre être irréflexives, ce qui signifie qu'aucun individu ne peut être relié à lui-même par un tel rôle. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.32.

Functional-Style Syntax	IrreflexiveObjectProperty(:parentOf)
RDF/XML Syntax	<owl:IrreflexiveProperty rdf:about="parentOf"/>
Turtle Syntax	:parentOf rdf:type owl:IrreflexiveProperty .
Manchester Syntax	ObjectProperty: parentOf Characteristics: Irreflexive
OWL/XML Syntax	<IrreflexiveObjectProperty> <ObjectProperty IRI="parentOf"/> </IrreflexiveObjectProperty>

Tableau 4. 32: Exemple des différentes syntaxes OWL2 pour exprimer la propriété irréflexive.

4.5.8.6 La propriété fonctionnelle :

Si une propriété P est marquée comme fonctionnelle, alors pour tout x, y, z :

P(x,y) et P(x,z) implique y = z.

Autrement dit, un axiome de fonctionnalité de propriété d'objet **FunctionalObjectPropert**y(OPE) déclare que l'expression de propriété d'objet OPE est fonctionnelle - c'est-à-dire que pour chaque individu x, il peut y avoir au plus un individu distinct y tel que x est connecté par OPE à y. Chacun de ces axiomes peut être vu comme un raccourci syntaxique pour l'axiome suivant :

SubClassOf(owl:Thing ObjectMaxCardinality(1 OPE))

Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.33.

Functional-Style Syntax	FunctionalObjectProperty(:hasHusband)
RDF/XML Syntax	<owl:FunctionalProperty rdf:about="hasHusband"/>
Turtle Syntax	:hasHusband rdf:type owl:FunctionalProperty .
Manchester Syntax	ObjectProperty: hasHusband Characteristics: Functional
OWL/XML Syntax	<FunctionalObjectProperty> <ObjectProperty IRI="hasHusband"/> </FunctionalObjectProperty>

Tableau 4. 33: Exemple des différentes syntaxes OWL2 pour exprimer la propriété fonctionnelle.

4.5.8.7 La propriété inverse fonctionnelle

Un axiome de fonctionnalité inverse de propriété d'objet

InverseFunctionalObjectProperty(OPE) stipule que l'expression de propriété d'objet OPE est à fonctionnalité inverse - c'est-à-dire que pour chaque individu x, il peut y avoir au plus un individu y tel que y est connecté par OPE avec x. Chacun de ces axiomes peut être vu comme un raccourci syntaxique pour l'axiome suivant :

SubClassOf(owl:Thing ObjectMaxCardinality(1 ObjectInverseOf(OPE)))

Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.34.

Functional-Style Syntax	InverseFunctionalObjectProperty(:hasHusband)
RDF/XML Syntax	<owl:InverseFunctionalProperty rdf:about="hasHusband"/>
Turtle Syntax	:hasHusband rdf:type owl:InverseFunctionalProperty .
Manchester Syntax	ObjectProperty: hasHusband Characteristics: InverseFunctional
OWL/XML Syntax	<InverseFunctionalObjectProperty> <ObjectProperty IRI="hasHusband"/> </InverseFunctionalObjectProperty>

Tableau 4. 34: Exemple des différentes syntaxes OWL2 pour exprimer la propriété inverse fonctionnelle.

4.5.8.8 La propriété transitivité

Un axiome de transitivité de propriété d'objet **TransitiveObjectProperty(** OPE) déclare que l'expression de propriété d'objet OPE est transitive - c'est-à-dire que si un individu x est connecté par OPE à un individu y qui est connecté par OPE à un individu z, alors x est également connecté par OPE à z. Chacun de ces axiomes peut être considéré comme un raccourci syntaxique pour l'axiome suivant :

SubObjectPropertyOf(ObjectPropertyChain(OPE OPE) OPE)

Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.35.

Functional-Style Syntax	TransitiveObjectProperty(:hasAncestor)
RDF/XML Syntax	<owl:TransitiveProperty rdf:about="hasAncestor"/>
Turtle Syntax	:hasAncestor rdf:type owl:TransitiveProperty .
Manchester Syntax	ObjectProperty: hasAncestor Characteristics: Transitive
OWL/XML Syntax	<TransitiveObjectProperty> <ObjectProperty IRI="hasAncestor"/> </TransitiveObjectProperty>

Tableau 4. 35: Exemple des différentes syntaxes OWL2 pour exprimer la propriété transitive.

4.5.8.9 Les propriétés disjointes :

Un axiome de propriétés d'objet disjoint DisjointObjectProperties(OPE_1 ... OPE_n) déclare que toutes les expressions de propriétés d'objet OPE_i, $1 \leq i \leq n$, sont disjointes par paire ; c'est-à-dire qu'aucun individu x ne peut être relié à un individu y à la fois par OPE_i et OPE_j pour $i \neq j$.

DisjointObjectProperties := 'DisjointObjectProperties' '(' axiomAnnotations
ObjectPropertyExpression axio Annotations { ObjectPropertyExpression } ')'

Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.36.

Functional-Style Syntax	DisjointObjectProperties(:hasParent :hasSpouse)
RDF/XML Syntax	<rdf:Description rdf:about="hasParent"> <owl:propertyDisjointWith rdf:resource="hasSpouse"/> </rdf:Description>
Turtle Syntax	:hasParent owl:propertyDisjointWith :hasSpouse .
Manchester Syntax	DisjointProperties: hasParent, hasSpouse
OWL/XML Syntax	<DisjointObjectProperties> <ObjectProperty IRI="hasParent"/> <ObjectProperty IRI="hasSpouse"/> </DisjointObjectProperties>

Tableau 4. 36: Exemple des différentes syntaxes OWL2 pour exprimer 9 Les propriétés disjointes**.**

On peut également exprimer :

- **Sous-propriétés des objets :**

Les axiomes de sous-propriété d'objet sont analogues aux axiomes de sous-classe, et ils se présentent sous deux formes.

La forme de base est **SubObjectPropertyOf(** OPE_1 OPE_2). Cet axiome affirme que l'expression de propriété d'objet OPE1 est une sous-propriété de l'expression de propriété d'objet OPE_2 - c'est-à-dire que si un individu x est relié par OPE1 à un individu y, alors x est également relié par OPE2 à y.

- **propriétés d'objet équivalentes :**

Un axiome de **EquivalentObjectProperties**(OPE_1 ... OPEn) déclare que toutes les expressions de propriétés d'objet OPE_i, $1 \leq i \leq n$, sont sémantiquement équivalentes les unes aux autres. Cet axiome permet d'utiliser chaque OPE_i comme synonyme de chaque OPE_j - c'est-à-dire que dans toute expression de l'ontologie contenant un tel axiome, on peut remplacer OPE_i par OPE_j sans affecter le sens de l'ontologie. L'axiome EquivalentObjectProperties(OPE1 OPE2) est équivalent aux deux axiomes suivants :

SubObjectPropertyOf(OPE1 OPE2)

SubObjectPropertyOf(OPE2 OPE1)

4.5.8.10 Les chaines de propriétés :

La forme la plus complexe est SubObjectPropertyOf(**ObjectPropertyChain**(OPE_1 ... OPE_n) OPE). Cet axiome stipule que, si un individu x est connecté par une séquence d'expressions de propriété d'objet OPE_1, ..., OPE_n avec un individu y, alors x est également connecté avec y par l'expression de propriété d'objet OPE. De tels axiomes sont également connus sous le nom d'inclusions de rôles complexes. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.37.

Functional-Style Syntax	SubObjectPropertyOf(ObjectPropertyChain(:hasParent :hasParent) :hasGrandparent)
RDF/XML Syntax	<rdf:Description rdf:about="hasGrandparent"> <owl:propertyChainAxiom rdf:parseType="Collection"> <owl:ObjectProperty rdf:about="hasParent"/> <owl:ObjectProperty rdf:about="hasParent"/> </owl:propertyChainAxiom> </rdf:Description>
Turtle Syntax	:hasGrandparent owl:propertyChainAxiom (:hasParent :hasParent) .
Manchester Syntax	ObjectProperty: hasGrandparent SubPropertyChain: hasParent o hasParent
OWL/XML Syntax	<SubObjectPropertyOf> <ObjectPropertyChain> <ObjectProperty IRI="hasParent"/> <ObjectProperty IRI="hasParent"/> </ObjectPropertyChain> <ObjectProperty IRI="hasGrandparent"/> </SubObjectPropertyOf>

Tableau 4. 37: Exemple des différentes syntaxes OWL2 pour exprimer les chaînes de propriété.

4.5.9 Les clés (keys)

Dans OWL 2, un ensemble de propriétés (données ou objets) peut être attribué comme clé à une expression de classe. Cela signifie que chaque instance nommée de l'expression de classe est identifiée de manière unique par l'ensemble des valeurs que ces propriétés atteignent en relation avec l'instance. Cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.38.

Functional-Style Syntax	HasKey(:Person () (:hasSSN))
RDF/XML Syntax	<owl:Class rdf:about="Person"> <owl:hasKey rdf:parseType="Collection"> <owl:DataProperty rdf:about="hasSSN"/> </owl:hasKey> </owl:Class>
Turtle Syntax	:Person owl:hasKey (:hasSSN) .
Manchester Syntax	Class: Person HasKey: hasSSN
OWL/XML Syntax	<HasKey> <Class IRI="Person"/> <DataProperty IRI="hasSSN"/> </HasKey>

Tableau 4. 38: Exemple des différentes syntaxes OWL2 pour exprimer les chaînes de propriété.

4.5.10 Restrictions des propriétés de données

Les expressions de classes dans OWL 2 peuvent être formées en plaçant des restrictions sur les expressions de propriétés de données, comme le montre la figure 4.7. Ces restrictions sont similaires à celles des expressions de propriétés d'objets, la principale différence étant que les expressions de quantification existentielle et universelle permettent des plages de données n-aires. Toutes les plages de données explicitement prises en charge par cette spécification sont unaires. Cependant, la fourniture de plages de données n-aires dans la quantification existentielle et universelle permet aux outils OWL 2 de prendre en charge des extensions telles que les comparaisons de valeurs et, par conséquent, des expressions de classe telles que "individus dont la largeur est supérieure à la hauteur". Ainsi, l'expression de classe **DataSomeValuesFrom** permet une quantification existentielle restreinte sur une liste d'expressions de propriétés de données, et contient les individus qui sont connectés par les expressions de propriétés de données à au moins un littéral dans la plage de données donnée. L'expression de la classe **DataAllValuesFrom** permet

une quantification universelle restreinte sur une liste d'expressions de propriétés de données, et contient les individus qui sont connectés par l'intermédiaire des expressions de propriétés de données uniquement à des littéraux dans la plage de données donnée. Enfin, l'expression de classe **DataHasValue** contient les individus qui sont reliés par une expression de propriété de données à un littéral particulier.

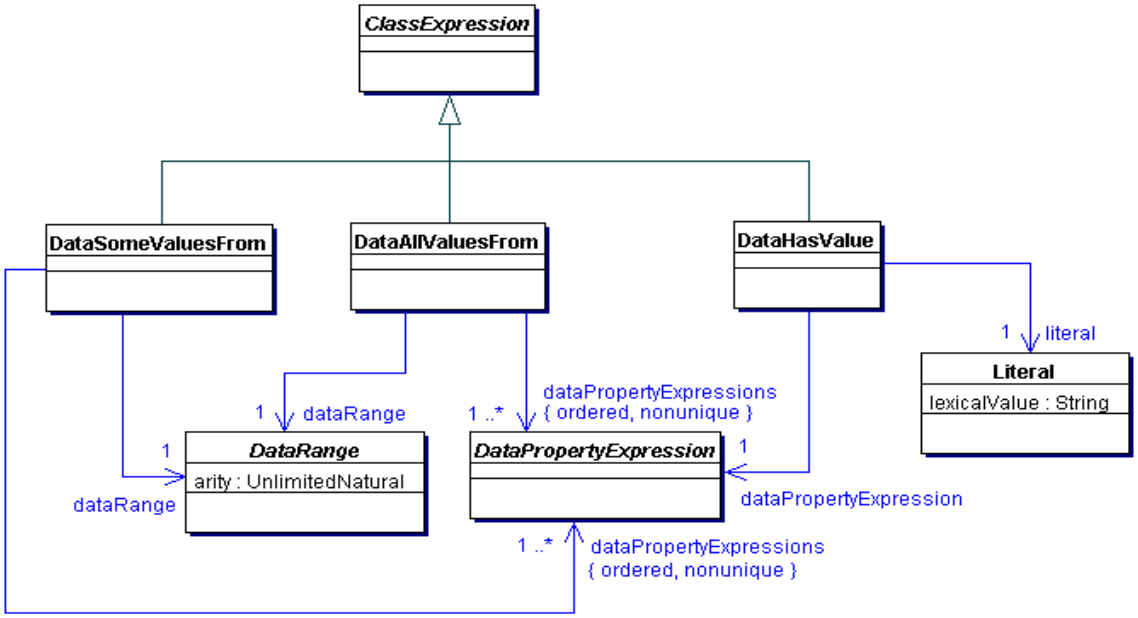


Figure 4. 6 : Restrictions des propriétés de données dans OWL 2.

4.5.10.1 Quantification existentielle DataSomeValuesFrom

Une expression de classe existentielle **DataSomeValuesFrom**(DPE_1 ... DPE_n DR) est constituée de n expressions de propriétés de données DPE_i, $1 \leq i \leq n$, et d'un intervalle de données DR dont l'arité doit être n. Une telle expression de classe contient tous les individus qui sont reliés par DPE_i à des littéraux lt_i, $1 \leq i \leq n$, tels que le tuple (lt_1 , ..., lt_n) est dans DR. Une expression de classe de la forme DataSomeValuesFrom(DPE DR) peut être vue comme un raccourci syntaxique de l'expression de classe DataMinCardinality(1 DPE DR).

4.5.10.2 Quantification universelle DataAllValuesFrom

Une expression de classe universelle **DataAllValuesFrom**(DPE_1 ... DPE_n DR) est constituée de n expressions de propriétés de données DPE_i, $1 \leq i \leq n$, et d'une plage de données DR dont l'arité doit être n. Une telle expression de classe contient tous les individus qui sont reliés par DPEi uniquement à des littéraux lti, $1 \leq i \leq n$, de telle sorte que chaque tuple (lt_1 , ..., lt_n) soit dans DR. Une expression de classe de la forme **DataAllValuesFrom**(DPE DR) peut être vue comme un raccourci syntaxique de l'expression de classe DataMaxCardinality(0 DPE DataComplementOf(DR)).

4.5.10.3 Restriction de valeur littérale DataHasValue

Une expression de classe has-value **DataHasValue**(DPE l_t) se compose d'une expression de propriété de données DPE et d'un littéral $_{lt,}$ et contient tous les individus qui sont reliés par DPE à l_t. Chacune de ces expressions de classe peut être considérée comme un raccourci syntaxique de l'expression de classe DataSomeValuesFrom(DPE DataOneOf(lt)).

4.5.11 Restrictions de cardinalité des propriétés de données

Les expressions de classes dans OWL 2 peuvent être formées en plaçant des restrictions sur la cardinalité des expressions de propriétés de données, comme le montre la figure 4.8 . Ces restrictions sont similaires aux restrictions sur la cardinalité des expressions de propriétés d'objets. Toutes les restrictions de cardinalité peuvent être qualifiées, ou non. Dans le premier cas, la restriction de cardinalité ne s'applique qu'aux littéraux qui sont connectés par l'expression de propriété de données et qui sont dans la plage de données qualifiante, dans le second cas, elle s'applique à tous les littéraux qui sont connectés par l'expression de propriété de données (ceci est équivalent au cas qualifié avec la plage de données qualifiante égale à rdfs:Literal). Les expressions de classe **DataMinCardinality**, **DataMaxCardinality** et **DataExactCardinality** contiennent les individus qui sont reliés par une expression de propriété de données à au moins, au plus et exactement un nombre donné de littéraux dans la plage de données spécifiée, respectivement.

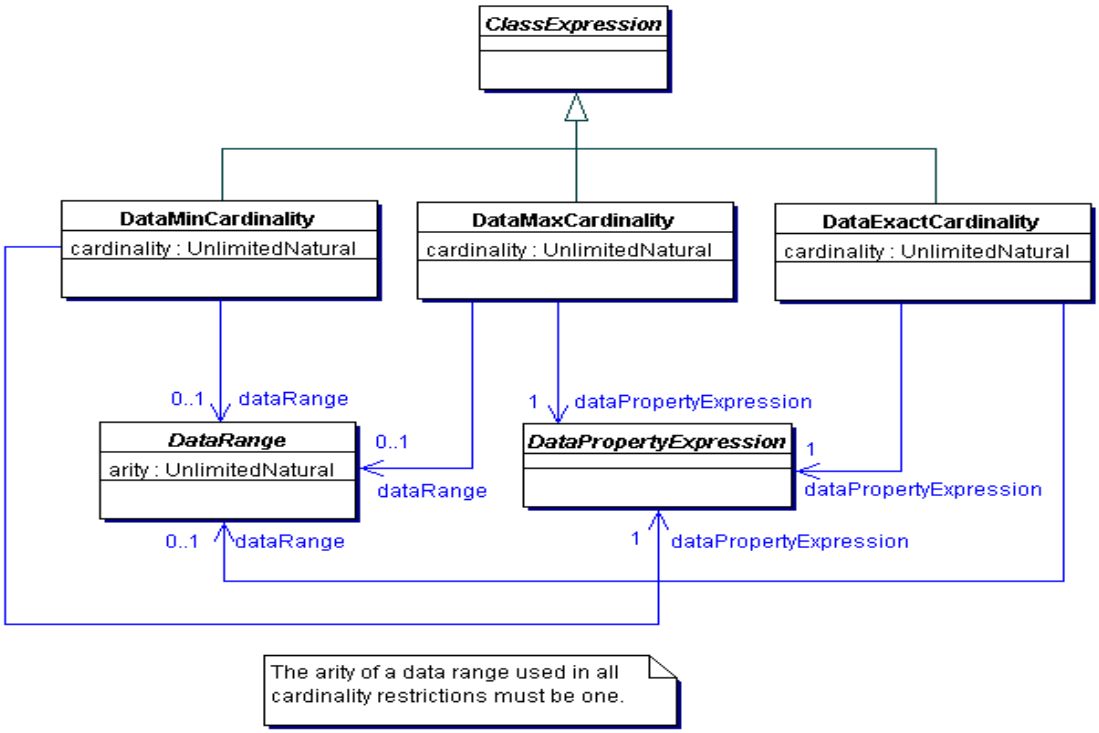


Figure 4. 7 : Restrictions de cardinalité des propriétés de données.

4.5.11.1 Cardinalité minimale

Une expression de cardinalité minimale **DataMinCardinality(** n DPE DR) se compose d'un nombre entier non négatif n, d'une expression de propriété de données DPE et d'une plage de données unaire DR. Elle contient tous les individus qui sont reliés par DPE à au moins n littéraux différents dans DR. Si DR n'est pas présent, il est considéré comme étant rdfs:Literal.

4.5.11.2 Cardinalité maximale

Une expression de cardinalité maximale **DataMaxCardinality**(n DPE DR) se compose d'un entier n-négatif, d'une expression de propriété de données DPE et d'une plage de données unaire DR. Elle contient tous les individus qui sont reliés par DPE à au plus n littéraux différents dans DR. Si DR n'est pas présent, il est considéré comme étant rdfs:Literal.

4.5.11.3 Cardinalité exacte

Une expression de cardinalité exacte **DataExactCardinality**(n DPE DR) se compose d'un nombre entier non négatif n, d'une expression de propriété de données DPE et d'une plage de données unaire DR. Elle contient tous les individus qui sont reliés par DPE à exactement n littéraux différents dans DR. Si DR n'est pas présent, il est considéré comme étant rdfs:Literal.

4.5.12 Déclarations d'entités

Pour faciliter la gestion des ontologies, OWL dispose de la notion de déclarations. L'idée de base est que chaque classe, propriété ou individu est censé être déclaré dans une ontologie, et qu'il peut ensuite être utilisé dans cette ontologie et dans les ontologies qui importent cette ontologie : cela peut être exprimé en OWL 2 sous les différentes syntaxes. Consulter le tableau 4.39.

Functional-Style Syntax	Declaration(NamedIndividual(:John)) Declaration(Class(:Person)) Declaration(ObjectProperty(:hasWife)) Declaration(DataProperty(:hasAge))
RDF/XML Syntax	<owl:NamedIndividual rdf:about="John"/> <owl:Class rdf:about="Person"/> <owl:ObjectProperty rdf:about="hasWife"/> <owl:DatatypeProperty rdf:about="hasAge"/>
Turtle Syntax	:John rdf:type owl:NamedIndividual . :Person rdf:type owl:Class . :hasWife rdf:type owl:ObjectProperty . :hasAge rdf:type owl:DatatypeProperty .
Manchester Syntax	Individual: John Class: Person ObjectProperty: hasWife DataProperty: hasAge
OWL/XML Syntax	<Declaration> <NamedIndividual IRI="John"/> </Declaration> <Declaration> <Class IRI="Person"/> </Declaration> <Declaration> <ObjectProperty IRI="hasWife"/> </Declaration> <Declaration> <DataProperty IRI="hasAge"/> </Declaration>

Tableau 4. 39 : Exemple des différentes syntaxes OWL2 pour exprimer les déclarations d'entités.

En pratique, une syntaxe concrète est nécessaire pour stocker les ontologies OWL 2 et les échanger entre outils et applications. La principale syntaxe d'échange pour OWL 2 est RDF/XML [RDF Syntax]. C'est en effet la seule syntaxe qui doit être supportée par tous les outils OWL 2.

Bien que **RDF/XML** assure l'interopérabilité entre les outils OWL 2, d'autres syntaxes concrètes peuvent également être utilisées. Il s'agit notamment de sérialisations RDF alternatives, telles que **Turtle** [Turtle]. **Une sérialisation XML** [OWL 2 XML] possède une syntaxe plus "lisible", appelée la **syntaxe Manchester** [OWL 2 Manchester Syntax], qui est utilisée dans plusieurs outils d'édition d'ontologie. Enfin, **la syntaxe de style fonctionnel** peut également être utilisée pour la sérialisation, bien que son objectif principal soit de spécifier la structure du langage [OWL 2 Structural Specification]. Pour plus de détails sur la syntaxe, consulter les liens dans la colonne spécification. Voir le tableau 4.40

Name of Syntax	Specification	Status	Purpose
RDF/XML	Mapping to RDF Graphs, RDF/XML	Mandatory	Interchange (can be written and read by all conformant OWL 2 software)
OWL/XML	XML Serialization	Optional	Easier to process using XML tools
Functional Syntax	Structural Specification	Optional	Easier to see the formal structure of ontologies
Manchester Syntax	Manchester Syntax	Optional	Easier to read/write DL Ontologies
Turtle	Mapping to RDF Graphs, Turtle	Optional, Not from OWL-WG	Easier to read/write RDF triples

Tableau 4. 40: les différents syntaxes OWL2 et leurs spécifications

Les tableaux : tableau 4. 41et tableau 4 .42 présentent successivementt les constructeurs OWL et les axiomes OWL basées sur la syntaxe LD.

Constructeur OWL	Syntaxe LD
intersectionOf (C_1, C_2, ...)	$C_1 \sqcap C_2$
unionOf (C_1, C_2, ...)	$C_1 \sqcup C_2$
complementOf(C)	$\neg C$
oneOf(I_1,I_2, ...)	$\{I_1, I_2, ...\}$
Restriction(P allValuesFrom(C))	$\forall P.C$
Restriction(P someValuesFrom(C))	$\exists P.C$
Restriction(P hasValue(I))	$P : I$
Restriction(P cardinality(n))	$= nP$
Restriction(P minCardinality(n))	$\leqslant nP$
Restriction(P maxcardinality(n))	$\geqslant nP$

Tableau 4. 42 Les constructeurs OWL

Constructeur OWL	Syntaxe LD
Axiomes sur les classes	
subClassOf (C_1, C_2)	$C_1 \sqsubseteq C_2$
equivalentClasses (C_1, ..., C_n)	$C_1 \equiv ... \equiv C_n$
disjointWith (C_1, ..., C_2)	$C_1 \sqsubseteq \neg C_2$
Axiomes sur les propriétés	
subPropertyOf (P_1, P_2)	$P_1 \sqsubseteq P_2$
equivalentProperties (P_1, ..P_n	$P_1 \equiv ... \equiv P_n$)
inverseOf (P_1, P_2)	$P_1 \equiv P_2^-$
symetricProperty(P)	$P \equiv P^-$
functionalProperty (P)	$\top \sqsubseteq \leqslant 1P$
inverseFunctionalProperty (P)	$\top \sqsubseteq \leqslant 1P^-$
transitiveProperty(P)	$Tr(P)$
Axiomes sur les individus	
sameAs(I_1, ..., I_n)	$I_1 = ... = I_n$
differentFrom(I_1, ..., I_n)	$I_1 \neq ... \neq I_n$

Tableau 4. 43 Les axiomes OWL.

4.6 Conclusion

Le langage d'ontologie Web OWL 2, informellement OWL 2, est un langage d'ontologie pour le Web sémantique avec une signification formellement définie. Les ontologies OWL 2 fournissent des classes, des propriétés, des individus et des valeurs de données et sont stockées sous forme de documents du Web sémantique. OWL 2 est une "version actualisée" de OWL Extensions dues aux expériences pratiques avec OWL 1.0. Cette version est caractérisée par :

- L'expressivité supplémentaire due à de nouveaux axiomes ontologiques
- Les extensions extralogiques (syntaxe, métadonnées, etc.)
- La révision complète des variantes de OWL (Lite/DL/Full).

OWL 2 utilise la sémantique basée sur RDF [OWL 2 RDF-Based Semantics] ou la sémantique directe [OWL 2 Direct Semantics] pour le raisonnement et elle peut être exprimée sous les différentes syntaxes.

- **La syntaxe abstraite OWL** présente une ontologie comme une séquence d'annotations, d'axiomes et de faits. Les annotations transportent des métadonnées orientées machine et humaine. Les informations sur les classes, les propriétés et les individus qui composent l'ontologie sont contenues dans des axiomes et des faits uniquement. Chaque classe, propriété et individu est soit anonyme, soit identifié par une référence URI. Les faits indiquent des données soit sur un individu, soit sur une paire d'identifiants individuels (que les objets identifiés sont distincts ou identiques). Les axiomes spécifient les caractéristiques des classes et des propriétés. Ce style, similaire aux langages de cadre, est assez différent des syntaxes bien connues pour les DL et Cadre de description des ressources (RDF).
- **Syntaxe fonctionnelle OWL2** : Cette syntaxe suit de près la structure d'une ontologie OWL2. Il est utilisé par OWL2 pour spécifier la sémantique, les mappages pour échanger des syntaxes et des profils.
- **Syntaxe RDF /XML** : Les mappages syntaxiques dans RDF sont spécifiés pour les langages de la famille OWL. Plusieurs formats de sérialisation RDF ont été imaginés. Chacun conduit à une syntaxe pour les langages de la famille OWL à travers ce mappage. RDF/XML est normatif.
- **Syntaxe XML OWL2** : OWL2 spécifie une sérialisation XML qui modélise étroitement la structure d'une ontologie OWL2.
- **Syntaxe de Manchester** : La syntaxe Manchester est une syntaxe compacte et lisible par l'homme avec un style proche des langages de trame. Des variantes sont disponibles pour OWL et OWL2. Toutes les ontologies OWL et OWL2 ne peuvent pas être exprimées dans cette syntaxe.

- **Syntaxe Turtle** (Terse RDF Triple Language) est une syntaxe d'un langage qui permet une sérialisation non-XML des modèles RDF. C'est un sous-ensemble de la syntaxe Notation3.

Dans notre approche ; nous allons adopter la syntaxe fonctionnelle OWL2 (Functional Syntax).

CHAPITRE 5

Contributions

Sommaire

5.1 Introduction

Les modèles conceptuels représentent les caractéristiques statiques d'un système, qui se réfèrent aux modèles formés après une conceptualisation. L'ontologie est une description formelle de la conceptualisation d'un domaine, qui est composée d'un ensemble de noms pour les concepts, les rôles et les individus afin de représenter une base de connaissances composée d'un ensemble terminologique et d'un ensemble d'assertions.

Dans la partie terminologique (TBOX), nous définissons des axiomes pour les concepts généraux et les inclusions de rôles. Les rôles (relations) peuvent être entre paires de concepts (appelés propriétés d'objet) ou entre concepts et type de données (appelés propriétés de type de données).

L'ensemble des assertions (ABOX) sont des instances de concepts (axiomes d'appartenance aux concepts) pour affirmer qu'un individu (objet) appartient à un concept, des instances de propriétés d'objets (axiomes d'appartenance aux propriétés d'objets) pour affirmer qu'un individu a une relation avec un autre individu, ou des instances de propriétés de types de données (axiomes d'appartenance aux propriétés de types de données) pour affirmer qu'un individu a une propriété de données.

Le principal domaine d'application des ontologies est le Web sémantique, où les documents Web sont annotés avec des informations (méta-données) issues de la terminologie ontologique.

Ces documents Web sont des instances de concepts et de rôles de l'ontologie. Ainsi, le processus d'annotation est considéré comme la création d'assertions. La terminologie et les ensembles d'assertions représentent la base de connaissances des agents Web (programmes). Le Web sémantique a une architecture composée de ressources (documents Web) et d'un ensemble d'agents intelligents, chacun ayant sa propre ontologie. Les ontologies sont utilisées dans la recherche d'informations comme outil de recherche intelligent utilisant le mécanisme d'inférence comme alternative à la correspondance des mots clés. Le rôle de ces agents intelligents est de répondre aux requêtes des utilisateurs en exécutant des règles d'inférence sur leur base de connaissances (ontologie).

L’Ingénierie Dirigée par les Modèles (IDM) est une approche de développement mettant à disposition de l'utilisateur des concepts, des langages et des outils. Les modèles sont considérés comme des éléments de base. Le raisonnement est entièrement à un haut niveau d'abstraction. L’application sera générée (en tout ou en partie, automatiquement ou semi-automatiquement) à

partir de modèles. Les outils permettant de créer et d'exploiter ces modèles sont construits autour des concepts de Méta-modélisation et de Transformation de modèles et génération de code.

Dans ce chapitre nous proposerons une approche MDA (Model Driven Architecture) pour la formalisation et la transformation d'ontologies.

Nous développerons également les méta- modèles source (méta- modèles des diagrammes UML), les méta- modèles destination (méta- modèles des ontologies) et le méta-modèle de correspondance entre elles. Nous poursuivrons en décrivant les règles de transformation (37 regèles s de transformation) qui assurent la transformation bidirectionnelle (transformation modèle à modèle) entre la source et la destination, qui par la suite effectue une transformation modèle 2 texte pour générer le code OWL 2 équivalent.

5.2 Environnement d'implémentation

Dans cette section, nous présenterons succinctement les techniques et les outils que nous avons utilisés pour implémenter notre approche. Puisque notre approche est basée sur MDA, nous avons besoin d'une infrastructure adaptée à la méta-modélisation, la modélisation et les transformations des modèles M2M (Model-To-Model) et M2T (Model-To-Text). Pour cette raison, nous avons retenu EMF (Eclipse Modeling Framework) (Budinsky, et al.. (August 1, 2003)) (Steinberg , et al.. 2008) comme plateforme d'implantation.
EMF fournit un ensemble d'outils destinés à introduire une approche dirigée par les modèles au sein de l'environnement Eclipse[40]. Ces outils apportent deux fonctionnalités primordiales :

- La première est la métamodélisation : La métamodélisation est l'activité consistant à définir le métamodèle d'un langage de modélisation. Elle vise donc à modéliser un langage correctement, qui jouera alors le rôle de système à modéliser.
- La deuxième fonctionnalité correspond à la transformation de modèles : Une transformation est une génération automatique d'un ou de plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources, en respectant une définition de transformation. Une définition de transformation est un ensemble de règles de transformation qui décrivent la manière avec laquelle un modèle dans le langage source peut être transformé en un modèle dans le langage cible. Une règle de transformation est une description de la façon avec laquelle une ou plusieurs constructions dans le langage source peuvent être transformées en une ou plusieurs constructions dans le langage cible.

[40] Projets EMF et SOA Tools Platform de Eclipse. Disponible www.eclipse.org/stp/ and www.eclipse.org/emf/

5.2.1 Plateforme Eclipse :

Eclipse est une communauté open source dont les projets sont axés sur la construction d'une plate-forme de développement. La plateforme Eclipse, initiée par IBM, est un atelier de génie logiciel (AGL) conçu dans le but de fournir une plateforme modulaire et extensible pour la construction, le déploiement et la gestion des logiciels, couvrant l'ensemble de leurs cycles de vie.

Tout le code de la plateforme a été mis à la disposition de la communauté open source afin de contribuer à son développement. Eclipse propose un ensemble de projets adaptés en fonction du métier des utilisateurs. On peut citer: BIRT, RT, SOA Platform Project, Eclipse Web Tools Platform Projects, Data Tools Platform, Eclipse Modeling Project, etc. La communauté du MDE, s'intéresse plus particulièrement au projet Eclipse Modeling Project (EMP).

Nous exposerons sur la figure 5.1 une présentation de la structure de l'EMP sous forme moléculaire, et sous forme atomique, celle de ses domaines fonctionnels organisés en différents projets à savoir : Abstract Syntax Development (ASD), Model-to-Model (M2M), Model-to-Text (M2T), Textual Concrete Syntax (TCS), Graphical Concrete Syntax (GCS), Model Development Tools (MDT) et le Eclipse Model Framework Technology (EMFT).

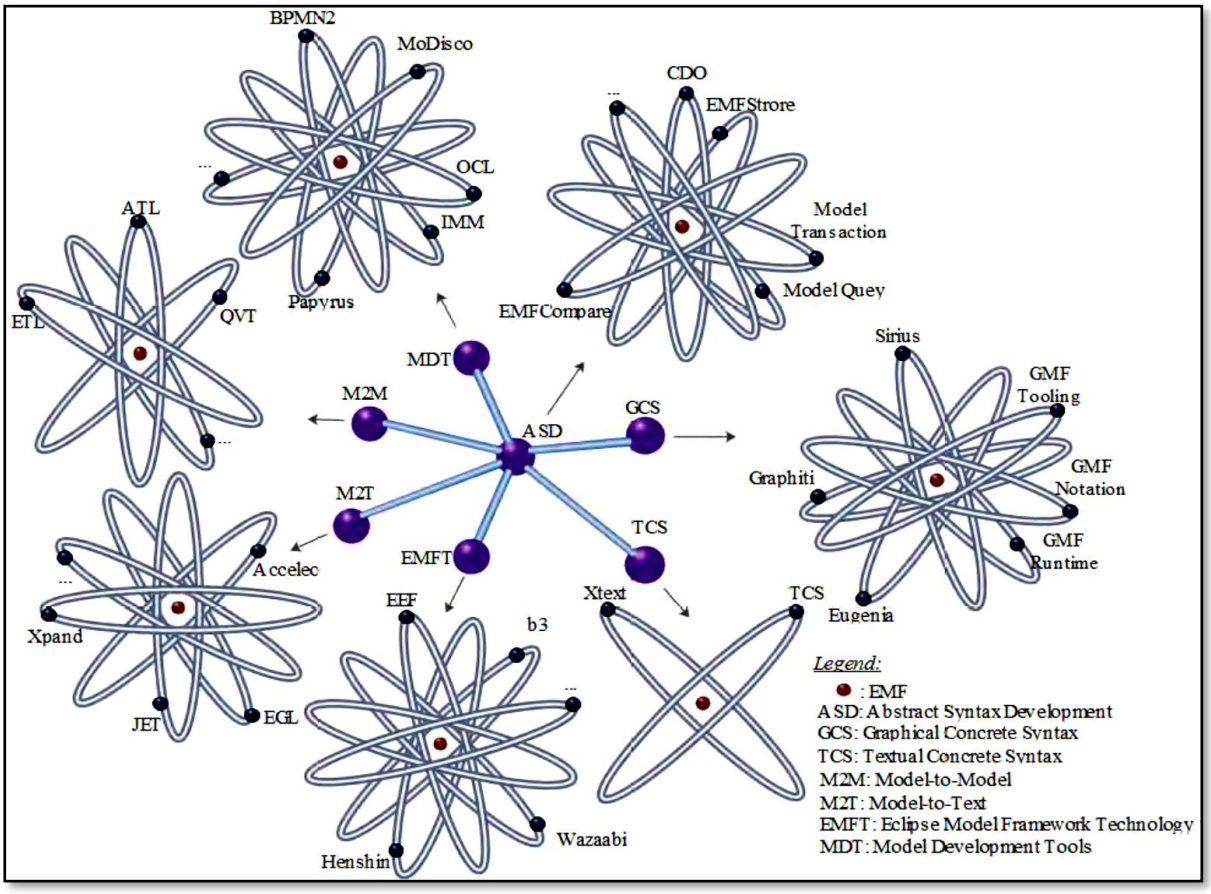


Figure 5. 1 : Structuration de Eclipse Modeling Project (El Hamlaoui 30 May 2016)

La Figure 5.2 illustre les composants de la version 4.x. Elle introduit l'E4AP (Eclipse 4 Application Plateform) qui se base sur de nouvelles technologies à savoir :

- Interface utilisateur à base du modèle EMF (Eclipse Modeling Framework) qui sera utilisée par la suite pour générer le code SWT (Standard Widget Toolkit),
- Utilisation des feuilles de style CSS (Cascading Style Sheets) pour changer facilement l'apparence et la convivialité de la plateforme Eclipse,
- Modèle d'injection de dépendances en utilisant les annotations de la JSR 330,
- Modèle de programmation à base de services.

L'E4AP contient aussi quatre composants :

- Le premier est une implémentation de la spécification OSGi (Open Service Gateway initiative) : Equinox. Elle fournit le cadre nécessaire pour exécuter une application Eclipse modulaire.
- Le deuxième composant est EMF qui est utilisé pour la manipulation des modèles,
- Les deux derniers sont les bibliothèques graphiques SWT et JFace.

Au-dessus de l'EA4P, la version 4.x définit un Workbench qui fournit le Framework pour l'application. Il est responsable de l'affichage de tous les autres composants de l'interface utilisateur. Etant donné que la majorité des plug-ins fonctionnent encore sous la version 3.x, une couche de compatibilité a été définie afin de les exécuter sous la version 4.x, sans modification de leurs contenus (Vogel 2014).

Finalement, à l'instar de la version 3.x, la version 4.x comprend un ensemble de composants : outil de développement Java (*Java Developement Tooling*), environnement de développement de plug-ins (*Plug-in Development Environment*), support de contrôle de versions, etc.

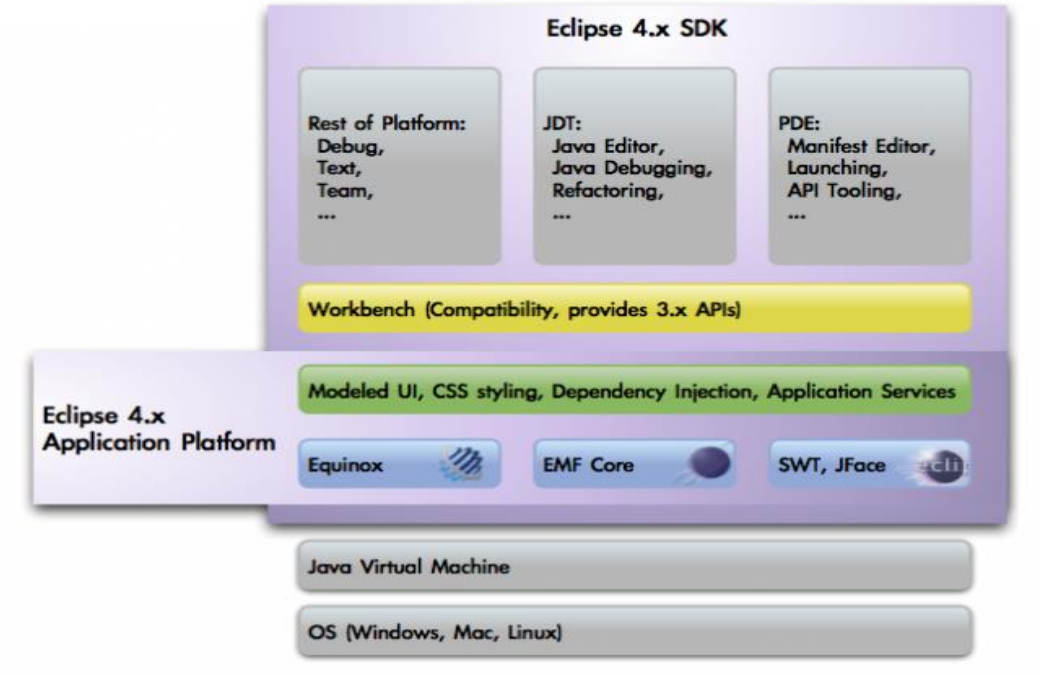


Figure 5. 2 : Architecture d'Eclipse 4.x (Eclipse 2011).

EMF est une plate-forme de modélisation et de génération de code qui facilite la construction d'outils. Il s'agit d'un ensemble d'outils de développement intégré à l'environnement Eclipse sous forme de plugins parmi lesquels on cite : le méta-modèle *Ecore*, l'éditeur *EMF.Edit*, le modèle de génération *GenModel*, etc. EMF a été conçu pour ouvrir Eclipse au développement dirigé par les modèles, c'est une approche basée sur une simplification du MOF. Il permet de définir des méta-modèles puis d'en dériver une implantation en Java pour construire des modèles instances.

La Figure 5.3 présente l'architecture globale du Framework EMF. Le rôle principal de cet outillage est d'accepter en entrée des modèles ou des fichiers et de générer en sortie du code correspondant à des outils (plug-in) manipulant les données fournies en entrée. Parmi les fonctionnalités d'EMF, on peut citer :

- La génération automatique d'un simple éditeur graphique permettant l'édition des modèles sous forme arborescente, c'est à dire générer automatiquement, à partir d'un méta-modèle, un éditeur graphique offrant une vue arborescente d'un modèle. Chacun des nœuds de l'éditeur représentera une instance d'une méta-classe.
- La génération des interfaces de manipulation des modèles consiste à fournir des interfaces graphiques génériques pour manipuler des modèles.
- La génération de code, c'est bien là son objectif premier, améliore la productivité de développement d'application par l'automatisation de la génération de code à partir du modèle. Effectivement, une fois le modèle créé, « quelques clics » suffisent à cette génération.

Cette génération de code est paramétrée à l'aide d'un méta-modèle appelé *GenModel*. Le modèle de génération est construit automatiquement à partir du modèle Ecore en associant un élément de paramétrage correspondant à chaque élément du modèle. Les informations détenues par ces éléments de paramétrage concernent différents aspects qui ont un impact sur la génération : règles de nommage, localisation des fichiers générés, mode de visualisation et d'édition, etc.

Cette approche de génération paramétrée par un modèle présente comme avantages de ne pas polluer les éléments de modélisation avec ingrédients relatifs à la génération et de pouvoir appliquer plusieurs générations à un même modèle. Par contre, cette approche présente un inconvénient concernant l'expression de la sémantique d'exécution qui est dans ce cas totalement transparent par rapport aux utilisateurs (Mallouli 2014).

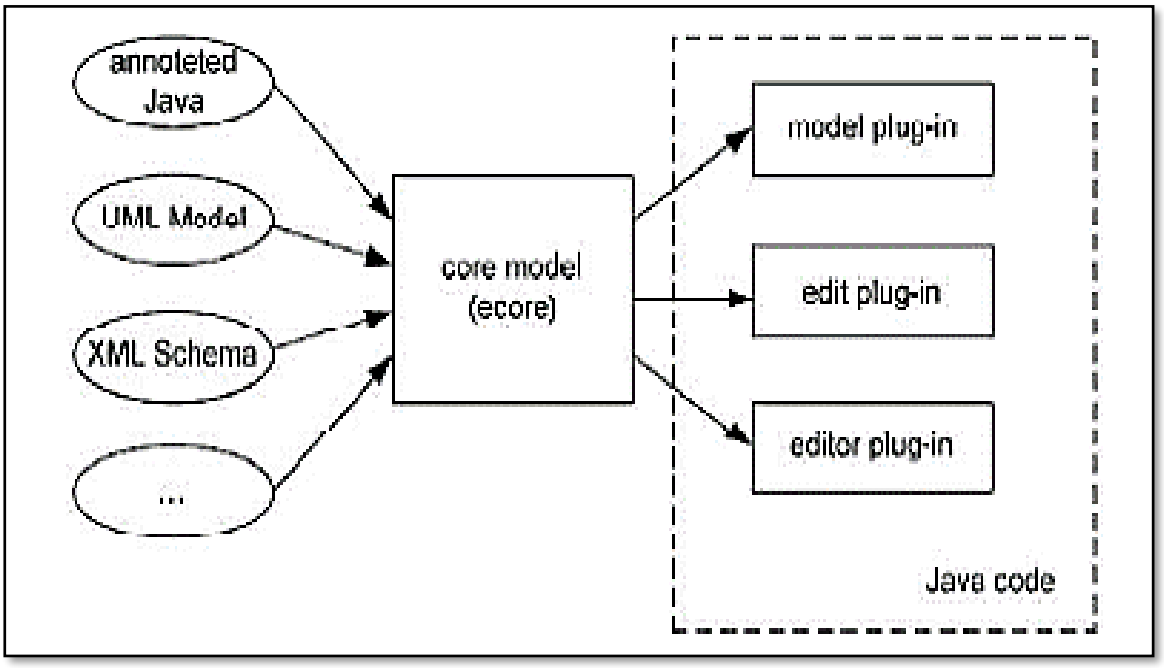


Figure 5. 3 : Architecture d'EMF

5.2.2 Les langages de méta-Modélisation:

Dans l'architecture à 4 niveaux de MDA, les langages de métamodélisation se situent au niveau M3, et permettent de spécifier des métamodèles au niveau M2. Dans l'approche MDA, le langage MOF incarne le socle architectural. Dans l'environnement Eclipse, le langage Ecore joue le même rôle que MOF, il permet la spécification de métamodèles.

5.2.2.1 Meta Object Facility (MOF) :

Le MOF est un standard de l'OMG (Object Management Group) depuis novembre 1997. C'est un formalisme, pour établir des langages de modélisation (méta-modèles) permettant eux-mêmes de définir des modèles.

Le but du MOF est de définir un langage de métamodélisation (un méta-méta-modèle) unique et utilisé par tous, pour représenter des métamodèles et les modèles qui en découlent. Il est constitué d'un ensemble relativement petit de concepts "objet" permettant de modéliser ce type d'information.

Le MOF peut être étendu par héritage ou par composition de manière à représenter des modèles plus évolués. Dans la Figure 5.4, on peut constater qu'avec MOF il est possible de décrire un métamodèle sous forme de diagramme de classes (une description statique), on peut exprimer partiellement la dynamique d'un objet grâce aux opérations liées à une classe (mais cela s'arrête à une signature de l'opération). Avec des contraintes OCL, il est aussi possible d'exprimer partiellement l'aspect dynamique d'un modèle, mais cette expression reste limitée car avec des conditions OCL on ne peut pas invoquer, ou déclencher, des opérations et on ne peut pas non plus capturer le détail de l'algorithme d'exécution ou de simulation d'un modèle. La liaison entre MOF et OCL n'existe pas, n'est pas représenté à la Figure 5.4, mais même si on l'ajoute, ce lien n'est pas formalisé. L'absence de cette relation au niveau conceptuel nécessite un effort important de programmation et une expertise de la part du développeur de l'outil d'exécution du modèle (qui doit savoir exactement comment enchainer les contraintes par exemple).

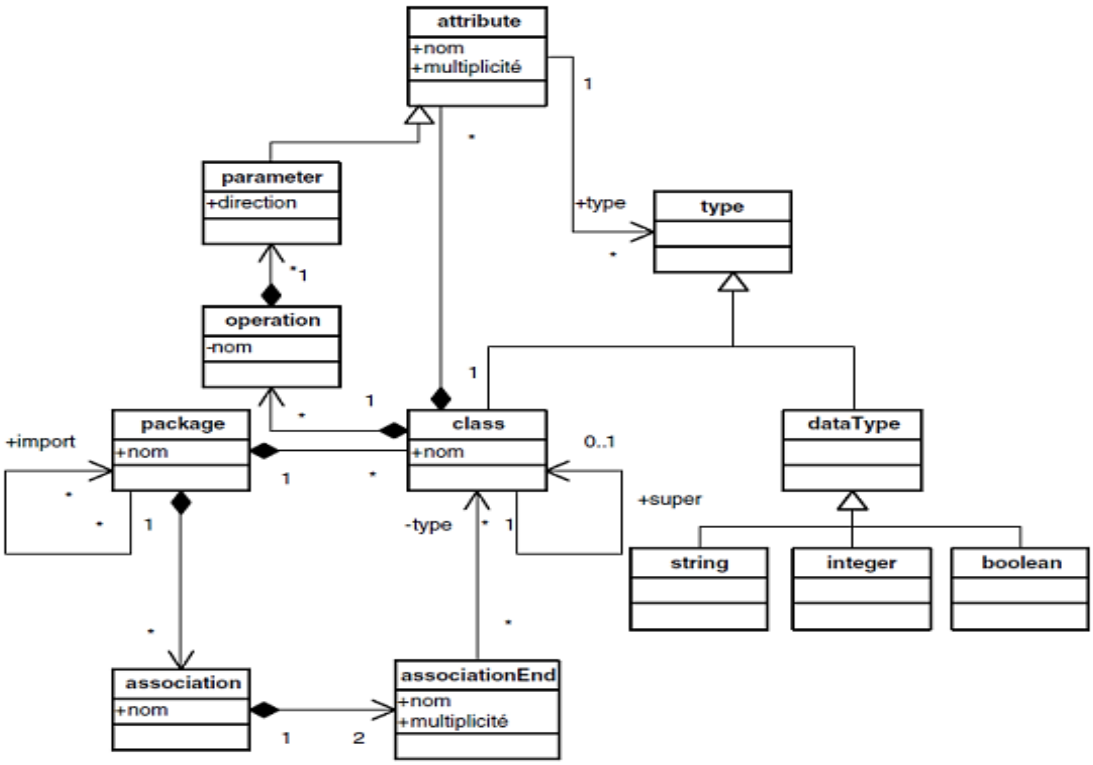


Figure 5. 4 : *Les concepts principaux de MOF 1.4 (Blanc et Salvator avril 2005).*

Les principaux concepts de MOF 1.4 illustrés par la figure 5.4 sont les suivants :

- **Class** : Une classe (désignée par métaclasse dans MOF) permet de définir la structure de ses instances (désignées par méta-objets). Un ensemble de méta-objets constitue un modèle. Une métaclasse possède un nom, contient des attributs et des opérations aussi appelés méta-attributs et méta-opérations. Les désignations en "méta" permettent de discerner les entités du niveau M2 de celles du niveau M1.
- **DataType** : Un type de donnée permet de spécifier le type non-objet d'un méta-attribut ou d'un paramètre d'une méta-opération.
- **Association**. : Pour exprimer des relations entre métaclasses, MOF 1.4 propose le concept de méta-association qui est une association binaire entre deux métaclasses. Une métaassociation porte un nom, et chacune de ses extrémités peut porter un nom de rôle et une multiplicité -Package. Pour grouper entre eux les différents éléments d'un métamodèle, MOF 1.4 propose le concept de Package.
- **Un Package** : aussi appelé méta-package, est un espace de nommage servant à identifier par des noms les différents éléments qui le constituent.

La version MOF 2.0 est la version courante de MOF. Conceptuellement, son architecture ne diffère que très peu de celle de MOF 1. (Blanc et Salvator avril 2005). Techniquement, l'un des objectifs de MOF 2.0 est de capitaliser les points communs existants entre UML et MOF au niveau des diagrammes de classes et d'en expliciter les différences. Pour y parvenir, il a été convenu qu'un métamodèle instance de MOF 2.0 pouvait, ou non, contenir des méta-associations. Dans la version MOF 2.0, La Figure 5.5 affirme que le méta méta-modèle MOF2.0 est composé de deux parties : EMOF (Essential MOF) et CMOF (Complete MOF). EMOF est utilisé pour élaborer des méta-modèles sans association, tandis que CMOF est conçu pour les méta-modèles avec association. En outre, MOF2.0 intègre le méta-modèle Infrastructure de l'UML2.0, EMOF intègre le package Basic et CMOF le package Constructs. Notons que ces relations d'intégration sont assez complexes (Blanc et Salvator avril 2005).

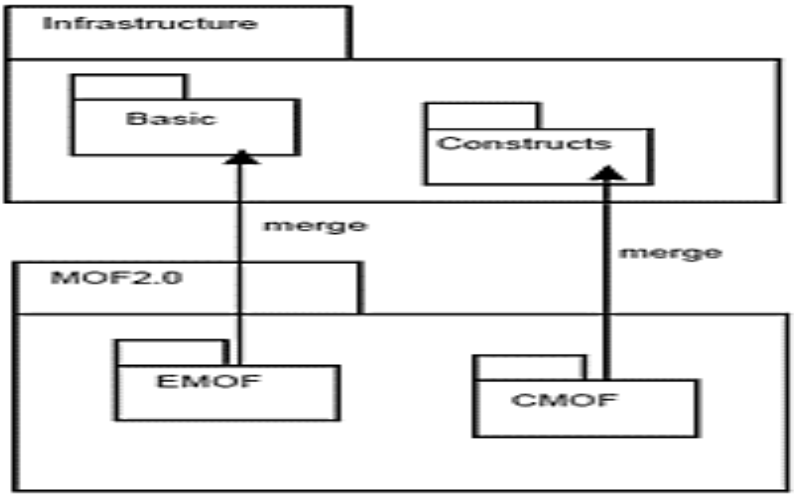


Figure 5. 5 : Représentation schématique du méta méta-modèle MOF2.0 **(Blanc et Salvator avril 2005)**

5.2.2.2 Le langage Ecore :

Le langage de métamodélisation Ecore est proposé par la fondation Eclipse dans le cadre du projet EMF[41] (Eclipse Modeling Framework). Ce framework permet de générer automatiquement des interfaces Java à partir de métamodèles Ecore. La particularité d'EMF est qu'il se fonde sur la version MOF 2.0 et non MOF 1.4. La figure 5.6 présente extrait simplifié du métamodèle Ecore.

EMF utilise le langage de métamodélisation Ecore[42] pour la définition et la vérification des métamodèles. Ce langage se concentre sur la spécification structurelle des métamodèles (Combemale 2008) . Il s'agit d'une implantation d'un sous ensemble du standard MOF (Meta Object Facility) de l'OMG[43].

Le méta-métamodèle Ecore ressemble fortement au méta-métamodèle EMOF, car il ne supporte que la notion de métaclasse sans méta-association. Pour exprimer une relation entre deux métaclasses, il faut utiliser des méta-attributs et les typer par des métaclasses. EMF impose cette contrainte pour faciliter la génération des interfaces Java. Le concept d'association n'existant pas en Java, il faudrait en effet une transcription particulière. Ainsi, les métamodèles conformes à Ecore sont composés d'EClass contenant des EAttribute et des EReference.

[41] https://www.eclipse.org/modeling/emf/
[42] ECore. The eclipse modeling framework project home page. http://www.eclipse.org/emf
[43] Object Management Group, Inc. Meta Object Facility (MOF) 2.0 Core Specification, January 2006. Final Adopted Specification.

Les principaux concepts d'Ecore sont les suivants[44] :

EObject : Elle représente n'importe quel élément, qu'il appartienne à un modèle ou à un métamodèle. Cette interface offre l'opération eClass(), qui permet d'obtenir l'EClass de l'élément et retourne un objet Java de type EClass. L'interface EObject offre aussi les opérations eGet() et eSet(), qui permettent respectivement de lire et d'écrire les valeurs des différentes propriétés de l'élément (attributs et références).

EClass : L'interface EClass représente une métaclasse d'un métamodèle. Cette interface offre les opérations getEAttributes() et getEReferences (), qui permettent d'obtenir la liste, respectivement, de tous les attributs et références contenus dans la métaclasse. Elle offre aussi l'opération getEStructuralFeature(), qui permet d'obtenir une propriété (attribut ou référence) d'une EClass à partir de son nom.

EPackage : L'interface EPackage représente le moyen d'accès à toutes les EClass définies dans un package. Elle offre l'opération getEClassifier(String name), qui permet de récupérer la référence vers une EClass d'un métamodèle à partir de son nom.

EFactory : Comprend une méthode « Create » pour chacune des classes du modèle d'entrée. Cela va permettre de créer des instances (des objets) des classes de l'application.

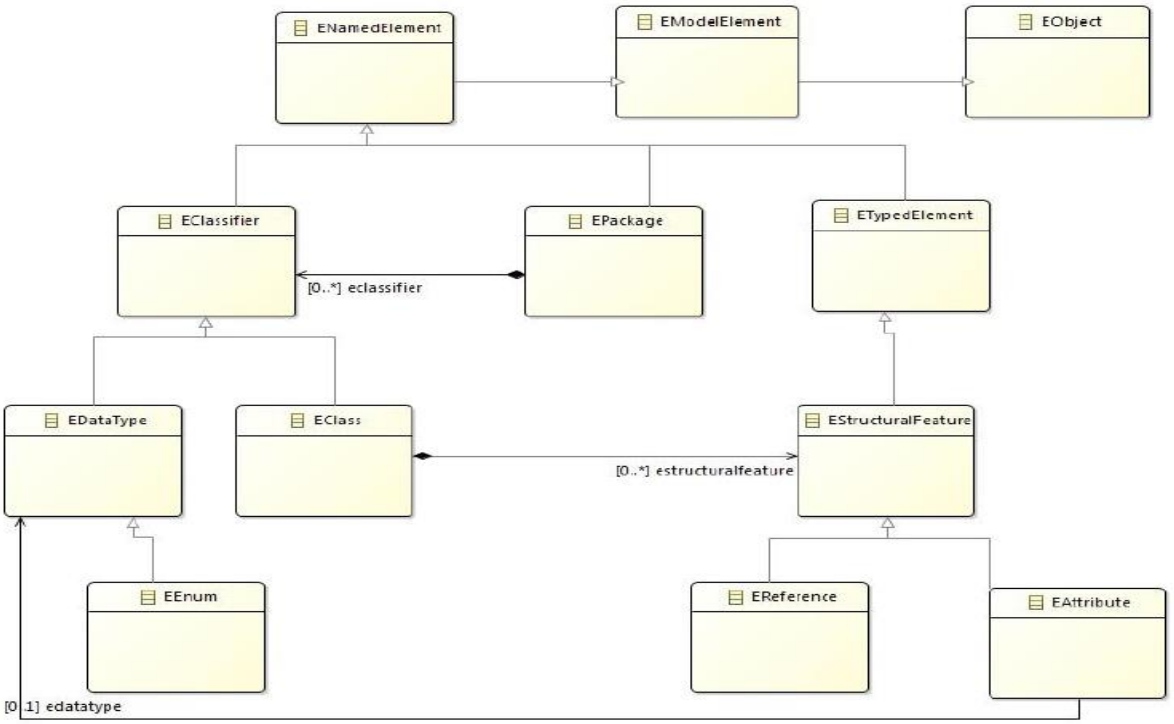


Figure 5. 6: Extrait simplifié du métamodèle Ecore **(Budinsky, et al.. (August 1, 2003))**

[44] https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html

Avec ces concepts, exprimés à un bon niveau d'abstraction, Ecore permet de définir des modèles et méta-modèles (uniquement la structure). Malgré le manque d'expressivité, de ce langage de méta-modélisation, plusieurs outils ont été développés en se basant sur ecore. On cite à titre d'exemples : l'outil TopCased (Farail 2012) (Farail , et al.. 2006)EMF, GMF (Graphical Modeling Framework), Visual Editor ou encore Tiger (Ehrig, et al.. November 7-11, 2005,). La plupart de ces outils se limitent à des fonctionnalités d'édition textuelle ou graphique et offrent parfois la possibilité de faire de la vérification.

Les langages de métamodélisation tel que Ecore ne permettent pas au concepteur d'un langage d'exprimer toutes les règles qu'il souhaite appliquer à son métamodèle. Pour exprimer ces règles, l'OMG définit le langage OCL (Object Constraint Language). Appliqué au niveau du métamodèle, il permet d'ajouter des propriétés qui n'ont pas pu être spécifiées par les concepts fournis par le méta-métamodèle. Il s'agit donc d'un moyen de préciser la sémantique du métamodèle en limitant les modèles conformes.

5.2.2.3 GOPRR (Mallouli 2014)

GOPRR (Graph, Objects, Property, Role and Relationship) est un langage de métamodélisation développé à partir d'OPRR (Object, Property, Relationship, Rôle), une forme de modification du modèle de données Entité-Relation (ER) (Smolander 1992) .C'est un langage qui n'est pas indépendant comme c'est le cas de MOF ou Ecore, mais il est plutôt proposé avec son outil MetaEdit+ qui le supporte. Comme le montre la Figure 5.7 (récupérée à partir de la thèse de Steven Kelly (Kelly, Towards a comprehensive MetaCASE and CAME environment: conceptual, architectural, functional and usability advances in MetaEdit+ 1997), ce méta-méta-modèle est structuré avec les concepts suivants : *Graphes, Objets, Propriétés, Relations n-aires et des Rôles*).

Les propriétés décrivent les objets, les relations, les graphiques et les rôles. Les rôles peuvent avoir des cardinalités et peuvent être joués par plusieurs objets. Un graphe est composé d'objets et de relations (avec des rôles spécifiques). En outre, les graphes peuvent partager des composants. Les objets peuvent avoir des graphes de décomposition et plusieurs graphes d'explosion. La décomposition permet de décrire plusieurs objets à la fois, alors qu'une explosion est un mécanisme qui est associée à un seul objet. La spécialisation et la généralisation entre les objets sont possibles via l'héritage simple. Les propriétés peuvent être contraintes par des règles qui peuvent être redéfinies dans le sous-type de propriétés.

Une représentation graphique des spécifications est associée au langage GOPRR à travers un éditeur graphique qui permet à l'ingénieur méthode de décrire chaque objet et ses attributs avec une forme. Cette forme est automatiquement utilisée lorsque les objets sont affichés.

Afin d'interpréter les spécifications graphiques d'un produit, il est possible, à travers l'outil de GOPRR (MetaEdit+) de définir des rapports personnalisables avec un langage de script (Merl).

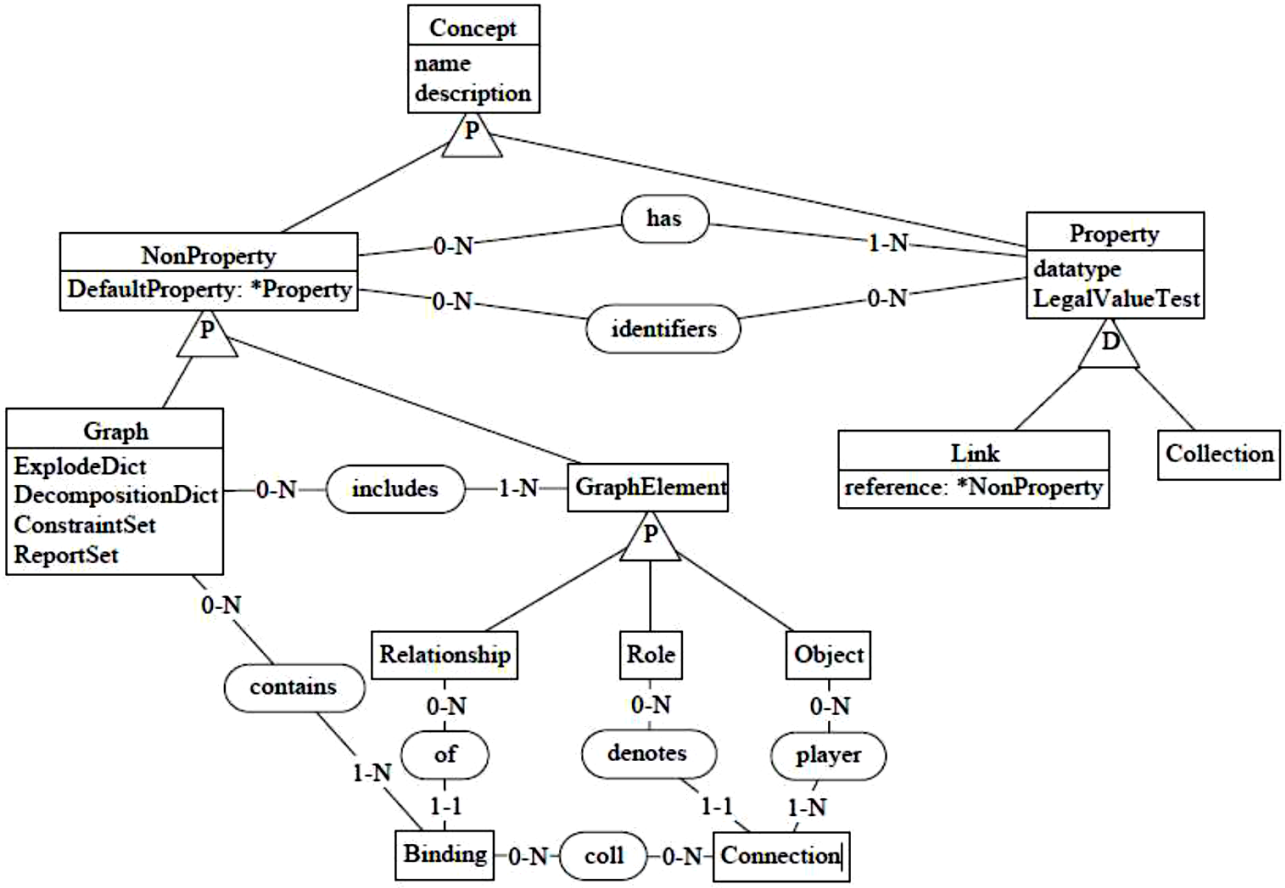


Figure 5. 7: Les concepts du langage GOPRR.

GOPRR est un langage simple, facile à utiliser et à mettre en œuvre puisqu'il est soutenu par un environnement méta-CASE. La manipulation de GOPRR à travers MetaEdit+ est partiellement guidée et offre ainsi de nombreux avantages. Tout d'abord, elle donne la possibilité au concepteur d'envisager le méta-méta-modèle. Deuxièmement, elle permet de valider et tester les méta-modèles avec des méthodes réelles, une procédure qui prend énormément de temps avec papier et crayon. Troisièmement, elle permet à un public beaucoup plus large (que scientifique) et plus représentatif pour évaluer les méta-modèles.

Concernant l'expression de la sémantique d'exécution, la sémantique des méta-modèles qui sont conformes à GOPRR est implicite et elle est directement exprimer dans les scripts en Merl.

5.2.2.4 OCL (Object Constraint Language)

Le langage OCL (Warmer November 28-29, 2002), ***Object Constraint Language***, a été défini par l'OMG pour permettre la modélisation du corps des opérations dans un modèle UML. Il est utilisé pour exprimer des pré et post-conditions sur les opérations. OCL dispose d'un formalisme textuel, on parle d'expression OCL. Les expressions OCL ne génèrent aucun effet de bord. C'est-à-dire que l'évaluation d'une expression OCL n'entraîne aucun changement d'état dans le modèle auquel elle est rattachée. La valeur retournée est vrai ou faux. L'évaluation de la contrainte OCL permet de la sorte de savoir si la contrainte est respectée ou non. Une expression OCL porte sur un élément du modèle. Pour être évaluée, une expression OCL doit être rattachée à un contexte, qui doit être directement relié à un modèle. L'exemple suivant permet de spécifier, pour un compte bancaire, que la valeur du solde doit être positive avant toute invocation de l'opération debit, c'est donc une pré-condition :

```
1 context CompteBancaire :: debit():Integer
2 pre: solde>0
```

Le métamodèle OCL permet de représenter n'importe quelle expression OCL sous forme de modèle. Pour des raisons de simplicité d'utilisation, OCL reste avant tout un langage textuel, ce qui rend son intégration aux modèles exprimés à l'aide de DSL textuels tout aussi simple. Jusqu'à sa version 1.4, UML était très critiqué parce qu'il ne permettait pas de spécifier des créations, des suppressions ou des modifications d'éléments de modèles. Ces actions ne pouvant pas non plus être spécifiées à l'aide du langage OCL, puisque celui-ci est sans effet de bord. Il était nécessaire de standardiser un nouveau langage. C'est ce qui a donné naissance au langage AS (Action Semantics).

5.2.2.5 Action Semantics : (GAOUAR 2019)

L'objectif de Action Semantics est de permettre de définir des actions. Une action au sens AS est une opération sur un modèle qui fait changer l'état du modèle. Grâce à ces actions, il est possible de modifier les valeurs des attributs, de créer ou de supprimer des objets, de créer de nouveaux liens entre les objets, etc. Ainsi le concept d'Action Semantics permet de spécifier pleinement le corps des opérations UML. Au départ AS était un langage développé à part entière hors de UML puis inclus dans la version 1.5. Dans UML 2.0, il est totalement intégré au métamodèle. AS n'est standardisé que sous forme de métamodèle, et aucune syntaxe concrète n'est définie, contrairement à OCL. Ainsi, le manque de format concret (textuel), rend l'utilisation de AS complexe. C'est la raison pour laquelle AS n'est pas encore pleinement exploité, contrairement

à OCL. Les modèles étant des entités abstraites, ils ne disposent pas intrinsèquement de représentation informatique.

5.2.2.6 Le standard XMI (XML Metadata Interchange)

OMG a proposé un standard XMI pour donner une représentation concrète des modèles sous forme de documents XML, et cela est le but principal de l'XMI. XMI fournit les balises nécessaires et suffisantes à la représentation des modèles en format XML en se basant sur la définition de la structure des balises : XML DTD et XML Schéma. L'OMG a donc décidé de standardiser XMI (XML Metadata Interchange), qui permet l'échange de modèles sérialisés en XML. XM) de se focaliser sur les échanges de métadonnées conformes au standard MOF. L'objectif de XM) est de permettre de sérialiser et d'échanger des métamodèles MOF et des modèles basés sur ces métamodèles sous forme de fichiers en utilisant des dialectes XML. Il permet aussi de sérialiser des métamodèles qui sont créés avec Ecore. Ceci est possible car XMI ne définit pas un dialecte XML unique, mais un ensemble de règles qui permettent de créer une DTD ou un schéma XML pour différents métamodèles. Ainsi, les métamodèles conformes à MOF ou à Ecore et leurs modèles respectifs peuvent être portables en utilisant XMI. Toutes nos expérimentations autour de la plateforme Eclipse ont utilisé ce format d'échange pour le traitement des modèles et des métamodèles.

5.2.3 Les langages de Modélisation:

Les langages de modélisation ou métamodèles sont situés au niveau M2 de l'architecture MDA et permettent de spécifier des modèles. Nous distinguons deux grandes catégories : les spécifiques ou *Domain-Specific Languages*, répondant aux exigences d'un domaine bien précis, et les génériques ou *General Purpose Modeling Languages* en anglais, comme UML, qui peuvent s'adapter à tous les domaines et problèmes de conception.

5.2.3.1 Les DSL

L'utilisation d'un DSL aide à se concentrer sur un sujet déterminé en fixant un cadre précis. Le même niveau d'expression et la compréhension d'un domaine sont possibles en utilisant un langage générique tel que UML, mais le niveau attendu de connaissances concernant le domaine est considérablement plus élevé avec une approche basée sur un DSL. La définition d'un DSL est une tâche dont la complexité est proportionnelle à celle du domaine cible.

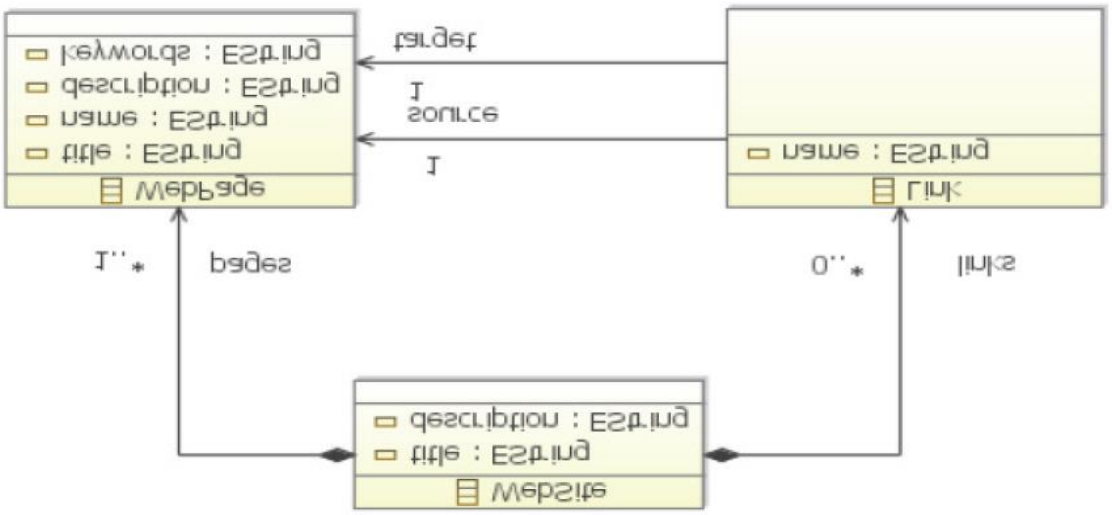


Figure 5. 8: Exemple d'un DSL pour la modélisation de sites Web.

La figure 5.8 présente Exemple d'un DSL pour la modélisation de sites Web. Elle illustre un métamodèle Ecore pour la modélisation de sites Internet ordinaires. Le point d'entrée de ce métamodèle est le métatype WebSite qui a deux attributs : un titre (title) et une description (description) qui sont tous les deux de type chaîne de caractères. Les deux losanges noirs sur le rectangle WebSite modélisent une relation de composition, c'est à-dire, une instance de WebSite peut contenir une ou plusieurs instances de WebPage et 0 ou plusieurs instances de Link. Le métatype WebPage représente une page Web qui a un titre (title), un nom (name), une description (description) et des mots clés (keywords). Le métatype Link quant à lui a un seul attribut (name). Une instance de type Link est liée à deux WebPage ; l'une représente la source du lien (flèche de référence « source »), l'autre sa cible (flèche de référence « target »). Au niveau de ce métamodèle, rien n'interdit qu'une instance de WebPage soit à la fois la cible et la source d'un lien. Si le concepteur du métamodèle souhaite ajouter une telle contrainte alors il faudra l'exprimer avec la règle OCL décrite ci-dessous. Contrainte. Une WebPage ne peut pas être à la fois la cible et la source d'un lien.

```
1 Context Link inv:
2 source <> target;
```

Une fois la conception du métamodèle terminée, le concepteur peut modéliser des sites Web en utilisant son métamodèle. Un éditeur par défaut est disponible dans *Eclipse Modeling Framework*. Même si cet éditeur permet de valider la cohérence du métamodèle, il reste néanmoins limité et ne propose pas de représentation graphique pour les modèles, d'où l'intérêt d'utiliser la composante *Graphical Modeling Framework* qui permet de créer des modeleurs visuels puissants et plus faciles à utiliser.

La fabrication d'un DSL est principalement axée sur deux phases : l'élaboration de la syntaxe abstraite et la définition de la syntaxe concrète.

- **Syntaxe abstraite**

Dans le contexte de l'IDM, la syntaxe abstraite est la base de tout langage de modélisation : il s'agit de l'ensemble de ses concepts et leurs relations. Les langages et environnements de métamodélisation tel que MOF, Ecore, Kermeta[45], Xtext [46]ou encore MetaEdit+[47]offrent les concepts et relations élémentaires avec lesquels il est possible de décrire un métamodèle représentant cette syntaxe abstraite. La majorité de ces langages reposent sur les mêmes constructions élémentaires. Consulter la figure 5.9.

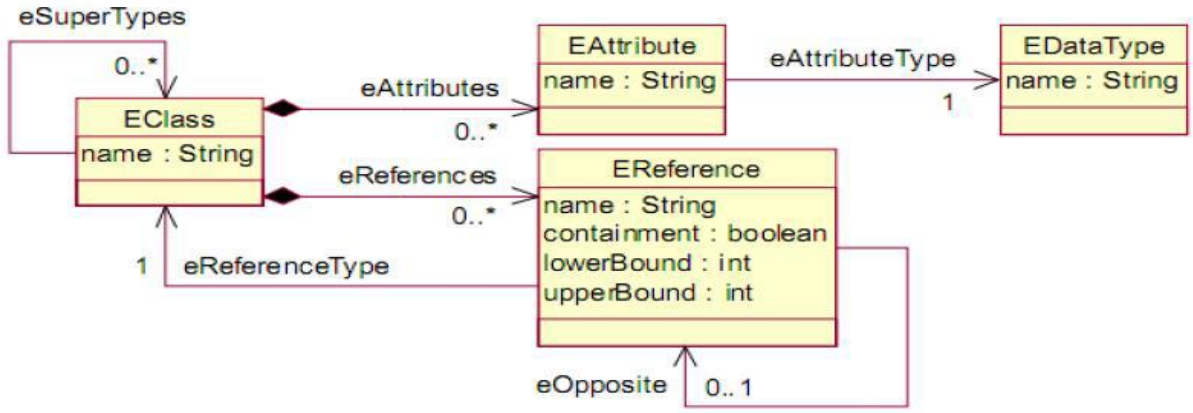


Figure 5. 9 : Concepts de base pour la métamodélisation (EMF/Ecore).

- **Syntaxe concrète**

Chaque élément d'un modèle (instance d'un métamodèle et donc du DSL) a une représentation/forme graphique ou textuelle particulière. La définition de la syntaxe concrète est donc la définition de cette représentation textuelle ou graphique. Il est envisageable de définir plusieurs syntaxes concrètes pour une même syntaxe abstraite et donc d'avoir plusieurs représentations d'un même modèle. Le langage peut alors être manipulé avec différents formalismes mais avec les mêmes constructions et la même représentation abstraite.

Le projet *Eclipse Modeling*[48] propose le projet *Graphical Modeling Framework* (GMF) pour définir des représentations graphiques. GMF permet de décrire la représentation graphique de chaque concept et construit un *Domain-Specific Modeler* (DSM). Les modeleurs ainsi produits possèdent une bonne ergonomie et une standardisation du format de stockage des informations

[45] http://www.kermeta.org
[46] http://www.eclipse.org/Xtext/
[47] http://www.metacase.com/fr/products.html
[48] http://www.eclipse.org/modeling/

graphiques. Il existe aussi d'autres projets qui permettent de définir des représentations textuelles. Nous citons par exemple Xtext et TCS[49].

5.2.3.2 Le métamodèle UML et les profils UML

La première version de UML a été publiée en 1997 par l'OMG. Depuis, UML est devenu la référence pour la création de modèles et le métamodèle d'UML est le plus connu de l'approche MDA. Même si nous n'utilisons pas le standard UML dans nos travaux, nous discuterons dans cette section du métamodèle UML, des profils UML ainsi que de la position de chacun dans le processus de développement MDA.

Le métamodèle UML

1. Nous n'allons pas détailler ici le métamodèle d'UML, il existe aujourd'hui une multitude de documents pour cela. Une description plus détaillée est présentée dans le livre «Mda en action : Ingénerie logicielle guidée par les modèles» (Blanc et Salvator avril 2005). Les auteurs affirment que le métamodèle UML définit la structure que doit avoir tout modèle UML. Ils expriment, par exemple, qu'une classe UML peut avoir des attributs et des opérations. La figure 5.10 illustre un fragment du métamodèle UML et précise au niveau métamodèle par exemple qu'une association UML est composée de deux extrémités, chacune représentant une classe.

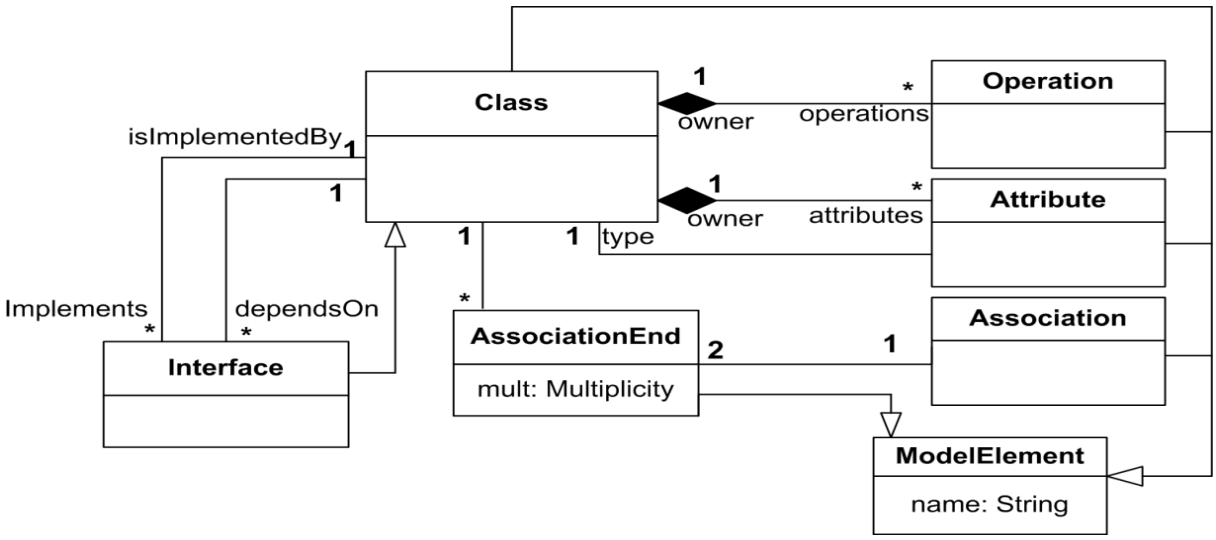


Figure 5. 10 : Un fragment du métamodèle du langage UML.

[49] Textual Concrete Syntax: http://www.eclipse.org/gmt/tcs/

Le langage UML propose 13 diagrammes dans sa version 2.0

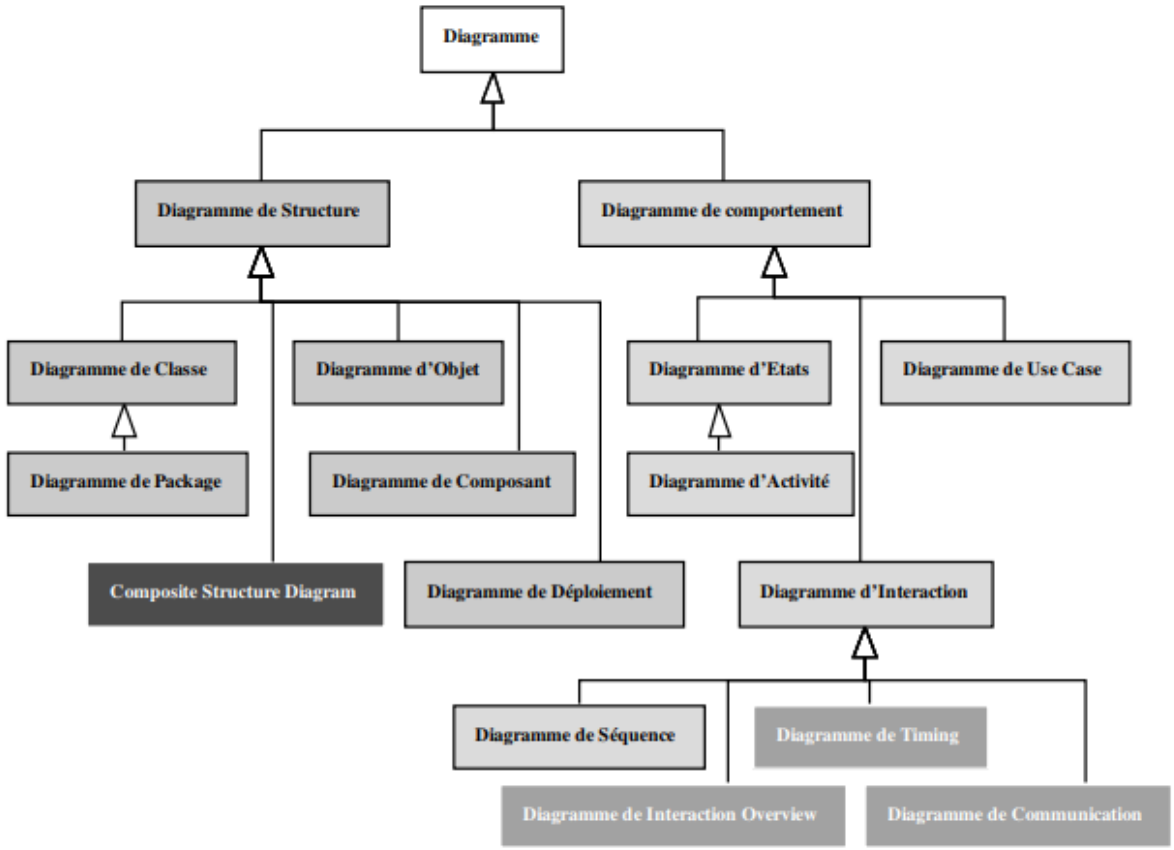


Figure 5. 11 : Classification des types de diagrammes UML.

La figure 5.11 illustre ces 13 diagrammes. Certains d'entre eux permettent de représenter la structure d'un système (les aspects statiques), d'autres représentent son comportement (aspects dynamiques). La notion d'héritage entre certains diagrammes (comme entre le diagramme d'activités et le diagramme d'états) exprime le fait que la syntaxe (méta-modèle et notation) de ces diagrammes s'appuie sur celle d'un autre. Les diagrammes représentés par des rectangles foncés dans la figure 5.11 font référence aux nouveaux diagrammes proposés dans UML 2 (le diagramme de communication d'UML 2 fait référence au diagramme de collaboration d'UML 1.X). Les diagrammes de timing, de communication, de séquence et d'interaction overview sont regroupés sous l'appellation de diagrammes d'interaction. Ces diagrammes permettent de décrire différentes perspectives d'un système mais ils peuvent également être utilisés pour modéliser une même vue à des niveaux d'abstraction variés. Ceci est possible de deux manières : en décrivant, dans un même modèle, un diagramme pour chaque niveau de détail, ou en créant plusieurs modèles à des niveaux d'abstraction différents et en « traçant » les diagrammes d'un modèle à un autre. (Laforcade 2004).

Les profils UML

La définition officielle d'un profil selon la dernière spécification d'UML (1.5) en vigueur est la suivante : « A profile is a stereotyped package that contains model elements that have been customized for a specific domain or purpose by extending the metamodel using stereotypes, tagged definitions, and constraints. A profile may specify model libraries on which it depends and the metamodel subset that it extends. » (Laforcade 2004)

UML est un langage général, et a donc été conçu pour prendre en charge une grande variété d'applications et de contextes de mise en œuvre. Cependant, même avec cette intention d'être général, UML ne peut pas couvrir tous les contextes et offre ainsi un mécanisme d'extensibilité basé sur les profils. Un profil permet la personnalisation d'UML pour prendre en charge des domaines spécifiques qui ne peuvent pas être représentés avec UML dans son état original. Le profil est constitué de trois concepts : les stéréotypes, les contraintes et les valeurs marquées.

Ces éléments permettent d'adapter la sémantique sans changer le métamodèle d'UML.

D'un point de vue technique, un profil est un package stéréotypé spécifique à un domaine. Un stéréotype est défini comme une extension d'une métaclasse UML, il peut avoir des propriétés et/ou des opérations particulières. Ainsi, un profil est une spécification qui spécialise un ou plusieurs métamodèles standards appelés les métamodèles de référence.

Dans la thèse de doctorat Lafocarde (Laforcade 2004), ce dernier présente une synthèse des différents éléments structurant un profil UML à savoir :

- **les éléments sélectionnés du métamodèle de référence :** Un profil fournit la sélection du métamodèle de référence qui constitue la focalisation particulière choisie. Cette sélection n'exclut pas les autres éléments du métamodèle de référence, mais simplement spécifie ceux qui sont spécialisés.

- **les stéréotypes :** Un stéréotype est défini pour une métaclasse spécifique du métamodèle de référence. Dans un profil UML, le stéréotype crée une métaclasse UML virtuelle basée sur la métaclasse UML existante et peut également aussi spécifier des contraintes additionnelles ou des valeurs marquées requises.

- **les définitions de valeurs marquées :** La définition d'une valeur marquée contient le nom des valeurs marquées correspondantes, le type des valeurs qu'elles peuvent prendre, la description de oratla sémantique et des contraintes s'appliquant à chaque valeur marquée correspondante. La valeur marquée agit comme un attribut d'une métaclasse UML, permettant ainsi l'attachement

arbitraire d'informations à une instance. Un ensemble de valeurs marquées peut être associé à un stéréotype afin d'être appliqué aux éléments de modélisation portant ce stéréotype.

- **les contraintes :** Elles peuvent être définies au niveau d'une métaclasse particulière comme au niveau d'un stéréotype particulier. Elles permettent de spécialiser davantage la sémantique des éléments du métamodèle de référence utilisés dans le profil. Cette spécification est écrite sous la forme d'une expression dans un langage de contrainte particulier. Le langage de contrainte formel utilisé par le métamodèle UML est le *Object Constraint Language* (OCL). Mais les contraintes peuvent être spécifiées de manière informelle en langage naturel.

- **les descriptions :** Il est possible de préciser la sémantique d'un profil (comme des éléments qu'il contient) par des descriptions en langage naturel. Par exemple, les objectifs d'un profil ou sa compatibilité avec d'autres profils peuvent ainsi être décrits en détail.

- **la notation :** La notation d'UML peut être personnalisée par le mécanisme des profils : définition d'icônes associés aux stéréotypes, disposition pour les diagrammes, etc.

-**les règles.** Les profils doivent être capables de définir des règles dédiées à leur domaine spécifique. Elles peuvent être de différents types :

- **Règles de transformation :** pour exprimer comment un modèle peut être transformé pour être modélisé ou implémenté dans un but spécifique. Par exemple, le profil CORBA exprime comment un modèle UML peut être transformé en une implémentation CORBA.
- **Règles de validation :** pour vérifier que le modèle possède les bonnes propriétés du domaine du profil. Ces règles vérifient les critères de cohérence sur le modèle.
- **Règles de présentation :** pour définir quels types d'éléments de modélisation doivent apparaître dans tel type de diagramme et indiquer aussi quelles informations doivent être cachées.

5.2.4 TGG Interpreter

L’approche des grammaires de graphes triples (Triple Graph Grammars : TGG), introduite par Andy (Schürr , Specification of graph translators with triple graph grammars 1995) est une approche de transformation de modèles, qui se réfère à un une variante de grammaires graphiques dédiée, principalement aux transformations bidirectionnelles de modèle à modèle, et elle a un langage déclaratif similaire au langage QVT-Relations, qui est un langage déclaratif QVT (Query/View/Transformation) (Greenyer et Kindler 2010).

Le TGG définit trois langages de transformation de modèles pour la source, la cible et la correspondance opérant sur des modèles, qui sont conformes à leurs méta-modèles Ecore. La

norme TGG intègre l'OCL (Object Constraint Language) pour fournir des expressions de contraintes sur tout modèle ou méta-modèle MOF/Ecore (OMG (Object Management Group), 2006). Une transformation TGG incarne une relation de cohérence sur des ensembles de modèles. Elle peut être utilisée pour générer un des modèles afin que l'ensemble des modèles soit cohérent.

La spécificité des grammaires à triple graphe est que leurs règles de production consistent en trois sous-graphes, dont deux représentent les deux modèles/graphes impliqués (graphe source et graphe cible). Le troisième sous-graphe, appelé graphe de correspondance, relie les parties de graphe apparentées du graphe source et du graphe cible et se situe donc entre les deux autres graphes.

L'avantage principal des TGG est qu'ils permettent de définir des transformations d'une façon déclarative, avec une exécution dans les deux directions de la transformation. Cette définition de relation entre les différents modèles a une bonne implication, dans le sens ou, premièrement, les règles de TGGs " TGG-rules" peuvent ˆêtre rendues opérationnelles dans les scénarios d'application (modèle de transformation, modèle d'intégration et modèle de synchronisation), et deuxièmement, le processus de transformation est possible dans les deux directions, (i.e., la définition est donc bi-directionnelle). En effet, les règles de TGGs participent à l'intégration du modèle et au scénario de sa synchronisation (Kindler et Wagne 2007). De plus, la définition locale de la relation entre les deux modèles est fortement similaire au style de la sémantique Opérationnelle Structurelles (SOS) (Plotkin 2004), ce qui permet de vérifier la correction s´sémantique des relations entre ces modèles.

Parmi les inconvénients de la méthode de transformation avec les TGGs, est que toutes les contraintes sont ajoutées au moment de l'application des règles et pendant la construction des modèles. Par contre, ces contraintes ne peuvent être évaluées seulement qu'après la construction du modèle complet.

EMF Henshin (Arendt, et al.. October 3-8, 2010) est considéré comme la continuation du langage de transformation EMF (Biermann, et al.. 2006). C'est un langage de modèle de transformation qui utilise Triple graph grammars (TGG), basé sur le Framework Eclipse Modeling Framework EMF (Steinberg, Budinsky, et al.., EMF: Eclipse Modeling Framework 2009).

TGG Interpreter est considéré comme le meilleur choix pour les transformations de graphes et les transformations de modèles bidirectionnelles. Il s'agit d'une technique utilisée pour définir la relation entre deux types de modèles. Et par la suite, transformer un modèle d'un type à un autre, afin de calculer la correspondance entre deux modèles existants. Ainsi, pour le but de maintenir la cohérence entre deux types de modèles, tel qu'il est définit en TGG. Lorsqu'un des modèles est

changé, le modèle cible peut être modifié, ce qui signifie que la transformation ou la synchronisation peut être appliquée progressivement. La figure 5.12 Illustre l'architecture de TGG Interpreter. Pour plus de caractéristiques du TGG, consulter MDETOOLS [50] .

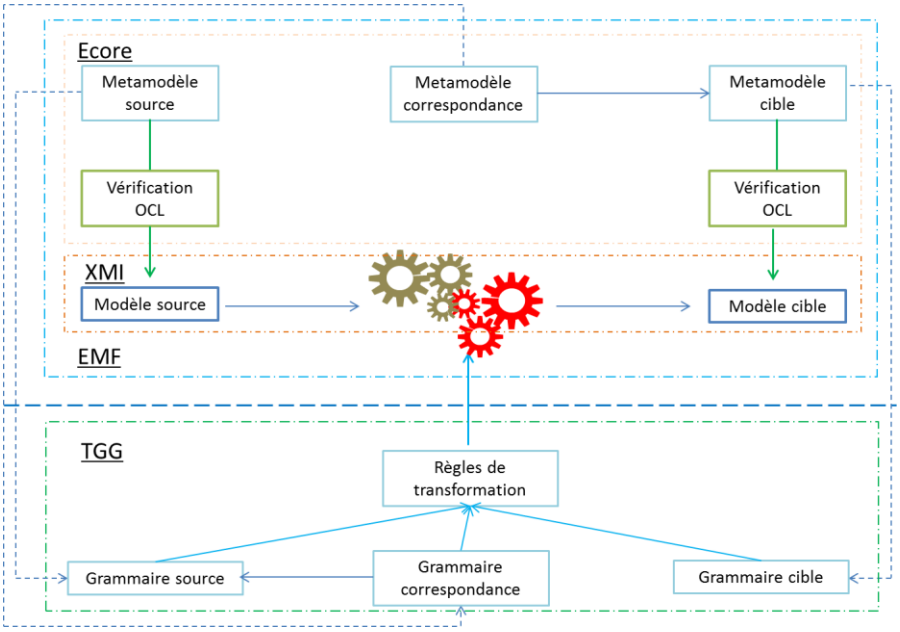


Figure 5. 12 : L'architecture de TGG Interpreter

La transformation de modèles se base sur la définition des règles de transformation où une bonne transformation de modèles dépend d'une bonne définition des règles de transformation. TGG se compose de trois graphes ou chaque graphe appartient à un domaine. Chaque domaine contient une règle de grammaire de graphe. Habituellement, le domaine sur la gauche est appelé source et le domaine sur la droite est appelé cible. Le domaine de correspondance relie le domaine source et cible. Chaque domaine est attribué à un métamodèle. Les nœuds de ce domaine sont les classes de ce métamodèle (on peut considérer le domaine comme une grammaire du métamodèle).

La description de transformation est un modèle de transformation qui se base sur le mécanisme de left-hand-side graph, un right-hand-side et une liste de correspondance (Mapping). Les nœuds du graphe sont des instances des éléments du modèle de méta-model source et le méta-model cible, respectivement. Il est ainsi possible de créer des transformations d'ordre-élevé (higher -order) avec EMF Henshin. Par contre, il n'existe pas de support (no built-in support) pour créer les traces, pour le modèle de transformation incrémental, ou même le multiple-directionel (Menad 2015).

[50] http://www.mdetools.com/detail.php?toolId=39

Avant de spécifier les règles, on doit définir les différents nœuds qu'on va rencontrer dans les règles. Il existe différents nœuds de règle de transformation : nœud de contexte, nœud de production, les contraintes et les nœuds réutilisables.

- **Nœud de contexte**

Parfois, seule une partie d'un modèle est pertinente et doit être transformée. Ensuite, les règles de TGG devraient être conçues pour les parties pertinentes de la transformation. Si les pièces d'un modèle moins pertinents influencent la transformation, ils peuvent apparaître comme des nœuds de contexte dans les règles de TGG. La présentation graphique d'un nœud de contexte est présentée d'une case avec un contour noire.

Les nœuds de contexte doivent être appariés avec les objets du modèle précédemment traités. Cela signifie que le TGG, tel qu'il est présenté à ce jour, besoin de spécifier une grammaire complète pour toutes les parties de modèle. En revanche, TGG permet également de formuler une grammaire partielle sur un modèle. Là, les nœuds de contexte ne doivent pas nécessairement être préalablement traités par une autre règle (Greenyer 2006).

- **Nœud de production**

Les nœuds de production sont affichés sous forme de boîtes vertes avec une bordure verte et un label "++". Les arrêtes de production sont affichés sous forme de flèches de couleur vert foncé avec une étiquette "++". Les nœuds de production ne doivent pas correspondre à nœud qui existe déjà. Si on trouve ces éléments dans le domaine source, les éléments de modèle cible et de modèle de correspondance peuvent être créés selon la règle. Tous les objets créés sont liés aux nœuds de contexte de la règle. En conséquence, un objet de modèle ne peut être présenté comme un nœud de production qu'une seule fois. Nous appelons cela la sémantique bind-only-once pour les nœuds de productions (Greenyer et Rieke, Applying advanced tgg concepts for a complex transformation of sequence diagram specifications to timed game automata 2012).

- **Nœud réutilisable**

A noter que les types de composants ne doivent pas être générés, mais peuvent être réutilisés encore et encore depuis des nœuds déjà existants avec les propriétés requises. Par conséquent, nous appelons ces nœuds "les nœuds réutilisables". Graphiquement, ces nœuds sont représentés en gris. La sémantique des nœuds réutilisables est qu'ils peuvent être nouvellement générées ou réutilisés de façon arbitraire. La couleur grise reflète le fait que, de la règle de TGG, à savoir qu'il est réutilisé) ou en vert (un nœud du côté droit de la règle de TGG, à savoir qu'il est nouvellement généré), qui peut être choisi à chaque fois que la règle est appliquée. Cette interprétation montre

en fait que, les nœuds réutilisables ne sont pas strictement nécessaires. Nous pourrions remplacer par un nombre exponentiel de règles de TGG. Mais, le concept de nœuds réutilisables permet de réduire le nombre et la complexité des règles de TGG. En plus si l'exemple est complexe, les règles sans nœuds réutilisables peuvent générer un désordre. Les nœuds réutilisables nous permettre d'avoir plus de règles simples et de concentrer sur l'essentiel des modèles pertinents. (Kindler et Wagne 2007).

Les contraintes

La contrainte réelle dans un nœud de contrainte peut être toute expression OCL (OMG (Object Management Group)2006) qui se réfère à des objets qu'il est attaché. Les restrictions sont seulement nécessaires pour les mettre en œuvre des transformations ou des synchronisations plus efficace. Une contrainte OCL représentée avec la couleur jaune et doit être attachée avec un autre nœud.

5.2.5 Le langage de génération de code Xpand (Efftinge, et al.. 2004 - 2014)

Dans cette section nous exposerons l'outil Xpand (Xpand) du projet oAW (oAW) intégré dans le framework de modélisation d'Eclipse EMF.

L'outil openArchitectureWare (oAW) est ouvert, puisqu'il se permet de s'intégrer avec d'autres outils comme ATL, TIGER, etc. et notamment EMF et TGG-Interpreter. Dans notre approche. Nous allons adopter Xpand pour générer les spécifications OWL2 à partir des modèles (Model 2 Text).

Xpand est qualifié par la « réutilisation », cet outil offre la capacité de structuration des transformations comme des briques réutilisables dont on peut intégrer au sein d'un processus unique (*Workflow*), permettant la configuration et réutilisation des transformations.

OAW fournit encore une série de langages de programmation déclaratifs manipulant des modèles à travers les API d'EMF. Dans notre approche, nous optons pour le langage d'écriture de templates Xpand (Klatt 2007) dont le texte généré est une spécification OWL2.

Ce *Framework* de génération de code offre la possibilité d'utiliser des langages textuels, qui sont employés à leur tour dans différents contextes pour l'approche IDM (e.g. génération du code, extensions des méta-modèles, validation, transformation des modèles, etc.

Xpand correspond à DSL pour la génération du code à base de Templates. Il dispose, dans ce contexte, d'un petit vocabulaire effectif offrant la capacité d'utiliser les fonctions implémentées

en langage Xtend (un autre langage de programmation déclaratif de la plateforme oAW) et le langage Check qui sert à définir des contraintes assurant la validité des modèles.

Le projet Xpand contient essentiellement trois packages et un modèle chek dont l'extension « .chk » pour définir les contraintes. La figure 5.131illustre la structure d'un projet Xpand.

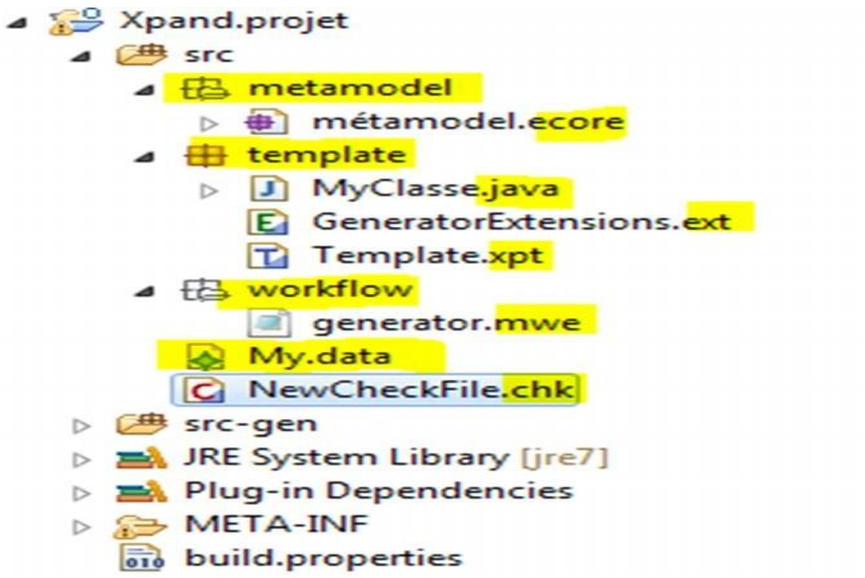


Figure 5. 13: Structure d'un projet Xpand

- Le package méta model : contient un méta modèle de sortie (ex le méta modèle d'un langage).
- Le package Template : contient un fichier java qui a l'extension « .java », un fichier Xpand qui a l'extension « .xpt » et un fichier xtend qui a l'extension « .ext ».
- Le package Workflow : dont l'extension « .mwe », contient le workflow qui permet d'appliquer le Template sur un modèle pour engendrer un ou plusieurs fichiers textes

5.2.5.1. Structure générale d'un template Xpand (Klatt 2007)

Le template Xpand (Klatt, 2007) permet le contrôle de la génération de code correspondant à un modèle. Le modèle doit être conforme à un méta-modèle donné. Le template est stocké dans un fichier ayant l'extension «. xpt ».

Un fichier template se compose d'une ou plusieurs instructions **IMPORT** afin d'importer les méta-modèles, de zéro ou plusieurs **EXTENSION** avec le langage Xtend et d'un ou plusieurs blocks **DEFINE**.

Exemple:

«**IMPORT** meta::model»

«**EXTENSION** my::ExtensionFile»

```
«DEFINE javaClass FOR Entity»
«FILE fileName()»
package
«javaPackage()»; public
class «name» {
// implementation
}
«ENDFILE»
«ENDDEFINE»
```

Le template de cet exemple importe la définition du méta-modèle, charge une extension Xtend et définit un template pour des simples classes java appliquées à un élément de la méta-classe Entity. fileName est une fonction Xtend appelée pour retourner le nom du fichier de sortie. Nameest un méta-attribut de la méta-classe Entity.

- **Le bloc DEFINE** (Klatt 2007)

Les blocs DEFINE, aussi appelés templates, constituent le concept central du langage Xpand. C'est la plus petite unité du fichier template. La balise DEFINE se compose d'un nom, une liste optionnelle de paramètres et du nom de la méta-classe pour laquelle le template est défini. Les templates peuvent être polymorphes, ils ont le format suivant :

```
«DEFINE templateName(formalParameterList) FOR
MetaClass»a sequence of statements
«ENDDEFINE»
```

- **L'instruction FILE** (Klatt 2007)

L'instruction FILE redirige la sortie produite, à partir des instructions de son corps, vers la cible spécifiée. La cible est un fichier dont le nom est spécifié par expression. Le format de l'instruction FILE se présente comme suit :

```
«FILE expression »
a sequence of statements
«ENDFILE»
```

- **L'instruction EXPAND** (Klatt 2007)

L'instruction EXPAND appelle un bloc DEFINE et insère sa production "*output*" à son emplacement. Il s'agit d'un concept similaire à un appel de sous-routine (méthode). Le format

de l'instruction EXPAND se présente comme suit :

«**EXPAND** definitionName [(parameterList)]

[**FOR** expression | **FOREACH** expression [**SEPARATOR** expression]]»

definitionName est le nom du bloc DEFINE appelé. Si FOR ou FOREACH est omise l'autre template est appelé pour l'instance courante *this*. Si FOR est spécifié, la définition est exécutée pour le résultat d'une expression cible. Si FOREACH est spécifiée, l'expression cible doit être évaluée à un type collection. Dans ce cas, la définition spécifiée est exécutée pour chaque élément de cette collection. Il est possible de spécifier un séparateur pour les éléments générés de la collection.

Exemple:

«**DEFINE** javaClass **FOR** Entity»

..

«**EXPAND** methodSignature **FOR this**.methods»

..

«**ENDDEFINE**»

«**DEFINE** methodSignature **FOR** Method»

...

«**ENDDEFINE**»

5.2.6 Langage XTEND[51]

C'est un langage de programmation généraliste compilé vers du Java qui permet notamment la redéfinition d'opérateurs que Java ne permet pas. Inspiré des règles ATL, les règles YAMTL sont composées de deux mêmes parties. Une première déclarant une liste de modèles d'entrée ainsi que des filtres facultatifs. Une seconde partie contient les modèles de sortie ainsi que les expressions servant à calculer les valeurs des propriétés des éléments de ces modèles.

Comme les expressions sous-language (*sublanguage*) qui résument la syntaxe des expressions pour tous les autres langages textuels livrés avec le framework Xpand il existe un autre langage appelé Xten (Xtend User Guide September 10, 2014) .Ce langage offre la possibilité de définir des bibliothèques riches d'opérations indépendantes et des extensions de métamodèle non invasives basées sur des méthodes Java ou des expressions Xtend (Xtend User Guide 2014). Ces

[51]https://www.eclipse.org/xtend/

bibliothèques peuvent être référencées à partir de tous les autres langages textuels basés sur le cadre des expressions.

5.2.7 Le langage Check pour la vérification de contraintes

La plateforme (OAW) Open Architecture Ware fournit aussi un langage formel appelé Chek (Xpand) pour la spécification des contraintes que le modèle doit respecter pour être correct est basé sur OCL.

Les contraintes spécifiées avec ce langage doivent être stockées dans des fichiers avec l'extension «. Chk ». Ce fichier doit commencer par une importation du méta-modèle selon le format « import metamodel ; ».

Chaque contrainte est spécifiée dans un contexte, soit une méta-classe du méta-modèle importé, pour lequel la contrainte s'applique.

Les contraintes peuvent être de deux types :

- Warning : dans ce cas, si la contrainte n'est pas vérifiée un message sera affiché sans que l'exécution s'arrête.
- Error : dans ce cas, si la contrainte n'est pas vérifiée un message sera affiché et l'exécution sera arrêtée.

5.2.8 MWE (Modeling Workflow Engine)

MWE, est une composante de EMFT (EMF Technology), est un moteur de génération déclaratif configurable qui s'exécute sous Eclipse. Il fournit un langage de configuration simple, basé sur les balises XML. Un générateur workflow est composé d'un ensemble de composants qui sont exécutées séquentiellement dans une JVM (Java Virtual Machine) unique.

5.3 L'approche proposée

Dans cette approche (Boudia et Bourahla 2022) nous proposons un outil de modélisation et de formalisation d'ontologies. Un éditeur graphique est développé pour modéliser la conceptualisation des ontologies par rapport au méta-modèle défini avec le standard Ecore, en utilisant les notations UML. Ecore (Steinberg , Budinsky , et al.., EMF: Eclipse Modeling Framework 2nd Revised Edition 2008) (Steinberg, Budinsky, et al.., EMF: Eclipse Modeling Framework 2009) est le méta-modèle central d'EMF. Il permet d'exprimer d'autres modèles en s'appuyant sur ses constructions. Ecore est défini en fonction de lui-même (son propre méta-modèle). Le résultat de la modélisation sera transformé, en utilisant des grammaires de graphes triples (Anjorin, Leblebici et Schürr 2015) en un modèle d'ontologie par rapport à un méta-modèle

d'ontologie OWL2 défini développé avec Ecore. Cette transformation bidirectionnelle peut être réalisée en exécutant un ensemble de règles définies par rapport à un méta-modèle de correspondance entre le méta-modèle ontologique-conceptuel et le méta-modèle ontologique basé sur OWL2.

Le modèle d'ontologie généré basé sur OWL2 représente le résultat de la conceptualisation de l'ontologie et il peut être formalisé en utilisant un langage de description basé sur la syntaxe RDF/XML avec un format spécifique comme format fonctionnel. Dans ce travail, la transformation du modèle d'ontologie en texte (code) est utilisée pour générer cette description formelle, qui est développée en utilisant la technique Xpand (Efftinge, et al.. 2004 - 2014). Compte tenu du nombre de personnes familiarisées avec les notations UML, cette solution sera une bonne approche pour la modélisation de la conceptualisation des ontologies pour les développeurs ordinaires.

L'objectif est de générer du code OWL2 à partir de modèles conceptuels d'ontologies de domaine spécifiques. Pour cela, nous utilisons des transformations combinées modèle-modèle et modèle-texte (code). L'ingénierie dirigée par les modèles (MDE), l'architecture dirigée par les modèles (MDA) et les tests basés sur les modèles (MBT) sont des approches dirigées par les modèles pour l'analyse et la transformation des modèles, qui ont été adoptées par divers domaines (par exemple, l'ingénierie d'entreprise et l'ingénierie logicielle).

L'abstraction d'un système centré sur un point de vue spécifique est décrite comme un modèle avec un langage de modélisation pour permettre de mieux le comprendre et de raisonner à son sujet. Les modèles ont été divisés en différentes catégories et ils ont différentes formes. Il y a des modèles mathématiques, des modèles de graphes, des modèles de textes, etc. Dans le contexte de MDA (Mellor, et al.. March 2004), les modèles sont classés en quatre niveaux d'abstraction : méta-méta-modèle, méta-modèle, modèle et système (sujet). Selon la précision, les modèles sont divisés en trois niveaux, à savoir : les modèles conceptuels, les modèles de spécification et les modèles de mise en œuvre (Fowler , Scott et Booch 1999).

Comme un modèle ne capture qu'une partie du système complet (sujet), plusieurs modèles pourraient être construits pour représenter un système, un modèle pour chaque point de vue du système. Dans ce cas, nous faisons de la transformation de modèle, qui est un autre pilote des approches dirigées par le modèle. La transformation de modèle est un processus (programme), qui contient une séquence d'activités opérant sur les modèles. Le but de la transformation de modèle est de générer des modèles cibles basés sur des modèles sources (Kleppe , Warmer et Bast April 21, 2003).

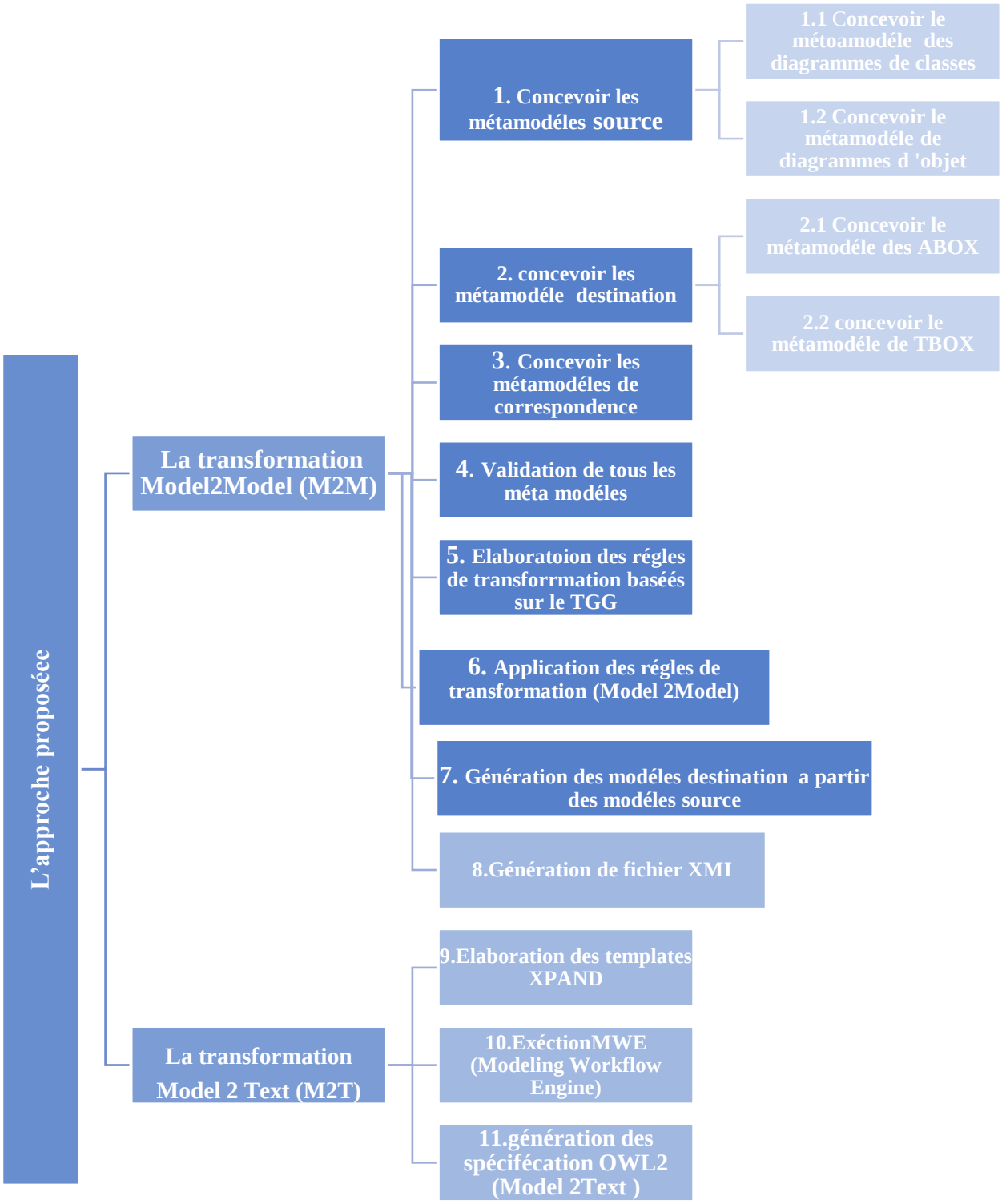


Figure 5. 14 : Les étapes de l'approche proposée

Selon le contenu expédié dans les modèles, les transformations de modèles peuvent être divisées en deux classes : modèle-à-modèle et modèle-à-texte (code). Il existe deux types de transformations (Mens, 2006). Le premier type de transformation est la transformation horizontale, qui est une transformation où les modèles source et cible résident au même niveau d'abstraction.

La transformation horizontale est divisée en deux types : les transformations endogènes et exogènes. Les transformations endogènes sont des transformations entre des modèles exprimés dans le même langage de modélisation, par exemple, le refactoring. Les transformations exogènes sont des transformations entre des modèles exprimés dans des langages de modélisation différents, par exemple, la migration.

Le deuxième type de transformation est la transformation verticale, qui est une transformation où les modèles source et cible résident à des niveaux d'abstraction différents. Le raffinement est un exemple typique, où une spécification initiale peut être progressivement raffinée par rapport à des règles de raffinement qui ajoutent des détails plus concrets aux raffinements successifs jusqu'à arriver à une implémentation complète. Les méthodes de transformation de modèles sont définies en s'appuyant sur certaines technologies de transformation de modèles, telles que QVT (QVT/ (OMG 2008).

Un méta-modèle est un modèle qui définit les règles de modélisation (syntaxe, sémantique et contraintes). C'est un modèle d'un langage de modélisation. Les méta-modèles sont généralement décrits à l'aide de diagrammes de classe UML (OMG, 2005). Il est construit en exploitant les langages de méta-modélisation comme Ecore (Steinberg, Budinsky, et al.., EMF: Eclipse Modeling Framework 2009), qui est un langage de méta-modélisation mature et largement utilisé, et c'est une version légère de MOF (Meta Object Facility) (OMG, Meta Object Facility (MOF) 2.0 Core Specification, Specification Version 2.0, 2003). Le méta-modèle Ecore est défini pour le cadre de modélisation Eclipse (EMF (Budinsky, et al.. (August 1, 2003)), où les éléments du modèle peuvent être représentés en utilisant XMI, qui signifie XML Metadata Interchange (OMG, MOF 2 XMI Mapping Specification, Specification, Version 2.4.1, Object Management Group. 2013)

Pour la génération automatique de descriptions formelles de modèles conceptuels spécifiques à une ontologie, nous avons développé deux processus de transformation de modèles.

- Le premier processus est la transformation de modèle à modèle, qui est une transformation horizontale et endogène, où la méthode TGG est utilisée pour la transformation du modèle. Pour cela, nous définissons trois méta-modèles de base pour les modèles conceptuels spécifiques à l'ontologie (les modèles sources) : les modèles OWL2 (les modèles cibles) et les modèles de correspondance pour le mappage des éléments des modèles sources et cibles.
- Le deuxième processus est la transformation de modèle en texte (code), qui est une transformation verticale et exogène, où la méthode Xpand (Xpand Documentation, 2020), qui est un langage spécialisé dans la génération de code basé sur des modèles EMF est utilisée pour la

transformation de modèle. Pour cela, nous utilisons les modèles OWL2 produits par le premier processus par rapport à son méta-modèle Ecore comme source de cette transformation de modèle. La cible est le code OWL2 (texte) par rapport aux modèles Xpand (représentant son méta-modèle Xpand). Les figures 5.14 et 5.15 suivent respectivement les étapes et le processus de l'approche proposée (Boudia et Bourahla 2022).

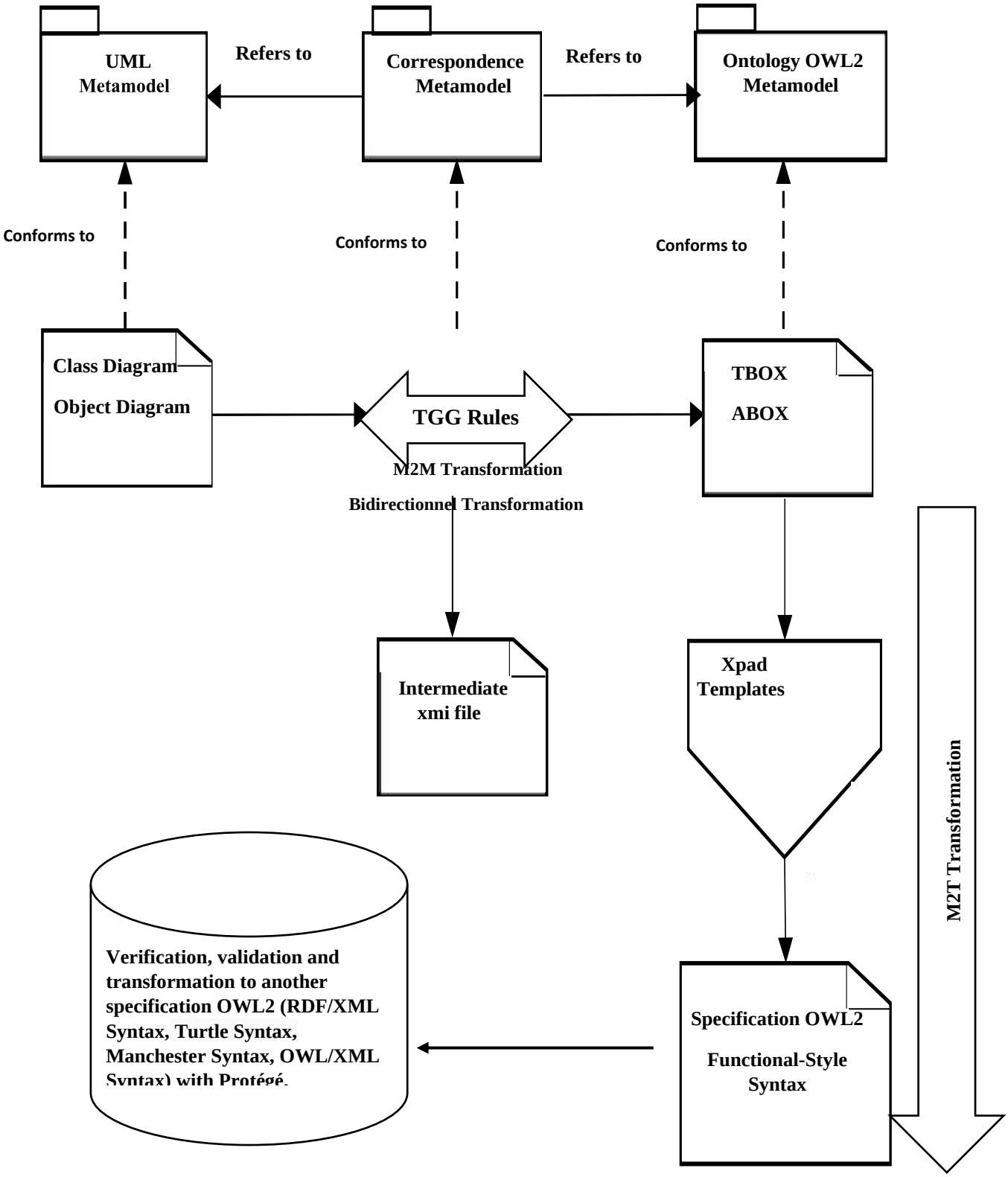


Figure 5. 15: Le processus de l'approche proposée

Pour cette génération automatique de modèles d'ontologie OWL2, nous utilisons la transformation de modèles basée sur l'architecture dirigée par les modèles (MDA). Nous définirons, en utilisant Eclipse Modeling Framework (EMF), le méta-modèle conceptuel spécifique au domaine, le méta-modèle OWL2 et le méta-modèle de correspondance.

Nous définissons les règles TGG : une règle de transformation est une description de la façon dont une ou plusieurs constructions dans le langage source peuvent être transformées en une ou plusieurs constructions dans le langage cible (Kleppe, 2003), pour la réécriture de graphes afin d'effectuer la transformation de modèle à modèle (ou la cohérence des modèles), en utilisant l'interpréteur TGG, par rapport aux trois méta-modèles définis. Tous ces méta-modèles et les règles TGG sont utilisés pour générer du code Java dans l'environnement d'Eclipse Modeling Framework (EMF) pour éditer des modèles conceptuels d'ontologies de domaine, et ensuite générer leur description formelle avec le langage OWL2.

5.3.1 La Méta-modélisation

La méta-modélisation est l'activité de construire des méta-modèles, elle représente un concept fondamental sur lequel se base l'ingénierie des outils. L'utilisation de cette technique s'est accentuée avec l'arrivée de l'IDM qui vise à fournir des langages de modélisation, plus abstraits et plus facilement maîtrisables que des langages de programmation tel que Java ou C, tout en ayant les qualités d'un langage directement interprétable par une machine. Dans le domaine de l'informatique, la méta-modélisation se définit comme la mise en évidence d'un ensemble de concepts pour un domaine particulier (Barais 2007). Parmi ces concepts, il y a le modèle qui représente un phénomène du monde réel, le méta-modèle qui est une abstraction mettant en évidence les concepts utilisés pour définir le modèle et le méta-méta-modèle qui est forme d'abstraction d'ordre supérieur composée de concepts génériques permettant de définir des méta-modèles (consulter la figure 5.16). Un méta-modèle est une « définition formelle » d'un modèle qui aide à le comprendre et qui facilite le raisonnement sur sa structure, sa sémantique et son usage. De la même manière qu'il est nécessaire d'avoir un méta-modèle pour interpréter un modèle, pour pouvoir interpréter un méta-modèle il faut disposer d'une description du langage dans lequel il est écrit : un méta-modèle pour les métamodèles. C'est naturellement que l'on désigne ce méta-modèle particulier par le terme de méta-méta-modèle (ou langage de méta-modélisation). En pratique, un méta-méta-modèle détermine le paradigme utilisé dans les méta-modèles (les concepts, les liens entre eux et leurs sémantiques). Un langage de modélisation conceptuel peut servir, dans la plupart des cas comme un langage de méta-modélisation. Par exemple, le langage

UML peut être considéré comme un langage de modélisation ou encore un langage de méta-modélisation.

La méta-modélisation, est très utilisée dans le domaine de l'ingénierie des systèmes d'information et particulièrement dans l'ingénierie des modèles et des méthodes. Cette technique bénéficie des avantages de la modélisation, notamment la séparation des préoccupations pour définir un système qui est une propriété de plus en plus utilisée dans le développement de systèmes informatiques afin d'en maîtriser la complexité. Dans l'ingénierie des méthodes, la méta-modélisation représente un outil conceptuel indispensable pour définir et raisonner sur de nouveaux modèles. En effet, une méthode dispose habituellement d'un ou de plusieurs « langages de modélisation », permettant de représenter différents aspects (statiques, dynamiques, etc.), de différentes phases (analyse, conception, etc.) du développement d'un système d'information .Il est donc intéressant de disposer d'un méta-langage définit à un niveau d'abstraction en plus de la manière avec laquelle ces éléments interagissent lors de l'exécution haut, pour pouvoir lier les concepts des différents langages de modélisation mis en oeuvre dans la méthode. Dans l'ingénierie des outils, les méta-modèles sont une définition formelle nécessaire pour concevoir et construire les outils pour éditer, vérifier, transformer et éventuellement exécuter des instances de ces modèles. Dans ce qui suit, nous présenterons un certain langage de méta modélisation dans le contexte de l'ingénierie des outils. (Mallouli 2014)

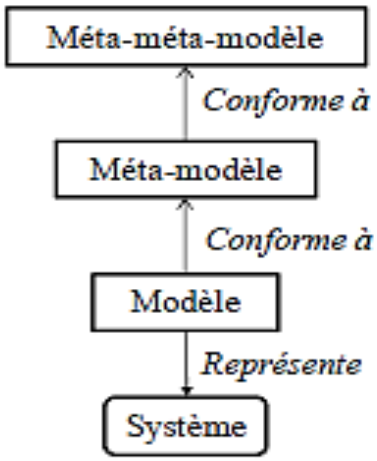


Figure 5. 16 : Notions de base de la méta-modélisation.

5.3.1.1 MÉTA-MODÈLE “ONTOLOGY-CONCEPTUAL”

Les modèles sources de la transformation de modèle à modèle doivent être conformes au méta-modèle ontologique-conceptuel, qui est défini conformément à la norme Ecore. Ce méta-modèle est divisé en trois packages Ecore. Le premier package contient une classe Ecore (Consulter la figure 5.17) appelée "UMLDiagrams" pour relier les deuxième et troisième packages Ecore par une relation d'extension. Cette classe indique que le méta-modèle doit contenir un package pour le diagramme de classes et plusieurs packages Ecore pour les diagrammes d'objets. Un diagramme de classes peut être une extension de diagrammes de classes existants référencés par leur URI.

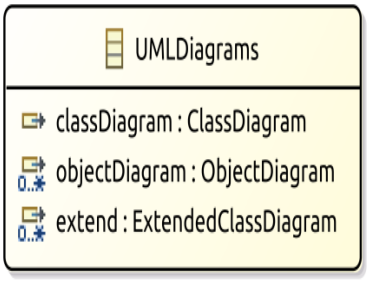


Figure 5. 17: Méta-modèle de la classe Ecore “UMLDiagrams”

Le deuxième package Ecore contient un diagramme de classes (Figure 5.18), qui contient des classes Ecore (au moins une classe) pour définir les concepts de l'ontologie et les associations pour définir les relations conceptuelles entre les concepts. Les concepts conceptuels et les associations sont des sous-classes de la classe Ecore "Entity" avec les attributs "name", "label" et "comment". Un concept conceptuel est une classe nommée "Class", qui est une spécialisation de la classe "Entity". Une association est une autre classe appelée "Association", qui est une généralisation de deux classes Ecore, l'association de classe "ClassAssociation" et l'association de données "DataAssociation".

Une association de classe est définie pour mettre en relation deux concepts ("Class"). Le premier concept est sa source ("AssociationSource") et le second concept est sa cible ("ClassAssociationTarget"). Une association de classe a pour attributs caractéristiques d'être fonctionnelle, fonctionnelle inverse, transitive, symétrique, asymétrique, réflexive et/ou irréflexive. Une association de données, qui a l'attribut caractéristique fonctionnel, est utilisée pour définir la propriété des données pour les objets des classes (concepts). Dans ce cas, la cible de l'association de données ("DataAssociationTarget") a un attribut de type "DataType" pour donner des valeurs à la propriété du concept, qui peut être de différents types comme un entier, une chaîne de caractères, etc.

Un concept ("Classe") peut être équivalent, sous-classe ou disjoint avec un autre concept qui peut être un concept simple, qui est précédemment déclaré ou une expression complexe d'une classe Ecore "Class". Une expression d'une "classe" complexe peut être créée à l'aide des opérateurs de complément de classe, d'union et d'intersection et/ou de restriction d'association, qui peut être une restriction d'association de classe ou une restriction d'association de données. Une restriction d'association peut être une restriction existentielle (certains), universelle (seulement) ou de cardinalité (min, max, exact) sur une association avec une qualification représentée comme expression de classe Ecore. Un opérateur d'une expression de classe Ecore peut être unaire avec un opérande (classe) ou binaire avec deux opérandes (expressions de classe Ecore gauche et droite). Nous pouvons déclarer des associations comme étant des associations équivalentes, super, sous, inverses et disjointes avec une autre association (consulter la figure 5.18 pour plus de détails).

Le troisième package Ecore contient un diagramme de classes appelé "ObjectDiagram" (Consulter la figure 5.19), qui contient des classes Ecore pour définir les instances des classes de concepts et d'associations. Le modèle conceptuel peut avoir plusieurs diagrammes d'objets pour modéliser des ensembles de données multi-sources. Un diagramme de classes de type "ObjectDiagram" est composé d'un ensemble d'entités objets, chacune ayant un nom et pouvant avoir une étiquette et/ou un commentaire. Un objet peut être une instance de classes (concepts). Il peut avoir des instances d'association de classe avec d'autres objets et il peut avoir des instances d'association de données avec une donnée d'un type spécifié.

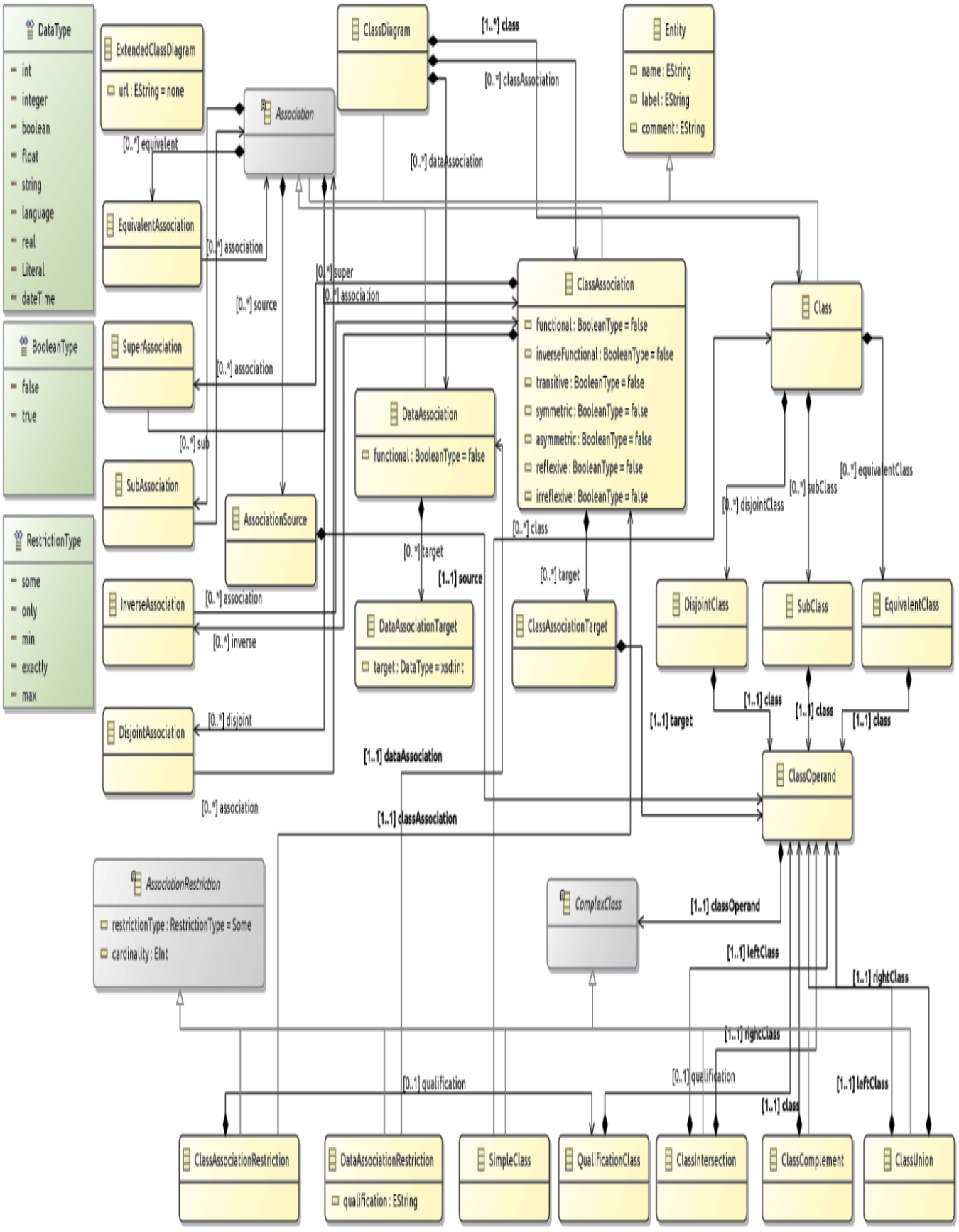


Figure 5. 18 : Le méta-modèle de diagramme de classes "ClassDiagram ».

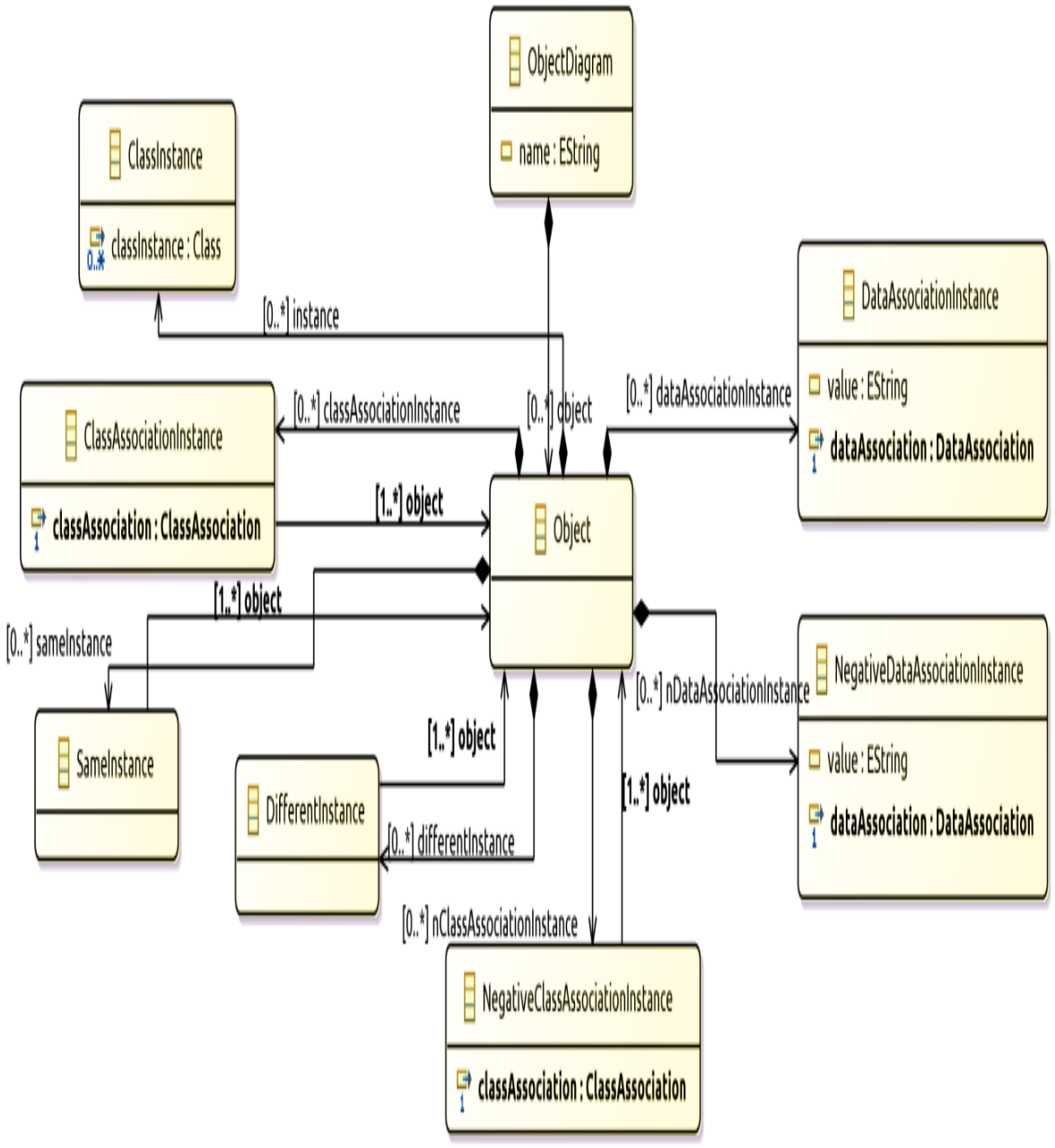


Figure 5. 19 : Le méta-modèle des diagrammes d'objet "ObjectDiagram"

On peut préciser que deux instances d'objets sont identiques (même instance) ou différentes (instance différente). Il est également possible de dire qu'un objet n'est pas instance d'une association de classes (instance négative d'association de classes), par la même, on peut dire qu'il n'est pas instance d'une association de données (instance négative d'association de données). Par exemple, la figure5 .20 décrit, par rapport à ce méta-modèle conceptuel, un modèle d'ontologie dans le domaine des relations familiales. Il contient les noms "Personne", "Parent", "Mère" et "Père" pour les concepts (classes Ecore), où "Parent", "Mère" et "Père" ont des sous-classes de la classe "Personne" et la classe "Parent" est l'union des classes "Mère" et "Père".

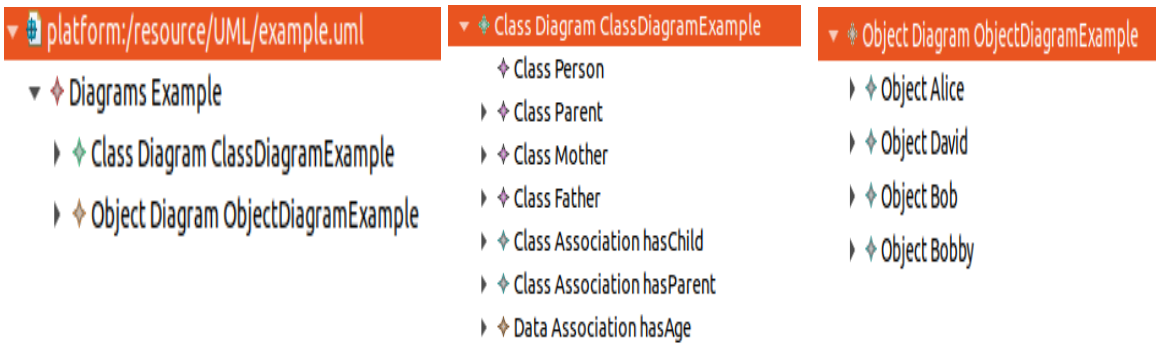


Figure 5. 20 : Exemple d'un modèle conceptuel Ecore pour les relations familiales.

La classe "Mother" est disjointe de la classe "Father". Les noms "hasChild" et "hasParent" sont définis pour des associations de classes entre la classe "Person" et elle-même, les noms "hasName" et "has Age" sont définis pour des associations de données entre la classe "Person" et un type de données (String et integer, respectivement).

Le diagramme d'objets contient les objets "Alice" (instance de la classe "Mother"), "David" (instance de la classe "Father"), "Bob" et "Bobby" (instances de la classe "Person"), et ils sont identiques. Nous pouvons définir que "Bob" a 16 ans (instance de l'association de données "hasAge"), que "David" a un enfant "Bob" (instance de l'association de classes "hasChild") et que "Bob" a un parent "David" (instance de l'association de classes "hasParent"), que "Alice" est la mère de "Bob" (instance de l'association de classes "hasChild").

5.3.1.2 MÉTA-MODÈLE DES ONTOLOGIES OWL2 “ OWL2-ONTOLOGY META-MODEL ”

OWL 2 est un langage pour représenter des ontologies dans le Web sémantique. C’est une extension du vocabulaire de RDF(S). Le langage OWL offre aux machines de plus grandes capacités d’interprétation du contenu Web que celles permises par XML, RDF et RDF schéma (RDFS), grâce à un vocabulaire supplémentaire et une sémantique formelle. Inspiré des logiques de descriptions, OWL2 fournit un grand nombre de constructeurs permettant d’exprimer de façon très fine les classes de manière plus complexe correspondant aux connecteurs de la logique de description équivalente (intersection, union, restrictions diverses, etc.), les propriétés des classes définies (telles que la disjonction), la cardinalité (par exemple "exactement un"), plus des types des propriétés (propriétés d’objet ou d’annotation. . .), des caractéristiques des propriétés (par exemple la symétrie, la transitivité), et des classes énumérées. Le langage d’ontologie Web OWL définit et instancie des ontologies Web. Une ontologie OWL2 peut contenir des descriptions de classes, de propriétés et de leurs instances. Pour une telle ontologie, la sémantique formelle OWL

indique comment déduire ses conséquences logiques, c'est-à-dire les faits non pas littéralement présents dans l'ontologie mais qui découlent de la sémantique. Ces inférences peuvent être fondées sur un seul document ou sur plusieurs documents répartis combinés à l'aide de mécanismes OWL définis. Une ontologie formalisée en OWL2 comprend un espace de nom, l'entête pour décrire l'ontologie, la définition des classes, des propriétés et des instances.

OWL2 est un langage du Web sémantique basé sur la logique de description (SROIQ(D)) (Boris, 2012), conçu par le W3C pour exprimer des connaissances riches et complexes sur les concepts, les groupes de concepts et les relations entre les concepts. Il s'agit d'un langage basé sur la logique informatique, de sorte que les connaissances exprimées dans OWL2 peuvent être traitées par des programmes informatiques, soit pour vérifier la cohérence de ces connaissances, soit pour rendre explicites des connaissances implicites. Les documents OWL2, appelés ontologies, peuvent être publiés sur le World Wide Web et peuvent faire référence à d'autres ontologies OWL2 ou être référencés par elles.

Dans cette section, nous définissons un méta-modèle Ecore pour OWL2, où les éléments du méta-modèle conceptuel peuvent être mis en correspondance avec les éléments du méta-modèle OWL2. Ensuite, les modèles respectant sa syntaxe et sa sémantique peuvent être utilisés pour générer du code OWL2. Le but est de faire en sorte que les utilisateurs non familiers avec ce modèle puissent se débrouiller avec la conceptualisation de l'ontologie du domaine et que le programme de transformation génère son code OWL2. Ainsi, la définition de ce méta-modèle Ecore est basée sur la syntaxe OWL2 et la structure du méta-modèle conceptuel Ecore.

Le méta-modèle OWL2 est composé de trois packages Ecore. Le premier package contient une classe Ecore appelée "Ontology" (Consulter la figure 5.21) et possède trois attributs : le premier attribut permet d'importer des ontologies externes en spécifiant leur URI, le deuxième attribut est une référence au deuxième package Ecore, qui contient la terminologie et le troisième attribut est une référence à un ensemble d'assertions. Ainsi, ce package peut-être considérer comme une extension d'un package TBOX et de plusieurs packages ABOX, qui sont déployés ci-dessous.

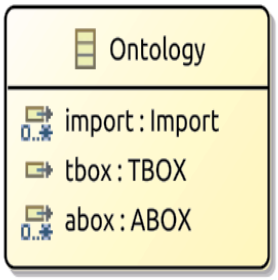


Figure 5. 21 : Méta-modèle de la classe ontologie OWL2 Ecore "Ontology OWL 2".

Le deuxième package Ecore (Figure 5.22) est composé de classes Ecore pour définir la terminologie de l'ontologie (TBOX). La classe Ecore TBOX est composée de classes permettant de mettre en correspondance les éléments du méta-modèle conceptuel avec les informations supplémentaires requises pour la génération du code OWL2.

La classe TBOX est composée de classes "Concept" et "Property" (généralisation de "ObjectProperty" et "DataProperty"), qui sont des spécialisations de la classe "Entity". Une classe "Concept" peut être équivalente, sous-classe ou disjointe avec une autre classe "Concept" qui peut être une expression complexe de concept. Une propriété peut avoir des attributs représentant certaines de ses caractéristiques et elle peut être spécifiée comme propriété équivalente, sous-propriété, etc.

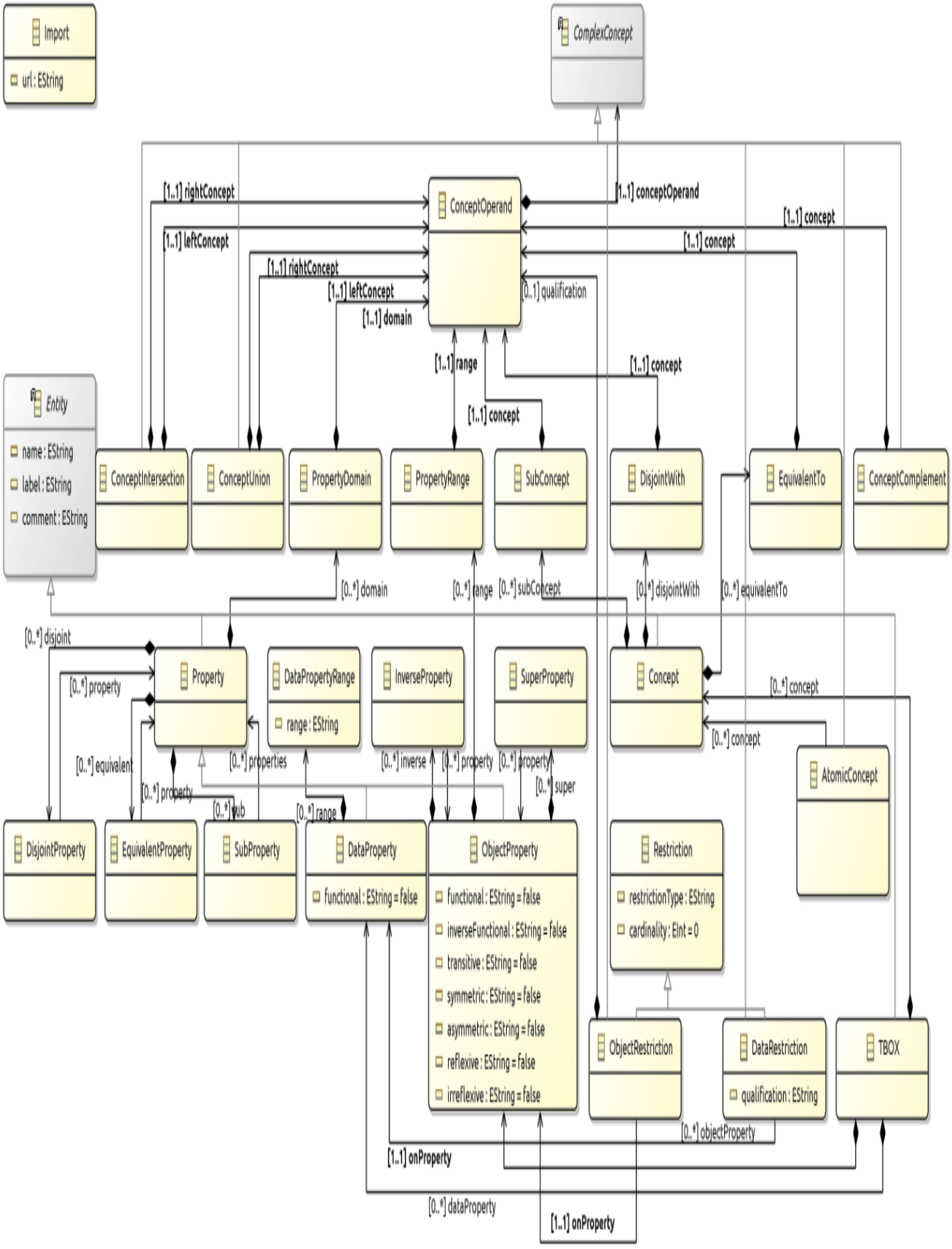


Figure 5. 22: Méta-modèle de diagramme de classes Ecore "TBOX" du OWL2.

Le méta-modèle OWL2 peut également contenir un ensemble de packages Ecore représentant des boîtes d'assertions (ABOX) comme ensembles de données multi-sources pour définir des instances pour les concepts et les propriétés (Consulter la figure 5.23). La principale classe Ecore est la classe "Individual" pour créer des individus (objets) en tant que membres de concepts. Les individus peuvent être reliés par des propriétés d'objet (instances de ObjectProperty") et ils peuvent avoir des propriétés de données (instances de "DataProperty"). Nous pouvons nier ces instances en déclarant des instances de NegativeObjectPropertyAssertion" et "NegativeDataPropertyAssertion", respectivement. Deux individus peuvent être déclarés identiques par l'instance "SameAs", différents par l'instance "DifferentFrom".

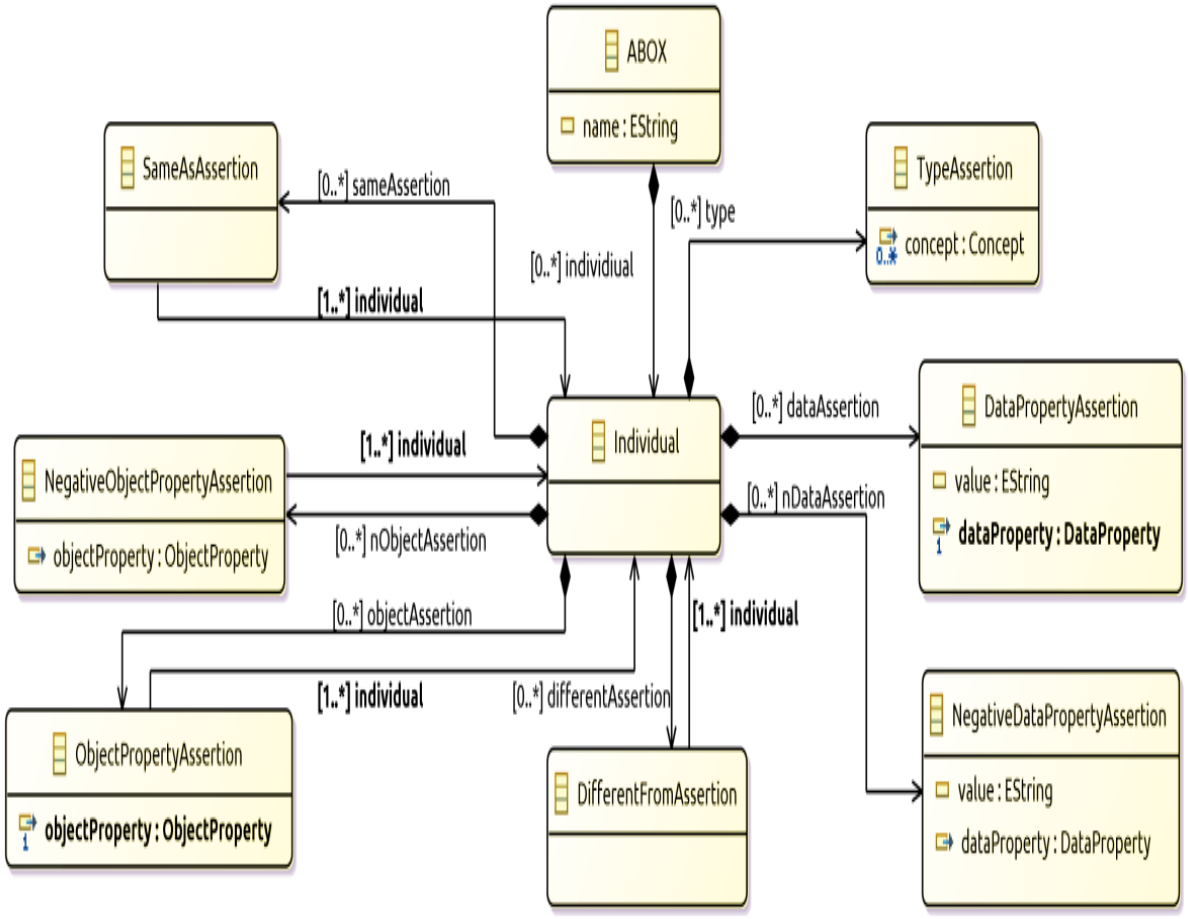


Figure 5. 23 : Méta-modèle de diagramme de classes Ecore "ABOX" du OWL2.

Par exemple, le même modèle de la figure (5.20) peut être décrit avec le méta-modèle OWL2 comme suit :

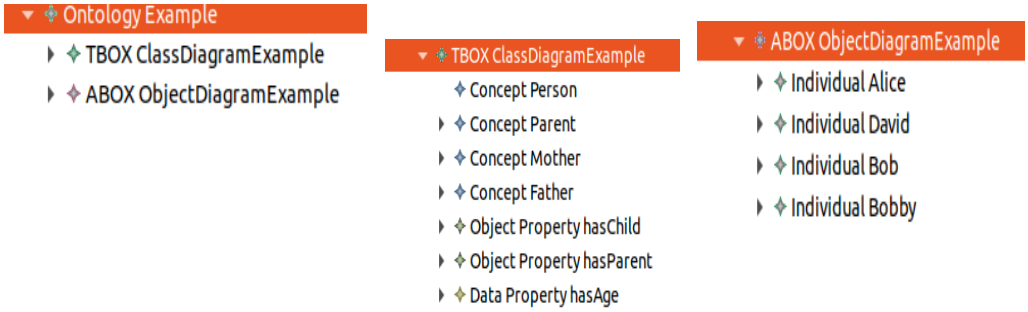


Figure 5. 24: Exemple d'un modèle Ecore OWL2 pour les relations familiales.

5.3.2 TRASFORMATION MODEL-TO-MODEL(M2M)

La transformation de modèle est un processus de génération d'un modèle cible basé sur un modèle source. Les règles de transformation doivent être construites entre les concepts similaires qui sont issus des deux modèles, respectivement. Les grammaires à triple graphes (TGG) sont une technique formellement bien définie pour la transformation de modèle à modèle. Il s'agit d'une transformation bidirectionnelle déclarative et basée sur des règles (Anjorin, Leblebici et Schürr, 20 Years of Triple Graph Grammars: A Roadmap for Future Research 2015). L'idée générale des TGGs est de définir un langage de modèles intégrés, qui consiste en un modèle du domaine source, un modèle du domaine cible, et des structures de correspondance (mapping) explicites dans la composante intermédiaire.

Le résultat de la transformation TGG consiste en un ensemble de règles qui décrivent comment un ensemble de triplets de modèles de domaine source (conceptualisation d'ontologie), de correspondance et de domaine cible (OWL2) peut être produit simultanément et peut être considéré comme une relation de cohérence. Si nous désignons les modèles par G_X, où G représente le graphe et G et $X \in \{S, C, T\}$ représente les domaines (source, correspondance, ou cible) du modèle. Les triples de modèles sont par conséquent dénotés par $G_S \leftarrow G_C \rightarrow G_T$. Les règles d'un TGG génèrent un langage de triples cohérents, noté$\mathcal{L}(TGG)$. Avec cette notation, nous pouvons maintenant formuler succinctement la cohérence (consistance) basée sur les TGG comme suit :

G_S est cohérent avec G_T si et seulement si $\exists\ G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)$ (Hildebrandt al. 2013). En d'autres termes, le triple $G_S \leftarrow G_C \rightarrow G_T$ peut-être généré avec le TGG.

5.3.2.1 MÉTA-MODÈLE DE CORRESPONDANCE

Avec les TGG, pour garantir qu'un modèle source est cohérent avec un modèle cible, nous spécifions un modèle de correspondance entre eux, qui sera utilisé par un ensemble de règles TGG pour vérifier la cohérence. Ce modèle de correspondance doit être conforme à un méta-modèle de correspondance défini. De même, les modèles source et cible sont conformes à leurs méta-modèles correspondants qui sont reliés par le méta-modèle de correspondance. Le triple des méta-modèles source, cible et de correspondance est appelé schéma TGGs.

Dans ce contexte, nous définissons le méta-modèle Ecore de correspondance (Consulter figure 5.25) pour la transformation de modèle à modèle (modèle conceptuel d'ontologie à modèle OWL2), qui crée des éléments du modèle cible. Il peut être considéré comme un ensemble de liens de traçabilité entre les éléments de modèle correspondants des domaines source et cible.

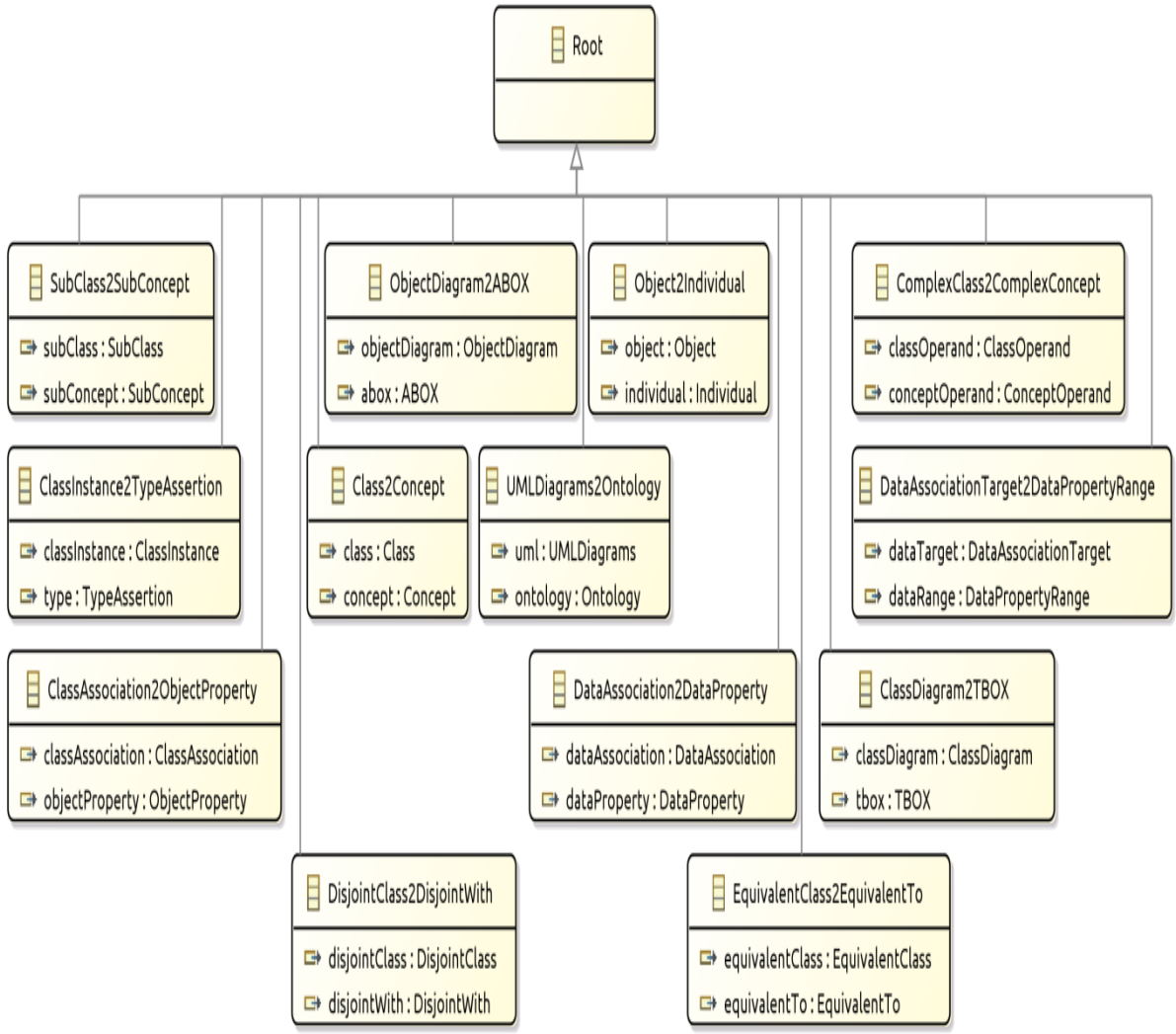


Figure 5. 25: MÉTA-MODÈLE DE CORRESPONDANCE

Le méta-modèle de correspondance est composé de treize classes Ecore, qui sont toutes des spécialisations d'une classe Ecore (la classe racine). Chaque classe spécialisée est définie pour mettre en correspondance les éléments des modèles source et cible. Par exemple, la classe Ecore "Class2Concept" sera utilisée pour mettre en correspondance l'élément "Class" du modèle source avec l'élément "Concept" du modèle cible.

5.3.2.2 les règles TGG

Principalement, une grammaire de graphes triple est considérée comme une grammaire de graphes régulière avec un ensemble de règles de grammaire de graphes, où le graphe vide est l'axiome. Ces règles génèrent un langage de graphes ou, plus précisément, l'ensemble de tous les graphes cohérents, qui est un sous-ensemble de l'ensemble de tous les graphes conformes au schéma. Le point intéressant d'un TGG est le fait que les règles spécifiées consistent en trois sous-règnes et que les graphes générés consistent en trois sous-graphes liés. Un composant source avec un composant cible représente une paire de graphes liés et le composant central (correspondance) introduit des relations de traçabilité entre la paire de graphes. Le processus de transformation utilise les TGG pour générer un ensemble de règles opérationnelles régulières de transformation de graphes adaptées à un objectif spécifique, comme la transformation "en avant" du modèle ou la propagation "en arrière" des changements (Jakob et Schürr 2008) (Jacob, 2008).

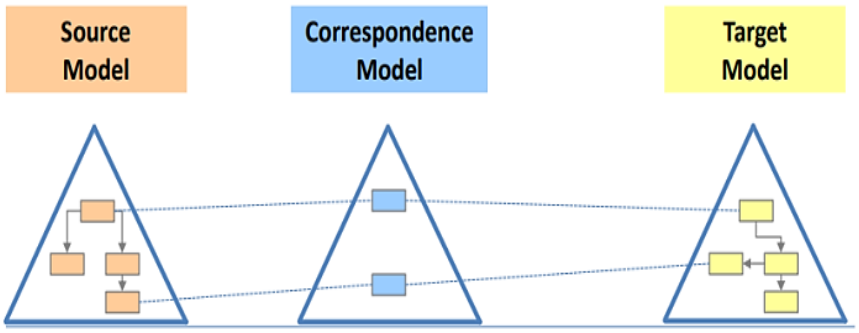


Figure 5. 26 : Principe TGGs (Triple Graph Grammar).

La figure 5.26 illustre le principe des TGG, où les éléments du modèle du domaine source sont mis en correspondance avec les éléments du modèle du domaine cible en utilisant le modèle de correspondance. L'idée générale de la définition des règles de transformation est d'éditer des règles TGG comme ressources de transformation. Pour commencer la transformation, il faut déterminer un nœud de départ dans le modèle d'entrée et une règle opérationnelle pour traduire ce nœud (Figure 5.27). Une telle règle de départ valide sans éléments de contexte, c'est-à-dire sans précondition, est appelée axiome.

- **Règle 1 UMLDiagramms2 Ontology (l'axiome)** : Pour démarrer la transformation, un nœud de départ dans le modèle d'entrée (UMLDiagrams) et une règle opérationnelle *uml2ontology* pour traduire ce nœud doivent être déterminés (figure 5.27). Une telle règle de départ valide sans aucun élément de contexte, c'est-à-dire aucune condition préalable, est appelée un *axiome*. Après la traduction du premier nœud, une stratégie est requise pour couvrir systématiquement tous les éléments du modèle d'entrée dans un ordre approprié. Dans l'axiome, le modèle (de sortie) ontologique prend le même nom, le même label et le même commentaire que le modèle conceptuel ("UMLDiagrams").

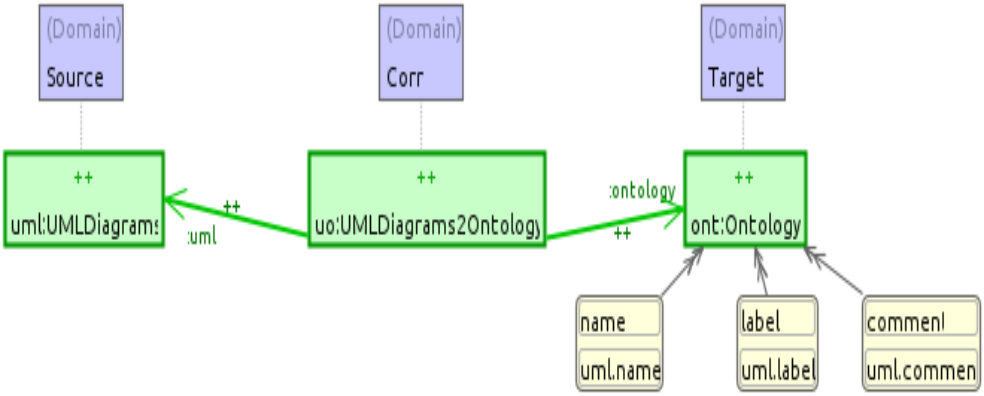


Figure 5. 27 : The axiom(axiom).

- **Règle 2 Adding importations :** Dans ce contexte, la règle (dans la figure 5.28) sera appliquée si le modèle conceptuel est une extension de modèles existants. Après la transformation du premier nœud, une stratégie est nécessaire pour couvrir systématiquement tous les éléments du modèle d'entrée dans un ordre approprié.

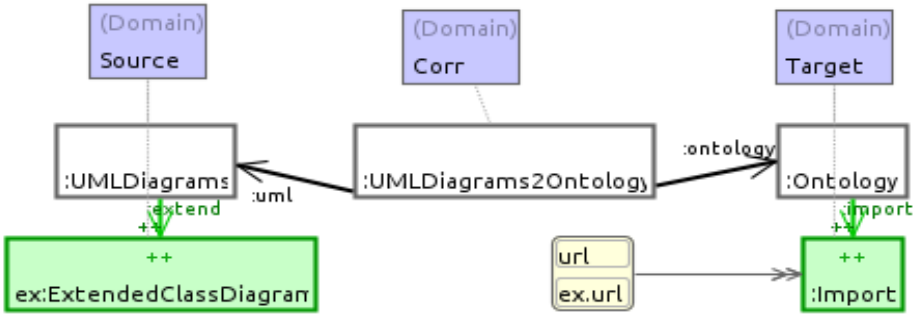


Figure 5. 28 : Adding importations (Ajout d'importations).

- **Règle 3 Creating the context "TBOX"** : La règle de (Figure 2.29) va créer le contexte cible de la terminologie de l'ontologie ("TBOX") à partir du contexte source nommé "ClassDiagram".

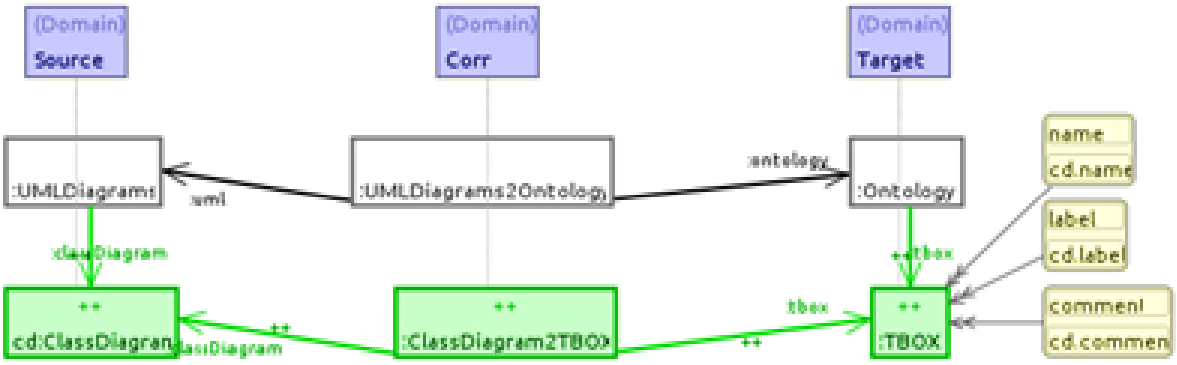


Figure 5. 29 : Creating the context "TBOX" (Création du contexte "TBOX).

- **Règle 4 Mapping Class to Concept** : Chaque classe du contexte source sera traduite en concept dans le contexte cible (Figure 5.30).

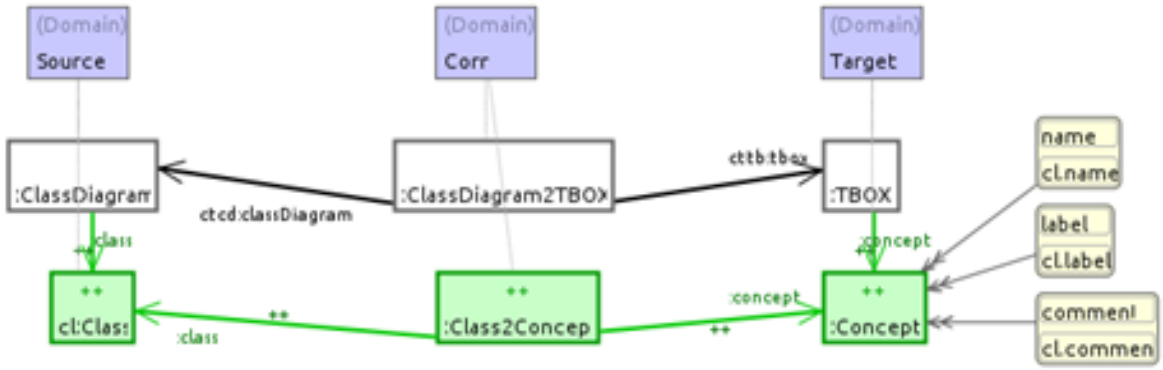


Figure 5. 30 : Mapping Class to Concept (Mapping les classes vers les concepts).

Les règles de la Figure 5.31 et de la Figure 5.32 produiront respectivement les entités de propriété d'objet et de propriété de données pour le contexte TBOX.

- **Règle 5 Class association :** chaque ClassAssociation dans ClassDiagram se transforme en ObjectPrperty dans TBOX ; en conservant le nom du ClassAssociation. L'ObjectProperty peut être : Functional, InversFunctional, Transitive, Symmetric, Asymmetric, Reflexive Irreflexive.et elle peut avoir des labels et comments.

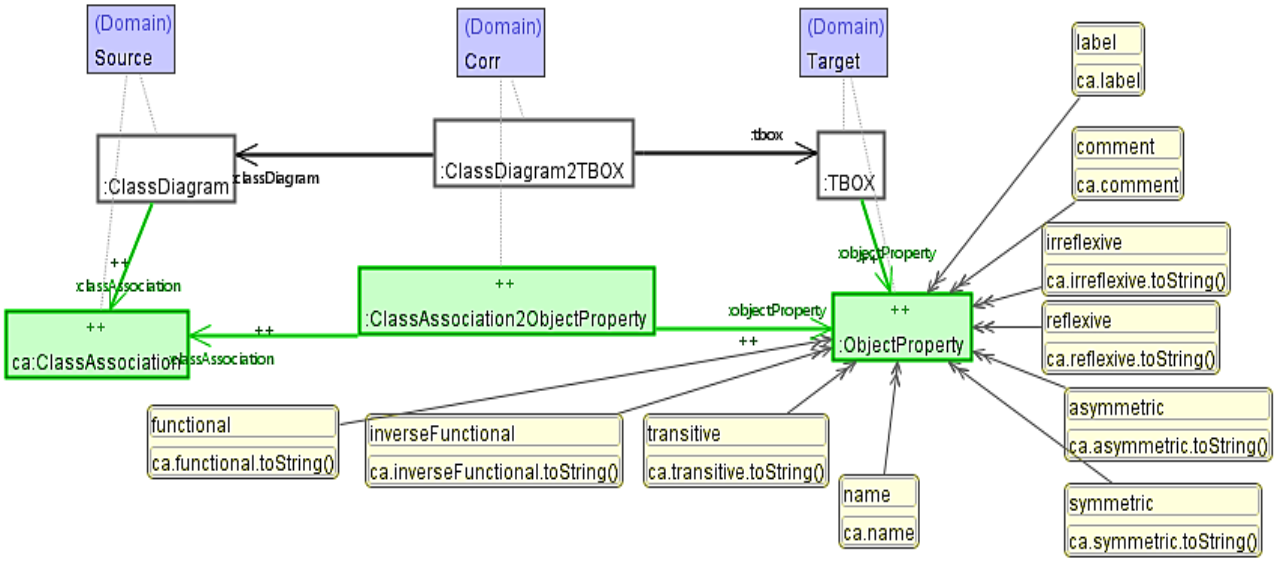


Figure 5. 31 : Class association (Association de classes).

- **Règle 6 Data association :** chaque DataAssociation dans ClassDiagram se transforme en DataPrperty Dans TBOX ; en conservant le nom du DataAssociation. DatatProperty peut être : functional et elle peut avoir des labels et comments.

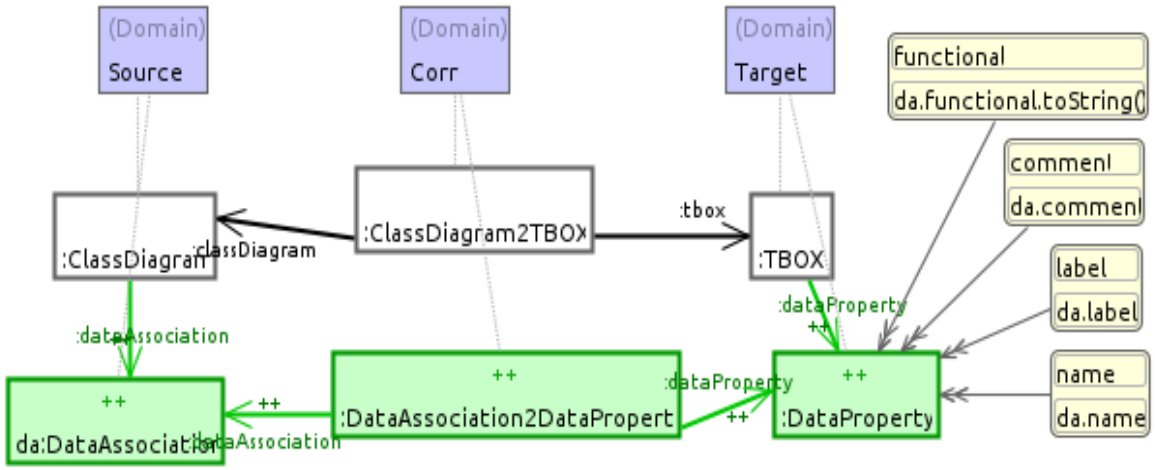


Figure 5. 32 : Data association (Association des données).

Les quatre règles suivantes sont définies pour mettre en correspondance les domaines et les ranges des propriétés (propriété de l'objet et propriété des données).
Les propriétés (propriétés d'objet et propriétés de données) sont mises en correspondance avec les domaines et les ranges à l'aide des quatre règles définies ci-dessous.

- **Règle 7 Object property domain :** cette regèle de transformation génère le domaine du propriété d'objet (Object property domain). La figure 5.33 montre que chaque ClassAssociation

possède une source (ClassOperand) , cette ClassOperand devient un ConceptOperand qui possède la propriété (Property Domain).

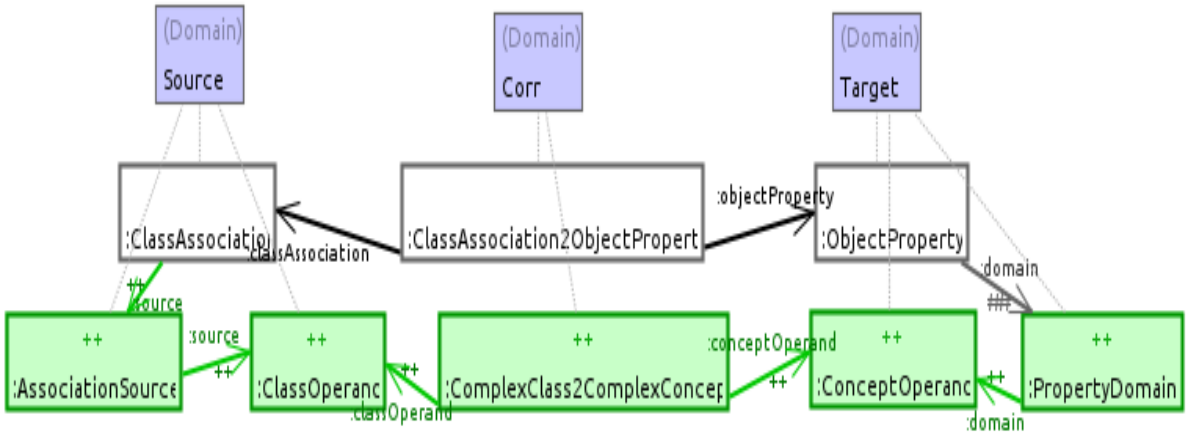


Figure 5. 33 : Object property domain (Domaine de propriété de l'objet).

- **Règle 8 Object property range : Object property range :** cette regèle de transformation génère le range de la propriété d'objet (Object property range). La figure 5.34 montre que chaque ClassAssociation possède une target (ClassOperand) , cette ClassOperand devient un ConceptOperand qui possède la propriété (Property Range).

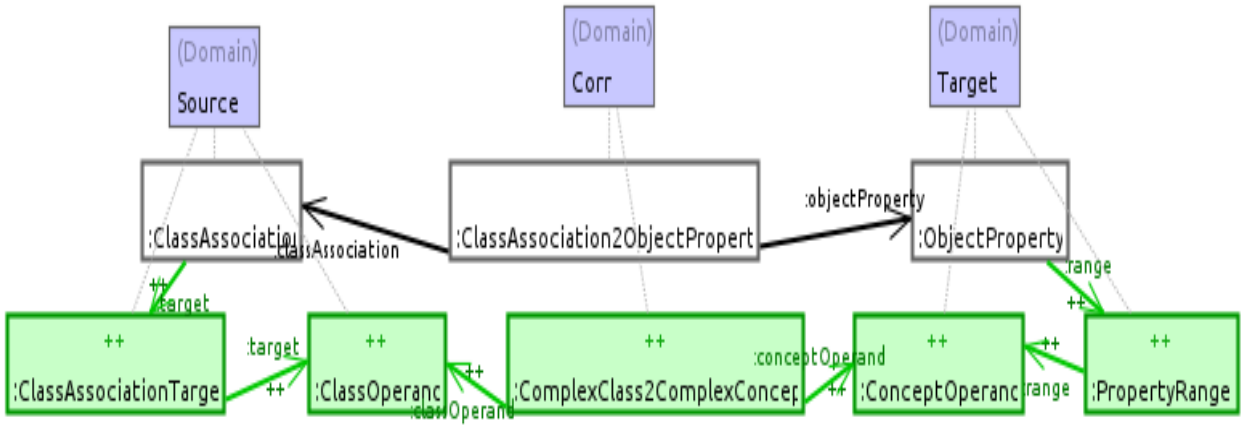


Figure 5. 34 : Object property range (range de propriété d'objet).

- **Règle 9 Data property Domain :** cette regèle de transformation génère le domaine de la propriété de données (Data property domain). La figure 5.35 montre que chaque DataAssociation possède une source (ClassOperand), cette **ClassOperand** devient un ConceptOperand qui possède la propriété (Property Domain).

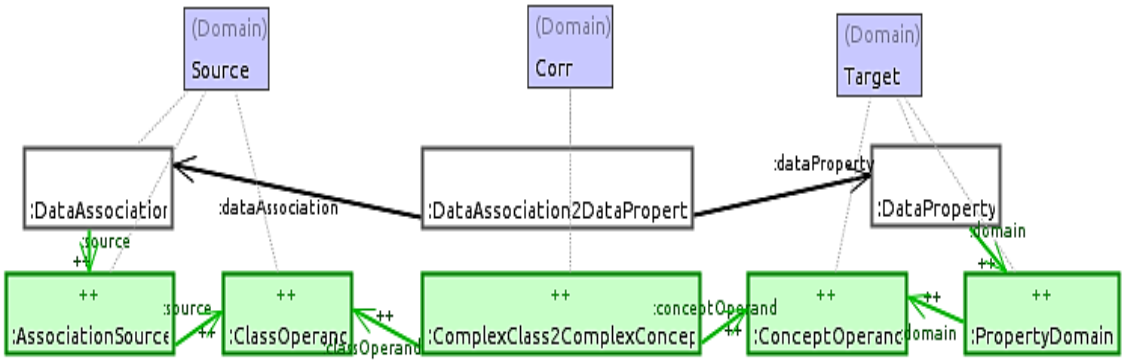


Figure 5. 35 : Data property domain (Domaine de la propriété des données).

- **Régle 10 Data property Range :** cette regèle de transformation génère le range de la propriété de données (Data property range). La figure 5.36 montre que chaque DataAssociation possède une target (DataAssociationTarget), cette DataAssociationTarget se transforme en DataPropertyRange qui possède un range. Il faut également spécifier le type du range.

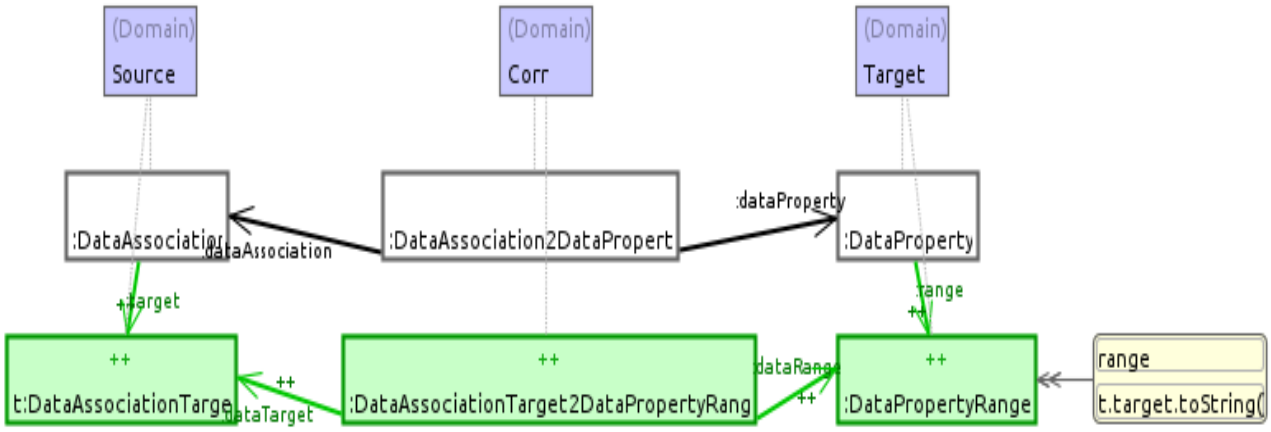


Figure 5. 36 *:* Data property range (range de la propriété des données).

Les trois règles suivantes (Figure 5.37, Figure 5.38 et Figure 5.39), si elles sont appliquées, génèrent pour le contexte de TBOX, les éléments équivalents, sous-classes et disjoints du contexte du modèle de diagramme de classes.

- **Règle 11 Mapping for equivalence :** deux ou plusieurs classes équivalentes génèrent deux ou plusieurs concepts équivalents ; cette transformation est effectuée en exécutant la règle illustrée dans la figure 5.37, ou chaque EquivalentClass génère EquivalentTo et chaque ClassOperand génère un ConceptOperand.

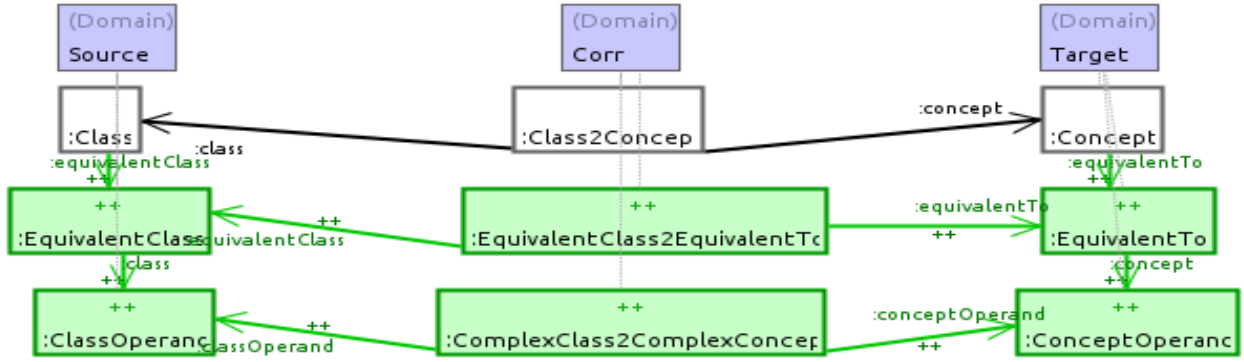


Figure 5. 37 : Mapping for equivalence (Mapping pour l'équivalence).

- **Règle 12 Mapping for inclusion :** pour exprimer l'inclusion des classes (ou subClass). Une classe peut avoir un ou plusieurs sousClass (ou subClass). Cette transformation est effectuée en exécutant la règle illustrée dans la figure 5.38. Ou chaque SubClass génère SubConcept et chaque ClassOperand génère un ConceptOperand.

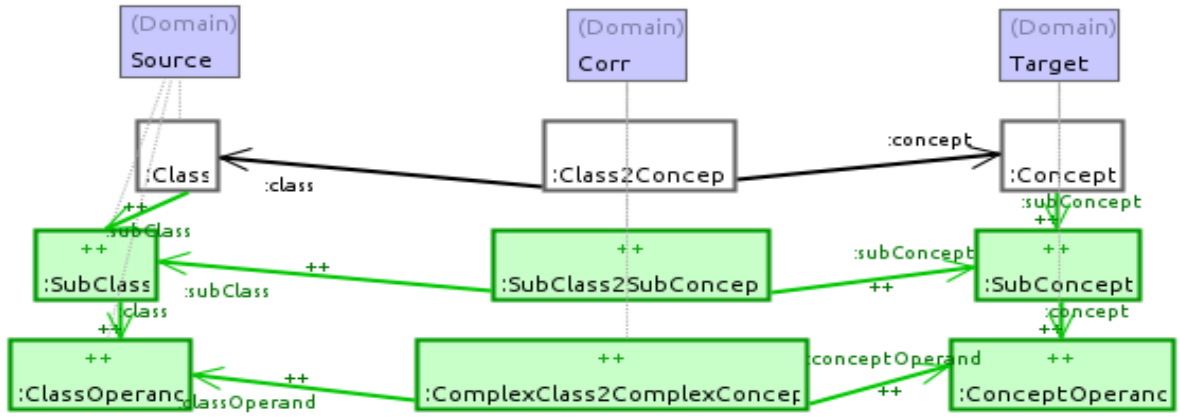


Figure 5. 38 : Mapping for inclusion (Mapping pour l' inclusion).

- **Règle 13 Mapping for disjointness :** pour exprimer la disjonction des classes (ou DisjointClass). Cette transformation est effectuée en exécutant la règle illustrée dans la figure 5.39, ou chaque DisjointClass génère DisjointWith et chaque ClassOperand génère un ConceptOperand.

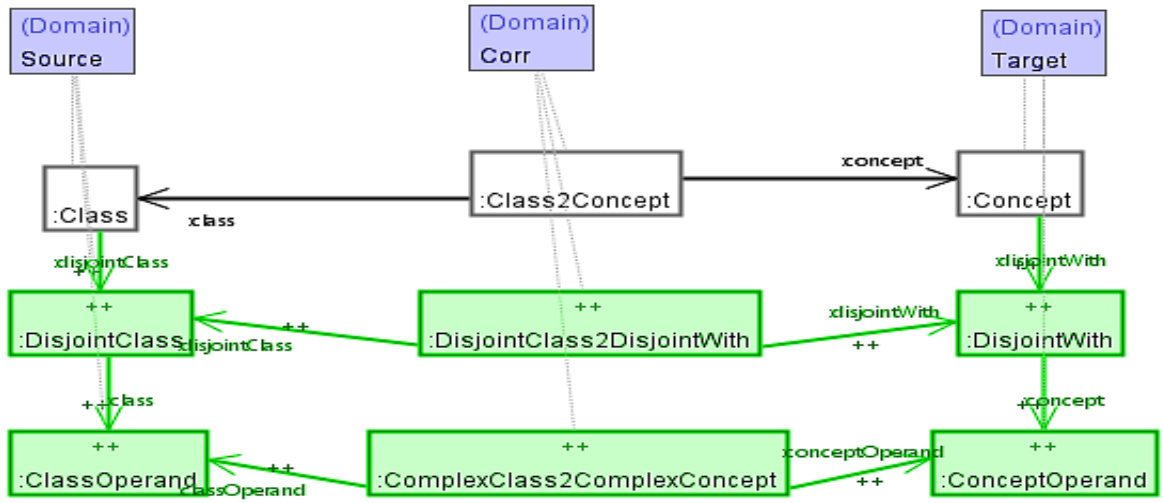


Figure 5. 39 : Mapping for disjointness (Mapping pour la disjunction).

Les cinq règles TGG (Figure 5.40, Figure 5.41, Figure 5.42, Figure 5.43 et Figure 5.44), si elles sont appliquées, ajouteront au contexte de TBOX les propriétés équivalent, sous, disjoint, inverse et super objet pour une association classe source définie dans le modèle conceptuel.

- **Règle 14 Object property equivalence** : deux ClassAssociation (s) ou moins peuvent être équivalent. Cette règle de transformation (Consulter la figure 5.40) génère EquivalentProperty qui exprime l'équivalence entre au moins deux ObjectProperty.

Les ObjectProperties conservent les noms (names) des ClassAssociation incluses dans EquivalentAssociation.

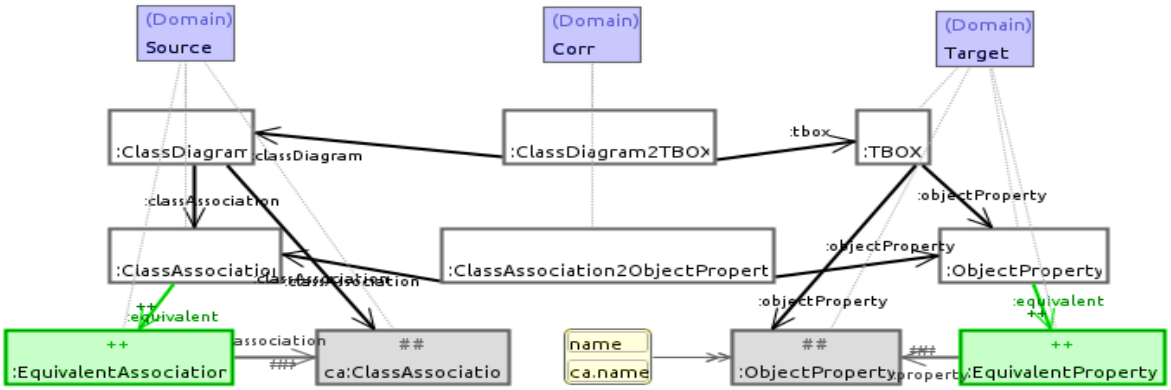


Figure 5. 40 : Object property equivalence (Équivalence des propriétés d'objets)

- **Règle 15 Object property inclusion** : Une ClassAssociation (s) peuvent avoir ou moins un SubAssociation. Cette règle de transformation (Consulter la figure 5.41) génère SubProperty qui

exprime l'inclusion entre au moins deux ObjectProperty. Les ObjectProperties conservent les noms (names) des ClassAssociation incluses dans SubAssociation**.**

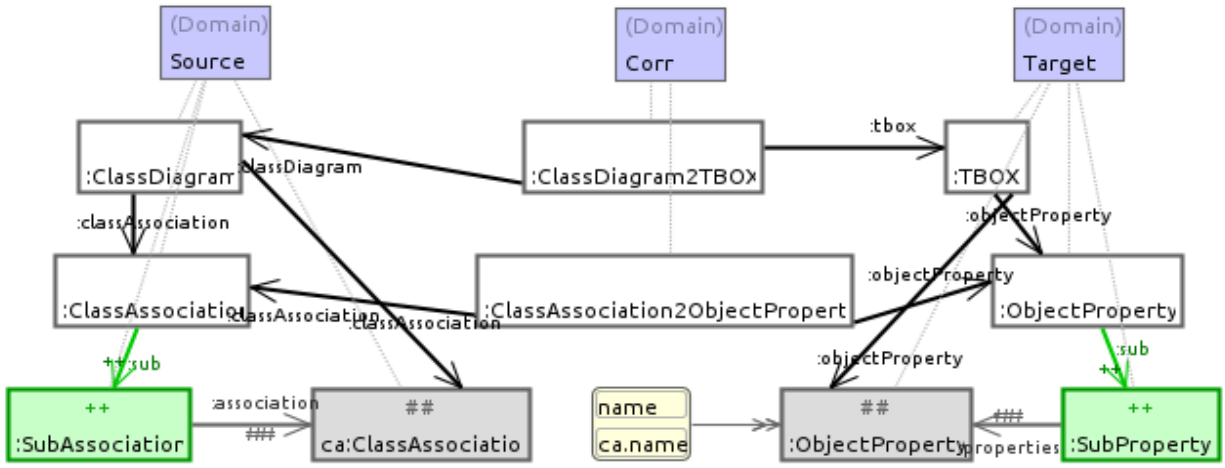


Figure 5. 41 : Object property inclusion (Inclusion des propriétés d'objet).

- **Règle 16 Object property disjointness :** Deux ClassAssociation peuvent être disjoint (DisjointAssociation). Cette règle de transformation (Consulter la figure 5.42) génère DisjointProperty qui exprime la disjonction entre deux ObjectProperty. Les ObjectProperties conservent les noms (names) des ClassAssociation incluses dans DisjointAssociation**.**

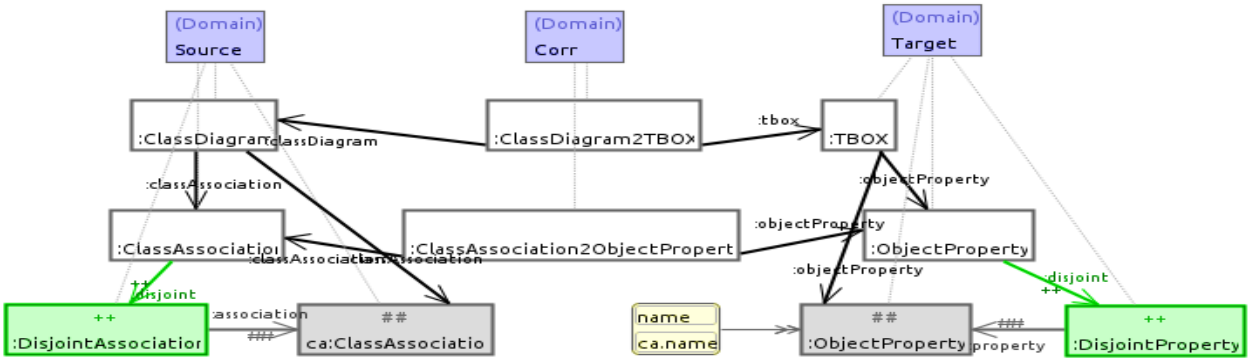


Figure 5. 42 : Object property disjointness (Disjonction de propriétés d'objets).

- **Règle 17 Object property inversion :** Deux ClassAssociation(s) peuvent être inverse l'un de l'autre (InverseAssociation). Cette règle de transformation (Consulter la figure 5.43) génère InverseProperty qui exprime l'inversion entre deux ObjectProperty. Les ObjectProperties conservent les noms (names) des ClassAssociation incluses dans InverseAssociation**.**

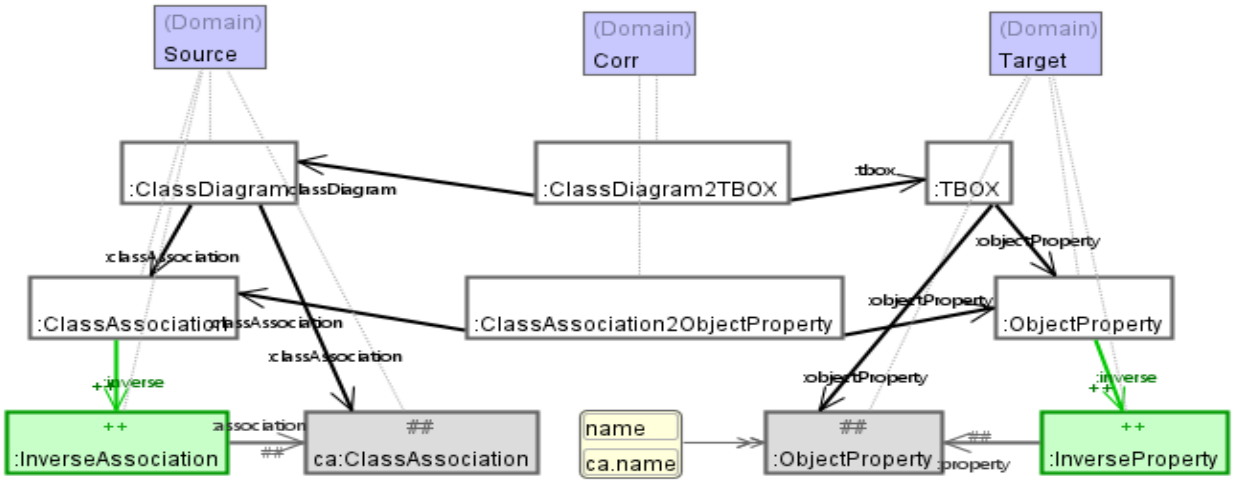


Figure 5. 43 : Object property inversion (Inversion des propriétés d'objets).

- **Règle 18 Object property super :** Une ClassAssociation peut avoir un SuperClassAssocition (SuperAssociation).Cette règle de transformation (Consulter la figure 5.44) génère SuperProperty qui exprime la supériorité entre deux ObjectProperty. Les ObjectProperties conservent les noms (names) des ClassAssociation incluses dans SuperAssociation**.**

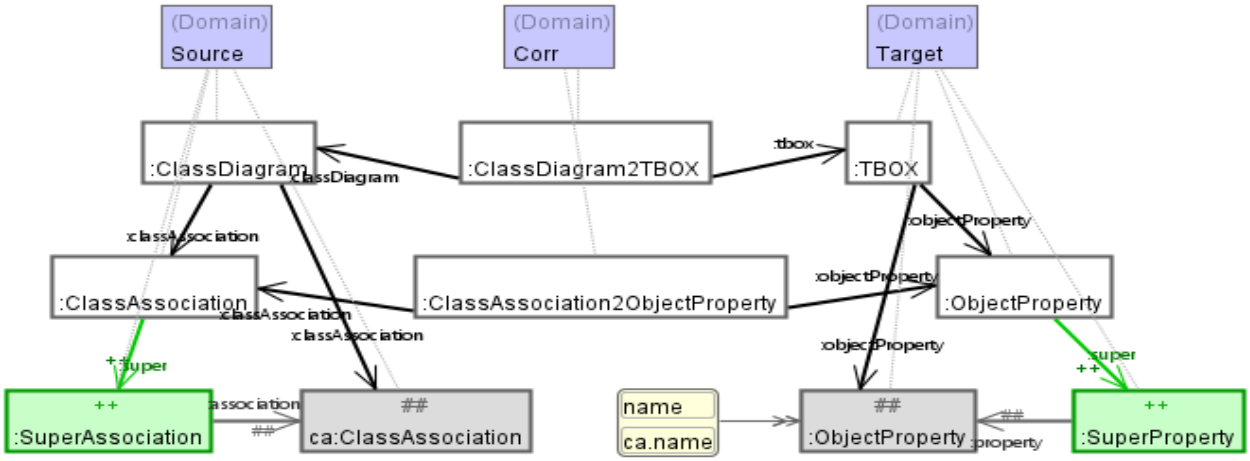


Figure 5. 44 : Object property super (supériorité des propriétés d'objets).

Les trois règles TGG suivantes (Figure 5.45, Figure 5.46 et Figure 5.47) produisent pour une association conceptuelle de données des propriétés de données équivalentes, sub et disjointes dans le contexte de TBOX, respectivement.

- **Règle 19 Data property equivalence :** deux DataAssociation (s) ou moins peuvent être équivalents. Cette règle de transformation (Consulter la figure 5.45) génère EquivalentProperty qui exprime l'équivalence entre au moins deux DataProperty. Les DataProperties conservent les noms (names) des DataAssociation incluses dans EquivalentAssociation.

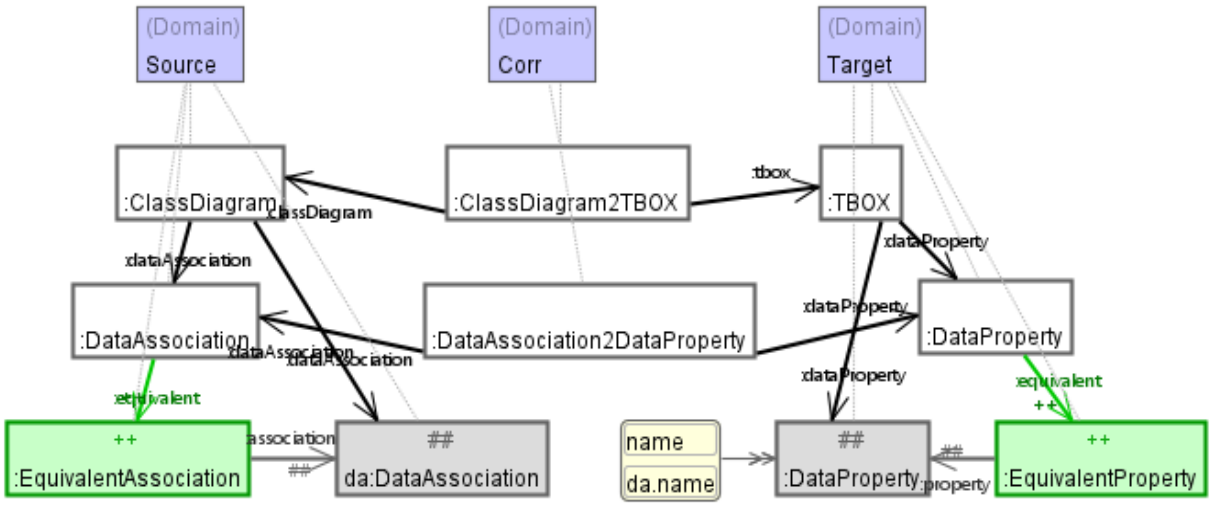


Figure 5. 45 : Data property equivalence (Équivalence des propriétés de données).

- **Règle 20 Data property inclusion** Une DataAssociation peut avoir au moins une Sub DataAssociation (Une ou plusieurs DataAssociation). Cette règle de transformation (Consulter la figure 5.46) génère SubProperty qui exprime l'inclusion entre au moins deux DataProperty. Les DataProperties conservent les noms (names) des DataAssociation incluses dans SubAssociation.

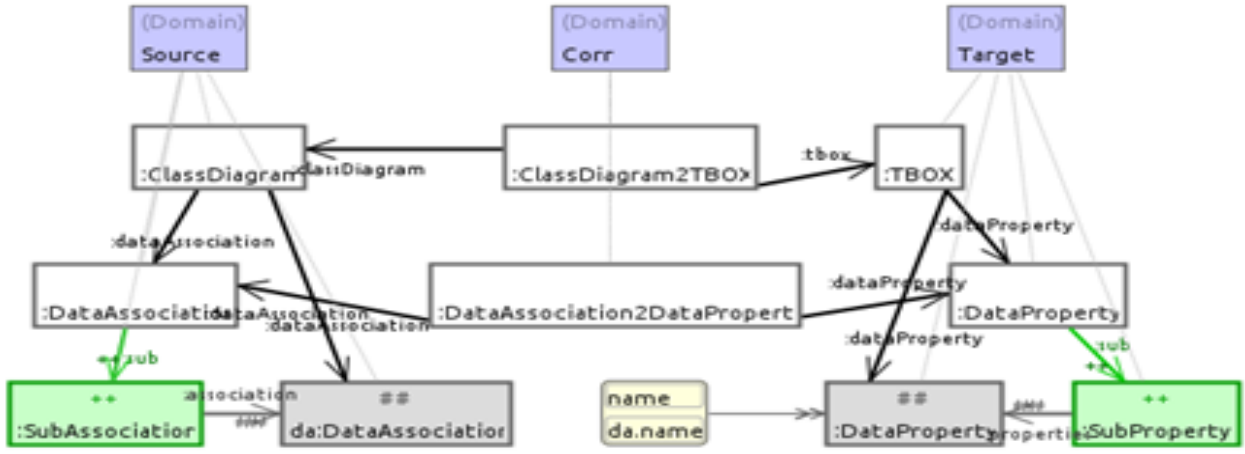


Figure 5. 46 : Data property inclusion (Inclusion des propriétés de données).

- **Règle 21 Data property disjointness** : deux DataAssociation (s) peuvent être disjointes . Cette règle de transformation (Consulter la figure 5.47) génère DisjointProperty qui exprime la disjonction entre deux DataProperty. Les DataProperties conservent les noms (names) des DataAssociation incluses dans DisjointAssociation.

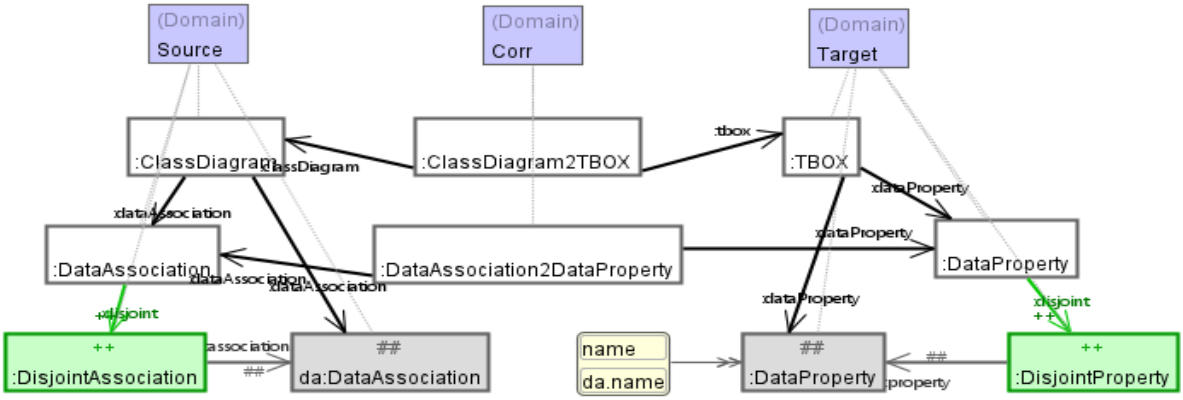


Figure 5. 47 : Data property disjointness (Disjonction des propriétés de données).

Une classe (dans le contexte du modèle conceptuel) ou un concept (dans le contexte du modèle TBOX) équivalent, sous- ou disjoint peut être exprimé par une expression complexe. Les règles TGG suivantes seront utilisées pour mettre en correspondance leurs éléments Ecore.

- **Règle 22 Simple class** : chaque simple class dans le ClassDiagm génère un Atomic Concept dans TBOX. Consulter la figure 5.48

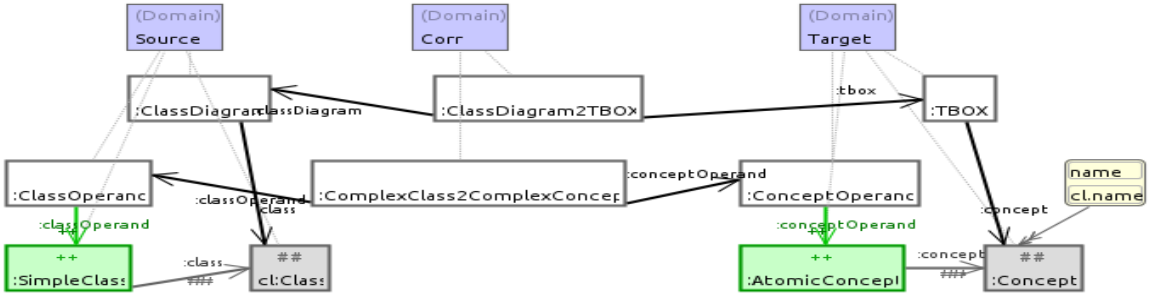


Figure 5. 48 : Simple class (Une classe simple).

- **Règle 23 Class complement** : chaque ClassOperand peut avoir une ClassComplent. Dans ce cas la règle de transformation (Class complement) ; s'exécute pour générer le ConceptComplent (Consulter la figure 5.49).

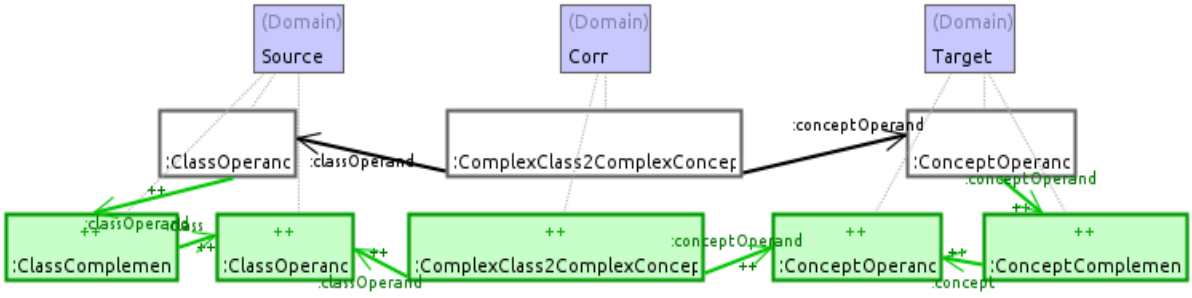


Figure 5. 49: Class complement (Le complément de classe).

- **Régle 24 Classes union :** deux ClassOperand peuvent être unies par ClassUnion ; en appliquant la règle de transformation « Classes union », ClassUnion génère Concept Union qui exprime l'union entre deux ou plus de ConceptOperands. Consulter la figure 5.50.

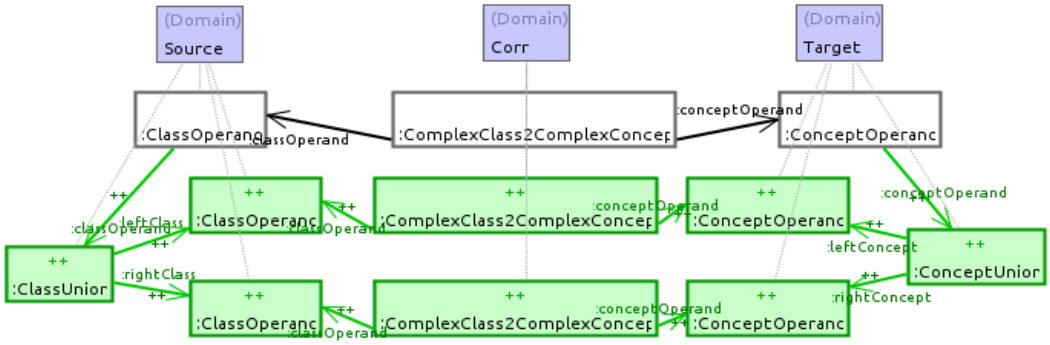


Figure 5. 50 : Classes union (Union de classes).

- **Régle 25 Classes intersection :** deux ClassOperand peuvent être en relation d'intersection, exprimée par ClassIntersection ; en appliquant la règle de transformation « Classes intersection », ClassIntersection génère ConceptIntersection qui exprime l'intersection entre deux ou plus de ConceptOperands. Consulter la figure 5.51.

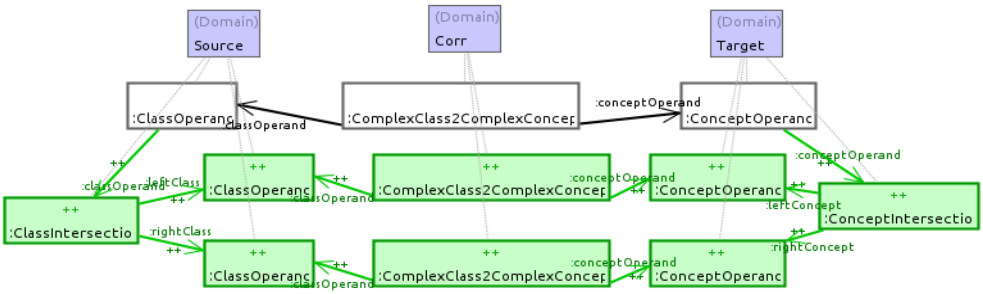


Figure 5. 51 : Classes intersection (Intersection des classes).

Une expression de classe (concept) peut être une restriction sur une association de classe (propriété d'objet), ou une association de données (propriété de données). Le type de restriction indique s'il s'agit d'une restriction existentielle, universelle ou de cardinalité. Si c'est une restriction de cardinalité, elle peut être de cardinalité minimale, maximale ou exacte avec un nombre de cardinalité de type naturel.

- **Régle 26 Restriction on class association :**ClassAssociationRestriction qui lient entre une ClassOperand et une ClassAssociation génère ObjectRestriction qui lient entre ObjectProperty et ConceptOperand .ObjectProperty, selon le contexte peut exprimer une restriction existentielle, universelle ou de cardinalité. S'il s'agit d'une restriction de cardinalité, elle peut être de cardinalité

minimale, maximale ou exacte avec un nombre de cardinalité de type naturel. Cette règle de transformation est illustrée dans la figure 5.52.

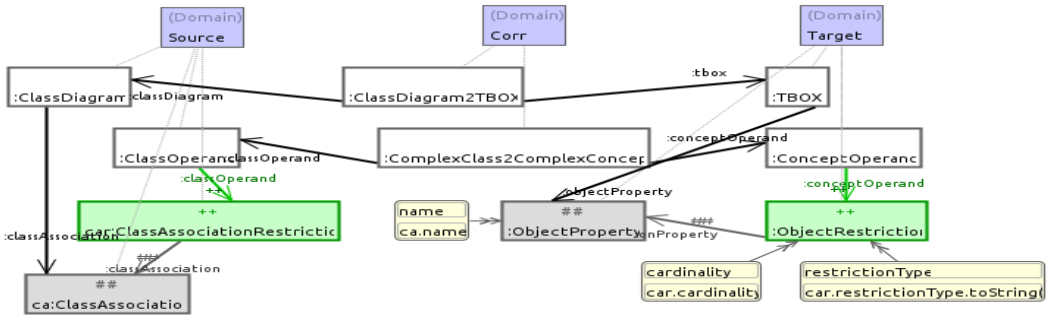


Figure 5. 52 : Restriction on class association (Restriction de l'association de classes).

- **Règle 27 Restriction on data association :** DataAssociationRestriction qui lient entre une ClassOperand et une DataAssociation génère DataRestriction qui lient entre DataProperty et ConceptOperand. DataProperty, selon le contexte peut exprimer une restriction existentielle, universelle ou de cardinalité. S'il s'agit d'une restriction de cardinalité, elle peut être de cardinalité minimale, maximale ou exacte avec un nombre de cardinalité de type naturel. Cette règle de transformation est illustrée dans la figure 5.53.

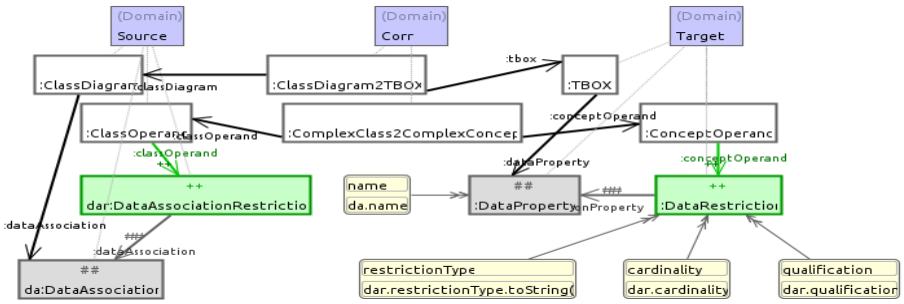


Figure 5. 53 : Restriction on data association (Restriction de l'association de données).

- **Régle 28 Restriction qualification :** Une restriction peut avoir une qualification, qui est une expression de classe (concept). La figure 5.54 montre sa transformation en règle TGG.

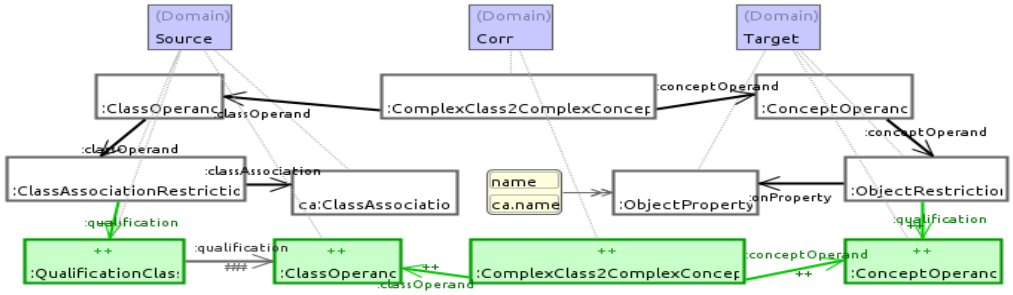


Figure 5. 54 : Restriction qualification (Qualification de restriction).

La correspondance entre les éléments des contextes contenant des instances (objets) dans le modèle conceptuel (paquet Ecore de "ObjectDiagram") et le modèle OWL2 (méta-modèle Ecore défini dans le paquet "ABOX") est définie par les règles TGG suivantes.

- **Régle 29 Creation of the context "ABOX" :** La règle TGG de la Figure 5.55 est définie pour créer le contexte de la boîte d'assertions "ABOX" à partir du contexte des instances du modèle conceptuel. Cette règle de transformation génére ABOX dans le domaine cible ONTOLOGY pour chaque ObjectPropertyDiagramm du domaine source UMLDiagrams.

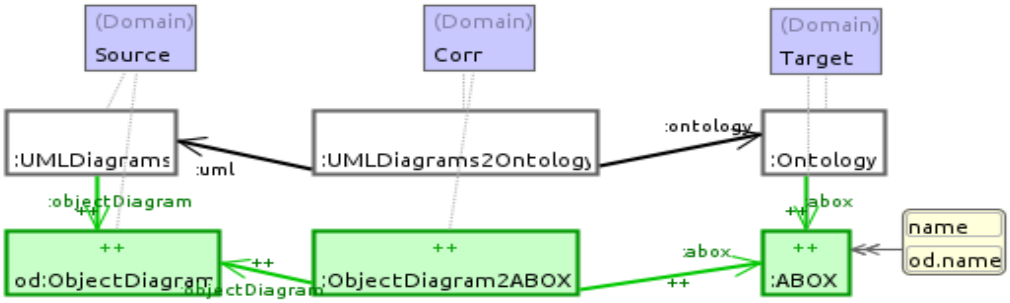


Figure 5. 55 : Creation of the context "ABOX" (Création du contexte "ABOX").

- **Règle 30 Creation of individuals** :Dans ce contexte, nous pouvons produire des éléments individuels "Individual" à partir de l'élément objet ;en appliquant la règle TGG de la figure 5.56 (chaque objet (object)génére un individu (individual).

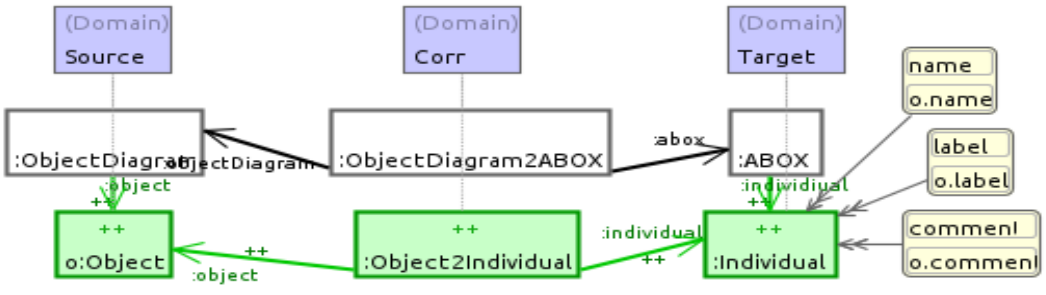


***Figure 5. 56** : Creation of individuals (Création des individus).*

- **Régle 31 Individual types :** La règle TGG de la figure 5.57 produit les types (classes) des individus créés.ou ClassInstance génére TypeAssertion qui lient un Concept à un Individual.

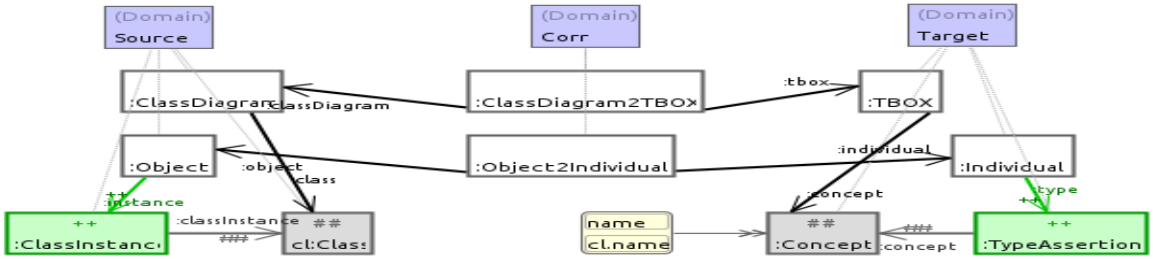


Figure 5. 57 : Individual types (Types d'individus).

Les instances de propriétés d'objets et de données à partir d'instances d'associations de classes et de données peuvent être produites par les règles TGG de la figure 5.58 et de la figure 5.59, respectivement.

- **Régle 32 Instances of object properties :** cette règle de transformation (Consulter la figure 5.58) génère les instances de propriétés d'objets (ObjectPropertyAssertion) qui lient les individues (Individual(s)) et ObjectProperty assertion à partir de ClassAssociationInstance qui lient les objets et ClassAssociation.

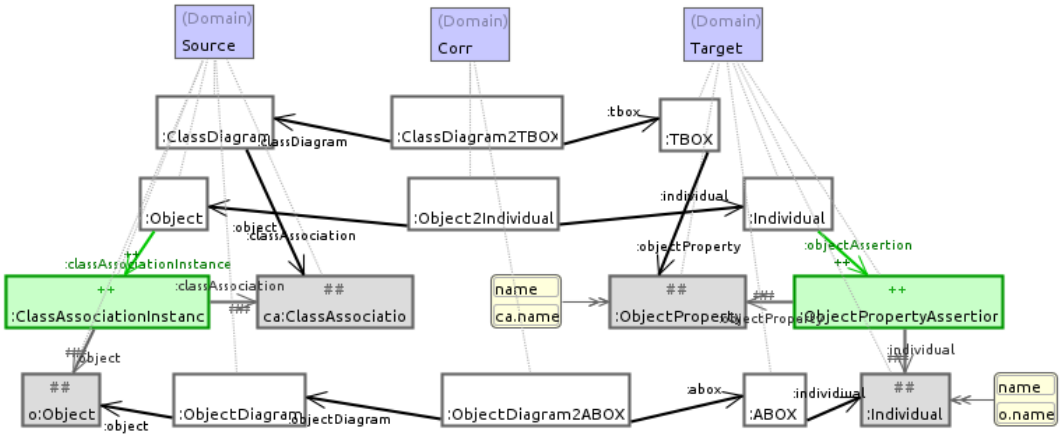


Figure 5. 58 : Instances of object properties (Instances de propriétés d'objets).

- **Règle 33 Instances of data properties :** cette règle de transformation (Consulter la figure 5.59) génère les instances de propriétés de données (DataPropertyAssertion) qui lient les individues (Individual(s)) et DataPropertyAssertion à partir de DataAssociationInstance qui lient les objets et DataAssociation.

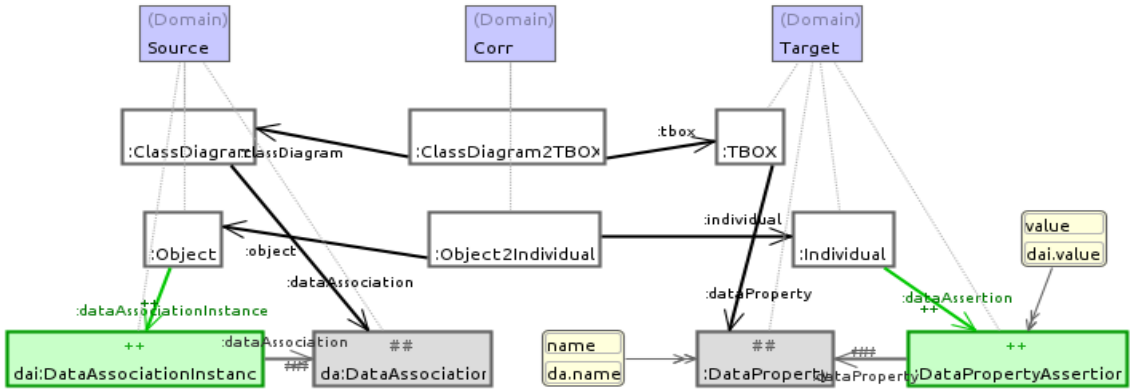


Figure 5. 59 : Instances of data properties (Instances des propriétés de données).

Les figure 5.60 et 5.61 présentent successivement les règles TGG permettant de produire une correspondance entre les éléments pour l'instance de négation des propriétés d'objet et de données.

- **Régle 34 Instances negation of objects** : cette règle de transformation (Consulter la figure 5.60) génère la négation des instances de propriétés d'objets (NegativeObjectPropertyAssertion) qui lient les individues (Individual(s)) et ObjectProperty assertion à partir de NegativeClassAssociationInstance qui lient les objets et ClassAssociation. Cette règle de transformation exprime la négation de propriétés d'objets.

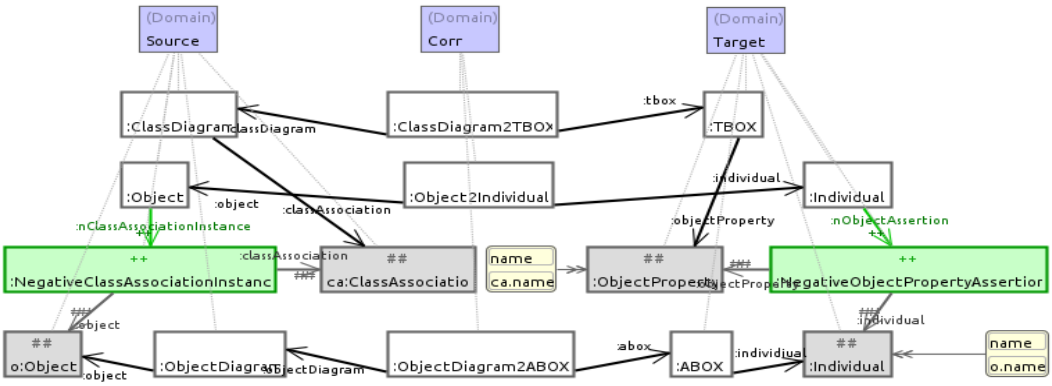


Figure 5. 60 : Instances negation of objects (Instances de négation d'objets).

- **Régle 35 Instances of data properties :** cette règle de transformation (consulter la figure 5.61) génère les instances de propriétés de données (NegativeDataPropertyAssertion) qui lient les individues (Individual(s)) et DataProperty assertion à partir de NegativeDataAssociationInstance qui lient les objets et DataAssociation. Cette règle de transformation exprime la négation de propriétés de données.

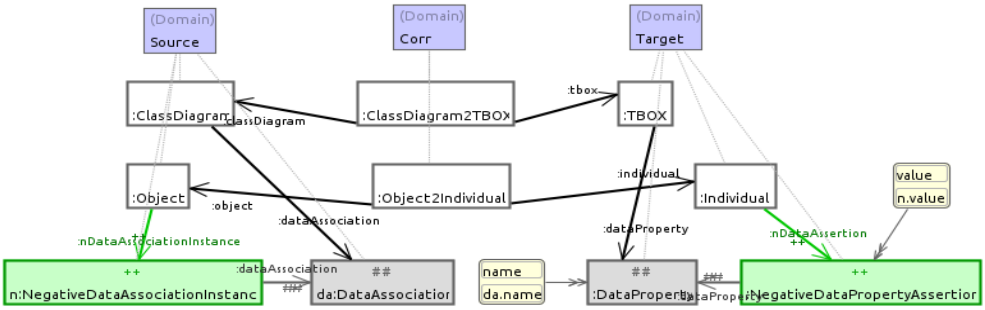


Figure 5. 61 : Instances negation of data (Instances de négation d'objets).

Les deux règles TGG suivantes sont utilisées pour mettre en correspondance des éléments pour des assertions sur le fait que deux objets (individus) sont identiques (figure 5.62) ou différents (figure 5.63).

- **Régle 36 Same instances :** cette règle de transformation (voir la figure 5.62) transforme La classe SameInstance qui lie les deux objets (2objects) en classe (SameAssertion) qui lie les deux individus(02Individuals).

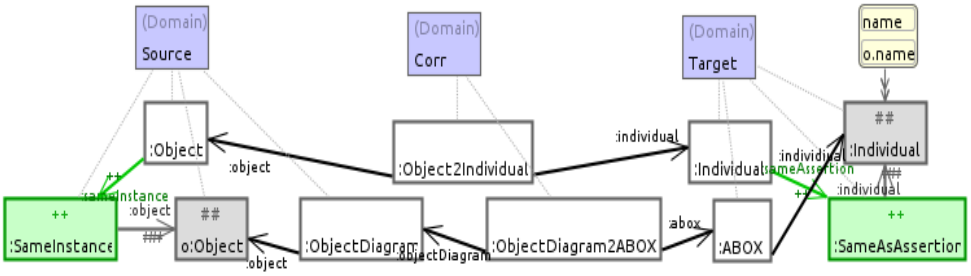


Figure 5. 62 : Same instances (Les mêmes instances).

- **Régle 37 Different instances :** cette règle de transformation (voir la figure 5.63) convertit la classe DifferentInstance qui lie les deux objets (2objects), en classe (DifferentFromAssertion) qui lie les deux individus(02Individuals).

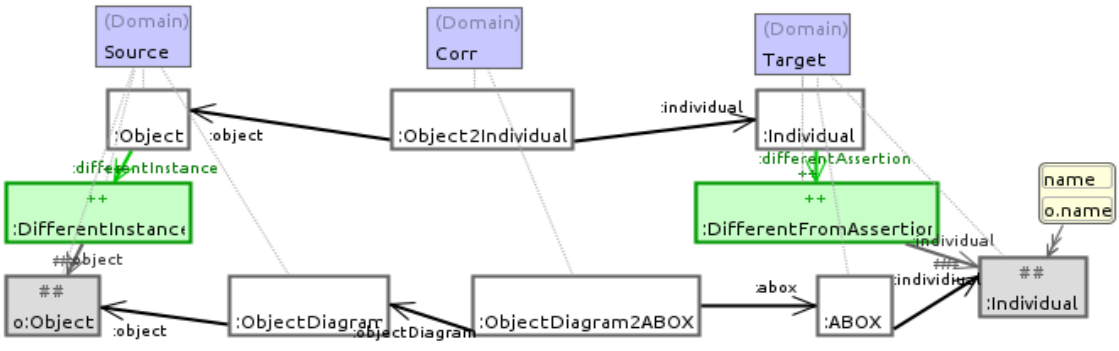


Figure 5. 63 : Different instances (Différentes instances).

Chaque règle TGG est enregistrée dans un fichier en tant que ressource de règle. Toutes ces ressources sont référencées par des noms de règles créés dans un fichier TGG qui seront utilisés pour la transformation de modèle à modèle. Par exemple, après validation du méta-modèle de correspondance par rapport au méta-modèle Ecore et validation de l'ensemble des règles TGG par rapport au langage TGG, nous pouvons traduire le modèle de la figure Fig. 5.20 en modèle de la figure Fig. 5.24 en utilisant l'interpréteur TGG.

5.3.3 TRASFORMATION MODEL-TO-TEXT (M2T)

Les transformations de modèles en texte (M2T) sont généralement utilisées pour gérer du code à partir d'un (ou plusieurs) modèles. Mais elles peuvent également être utilisées pour générer d'autres artefacts textuels, tels que de la documentation, des spécifications formelles à des fins de vérification ou pour sérialiser un modèle dans un format d'échange (ex. XMI).

Dans le contexte de la transformation model-to-text (génération de code), l'utilisation d'une transformation aurait pour objectif de transformer un modèle de haut-niveau d'abstraction en un second modèle de plus bas niveau d'abstraction (proche du code exécuté). De cette manière, on facilite la conception en épargnant à l'utilisateur les détails d'implémentation, en traduisant automatiquement ce modèle. Nous faisons appel pour cela à des transformations de type M2T que nous appliquons à des modèles UML afin de générer le code OWL2 équivalent et compatible aux diagrammes UML (diagramme de classe et les diagrammes d'objets).

Dans notre approche (Boudia et Bourahla 2022), nous proposons une transformation M2T, qui est une transformation verticale et exogène, (La transformation exogène est une transformation entre des modèles exprimés dans des langages de modélisation différents) et verticale (la transformation verticale, qui est une transformation où les modèles source et cible résident à des niveaux d'abstraction différents.). En utilisant Xpand (Efftinge, et al. 2004 - 2014), qui est un langage spécialisé dans la génération de code basé sur des modèles , EMF est utilisée pour la transformation de modèles.

Le projet Model to Text (M2T) se concentre sur la génération d'artefacts textuels à partir de modèles. Une grande classe de transformations traduit les modèles en texte. Le texte peut être du code généré, d'autres modèles dans la syntaxe textuelle ou d'autres objets textuels tels que des rapports ou des documentations. Le code est généralement généré à l'aide de modèles, une technique extrêmement bien établie (par exemple dans le développement Web). Un modèle peut être considéré comme le code cible avec des trous pour les parties variables. Les trous contiennent du méta-code (donc du code créant du code), qui est exécuté au moment de l'instanciation du

modèle pour calculer les parties variables. Un modèle est écrit avec le langage de modèle de Xpand.

Un modèle doit importer les méta-modèles de oWL, TBOXModel et ABOXModel définis et ajouter quelques contrôles de contraintes. Ainsi, un modèle en entrée du générateur de code aura une structure conforme à ces méta-modèles. Le code généré respecte le format fonctionnel de la syntaxe RDF/XML. Les éléments du modèle source correspondent à des fragments de code arbitraires.

Le module de modèle principal, présenté à la Figure 5.64, crée un fichier ontologique dont le nom est celui de l'ontologie donnée dans le modèle conceptuel avec le préfixe «. owl". Le générateur écrit dans ce fichier les en-têtes de l'ontologie, puis il appelle les modules de template pour l'expansion des déclarations de l'ontologie, des axiomes de classe, des axiomes d'objet, des axiomes de données et des individus nommés.

```
«IMPORT oWL»
«IMPORT TBOXModel»
«IMPORT ABOXModel»
«EXTENSION template::GeneratorExtensions»

«DEFINE main FOR Ontology»
«FILE name+".owl"»
  Prefix(:=<http://www.ontology.org/«name».owl#>)
  Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
  Prefix(rdf:=<http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
  Prefix(xml:=<http://www.w3.org/XML/1998/namespace>)
  Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
  Prefix(rdfs:=<http://www.w3.org/2000/01/rdf-schema#>)

  Ontology(<http://www.ontology.org/«name».owl>
  «IF label != null»«"Annotation(rdfs:label "+ "\""+label+"\")"»«ENDIF»
  «IF comment != null»«"Annotation(rdfs:comment "+ "\""+comment+"\")"»«ENDIF»

  «EXPAND declarations FOR this-»
  «EXPAND class_axioms FOR this-»
  «EXPAND object_axioms FOR this-»
  «EXPAND data_axioms FOR this-»
  «EXPAND named_individuals FOR this-»)
«ENDFILE»
«ENDDEFINE»
```

Figure 5. 64: le module principal de Xpand.

L'extension des déclarations de l'ontologie peut être réalisée à l'aide du module modèle nommé "déclarations" dans la Figure 5.65. Nous avons des déclarations de classes (concepts), de propriétés d'objets et de propriétés de données, qui sont déclarées dans le modèle TBOX. Les axiomes de classe (concept) sont générés avec le module de modèle nommé "class_axioms" dans

la Figure 5.66. Pour chaque concept dans le modèle TBOX, nous avons les axiomes : étiquettes, commentaires, annotations, disjoint avec, équivalent à et sous-éléments de concept.

```
«DEFINE declarations FOR Ontology»
#########################################################################
#   Declarations of classes, object properties and data properties
#########################################################################
Declaration(Class(owl:Thing))
«FOREACH tbox.concept AS c»
  Declaration(Class(:«c.name»))
«ENDFOREACH-»
«FOREACH tbox.objectProperty AS o»
  Declaration(ObjectProperty(:«o.name»))
«ENDFOREACH-»
«FOREACH tbox.dataProperty AS d»
  Declaration(DataProperty(:«d.name»))
«ENDFOREACH-»
«ENDDEFINE»
```

Figure 5. 65 : Declarations (les déclarations).

```
«DEFINE class_axioms FOR Ontology»
######################
#   Axioms for Classes
######################
# Class: owl:Thing (owl : Thing)
AnnotationAssertion(rdfs:label owl:Thing "owl : Thing")
«FOREACH tbox.concept AS c»«"# Class: :"+c.name+" ("+c.name+")"»
«IF c.label !=   null»
«"AnnotationAssertion(rdfs:label :"+c.name+" \""+c.label+"\")"»
«ENDIF-»
«IF c.comment != null»
«"AnnotationAssertio
(rdfs:comment :"+c.name+"   \""+c.comment+"\")"»
«ENDIF-»
«FOREACH c.disjointWith.concept AS e»«"DisjointClasses(:"+c.name+" "»
«EXPAND concept_expression(e)»«")"»
«ENDFOREACH-»
«FOREACH c.equivalentTo.concept AS e-»«"EquivalentClasses(:"+c.name+" "»
«EXPAND concept_expression(e)»«")"»
«ENDFOREACH-»
«FOREACH c.subConcept.concept AS e»«"SubClassOf(:"+c.name+" "»
«EXPAND concept_expression(e)»«")"»
«ENDFOREACH-»
«ENDFOREACH»
«ENDDEFINE»
```

Figure 5. 66 : Class axiomes (les axiomes des Classes).

Le code OWL2 pour les axiomes de propriété d'objet définis dans le modèle TBOX est généré avec le module d'extension nommé "objet_axioms" tel qu'il est décrit dans la Figure 5.67. Il contient des extensions pour les axiomes de domaine et du range, les caractéristiques des axiomes (fonctionnels, fonctionnels inverses, symétriques, asymétriques, réflexifs, irréflexifs) et les axiomes d'équivalence, d'inclusion, de disjonction et de super. De la même manière, le module

Xpand "data_axioms" de la Figure 5.68, est utilisé pour produire le code OWL2 des axiomes TBOX concernant les différentes propriétés des données de l'ontologie.

```
«DEFINE object_axioms FOR Ontology»
################################
#   Axioms for Object Properties
################################
«FOREACH tbox.objectProperty AS o»
  «"# Object Property: :"+o.name+" (:"+o.name+")"»
  «IF o.label != null»
    «"AnnotationAssertion(rdfs:label :"+o.name+" \""+o.label+"\")"»
  «ENDIF-»
  «IF o.comment != null»
    «"AnnotationAssertion(rdfs:comment :"+o.name+" \""+o.comment+"\")"»
  «ENDIF-»
  «FOREACH o.domain.domain AS e»
    «"ObjectPropertyDomain(:"+o.name+" "»
    «EXPAND concept_expression(e)»«")"»
  «ENDFOREACH-»
  «FOREACH o.range.range AS e-»
    «"ObjectPropertyRange(:"+o.name+" "»
    «EXPAND concept_expression(e)»«")"»
  «ENDFOREACH-»
  «IF o.functional.toString() == "true"»
    «"FunctionalObjectProperty(:"+o.name+")"»
  «ENDIF-»
  «IF o.inverseFunctional.toString() == "true"»
    «"InverseFunctionalObjectProperty(:"+o.name+")"»
  «ENDIF-»
  «IF o.irreflexive.toString() == "true"»
    «"IrreflexiveObjectProperty(:"+o.name+")"»
  «ENDIF-»
  «IF o.reflexive.toString() == "true"»
    «"ReflexiveObjectProperty(:"+o.name+")"»
  «ENDIF-»
  «IF o.symmetric.toString() == "true"»
    «"SymmetricObjectProperty(:"+o.name+")"»
  «ENDIF-»
  «IF o.asymmetric.toString() == "true"»
    «"AsymmetricObjectProperty(:"+o.name+")"»
  «ENDIF-»
  «IF o.transitive.toString() == "true"»
    «"TransitiveObjectProperty(:"+o.name+")"»
  «ENDIF-»
  «FOREACH o.inverse.property AS e»
    «"InverseObjectProperties(:"+o.name+" :"+e.name+")"»
  «ENDFOREACH-»
  «FOREACH o.super.property AS e»
    «"SuperObjectProperty(:"+o.name+" :"+e.name+")"»
  «ENDFOREACH-»
  «FOREACH o.disjoint.property AS e»
    «"DisjointObjectProperties(:"+o.name+" :"+e.name+")"»
  «ENDFOREACH-»
  «FOREACH o.equivalent.property AS e»
    «"EquivalentObjectProperties(:"+o.name+" :"+e.name+")"»
  «ENDFOREACH-»
  «FOREACH o.sub.properties AS e»
    «"SubObjectPropertyOf(:"+o.name+"  :"+e.name+")"»
  «ENDFOREACH-»
«ENDFOREACH»
«ENDDEFINE»
```

Figure 5. 67 : Object property axioms (les axiomes Object propriétés).

```
«DEFINE named_individuals FOR Ontology»
############################
#   Named Individuals
############################
«FOREACH abox.individiual AS i»
  «"# Individual: :"+i.name+" (:"+i.name+")"»
  «IF i.label != null»
    «"AnnotationAssertion(rdfs:label :"+i.name+" \""+i.label+"\")"»
  «ENDIF-»
  «IF i.comment != null»
    «"AnnotationAssertion(rdfs:comment :"+i.name+
      " \""+i.comment+"\")"»
  «ENDIF-»
  «FOREACH i.type.concept AS e»
    «"ClassAssertion(:"+e.name+" :"+i.name+")"»
  «ENDFOREACH-»
  «FOREACH i.objectAssertion AS e»
    «"ObjectPropertyAssertion(:"+e.objectProperty.name +
      " :"+i.name+" :"+e.individual.first().name+")"»
  «ENDFOREACH-»
  «FOREACH i.nObjectAssertion AS e»
    «"NegativeObjectPropertyAssertion(:"+e.objectProperty.name
      +" :"+i.name+" :"+e.individual.first().name+")"»
  «ENDFOREACH-»
  «FOREACH i.dataAssertion AS e»
    «"DataPropertyAssertion(:"+e.dataProperty.name +
      " :"+i.name+" \""+e.value+"\"^^"+
      e.dataProperty.range.range.first().toString()+")"»
  «ENDFOREACH-»
  «FOREACH i.nDataAssertion AS e»
    «"NegativeDataPropertyAssertion(:"+e.dataProperty.name +
      " :"+i.name+" \""+e.va lue+"\"^^"+
      e.dataProperty.range.range.first().toString()+")"»
  «ENDFOREACH-»
  «FOREACH i.sameAssertion AS e»
    «FOREACH e.individual AS s»
      «"SameIndividual(:"+i.name+" :"+s.name+")"»
    «ENDFOREACH»
  «ENDFOREACH-»
  «FOREACH i.differentAssertion AS e»
    «FOREACH e.individual AS s»
      «"DifferentIndividuals(:"+i.name+" :"+s.name+")"»
    «ENDFOREACH»
  «ENDFOREACH-»
«ENDFOREACH»
«ENDDEFINE»
```

Figure 5. 68 : Data property axioms (les axiomes de propriétés de données).

Les expansions pour les axiomes d'assertion définis dans le modèle ABOX sont réalisées par le module Xpand « named_individuals". Il existe des expansions pour les assertions sur les types d'individus, les propriétés des objets (données) en tant que relations entre deux individus (individu et données), la négation des assertions sur les propriétés des objets (données) et les assertions sur l'indication si les individus sont identiques ou différents. Le module Xpand nommé "concept expression" dans les figures 5.69, 5.70 sont utilisés pour développer les extensions des axiomes d'assertion et les expressions de concept.

```
«DEFINE named_individuals FOR Ontology»
############################
#   Named Individuals
############################
«FOREACH abox.individiual AS i»
  «"# Individual: :"+i.name+" (:"+i.name+")"»
  «IF i.label != null»
    «"AnnotationAssertion(rdfs:label :"+i.name+" \""+i.label+"\")"»
  «ENDIF-»
  «IF i.comment != null»
    «"AnnotationAssertion(rdfs:comment :"+i.name+
      " \""+i.comment+"\")"»
  «ENDIF-»
  «FOREACH i.type.concept AS e»
    «"ClassAssertion(:"+e.name+" :"+i.name+")"»
  «ENDFOREACH-»
  «FOREACH i.objectAssertion AS e»
    «"ObjectPropertyAssertion(:"+e.objectProperty.name +
      " :"+i.name+" :"+e.individual.first().name+")"»
  «ENDFOREACH-»
  «FOREACH i.nObjectAssertion AS e»
    «"NegativeObjectPropertyAssertion(:"+e.objectProperty.name
      +" :"+i.name+" :"+e.individual.first().name+")"»
  «ENDFOREACH-»
  «FOREACH i.dataAssertion AS e»
    «"DataPropertyAssertion(:"+e.dataProperty.name +
      " :"+i.name+" \""+e.value+"\"^^"+
      e.dataProperty.range.range.first().toString()+")"»
  «ENDFOREACH-»
  «FOREACH i.nDataAssertion AS e»
    «"NegativeDataPropertyAssertion(:"+e.dataProperty.name +
      " :"+i.name+" \""+e.va lue+"\"^^"+
      e.dataProperty.range.range.first().toString()+")"»
  «ENDFOREACH-»
  «FOREACH i.sameAssertion AS e»
    «FOREACH e.individual AS s»
      «"SameIndividual(:"+i.name+" :"+s.name+")"»
    «ENDFOREACH»
  «ENDFOREACH-»
  «FOREACH i.differentAssertion AS e»
    «FOREACH e.individual AS s»
      «"DifferentIndividuals(:"+i.name+" :"+s.name+")"»
    «ENDFOREACH»
  «ENDFOREACH-»
«ENDFOREACH»
«ENDDEFINE»
```

Figure 5. 69 : Expansion of assertion axioms (Expansion des axiomes d'assertion).

```
«DEFINE concept_expression(ConceptOperand c) FOR Ontology-»
 «IF c.eContents.typeSelect(
   TBOXModel::AtomicConcept).first() != null»
  «":"+c.eContents.typeSelect(
    TBOXModel::AtomicConcept).concept.first().name-»
 «ENDIF-»
 «IF c.eContents.typeSelect(
   TBOXModel::ConceptComplement).first() != null-»
   «"ObjectComplementOf("»
   «EXPANDconcept_expression(
    c.eContents.typeSelect(
       TBOXModel::ConceptComplement).first().
       concept)»«")"-»
 «ENDIF-»
 «IF c.eContents.typeSelect(
    TBOXModel::ConceptIntersection).first() != null-»
   «"ObjectIntersectionOf("»
   «EXPAND concept_expression(c.eContents.
      typeSelect(TBOXModel::ConceptIntersection).
      first().leftConcept)»«" "»
   «EXPAND concept_expression(c.eContents.
     typeSelect(TBOXModel::ConceptIntersection).
     first().rightConcept)»«")"-»
 «ENDIF-»
 «IF c.eContents.typeSelect(
    TBOXModel::ConceptUnion).first() != null»
   «"ObjectUnionOf("»
   «EXPAND concept_expression(c.eContents.
      typeSelect(TBOXModel::ConceptUnion).
     first().leftConcept)»«" "»
   «EXPAND concept_expression(c.eContents.
     typeSelect(TBOXModel::ConceptUnion).
     first().rightConcept)»«")"-»
 «ENDIF-»
 «IF c.eContents.typeSelect(
    TBOXModel::ObjectRestriction).first() != null»
   «"Object"+c.eContents.
      typeSelect(TBOXModel::ObjectRestriction).
      restrictionType.first().toString()+"ValuesFrom("»
   «":"+c.eContents.typeSelect(
     TBOXModel::ObjectRestriction).onProperty.
     first().name+" "»
   «EXPAND concept_expression(c.eContents.
      typeSelect(TBOXModel::ObjectRestriction).
      first().qualification)»«")"-»
 «ENDIF-»
```

Figure 5. 70 : Extension of concept expressions (Extension des expressions de concepts).

5.4 Conclusion

Dans ce chapitre, Nous avons proposé une approche dirigée par les modèles pour la transformation de modèles basée sur l'utilisation des grammaires de graphes. Plus formellement, nous avons proposé une approche MDA (Model Driven Architecture) pour la formalisation et la transformation des ontologies afin de générer le code OWL2 équivalent.

Cette approche (Boudia et Bourahla 2022) consiste en trois processus : la méta-modélisation, la transformation modèle à modèle, la transformation modèle à texte.

L'implantation de notre approche nécessite l'utilisation d'environnements dédiés tels que la plateforme Eclipse Modeling Framework (EMF). Cette plateforme nous a permis d'implémenter les méta-modèles, les modèles et les transformations. Ensuite, nous avons utilisé l'outil TGG Interpreter, intégré dans l'environnement EMF, pour créer les règles de transformation TGG. La phase finale de la génération de code est réalisée à l'aide du langage Xpand.

CHAPITRE 6

Étude de cas et évaluation de performance

Sommaire

6.1 Introduction

Dans ce chapitre, nous tenterons d'appliquer notre approche (Boudia et Bourahla 2022) présentée dans le chapitre précédent sur les ontologies formalisées afin de générer le code OWL 2 équivalent.

Cette approche consiste en la génération automatique de descriptions formelles de modèles conceptuels spécifiques à une ontologie, elle comporte ainsi trois processus principaux.

Le premier processus est la méta-modélisation, qui consiste à concevoir le méta modèle pour la source (diagrammes UML), méta modèles de destination (les ontologies OWL2) et celui de correspondance entre elles.

Le deuxième processus est la transformation de modèle à modèle, qui est une transformation horizontale et endogène, où la méthode TGG est utilisée pour la transformation du modèle. Pour cela, nous élaborons trois méta-modèles de base pour les modèles conceptuels spécifiques à l'ontologie : les méta-modèles UML (méta-modèles source), les méta-modèles OWL2 (les modèles cibles), et enfin les modèles de correspondance, qui assurent le mappage des éléments des modèles sources vers les modèles cibles.

Le troisième processus est la transformation de modèle en texte (code), qui est une transformation verticale et exogène, où la méthode Xpand (Xpand Documentation, 2020), qui est un langage spécialisé dans la génération de code basé sur des modèles EMF est utilisée pour la transformation de modèle. Pour cela, nous utilisons les modèles OWL2 produits par le premier processus par rapport à son méta-modèle Ecore comme source de cette transformation de modèle. La cible est le code OWL2 (texte) par rapport aux modèles Xpand (représentant son méta-modèle Xpand.

Le présent chapitre récapitulera l'étude de cas ainsi que l'implémentation et l'évaluation de notre approche.

6.2 Étude de cas

Pour exécuter notre approche nous avons utilisé les outils suivants :

1) Eclipse: eclipse-modeling-2021-03-R-win32-x86_64.zip
2) TGG (Help -> Install New Software)TGG - https://svn-serv.cs.upb.de/updatesites/trunk/de.upb.swt.qvt.tgg-updatesite/3.7/
3) Xpand: Xpand/Xtend Build Site http://download.eclipse.org/modeling/m2t/xpand/updates/nightly/
4) MWE:MWE Repository - http://download.eclipse.org/modeling/emft/mwe/updates/nightly/

L'exécution de cette approche nécessite deux phases principales :

La transformation modèle vers modèle : Pour cela, nous créons d'abord la configuration de l'interpréteur TGG en sélectionnant les fichiers contenant le modèle source, les règles TGG et les trois méta-modèles (source, correspondance et cible). Dans la deuxième étape, nous effectuons la transformation TGG en sélectionnant le fichier de configuration créé, un modèle cible est généré comme un fichier XMI qui peut être montré avec l'éditeur de modélisation Ecore. L'interpréteur TGG nous donne le nombre de règles appliquées (197 règles TGG, pour cet exemple) et un temps de transformation est calculé (812 ms, pour cette cas) (Consulter la figure 6.4). Nous avons choisi comme études de cas le fameux exemple family ontology (Consulter la figure 6.1). Les figures suivantes 6.2, 6.3, et 6.4 présentent le modèle source (family.uml), le modèle destination (résultat de transformation (family.owl), ainsi que l'ensemble de règles exécutées. (Contenu de fichier intermédiaire), et la figure 6.5 illustre le fichier intermédiaire family.corr.xmi.

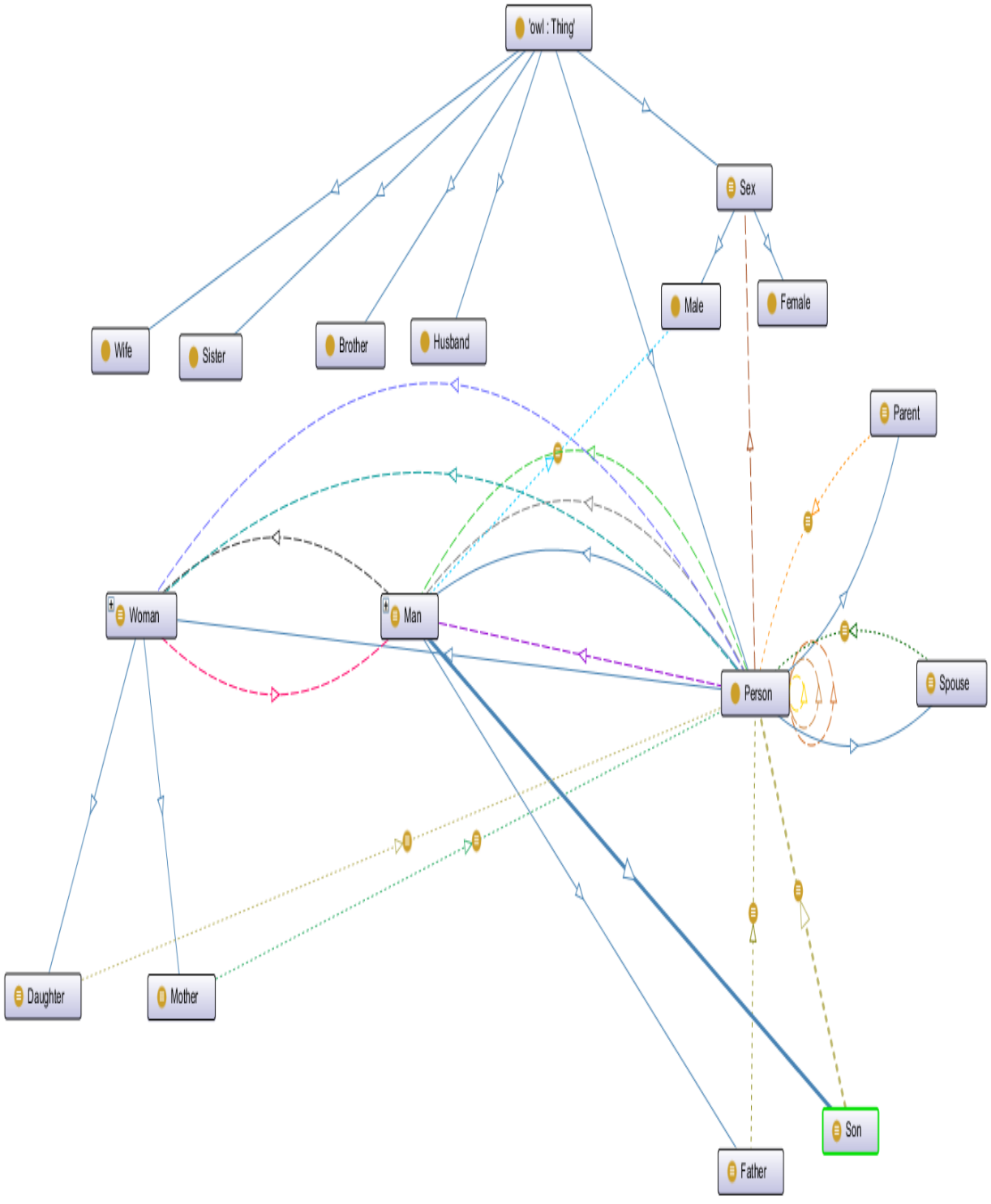


Figure 6. 1 : Family Ontology

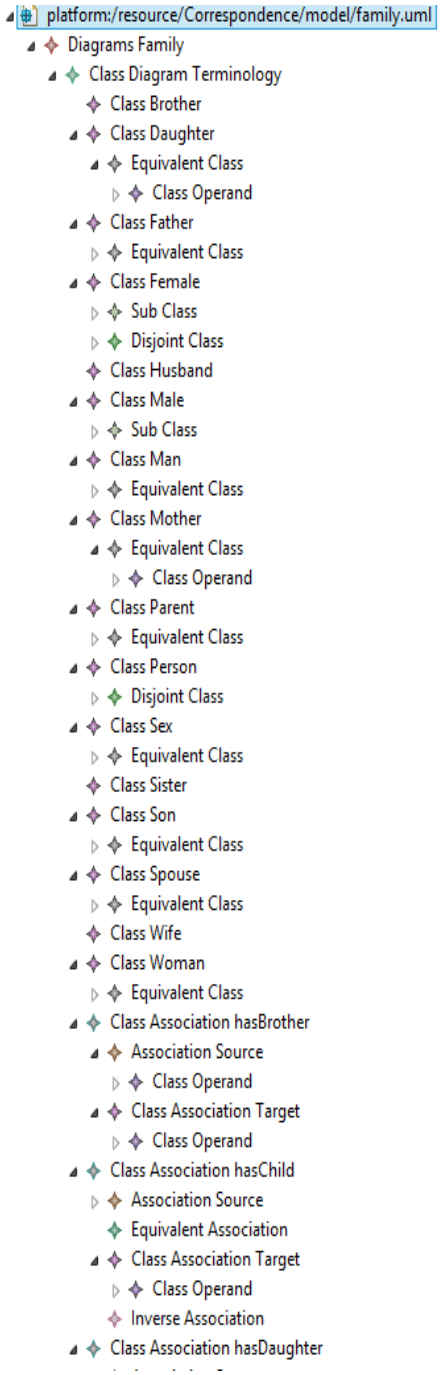


Figure 6. 2 : Modèle EMF pour la source « diagrams family »

Figure 6. 3: Modèle EMF pour la destination « ontology family ».

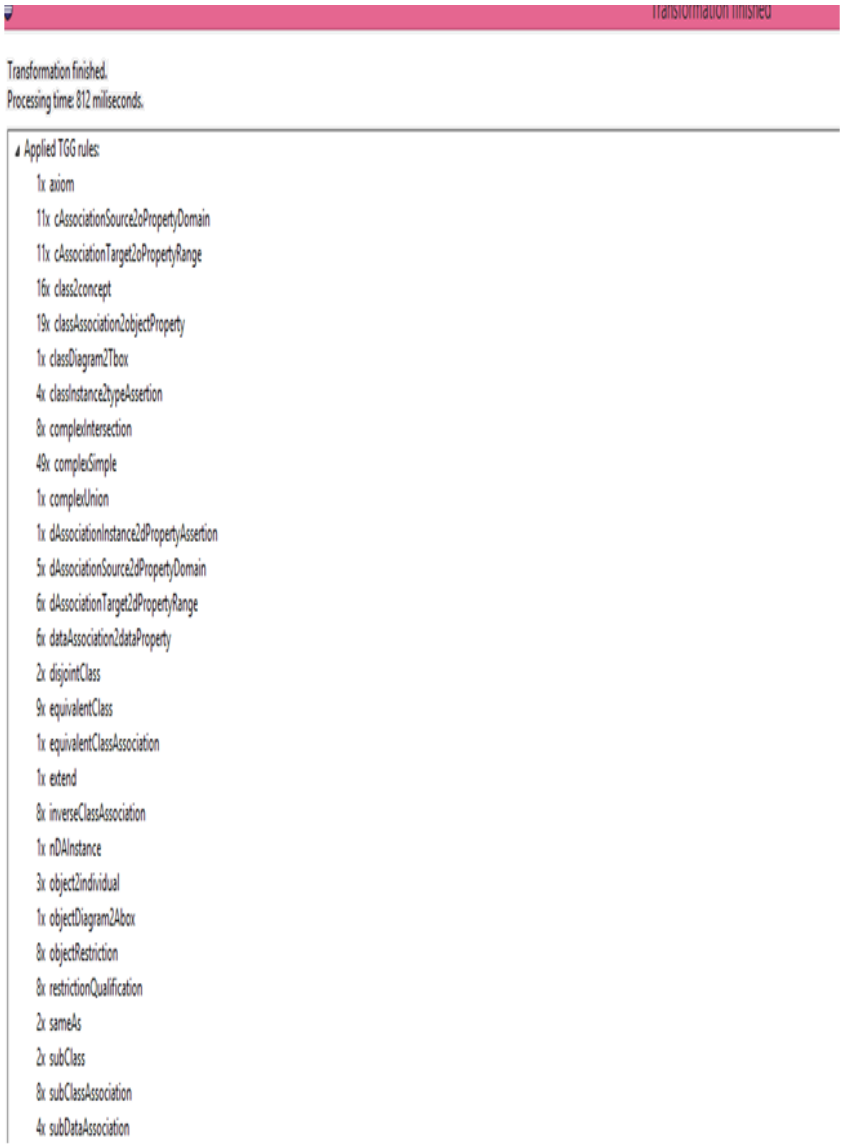


Figure 6. 4 : Les règles de transformation exécutées (pour chaque règle exécutée ; le nombre d'exécution est mentionné).

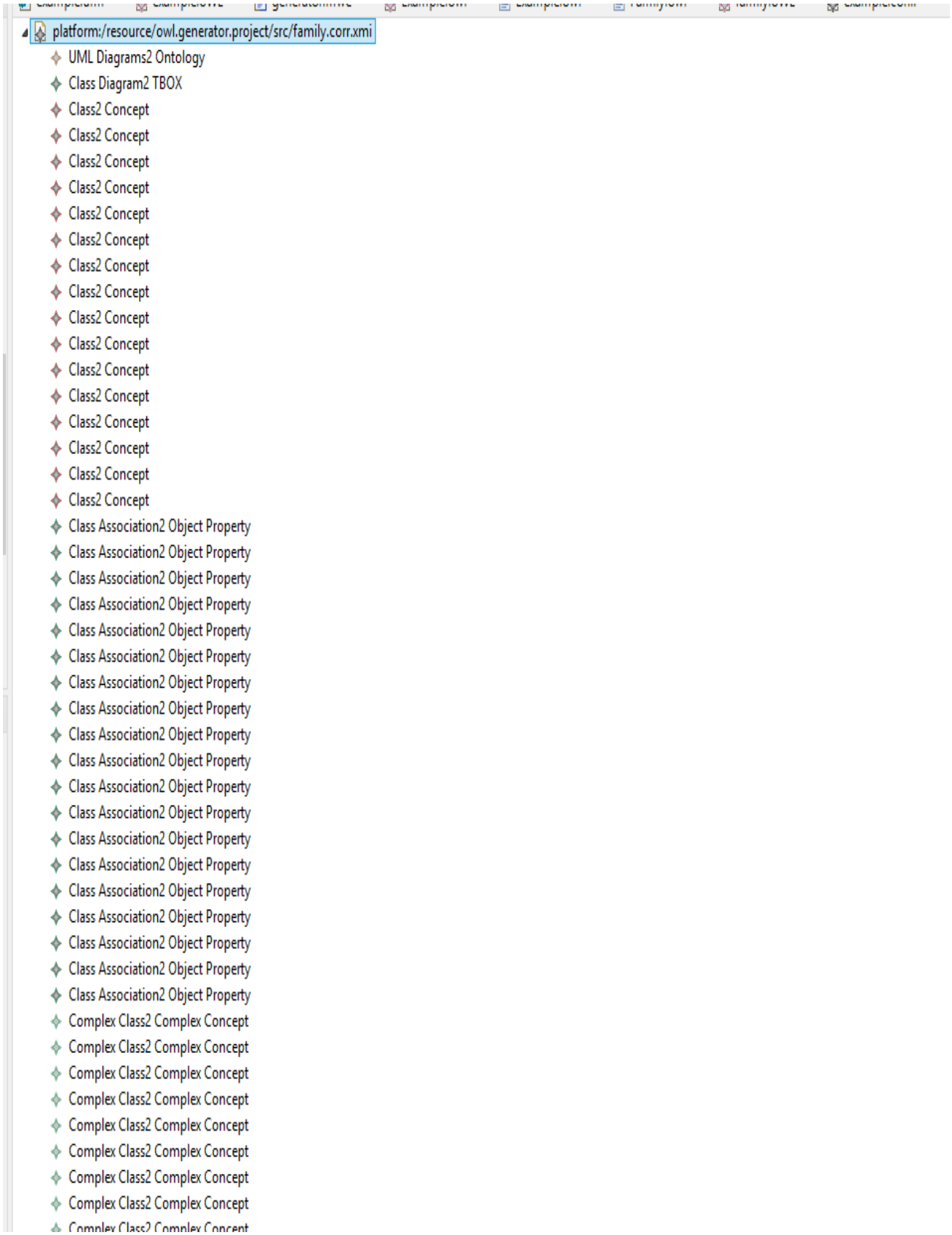


Figure 6. 5 : Le fichier intermédiaire family.corr.xmi

Après avoir modélisé notre exemple diagrammes family (les diagrammes de classes, les diagrammes objet) nous sommes en mesure d'automatiser la génération du modèle destination ontologie family (ABOX, TBOX), en exécutant la transformation bidirectionnelle TGG. Le modèle présenté à la figure 6.2 est utilisé comme fichier de modèle source. Deux fichiers sont générés en tant que modèles de sortie, fichier intermédiaire «. xmi » (voir figure 6.5) et le fichier de modèle cible avec l'extension « .owl » (voir figure 6.3). L'exécution de la transformation résultante est illustrée à la figure 6.4. Tous les éléments du modèle source ont été transformés. Selon notre contexte d'exemple, la règle class2Concept a été appliquée seize (16) fois et la règle calssassociation2objectporoperty dix-neuf (19) fois comme illustrée dans la figure 6.4.

La transformation modèle vers texte : est une transformation modèle-code, qui prend un modèle OWL2 et génère un code au format fonctionnel OWL2/XML. Cette étape peut être accomplie en appelant le modèle Xpand depuis un workflow. Après avoir exécuté le workflow, le code généré doit être acceptable par le langage OWL2. Par exemple, le modèle OWL2 Ecore de la Figure 6.3 est transformé en code OWL2 suivant (Figure 6.6) avec ce générateur de code en un temps de 379 ms.

```
Prefix(:=<http://www.ontology.org/Family.owl#>)
Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
Prefix(rdf:=<http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
Prefix(xml:=<http://www.w3.org/XML/1998/namespace>)
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(rdfs:=<http://www.w3.org/2000/01/rdf-schema#>)

Ontology(<http://www.ontology.org/Family.owl>

#################################################################
#   Declarations of classes, object properties and data properties
#################################################################
Declaration(Class(owl:Thing))
Declaration(Class(:Brother))
Declaration(Class(:Daughter))
Declaration(Class(:Father))
Declaration(Class(:Female))
Declaration(Class(:Husband))
Declaration(Class(:Male))
Declaration(Class(:Man))
Declaration(Class(:Mother))
Declaration(Class(:Parent))
Declaration(Class(:Person))
Declaration(Class(:Sex))
Declaration(Class(:Sister))
Declaration(Class(:Son))
Declaration(Class(:Spouse))
Declaration(Class(:Wife))
Declaration(Class(:Woman))
Declaration(ObjectProperty(:hasBrother))
Declaration(ObjectProperty(:hasChild))
Declaration(ObjectProperty(:hasDaughter))
Declaration(ObjectProperty(:hasFather))
Declaration(ObjectProperty(:hasHusband))
```

```
Declaration(ObjectProperty(:hasMother))
Declaration(ObjectProperty(:hasParent))
Declaration(ObjectProperty(:hasSex))
Declaration(ObjectProperty(:hasSon))
Declaration(ObjectProperty(:hasWife))
Declaration(ObjectProperty(:isChildOf))
Declaration(ObjectProperty(:isDaughterOf))
Declaration(ObjectProperty(:isFatherOf))
Declaration(ObjectProperty(:isHusbandOf))
Declaration(ObjectProperty(:isMotherOf))
Declaration(ObjectProperty(:isParentOf))
Declaration(ObjectProperty(:isSonOf))
Declaration(ObjectProperty(:isSpouseOf))
Declaration(ObjectProperty(:isWifeOf))
Declaration(DataProperty(:hasBirthYear))
Declaration(DataProperty(:hasDeathYear))
Declaration(DataProperty(:hasEventYear))
Declaration(DataProperty(:hasFamilyName))
Declaration(DataProperty(:hasFirstGivenName))
Declaration(DataProperty(:hasName))

######################
#   Axioms for Classes
######################
# Class: owl:Thing (owl : Thing)
AnnotationAssertion(rdfs:label owl:Thing "owl : Thing")
# Class: :Brother (Brother)
AnnotationAssertion(rdfs:label :Brother "Brother")
# Class: :Daughter (Daughter)
AnnotationAssertion(rdfs:label :Daughter "Daughter")
EquivalentClasses(:Daughter ObjectIntersectionOf(:Woman
ObjectSomeValuesFrom(:hasParent :Person)))
# Class: :Father (Father)
AnnotationAssertion(rdfs:label :Father "Father")
EquivalentClasses(:Father ObjectIntersectionOf(:Man
ObjectSomeValuesFrom(:isFatherOf :Person)))
# Class: :Female (Female)
AnnotationAssertion(rdfs:label :Female "Female")
DisjointClasses(:Female :Male)
SubClassOf(:Female :Sex)
# Class: :Husband (Husband)
AnnotationAssertion(rdfs:label :Husband "Husband")
# Class: :Male (Male)
AnnotationAssertion(rdfs:label :Male "Male")
SubClassOf(:Male :Sex)
# Class: :Man (Man)
AnnotationAssertion(rdfs:label :Man "Man")
EquivalentClasses(:Man ObjectIntersectionOf(:Person
ObjectSomeValuesFrom(:hasSex :Male)))
# Class: :Mother (Mother)
AnnotationAssertion(rdfs:label :Mother "Mother")
EquivalentClasses(:Mother ObjectIntersectionOf(:Woman
ObjectSomeValuesFrom(:isMotherOf :Person)))
# Class: :Parent (Parent)
AnnotationAssertion(rdfs:label :Parent "Parent")
EquivalentClasses(:Parent ObjectIntersectionOf(:Person
ObjectSomeValuesFrom(:isParentOf :Person)))
# Class: :Person (Person)
AnnotationAssertion(rdfs:label :Person "Person")
DisjointClasses(:Person :Sex)
# Class: :Sex (Sex)
AnnotationAssertion(rdfs:label :Sex "Sex")
EquivalentClasses(:Sex ObjectUnionOf(:Female :Male))
```

```
# Class: :Sister (Sister)
AnnotationAssertion(rdfs:label :Sister "Sister")
# Class: :Son (Son)
AnnotationAssertion(rdfs:label :Son "Son")
EquivalentClasses(:Son ObjectIntersectionOf(:Man
ObjectSomeValuesFrom(:hasParent :Person)))
# Class: :Spouse (Spouse)
AnnotationAssertion(rdfs:label :Spouse "Spouse")
EquivalentClasses(:Spouse ObjectIntersectionOf(:Person
ObjectSomeValuesFrom(:isSpouseOf :Person)))
# Class: :Wife (Wife)
AnnotationAssertion(rdfs:label :Wife "Wife")
# Class: :Woman (Woman)
AnnotationAssertion(rdfs:label :Woman "Woman")
EquivalentClasses(:Woman ObjectIntersectionOf(:Person
ObjectSomeValuesFrom(:hasSex :Female)))


################################
#   Axioms for Object Properties
################################
# Object Property: :hasBrother (:hasBrother)
ObjectPropertyDomain(:hasBrother :Person)
ObjectPropertyRange(:hasBrother :Man)
# Object Property: :hasChild (:hasChild)
ObjectPropertyDomain(:hasChild :Person)
ObjectPropertyRange(:hasChild :Person)
InverseObjectProperties(:hasChild :isChildOf)
EquivalentObjectProperties(:hasChild :isParentOf)
# Object Property: :hasDaughter (:hasDaughter)
ObjectPropertyDomain(:hasDaughter :Person)
ObjectPropertyRange(:hasDaughter :Woman)
InverseObjectProperties(:hasDaughter :isDaughterOf)
SubObjectPropertyOf(:hasDaughter :hasChild)
# Object Property: :hasFather (:hasFather)
ObjectPropertyDomain(:hasFather :Person)
ObjectPropertyRange(:hasFather :Man)
FunctionalObjectProperty(:hasFather)
InverseObjectProperties(:hasFather :isFatherOf)
SubObjectPropertyOf(:hasFather :hasParent)
# Object Property: :hasHusband (:hasHusband)
ObjectPropertyDomain(:hasHusband :Woman)
ObjectPropertyRange(:hasHusband :Man)
InverseObjectProperties(:hasHusband :isHusbandOf)
SubObjectPropertyOf(:hasHusband :isSonOf)
# Object Property: :hasMother (:hasMother)
ObjectPropertyDomain(:hasMother :Person)
ObjectPropertyRange(:hasMother :Woman)
FunctionalObjectProperty(:hasMother)
InverseObjectProperties(:hasMother :isMotherOf)
SubObjectPropertyOf(:hasMother :hasParent)
# Object Property: :hasParent (:hasParent)
ObjectPropertyDomain(:hasParent :Person)
ObjectPropertyRange(:hasParent :Person)
InverseObjectProperties(:hasParent :isParentOf)
# Object Property: :hasSex (:hasSex)
ObjectPropertyDomain(:hasSex :Person)
ObjectPropertyRange(:hasSex :Sex)
FunctionalObjectProperty(:hasSex)
# Object Property: :hasSon (:hasSon)
ObjectPropertyDomain(:hasSon :Person)
ObjectPropertyRange(:hasSon :Man)
InverseObjectProperties(:hasSon :isSonOf)
```

```
SubObjectPropertyOf(:hasSon :hasChild)
# Object Property: :hasWife (:hasWife)
ObjectPropertyDomain(:hasWife :Man)
ObjectPropertyRange(:hasWife :Woman)
InverseObjectProperties(:hasWife :isWifeOf)
SubObjectPropertyOf(:hasWife :isSpouseOf)
# Object Property: :isChildOf (:isChildOf)
# Object Property: :isDaughterOf (:isDaughterOf)
SubObjectPropertyOf(:isDaughterOf :hasParent)
# Object Property: :isFatherOf (:isFatherOf)
# Object Property: :isHusbandOf (:isHusbandOf)
# Object Property: :isMotherOf (:isMotherOf)
# Object Property: :isParentOf (:isParentOf)
# Object Property: :isSonOf (:isSonOf)
SubObjectPropertyOf(:isSonOf :hasParent)
# Object Property: :isSpouseOf (:isSpouseOf)
ObjectPropertyDomain(:isSpouseOf :Person)
ObjectPropertyRange(:isSpouseOf :Person)
SymmetricObjectProperty(:isSpouseOf)
# Object Property: :isWifeOf (:isWifeOf)


################################
#   Axioms for Data Properties
################################
# Data Property: :hasBirthYear (:hasBirthYear)
DataPropertyDomain(:hasBirthYear :Person)
DataPropertyRange(:hasBirthYear xsd:int)
FunctionalDataProperty(:hasBirthYear)
SubDataPropertyOf(:hasBirthYear :hasEventYear)
# Data Property: :hasDeathYear (:hasDeathYear)
DataPropertyDomain(:hasDeathYear :Person)
DataPropertyRange(:hasDeathYear xsd:int)
FunctionalDataProperty(:hasDeathYear)
SubDataPropertyOf(:hasDeathYear :hasEventYear)
# Data Property: :hasEventYear (:hasEventYear)
DataPropertyRange(:hasEventYear xsd:int)
# Data Property: :hasFamilyName (:hasFamilyName)
DataPropertyDomain(:hasFamilyName :Person)
DataPropertyRange(:hasFamilyName xsd:string)
FunctionalDataProperty(:hasFamilyName)
SubDataPropertyOf(:hasFamilyName :hasName)
# Data Property: :hasFirstGivenName (:hasFirstGivenName)
DataPropertyDomain(:hasFirstGivenName :Person)
DataPropertyRange(:hasFirstGivenName xsd:string)
FunctionalDataProperty(:hasFirstGivenName)
SubDataPropertyOf(:hasFirstGivenName :hasName)
# Data Property: :hasName (:hasName)
DataPropertyDomain(:hasName :Person)
DataPropertyRange(:hasName xsd:string)


############################
#   Named Individuals
############################
# Individual: :Alice (:Alice)
AnnotationAssertion(rdfs:label :Alice "Alice")
ClassAssertion(:Person :Alice)
ObjectPropertyAssertion(:hasBrother :Alice :Bob)
NegativeObjectPropertyAssertion(:hasFather :Alice :Bob)
DataPropertyAssertion(:hasFamilyName :Alice "aalice"^^xsd:string)
NegativeDataPropertyAssertion(:hasFamilyName :Alice "nnn"^^xsd:string)
DifferentIndividuals(:Alice :Boby)
```

```
DifferentIndividuals(:Alice :Bob)
# Individual: :Bob (:Bob)
AnnotationAssertion(rdfs:label :Bob "Bob")
ClassAssertion(:Person :Bob)
ClassAssertion(:Man :Bob)
SameIndividual(:Bob :Boby)
SameIndividual(:Bob :Alice)
# Individual: :Boby (:Boby)
AnnotationAssertion(rdfs:label :Boby "Boby")
ClassAssertion(:Person :Boby)

)
```

Figure 6. 6 : Code OWL2 généré automatiquement(family.owl) (format fonctionnel RDF/XML)

6.3 Implémentation

L'implémentation combine deux tâches principales, la transformation de modèle à modèle et la transformation de modèle à texte (code). La transformation modèle-modèle consiste à définir : un méta-modèle pour la conceptualisation de l'ontologie, un méta-modèle pour la description de l'ontologie basée sur OWL2/XML, un méta-modèle pour la correspondance entre les méta-modèles source et destination, et une grammaire de transformation basée sur les TGG pour traduire le modèle conceptuel de l'ontologie en modèle de description OWL2/XML. La transformation du modèle en code consiste à définir le modèle Xpand par rapport au méta-modèle OWL2 et la syntaxe RDF/XML par rapport au format fonctionnel.

Le projet Eclipse de transformation de modèle à modèle est développé pour intégrer tous ces méta-modèles avec les règles TGG pour réaliser la génération de modèles cibles à partir des modèles sources édités. L'EMF (Eclipse Modeling Framework) ECore (Steinberg, 2008) de la plateforme Eclipse est utilisé pour l'implémentation de cette transformation automatique. Le méta-modèle Ecore) de EMF Eclipse est utilisé pour définir les modèles et il fournit un support d'exécution pour la manipulation des modèles du cadre EMF de base, comme pour générer des interfaces pour éditer des objets, qui séparent le développement de l'application de l'implémentation des classes.

Ainsi, la couche de méta-modélisation de l'outil EMF est utilisée pour modéliser graphiquement les méta-modèles des diagrammes conceptuels (classes et objets), qui représentent le méta-modèle source de la conceptualisation de l'ontologie ("UMLdiagrammes") et le méta-modèle cible de l'ontologie OWL2 ("TBOX" et "ABOX"). Le méta-modèle de correspondance est ajouté pour connecter les méta-modèles source et cible. Après la validation de tous les méta-modèles par rapport à la syntaxe du méta-modèle EMF Ecore, le code Java est généré à partir du modèle EMF Generator pour compléter l'implémentation du projet pour les codes du modèle et de l'éditeur.

L'interpréteur TGG est utilisé pour générer des modèles d'ontologie basés sur OWL2, en exécutant des règles de réécriture de graphes basées sur les méta-modèles source, cible et de correspondance. Ces règles sont spécifiées pour effectuer des transformations de modèle à modèle (M2M) en utilisant la technique TGG. Ces transformations sont réalisées entre des graphes modélisés avec Eclipse Modeling Framework (EMF). Ainsi, pour générer les modèles d'ontologie basés sur OWL2 avec l'interpréteur TGG, nous commençons par écrire des règles de transformation (TGG) en utilisant TGG, qui sont sensibles aux méta-modèles définis. Ensuite, on écrit des déclarations avec le langage de contrainte d'objet (OCL) afin d'assigner des valeurs pour des contraintes spécifiques et après la validation des règles, les règles TGG seront renforcées par des relations de généralisation.

Le projet Eclipse de transformation de modèle en code (Xpand generator) intègre le modèle Xpand et le méta-modèle OWL2 pour générer une description formelle OWL2/XML de l'ontologie à partir de son modèle OWL2. Xpand est un moteur de modèle pour générer du code à partir de modèles EMF (Documentation Xpand, 2020). Il utilise le moteur de workflow de modélisation (MWE) pour exécuter différents composants de modélisation dans Eclipse, et de manière autonome des workflows configurés avec un langage déclaratif basé sur XML (Eclipse Modeling M2T, 2020).

Le projet Xpand fournit le runtime pour exécuter les flux de travail et l'outil IDE pour les éditer. Le modèle Xpand est un code cible contenant des trous comme parties variables. Ces trous contiennent du méta-code, c'est-à-dire du code créant du code. Les parties variables sont calculées au moment de l'instanciation du modèle. Dans ce cas, nous utilisons le langage Xpand pour créer des modèles et ajouter des contraintes de vérification pour la génération du code OWL2 à partir des modèles EMF Ecore de l'ontologie OWL2 générée par l'interpréteur TGG en utilisant les règles TGG pour la transformation de modèle à modèle des modèles conceptuels de l'ontologie.

6.4 Evaluation des performances

L'évaluation des performances de ce processus de formalisation de la conceptualisation d'ontologies est basée sur l'évaluation de sa scalabilité sur des données du Web sémantique de grande taille, ce qui devrait engager des ontologies sémantiquement riches et de exactes. Par conséquent, l'évaluation mesurera les compromis entre la scalabilité et l'exactitude de la formalisation.

L'évaluation des performances est basée sur un ensemble de métriques. La première métrique est le nombre de règles TGG appliquées pour effectuer la transformation de modèle à modèle. La deuxième métrique est le temps passé par cette transformation (la somme des temps passés par les transformations modèle à modèle et modèle à code), et la troisième métrique est l'exactitude de la transformation (sa complétude et sa solidité).

Une transformation est complète si elle génère toutes les entités (correctes) composant l'ontologie. Le degré de complétude d'une transformation est mesuré comme le pourcentage d'entités ontologiques générées. Le pourcentage d'entités ontologiques générées par la transformation qui compose réellement l'ontologie est utilisé pour déterminer le degré de solidité d'une transformation. Par exemple, si l'ontologie est composée de N entités (selon le benchmark, toutes correctes) et que la transformation a généré M entités (correctes et non correctes), si nous avons K entités correctes parmi les M entités générées, alors la complétude est (K/N)*100% et la solidité est (1-((M-K)/M))*100%.

En plus de ces métriques, il y a une métrique combinée pour la performance de la formalisation pour mesurer le temps de formalisation et son exactitude en combinaison pour aider à évaluer le compromis potentiel entre le temps de formalisation, la capacité de formalisation, et la performance globale de la formalisation de la conceptualisation.

Pour effectuer ces mesures de performance, de nombreuses expériences doivent être réalisées sur des ontologies populaires et bien établies, qui ont été utilisées dans les précédents benchmarks. Le benchmark choisi est le Lehigh University Benchmark (LUBM) (Guo, Pan et Heflin October 2005), qui possède une ontologie sur le domaine universitaire et les données de test (ABoxes). Afin d'évaluer la capacité de la formalisation de la conceptualisation de l'ontologie à gérer de grandes ABoxes, nous devons être en mesure de faire varier la taille des données (assertions sur les instances), et voir comment cela évolue.

L'ontologie de l'Université est exprimée en OWL Lite, qui est un sous-langage de OWL2. L'ontologie décrit 43 concepts, 32 propriétés composées de 25 propriétés d'objet et de 7 propriétés de type de données. Elle utilise les caractéristiques du langage OWL Lite, notamment : l'inverse de, la propriété transitive, certaines valeurs des restrictions (existentielles), l'intersection de, où 263 axiomes sont décrits, qui sont composés de 93 axiomes logiques, 44 axiomes d'inclusion de concepts, 5 axiomes d'inclusion de propriétés, 2 axiomes inverses, 1 axiome transitif, 25 axiomes de domaine, 18 axiomes d'étendue et 75 axiomes d'assertion d'annotation.

Un générateur appelé UBA (Univ-Bench Artificial data generator), est développé pour générer des données de test, qui peuvent être mises à l'échelle à une taille arbitraire, où une université est l'unité minimale de génération de données. Pour chaque université, un ensemble de fichiers OWL décrivant ses départements est généré. Des paramètres permettent au générateur UBA de spécifier le nombre et le type d'universités à générer. Les données de test créées par le générateur UBA sont également décrites avec OWL Lite. Chaque ontologie utilise un ensemble de données différent avec une taille ABox croissante. Dans un ensemble de données LUBM, chaque université contient 15 à 25 départements ; chacun est décrit par un fichier OWL distinct.

Nous avons créé 9 jeux test de donnéesDS_k^{Org}, qui contiennent des fichiers OWL pour k universités. Chaque jeu de données DS_k^{Org} a un nombre de départements compris entre 15 et 25, où chaque département universitaire est décrit par un fichier OWL. Ces ensembles de données sont l'entrée d'un autre outil, que nous avons développé pour générer des modèles conceptuels Ecore par rapport au méta-modèle conceptuel, et ils seront effectivement utilisés pour évaluer la performance (performance). Nous générons pour chaque donnée de test originale DS_k^{Org} un modèle conceptuel Ecore DS_k^{Con}, qui sera utilisé comme entrée pour les transformations combinées modèle à modèle et modèle à code, pour générer l'ensemble de donnéesDS_k^{Gen}. L'exactitude est mesurée en calculant la différence entreDS_k^{Org} etDS_k^{Gen}.

Les tests ont été effectués sur un ordinateur portable équipé d'un processeur Intel(R) Core (TM) i5-4210U à 1,70 GHz et de 8 Go de RAM, sous Windows XP Service Pack 1 (64 bits). Ces expériences ont pour but de voir si nous pouvons gérer ces ensembles de données à grande échelle. Le tableau 6.1 montre la taille en méga-octets, le nombre de règles appliquées et le temps de transformation de chaque ensemble de données. Les résultats montrent que l'outil a pu traiter une université de 24 départements en 176,52 minutes. Cependant, pour 25 départements, il a passé plus de 180 minutes (3 heures), ce qui est le temps limite. Ainsi, il montre une bonne scalabilité dans la transformation des données par rapport au chargement d'ontologie testé avec le même benchmark (Guo, Pan et Heflin October 2005). Cette forte scalabilité est due au gain d'espace dans le processus de transformation et à la manière dont les règles de grammaire sont réécrites.

#	$S(DS_k^{Org})$ (MB)	$S(DS_k^{Con})$ (MB)	# Règles TGG appliqués	Temps de génération (mn)	$S(DS_k^{Gen})$ (MB)	Complétude ($C_{DS_k^{Con}}$)	Solidité (Exactitude) ($S_{DS_k^{Con}}$)
1	8.43	8.09	103481	1.02	8.42	100%	99%
2	19.39	18.78	237524	2.33	19.35	98%	97%
3	28.44	26.95	348417	8.62	28.39	100%	91%
4	40.37	35.74	494152	20.81	40.31	95%	91%
5	52.80	47.81	646143	74.78	52.74	96%	92%
6	61.11	54.92	748020	108.28	60.94	96%	89%
7	74.76	64.73	914826	149.93	73.98	93%	92%
8	84.67	71.52	1036201	176.52	82.87	91%	88%

Tableau 6.1 : Résultats expérimentaux.

Le tableau 6.1 montre la comparaison de la complétude des ensembles de données pour voir quelle ontologie de test de référence peut être générée par la procédure de transformation et ensuite tester sa capacité de transformation. Les résultats montrent que la transformation pourrait générer presque complètement toutes les entités de l'ontologie (presque toutes les entités dansDS_k^{Org} $sont$ dansDS_k^{Gen}). De plus, le tableau 6.1 montre la solidité de la procédure de transformation pour chaque ensemble de données. On a découvert qu'elle transforme presque parfaitement tous les jeux de données (toutes les entités dans DS_k^{Gen} sont dansDS_k^{Org}).

En examinant ces résultats, nous avons constaté que les ontologies originales contenaient des informations qui n'ont pas été prises en compte parce qu'elles étaient initialement considérées comme des informations inutiles. Pour cette raison, la formalisation n'est pas complètement complète et solide.

D'après le tableau 6.1, l'outil montre qu'il y a une augmentation proportionnelle du nombre de règles appliquées et du temps de transformation lorsque la taille des données augmente. Il est important d'évaluer la performance globale de l'outil en combinant les résultats des métriques ci-dessus. La métrique $F_{DS_k^{Con}}$ est utilisée pour calculer le compromis entre la complétude et la

solidité de la transformation, puisqu'elles sont essentiellement analogues au rappel et à la précision dans la recherche d'information, respectivement.

$$F_{DS_k^{Con}} = \frac{(\beta^2 + 1) \times C_{DS_k^{Con}} \times S_{DS_k^{Con}}}{\beta^2 \times C_{DS_k^{Con}} + S_{DS_k^{Con}}}$$

Dans la formule ci-dessus, $C_{DS_k^{Con}}$ and $S_{DS_k^{Con}}$ ($\in [0,1]$) sont la Complétude de la transformation et la solidité (Exactitude) pour l'ensemble de données DS_k^{Con}, respectivement. Ainsi, cette formule est utilisée pour déterminer la pondération relative entre $S_{DS_k^{Con}}$ et$C_{DS_k^{Con}}$. En plus de calculer la performance de la transformation, nous utilisons la métrique, pour le jeu de données $P_{DS_k^{Con}}$ pour le jeu de donnéesDS_k^{Con}.

$$.P_{DS_k^{Con}} = \frac{1}{1+e^{\gamma \times T_{DS_k^{Con}/N_k - \delta}}}$$

Dans la formule, $T_{DS_k^{Con}}$ est le temps de transformation T (ms) pour le jeu de données DS_k^{Con} (somme des temps dépensé par la transformation de modèle à modèle et de modèle à code). Le nombre N_k est le nombre total de règles TGG appliquées pour transformer le modèle conceptuel de DS_k^{Con}. Pour permettre la comparaison des valeurs métriques sur des ensembles de données de différentes tailles, nous utilisons le temps de transformation par règle TGG (c'est-à-dire$T_{DS_k^{Con}}/N_k$) dans le calcul. Pour distinguer les résultats des expériences sur différents jeux de données, deux paramètres de contrôle γ (appelé le décalage) et δ (appelé la pente) sont utilisés. Leurs valeurs sont choisies pour rendre $P_{DS_k^{Con}}$ (la transformation rapide par rapport à Nk) assez proche d'un (supérieur à 0,99). Pour ces expériences, γ = 2 et δ = 20.

La métrique de performance globale OP récompense la procédure de transformation lorsqu'elle produit des entités plus rapidement, plus complètement et plus solidement et elle est définie comme une métrique composite de temps de transformation, de complétude et de solidité pour chaque jeu de données DS_k^{Con}, où k = 1, ⋯, 8 comme dans ce qui suit.

$$OP_{DS_k^{Con}} = w_{DS_k^{Con}} \times \frac{(\alpha^2 + 1) \times P_{DS_k^{Con}} \times F_{DS_k^{Con}}}{\alpha^2 \times P_{DS_k^{Con}} + F_{DS_k^{Con}}}$$

Où D est le nombre total de jeux de données de test ($D = 8$), $w_{DS_k^{Con}}$ ($\Sigma_{i=1}^{D} w_{DS_i^{Con}} = 1$) est le poids donné au jeu de données DS_k^{Con} qui détermine la pondération relative entre $F_{DS_k^{Con}}$ et $P_{DS_k^{Con}}$.

La figure 6.7 montre les résultats de la métrique de performance globale OP de la procédure de transformation qui est calculée par rapport à chaque ensemble de données, où β et α sont tous deux fixés à 1, ce qui signifie que les poids de la complétude et de la solidité de la transformation sont égaux, et que le poids du temps de transformation par rapport au poids de la complétude et de la solidité sont également égaux. Le poids $w_{DS_k^{Con}}$ de DS_k^{Con} est calculé par la formule $w_{DS_k^{Con}} = N_k / \Sigma_{j=1}^{D} N_j$, où $k \in \{1, \cdots, D\}$ et N_k est le total des règles TGG appliquées pour transformer l'ensemble de données DS_k^{Con}.

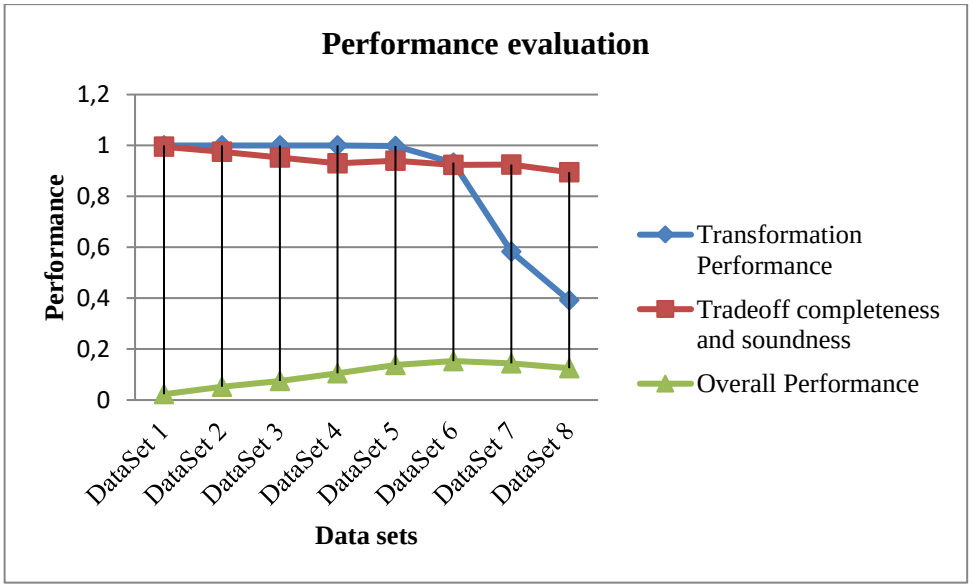


Figure 6. 7 : Evaluation des performances

La figure 6.7 montre que le compromis entre la complétude et la solidité de la transformation est supérieure à 90%, la performance de la transformation est de 100% pour les cinq premiers ensembles de données, puis elle diminue pour les derniers ensembles de données car ils ont des tailles plus importantes. La performance globale indique que le processus de transformation est très performant pour l'ensemble de données numéro 6 par rapport aux autres ensembles de données. Nous trouvons que ces résultats numériques montrés dans la Figure 6.7 sont utiles pour apprécier la performance globale de transformation de notre approche.

6.5 Conclusion

Dans ce chapitre, nous avons appliqué l'approche MDA (Model Driven Architecture) basée sur les TGG (Triple Graph Grammars) pour la transformation des ontologies formalisées (Boudia et Bourahla 2022). Cette approche implique la génération automatique de descriptions formelles de modèles conceptuels spécifiques à l'ontologie, dont l'objectif est de générer le code OWL2 équivalent. Elle se compose donc de trois processus principaux. Le premier processus, la métamodélisation, consiste à concevoir des métamodèles : le métamodèle source (diagramme UML), le métamodèle cible (ontologie OWL2) et le métamodèle de correspondance entre eux. Le deuxième processus est la transformation de modèle à modèle(**M2M)**, c'est-à-dire la transformation horizontale et endogène. Le troisième processus est la transformation modèle-texte (code) (**M2T)**, qui est une transformation exogène verticale utilisant l'approche Xpand, un langage désigné pour générer du code basé sur des modèles EMF.

Nous avons implémenté cette approche sur l'ontologie "Family Ontology", exécutant 37 règles de la transformation en 812 millisecondes.

L'évaluation des performances de ce processus de conceptualisation d'ontologies formalisées est basée sur l'évaluation de sa scalabilité sur des données Web sémantiques à grande échelle, ce qui devrait impliquer des ontologies sémantiquement riches et précises. Ainsi, l'évaluation mesure le compromis entre l'évolutivité et l'exactitude formelle. L'évaluation des performances est basée sur un ensemble de mesures. La première métrique est le nombre de règles TGG appliquées pour effectuer une transformation de modèle à modèle. La deuxième métrique est le temps passé sur cette transformation (somme des transformations modèle à modèle et modèle à code). La troisième métrique est l'exactitude de la transformation (sa complétude et sa solidité).

Finalement, nous avons prouvé l'efficacité, l'exactitude (la complétude et la solidité) et la fiabilité de notre approche par une étude comparative des résultats expérimentaux.

CHAPITRE 7

Conclusion générale

Sommaire

7.1 Conclusion

Model-Driven Engineering (IDM) ou Model-Driven Engineering (MDE) en anglais, est une mise en œuvre de plusieurs améliorations significatives dans le développement de systèmes complexes, et permet de se concentrer sur des préoccupations plus abstraites que la programmation classique. C'est une forme d'ingénierie générative dans laquelle tout ou partie de l'application est générée à partir du modèle. Un modèle est une abstraction, une simplification du système, et cela suffit pour comprendre le système de modélisation, et répondre aux questions à son sujet. Un système peut être décrit par différents modèles qui sont liés les uns aux autres. L'idée principale est d'utiliser autant de langages de modélisation différents (Domain Specific Modeling Languages - DSML) que possible selon le temps ou les aspects techniques du développement du système. Exiger, par conséquent, la définition de ces DSMLs (appelée méta-modélisation), est une problématique clé de cette nouvelle approche. Par ailleurs, afin de rendre opérationnels les modèles (pour la génération de code, de documentation et de test, la validation, la vérification, l'exécution, etc.), une autre problématique clé est celle de la transformation de modèle.

Dans ce travail, nous avons proposé une approche MDA (Model Driven Architecture) pour la formulation automatique de la conceptualisation spécifique à l'ontologie. La conceptualisation de l'ontologie est éditée en tant que modèle par rapport au méta-modèle défini avec le langage de méta-modélisation EMF Ecore. Le modèle conceptuel sera d'abord transformé en modèle OWL2 en utilisant la technique TGG, où un méta-modèle OWL2 cible et un méta-modèle de correspondance sont définis avec le langage de méta-modélisation EMF Ecore, pour utiliser la technique de transformation TGG. Cette transformation de modèle à modèle utilise un ensemble de règles TGG définies avec le langage interprète TGG (Triple Graph Grammar). Ensuite, le modèle OWL produit sera transformé au format fonctionnel RDF/XML en utilisant la transformation modèle-code avec la technique de Xpand/MWE. Cette approche est testée de manière extensive avec un benchmark connu du domaine universitaire, qui contient de très grands

ensembles de données. Les résultats dévoilent sa complétude, sa solidité, sa fiabilité et son exactitude.

Dans cette approche, nous avons conçu un outil de modélisation et de formalisation des ontologies. Un éditeur graphique a été développé pour modéliser la conceptualisation des ontologies par rapport au métamodèle défini dans la norme Ecore, en utilisant les notations UML. Ecore est le méta-modèle fondamental d'EMF.Il permet l'expression d'autres modèles en s'appuyant sur ses constructions. L'environnement est défini en fonction de lui-même (son propre méta-modèle). Le résultat de la modélisation sera transformé en utilisant des grammaires de graphes triples dans un modèle ontologique par rapport à un méta-modèle ontologique OWL2 défini et développé avec Ecore. Cette transformation bidirectionnelle peut être réalisée en exécutant un ensemble de règles définies par rapport à un méta-modèle de correspondance entre le méta-modèle ontologique conceptuel, et le méta-modèle d'ontologie basé sur OWL2.Le modèle d'ontologie généré, basé sur OWL2, représente le résultat de la conceptualisation de l'ontologie et peut être formalisé en utilisant un langage de description basé sur la syntaxe de RDF/XML, avec un format spécifique comme format fonctionnel.

Dans ce travail, la transformation du modèle d'ontologie en texte (code) est utilisée pour générer cette description formelle, qui est développée en utilisant la technique Xpand. Considérant le nombre de personnes familiarisées avec les notations UML, cette solution sera une bonne approche de la modélisation de conceptualisation d'ontologies pour les développeurs ordinaires et de sémantiser (rendre sémantique) les diagrammes UML.

Dans cette approche de génération automatique de descriptions formelles de modèles conceptuels spécifiques à une ontologie, nous avons développé trois processus principaux.

- **Le premier processus**, la métamodélisation, consiste à concevoir un métamodèle. Un métamodèle source (diagramme UML), un métamodèle cible (ontologie OWL2), et un métamodèle de correspondance entre eux.
- **Le deuxième processus** est la transformation de modèle à modèle(M2M). Il s'agit d'une transformation horizontale et endogène, qui transforme le modèle selon la méthode TGG. Pour ce faire, nous définissons trois métamodèles de base de modèles conceptuels spécifiques à l'ontologie (modèles sources). Un modèle OWL2 (le modèle cible) et un modèle correspondant pour mapper les éléments des modèles source et cible.
- **Le troisième processus** est la conversion du modèle en texte (code)(M2T). Il s'agit d'une transformation verticale et exogène utilisant l'outil Xpand, un langage spécialisé pour générer du code basé sur des modèles EMF, pour la transformation de modèles. Pour cela, nous

utilisons le modèle OWL2 généré par le deuxième processus pour ce métamodèle Ecore comme source de cette transformation de modèle. La cible est le code OWL2 (texte) par rapport à un modèle Xpand (représentant un métamodèle Xpand).

Pour conclure nous pouvons citez les avantages de cette approche et les inconvénients :

Avantages :

Les avantages de notre approche :

- Les expérimentations réalisées dans cette approche justifient comment une approche MDA peut être réalisée dans la pratique.
- Elaboration et validation de métamodèle source (diagrammes UML) et de métamodèle destination (ontologyOWL2).
- Elaboration de métamodèle de correspondance qui assurent la transformation des éléments de métamodèle source (diagrammes UML) vers le métamodèle destination (ontologyOWL2).
- Effectuer les transformations bidirectionnelles(M2M) en utilisant TGG.
- Générer le code(M2T) équivalent au modèle de destination(ontologyOWL2).
- Sémantiser (rendre sémantiques) les spécifications UML.

Inconvénients :

Les désavantages de notre approche :

- Notre méthode est efficace, pertinente et fiable car elle offre une complétude et une solidité acceptables. Cependant, Ces critères d’efficacité, pertinence et fiabilité diminuent lorsque la taille de l'ontologie dépasse certains seuils.

7.2 Contributions

Les résultats de ce processus de recherche apparaissent dans l'article intitulé "**Formalization of Ontology Conceptualizations using Model Transformation**", publié dans le journal international "**Journal international de modélisation et de conception de systèmes d'information (IJISMD)** (Boudia et Bourahla 2022).

BOUDIA, M., & BOURAHLA, M. (2022). Formalization of Ontology Conceptualizations using Model Transformation. International Journal of Information System Modeling and Design (IJISMD)"., Volume 13, issue1.

7.3 Les perspectives

Les perspectives de ce travail sont :

- De l'étendre au langage de règles du Web sémantique (SWRL) et de l'utiliser pour aider les utilisateurs à développer une conceptualisation d'ontologie basée sur la réutilisation des systèmes d'information d'entreprise.
- De réaliser une implémentation de notre approche en utilisant d'autres outils de transformation tels que AGG et VIATRA2 etc., dans le but de comparer et améliorer les performances.
- De générer le code OWL2 équivalent dans les différentes syntaxes offertes par OWL2, en révisant les Templates crées par Xpand.
- D'Appliquer d'autres transformations sur les spécifications UML afin de réduire leur complexité et garantir un niveau plus élevé d'abstraction.

Bibliographie

Antoniou, Grigoris, Paul Groth, Frank van Harmelen, et Rinke Rinke Hoekstra. *A Semantic Web Primer.* Cambridge, Massachusetts: The MIT Press, 2012.

Arendt, Thorsten, Enrico Biermann, Stefan Jurack, Christian Krause, et Gabriele Taentzer. «Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations.» *Conference: Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010.* Oslo, Norway, October 2010. 121-135.

Azaiez, Selma. «Approche dirigée par les modèles pour le développement de systèmes multi-agents.» . Thèse de doctorat, Université de Savoie, 2007.

Baader, Franz, Diego Calvanese, Deborah L Mcguinness, Daniele Nardi, et Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, and Applications.* 2nd edition. New York, NY: Cambridge University Press, 2010.

BACHIMONT, Bruno. «Engagement Sémantique et Engagement Ontologique.» Chap. 19 dans *Conception et Réalisation D'ontologies En Ingénerie Des Connaissances*, 305-324. Eyrolles, 2000.

Bachimont, Bruno, Antoine Isaac, et Raphaël Troncy. «Semantic Commitment for Designing Ontologies: A Proposal.» *Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web, 13th International Conference, EKAW 2002, Proceedings.* Siguenza, Spain, 2002. 114-121.

Baget , Jean-François, Étienne Canaud, Jérôme Euze, et Mohand Saïd Hacid . *Les langages du web sémantique.* INRIA, 2003.

Barbier, Franck. *UML 2 et MDE Ingénierie des modèles avec études de cas.* Dunod, 07/11/2005.

BechhoferIan , Sean, Ian Horrocks, Goble Carole , et Robert Stevens. «OilEd: A Reason-able Ontology Editor for the Semantic Web.» *KI 2001: Advances in Artificial Intelligence.* Springer, 2001. 396-408.

Belghiat, Aissam, et Mustapha Bourahla. «UML Class Diagrams to OWL Ontologies: A Graph Transformation based Approach.» *International Journal of Computer Applications* 3, n° 41 (March 2012): 41-46.

Berners-Lee, Tim, James Hendler , et Ora Lassila. «The semantic web.» *Scientific american*, 2001.

Bézivin, Jean, et Xavier Blanc. *MDA : Vers un important changement de paradigme en génie.* Juillet 2002.

Bézivin, Jean, Grégoire Dupé, Frédéric Jouault, et Gilles Pitette. «First experiments with the ATL model transformation language: Transforming XSLT into XQuery.» *OOPSLA 2003 Workshop.* January 2003.

Bézivin, Jean, Sébastien Gérard , Pierre-Alain Muller , et Laurent Rioux. «MDA components: Challenges and Opportunities.» *Proceedings of Metamodelling for MDA (York, England),.* 2003.

Biermann, Enrico, Claudia Ermel, Leen Lambers, Ulrike Golas, Olga Runge, et Gabriele Taentzer. «Introduction to AGG and EMF Tiger by modeling a Conference Scheduling System.» *International Journal on Software Tools for Technology Transfer* (3-4), n° 12 (July 2010): 245-261.

Blanc, Xavier. *Mda en action :Ingénerie logicielle guidée par les modèles.* Etrolles, 2005.

Borst, Willem Nico. «Construction of Engineering Ontologies for Knowledge Sharing and Reuse.» Centre for Telematica and Information Technology, University of Tweenty, Enschede, The Netherlands, 1997.

BOUDIA, M., & BOURAHLA, M. (2022). Formalization of Ontology Conceptualizations using Model Transformation. International Journal of Information System Modeling and Design (IJISMD)"., Volume 13, issue 1.).

Brickley, Dan, et R.V. Guha. *RDF Vocabulary Description Language 1.1: RDF Schema. RDF Schema 1.1.W3C Recommendation 25 February 2014.* 2014. https://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

Bry, François, Tim Furche, Clemens Ley, Benedikt Linse, et Bruno Marnette. «RDFLog: It's like datalog for RDF.» *In WLProceedings ACM Symposium on Principles of Database Systems (PODS).* Paris, France, January 2008. 95 - 106.

Chasseray, Yohann, Anne-Marie Barthe-Delanoe, Stéphane Negny, et Jean-Marc Le Lann. «A generic metamodel for data extraction and generic ontology population.» *Journal of Information Science*, February 2021.

Civili, Cristina, Marco Console, Giuseppe De Giacomo, et Domenico Lembo. «MASTRO STUDIO: Managing ontology-based data access applications.» *International Conference On Very Large Data Bases (VLDB 2013).* 2013. 1314-1317.

Combemale, Benoît. *Metamodeling Approach for Model Simulation and Verification : Application to Process Engineering.* Toulouse: Institut National Polytechnique de Toulouse - INPT, July 2008.

Corcho, Óscar, Mariano Fernández-López, Asunción Gómez-Pérez, et Óscar Vicente. «WebODE: An Integrated Workbench for Ontology Representation, Reasoning, and Exchange.» *13th International Conference on Knowledge Engineering and Knowledge Management EKAW02.* 13 September 2002. 138-153.

Czarnecki, Krzysztof, et Simon Helsen. «Classification of model transformation approaches.» Canada: OPSLA‘03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, University of Waterloo, January 2003.

Davis, R, H Shrobe, et P Szolovits. «What is a Knowledge Representation? .» 1993: 17-33.

Efftinge, Sven, et al.. «Xpand Documentation.» Xpand Documentation, 2004 - 2014.

Ehrig, Marc, et Steffen Staab. «QOM – Quick ontology mapping.» *INFORMATIK 2004.* 2004.

Essaid, Amira, Boutheina Ben Yaghlane, et Arnaud Marti. «Gestion du conflit dans l'appariement des ontologiesExtraction et Gestion des Connaissances.» Communication dans un congrès, 2011, 50-60.

Favre, Jean-Marie. «Foundations of Model (Driven) (Reverse) Engineering : Models -- Episode I: Stories of The Fidus Papyrus and of The Solarus.» January 2004.

Feldman, Howard J, Michel Dumontier, Susan Ling, Norbert Haider, et Christopher W.V. Hogue. «CO: A chemical ontology for identification of functional groups and semantic comparison of small molecules.» August 2005: 4685-4691.

Fernández-López, Mariano, Asuncion Gomez-Perez, et Natalia Juristo. «METHONTOLOGY: from ontological art towards ontological engineering.» *In Proceedings of the AAAI97 Spring Symposium Series on Ontological Engineering.* Stanford , 1997. 33-40.

Gamma, Erich, Richard Helm, et Johnson Ralph. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-, 1995.

GAOUAR, Lamia, **A**pproche MDA pour la construction d'interfaces homme-machine. Thèse de Doctorat, Laboratoire Laboratoire de Télécommunications de Tlemcen (LTT), UNIVERSITE ABOU-BEKR BELKAID – TLEMCEN,Février 2019.

Gómez-Pérez, Asunción, Mariano Fernández-López, et Crocho Oscar . «Ontological Engineering.» Dans *Advanced Information and Knowledge Processing*. Springer-Verlag , 2004.

GÒMEZ-PÉREZ, Asunciòn, Natalia JURISTO , et Juan PAZOS. «Evaluation and assessment of the knowledge sharing technology.» Dans *In Towards*, 289-296. IOS Press, 1995.

Guihal, David. *Modélisation en langage EN LANGAGE VHDL-AMS Des systèmes pluridisciplinaires.* Thèse de Doctorat, Laboratoire d'analyse et des systèmes du CNRS, l'Université Toulouse III, Mai 2007.

Hadzic, Maja, Pornpit Wongthongtham, Tharam Dillon, et Elizabeth Chang. *Ontology-Based Multi-Agent Systems.* Springer, 2009.

Hidaka, Soichiro, Massimo Tisi, Jordi Cabot , et Zhenjiang Hu . «Feature-based classification of bidirectional transformation approaches.» *Software & Systems Modeling*, 2016: 907–928.

Hitzler, Pascal, Markus Krötzsch, Bijan Parsia, Peter F Patel-Schneider, et Sebastian Rudolph. *OWL 2 Web Ontology Language Primer.* September 2012.

Hofmann, Martin, Benjamin Pierce, et Daniel Wagner. «Symmetric lenses.» *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '11.* ACM Press, 1January 2011. 371–384.

Horridge, Matthew, et Peter F Patel-Schneider. *OWL 2 Web Ontology Language Manchester Syntax.* 14 April 2009.

Horridge, Matthew, et Peter F. Patel-Schneider. *OWL 2 Web Ontology Language Manchester Syntax (Second Edition).* 11 December 2012.

Horrocks, Ian, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, et Mike Dean. «SWRL: A Semantic Web Rule Language.» Édité par W3C. 21 May 2004. http://www.w3.org/Submission/SWRL/.

Horváth , Ákos, István Ráth , et Zoltán Ujhelyi. «Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework.» *Software & Systems Modeling* 15, n° 3 (May 2016): 609-629.

Huth , Michael, et Mark Ryan. *ogic in computer science : modelling and reasoning about systems.* 2004.

Inaba, Akiko, Thepchai Supnithi, Mitsuru Ikeda, Riichiro Mizoguchi, et Jun'ichi Toyoda. «An Overview of \Learning Goal Ontology.» *In ECAI2000 Workshop on Analysis and Modelling of Collaborative Learning Interactions.* 2000. 23-30.

Jouault , Frédéric, et Massimo Tisi. «Towards Incremental Execution of ATL Transformations.» Édité par Laurence TRATT , & Martin GOGOLLA. *ICMT 2010: Theory and Practice of Model Transformations.* Berlin Heidelberg: Springer, 2010. 123-137.

Jouault , Frédéric, et Ivan Kurtev . «Transforming Models with ATL.» *MODELS 2005: Satellite Events at the MoDELS 2005 Conference.* s.d. 128–138.

Kadima, Hubert. *MDA - Conception orientée objet guidée par les modèles.* Dunod, 2005.

Kazakov, Yevgeny, et Pavel Klinov. «Goal-Directed Tracing of Inferences in EL Ontologies.» *The Semantic Web – ISWC 2014.* Springer International Publishing, 2014. 196-211.

Kchaou, Mariem, Wiem Khlif, Faiez Gargouri, et Mariem Mahfoudh. «Transformation of BPMN Model into an OWL2 Ontology.» *Conference: 16th International Conference on Evaluation of Novel Approaches to Software Engineering ENASE.* January 2021. pp. 380-388.

Kifer, Michael, Georg Lausen, et James Wu. «Logical foundations of object-oriented and frame-based languages.» *Journal of the ACM* 42, n° 4 (July 1995): 741-843.

Klein, Stefan, et Claudia Loebbecke. «The Transformation of Pricing Models on the Web: Examples from the Airline Industry.» *13th International Bled Electronic Commerce Conference,.* 2000. 19-21.

Kleppe , Anneke, Jos Warme , et Wim Bast. *MDA Explained: The Model-Driven Architecture: Practice and Promise. Addison Wesley (2003).* 2003.

KLEPPE , Anneke, Jos WARMER , et Wim BAST. *MDA Explained. The Model Driven Architecture Practice and Promise.* Addison-Wesley, 2003.

Kuhn, Adrian, et Toon Verwaest. «FAME, A Polyglot Library for Metamodeling at Runtime.» *In Third InternationalWorkshop onModels@run.time.* 2008.

Kusel, Angelika, et al.. «A survey on incremental model transformation approaches.» *Vortrag: Models and Evolution Workshop (MODELS 2013) , Proceedings of the Models and Evolution Workshop @ (MODELS 13), (2013),*. Miami, 2013. 1-10.

Lara , Juan de, et Esther Guerra. «Formal support for model driven development with graph transformation techniques.» Édité par UAM. Departamento de Ingeniería Informática. 157 (2005): 30-39.

Lauder, Marius, Anthony Anjorin, Gergely Varró, et Andy Schürr. «Bidirectional Model Transformation with Precedence Triple Graph Grammars.» *ECMFA 2012: Modelling Foundations and Applications pp 287-302.* July 2012. pp 287-302.

Le Calvar, Theo. «Exploration d'ensembles de modèles.» Université d'Angers, 2019.

Leblebici , Erhan, Anthony Anjorin, Schürr Andy, Stephan Hildebrandt, Jan Rieke, et Joel Greenyer. «A Comparison of Incremental Triple Graph Grammar Tools.» *Electronic Communications of the EASST* 67 (2014).

LEHIRECHE, Nesrine. *INTEGRATION DES SYSTEMES D'INFORMATION DIRIGÉE PAR LES LINKED DATA.* thèse de doctorat , Université de Sidi Bel Abbès, 2021.

Levendovszky, Tihamér, et Hassan Charaf. «Applying Metamodels in Software Model Transformation Methods.» PhD thesis, Budapest, 2005.

Lindberg , D A, B L Humphreys, et A T McCray. «The Unified Medical Language System.» *Methods Inf Med*, 1993: 281--291.

Lohmann, Carsten, Joel Greenyer, Juanjuan Jiang, et Tarja Systä. «Applying Triple Graph Grammars For Pattern-Based Workflow Model Transformations.» *Journal of Object Technology*, October 2007: 253-273.

Maedche, Alexander, et Valentin Zacharias. «Clustering Ontology-Based Metadata in the Semantic Web.» *European Conference on Principles of Data Mining and Knowledge Discovery PKDD 2002: Principles of Data Mining and Knowledge Discovery* . 2002. 348-360.

Marin, Draltan. «RDF Formalization.» Technical report, Universidad de Chile, 2004.

Martínez Pérez, Salvador, Massimo Tisi, et Rémi Douence. «Reactive Model Transformation with ATL.» *Science of Computer Programming* 136, n° 4 (Mars 2017): 1-16.

Mcguinness, Deborah L, Richard Fikes, James Rice, et Steve Wilder. «An Environment for Merging and Testing Large Ontologies.» Édité par F. Giunchiglia and B. Selman A. G. Cohn. *Principles of Knowledge Representation and Reasoning. Proceedings of the Seventh International Conference (KR2000.* San Francisco, CA, Morgan Kaufmann Publishers, 2000.

Michard, Alain. *XML : Langage et applications.* Eyrolles, 1999.

Muller, Pierre-Alain, Franck Fleurey, et Jean-Marc Jézéquel. «Weaving Executability into Object-Oriented Meta-languages.» *MODELS 2005: Model Driven Engineering Languages and Systems* . 2005. 264–278.

Muñoz, Sergio, Jorge Pérez, et Claudio Gutierrez. «Minimal Deductive Systems for RDF.» *ESWC '07: Proceedings of the 4th European conference on The Semantic Web: Research and Applications.* June 2007. 53–67.

Napoli, Amedeo. «Une introduction aux logiques de descriptions.» Rapport dee recherche, INRIA Lorraine, LORIA - Laboratoire Lorrain de Recherche en Informatique et ses Applications, 1997.

Noy, Natalya Fridman, Ray W Fergerson, et Musen Mark A. «The Knowledge Model of Protégé-2000: Combining Interoperability and Flexibility.» *Proceedings of the 12th International Conference on Knowledge Engineering and Knowledge Management: Methods, Models, and Tools (EKAW 2000).* Juan-les-Pins,France, July 2002. 17-32.

OWL Working Group . «OWL 2 Web Ontology Language : Document Overview.» Technical report, 2009.

PIM, Borst, Akkermans HANS , et JAN Top . «Engineering ontologies.» *International Journal of Human-Computer Studies* 46, n° 2–3 (February 1997): 365-406.

Psyché, Valéry, Olavo Mendes, et Jacqueline Bourdeau. «Apport de l'ingénierie ontologique aux environnements de formation à distance.» *Sciences et Technologies de l Information et de la Communication pour l Éducation et la Formation*, January 2004.

Seidwitz, Ed. «What Models Mean IEEE Software (September 2003).» October 2003: 26 - 32.

Smith, Michael K., Chris Welty, et Deborah L. McGuinness. *Le langage d'ontologie Web OWL.* 10 février 2004.

Sowa, John F. «Semantic Networks.» *Encyclopedia of Artificial Intelligence.* 1987.

SOWA, John. *Knowledge Representation : Logical, Philosophical, and Computational Foundations.* Vol. vii. Brooks/Cole, August 2000.

Steinberg, Dave, Stephen A Brodsky, Ed Merks, et Frank Budinsky. *Eclipse Modeling Framework.* Pearson Education, 2008.

Stevens , Perdita. «A Landscape of Bidirectional Model Transformations.» *International Summer School on Generative and Transformational Techniques in Software Engineering.GTTSE 2007: Generative and Transformational Techniques in Software Engineering II.* 2008. 408–424.

Studer, Rudi, V. Richard Benjamins, et Dieter Fensel. «Knowledge engineering: principles and methods.» *Data & Knowledge Engineering* 25, n° (1-2) (March 1998): 161-197.

Sure, York, Michael Erdmann, Jürgen Angele, Steffen Staab, Rudi Studer, et Dirk Wenke. «OntoEdit: Collaborative Ontology Development for the Semantic Web.» *First International Semantic Web Conference (ISWC'02).* Sardinia, Italy: Springer Verlag , 2002. 221-235.

Taentzer, Gabriele. «AGG: A tool environment for algebraic graph transfor-mation. In Applications of Graph Transformations with Industrial Relevance.» Springer, 2000.: pages 481–488.

Taentzer, Gabriele, et al.. «Model transformation by graph transformation: A comparative study.» January 2013.

Uschold, Michael, et Michael Grüninger. «Ontologies: Principles, methods and applications.» *The Knowledge Engineering Review* 11, n° 02 (January 1996).

Varro , Daniel, et Andras Balogh. «The model transformation language of the VIATRA2 framework.» *Science of Computer Programming.Special Issue on Model Transformation*, 2007: 214-234.

Xavier, Blanc, et Isabelle Mounier. *Uml2 pour les développeurs Cours avec exercices corrigés.* Paris: Eyrolles, 28 September,2006.

Yohan , Chabot. «Langage d'ontologies Web OWL 2. Vue d'ensemble (Deuxième édition).» 2014.

AIT YAKOUB , Zina . «Logiques de Description & Analyse de Concepts Formels pour des.» thèse de Doctorat, UNIVERSITÉ MOULOUD MAMMERI DE TIZI OUZOU, 2018.

AUDIBERT, Laurent. «UML 2 - de l'apprentissage à la pratique.» institut universitaire de technologie de villetaneuse, département informatique,, 12 janvier 2009 .

Baader, Franz, Sebastian Brandt, et Carste Lutz. «Pushing the EL Envelope.» *IJCAI.* 2005. 364–369.

Baader, Franz, Diego Calvanese, Deborah Mcguinness, Daniele Nardi, et Peter F Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, and Applications.* Édité par 2nd Edition Cambridge University Press. January 2007.

Beaudoux, Olivier , Arnaud Blouin, Olivier Barais , et Jean-Marc Jézéquel. «Active Operations on Collections.» *MODELS 2010: Model Driven Engineering Languages and Systems.* Berlin Heidelberg: Springer , 2010. 91–105.

Bendiaf,Messaoud. «SPÉCIFICATION ET VÉRIFICATION DES SYSTÈMESEMBARQUÉS TEMPS RÉEL EN UTILISANT LA LOGIQUE DE RÉÉCRITURE.» Thèse de Doctorat , UNIVERSITE MOHAMED KHIDER BISKRA, 2017.

Bendiaf Messaoud , Bourahla Mustapha, Boudia Malika, Seidali Rehab. «A Model Transformation Approach for Specifying Real-Time Systems and Its Verification Using RT-Maude.» International Journal of Information Technology and Web Engineering (IJITWE) 12(4).2017. pp20-41.

Bernaschina, Carlo . «ALMOsT.js: An Agile Model to Model and Model to Text Transformation Framework.» *CWE 2017: Web Engineering.* 2017. pp 79-97.

Berners Lee, Tim, James Hendler, et Olli Lassila. «The Semantic Web".» *in Scientific American*, May 2001: 35-43.

Beugnard, Antoine , Fabien Dagnat, Sylvain Guerin, et Christophe Guychard. «Des situations de modélisation pour décrire un processus de modélisation.» April 2015.

Béziven , jean, et Olivier Gerbé. *Towards a Precise Definition of the OMG/MDA Framework, presented at ASE'01, Automated Software Engineering.* San Diego, USA, 2001.

Bézivin , Jean , et Xavier Blanc. «Promesses et interrogations de l'approche MDA.» *Développeur Référence.* 2002.

Bézivin, Jean. «On the Unification Power of Models.» *Software and Systems Modeling* 4 (May 2005): 171-188.

Bézivin, Jean. «Sur les principes de base de l'ingénierie des modèles.» *L'OBJET: Software, Databases, Networks* 10(4) (Novembre 2004): 145-157.

Bézivin, Jean, et Olivier Gerbé. «Towards a precise definition of the OMG/MDA framework.» *Automated Software Engineering, 2001. (ASE 2001).* 2001.

Bock, Conrad , et al.. *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition).* 11 December 2012.

Bohannon, Aaron , J. Nathan Foster, Benjamin C. Pierce , Alexandre Pilkiewicz , et al.an Schmitt. «Boomerang.» *Proceedings of the 35th annual ACM SIGPLANSIGACT symposium on Principles of programming languages - POPL '08.* ACM Press, 2008.

Boronat, Artur. «Expressive and Efficient Model Transformation with an Internal DSL of Xtend.» *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems MODELS '18.* ACM Press, 2018.

Brisson, Laurent. *Mesures d'intérêt subjectif et représentation des.* Rapport technique, Nice (France): Laboratoire I3S, Université Sophia Antipolis,, 2004, 21.

Brockmans, Saartje , et al.. «A Model Driven Approach for Building OWL DL and OWL Full Ontologies.» *ISWC 2006: The Semantic Web.* 2006. pp 187-200.

Budinsky, Frank , Dave Steinberg, Marcelo Paternostro, et Ed Merks. *Eclipse Modeling Framework.* The Eclipse series. Pearson Education, 2003.

Casellas, Núria . *Methodologies, Tools and Languages for Ontology Design.* Vol. 3, chez *the Law, Governance and Technology Series book series*. June 2011.

Chebotko, Artem, , Shiyong Lu, M Jamil Hasan, et Farshad Fotouhi . «Semantics Preserving SPARQL-to-SQL Query Translation.» Technical Report TR-DB-052006-CLJF, May 2006. Revised November 2006, 2006.

Chein, Michel , et Marie-Laure Mugnier. *Graph-based Knowledge Representation, Computational Foundations of Conceptual Graphs, Advanced Information and Knowledge Processing,* . Springer, 2009.

Combemale, benoît. *Approche de méta-modélisation pour la simulation et la vérification de modèle.* Thèse soutenu à l'ENSEEIHT, juillet 2008.

Corcho, Oscar, Mariano Fernández-López, et Asunción Gómez-Pérez. «Methodologies, tools and languages for building ontologies. Where is their meeting point?» *Data & Knowledge Engineering* 46, n° 1 (July 2003): 41-64.

Csertan, György, Gábor Huszerl, István Majzik, Zsigmond Pap, András Patricza, et Dániel Varro. «VIATRA - visual automated transformations for formal verification and validation of UML models.» *Proceedings 17th IEEE International Conference on Automated Software Engineering,*. 2002.

Czarnecki, K, et S Helsen. «feature –based survey of model transformation approaches.» *IBM SYSTEMS JOURNAL,* 2006.

De Lara , Juan , et Hans Vangheluwe. «AToM3: A Tool for Multi-formalism and Meta-modelling.» Édité par Springer. *Fundamental Approaches to Software Engineering.* 2002. pp 174-188.

Diaw, Samba, Redouane Lbath, et Bernard Coulette. «Etat de l'art sur le développement logiciel basé sur les transformations de modèles.» *Computer Science* Vol.29 n°4-5 (2010): pp.505-536.

Dimitrov, M. «Introduction to Semantic Technologies , O ntologies an the S emantic Web.» 3rd GATE Trainaing Course, 2010.

DJAKHDJAKHA, Lynda . «Un formalisme pour l'alignement des ontologies multipoints de vue basé sur une extension de la logique de description.» Thèse de Doctorat , Université Constantine 2, 2014.

DURIF, THOMAS . «Environnement informé sémantiquement enrichi pour la simulation multi-agents Application a la simulation en environnement virtuel 3D.» thèse de doctorat, Université de Bourgogne, 2014.

ELLI, Abdelmoutia T. «Raisonnement sur les ontologies légères.» thèse de doctorat, Université Mohamed Khider de BISKRA, 2018.

Erik Erikson , Hans, Magnus Penker, David Fado, et Lyons Brian. *UML 2 ToolKit.* Wiley publishing, 2004.

FAVRE , Jean-Marie, Jacky ESTUBLIER , et Mireille BLAY-FORNARINO. *L'ingénierie dirigée par les modèles : au-delà du mda.* Hermes Science publications – Lavoisier , 2006.

Flater, David . *Sumo2loom documentation.* 2003.

Forgy, Charles L. «Rete: A fast algorithm for the many pattern/many object pattern match problem.» Édité par Elsevier. *Readings in Artificial Intelligence and Databases*, 1989: 547-559.

Franz, Baader, Diego Calvanese, Deborah L Mcguinness, Daniele Nardi, et Peter F. Patel-Schneider. *The Description Logic Handbook : Theory, Implementation and Applications.* 2 nd edition. New York,NY, USA: Cambridge University Press, 2010.

FÜRST, Frédéric . «L'ingénierie ontologique.» Rapport technique, Institut de recherche en Informatique , 2002, 15,30.

Ghamarian, Amir Hossein , Maarten de Mol, Arend Rensink, Eduardo Zambon , et Maria Zimakova . «Modelling and analysis using GROOVE.» *International Journal on Software Tools for Technology Transfer*, 2012: 15-40.

Giese , Holger , et Robert Wagner . «Incremental Model Synchronization with Triple Graph Grammars.» *MODELS 2006: Model Driven Engineering Languages and Systems.* Berlin Heidelberg: Springer, 2006. 543–557.

Gomez-Perez, A, J Angele, M Fernandez-Lopez, V Christophides, A Stutt, et Y Sure. «A survey on ontology tools. OntoWeb -Ontology-based information exchange for knowledge management and electronic commerce.» *IST-2000-29243.* Universidad Politecnica de Madrid, 2002.

Gómez-Pérez, A., 1999. . Expert Update, 2(3), pp. «Ontological Engineering: A state of the art.» 2, n° 3 (1999): 33-43.

Grau, Bernardo Cuenca , Peter F Patel-Schneider, et Boris Motik. *OWL 2 web ontology language direct semantics (second edition). W3C recommendation.* 2012.

Greenyer Joel. A study of model transformation technologies: Reconciling TGGs with

QVT. Master's thesis, University of Paderborn, 2006.

Greenyer Joel and Rieke Jan. Applying advanced tgg concepts for a complex transformation of sequence diagram specifications to timed game automata. In Applications of Graph Transformations with Industrial Relevance, pages 222–237. Springer, 2012

GROOVE. *GROOVE; HomePage.* s.d.

Gruber, Thomas R. «A translation approach to portable ontology specifications.» *Knowledge Acquisition* 5, n° 2 (June 1993): 199-220.

Gruber, Thomas R, et Gregory R Olsen. «An Ontology for Engineering Mathematics.» *Principles of Knowledge Representation and Reasoning Proceedings of the Fourth International Conference (KR '94).* 1994. 258-269.

Gruber, Thomas R. «A translation approach to portable ontology specifications.» *Knowledge Acquisition, An International Journal of Knowledge Aquisition for Knowledge-Based Systems* 5, n° 2 (1993): 199-220.

GUARINO , Nicola , et Pierdaniele GIARETTA. *Ontologies and knowledge bases: towards a terminological clarification.* Vol. II, chez *Towards Very Large Knowledge Bases : Knowledge Building and Knowledge Sharing*, édité par IOS PresS, 14, 28, 29. January 1995.

Guarino, N. «Some organizing principles for a unified top-level ontology.» *AAAI Spring Symposium on Ontological Engineering.* 1997. 57-63.

Guber, T.R. «Toward Principles for the Design of Ontologies Used for Knowledge Sharing.» Padova: presented at International Workshop on Formal Ontology, 1993.

Gueffaz, Mahdi . *ScaleSem :modelchecking et websémantique.* Thèse de Doctorat , UniversitédeBour-, 2012.

Guo, Yuanbo, Zhengxiang Pan, et Jeff Heflin. «LUBM: A benchmark for OWL knowledge base system.» *Journal of Web Semantics* Volume 3, Issues 2–3 (October 2005): 158-182.

Gutierrez, Claudio, Carlos A Hurtado, et al.berto O Mendelzon. «Foundations of Semantic Web Databases.» *Proceedings ACM Symposium on Principles of Database Systems (PODS).* Paris, France, 2004. 95 - 106 .

Haarslev, Volker , Kay Hidde, Ralf Möller, et Michael Wessel. «The RacerPro Knowledge Representation and.» *Semantic Web* 3, n° 3 (2012).

Hardebolle, Cécile. «Composition de modèles pour la modélisation multi-paradigme du comportement des systèmes.» Thèse de Doctorat , Université Paris-Sud XI UFR Scientifique d'Orsay, 2009.

Hildebrandt, Stephan , et al.. «A Survey of Triple Graph Grammar Tools.» *Electronic Communications of theEASST 57 : Bidirectional Transformations 2013 (2013),*, 2013.

Hildebrandt, Stephan , Leen Lambers, Holger Giese, Dominic Petrick, et Ingo Richter. «Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations.» *Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations. AGTIVE: Applications of Graph Transformations with Industrial Relevance,*. October 2011. pp 238-253 .

Jouault, Frédéric . *Contribution à l'étude des langages de transformation de modèles.* Université Nantes, 2006.

Jouault, Frederic , Freddy Allilaire, Jean Bezivin , et Ivan Kurtev. «ATL : A model transformation tool.» *Science of Computer Programming* 72, n° 1–2 (June 2008): 31-39.

Kent , Stuart . «Model driven engineering. Integrated Formal Methods (IFM).» Édité par Springer. *International conference on integrated formal methods.* Turku, 2002. 286-298.

Kerkouche , El-hillali . *Modélisation Multi-Paradigme : une approche basée sur la transformtion de graphes.* constantine, Université de Mentouri, 2011.

Kiczales, Gregor , et al.. «Aspect-Oriented Programming.» Édité par Springer. *European Conference on Object-Oriented Programming.* Finland, 1997. 220–242.

Kindler Ekkart and Wagne Robert. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, Technical Report, University of Paderborn, D-33098 Paderborn, Germany, 2007.

Kleppe, Anneke, Jos Warmer, et Wim Bast :. *MDA Explained. The Model Driven Architecture Practice and Promise.* Addison-Wesley, 2003.

Königs, Alexander , et Andy Schürr. «Tool Integration with Triple Graph Grammars - A Survey.» *Electronic Notes in Theoretical Computer Science*, February 2006: 113-150.

KUHN , Adrian , et Toon VERWAEST . «Fame – a polyglot library for metamodeling at runtime.» *In Third InternationalWorkshop onModels@run.time.* 2008.

Kûhne, Tomas. «Matters of (Meta-) Modeling Thomas kuh ne Darmstadt University of Technology, Darmstadt, Germany.» *Software and Systems Modeling* (Darmstadt University of Technology) Volume 5, n° Number 4 (December 2006): 369-385.

KuhÜne, Thomas. «Matters of (Meta-) Modeling.» *Software and Systems Modeling* Volume 5, n° Number 4 (, December 2006): 369-385.

Le Calvar, Théo , Frédéric Jouault, Fabien Chhel, et Mickael Clavreul. «Efficient ATL Incremental Transformations.» *Journal of Object Technology* 18, n° 3 (July 2019).

LEFRANCOIS, Maxime. «Logiques de description, et ontologies en logiques de description.» Mines Saint-Etienne, 2016.

Luke, Sean , Lee Spector, David Rager, et James Hendler. «Ontology-based Web agents.» *AGENTS '97: Proceedings of the first international conference on Autonomous agents.* Marina delRey, CA, USA, : ACM Press, 1997. 59–66.

Mallouli, Sana Damak. Méta-modélisation du Comportement d'un modéle de processus :Une Démarche de Construction d'un moteur d'exécution.Thèse de Doctorat. L'UNIVERSITE PARIS 1 PANTHEON – SORBONNE,2014

Marvie, Raphael . *Séparation des préoccupations et méta-modélisation pour environnements de manipulation d'architectures logicielles à base de composants.* Thèse de Doctorat, LIFL (laboratoire d'Informatique Fondamentale de Lille), Université des sciences et technologies de Lille, décembre 2002.

Mens, Tom, et Pieter Van Gorp. «A Taxonomy of Model Transformation.» *Electronic Notes in Theoretical Computer Science* 152, 27 (March 2006): 125-142.

Mens, Tom, et PieterVan Gorp. «A Taxonomy of Model Transformations.» *Electronic Notes in Theoretical Computer Science*, March 2006.

Minh , Hoang Lien Vo, et Hoang Quang . «Transformation of UML class diagram into OWL Ontology.» *Journal of Information and Telecommunication* 4, n° 1 (2019): 1-16.

Mizoguchi, R. «A Step Towards Ontological Engineering. Paper presented at .» *the 12th National Conference on AI of JSAI.* 1998.

Mizoguchi, Riichiro , et Jacqueline Bourdeau. «Using Ontological Engineering to Overcome Common AI-ED Problems.» *International Journal of Artificial Intelligence in Education IJAIED*, January 2001.

Motta, E, S. B. Shum, et J. Domingue. «Ontology-driven document enrichment: principles, tools and applications.» 52, n° 6 (June 2000): 1071-1109.

Muller, Pierre-Alain, et Natalie Gaertner. *Modélisation objet avec UML, 2000, Deuxième tirage 2001*. 2e édition. Eyrolle, 2000.

N. NOY et D. L. MCGUINNESS, Natalya F Noy , et Deborah L McGuinness. «« Ontology Development 101: A Guide to Creating Your First Ontology.» Technical Report , Stanford Knowledge Systems Laboratory., Mars 2001.

Neches, Robert , et al.. «Enabling Technology for Knowledge Sharing.» *AI Magazine* 12, n° (3) (1991): 36–56.

Nguyen, Thi Thanh Tam. « Codèle :Une Approche de Composition de Modèles pour la Construction de Systèmes à Grande Échelle.» Thèse de Doctorat , Université Joseph Fourier de Grenoble, niversité Joseph-Fourier - Grenoble I, 2008.

Nigel, Shadbolt, T Berners-Lee, et W Hall. «The Semantic Web Revisited.» Édité par IEEE. *IEEE Intelligent Systems* 21, n° 3 (May 2006): 96-101.

OMG. « Mda guide version 1.0.1.(copyright 2003).» 2003.

OMG. *Model Driven Architecture (MDA).* 2014.

OMG. *Common Warehouse Metamodel.* Vol. 1. mars 2003.

OMG. *CORBA Component Model Specification.* 2006.

OMG. «OMG : Meta object facility (MOF) core specification.» Rapport technique, 2006.

OMG. *OMG Meta Object Facility (MOF) Core Specification.* 01 Octobre 2019.

OMG. *Unified Modeling Language (UML) version 2.5.1.* December 2017.

OMG. «XML Metadata Interchange.» *MOF 2.0/XMI Mapping, Version 2.1.1.* décembre 2007.

Piel, Eric. «Ordonnancement de systèmes parallèles temps-réel De la modélisation à la mise en œuvre par l'ingénierie dirigée par les modèles.» Thèse de doctorat, Université des Sciences et Technologies de Lille., 2007.

Recommendation, W3C. *OWL 2 Web Ontology Language XML Serialization (Second Edition).* 11 December 2012.

Recommendation, W3C. *RDF/XML Syntax Specification (Revised).* 10 February 2004.

ROCHE, Christophe . «Terminologie et ontologie.» *Revue Langages* numéro 157 (Mars 2005): 12, 14, 37.

Schreiber, ATh, BJ Wielinga, et W Jansweijer. «The KACTUS View on the 'O' word.» *In Skuce D (ed) IJCAI95 Workshop on Basic Ontological Issues in Knowledge Sharing.* 1995.

Schreiber, G, et M Dean. *OWL web ontology language reference. W3C recommendation.* 2004.

Schürr , Andy . «Specification of graph translators with triple graph grammars.» *WG 1994: Graph-Theoretic Concepts in Computer ScienceInternational Workshop on Graph-Theoretic Concepts in Computer Science.* Berlin Heidelberg: Springer, 1995. 151–163.

Schürr, Andy . «Specification of Graph Translators with Triple Graph Grammars.» *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94.* Herrsching, Germany, January 1994.

Sendall, Shane , et Wojtek Kozaczynski. «Model transformation: the heart and soul of model-driven software development.» *IEEE Software*, 2003: 42 - 45.

Sirin, Evren, Bijan Parsia, Bernardo Cuenca, Aditya Kalyanpur, et Yarden Katz. «Pellet: A practical OWL-DL reasoner.» *Journal of Web Semantics* 5, n° 2 (june 2007): 51-53.

Smith, Barry. «The Blackwell Guide to the Philosophy of Computing and Information.» Dans *The Blackwell Guide to the Philosophy of Computing and Information*, de Luciano Floridi, 155-166. oxford: Blackwell Publishing, 2004.

Smith, Michael K , Chris Welty, et Deborah L Mcguinness. 2004.

Swartout, Bill, Ramesh Patil, Kevin Knight, et Tom Russ. «Toward Distributed Use of Large-Scale Ontologies.» Édité par A Farquhar, M Gruninger, A Gómez-Pérez, M Uschold, & V van der. *AAAI'97 Spring Symposium on Ontological Engineering.* California: Stanford University, 1997. 138--148.

Tiexin , Wang, Cao Jingwen , Tao Chuanqi , Yang Zhibin , Wu Yi , et Li Bohan. «A Configurable Semantic-Based Transformation Method towards Conceptual Models.» *Discrete Dynamics in Nature and Society* 5, n° 6 (September 2020): 1-14.

Uschold, M, et R Jasper. «A Framework for Understanding and Classifying Ontology Applications.» *Benjamins VR (ed) IJCAI'99 Workshop on Ontology and Problem Solving Methods: Lessons Learned and Future Trends. CEUR Workshop Proceedings 18:. Amsterdam. The Netherlands.* Stockholm, Sweden, 1999. 11.1–11.12.

Uschold, Michael , et Michael Grüninger. «Ontologies: Principles, methods and applications.» *The Knowledge Engineering Review* 11, n° 2 (January 1996).

Vega, G. «Développement d'Applications à Grande Echelle par Composition de Méta-modéles,Ph.D. thesis.» Vers. tel-00011325, version 1. Université Joseph Fourier. Décembre 2005. https://tel.archives-ouvertes.fr/tel-00011325 (accès le juin 20, 2021).

W3C OWL Working Group. «OWL 2 Web Ontology Language Document Overview (Second Edition).» 11 December 2012.

W3C OWL Working Group. *OWL 2 Web Ontology Language Document Overview (Second Edition).* 11 December 2012.

W3C, Workin Group. *W3c semantic web faq.* October 2009.

Wardhana, Helna , Ahmad AshariAhmad , et AshariAnny Kartika. «Transformation of SysML Requirement Diagram into OWL Ontologies.» *International Journal of Advanced Computer Science and Applications* 11, n° 4 (January 2020).

Yiling, Lu. «Roadmap for tool support for collaborative ontology engineering.» University of Victoria, 2003.

Zghal, Sami. *Contributions à l'alignement d'ontologies OWL par agrégation de similarité. Thèse de doctorat en informatique.* Thèse de doctorat en informatique, Université de Tunis, El manar, 2010.

Zweigenbaum, P. «Linguistic and medical knowledge bases: An access system for medical records using natural language.» Technical report, , MENELAS, 1993.