Transition de modèles de connaissances - Un système de connaissance fondé sur OWL, Graphes conceptuels et UML

Thomas Raimbault

UNIVERSITÉ
DE NANTES

Version du 30 septembre 2008
Année : 2008
N° d'enregistrement :

TRANSITION DE MODÈLES DE CONNAISSANCES

UN SYSTÈME DE CONNAISSANCE FONDÉ SUR OWL, GRAPHES CONCEPTUELS ET UML

THÈSE DE DOCTORAT

Spécialité : Informatique

ÉCOLE DOCTORALE STIM

« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATÉRIAUX »

Présentée et soutenue publiquement

Le 21 novembre 2008

À Nantes

Par **Thomas RAIMBAULT**

Devant le jury ci-dessous :

Rapporteurs :	Mohand-Saïd HACID,	Professeur à l'Université de Lyon 1
	Chantal REYNAUD,	Professeur à l'Université de Paris XI - Orsay
Examinateurs :	Gilles KASSEL,	Professeur à l'Université de Picardie
	Mourad Chabane OUSSALAH,	Professeur à l'Université de Nantes
Co-directeurs de thèse :	Henri BRIAND,	Professeur à l'Université de Nantes
	Stéphane LOISEAU,	Professeur à l'Université de d'Angers
Co-encadrants de thèse :	David GENEST,	Maître de conférences à l'Université d'Angers
	Rémi LEHN,	Maître de conférences à l'Université de Nantes

Laboratoire d'Informatique de Nantes Atlantique (LINA)
Laboratoire d'Étude et de Recherche en Informatique d'Angers (LERIA)

Remerciements

Sommaire

Introduction générale

Contexte de travail

L'informatique désigne la science et les techniques en rapport à l'automatisation du traitement de l'information effectué par un système artificiel ou abstrait. Née dans la première moitié du XX^{e} siècle, l'intelligence artificielle (IA) a pour objectif d'obtenir de la machine un comportement jugé « intelligent » par l'être humain [Charniak et McDermott, 1985; Moor, 2003].

La *représentation des connaissances* est une discipline de l'IA qui consiste à exprimer une modélisation des connaissances d'une partie du monde réel, appelée *domaine*, sous une forme adaptée pour qu'un opérateur - humain ou machine - puisse les interpréter et/ou les manipuler. Une modélisation de connaissances est définie selon un *langage de représentation*, qui est caractérisé par un ensemble de types de connaissances, appelés *notions*. Une modélisation est donc une *instanciation* de notions. Par exemple la notion de classe est instanciée par Humain pour représenter l'ensemble des humains.

Une modélisation est liée à un ensemble de types de raisonnements applicables sur les connaissances modélisées. Un *raisonnement* désigne de façon générale un processus d'élaboration d'inférences. L'inférence consiste à combiner un ensemble d'informations préalablement connues afin d'obtenir une information qui en découle. On appelle *modèle de connaissance*, ou simplement *modèle*, un couple formé d'un langage de représentation et d'un ensemble de types de raisonnements applicables sur les connaissances représentées par ce langage.

Dans cette thèse, nous nous intéressons principalement à la modélisation des connaissances d'un domaine sur deux niveaux conceptuels : l'un structurel et l'autre factuel (Figure 1). Le niveau structurel confère à la modélisation une vision générique des connaissances d'un domaine ; il constitue les *connaissances structurelles* du domaine. Le niveau factuel - ou assertionnel - fait référence à la spécificité des faits. Il est composé d'*instances* - on dit aussi d'*assertions* - d'éléments structurels, et constitue les *connaissances factuelles* du domaine. Ces deux niveaux offrent la possibilité de raisonner de façon générique et/ou de façon spécifique sur les connaissances modélisées. Par exemple, il peut être souhaitable de tenir compte du concept Humain en général, ou plus spécifiquement de ne s'intéresser qu'à l'individu Pierre.

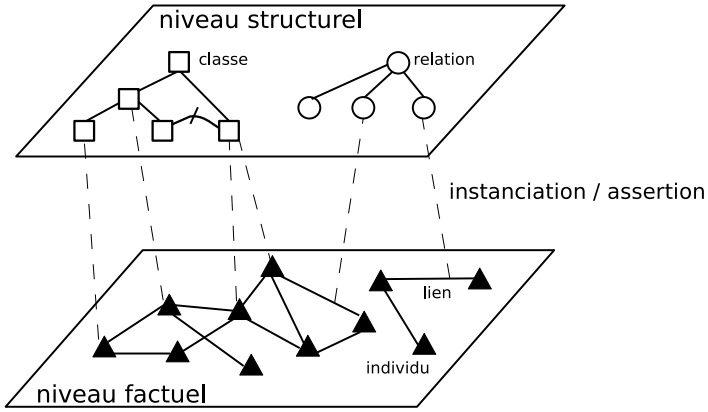


Figure 1 – Modélisation de connaissances sur deux niveaux conceptuels : *connaissances structurelles* (classes, relations, *etc.*) et *connaissances factuelles* (individus, liens, *etc.*).

Problématique

Un problème récurrent en représentation et raisonnement est le suivant. Que faire avec un modèle de connaissance donné face à de nouveaux besoins d'expressivité ou de capacité de raisonnement nécessités par la modélisation en cours ? Plusieurs approches sont envisagées.

Une première approche consiste à utiliser, par *transformations de modèles*, un autre modèle de connaissance autorisant la modélisation du type de connaissance ou plus généralement du type de raisonnement souhaité. Une transformation de modèles consiste à indiquer comment une modélisation dans un modèle source peut être représentée de manière la plus équivalente possible dans un modèle destination. Dans ce cas, le problème n'est que déplacé, car certaines modélisations ou certains types de raisonnements possibles dans le premier modèle peuvent ne plus être possibles dans le nouveau. À cela s'ajoute le fait qu'une transformation de modèle génère nécessairement (si les deux modèles ne sont pas équivalents) une perte d'information d'un modèle à un autre.

Une deuxième approche consiste à *étendre* le modèle de connaissance pour obtenir à la fois l'expressivité et les types de raisonnements voulus. Cependant, le modèle avant extension a généralement été conçu comme un des meilleurs compromis entre expressivité et efficacité. Plus le modèle est expressif, plus la complexité des raisonnements applicables dans le modèle est élevée et donc son efficacité diminuée. Ainsi, l'extension du modèle s'avèrera délicate et fastidieuse à mettre en place avant l'obtention d'un nouveau modèle abouti.

Une troisième approche consiste à utiliser conjointement plusieurs modèles de connaissances, au sein de ce que nous appelons un *système de connaissance*. L'idée que nous proposons d'utiliser dans cette thèse est de modéliser les connaissances au niveau du système et de les faire « véhiculer » d'un modèle à un autre par *transition de modèles*. Un système de connaissance basé sur un ensemble de modèles de connaissances repose

sur deux piliers : un ensemble de notions et des règles d'instanciation de notions. Une *règle d'instanciation de notion* spécifie pour une notion donnée comment elle est instanciée au niveau de chaque modèle. Ainsi, les connaissances d'une modélisation sont réparties au sein des modèles du système de connaissance, de sorte que l'application des règles d'instanciation de notions constitue les transitions de modèles pour les connaissances. L'expressivité d'un système de connaissance est donc fonction de ces notions et de leurs instanciations possibles dans les modèles du système.

L'intérêt de l'approche à transition de modèles dans cette thèse comparée aux approches de transformation et d'extension de modèles est le suivant. Par rapport à l'utilisation de transformations de modèles la contribution est d'exploiter la complémentarité des modèles tant en terme de modélisation que de raisonnement. Les transitions de modèles ont pour objectif de bénéficier des différences d'expressité entre les modèles et d'aboutir à un système au pouvoir expressif fort. Là où les seules connaissances modélisables par transformations successives d'un modèle à un autre seraient celles figurant dans le plus petit dénominateur commun des connaissances modélisables par les modèles considérés, les transitions de modèles ne génère aucune perte de connaissance modélisée. Par rapport à l'utilisation d'un modèle de connaissance étendu la contribution est triple. Premièrement, un système de connaissance est relativement aisé et rapide à mettre en place. Chaque modèle du système de connaissance est utilisé « simplement » dans la limite de ses capacités tant représentationnelles qu'inférentielles. C'est de leur utilisation conjointe que provient ce surcroît d'expressivité et de capacité de raisonnement. Deuxièmement, l'évolution du système de connaissance est ouverte aux évolutions des modèles - soutenus par leurs communautés respectives - qui le composent. Troisièmement, l'intégration d'un nouveau modèle nécessitée par des travaux ultérieurs est toujours possible dans le système de connaissance.

Objectif

Dans cette thèse, nous procédons à la mise en place d'un système de connaissance basé sur trois modèles de connaissances : le modèle des graphes conceptuels [Sowa, 1984; Mugnier et Chein, 1996] noté *GCs*, le modèle noté *OWL* constitué du langage OWL associé à des raisonnements en logiques de descriptions [Baader *et al.*, 2003] et le modèle noté *UML* constitué du langage UML [Booch *et al.*, 1998] couplé à des raisonnements en OCL [Warmer et Kleppe, 1998]. Le cœur du travail de la thèse consiste donc aux choix des notions instanciables dans ce système de connaissance et à l'élaboration des règles d'instanciation de ces notions dans ces trois modèles.

La Figure 2 présente le schéma de principe de notre système de connaissance, où l'utilisateur ou la machine « passe » d'un modèle à un autre au gré de ses besoins, en fonction des capacités offertes par chaque modèle sur le plan modélisation et raisonnement. Cette figure montre la modélisation des connaissances d'un domaine au sein du système de connaissance. Cette modélisation, notée K sur la figure, est composée de l'union des connaissances partielles K_{OWL}, K_{GCs} et K_{UML} issues des trois modèles de

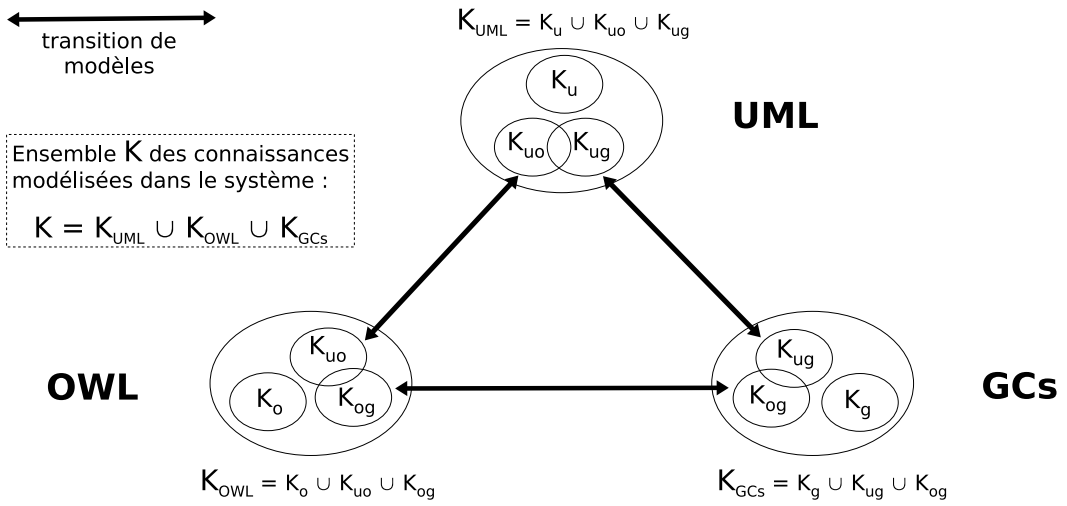


Figure 2 – Schéma de principe du *système de connaissance*.

connaissances. Cette répartition est le résultat produit par l'application des règles d'instanciation de notions. La modélisation partielle $K_{OWL} = K_o \cup K_{og} \cup K_{uo}$ des connaissances du domaine dans le modèle OWL n'est pas identique à la modélisation partielle $K_{GCs} = K_g \cup K_{og} \cup K_{ug}$ (respectivement $K_{UML} = K_u \cup K_{uo} \cup K_{ug}$) en GCs (respectivement en UML). Une partie des connaissances modélisées K_{og} (respectivement K_{uo}) est équivalente dans GCs (respectivement dans UML). Une autre partie des connaissances $K_{uo} \setminus K_{og}$ (respectivement $K_{og} \setminus K_{uo}$) ne peut pas être modélisée dans GCs (respectivement dans UML). Enfin, K_o n'est modélisable ni dans GCs ni dans UML.

Le choix des trois modèles OWL, GCs et UML est motivé par plusieurs raisons. Tout d'abord, OWL, GCs et UML sont des modèles de connaissances bien connus, qui permettent de modéliser des connaissances sur deux niveaux conceptuels. Ensuite, il existe des outils logiciels pour modéliser des connaissances et raisonner sur ces connaissances modélisées en OWL, GCs et UML. Enfin, ces trois modèles disposent de suffisamment de notions communes pour en permettre une utilisation conjointe, ainsi que de spécificités propres complémentaires.

Ces notions communes que nous avons dégagées sont brièvement les suivantes. Au niveau structurel d'une modélisation, les notions instanciables sont en terme de concepts génériques celles de classe, de datatype, et en terme d'associations génériques celles de relation et de slot. Une *classe* est une entité générique ayant des caractéristiques communes à un ensemble d'individus. Un *datatype* est une entité générique regroupant un ensemble de données, comme les entiers naturels ou les chaînes de caractères. Une *relation binaire* est une association générique entre deux classes, l'une appelée le domaine et l'autre le co-domaine. Un *slot* est une association générique entre une classe, appelée son domaine, et un dadatype, appelé son co-domaine. Au niveau factuel d'une modélisation, les notions instanciables sont celles des *assertions*. Il s'agit des notions d'individu, de donnée, de lien et d'attribution. Un *individu* est une assertion (ou instance) d'une ou plusieurs

classes, possédant ses caractéristiques propres. Une *donnée* est un élément « brut » typée par un datatype. Un *lien* est une assertion d'une relation binaire qui lie deux individus. Une *attribution* est une assertion de slot. Lien et attribution sont tels que le premier argument est une assertion du domaine de la relation ou du slot, et le second argument est une assertion du co-domaine.

La synergie créée entre les modèles OWL, GCs et UML peut être succinctement vue de la manière suivante. UML est très largement visuel et intuitif pour l'utilisateur de par ses schématisations sous forme de diagrammes. Il peut être privilégié par l'utilisateur lors de la création d'une modélisation des connaissances d'un domaine ou de sa mise à jour « manuelle ». OWL, basé sur des logiques de descriptions, permet d'aider l'utilisateur, par des raisonnements de *classification* et d'*instanciation*, à organiser ou réorganiser en une ontologie la structure d'une modélisation au cours de son cycle de vie et à valider les connaissances factuelles modélisées sur cette structure. Pour l'exploitation des connaissances modélisées, le modèle GCs offre la possibilité, via l'opérateur de *projection*, de rechercher des connaissances parmi celles modélisées, et de façon complémentaire aux deux autres modèles l'application de règles et la vérification de contraintes.

Naturellement d'autres travaux se sont déjà penchés sur une utilisation conjointe deux à deux des modèles OWL, GCs et UML. Tout d'abord, un grand nombre de travaux ont combiné logiques de descriptions et logique classique, et ont donc indirectement contribué à développer l'axe OWL−GCs. Le modèle des GCs est en effet muni d'une sémantique logique exprimable avec une sous partie de la logique des prédicats [Chein et Mugnier, 1992; Mugnier et Chein, 1996]. Citons par exemple CARIN [Levy et Rousset, 1998] qui associe la logique de description $\mathcal{ALCNR}$ [1] aux règles de Horn, ou d'autres applications antérieures comme dans [Brachman *et al.*, 1985; Falkenhainer et Forbus, 1991; Catarci et Lenzerini, 1993]. Plus récemment et dans la mouvance du MDA [2], se sont développées des transformations de modèles contribuant au développement des axes UML−OWL et UML−GCs. Par exemple, les travaux dans [Corby *et al.*, 2000] et [Raimbault *et al.*, 2006a] transforment des connaissances en OWL ou du Web sémantique dans GCs pour effectuer des recherches ou des vérifications sur ces connaissances. Les travaux dans [Raimbault *et al.*, 2006b; Raimbault *et al.*, 2005] s'appuient sur GCs ou plus généralement sur la logique du premier ordre dans [Roth et Schmitt, 2007] pour raisonner sur des connaissances en UML. Le travail global de transitions de modèles de cette thèse, pour l'obtention d'un système de connaissance, peut être vu comme trois contributions exploitant des complémentarités de modèles sur chacun des axes OWL−GCs, UML−OWL et UML−GCs.

Organisation de la thèse

Cette thèse est composée de huit chapitres organisés de le manière suivante.

[1] moins expressive cependant que le modèle OWL (voir le Chapitre 2)

[2] MDA - *Model Driven Architecture* - est une démarche de réalisation logiciel, proposée par l'OMG et basée sur UML, qui consite en l'élaboration de modélisations indépendantes de toute plate-forme (PIM) puis par *transformations de modèles* vers des modélisations dépendantes de plates-formes (PSM).

Les trois premiers chapitres exposent un état de l'art sur les trois modèles de connaissances utilisés dans cette thèse : le modèle des graphes conceptuels, le langage OWL associé aux logiques de descriptions, et le langage UML couplé à OCL.

Les Chapitres 4, 5 et 6 forment le cœur de la thèse en montrant comment les transitions de connaissances entre les modèles OWL, GCs et UML peuvent s'opérer. Le Chapitre 4 définit les règles d'instanciation pour les notions de classe et de datatype dans ces trois modèles. Le Chapitre 5 définit les règles d'instanciation pour les notions de relation et de slot. Le Chapitre 6 définit les règles d'instanciation pour les notions d'individu et de donnée d'une part et de lien et d'attribution d'autre part.

Les deux derniers chapitres présentent le système de connaissance réalisé, basé sur les règles d'instanciation de notions dans les modèles OWL, GCs et UML. Le Chapitre 7 fournit des modes d'utilisations complémentaires de ces trois modèles dans le système de connaissance. Ceux-ci concernent la modélisation, l'interrogation, la déduction et la vérification de connaissances. Cette complémentarité est une conséquence directe de l'exploitation des transitions de modèles produites entre OWL, GCs et UML. Le Chapitre 8 présente KR-suite une implémentation en C++ de notre système de connaissance qui nous permet de valider dans la pratique ce travail de thèse.

Chapitre 1

Modèle des graphes conceptuels

Le modèle des graphes conceptuels (GCs) est un modèle formel de représentation des connaissances, qui tient son origine des réseaux sémantiques [Quillian, 1969; Lehmann, 1992]. Un modèle des GCs est introduit dans [Sowa, 1984] et formalisé dans [Chein et Mugnier, 1992; Mugnier et Chein, 1996]. Plusieurs extensions de ce modèle ont été proposées formant ainsi une famille de modèles de GCs selon les extensions utilisées, comme les types conjonctifs [Chein et Mugnier, 2004], les règles [Salvat, 1998] et les contraintes [Baget et Mugnier, 2002].

De façon générale, l'utilisation du modèle des GCs pour modéliser les connaissances d'un domaine se fait en deux étapes : d'abord la définition d'un *support* qui spécifie les connaissances structurelles du domaine, ensuite la création de *graphes conceptuels* qui en modélisent les connaissances factuelles. Cette séparation explicite de différents types de connaissances entraîne une grande clarté lors de l'utilisation de ce modèle pour l'acquisition ou la représentation de connaissances.

Un des intérêts de ce modèle est de permettre de représenter des connaissances de façon graphique. Plus précisément, un graphe conceptuel est un graphe biparti étiqueté ; les deux classes de sommets étant étiquetés respectivement par des noms de *concepts* et des noms de *relations* conceptuelles entre ces concepts. Le support représente sous la forme d'un ou plusieurs arbres le lien « sorte-de » qui organise les connaissances structurelles en taxonomies. Une telle représentation graphique des connaissances permet à des utilisateurs de comprendre, créer ou modifier directement des connaissances, de façon plus intuitive qu'avec une représentation sous forme de formules logiques par exemple.

Un autre intérêt du modèle vient du fait que des raisonnements peuvent être effectués sur les connaissances représentées. Ces raisonnements peuvent être vus soit comme des opérations de graphes, soit comme des inférences logiques. Dans le premier cas, les graphes conceptuels sont considérés comme des graphes étiquetés et les raisonnements se basent sur des travaux d'algorithmique de graphe. Dans le second cas, un graphe conceptuel est considéré comme une représentation graphique d'une formule logique ; les raisonnements sont alors effectués sur les formules logiques par un démonstrateur logique. Le modèle présenté ici se place dans le premier cas, ce qui n'empêche pas le modèle d'être logiquement formé : ce modèle dipose en effet d'une sémantique logique qui

est adéquate et complète en logique du premier ordre. Ainsi, les raisonnements effectués par des opérations de graphes ou par des déductions logiques sont équivalents.

Ce chapitre donne dans une première section une introduction au modèle dit de base des GCs, en présentant le support, les graphes conceptuels, la projection et la sémantique logique du modèle. Dans une seconde section, des extensions majeures du modèle de base des GCs sont présentées, comme les règles, les contraintes positives et négatives, et la notion de types conjonctifs. À ces extensions, nous présentons notre adaptation et extension du modèle des GCs pour un assouplissement de l'utilisation des marqueurs individuels et pour la prise en compte de *données*. Une troisième et dernière section présente quelques outils logiciels pour l'édition des graphes conceptuels, pour la manipulation de graphes par les opérations du modèle, ou pour le développement d'applications basées sur le modèle des GCs.

Sommaire

1.1 Le modèle de base

Nous présentons dans cette section un modèle de base des GCs [Mugnier et Chein, 1996], aussi appelé le *modèle des GCs simples*. Le support, les graphes conceptuels et l'opération de projection y sont abordés d'un point de vue graphique et logique.

1.1.1 Support

Le support définit, pour un domaine donné, les connaissances structurelles permettant la représentation de connaissances factuelles sous forme de graphes conceptuels. Un graphe conceptuel (voir la section suivante) donné n'a de sens que par rapport au support sur lequel il est défini.

Définition 1.1 (Support) *Un support est un quintuplet $S = (T_C, T_R, \sigma, I, \tau)$ [Mugnier et Chein, 1996], où :*

- *T_C est l'ensemble partiellement ordonné de* types de concepts, *où $\top$ est le plus grand élément.*
- *T_R l'ensemble de* types de relations *partitionné : $T_R = T_{R_{i_1}} \cup \cdots \cup T_{R_{i_p}}$, où $T_{R_{i_j}}$ est l'ensemble des types de relations d'arité i_j pour $i_j > 0$. Chaque $T_{R_{i_j}}$ est partiellement ordonné et possède un plus grand élément noté $\top_{i_j}$. La relation d'ordre pour les ensembles des types de concepts et des types de relations est notée $\leq$.*
- *σ est une application associant à chaque type de relation t_r une* signature, *qui spécifie l'arité de t_r et le plus grand type possible pour chaque argument. Plus précisément, à tout $t_r \in T_{R_{i_j}}$, σ associe un uplet $\sigma(t_r) \in (T_C)^{i_j}$, et vérifie : pour tous t_{r_1} et t_{r_2} de $T_{R_{i_j}}$, si $t_{r_1} \leq t_{r_2}$ alors $\sigma(t_{r_1}) \leq \sigma(t_{r_2})$ (où l'ordre considéré sur les signatures est l'ordre produit sur $(T_C)^{i_j}$. Autrement dit, le type associé au $k^{ième}$ argument de t_{r_1} est inférieur ou égal au type associé au $k^{ième}$ argument de t_{r_2}). On note $\sigma_i(t_r)$ le $i^{ème}$ argument de $\sigma(t_r)$.*
- *I est l'ensemble des* marqueurs individuels. *À cet ensemble I s'ajoute un* marqueur générique, *noté $*$, qui permet de représenter un individu non spécifié. $I \cup \{*\}$ est muni de l'ordre suivant : $*$ est plus grand que tous les marqueurs individuels, et ces derniers sont deux à deux incomparables.*
- *τ est une application, dite de* conformité, *de I dans T_C, qui à tout marqueur individuel m associe un type de concept t.*

Exemple

Un exemple de support est donné par les Figures 3 et 4 ainsi que par la définition de l'ensemble des marqueurs individuels $I = \{$ Serieys, Alphand, Mitsubishi, dakar2007, AtârTichit, TichitNéma $\}$ et la relation de conformité $\tau = \{$ (Serieys,Directeur), (Alphand,Pilote), (Mitsubishi,Équipe), (dakar2007,Rallye), (AtârTichit,Étape), (TichitNéma,Étape) $\}$. La relation d'ordre $\leq$ est représentée, en Figures 3 et 4, par une flèche et s'interprète dans le sens de lecture de la flèche avec la signification « sorte-de ». Par exemple, le concept Auto est une sorte-de Véhicule. On dit que Auto *est subsumé par* Véhicule, ou que Véhicule *subsume* Auto ; que Véhicule est le *subsumant*, et Auto le *subsumé*. Pour chaque type de relation, la signature σ est donnée.

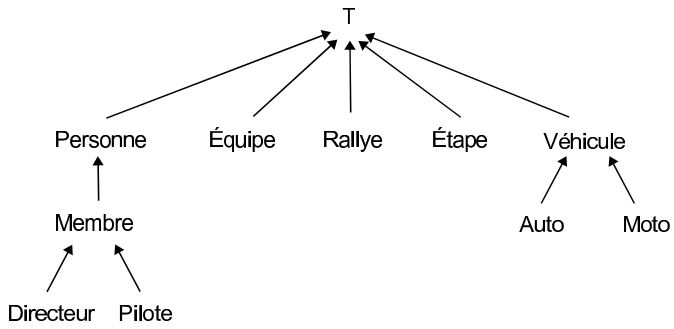


Figure 3 – Un ensemble partiellement ordonné de types de concepts (T_C).

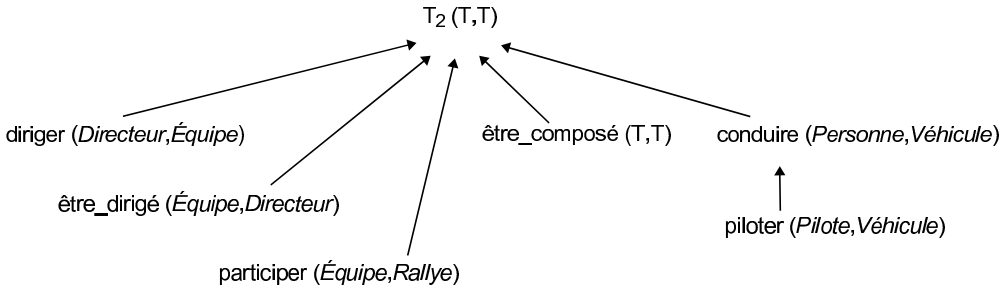


Figure 4 – Un ensemble partiellement ordonné de types de relations binaires (T_R) et leurs signatures (σ).

Ainsi, conduire a pour premier argument une Personne (ou tout subsumé) et pour second argument un Véhicule (ou tout subsumé).

1.1.2 Graphe conceptuel

Un graphe conceptuel représente des connaissances factuelles d'un domaine donné. Ces connaissances constituent une représentation partielle des faits attachés au domaine. Un graphe conceptuel est défini sur un support qui représente les connaissances structurelles du domaine.

Définition 1.2 (Graphe conceptuel) *Un* graphe conceptuel $G = (C, R, U, etiq)$ *[Chein et Mugnier, 1992; Mugnier et Chein, 1996] défini sur un support* $S = (T_C, T_R, \sigma, I, \tau)$ *est un multigraphe non orienté, biparti, non nécessairement connexe, où :*

- *C est l'ensemble des* sommets concepts *et R l'ensemble des* sommets relations*. Ces deux ensembles sont disjoints ($C \cap R = \emptyset$).*
- *U est l'ensemble des* arêtes*. Toutes les arêtes de G ont une extrémité dans R et l'autre extrémité dans C.*
- *etiq est une application qui à tout sommet et à toute arête associe une étiquette, tel que :*
  - *$\forall r \in R$, $etiq(r) \in T_R$*
  - *$\forall c \in C$, $etiq(c) \in T_C \times (I \cup \{*\})$. L'étiquette d'un sommet concept c est le couple*

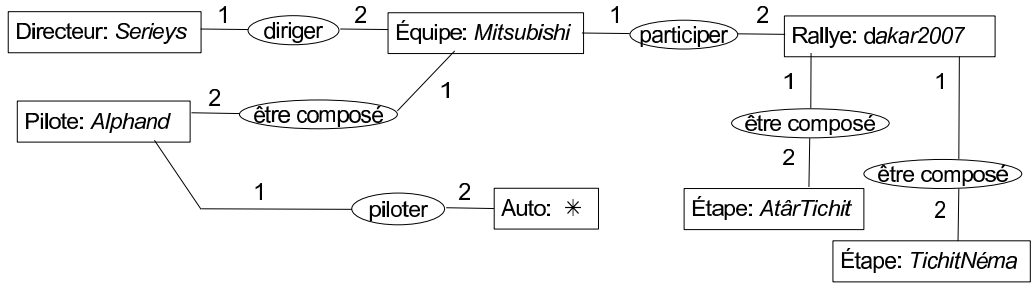


Figure 5 – Un graphe conceptuel, défini sur le support présenté en Section 1.1.1.

(type(c), marqueur(c)). Un sommet concept tel que marqueur(c) $\in I$ *est appelé* sommet concept individuel*, dans le cas contraire (marqueur(c)* $= *$*) il est appelé* sommet concept générique*. Il existe un ordre partiel sur les étiquettes des sommets concepts, qui est l'ordre produit des ordres de* T_C *et* $I \cup \{*\}$ *: étant données deux étiquettes* $e = (t, m)$ *et* $e' = (t', m')$*,* $e \leq e'$ *si et seulement si* $t \leq t'$ *et* $m \leq m'$*.*

- $\forall e \in U, etiq(e) \in \mathbb{N}$*.*

- *De plus,* $etiq$ *obéit aux contraintes fixées par* σ *et* τ *:*
  - *L'ensemble des arêtes adjacentes à tout sommet relation* r *est totalement ordonné, ce que l'on représente en étiquetant les arêtes par une numérotation de 1 à l'arité de* r*. On note* $G_i(r)$ *le* $i^{ème}$ *voisin de* r *dans* G*.*
  - $\forall r \in R$*,* $type(G_i(r)) \leq \sigma_i(type(r))$
  - $\forall c \in C$ *si* $marqueur(c) \in I$ *alors* $\tau(marqueur(c)) \leq type(c)$

Exemple

Le graphe conceptuel en Figure 5, défini sur le support présenté en Section 1.1.1, représente les connaissances factuelles suivantes. Le directeur Serieys dirige l'équipe Mitsubishi, qui participe au rallye dakar2007. Ce rallye est composé (entre autres) des étapes AtârTichit et TichitNéma. Le pilote Alphand est un membre de l'équipe Mitsubishi, et pilote une Auto. Remarquons que l'Auto en question, représentée par un sommet concept générique, n'est pas identifiée mais qu'elle existe.

Notons que l'étiquette d'une arête est souvent remplacée par une flèche dans le cas où les relations sont d'arité 2. Ainsi, l'arête étiquetée *1* est représentée par une flèche d'un sommet concept vers le sommet relation, et l'arête étiquetée *2* par une flèche du sommet relation vers un sommet concept. Cette notation sera utilisée par la suite, car seules des relations binaires sont utilisées dans cette thèse.

1.1.3 Forme normale

Il existe une forme particulière pour un graphe conceptuel, nommée *forme normale*.

Définition 1.3 (Forme normale [Mugnier et Chein, 1996]) *Un graphe conceptuel est sous*

forme normale *si et seulement si il n'existe pas deux sommets concepts portant le même marqueur individuel .*

Tout graphe conceptuel peut être mis sous forme normale. Concrètement, la mise sous forme normale consiste à *fusionner* les sommets concepts individuels du graphe ayant le même marqueur individuel et en gardant pour type celui le plus spécialisé. La Figure 6 donne un exemple où le graphe conceptuel G_2 est la (mise sous) forme normale du graphe conceptuel G_1. Cette action permet, comme son nom l'indique, la normalisation de graphes conceptuels sémantiquement équivalents mais non nécessairement équivalents au départ dans leurs compositions, c'est-à-dire concernant le nombre de sommets et d'arêtes, les étiquettes de ces sommets et arêtes, et les associtions entre ces sommets. La mise sous forme normale s'avère utile notamment comme étape préliminaire avant la recherche de projections d'un graphe dans un autre (voir la section ci-après).

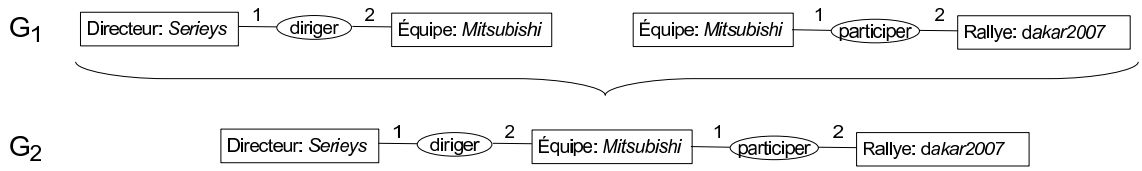


Figure 6 – Mise du graphe G_1 sous *forme normale* : le graphe G_2.

1.1.4 Projection

L'opération de projection est l'opération de base du modèle et correspond à un morphisme de graphes. La recherche d'une projection d'un graphe H dans un graphe G peut être vue comme la recherche de « l'inclusion » de l'information représentée par H dans G. Si tel est le cas, on dit que G est une spécialisation de H ou H subsume G. Notons que la recherche de projections sont de façon générale des problèmes NP-complets [Chein et Mugnier, 1992].

Définition 1.4 (Projection) *Une projection [Chein et Mugnier, 1992; Mugnier et Chein, 1996] d'un graphe conceptuel $H = (C_H, R_H, U_H, etiq_H)$ dans un graphe conceptuel $G = (C_G, R_G, U_G, etiq_G)$, tous deux définis sur un même support, est un couple d'applications $\Pi = (f, g)$, avec $f : C_H \rightarrow C_G$ et $g : R_H \rightarrow R_G$, tel que :*

- Π *peut spécialiser les étiquettes des sommets :*
  - $\forall c \in C_H$, $etiq_G(f(c)) \leq etiq_H(c)$,
  - $\forall r \in R_H$, $etiq_G(g(r)) \leq etiq_H(r)$.
- Π *préserve les arêtes et leurs étiquettes : pour toute arête rc de U_H, $\Pi(rc)$ est une arête de U_G de même étiquette. De plus si $c = H_i(r)$, alors $f(c) = G_i(g(r))$.*

De par cette définition, la projection est un morphisme de graphes qui conserve la bipartition : l'image d'un sommet concept est un sommet concept et l'image d'un sommet relation est un sommet relation. De plus, la condition sur les étiquettes fait intervenir les connaissances représentées dans le support, ainsi, l'étiquette de l'image d'un sommet est identique ou une spécialisation de l'étiquette de son antécédent.

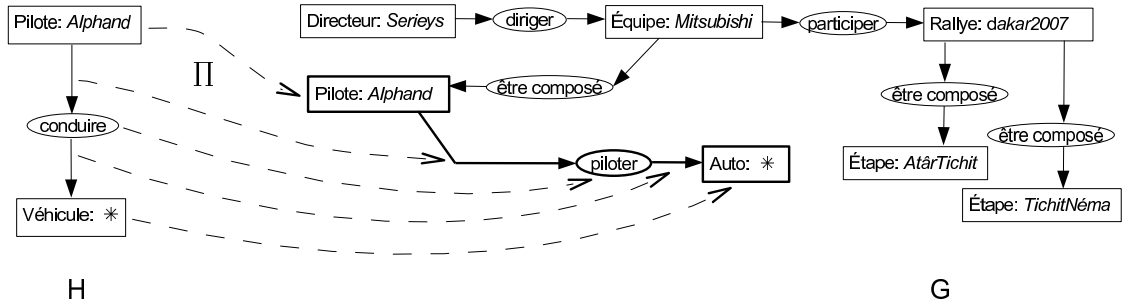


Figure 7 – Une projection Π du graphe conceptuel H dans le graphe conceptuel G ; H et G sont défini sur le même support présenté en Section 1.1.1.

Exemple

La Figure 7 illustre la projection Π du graphe H dans le graphe G, H et G étant tous deux définis sur le support présenté en Section 1.1.1. Cette projection fait correspondre à H le sous-graphe de G mis en gras. Cette projection fait correspondre l'individu Alphand avec exactement l'individu Alphand, l'existence d'un individu de type Véhicule avec l'existence d'un individu du type subsumé Auto et la relation conduire avec la relation subsumée piloter. Le correspondance entre conduire et piloter respecte les étiquettes (la numérotation) des arêtes.

On notera que pour une base de connaissances formée d'un support (la structure) et d'un ou plusieurs graphes conceptuels (les faits), l'interrogation de cette base est une utilisation immédiate de l'opération de projection. Une question est formulée par un graphe conceptuel, et l'ensemble des projections de la question sur les faits constitue l'ensemble des réponses à la question. Ici, le graphe conceptuel H en Figure 7 peut correspondre à la question « Le pilote Alphand conduit-il un véhicule ? ». L'existence d'une projection fournit une réponse booléenne positive, et la projection elle-même apporte la preuve à cette réponse : « (Oui, et plus précisément) Le pilote Alphand pilote une auto ».

1.1.5 Sémantique logique

Il est possible d'associer à un support et à tout graphe conceptuel une formule de la logique du premier ordre par le biais de l'opérateur Φ [Sowa, 1984]. Dans ce cas, l'existence d'une projection de H dans G définis sur le support S est équivalente à $\Phi(S), \Phi(G) \models \Phi(H)$. L'adéquation de la projection par rapport à la déduction en logique du premier ordre est montrée dans [Sowa, 1984], la complétude est montrée quant à elle dans [Chein et Mugnier, 1992] [1].

À tout type de concept du support, Φ associe un prédicat unaire et à tout type de relation un prédicat de même arité que le type. À un marqueur individuel est associée une constante. Dans un graphe conceptuel, Φ associe à tout sommet concept un terme

[1] La preuve de la complétude n'est cependant valable que si G est sous forme normale, comme cela est précisé dans [Mugnier et Chein, 1996].

(variable ou constante). À chaque sommet concept générique est associée une nouvelle variable, alors qu'à tous les sommets concepts individuels ayant même marqueur est associée la même constante.

Plus précisément, à un support S, Φ associe un ensemble de formules de la logique du premier ordre, notée $\Phi(S)$, construit de la façon suivante :

- au type universel $\top$ de T_C, on associe la formule $\forall x.\top(x)$,
- à tout couple de types de concepts t_1 et t_2 de T_C tel que $t_1 \leq t_2$ on associe la formule $\forall x\ t_1(x) \rightarrow t_2(x)$ (ces formules correspondent aux liens « sorte-de » de l'ensemble des types de concept),
- à tout couple de types de relations t_1 et t_2 de T_{R_p} tel que $t_1 \leq t_2$ correspond la formule $\forall x_1 \dots x_p\ t_1(x_1, \dots, x_p) \rightarrow t_2(x_1, \dots, x_p)$, où p est l'arité de t_1 et t_2,
- à toute signature $\sigma(t_r) = (t_1, \dots, t_p)$ d'un type de relation t_r de T_{R_p}, on associe la formule $\forall x_1 \dots x_p\ t_r(x_1, \dots, x_p) \rightarrow t_1(x_1) \wedge \dots \wedge t_p(x_p)$,
- pour tout m de I, si c est la constante associée à m, alors on a $\tau(m)(c)$.

Par exemple, pour le support considéré en 1.1.1, l'ensemble des formules $\Phi(S)$ est constitué de la façon suivante :
$\forall x\ \top(x)$
$\forall x\ \textit{Personne}(x) \rightarrow \top(x)$
$\forall x\ \textit{Équipe}(x) \rightarrow \top(x)$
...
$\forall x\ \textit{Membre}(x) \rightarrow \textit{Personne}(x)$
$\forall x\ \textit{Directeur}(x) \rightarrow \textit{Membre}(x)$
...
$\forall x\ \forall y\ \textit{diriger}(x, y) \rightarrow \top_2(x, y)$
$\forall x\ \forall y\ \textit{conduire}(x, y) \rightarrow \top_2(x, y)$
$\forall x\ \forall y\ \textit{piloter}(x, y) \rightarrow \textit{conduire}(x, y)$
...
$\forall x\ \forall y\ \top_2(x, y) \rightarrow \top(x) \wedge \top(y)$
$\forall x\ \forall y\ \textit{diriger}(x, y) \rightarrow \textit{Directeur}(x) \wedge \textit{Équipe}(y)$
...
Directeur(Serieys)
Rallye(dakar2007)
...

À tout graphe conceptuel G sur un support S, Φ associe une formule de la logique du premier ordre, notée $\Phi(G)$, construite comme suit :

- à un sommet concept c, on associe l'atome $t_c(id_c)$, tel que t_c est le prédicat associé au type de c et id_c le terme (variable ou constante) associé à c,
- à un sommet relation r, on associe l'atome $t_r(id_1, \dots, id_p)$, où t_r est le prédicat associé au type de r, p l'arité de ce prédicat, et chaque id_i le terme associé au $i^{\text{ème}}$ voisin de r dans G,
- la formule $\Phi(G)$ est construite en faisant la fermeture existentielle de la conjonction de ces atomes.

	Syntaxe $^{(*)}$	**Sémantique**
type universel	$\top$	$\forall x \top(x)$
hiérarchie de TCs	$t_{c_1} \leq t_{c_2}$	$\forall x\, t_{c_1}(x) \rightarrow t_{c_2}(x)$
hiérarchie de TRs	$t_{r_1} \leq t_{r_2}$	$\forall x_1, ..., x_p\, t_{r_1}(x_1, ..., x_p) \rightarrow t_{r_2}(x_1, ..., x_p)$
signature	$\sigma(t_r) = (t_{c_1}, ..., t_{c_n})$	$\forall x_1, ..., x_n\, t_r(x_1, ..., x_n) \rightarrow t_{c_1}(x_1) \wedge ... \wedge t_{c_n}(x_n)$
conformité	$\tau(m) = t$	$t(m)$

TC : type de concept, TR : type de relation
$^{(*)}$ t_{c_i} sont des noms de types de concepts, t_{r_i} des noms de types de relations, et m un marqueur individuel.

Tableau 1 – Syntaxe et sémantique logique du support.

	Syntaxe $^{(*)}$	**Sémantique**
SC individuel	$\boxed{t_c : m}$	$t_c(m)$, où m est la constante associée au SC individuel
SC générique	$\boxed{t_c : *}$	$\exists x\, t_c(x)$, où x est la variable associée au SC ; une variable distincte est associée à chaque SC générique.
SR unaire	$(t_r) \overset{1}{—} C_1$	$t_r(id_1)$, où id_1 est le terme associé au SC C_1
SR binaire	$C_1 \overset{1}{—} (t_r) \overset{2}{—} C_2$	$t_r(id_1, id_2)$, où id_1 et id_2 sont les termes associés aux SCs C_1 et C_2
SR n-aire	$(t_r) \overset{1}{—} C_1$ avec $2 / C_2$ ... $\backslash^n C_n$	$t_r(id_1, ..., id_n)$, avec n l'arité de t_r et $id_1, ..., id_n$ les termes associés aux voisins du SR

SC : sommet concept, SR : sommet relation
$^{(*)}$ t_{c_i} sont des noms de types de concepts, t_{r_i} des noms de types de relations, et m un marqueur individuel.

Tableau 2 – Syntaxe et sémantique logique d'un graphe conceptuel.

Par exemple, la formule logique associée par Φ au graphe H en Figure 7 est :

$$\Phi(H) = \exists x\, \textit{Pilote}(\textit{Alphand}) \wedge \textit{Véhicule}(x) \wedge \textit{conduire}(\textit{Alphand}, x)$$

La sémantique, ainsi que la syntaxe, du support et d'un graphe conceptuel sont respectivement récapitulées dans les Tableaux 1 et 2.

L'opérateur Φ ainsi défini permet d'énoncer le théorème d'adéquation et de complétude de Φ :
Soit $\Phi(S)$ l'ensemble des formules associées au support. Soient G et H deux graphes conceptuels sous forme normale, alors il existe une projection de H dans G si et seulement si $\Phi(S), \Phi(G) \vdash \Phi(H)$.

Ceci peut s'illustrer sur l'exemple de projection donné en figure 7, en considérant les formules logiques associées au support et aux graphes.

Nous pouvons, à ce stade de la présentation du modèle des GCs, faire la remarque suivante. Au sein de la communauté « graphes conceptuels », deux points de vue se

dégagent, selon que l'accent est mis sur l'aspect logique du modèle ou sur l'aspect graphique :

- Un graphe conceptuel est considéré comme une représentation graphique d'une formule logique. Ce choix est motivé par le fait que le modèle est doté d'une sémantique en logique du premier ordre. Dans ce cas, les raisonnements faisant intervenir des connaissances représentées par des graphes conceptuels ne se font pas directement sur les graphes mais sur les formules logiques correspondantes, en utilisant des outils tels que des démonstrateurs logiques.
- Un graphe conceptuel est considéré comme un graphe étiqueté représentant des connaissances. Dans ce cas, les raisonnements sont basés sur des opérations de graphes, en utilisant des algorithmes issus de la théorie des graphes. Ces opérations de graphes sont souvent définies de telle façon qu'elles produisent les mêmes résultats que ceux basés sur les formules données par la sémantique logique des graphes.

Dans les deux cas, la représentation de connaissances sous forme de graphes conceptuels favorise la lecture par un humain non familier des notations logiques, la représentation graphique du raisonnement produit par la projection (comme en Figure 7) est elle aussi plus simple à interpréter qu'une déduction logique basée sur les formules associées au support et aux graphes.

1.2 Extensions du modèle des GCs

Nous présentons dans cette section certaines extensions du modèle de base des GCs. Plus précisément, nous abordons les notions de règle, de contraintes positive et négative, et de type conjonctif.

1.2.1 Règle

Les règles de graphes conceptuels [Salvat et Mugnier, 1996; Salvat, 1997] [Salvat, 1998] constituent une extension au modèle des GCs simples et permettent de représenter des connaissances sous la forme de règles d'inférence du type « si l'information A est présente dans un graphe, alors l'information B peut être ajoutée au graphe ». A et B sont exprimées sous forme de graphes conceptuels liés par des liens de coréférence entre certains sommets concepts.

Un *lien de coréférence* [Sowa, 1984; Chein et Mugnier, 2004] entre deux sommets concepts indique que ces deux sommets représentent le même individu. Un lien de coréférence est représenté par une ligne en pointillés sur les dessins de graphes.

La définition d'une règle de graphe conceptuel fait intervenir la notion de *lambda abstraction* : une lambda abstraction $\lambda x_1 \dots x_n\ G$ ($n \geq 0$), est constituée d'un graphe conceptuel G dans lequel n sommets concepts génériques ont été distingués.

La représentation d'une règle est repérée par $\boxed{\Rightarrow}$. L'hypothèse et la conclusion de la règle sont séparées par une ligne verticale, l'hypothèse se trouvant au dessus de la ligne et la conclusion en dessous.

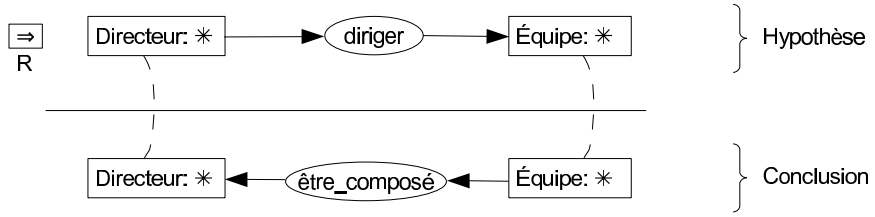


Figure 8 – Une règle R.

Définition 1.5 (Règle) *Une règle $R : H_R \Rightarrow C_R$ est un couple de lambda abstractions ($\lambda x_1 \ldots x_n$ H_R, $\lambda x_1 \ldots x_n$ C_R).*
*H_R est appelé l'*hypothèse *de R et C_R la* conclusion *de R. $x_1, \ldots, x_n$ sont appelés les* points d'attache*. Ils correspondent aux liens de coréférence entre R_H et R_C. L'occurrence d'un x_i dans H_R (respectivement dans C_R) sera notée $x_i^{H_R}$ (respectivement $x_i^{C_R}$).*

Notons que l'application de règles, nécessitant la la recherche de projections, est de façon générale un problème NP-complets [Chein et Mugnier, 1992].

Exemple

La règle R en Figure 8 représente la connaissance suivante : « Si un directeur x dirige une équipe y, alors y est composée (entre autre) de x ». En d'autres termes, toute personne qui dirige une équipe est un membre de cette équipe.

1.2.1.1 Application d'une règle

Si les règles de graphes permettent de représenter de la connaissance implicite, cette extension du modèle définit aussi une opération permettant d'appliquer une règle sur un graphe, celle-ci étant basée sur des opérations de graphes. L'application d'une règle correspond à un enrichissement du fait représenté par le graphe avec les connaissances implicites représentées par la règle.

Définition 1.6 (Application d'une règle) *Un graphe conceptuel G* vérifie l'hypothèse *d'une règle $R : H_R \Rightarrow C_R$ s'il existe une projection Π de H_R dans G.*
*Si tel est le cas, l'*application de R sur G selon Π *produit un graphe conceptuel formé de G et de C_R dans lequel pour tous les couples de sommets $(x_i^{C_R}, \Pi(x_i^{H_R}))$ (pour $i \in [1, n]$), une fusion de $x_i^{C_R}$ sur $\Pi(x_i^{H_R})$ a été effectuée.*

La règle de la Figure 8 peut être appliquée sur le graphe conceptuel de la Figure 5 (d'une seule façon puisqu'il n'existe qu'une projection de l'hypothèse de R dans le graphe) pour donner le graphe de la Figure 9. On constate l'ajout du sommet relation (être_composé) issu de la règle, après fusion des sommets concepts génériques [Directeur: $*$] et [Équipe: $*$] de la règle avec respectivement les sommets concepts [Directeur: Serieys] et [Équipe: Mitsubishi] du graphe conceptuel en en Figure 5.

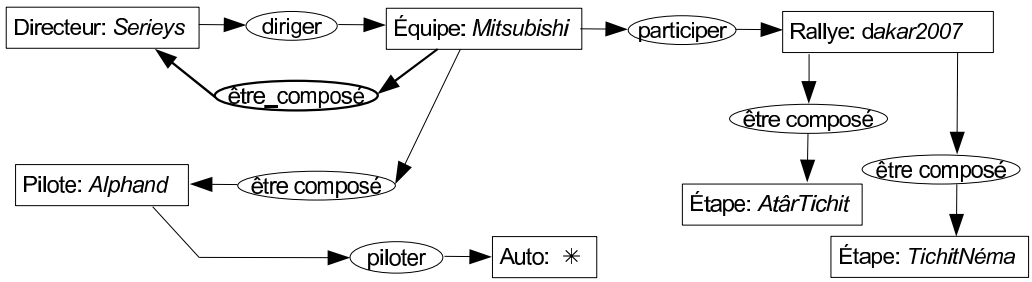


Figure 9 – Résultat de l'application de la règle R en Figure 8 sur le graphe conceptuel en Figure 5.

Apportons quelques précisions concernant la signification donnée à la *fusion* entre deux sommets concepts lorsqu'une règle peut être appliquée. Prenons un couple de sommets concepts $(x_i^{C_R}, \Pi(x_i^{H_R}))$ où une fusion de $x_i^{C_R}$ sur $\Pi(x_i^{H_R})$ doit être faite. Soient $(t_1, *)$ l'étiquette de $x_i^{C_R}$, et (t_2, m) l'étiquette de $\Pi(x_i^{H_R})$ avec $m \in I \cup \{*\}$. Si les types t_1 et t_2 sont liés par une relation d'ordre ($t_1 \leq t_2$ ou $t_2 \leq t_1$), l'étiquette du sommet concept résultant fusionné est formé du type le plus spécialisé et de m. Si par contre t_1 et t_2 ne sont pas comparables, alors aucune [2] fusion ne pourra pas être opérée. Dans ce cas, nous n'appliquons pas la règle. De même, l'application d'une règle sur un graphe conceptuel peut modifier ce graphe de telle sorte que la conformité ne soit plus respectée. Dans ce cas là aussi nous n'appliquons pas la règle.

1.2.1.2 Redondance d'information

Bien qu'une règle puisse être appliquée plusieurs fois suivant une même projection, et qu'une règle puisse ajouter de l'information déjà présente dans le graphe, les seules applications de règles « intéressantes » sont celles qui apportent de l'information supplémentaire au graphe.

Définition 1.7 (Graphes équivalents) *Deux graphes conceptuels G_1 et G_2 sont dits* équivalents *s'il existe une projection de G_1 dans G_2 et une projection de G_2 dans G_1.*

De façon intuitive, deux graphes équivalents représentent la même information, ce qui est vérifié par les formules logiques associées aux graphes ($\Phi(G_1) \cong \Phi(G_2)$). D'après cette définition, les applications de règles ajoutant de l'information à un graphe G seront celles telles que le graphe obtenu après l'application de la règle n'est pas équivalent à G.

Définition 1.8 (Dérivation) *Soit un graphe conceptuel G et un ensemble de règles $\mathcal{R} = \{R_1, \ldots, R_n\}$. G' est une* dérivation immédiate *de $(G, \mathcal{R})$ s'il existe une règle $R_i \in \mathcal{R}$ et une projection Π de l'hypothèse de R_i dans G tel que G' est le résultat de l'application de R_i sur G selon Π. G' est* dérivable *de $\{G, \mathcal{R}\}$ s'il existe une séquence de dérivations immédiates de G à G'.*

[2] Notons qu'avec l'utilisation de *types conjonctifs* (voir la Section 1.2.3) la fusion est possible ; l'étiquette du sommet concept résultant fusionné sera formé du type conjonctif $t_1 \cup t_2$ et de m. Si ce type conjonctif est banni, la règle sera toujours dite *incohérente* ou *bannie*.

Le problème consistant à déterminer, à partir de deux graphes conceptuels G et H et d'un ensemble de règles $\mathcal{R}$, s'il existe un graphe H' dérivable de $\{H, \mathcal{R}\}$ tel qu'il existe une projection de G dans H est semi-décidable [Baget et Mugnier, 2002]. Cependant, dans le cas où des contraintes sont posées sur la forme des règles (imposant par exemple que seul l'ajout de sommets relations est possible comme dans le cas de la règle R de la Figure 8), ce problème devient décidable et il est possible de calculer un graphe contenant toutes les informations pouvant être déduites du graphe initial et de l'ensemble de règles.

Définition 1.9 (Saturation) *Un graphe conceptuel G est dit* saturé *par un ensemble de règles $\mathcal{R}$ si toute l'information qui peut être ajoutée par une règle de $\mathcal{R}$ est déjà présente dans G. (Quelle que soit la règle R de $\mathcal{R}$ et quelle que soit la projection Π de l'hypothèse de R dans G, l'application de R suivant Π produit un graphe équivalent à G).*

Ainsi, le graphe de la Figure 9 est une saturation du graphe de la Figure 5 par l'ensemble de règles $\mathcal{R}$ réduit à la seule règle de la Figure 8 : toute application de la règle ne fait que rajouter de l'information redondante.

1.2.2 Contrainte

Les contraintes de graphes conceptuels [Baget et Mugnier, 2002] constituent une extension au modèle des GCs simples. Une contrainte permet de s'assurer de la justesse d'une modélisation en contraignant la façon d'associer certains sommets d'un graphe conceptuel. En effet, s'il est assez aisé de représenter des connaissances avec un graphe conceptuel, les seules limitations liées au support, comme la signature ou la de conformité, ne garantissent pas toujours que les connaissances représentées dans le graphe soient « valides » selon le point de vue de l'expert qui a défini la structure de domaine (le support). Les contraintes apportent donc une solution pour contraindre la manière de modéliser des connaissances via des graphes conceptuels.

Deux types de contraintes existent : les contraintes positives et les contraintes négatives. Une contrainte positive est de la forme « si l'information A est présente, alors on doit également trouver l'information B ». Une contrainte négative est de la forme « si l'information A est présente, alors on ne doit pas trouver l'information B »[3]. A et B sont exprimées sous forme de graphes conceptuels liés par des liens de coréférence entre certains sommets concepts.

La représentation d'une contrainte positive est repérée par $\boxed{+}$. La condition et l'obligation de la contrainte positive sont séparées par une ligne verticale, la condition se trouvant au dessus de la ligne et l'obligation en dessous. La représentation d'une contrainte négative est repérée par $\boxed{-}$. La condition et l'interdiction de la contrainte négative sont séparées par une ligne verticale, la condition se trouvant au dessus de la ligne et l'interdiction en dessous.

Les contraintes sont définies de manière relativement analogue à celle des règles, et ce de la façon suivante.

[3] Une forme plus générale de contrainte peut être manipulée, avec la sémantique suivante : « si l'information A est présente (et B est absente) alors on doit trouver l'information C mais pas D ».

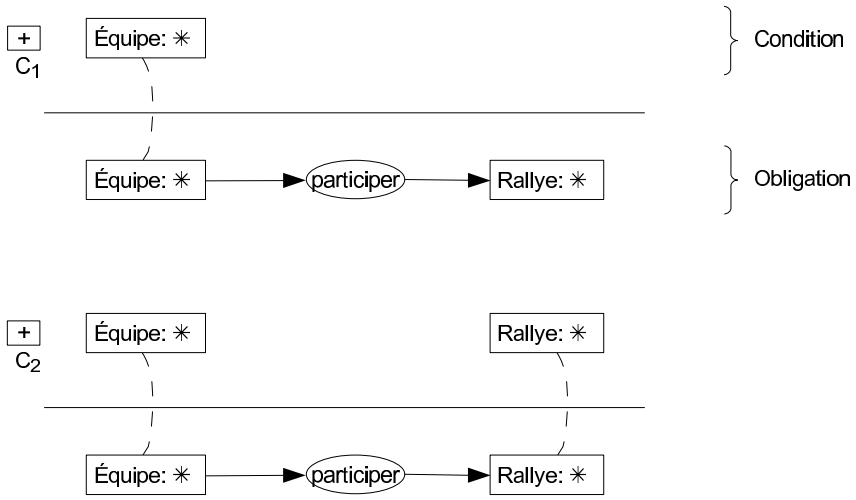


Figure 10 – Deux contraintes positives C_1 et C_2.

Définition 1.10 (Contrainte positive) *Une contrainte positive* $C : C_C \Rightarrow_\exists O_C$ *est un couple de lambda abstractions* ($\lambda x_1 \dots x_n\ C_C$, $\lambda x_1 \dots x_n\ O_C$).
C_C *est appelé la* condition *de* C *et* O_C *l'*obligation *de* C. $x_1, \dots, x_n$ *sont appelés les* points d'attache. *Ils correspondent aux liens de coréférence entre* C_C *et* O_C. *L'occurrence d'un* x_i *dans* C_C *(respectivement dans* O_C*) sera notée* $x_i^{C_C}$ *(respectivement* $x_i^{O_C}$*).*

Définition 1.11 (Contrainte négative) *Une contrainte négative* $C : C_C \nRightarrow_\exists P_C$ *est un couple de lambda abstractions* ($\lambda x_1 \dots x_n\ C_C$, $\lambda x_1 \dots x_n\ P_C$).
C_C *est appelé la* condition *de* C *et* P_C *l'*interdiction *de* C. $x_1, \dots, x_n$ *sont appelés les* points d'attache. *Ils correspondent aux liens de coréférence entre* C_C *et* P_C. *L'occurrence d'un* x_i *dans* C_C *(respectivement dans* P_C*) sera notée* $x_i^{C_C}$ *(respectivement* $x_i^{P_C}$*).*

Exemple

La Figure 10 représente deux contraintes positives. La contrainte positive C_1 a pour condition une équipe, et pour obligation que cette équipe doive participer à (au moins) un rallye. La contrainte positive C_2 représente l'obligation suivante : « toutes les équipes doivent participer à tous les rallyes ».

La Figure 11 représente deux contraintes négatives. La contrainte négative C_3 stipule l'interdiction suivante : « un pilote d'auto (car en pilotant une) ne peut piloter une moto ». La contrainte négative C_4 n'a pas de condition et interdit qu'« un pilote puisse à la fois piloter une auto et une moto ». Ces deux contraintes négatives C_3 et C_4 ont donc la même signification. Ceci nous amène à la remarque suivante. Une contrainte négative peut être ramenée à « Il ne faut pas *interdiction* », en incluant la condition dans l'interdiction [Baget et Mugnier, 2002].

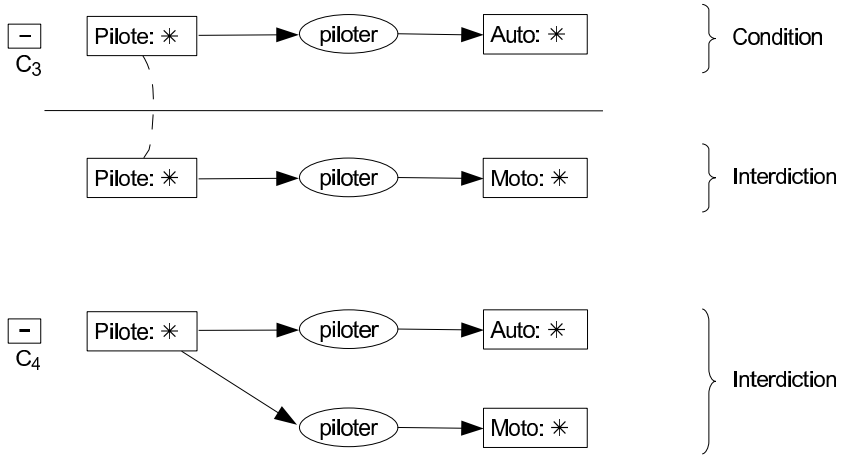


Figure 11 – Deux contraintes négatives C_1 et C_2.

Vérification de contraintes et validité

Un graphe conceptuel vérifie une contrainte positive si toute projection de la condition de la contrainte sur le graphe peut être étendue à une projection de la contrainte en entier (condition et obligation) sur le graphe. Ce graphe vérifie un contrainte négative si aucune projection de la condition de la contrainte sur le graphe ne peut être étendue à une projection de la contrainte en entier (condition et interdiction) sur le graphe.

On adopte la notation suivante. Soient C une contrainte positive et C' une contrainte négative ; on notera $\widehat{C}$ (respectivement $\widehat{C'}$) le graphe conceptuel formé de C_C et O_C (respectivement de $C_{C'}$ et $P_{C'}$) dans lequel tous les couples de sommets $(x_i^{C_C}, x_i^{O_C})$ (respectivement $(x_i^{C_{C'}}, x_i^{P_{C'}})$) ont été fusionnés.

Définition 1.12 (Vérification d'une contrainte positive) *Un graphe conceptuel G* vérifie *une contrainte positive C dans l'un des trois cas suivant :*

- *C_C est vide et il existe une projection de O_C (ou de $\widehat{C}$) dans G.*
- *C_C n'est pas vide et il n'existe pas de projection de C_C dans G.*
- *Il existe une projection de C_C dans G et cette projection peut être étendue à une projection de $\widehat{C}$ dans G.*

Définition 1.13 (Vérification d'une contrainte négative) *Un graphe conceptuel G* vérifie *une contrainte négative C dans l'un des trois cas suivant :*

- *C_C est vide et iln'existe pas de une projection de P_C (ou de $\widehat{C}$) dans G.*
- *C_C n'est pas vide et il n'existe pas de projection de C_C dans G.*
- *Il existe une projection de C_C dans G et cette projection ne peut pas être étendue à une projection de $\widehat{C}$ dans G.*

Définition 1.14 (Validité d'un graphe conceptuel) *Un graphe conceptuel G est dit* valide *sur un ensemble de contraintes positives et/ou négatives $\mathcal{C}$ si G vérifie toutes les contraintes de $\mathcal{C}$.*

Ainsi, le graphe de la Figure 5 satisfait les contraintes positives en Figure 10 et les contraintes négatives en Figure 11, et est donc valide sur cet ensemble de quatre contraintes. Notons que la vérification de contraintes est un problème co-NP-complet pour les contraintes négatives et co-NP^{NP}-complet pour les contraintes positives [Baget et Mugnier, 2002].

1.2.3 Type conjonctif

Les types conjonctifs [Chein et Mugnier, 2004] constituent une extension du support du modèle des GCs simples. Dans cette sous-section, certaines notations sur les ensembles ordonnés sont empruntées à [Davey et Priestley, 2002].

La définition des types de concepts vue précédemment (Définition 1.1) reste inchangée, mais ils sont désormais nommés *types de concepts primitifs*. Ceci en contraste avec les *types de concepts conjonctifs*.

Un type de concept conjonctif est un ensemble de types de concepts primitifs. Il peut être *acceptable* ou *interdit*. Un type est interdit s'il ne peut y avoir un individu de ce type, c'est-à-dire qu'un sommet concept ne peut être du type interdit. Par contre, il peut y avoir des individus d'un type acceptable. Si pour un type conjonctif acceptable $t = \{t_1, \ldots, t_n\}$, il existe deux types primitifs comparables $t_i \leq t_j$ ($t_i, t_j \in t$ avec $i \neq j$), alors le type conjonctif $t' = \{t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_n\}$ est aussi acceptable. En d'autres termes, si t_i et t_j sont deux types primitifs appartenant à un type conjonctif acceptable tel que $t_i \leq t_j$, alors t_j est « inutile » dans la définition de ce type conjonctif (seul le type primitif le plus spécifique est pris en compte). De la manière analogue, si un type conjonctif $t = \{t_1, \ldots, t_n\}$ est interdit, et si $t_i \leq t_j$ ($t_i, t_j \in t$ avec $i \neq j$), alors le type conjonctif $t' = \{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\}$ est aussi interdit. Ainsi, si t_i et t_j sont deux types primitifs appartenant à un type conjonctif interdit avec $t_i \leq t_j$, alors t_i est « inutile » dans la définition de ce type conjonctif (seul le type primitif le plus général est pris en compte).

Dans la théorie des ensembles ordonnés, un sous-ensemble d'éléments non comparables d'un ensemble ordonné s'appelle une *anti-chaîne*. L'ensemble des types de concepts conjonctifs est donc l'ensemble d'anti-chaînes de l'ensemble T_C des types de concepts primitifs.

Définition 1.15 (Type de concept conjonctif) *Un* type de concept conjonctif *est un ensemble (non vide) de types de concepts primitifs non comparables ; c'est-à-dire une anti-chaîne de T_C (T_C étant l'ensemble des types de concepts primitifs).*

Les types conjonctifs sont dotés d'un d'ordre partiel qui prolonge naturellement celui des types primitifs : un type conjonctif t est une spécialisation d'un type conjonctif s si chaque type primitif de s a une spécialisation (éventuellement égaler à lui-même) dans t.

Définition 1.16 (Ensemble des types de concepts conjonctifs) *Soit T_C l'ensemble des types de concepts primitifs. $T^{\sqcap}$ dénote l'ensemble des types de concepts conjonctifs qui peuvent être créés à partir de T_C. $T^{\sqcap}$ est doté de l'ordre partiel suivant, qui étant l'ordre partiel sur T_C : soient deux types conjonctifs $t = \{t_1, \ldots, t_n\}$ et $s = \{s_1, \ldots, s_p\}$,*

$t \leq s$ si pour chaque type primitif $s_j \in s$, $1 \leq j \leq p$, il existe un type primitif $t_i \in t$, $1 \leq i \leq n$, tel que $t_i \leq s_j$.

L'ensemble de types conjonctifs acceptables peut être exponentiellement plus grand que l'ensemble des types primitifs. C'est pourquoi la hiérarchie des types conjonctifs acceptables n'est pas définie par extension mais au moyen d'ensemble de types primitifs parmi lesquels certains sont interdits. L'ensemble de types conjonctifs acceptables est obtenu à partir de $T^{\sqcap}$ auquel on a retiré les types conjonctifs interdits, appelés aussi bannis.

Définition 1.17 (Ensemble des types de concepts conjonctifs bannis) *Soit B un sous-ensemble de types de concepts conjonctifs interdits non comparables. Un élément de $T^{\sqcap}$ est dit* banni, *selon B, s'il est inférieur ou égal à un élément de B. $\downarrow B$ dénote l'ensemble des types de concepts conjonctifs bannis ; c'est-à-dire $\downarrow B = \{t \in T^{\sqcap} \mid \exists t' \in B, t \leq t'\}$.*

Définition 1.18 (Hiérarchie des types de concepts) *Une hiérarchie des types de concepts T est donnée par le couple (T_C, B) avec :*

- *T_C l'ensemble des types de concepts primitifs,*
- *B l'ensemble des types de concepts conjonctifs bannis,*
- *$\forall b \in B, \nexists t \in T_C \mid t \leq b$; c'est-à-dire $\downarrow B \cap T_C = \emptyset$.*

T est définie comme l'ensemble $T^{\sqcap} \backslash \downarrow B$. $T^{\sqcap}$ est donc partitionné avec d'un coté les types de concepts conjonctifs [4] acceptables T et de l'autre les types de concepts conjonctifs bannis $\downarrow B$.

L'étiquetage donné à la Définition 1.2 sur les sommets concepts d'un graphe conceptuel est modifié. L'étiquette d'un sommet concept est le couple (type(c),marqueur(c)), mais cette fois-ci type(c) appartient à l'ensemble des types de concepts conjonctifs T, et non seulement l'ensemble des types de concepts primitifs T_C. type(c) est représenté sous la forme d'une liste de types primitifs séparés par une virgule.

Exemple

Prenons l'ensemble des types de concepts primitifs T_C de la figure 3. Nous pouvons dire qu'il est pertinent d'ajouter dans le support un type de concept héritant à la fois du concept Auto et du concept Moto, car il s'agit du type de concept Quadricycle, pour l'utiliser dans un graphe conceptuel par la suite. Mais toute autre combinaison, comme un type de concept héritant à la fois du concept Directeur et Pilote, a certes un sens mais ne possède pas de désignation propre. De plus, l'ajout de toutes les combinaisons possibles de types de concepts primitifs sur un support de taille raisonnable le rendraient vite illisible et pas nécessairement plus pertinent.

L'utilisation des types de concepts conjonctifs permet d'utiliser dans un graphe, comme en partie droite de la Figure 12, une conjonction de types de concepts primitifs sans qu'elle soit préalablement définie au niveau du support. Seule éventuellement une liste de types de concepts conjonctifs bannis peut être ajoutée au support, comme en partie gauche de la Figure 12, pour éviter de modéliser des faits qui n'ont aucun sens. Par exemple, créer un individu qui est à la fois de type Personne et Véhicule est absurde.

[4] *Un type primitif peut être vu comme type conjonctif constitué d'un seul type.*

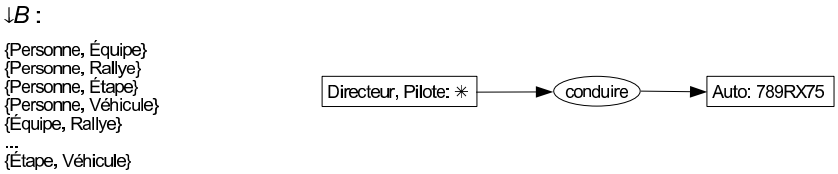


Figure 12 – Types de concepts conjonctifs bannis (à gauche). Un graphe conceptuel utilisant le type conjonctif formé des types de concepts primitifs {Directeur,Pilote} (à droite).

1.2.4 Notre adaptation et extension du modèle des GCs

1.2.4.1 Motivation et objectif

Partons du double constat suivant en représentation des connaissances.

D'une part, la notion d'individu est généralement associée aux connaissances factuelles d'une modélisation. Or avec le modèle des GCs actuel pour modéliser un domaine, les marqueurs individuels - représentant des individus - sont nécessairement définis dans un support, constituant le niveau structurel et non factuel d'une modélisation. Notre premier objectif ici est d'adapter le modèle des GCs afin d'assouplir cette « localisation » des marqueurs individuels.

D'autre part, les notions de données et de types de données (ou *datatypes* en anglais) sont très souvent utilisées en modélisation. Une donnée est caractérisée par un couple (type de donnée, valeur) où la valeur est aussi appelée un littéral. Notre second objectif est d'étendre le modèle des GCs afin de pouvoir modéliser des données et leurs types.

■ **Adaptation sur l'usage des marqueurs individuels**

Nous estimons avec le modèle des GCs actuel comme un obstacle de modélisation l'obligation de renseigner dans le support tout marqueur individuel ainsi que sa conformité, c'est-à-dire d'inclure nécessairement aux connaissances structurelles d'une modélisation les individus et leurs types associés. Notons bien qu'il ne s'agit pas d'abandonner cette pratique. En effet, dans le cas d'un type concept pour lequel l'ensemble des individus ayant pour type ce concept est connu et fixé dès la création du concept, il semble raisonnable qu'ils soient définis dans le support. Prenons par exemple le type de concept Continent dont les individus sont Eurasie, Afrique, Amérique, Australie et Antarctique et ceci de façon immuable.

Faisons l'hypothèse suivante : la modélisation des connaissances d'un domaine, c'est-à-dire la modélisation des connaissances structurelles et factuelles du domaine, est élaborée par un utilisateur considéré comme un *expert* du domaine. Sous cette hypothèse - qui s'impose d'elle même - il est alors envisageable d'*assouplir* la création d'un graphe conceptuel qui compose les faits d'une modélisation, sans pour autant modifier la sémantique logique de la modélisation composée du graphe conceptuel et de son support.

Pour cela, nous considérons que l'association d'un type à un marqueur individuel n'est plus nécessairement dictée par la conformité mais peut être fournie par l'étiquette d'un sommet concept individuel au sein du graphe conceptuel. En effet, étant donné

que les connaissances factuelles contenues dans le graphe conceptuel sont considérées comme exactes car émanant de l'expert, la sémantique logique $t(m)$ du sommet concept individuel d'étiquette (t, m) appartenant au graphe ferait « double emploi » avec la conformité qui associerait à un individu de marqueur individuel m le type t ayant pour sémantique logique $t(m)$ (voir les Tableaux 1 et 2).

Il en résulte qu'une adaptation du modèle des GCs, sans en changer la sémantique logique, est possible sous cette hypothèse de crétion de graphe par un expert. Cette adaptation a pour objectif de permettre à des marqueurs individuels de ne plus être définis dans le support, ni être associé par conformité à un type, mais de pouvoir appartenir à un graphe conceptuel.

■ **Extension pour la prise en compte de *donnée* et *type de donnée***

Le modèle de base des GCs ne gère pas ce qu'il est commun d'appeler en modélisation des *données*, comme les nombres ou les chaînes de caractères. Pour y remédier, il existe notamment l'extension du modèle des GCs de [Baget, 2007]. Cependant, cette extension s'éloigne du modèle des GCs « classique ». En ce sens, l'extension que nous proposons ici diffère en deux points principaux de celle proposée dans [Baget, 2007]. D'une part, un graphe conceptuel dans cette dernière approche comprend quatre familles de sommets (sommet concept, sommet relation, opérateur et donnée), tandis que nous conservons l'aspect biparti d'un graphe conceptuel. D'autre part, [Baget, 2007] autorise la constitution de données dont la valeur est un ensemble de littéraux, telle que l'existence d'une personne dont l'âge est supérieur à 16 ans. Nous avons choisi ne pas l'autoriser dans un graphe conceptuel représentant des faits d'une modélisation. Sinon, quelle serait la réponse à la question « Quelles sont les personnes de moins de 18 ans ? » sur ce graphe modélisant cette personne de 16 ans ou plus ? Cependant, nous acceptons dans la constitution d'une interrogation ou de la condition d'une contrainte l'emploi de données valuées par un ensemble de littéraux (intervalle de nombres ou ensemble de chaînes de caractères contenant une sous-chaîne donnée), comme la question précédente ou une contrainte (négative ici) interdisant pour une personne mineure de conduire un véhicule.

■ **Cadre de notre adaptation et extension**

Dans la mesure où nous supposons deux catégories d'utilisateurs, nous considérons deux familles de graphes conceptuels. D'un coté, les utilisateurs supposés experts du domaine modélisé élaborent le support et les graphes conceptuels représentant ce domaine. Ces graphes font partie de la famille des *graphes conceptuels factuels*. Ils constituent les connaissances factuelles du domaine modélisé. D'un autre coté, les utilisateus souhaitant interroger les connaissances factuelles modélisées, le font aux travers de graphes conceptuels appartenant à la famille des *graphes conceptuels requêtes*.

Dans les définitions de cette section concernant le support et les graphes conceptuels factuels et requêtes, nous faisons ressortir en gras ce qui change par rapport aux définitions du support et d'un graphe conceptuel présentées en Sections 1.1 et 1.2.3.

Nous considérons le modèle formel de [Mugnier et Chein, 1996] avec types conjonctifs [Chein et Mugnier, 2004] comme base de notre adaptation et extension du modèle des

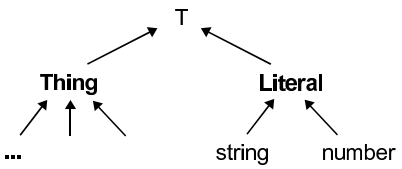


Figure 13 – Squelette de tout support.

GCs. Partant de cette base, nous complétons la définition du support en Définition 1.1 comme suit.

Définition 1.19 (Support étendu) *Un support est un sixtuplet $S = (T_C, B, T_R, \sigma, I_S, \tau)$ [Mugnier et Chein, 1996], où :*

- *T_C est l'ensemble partiellement ordonné de* types de concepts*; cet ordre est donné en Définitions 1.16 et 1.18. $\top$ est le plus grand élément,* ***Thing et Literal en sont les deux sous-types directs, et string et number sont les deux sous-types de Literal. Ces cinq types de concepts primitifs constituent le* squelette *des types de concepts (Figure 13)****.***
- *B est l'ensemble des types de concepts bannis, contenant notamment {Thing,Literal} et {string, number}.*
- *T_R l'ensemble de* types de relations *comme définis en Définition 1.1.*
- *σ est une application associant à chaque type de relation t_r une* signature*, comme définie en Définition 1.1 mais où cette fois-ci l'ensemble des types de concepts considéré est l'ensemble des types de concepts conjonctifs $T^{\sqcap}$ (et non seulement T_C l'ensemble des types de concepts primitifs).*
- *I est **un** ensemble des* marqueurs individuels*. À cet ensemble I s'ajoute un* marqueur générique*, noté $*$, qui permet de représenter un individu non spécifié.*
- *τ est une application, dite de* conformité*, de I_S dans T_C, qui à tout marqueur individuel m associe un type de concept t.*
- *$\#$ **est un littéral générique (dit aussi valeur générique), qui permet de représenter une donnée non-valuée.***

La contrainte stipulant que seuls Thing et Literal sont des sous-types directs de $\top$ nous permet de séparer dans le support de façon explicite les types de concepts « à valeur » de *types de données*, car sous-types de Literal, des autres types de concepts. Ainsi, pour une modélisation donnée, l'utilisateur sera amené à créer ses propres types de concepts, sous-types de Thing ou sous-types de string ou number. Ces deux derniers types de concepts ont pour vocation de représenter respectivement des chaînes de caractères et des nombres réels.

1.2.4.2 Graphe conceptuel factuel

Dans cette extension du modèle des GCs pour la prise en compte de données, il est nécessaire de définir l'ensemble des littéraux considéré.

Définition 1.20 (Ensemble des littéraux) *Soit Σ l'ensemble de caractères imprimables (espace blanc compris) du standard Unicode*[5]*. Une chaîne de caractères est une suite finie, éventuellement vide, de caractères ; la chaîne formée du caractère 'a' puis 'b', puis 'c' se note "abc". L'ensemble des chaînes de caractères que l'on peut former avec Σ est noté Σ^*. L'ensemble $L = \mathbb{R} \cup \Sigma^*$ est appelé l'*ensemble des littéraux.

Un graphe conceptuel factuel représente une partie des faits d'une modélisation des connaissances d'un domaine donné. Il est défini sur un support qui représente les connaissances structurelles de la modélisation.

Définition 1.21 (Graphe conceptuel factuel) *Un* graphe conceptuel factuel $F = (C, R, U, \boldsymbol{I}, etiq)$ *défini sur un support $S = (T_C, B, T_R, \sigma, I_S, \tau)$ (selon la Définition 1.1 étendue au types conjonctifs [Chein et Mugnier, 2004]) est un multigraphe non orienté, biparti, non nécessairement connexe, où :*

- *C est l'ensemble des* sommets concepts*, R est l'ensemble des* sommets relations *et U est l'ensemble des* arêtes *sont tels que définis à la Définition 1.2.*
- *I* ***est un ensemble de* marqueurs individuels*, tel que*** *:*
  - *$I \cup I_S \cup \{*\}$ est muni de l'odre suivant : $*$ est plus grand que tous les marqueurs individuels, et ces derniers sont deux à deux incomparables.*
  - *$I \cap I_S = \emptyset$*
- *etiq est une application qui à tout sommet et à toute arête associe une étiquette, tel que :*
  - *$\forall r \in R$, $etiq(r) \in T_R$*
  - *$\forall c \in C$, $etiq(c) \in T^{\sqcap} \times (\boldsymbol{I} \cup I_S \cup \{*\} \cup \boldsymbol{L} \cup \{\#\})$, avec L* ***l'ensemble des littéraux****. L'étiquette d'un sommet concept c est le couple (type(c), marqueur(c)). Un sommet concept tel que marqueur(c) $\in \boldsymbol{I} \cup I_S$ est appelé* sommet concept individuel*, tel que marqueur(c) $= *$ est appelé* sommet concept générique*, tel que marqueur(c) $\in L$ est appelé* **sommet concept donnée***, et tel que marqueur(c) $= \#$ est appelé* **sommet concept donnée non-valuée***. $I \cap L = \emptyset$,* ***$I_S \cap L = \emptyset$, et $*$ et $\#$ ne sont pas comparables****.*
  - *$\forall e \in U, etiq(e) \in \mathbb{N}$*
- *De plus, etiq obéit*
  - *aux contraintes fixées par σ et τ (comme en Définition 1.2) :*
    - *L'ensemble des arêtes adjacentes à tout sommet relation r est totalement ordonné, ce que l'on représente en étiquetant les arêtes par une numérotation de 1 à l'arité de r. On note $F_i(r)$ le $i^{ème}$ voisin de r dans F.*
    - *$\forall r \in R$, $type(F_i(r)) \leq \sigma_i(type(r))$.*
    - *$\forall c \in C$ si $marqueur(c) \in I_S$ alors $\tau(marqueur(c)) \leq type(c)$.*
  - *aux contraintes liées aux types conjonctifs (Section 1.2.3) :*
    $\forall c_1, c_2 \in C$ tels que $marqueur(c_1), marqueur(c_2) \in \boldsymbol{I} \cup I_S$, notons $t_1 = type(c_1)$ et $t_2 = type(c_2)$, si $marqueur(c_1) = marqueur(c_2)$ et $t_1 \neq t_2$ alors le type conjonctif $t_1 \cup t_2 \notin \downarrow B$; $\downarrow B$ étant l'ensemble des types conjonctifs bannis.
  - ***aux contraintes liées à la* filiation *type-marqueur*** *:*
    - *$\forall c \in C$ si $marqueur(c) \in I \cup I_S \cup \{*\}$ alors $type(c) \leq$ Thing.*

[5] *Unicode spécifie un ensemble de caractères indépendant de tout codage,* `http://www.unicode.org/`*. Le choix de Unicode n'est là que pour désigner explicitement un ensemble de caractères, et ainsi définir formellement L.*

– $\forall c \in C$ *si marqueur*$(c) \in L \cup \{\#\}$ *alors si marqueur*$(c) \in \mathbb{R}$ *alors type*$(c) \leq$ *number, sinon si marqueur*$(c) \in \Sigma^*$ *alors type*$(c) \leq$ *string, sinon (c'est-à-dire marqueur*$(c) = \#$*) type*$(c) \leq$ *Literal.*

Cette définition d'un graphe conceptuel factuel diffère donc de la Définition 1.2 en trois points principaux. Premièrement, le graphe conceptuel factuel dispose de son propre ensemble de marqueurs individuels. En effet, ici l'ensemble des marqueurs individuels du support ne constitue plus l'ensemble des marqueurs individuels d'une modélisation d'un domaine donné mais seulement un sous-ensemble. Deuxièmement, le choix du type [6] associé à un marqueur individuel via un sommet concept n'est plus soumis à l'application de conformité (τ), sauf si le marqueur individuel est dans l'ensemble des marqueurs individuels du support. Troisièmement, l'étiquetage d'un sommet concept peut être composé d'un marqueur (individuel ou générique) ou d'un littéral ; les ensembles des marqueurs et l'ensemble des littéraux étant disjoints ($(I_S \cup I) \cap L = \emptyset$).

La normalisation d'un graphe conceptuel factuel reste quasiment la même que pour un graphe conceptuel utilisant les types conjonctifs, avec en plus la prise en considération des littéraux : il sera dit sous forme normale si chaque marqueur individuel et chaque littéral non générique apparaît au plus une fois dans le graphe. Un graphe conceptuel factuel peut donc être mis sous forme normale en fusionnant tous les sommets concepts ayant le même marqueur individuel ou ayant le même littéral non générique. Dans le cas où deux sommets concepts de types différents t_1 et t_2 possèdent un même marqueur individuel ou un même littéral non générique, le sommet concept fusionné possèdera pour type le type conjonctif $t_1 \cup t_2$ (ou le plus spécialisé s'il existe un ordre entre t_1 et t_2).

La sémantique logique d'un graphe conceptuel factuel reste inchangée par rapport à celle associée à un graphe conceptuel avec types conjonctifs du Chapitre 1.

Définition 1.22 (Base de faits) *Soit $\mathcal{F} = \{F_1, \ldots, F_n\}$ l'ensemble des* graphes conceptuels factuels *$F_1, \ldots, F_n$ tous définis sur un même support. On appelle une* base de faits *formée sur l'ensemble $\mathcal{F}$, le graphe conceptuel factuel F formé de l'union disjointe des graphes F_1 à F_n.*
Soit $F_i = (C_i, R_i, U_i, I_i, etiq_i)$ un élément de $\mathcal{F}$, on note $I_{\mathcal{F}}^{\sqcup} = \bigcup_{i=1}^{n} I_i$. La base de faits F est dite légitime *si la forme normale de F ne possède pas de sommet concept dont le type est un type conjonctif banni.*

Définition 1.23 (Base de connaissances) *Soient S un support et F une base de faits définie sur S. Le couple (S,F) constitue une* base de connaissances *dans le modèle des GCs (adapté). La base de connaissances (S,F) est dite* bien fondée *si F est légitime et sous forme normale.*

■ **Exemple**

Le support en Figure 14 présente un exemple de support tel que défini en dans notre adaptation du modèle des GCs avec types conjonctifs. L'ensemble des types conjonctifs acceptables étant obtenu par construction sur l'ensemble des types primitifs auquels on retire les types conjonctifs bannis (ceux en Définition 1.19, et ici {Personne,Cours}), il

[6] type primitif ou conjonctif

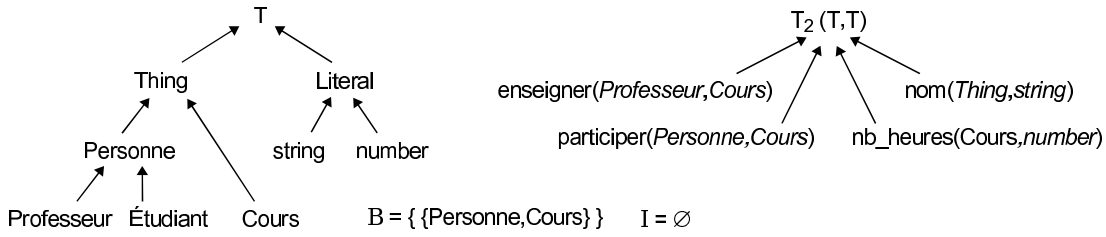


Figure 14 – Support avec types conjontifs, dépourvu de marqueur individuel.

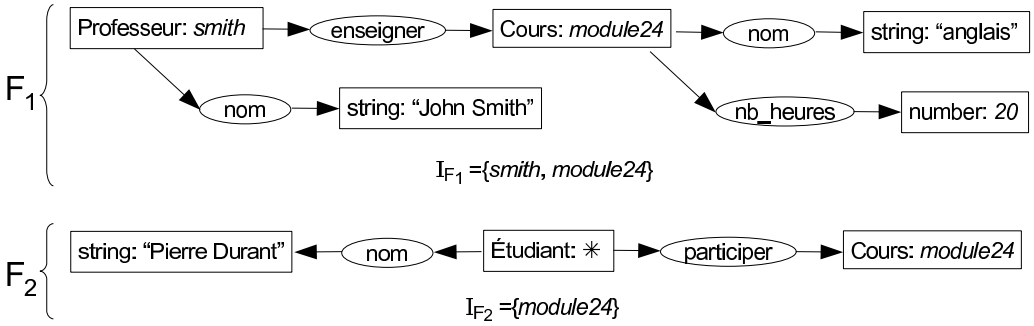


Figure 15 – Une base de faits formée des deux *graphes conceptuels factuels* F_1 et F_2 définis sur le support en Figure 14.

en résulte que l'ensemble des types conjonctifs acceptables est réduit au singleton {{Professeur,Étudiant}}. Sur cet exemple, en guise de contraste par rapport à la Définition 1.1, l'ensemble des marqueurs individuels définis au sein de ce support est vide.

La Figure 15 présente deux graphes conceptuels factuels F_1 et F_2 définis sur le support en Figure 14. Le graphe F_1 modélise le professeur identifié par smith, de nom (la chaîne de caractères) "John Smith", qui enseigne le cours module24 de 20 heures intitulé (de nom) "anglais". Le graphe F_2 modélise un étudiant de nom "Pierre Durant" qui participe au cours module24. On constate que le marqueur individuel smith appartient au graphe conceptuel factuel F_1, qui associe à ce marqueur individuel le type Professeur. De même, le marqueur individuel module24 a pour type Cours. Le marqueur individuel module24 est aussi défini dans F_2 et dans lequel module24 a pour type Cours. La base de faits formée sur F_1 et F_2 est légitime car ni F_1 ni F_2 ne possédent de sommet concept ayant un type banni. Cette base de faits est aussi cohérente car le marqueur individuel module24 commun à F_1 et F_2 est typé par Cours dans ces deux graphes, on a donc bien Cours $\leq$ Cours.

1.2.4.3 Graphe conceptuel requête

Nous définissons ici une seconde famille de graphes - après celle des graphes conceptuels factuels -, celle des *graphes conceptuels requêtes*. Nous apportons aussi une adaptation de l'opération de projection au niveau de sommets concepts issus d'un graphe requête sur ceux issus d'un graphe factuel.

Définition 1.24 (Graphe conceptuel requête) *Un* graphe conceptuel requête $Q = (C, R, U, etiq)$ *défini sur une base de connaissances (S,F), avec* $S = (T_C, B, T_R, \sigma, I_S, \tau)$ *et* F *formé sur l'ensemble de graphes conceptuels factuels* $\mathcal{F} = \{F_1, \ldots, F_n\}$ *définis sur* S*, est un multigraphe non orienté, biparti, non nécessairement connexe, où :*

- C *l'ensemble des* sommets concepts, R *l'ensemble des* sommets relations *et* U *est l'ensemble des* arêtes *sont tels que définis à la Définition 1.2.*
- *etiq est une application qui à tout sommet et à toute arête associe une étiquette, tel que :*
  - $\forall r \in R$, *etiq*$(r) \in T_R$
  - $\forall c \in C$, *etiq*$(c) \in T^{\sqcap} \times$ ***Op*** $\times (I_{\mathcal{F}}^{\sqcup} \cup I_S \cup \{*\} \cup \boldsymbol{L} \cup \{\#\})$. ***L'étiquette d'un sommet concept*** c ***est le triplet (type(***c***), operateur(***c***), marqueur(***c***))****, avec* L *l'ensemble des littéraux. Un sommet concept tel que marqueur(c)* $\in \boldsymbol{I}_{\mathcal{F}}^{\sqcup} \cup I_S$ *est appelé* sommet concept individuel*, tel que marqueur(c)* $= *$ *est appelé* sommet concept générique*, tel que marqueur(c)* $\in L$ *est appelé* **sommet concept donnée***, et tel que marqueur(c)* $= \#$ *est appelé* **sommet concept donnée non-valuée**. ***Op est l'ensemble d'opérateurs définis au Tableau 3****.*
  - $\forall e \in U, etiq(e) \in \mathbb{N}$.
- *De plus, etiq obéit aux contraintes fixées par* σ *et* τ *(comme en Définition 1.2) :*
  - *L'ensemble des arêtes adjacentes à tout sommet relation* r *est totalement ordonné, ce que l'on représente en étiquetant les arêtes par une numérotation de 1 à l'arité de* r*. On note* $F_i(r)$ *le* $i^{ème}$ *voisin de* r *dans* F*.*
  - $\forall r \in R$, *type*$(F_i(r)) \leq \sigma_i($*type*$(r))$
  - $\forall c \in C$ *si marqueur*$(c) \in I_S$ *alors* $\tau($*marqueur*$(c)) \leq$ *type*(c)
- ***aux contraintes liées à la* filiation *type-marqueur*** *:*
  - $\forall c \in C$ *si marqueur*$(c) \in I_{\mathcal{F}}^{\sqcup} \cup I_S \cup \{*\}$ *alors type*$(c) \leq$ *Thing.*
  - $\forall c \in C$ *si marqueur*$(c) \in L \cup \{\#\}$ *alors si marqueur*$(c) \in \mathbb{R}$ *alors type*$(c) \leq$ *number, sinon si marqueur*$(c) \in \Sigma^*$ *alors type*$(c) \leq$ *string, sinon (c'est-à-dire marqueur*$(c) = \#$*) type*$(c) \leq$ *Literal.*

Il est important de constater dans le Tableau 3 que l'opérateur $\preceq$ et $<=$ n'ont rien en commun puisque l'ensemble des marqueurs et l'ensemble des littéraux sont disjoints. En effet, $\preceq$ permet de tester l'ordre entre deux marqueurs (individuels ou génériques) tandis que $<=$ teste l'ordre sur les nombres réels.

Définition 1.25 (Compatibilité sur les étiquettes) *Soient* $Q = (C_Q, R_Q, U_Q, etiq_Q)$ *un graphe conceptuel requête et* $F = (C_F, R_F, U_F, I_F, etiq_F)$ *un graphe conceptuel factuel tous deux définis sur un même support* $S = (T_C, B, T_R, \sigma, I_S, \tau)$*. Soient* $e = (t, \nabla, m)$ *et* $e' = (t', m')$ *les étiquettes de deux sommets concepts où* $c \in C_Q$ *et* $c' \in C_F$ *avec* $etiq_Q(c) = e$ *et* $etiq_F(c') = e'$*. L'étiquette* e *et dite* compatible *avec* e' *si et seulement si* $t' \leq t$ *et* $m' \nabla m$ *est vrai.*

Définition 1.26 (Projection (adaptée)) *Soient* $F = (C_F, R_F, U_F, I_F, etiq_F)$ *un graphe conceptuel factuel et* $Q = (C_Q, R_Q, U_Q, etiq_Q)$ *un graphe conceptuel requête tous deux définis sur un même support* $S = (T_C, B, T_R, \sigma, I_S, \tau)$*. Une projection de Q dans* F[7] *est un couple d'applications* $\Pi = (f, g)$*, avec* $f : C_Q \to C_F$ *et* $g : R_Q \to R_F$ *, tel que :*

[7] *Il peut s'agir d'une base de faits mise sous forme normale.*

opérateur	opérande gauche	opérande droit	commentaires
$\preceq$	$I_{\mathcal{F}}^{\sqcup} \cup I_S \cup \{*\}$	$I_{\mathcal{F}}^{\sqcup} \cup I_S \cup \{*\}$	vrai si les marqueurs individuels sont identiques ou si l'opérande gauche est le marqueur générique $*$, faux sinon.
$=$	$\mathbb{R} \cup \{\#\}$	$\mathbb{R} \cup \{\#\}$	vrai s'il y a égalité numérique entre les deux opérandes ou si l'opérande gauche est $\#$, faux sinon.
$!=$	$\mathbb{R} \cup \{\#\}$	$\mathbb{R} \cup \{\#\}$	vrai s'il n'y a pas égalité numérique entre les deux opérandes, faux sinon ou si un des opérandes est $\#$.
$<$	$\mathbb{R} \cup \{\#\}$	$\mathbb{R} \cup \{\#\}$	vrai si l'opérande gauche est strictement inférieur numériquement à l'opérande droit, faux sinon ou si un des opérandes est $\#$.
$<=$	$\mathbb{R} \cup \{\#\}$	$\mathbb{R} \cup \{\#\}$	vrai si l'opérande gauche est inférieur ou égal numériquement à l'opérande droit, faux sinon ou si un des opérandes est $\#$.
$>$	$\mathbb{R} \cup \{\#\}$	$\mathbb{R} \cup \{\#\}$	vrai si l'opérande gauche est strictement supérieur numériquement à l'opérande droit, faux sinon ou si un des opérandes est $\#$.
$>=$	$\mathbb{R} \cup \{\#\}$	$\mathbb{R} \cup \{\#\}$	vrai si l'opérande gauche est supérieur ou égal numériquement à l'opérande droit, faux sinon ou si un des opérandes est $\#$.
eq	$\Sigma^* \cup \{\#\}$	$\Sigma^* \cup \{\#\}$	vrai s'il y a égalité stricte entre les deux chaînes de caractères ou si l'opérande gauche est $\#$, faux sinon.
neq	$\Sigma^* \cup \{\#\}$	$\Sigma^* \cup \{\#\}$	vrai s'il n'y a pas égalité entre les deux chaînes de caractères, faux sinon ou si un des opérandes est $\#$.
ll	$\Sigma^* \cup \{\#\}$	$\Sigma^* \cup \{\#\}$	vrai si l'opérande gauche est strictement inférieur alphanumériquement parlant à l'opérande droit, faux sinon ou si un des opérandes est $\#$.
leq	$\Sigma^* \cup \{\#\}$	$\Sigma^* \cup \{\#\}$	vrai si l'opérande gauche est inférieur ou égal alphanumériquement parlant à l'opérande droit ou si l'opérande gauche est $\#$.
gg	$\Sigma^* \cup \{\#\}$	$\Sigma^* \cup \{\#\}$	vrai si l'opérande gauche est strictement supérieur alphanumériquement parlant à l'opérande droit, faux sinon ou si un des opérandes est $\#$.
geq	$\Sigma^* \cup \{\#\}$	$\Sigma^* \cup \{\#\}$	vrai si l'opérande gauche est supérieur ou égal alphanumériquement parlant à l'opérande droit, faux sinon ou si un des opérandes est $\#$.
$like$	$\Sigma^* \cup \{\#\}$	$\Sigma^* \cup \{\#\}$	vrai si l'opérande gauche est une sous-chaîne de l'opérande droit, faux sinon ou si un des opérandes est $\#$.
cil	$\Sigma^* \cup \{\#\}$	$\Sigma^* \cup \{\#\}$	vrai si l'opérande gauche est une sous-chaîne de l'opérande droit avec un test non sensible à la « casse » (*Case Insensitive Like*), faux sinon ou si un des opérandes est $\#$.

Tableau 3 – Définition de l'ensemble des *opérateurs* composant les étiquettes de sommets concepts dans un graphe conceptuel requête.

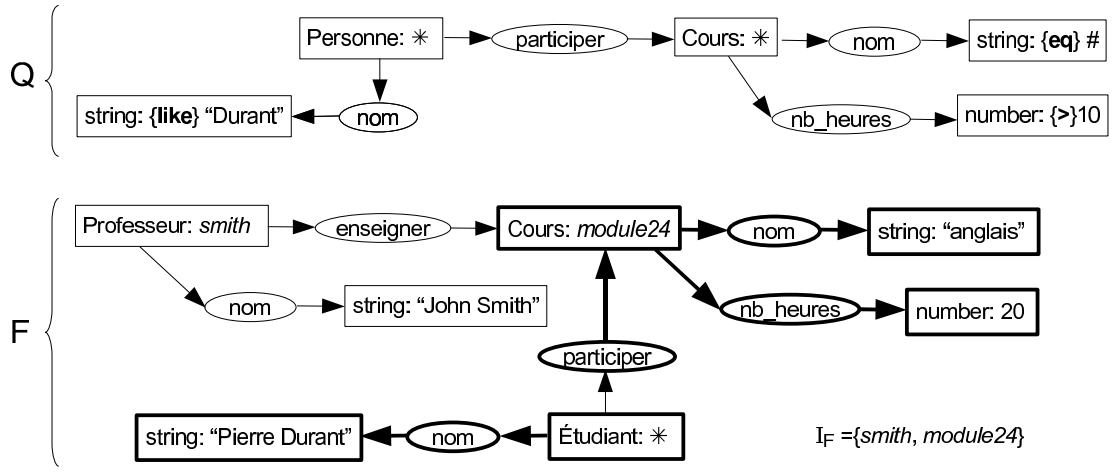


Figure 16 – *Graphe conceptuel requête Q*, et projection (matérialisée en gras) de Q dans F, la base de faits en Figure 15 mise sous forme normale.

- $\forall c \in C_Q$, $etiq_Q(c)$ ***est compatible avec*** $etiq_F(f(c))$,
- $\forall r \in R_Q$, $etiq_F(g(r)) \leq etiq_Q(r)$.
- Π *préserve les arêtes et leurs étiquettes : pour toute arête rc de U_Q, $\Pi(rc)$ est une arête de U_F de même étiquette. De plus si $c = Q_i(r)$, alors $f(c) = F_i(g(r))$.*

■ Exemple

En Figure 16 est présentée la requête Q définie sur la base de connaissance formée du support en Figure 14 et de la base de faits F formée sur les graphes F_1 et F_2 en Figure 15 et mise sous forme normale. Ce graphe requête formule l'interrogation suivante : « Quel est le nom des cours de plus de 10 heures auquel participe la personne ayant pour nom (non nécessairement complet) "Durant" ? ».

Si l'on interroge la base de connaissance F par cette interrogation Q, on obtient pour réponse le sous-graphe conceptuel factuel de F (mis en gras sur la figure) composé des sommets et arêtes de F dans lesquels se sont projetés les sommets et arêtes du graphe requête. La projection prise en compte est celle définie en Définition 1.26. La seule projection existante de Q dans F fournit la réponse : « L'étudiant de nom "Pierre Durant" participe au cours module24 de 20 heures et de nom "anglais" ». On constate notamment la projection du sommet concept donnée [string: {$like$}"Durant"] dans le sommet concept donnée [string: "Pierre Durant"], du sommet concept donnée [number: {>}10] dans le sommet concept donnée [number: 20], et du sommet concept donnée non-valuée [string: {eq} #] dans le sommet concept donnée [string: "anglais"]. Les projections des autres sommets et arêtes de Q dans F restent inchangées par rapport à ce qui est donné en Définition 1.4

Notons que l'opérateur d'un sommet concept individuel ou d'un sommet concept générique d'un graphe conceptuel requête peut ne pas être représenté, car il s'agit nécessairement de l'opérateur $\preceq$ puisque les opérandes sont des marqueurs individuels dans $I_{\mathcal{F}}^{\sqcup} \cup I_S$ ou le marqueur générique $*$.

1.2.4.4 Règle et contrainte

La définition d'une règle et la définition d'une contrainte (positive ou négative) sont elles aussi adaptées dans le modèle des GCs que nous proposons.

Pour une règle, l'hypothèse est un graphe conceptuel requête et la conclusion est un graphe conceptuel factuel. L'application d'une règle sur un graphe conceptuel factuel utilise notre adaptation de l'opération de la projection en Définition 1.26.

La condition d'une contrainte (positive ou négative), l'obligation (pour une contrainte positive) et l'interdiction (pour une contrainte négative) sont des graphes conceptuels requêtes. La vérification d'une contrainte positive ou négative sur un graphe conceptuel factuel utilise l'opération de la projection en Définition 1.26.

1.3 Outils

Le modèle des GCs a donné lieu à différentes utilisations, principalement dans le domaine du traitement du langage naturel, de l'acquisition et de la représentation de connaissances [Chein et Genest, 2000]. Selon les besoins, différents outils logiciels ont été développés. Ils n'exploitent pas forcément la même variante du modèle et sont souvent limités à un cadre bien précis d'utilisation.

Il existe toutefois un certain nombre d'outils qui fournissent une implantation (d'une variante) du modèle suffisamment générale pour pouvoir être utilisée dans des applications de différentes natures. Ces outils peuvent être classés en trois catégories :

- les bibliothèques permettant le développement d'applications. Elles sont conçues pour être utilisées par des personnes connaissant le modèle, et qui recherchent une implantation des fonctionnalités de base du modèle des GCs.
- les interfaces graphiques d'édition de graphes conceptuels. Elles sont principalement destinées à être utilisées par des non spécialistes du modèle.
- les logiciels (ou suites d'outils) intégrant l'édition et la manipulation, par les opérations du modèle (projection, règles, ...), de graphes conceptuels.

Nous présentons dans cette section quelques principaux outils génériques pour l'édition et la manipulation de graphes conceptuels.

Cogitant et CoGUI

La plateforme Cogitant - *Conceptual Graphs Integrated Tools allowing Nested Typed graphs* - [Genest, 2007] est un outil complet pour la manipulation de graphes conceptuels. Il s'agit d'une bibliothèque de classes C++, pour le développement de logiciels basés sur les graphes conceptuels. Cette bibliothèque permet de manipuler des graphes conceptuels emboîtés typés[8] avec liens de coréférence et types conjonctifs, de vérifier qu'un graphe conceptuel est correctement construit, d'utiliser la projection, d'appliquer des règles et de

[8] Les emboîtements (typés) constituent une autre extension du modèle des GCs simples, et permettent d'emboîter un graphe dans un sommet concept. Ce dernier jouant le rôle de contexte du graphe emboîté [Chein et Mugnier, 1997].

vérifier des contraintes. On peut de plus noter que Cogitant intègre aussi un éditeur minimal de graphes conceptuels emboîtés typés, mais ce n'est pas la fonctionalité première de cette bibliothèque.

CoGUI - *Conceptual Graphs Graphical User Interface* - [Gutierez, 2005] est un projet qui propose une interface graphique en Java pour l'édition de graphes conceptuels et supports, de règles et de contraintes. Cet outil, dont le développement est toujours actif, offre une connexion avec la bibliothèque Cogitant.

Notio et CharGer

Les premiers travaux sur Notio [Southey et Linders, 1999] concernaient la mise au point d'une API (*Application Programming Interface*) définissant les classes et les méthodes permettant de construire une plateforme personnalisable de manipulation de graphes conceptuels. Puis, fin 1999 une implantation en *Java* fût réalisée[9] pour la manipulation de graphes conceptuels. Cette bibliothèque n'est, à ce jour, plus maintenue.

CharGer est une application développée avec *Java* qui permet l'édition de graphes conceptuels sous forme graphique [Delugach, 2001]. Cet outil s'utilise conjointement avec la plateforme Notio et profite de ses possibilités d'inférence, mais reste moins fonctionnel que CoGUI (surtout pour les supports et graphes de grandes tailles).

Amine

Amine [Kabbaj, 2006] est une plateforme Java pour le développement de systèmes basés sur les graphes conceptuels. Amine propose l'édition de graphes conceptuels simples, et utilise les moteurs d'inférence Prolog+CG [Kabbaj *et al.*, 2001] et Synergy [Kabbaj, 1999] pour la projection et l'application de règles.

CGWorld

CGWorld [Dobrev et Toutanova, 2000] est un outil de manipulation de graphes conceptuels basé sur le *Web* et permettant un travail distribué sur une base de connaissances sous la forme de graphes conceptuels. Cet outil *Java* est muni d'une interface graphique, CGWorld CGE, fournie sous la forme d'une *applet* permettant l'édition des graphes, et utilise SICStus Prolog[10] pour les raisonnements.

Autres outils

Il existe un grand nombre d'autres outils de saisie de graphes, comme Graphlet [Himsolt, 1996], et des bibliothèques de manipulation de graphes, telles que Gtl [Forster *et al.*, 1999], Gfc[11] ou Leda [Mehlhorn et Näher, 1999]. Mais ces outils qui n'ont pas été conçus pour le modèle des GCs peuvent difficilement être adaptés à ce modèle. Car, en ce qui

[9] `http://www.cs.ualberta.ca/~finnegan/notio/`
[10] `http://www.sics.se/isl/sicstuswww/site/index.html`
[11] `http://alphaworks.ibm.com/aw.nsf/techmain/gfc`

concerne l'édition de graphes conceptuels, ces outils ne fournissent aucune fonctionnalité facilitant le choix des étiquettes des sommets. Pour ce qui est de la manipulation de graphes conceptuels, ces outils - même s'ils fournissent des opérations telles que la recherche d'homomorphismes - ne prennent pas en considération les connaissances représentées par le support, pourtant nécessaires à la recherche de projections.

1.4 Conclusion

Issu des réseaux sémantiques, le modèle des GCs permet de modéliser de façon formelle des connaissances. Les connaissances d'un domaine sont représentées sur deux niveaux conceptuels de façon relativement intuitive et visuelle. Le support modélise de manière taxonomique les connaissances structurelles, tandis que les connaissances factuelles sont modélisées au sein de graphes. Pour interroger, déduire et vérifier des connaissances, ce modèle dispose de la projection, adéquate et complète vis à vis de la déduction en logique du premier ordre, de règles et de contraintes. Il s'agit donc d'une approche concrète de modélisation disposant d'outils logiciels efficaces pour exploiter les connaissances.

Afin d'assouplir l'utilisation des marqueurs individuels dans le modèle des GCs actuel et de permettre en compte la notion de *données*, nous avons proposé une adaptation et extension du modèle des GCs. Ce travail est basé sur le modèle des GCs de [Mugnier et Chein, 1996] avec types conjonctifs [Chein et Mugnier, 2004]. Notre contribution est triple. Premièrement, deux familles de graphes sont définies. D'une part, les graphes conceptuels factuels sont définis pour représenter les faits d'une modélisation ; ils sont élaborés par un expert du domaine modélisé. Les notions de base de faits et de base de connaissances ont été introduites. D'autre part, les graphes conceptuels requêtes sont définis pour interroger les connaissances factuelles d'une modélisation. Deuxièmement, ce travail offre la possibilité de définir des marqueurs individuels au niveau d'un graphe conceptuel factuel et non plus uniquement au sein d'un support. Cette adaptation du modèle des GCs permet de ramener au niveau factuel la définition de certains individus. Troisièmement, la définition d'un graphe conceptuel (factuel ou requête) est étendue pour modéliser des valeurs numériques et des chaînes de caractères. Une adaptation de l'opération de projection a été réalisée pour prendre en compte cette extension, autorisant notamment certains traitements de comparaison entre les données.

Dans la suite de cette thèse, le modèle des GCs considéré est ce lui que nous avons adapté et étendu.

Chapitre 2

Langages du Web sémantique

À sa création[1] par Tim Berners-Lee[2] au début des années 90, l'objectif du *World Wide Web* (ou *Web* ou *WWW*) était de permettre des échanges de savoirs entre individus distants. C'est dans ce but que fût créé le langage HTML - *HyperText Markup Language* - [Raggett *et al.*, 1999]. Il permet d'une part une mise en forme aisée et rapide de documents à l'aide de *balises* (*tags* en anglais). D'autre part, HTML offre une mise à disposition en réseau de ces documents, dits *hypertextes*, car connectés entre eux via des URIs.

Le Web a fortement évolué en l'espace de quinze ans. Victime de son succès, l'essort considérable qu'a connu le Web a abouti en ce début du XXI^{e} siècle à une prolifération de documents hétérogènes difficiles à consulter. Le langage HTML apparaît comme non adapté au Web d'aujourd'hui. En effet, ce langage fournit un jeu de balises destinées quasi-exclusivement à la mise-en-page de documents hypertextes. Il ne permet donc pas une organisation sémantique de l'information contenue dans les documents HTML. La structure et le contenu de ces documents ont peu à peu été « enfouis » sous une couche présentationnelle qui, si elle n'a pas dénaturé l'information, l'a rendue moins accessible. Pourtant, le langage HTML est initialement dérivé du langage SGML - *Standard Generalized Markup Language* - apparu dans les années 70. SGML a été créé pour gérer de grandes documentations en y séparant la structure, le contenu et la présentation. SGML[3] va de nouveau fournir une base de réflexion pour la diffusion de l'information.

Au cours de cette dernière décennie, une nouvelle idée du Web prend corps : celle d'un *Web Sémantique*. Pour la naissance du Web sémantique, le W3C[4] - *World Wide Web*

[1] Cette introduction historique sur le *Web sémantique* est en partie inspirée de celle donnée par Xavier Lacot (`http://lacot.org/public/introduction_a_owl.pdf`) et par l'encyclopédie collaborative Wikipédia (`http://wikipedia.org/`).

[2] `http://www.w3.org/People/Berners-Lee/`

[3] Normmalisé ISO 8879 :1986, `http://www.iso.org/`.

[4] Le W3C est un consortium fondé en octobre 1994 pour promouvoir la compatibilité des technologies du Web, telles que HTML, XHTML, CSS, XML, RDF, *etc.* Le W3C n'émet pas des normes, mais des recommandations à valeur de standards industriels. Sa gestion est assurée conjointement par le MIT (Massachusetts Institute of Technology) aux États-Unis, l'ERCIM (European Research Consortium for Informatics and Mathematics) en Europe et l'Université Keio au Japon. `http://www.w3.org/`.

Consortium - se dote de nouveaux langages et outils, comme XML, RDF(S) puis OWL. Tous ont un objectif commun, celui de participer à une (re-)formalisation des savoirs selon une nouvelle architecture, et permettre ainsi une meilleure organisation de l'information pour un partage et une diffusion plus aisés. Les champs d'application sont vastes : recherche d'informations pertinentes, résolution de problèmes par inférences, traduction automatisée, *etc.*

De façon générale, l'utilisation de ces langages pour modéliser les connaissances d'un domaine passe par le langage RDF [Klyne et Carroll, 2004] qui organise au sein de *triplets* les connaissances factuelles du domaine, et par le langage RDFS [Brickley et Guha, 2004] pour la définition des connaissances structurelles du domaine. L'utilisation conjointe de RDF et RDFS est souvent notée RDF(S). Le langage OWL - pseudo acronyme de *Web Ontology Language* - est la génération suivante de RDF(S) et dispose d'une très grande richesse expressive. Il s'agit d'un langage formel pour décrire des ontologies mais aussi des faits, et les diffuser sur le Web [Dean *et al.*, 2004]. OWL repose sur la syntaxe de RDF, étend le vocabulaire de RDFS, et possède une sémantique logique issue des logiques de descriptions [Baader *et al.*, 2003]. Ce langage offre ainsi l'avantage d'être formel et de pouvoir disposer de raisonnements issus de ces logiques et applicables sur les connaissances qu'il modélise.

Ce chapitre introduit, dans une première section, les langages XML, RDF et RDFS. La seconde section fournit les principes fondamentaux du Web sémantique et des ontologies, et constitue une présentation du langage OWL. Une troisième et dernière section présente quelques outils logiciels pour l'édition d'ontologies et leurs manipulations en terme de classification ou d'instanciation, mais aussi d'outils plus récents œuvrant dans un cadre comparable à la logique classique pour l'application de règles d'inférences.

Sommaire

2.1 Web sémantique et modélisation de connaissances

Tim Berners-Lee, James Hendler et Ora Lassila présentèrent en mai 2001 l'idée du *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 *et al.*, 2001].

Pour la mise en place d'un Web sémantique, le W3C s'appuie sur un ensemble de langages. La Figure 17 présente une [5] des visions - légèrement adaptée par nos soins - de ces langages, organisés en couches d'expressivité croissante. Cet ensemble de « couches » est appelé, par le W3C, la *pyramide des langages* du Web sémantique.

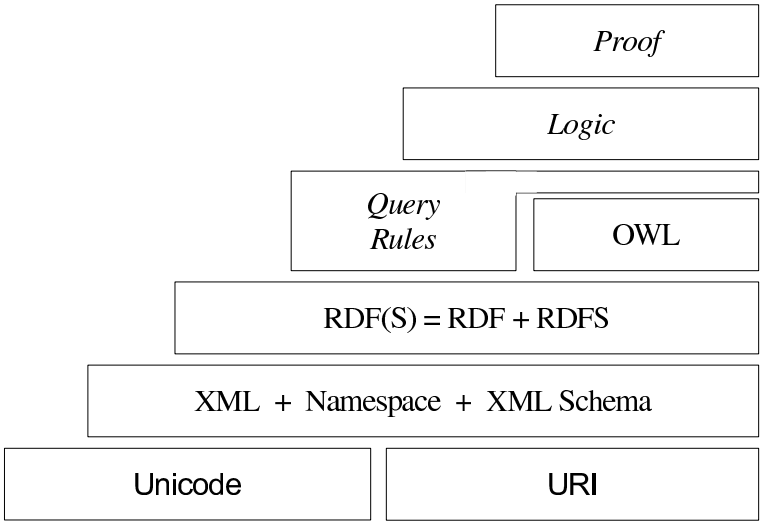


Figure 17 – La pyramide des langages du Web sémantique, par le W3C.

Deux types de bénéfices sont motivés par cette organisation en couches des langages du Web sémantique.

- Elle doit permettre de choisir le langage d'expressivité et de complexité adapté aux besoins d'une modélisation à réaliser.
- Elle permet une approche graduelle dans les processus de standardisation de ces langages naissants (OWL n'a vu le jour qu'en 2001, et est une recommendation du W3C depuis seulement 2004).

Les mots en italique dans cette pyramide ne sont pas des langages à proprement parler, mais ils caractérisent soit des langages en phase d'élaboration soit d'hypothétiques couches.

Une caractéristique commune à ces langages est d'être exprimables selon une syntaxe XML, et d'avoir une capacité à indentifier des ressources (sur le Web notamment) à l'aide d'URIs. Il s'agit d'un mécanisme de base du Web où tous les hyperliens sont exprimés sous forme d'URI. Une URI - *Uniform Resource Identifier* - est une chaîne de caractères

[5] `http://www.w3.org/2006/Talks/1023-sb-W3CTechSemWeb/Overview.html#(19)`

qui permet d'identifier de manière unique une entité élémentaire abstraite ou physique, comme une page sur le Web. La construction d'une URI répond à la recommandation RFC-3986 [6].

Une *ressource*, dans le milieu du Web sémantique, est une entité élémentaire servant à représenter des connaissances. Elle peut donc appartenir au niveau structurel ou au niveau factuel d'une modélisation. L'apparition du langage XML a permis de décrire les ressources portant sur un domaine donné au moyen d'un vocabulaire adapté à ce domaine. En effet, contrairement au HTML qui propose un ensemble limité de balises pour décrire des ressources, l'utilisateur à la possibilité de définir ses propres balises. L'arrivée du langage RDF donne un cadre à la façon de décrire les ressources sur le Web, en proposant l'agencement de ces ressources au sein de *triplets*. C'est avec l'utilisation conjointe des langages RDF et RDFS que les ressources du Web peuvent être organisées sur deux niveaux conceptuels pour représenter les connaissances d'un domaine ; le langage RDF pour représenter les connaissances factuelles d'un domaine et le langage RDFS pour les connaissances structurelles. Ces connaissances structurelles se composent d'un ensemble de *classes* et de *propriétés* (des relations) organisées en *taxonomies*. Pour offrir plus de « richesse » en matière de modélisation, le W3C préconise le langage OWL qui permet de créer des *ontologies* des domaines modélisés. Une *ontologie* offre une granularité plus fine pour décrire et définir les connaissances structurelles d'un domaine par rapport à une taxonomie. Une ontologie permet d'organiser classes et propriétés non seulement en hiérarchies mais aussi selon des liens dits horizontaux, comme la disjonction, l'équivalence, *etc.*

Il n'existe pas de définition communément admise de « ontologie ». Ce terme *ontologie*, qui renvoie étymologiquement à la « théorie de l'existence », est un mot que l'informatique a emprunté à la philosophie au début des années 1990. En philosophie, l'Ontologie est l'étude de « l'être en tant qu'être », des propriétés générales de ce qui existe. Il s'agit de la théorie qui tente d'expliquer les concepts qui existent dans le monde et comment ces concepts s'imbriquent et s'organisent. En Intelligence Artificielle, une définition communément admise d'ontologie est énoncée dans [Gruber, 1993] puis précisée dans [Studer *et al.*, 1998] : *« la spécification formelle et explicite d'une conceptualisation partagée »* (traduction libre). Nous adoptons la définition suivante : *une ontologie est une modélisation formelle d'une conceptualisation structurelle d'un domaine, décrite par des entités génériques appelées classes qui sont organisées par des relations*.

2.2 Les langages XML, RDF et RDFS

2.2.1 eXtensible Markup Language

À sa création au début des années 1990, le web était exclusivement destiné à partager des informations sous forme de pages HTML, affichables par un logiciel (navigateur Web), et destinées à être lues par un utilisateur humain.

[6] `http://tools.ietf.org/html/rfc3986` (version de janvier 2005)

Le langage XML - *eXtensible Markup Language* - [Bray *et al.*, 2006] est une recommandation du W3C depuis le 10 février 1998. Contrairement aux balises HTML qui permettent essentiellement la mise en page de documents Web, les balises XML permettent d'organiser l'information au sein de concepts propres au domaine traité par ces informations. La mise en forme n'en est qu'un cas particulier, voire une conséquence de cette organisation. Pour cela, XML offre la possibilité à l'utilisateur de créer ses propres balises, ce qui n'est pas le cas en HTML. Les contenus des documents XML sont ainsi plus accessibles que ceux des documents HTML difficiles d'accès sous une couche présentationnelle.

De manière plus formelle, XML est un méta-langage[7] facilitant l'élaboration de langages à balises spécialisées. XML fournit un cadre de définition et de structuration des *notions* qui constituent un format de document. Une notion est caractérisée par une balise ou un attribut XML. Ces notions sont ensuites instanciées au niveau des documents qui respectent ce format. L'instantation d'une notion correspond à l'utilisation d'une balise ou d'un attribut défini dans le format. Le format OpenDocument[8] utilisé par la suite bureautique OpenOffice[9] en est un exemple. On peut aussi constater que le langage XHTML [W3C, 2002] est une reformulation (et une révision) en XML de la version 4 du langage HTML. Ce langage XHTML est d'ailleurs considéré par le W3C comme la nouvelle version du HTML.

La définition d'un langage basé sur le méta-langage XML, ou simplement un *langage XML*, se fait soit par une DTD soit par un XSD. Une DTD - *Document Type Definition* - permet la définition simple de la syntaxe et de la grammaire du nouveau langage basé sur XML [Bray *et al.*, 2006]. Sous une forme plus élaborée, cette syntaxe et grammaire sont définies par un XSD - *XML Schema Definition* - [Thompson *et al.*, 2006; Peterson *et al.*, 2006], appelé aussi *Schéma XML*. Principalement, un Schéma XML permet, en plus de ce qui est faisable avec une DTD, de typer les données (booléen, entier, chaîne de caractères, ...) et d'offrir une gamme de cardinalités plus large (pas uniquement 0, 1 ou $*$). Le W3C fournit un ensemble de datatypes prédéfinis [Peterson *et al.*, 2006]. Lorsque nous ferons référence par la suite à ces datatypes, nous parlerons *du* Schéma XML et non d'*un* Schéma XML défini par l'utilisateur (pouvant éventuellement utiliser le Schéma XML).

Remarquons qu'un document utilisant un langage XML est un document XML. Remarquons aussi qu'un tel document doit préciser l'URI où se situe la définition du langage XML avec lequel il est écrit. En effet, alors qu'en HTML le nombre de balises est fixé et connu de tous (tout du moins des navigateurs Web), le nombre de balises XML est potentiellement infini.

2.2.1.1 Format d'un fichier XML

Un document XML est dit *bien formé* s'il est identifié comme étant XML par un entête et s'il satisfait les règles syntaxiques XML (voir ci-dessous). Un document XML bien formé peut être *valide* s'il satisfait de plus les contraintes imposées par la syntaxe et la grammaire du langage, contenues dans une DTD ou un XSD.

[7] Un *méta-langage* se dit d'un langage utilisé pour définir un autre langage.
[8] `http://www.odfalliance.org/`
[9] `http://www.openoffice.org/`

Un document XML est représenté physiquement sous la forme d'un fichier texte structuré en éléments à l'aide de balises, éventuellement imbriquées. Le document est composé d'un en-tête et d'un corps. Dans l'en-tête d'un document XML figurent :

- une déclaration qui identifie le document comme un document XML ;
- un prologue contenant :
  - la version de XML employée (la version 1.1 est recommendée depuis août 2006) ;
  - le codage de caractères ;
  - une DTD, ou bien une référence sur une DTD ou un XSD.

Le corps du document contient les balises XML. Chaque balise XML introduit un *élément XML*. Un tel élément XML est caractérisé de plusieurs manières, non exclusives : par des valeurs d'*attributs*, par d'autres éléments XML imbriqués dits *éléments enfants* ou par les *données* imbriquées. On dit qu'une donnée ou un élément est *imbriqué* s'il figure entre une balise ouvrante et une fermante de même nom. Les règles syntaxiques du corps d'un document XML sont les suivantes :

- Chaque élément doit commencer par une balise ouvrante (de la forme <nom_balise>) et se terminer par une balise fermante (de la forme </nom_balise>). Eventuellement, un élément « vide » (sans balise enfant) peut être représenté par une balise ouvrante et fermante à la fois (de la forme <nom_balise />).
- L'imbrication des éléments du document se fait sans chevauchement. Concrètement, cela signifie que si la balise ouvrante d'un élément se trouve à l'intérieur d'un élément parent, la balise fermante se trouvera dans le même élément.
- Un attribut et sa valeur est située dans la balise ouvrante, et la valeur doit être encadrée de guillemets simples ou doubles.
- XML est sensible à la casse.

De par le non-chevauchement des balises, le corps d'un document XML est structuré sous la forme d'une arborescence ; on parle d'*arbre XML* au sens informatique du terme « arbre ». Cette organisation en arbre a conduit à l'élaboration de différentes APIs [10] qui facilitent la manipulation de documents XML. Les deux APIs principales sont DOM - *Document Object Model* -, qui offre une vision de l'arbre en entier pour son traitement, et SAX - *Simple API for XML* -, qui permet de traiter une partie d'un document XML sans en connaître l'arbre complet.

2.2.1.2 Exemple

La Figure 18(a) présente un fichier XML qui décrit les membres d'une équipe. Ce fichier a pour en-tête les deux premières lignes qui indiquent que c'est un document XML suivant la version 1.1 de XML, et qu'il respecte la DTD "equipe.dtd". Les informations contenues dans le corps de ce fichier sont structurées à l'intérieur de balises XML. L'élément racine de ce document, celui qui contient les autres élément, est défini ici entre une balise ouvrante <équipe> et une fermante </équipe>. Cette équipe est définie d'une part par un attribut nom ayant pour valeur "Team Repsol Mitsubishi Ralliart" et d'autre part par ses éléments enfants. Ces éléments enfants définisent des membres (balises <membre>).

[10] API signifie *interface de programmation*.

```xml
<?xml version="1.1" encoding="utf-8" ?>
<!DOCTYPE équipe SYSTEM "equipe.dtd">

<équipe nom = "Team Repsol Mitsubishi Ralliart">
   <membre>
      <nom>Serieys</nom>
      <prénom>Dominique</prénom>
      <licence numéro = "13" />
   </membre>
   <membre>
      <nom>Alphand</nom>
      <prénom>Luc</prénom>
      <âge>38</âge>
      <licence numéro = "42" />
   </membre>
</équipe>
```

(a) Un fichier XML *valide* : *bien formé* et vérifiant la DTD "equipe.dtd".

```
<!ELEMENT équipe  (membre+)>
<!ATTLIST équipe  nom  CDATA #IMPLIED>

<!ELEMENT membre  (licence, nom, prénom, âge?)>
<!ELEMENT nom     (#PCDATA)>
<!ELEMENT prénom  (#PCDATA)>
<!ELEMENT âge     (#PCDATA)>
<!ELEMENT licence EMPTY>
<!ATTLIST licence  numéro CDATA #REQUIRED>
```

(b) La DTD "equipe.dtd".

Figure 18 – Un fichier XML et sa DTD.

Chaque membre est caractérisé par les éléments enfants nom, prénom, âge et licence. Ces trois premiers éléments sont caractérisés par leurs données imbriquées respectives ; l'élément licence l'est par la valeur de son attribut numéro. Ainsi, l'équipe "Team Repsol Mitsubishi Ralliart" contient deux membres. Le second par exemple a pour nom Alphand, pour prénom Luc, pour âge 38 et comme numéro de licence 42.

Ce fichier XML bien formé, car respectant la syntaxe XML, est de plus valide car il respecte les contraintes imposées par le DTD "equipe.dtd". En effet, cette DTD, présentée en Figure 18(b), indique qu'un élément équipe est composé d'un ou plusieurs (*cf.* le signe $+$) éléments membre. Il peut de plus posséder un attribut nom qui est facultatif (*cf.* le mot-clé #IMPLIED). Un élément membre est composé d'un élément licence, d'un élément nom, d'un élément prénom, et éventuellement (*cf.* le signe ?) d'un élément âge. L'élément licence ne peut contenir d'élément enfant (*cf.* le mot-clé EMPTY), mais doit posséder un attribut numéro (*cf.* le mot-clé #REQUIRED). Les mot-clés #PCDATA et CDATA indiquent qu'il s'agit d'une donnée quelconque, c'est-à-dire une suite de caractères non interprétables comme du XML.

Le document XSD - *XML Schema Definition* - en Figure 19 présente un Schéma XML, qui peut être employé à la place de la DTD en Figure 18(b) pour définir la syntaxe et la grammaire du langage utilisé dans le fichier XML en Figure 18(a). Ce document XSD se lit assez aisément. Nous le commentons tout de même en attirant l'attention sur ce qu'il apporte de plus comme contraintes dans l'écriture d'un fichier XML basé sur un tel

```
1 <?xml version="1.1" encoding="utf-8"?>
2 <xsd:schema   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3
4   <xsd:complexType name="équipe">
5     <xsd:sequence>
6       <xsd:element   name="membre"  type="membreType" minOccurs="1" />
7       <xsd:attribute name="nom"     type="xsd:string"/>
8     </xsd:sequence>
9   </xsd:complexType>
10
11  <xsd:complexType name="membreType">
12    <xsd:sequence>
13      <xsd:element name="nom"     type="xsd:string"  minOccurs="1" maxOccurs="1" />
14      <xsd:element name="prénom"  type="xsd:string"  minOccurs="1" maxOccurs="3" />
15      <xsd:element name="âge"     type="xsd:integer" minOccurs="0" maxOccurs="1" />
16      <xsd:element name="licence" type="licenceType" minOccurs="1" maxOccurs="1" />
17    </xsd:sequence>
18  </xsd:complexType>
19
20  <xsd:simpleType name="licenceType">
21    <xsd:attribute name="numéro" type="xsd:integer"/>
22  </xsd:complexType>
23
24 </xsd:schema>
```

Figure 19 – Un *Schéma XML (XSD)* pouvant servir d'alternative à la DTD en Figure 18(b) pour définir les méta-données du fichier XML en Figure 18(a).

Schéma XML. En plus des contraintes données par la DTD, cette XSD impose le nombre minimal et maximal des balises enfants d'un certain type qui peuvent être imbriquées dans une balise parent. Par exemple un élément membre doit posséder entre un et trois éléments enfants prénom. Cette XSD impose aussi le type des données imbriquées dans une balise. Par exemple la donnée imbriquée dans un élément âge doit être un nombre (xsd:integer). Notons que le mot xsd, en Figure 19 désigne un *espace de nommage* (voir la section suivante).

2.2.1.3 Espace de nommage

Un *espace de nommage* (*namespace* en anglais) correspond à un regroupement d'éléments et d'attributs au sein d'un même « espace », généralement un même fichier XML. Il se présente sous la forme d'une URI. Le nom complet d'un élément ou attribut - son URI - est composé de son espace de nommage et de son nom local.

La spécification d'un espace de nommage se fait en utilisant l'attribut spécial xmlns. La valeur de cet attribut est prise comme espace de nommage, dit par défaut, pour l'élément de l'attribut, ainsi que pour tous ses éléments enfants (mais pas pour ses autres attributs).

Il est possible d'associer à un espace de nommage un (voire plusieurs) nom logique, également appelé *préfixe d'espace de nommage* (*namespace prefix*). Ceci permet de faciliter l'écriture d'un élément XML, mais s'avère indispensable dès lors que plusieurs espaces de nommages sont utilisés simultanément, et permet de désambiguïser deux noms locaux provenant de deux espaces de nommage différents. Un tel espace de nommage se déclare en utilisant un attribut de la forme xmlns:prefix, où prefix est le préfixe que l'on sou-

haite associer à l'espace de nommage. Comme pour le cas précédent, la valeur de cet attribut spécifie l'espace de nommage concerné. On peut ensuite spécifier pour chaque élément ou attribut l'espace de nommage utilisé, avec la notation prefix:nom où prefix est un préfixe d'espace de nommage et où nom est le nom local de l'élément ou de l'attribut. Cette notation est appelée *nom qualifié* (*qualified name*). En l'absence de préfixe, c'est l'espace de nommage par défaut qui est utilisé ; et en l'absence d'espace de nommage par défaut c'est l'URI du fichier XML lui-même qui est pris en compte. L'attribut xml:base permet de renseigner l'URI du fichier XML.

Prenons l'exemple suivant,

```
<livres  xmlns="http://www.meslivres.fr"  xml:base="http://www.meslivres.fr"
         xmlns:encyclo="http://tousleslivres.org">
  <livre>
    <auteur>Douglas Adams</auteur>
    <titre>Le Guide du voyageur galactique</titre>
    <encyclo:titre>The Hitchhiker's Guide to the Galaxy</encyclo:titre>
    <année>1982</année>
    <encyclo:année>1979</encyclo:année>
  </livre>
</livres>
```

où tous les noms d'éléments sont dans l'espace de nommage par défaut http://www.meslivres.fr, sauf le deuxième élément titre et le deuxième élément année, qui sont dans l'espace de nommage http://tousleslivres.org. Ainsi, titre et année font référence aux éléments respectifs http://www.meslivres.fr#titre et http://www.meslivres.fr#année, tandis que encyclo:titre et encyclo:année font référence à http://tousleslivres.org#titre et à http://tousleslivres.org#année. Ces deux derniers éléments pourraient renseigner le titre original et l'année de parution de cette version originale du livre, alors que titre et année pourraient renseigner la version française du livre. Le caractère # fait la séparation entre l'espace de nommage, à gauche du caractère, et le nom local, à droite du caractère, d'une URI [11].

2.2.2 Resource Description Framework

L'arrivée de XML, en 1998, a donné un cadre à la structuration des informations présentes dans des documents disponibles sur le Web. Cependant, la grande souplesse d'utilisation de XML a entrainé une mise en œuvre très différente d'un document XML à un autre, même pour décrire des informations similaires. Cette disparité d'utilisation vient surtout du choix de l'emploi d'une balise ou d'un attribut pour renseigner une donnée. Prenons le cas du document XML en Figure 18(a) : la donnée 38 qui renseigne l'âge de Luc Alphand est imbriquée entre les balises <âge> et </âge> ; la donnée 42 qui renseigne le numéro de licence est sous la forme d'une valeur de l'attribut numéro de la balise licence. Un cas extrême d'utilisation d'attributs, au détriment de balises, est présenté en Figure 20 et reprend les informations présentes en Figure 18(a) (basé bien sûr sur une autre DTD ou une autre XSD). Il n'y a pas de règle pour dire quand utiliser un attribut plutôt qu'une balise. Cependant un consensus consiste à utiliser des attributs uniquement pour identifier ou référencer un élément XML. C'est ce qui est adopté par le langage RDF, avec

[11] Notons qu'une URI qui identifie une ressource par son nom local dans un espace de nommage, est appelée une URN - *Uniform Resource Name* -.

d'une part les attributs rdf:about ou rdf:ID pour identifier une ressource et d'autre part les attributs rdf:resource et rdf:datatype pour respectivement faire référence à une ressource ou à un datatype (voir par exemple la Figure 23).

```
1 <?xml version="1.1" encoding="utf-8" ?>
2
3 <équipe nom = "TMRM Team Repsol Mitsubishi Ralliart">
4    <membre nom ="Serieys", prénom ="Dominique", numéro_licence ="13" />
5    <membre nom ="Alphand", prénom ="Luc", âge ="38", numéro_licence ="42" />
6 </équipe>
```

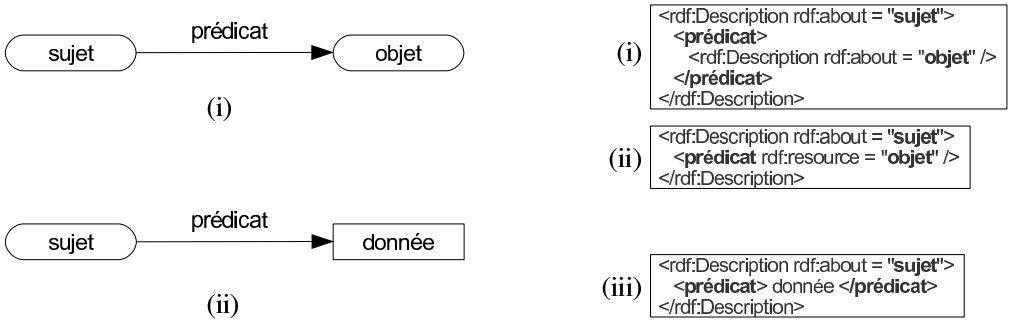
Figure 20 – Un fichier XML utilisant massivement des attributs pour caractériser les éléments XML, plutôt que l'emboîtement de balises.

C'est dans cet esprit de structurer l'information de façon plus homogène qu'a été créé en 1999, par le W3C, le langage RDF - *Resource Description Framework* - [Klyne et Carroll, 2004]. Il s'agit là aussi d'un méta-langage, mais qui cette fois-ci « recadre » la façon d'organiser l'information. Pour ce faire, un document écrit en RDF, aussi appelé *expression RDF*, organise l'information sous forme d'unités élémentaires qui sont toutes décrites par une même structure fixée. La structure fondamentale en RDF est un *triplet*, composé d'un *sujet*, un *prédicat* et un *objet*. Un triplet s'interprète comme suit : « le sujet a pour prédicat l'objet ». Par exemple, l'information « le prénom de Alphand est Luc » se reformulerait en « Alphand a pour prénom Luc ». Chaque composant du triplet est aussi appelé sous le terme générique de *ressource*, mais est généralement utilisé pour parler du sujet ou de l'objet.

Une expression RDF est aussi appelée un *graphe RDF*, le mot « graphe » étant à prendre au sens informatique du terme. Il s'agit plus précisement d'un multi-graphe orienté et étiqueté. Elle peut être illustrée par une figure composée de noeuds et d'arcs dirigés, dans laquelle chaque triplet est représenté par un lien noeud-arc-noeud. Le premier noeud a pour étiquette le sujet, le second noeud a pour étiquette l'objet, et l'arc orienté du sujet vers l'objet a pour étiquette le prédicat. Dans le cas où l'objet est simplement une donnée, il est représenté par une autre famille de noeud (en général un rectangle au lieu d'une ellipse). En effet une donnée n'est plus considérée comme une ressource RDF, car elle ne peut être le sujet d'un autre triplet. En théorie des graphes il s'agit d'un *puits*, c'est-à-dire qu'il ne peut y avoir d'arc qui parte de ce sommet. La Figure 21(a) présente un triplet RDF <*sujet, prédicat, objet*> sous la forme de son graphe, et du cas particulier <*sujet, prédicat, donnée*>.

2.2.2.1 Format d'un document RDF

Une description RDF est généralement représentée dans un fichier XML, mais ce n'est en rien une nécessité. En effet, les triplets d'une description RDF peuvent être représentés par n'importe quel format de graphe dans l'absolu. On utilise la notation XML/RDF pour préciser explicitement qu'une description RDF est stockée dans un (des) fichier(s) XML. Un fichier XML/RDF a pour balise racine de son arbre la balise <rdf:RDF>. Cette balise racine possède comme attributs les espaces de nommage utilisés dans le document RDF. Il y a a minima l'espace de nommage rdf pour pouvoir interpréter la balise racine ou toute



(a) Représentation sous forme d'un graphe d'un triplet RDF <*sujet, prédicat, objet*> (i), et d'un triplet <*sujet, prédicat, donnée*> (ii).

(b) Représentation en XML d'un triplet RDF <*sujet, prédicat, objet*> (i, ii), et d'un triplet <*sujet, prédicat, donnée*> (iii).

Figure 21 – Représentation sous forme d'un graphe RDF ou en XML de triplets RDF.

autre balise RDF. L'attribut xml:base, qui renseigne l'URI du document lui-même, permet lorsqu'une ressource n'est associée à aucun espace de nommage (pas même l'espace de nommage par défaut) d'indiquer que cette ressource fait partie du document XML/RDF. En pratique, cet attribut xml:base peut possèder la même valeur que celle de l'attribut xmlns de l'espace de nommage par défaut. Ainsi, les ressources utilisées (identifiées ou référencées) dans un document sont par défaut celles du document.

Un triplet RDF <*sujet, prédicat, objet*> est considéré en XML/RDF comme une *description* du sujet : « le sujet a pour prédicat l'objet »). Ce sujet est référencé par une balise rdf:Description qui possède un attribut rdf:about ayant pour valeur l'URI du sujet (voir Figure 21(b)). Cette balise possède une balise enfant ayant pour nom le prédicat. Ce prédicat possède à son tour soit une balise enfant rdf:Description ayant pour valeur d'attribut rdf:about l'objet, si l'objet est une ressource (voir Figure 21(b) (i)), soit une donnée imbriquée, sinon (voir Figure 21(b) (iii)). La Figure 21(b) (ii) exprime de façon compacte quel est l'objet du prédicat ; le triplet présenté sur cette figure est équivalent à celui en Figure 21(b) (i).

2.2.2.2 Exemple

La description RDF, sous la forme de son graphe RDF en Figure 22 et au format XML en Figure 23, reprend principalement les informations contenues dans l'exemple en Figure 18(a). De manière à respecter la syntaxe RDF sous forme de triplets, l'*équipe* représentée par la balise <équipe> en Figure 18(a) est identifiée ici par la ressource #TRMR. De même, les *membres* représentés par les deux balises <membre> sont identifiés par les ressources #serieys et #alphand. Finalement, une balise <#être_composé> a été ajoutée pour être utilisée comme prédicat entre #TRMR et #serieys ou #alphand. Cependant, contrairement aux informations contenues dans le fichier XML en Figure 18(a), on remarque que sur l'expression RDF de cet exemple les ressources #TRMR, #serieys et #alphand ne sont plus respectivement associées aux concepts d'*équipe* et de *membre*. En fait, RDF s'emploie rarement seul : une expression RDF utilise des éléments du langage RDFS, comme les *classes* qui constituent les types d'ensembles de ressources ayant des caractéristiques

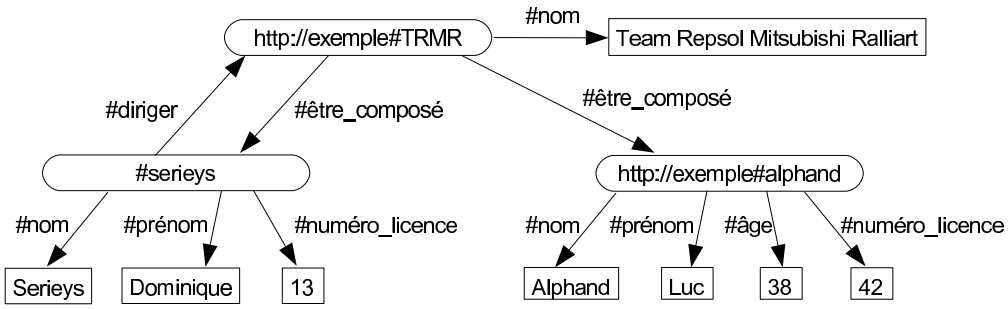


Figure 22 – Une description RDF, sous la forme de son graphe RDF.

```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          xmlns="http://exemple.eu#"  xml:base="http://exemple.eu">

<rdf:Description rdf:about = "#TMRM">
  <nom>Team Repsol Mitsubishi Ralliart</nom>
  <être_composé>
    <rdf:Description rdf:about = "#serieys">
      <nom>Serieys</nom>
      <prénom>Dominique</prénom>
      <numéro_licence>13</numéro_licence>
    </rdf:Description>
  </être_composé>
  <être_composé>
    <rdf:Description rdf:ID = "alphand">
      <nom>Alphand</nom>
      <prénom>Luc</prénom>
      <âge>38</âge>
      <numéro_licence>42</numéro_licence>
    </rdf:Description>
  </être_composé>
</rdf:Description>
<rdf:Description rdf:about = "#serieys">
  <diriger rdf:resource="#TMRM" />
</rdf:Description>

</rdf:RDF>
```

Figure 23 – Une description RDF, sous la forme d'un document XML, reprenant les informations du graphe RDF en Figure 23.

communes (voir Section 2.2.3 ci-après). De plus, étant donné que seuls des attributs prédéfinis, comme rdf:about ou rdf:resource existent en XML/RDF, l'attribut numéro associé à un élément licence en Figure 18(a) est replacé ici par une balise numéro_licence utilisée en prédicat de triplets.

Les attributs rdf:about et rdf:ID font partie intégrante du langage XML/RDF afin d'identifier une ressource RDF. Notons que l'utilisation de rdf:ID est une facilité syntaxique pour éviter de « déclarer » deux fois la même ressource (même URI). En effet, une même URI peut être utilisée plusieurs fois comme valeur d'un attribut rdf:about, alors qu'elle ne peut l'être qu'une seule fois avec l'attribut rdf:ID. En d'autres termes, l'utilisation de l'attribut rdf:ID au sein d'une balise contraint la ressoure associée à cet identifiant à être caractérisée uniquement via cette balise ou ses balises imbriquées. Tandis que l'utilisation de l'attribut rdf:about permet de compléter - éventuellement de commencer - la description de la ressource identifiée. Sur la Figure 23 par exemple, la ressource #alphand [12] est caractérisée par une unique description, tandis que la ressource #serieys est dans un premier temps caractérisée par son nom, prénom et numéro de licence, puis dans un second temps par le fait que #serieys dirige l'équipe #TRMR.

Remarquons qu'avec un fichier XML « pur » (pas une description RDF), l'organisation en arbre des balises rend impossible, en Figure 18(a) par exemple, le fait d'exprimer qu'une équipe est composée d'un membre qui la dirige. En effet, soit on dispose d'une balise équipe qui possède une balise enfant membre, mais alors on ne peut plus inclure l'équipe comme une balise enfant du membre qui la dirige (colonne de gauche ci-dessous). Soit on dispose d'une balise membre qui possède une balise enfant équipe pour celle dirigée, mais alors on ne peut plus inclure le membre comme une balise enfant membre de l'équipe (colonne de droite ci-dessous).

```
<équipe nom="TRMR">
  <membre>
    <nom>Serieys</nom>
    <prénom>Dominique</prénom>
    <licence numéro = "13" />
    <!--
      <diriger> ??? </diriger>
    -->
  </membre>
  ...
</équipe>
```

```
<membre>
  <nom>Serieys</nom>
  <prénom>Dominique</prénom>
  <licence numéro = "13" />
  <diriger>
    <équipe nom="TRMR">
      ...
    </équipe>
  </diriger>
  <!-- <est_membre_de> ??? </est_membre_de> -->
</membre>
```

En RDF, les ressources étant identifiées, il est possible comme en Figure 23 de réaliser de telles « références croisées ». Une expression XML/RDF constitue donc bien un graphe, et non simplement un arbre de par la syntaxe XML utilisée.

2.2.2.3 Avertissement

Dans la suite de cette thèse nous utilisons la « version » XML de RDF. C'est-à-dire que premièrement, les notions qui constituent le langage RDF sont définies par des balises XML. Deuxièmement, une expression RDF est un document XML. Troisièmement, les

[12] Lorsque l'attribut rdf:ID est employé, le caractère # ne figure pas devant le nom local de l'URI identifiant la ressource lorsque celle-ci est dans l'espace de nommage par défaut.

URIs sont utilisées pour identifier de manière unique les ressources d'une expression RDF.

À ce choix s'ajoute le fait que nous ferons l'amalgame entre d'une part les balises XML et leurs agencements syntaxiques et d'autre part la sémantique des notions sous-jacentes à ces balises et de tels agencements. Ce choix permettera d'éviter certaines longeurs rédactionnelles, et se justifie car actuellement XML est dans la grande majorité des cas utilisé pour « écrire » une description RDF.

2.2.3 RDF Schema

À partir de RDF a été proposé dès mars 1999 le langage RDFS - *RDF Schema* - [Brickley et Guha, 2004], qui a pour but d'étendre RDF en fournissant une structuration aux ressources RDF utilisées. Pour cela, il permet l'élaboration de *classes* et de *propriétés*. Une classe constitue un *type* pour un ensemble de ressources (sujets ou objets) ; et une propriété constitue un *type* pour un ensemble de prédicats. Ces ressources (respectivement prédicats) sont appelées des *assertions*, c'est-à-dire qu'elles [13] ont pour type une ou plusieurs classes (respectivement propriétés). L'ensemble des assertions d'un même type constituent l'*extension* de ce type, c'est-à-dire l'extension d'une classe ou l'extension d'une propriété. Notons, qu'une assertion dont le type est une classe est aussi appelée une *instance* de cette classe ; ce terme est souvent préféré dans ce cas. RDFS propose aussi le moyen d'organiser de manière *taxonomique*, c'est-à-dire en hiérarchie, des ensembles de classes ainsi que des ensembles de propriétés. De plus, la notion de *datatype* pour désigner un type de données est incorporée au langage RDFS. Dans la « version » XML de RDFS, notée XML/RDFS, les datatypes du Schéma XML [Peterson *et al.*, 2006] sont utilisées.

L'utilisation conjointe de RDF et de RDFS, notée RDF(S), offre donc la possibilité de représenter les connaissances d'un domaine sur deux niveaux conceptuels. Les éléments modélisés en RDFS représentent les *connaissances structurelles* du domaine représenté ; les éléments modélisés en RDF [14] représentent les *connaissances factuelles*, encore appelées connaissances assertionnelles.

Notons que RDFS n'est pas pour RDF ce qu'un Schéma XML est pour XML. Un Schéma XML contraint la structure d'un document XML d'un point de vue purement syntaxique, tandis que RDFS permet de structurer les éléments RDF selon une structure définie à un niveau générique par les classes et les propriétés, en tenant compte des hiérarchisations éventuelles entre classes et entre propriétés.

On appelera une *propriété intégrée* un élément du vocabulaire RDF(S) qui est utilisé comme prédicat d'un triplet RDF pour décrire une classe ou une propriété du domaine modélisé.

[13] *Assertion* est un nom féminin.

[14] mais n'appatenant pas, bien sûr, aux éléments RDFS (puisque qu'un élément RDFS est aussi par construction un élément RDF)

2.2.3.1 Format d'un document RDFS

Un document RDFS est un document RDF. Ceci est dû au fait que le langage RDFS est défini à l'aide du (méta-)langage RDF ; voir la partie du haut intitulée « extrait du vocabulaire RDFS » sur la Figure 24. Les définitions des classes et des proriétés en RDFS, ainsi que leurs organisations en hierarchies respectives, ne nécessitent pas d'être incorporées dans le même document RDF contenant les faits définis sur ces classes et propriétés. Une connaissance factuelle est liée au niveau structurel de la modélisation par la propriété intégrée rdf:type : elle indique qu'une ressource est une instance d'une classe ou qu'un prédicat est une instance d'une propriété.

Pour définir et organiser les classes et proriétés, le langage RDFS propose un ensemble de notions qui en XML/RDF(S) correspondent à des balises et à des attributs prédéfinis. Nous présentons maintenant ce vocabulaire RDFS.

■ **Classe**

Le langage RDFS fournit un mécanisme d'abstraction permettant de regrouper en *classes* des ressources ayant des caractéristiques communes. Une classe est définie par un triplet RDF. Ce triplet a syntaxiquement pour sujet une référence sur la classe (son nom), donnée par l'attribut rdf:about ou rdf:ID. Le prédicat de ce triplet est la propriété rdf:type, et l'objet est la ressource rdfs:Class.

```
<rdf:Description rdf:about = "NomDeLaClasse">
  <rdf:type rdf:resource = "&rdfs;Class" />
</rdf:Description>
```

Un raccourci d'écriture pour déclarer une classe est possible, et est généralement utilisé, à savoir :

```
<rdfs:Class rdf:about = "NomDeLaClasse" />
```

La propriété intégrée rdfs:subClassOf permet d'organiser les classes en hiérarchies. L'exemple suivant indique que la classe Homme est une sous-classe de (une « sorte-de ») Personne. Ainsi, un individu déclaré du type de la classe Homme sera aussi considéré de type Personne.

```
<rdfs:Class rdf:about="Homme">
  <rdfs:subClassOf rdf:resource="#Personne" />
</rdfs:Class>
```

■ **Datatype**

Un type de donnée, ou *datatype*, est une entité générique regroupant un ensemble de données, comme les entiers ou les chaînes de caractères.

Le langage XML/RDF fournit un système de typage de données permettant d'utiliser les datatypes prédéfinis du Schéma XML [Peterson *et al.*, 2006]. Pour cela, XML/RDF définit d'une part la ressource rdf:XMLLiteral qui est le type le plus général des tous les datatypes du Schéma XML. D'autre part, la ressource rdf:datatype est utilisée comme attribut pour typer une donnée.

Les datatypes prédéfinis par le Schéma XML et recommendés par le W3C sont principalement :

- xsd:string, le type des chaînes de caractères ;
- xsd:boolean, le type booléen ;
- différents types numériques, tel que xsd:decimal, xsd:float, xsd:double, et tous les dérivés de xsd:decimal : xsd:integer, xsd:positiveInteger, xsd:nonPositiveInteger, xsd:negativeInteger, xsd:nonNegativeInteger, xsd:long, xsd:int, *etc.* ;
- les types relatifs au temps, comme xsd:date, xsd:time, xsd:dateTime, *etc.* ;
- les types relatifs à une période de temps, comme xsd:duration.

■ Propriété

Comme pour les classes, RDFS permet de regrouper en *propriétés* des prédicats ayant des caractéristiques communes. Une propriété est elle aussi syntaxiquement définie par un triplet RDF, où le sujet est une référence sur la propriété (son nom), le prédicat est rdf:type, et l'objet est la ressource rdf:Property. Une version syntaxique raccourcie est aussi possible pour la déclaration de propriétés.

```
<rdf:Description rdf:about = "NomDeLaPropriété">
  <rdf:type rdf:resource = "&rdf;Property" />
</rdf:Description>
```

ou bien :

```
<rdf:Property rdf:about = "NomDeLaPropriété" />
```

Les propriétés ont une direction d'application, d'un *domaine* (*domain* en anglais) à un *co-domaine* (ou *image*, ou *range* en anglais).

La propriété intégrée rdfs:domain relie une propriété à une classe. Cette propriété intégrée fait valoir que les sujets de la propriété doivent appartenir à l'extension de la classe indiquée. L'utilisation multiple de rdfs:domain doit être interprétée comme une intersection des extensions de classe des classes impliquées.

La propriété intégrée rdfs:range relie une propriété soit à une classe soit à un datatype du Schéma XML [Peterson *et al.*, 2006]. Cette propriété intégrée fait valoir que les objets de le propriété doivent appartenir à l'extension de la classe ou du datatype indiqué. L'utilisation multiple de rdfs:range doit être interprétée comme une intersection des extensions de classe des classes impliquées.

L'exemple suivant introduit la propriété conduire_quad, dont ses assertions auront pour domaine un individu de la classe Pilote et pour co-domaine une instance à la fois de la classe Auto et Moto.

```
<rdf:Property rdf:about = "#conduire_quad">
  <rdfs:domain rdf:resource="#Pilote"/>
  <rdfs:range rdf:resource="#Auto"/>
  <rdfs:range rdf:resource="#Moto"/>
</rdf:Property>
```

■ Assertion

Une *instance* est l'assertion d'une ou plusieurs classes, qui en sont ses *types*. Elle est explicitement connectée à une classe par la propriété intégrée rdf:type. La déclaration d'une instance se fait par un triplet RDF dont syntaxiquement le sujet est une référence sur l'individu (son URI), le prédicat est rdf:type, et l'objet le nom (l'URI) de la classe. Une version syntaxique raccourcie est possible pour la déclaration de individu, avec une balise de

nom celui de la classe. Le typage de l'instance est alors implicitiment celui du nom de la balise.

Par exemple, déclarer que alphand est un Homme s'écrit :

```
<rdf:Description rdf:about = "#alphand">
  <rdf:type rdf:resource="#Homme" />
</rdf:Description>
```

ou bien (propriété rdf:type implicite) :

```
<Homme rdf:about="#alphand" />
```

L'assertion d'une propriété permet d'associer deux instances entre elles ou une instance et une donnée. Un lien est représenté par un triplet RDF où le prédicat est le type du lien. En XML/RDF, si ce prédicat lie une instance à une donnée, le type de la donnée (c'est-à-dire un datatype) est précisé. L'exemple suivant lie l'instance alphand à l'instance serieys par une assertion de la propriété a_pour_chef, et alphand à la donnée "Luc" (de type xsd:string) par une assertion de la propriété a_pour_prénom.

```
<Homme rdf:about = "#alphand">
  <a_pour_chef  rdf:resource="#serieys" />
  <a_pour_prénom rdf:datatype="&xsd;string">Luc</a_pour_prénom>
</Homme>
```

2.2.3.2 Exemple

L'exemple de modélisation XML/RDF(S) en Figure 24, ou en Figure 25 dans la « version textuelle » du graphe RDF(S), reprend et complète la description RDF en Figure 23 telle que cette fois-ci les connaissances modélisées le sont sur deux niveaux conceptuels : l'un structurel et l'autre factuel.

Au niveau des connaissances structurelles de la modélisation, on observe notamment l'organisation taxonomique entre les classes Personne et Membre obtenue à l'aide de la propriété rdfs:subClassOf, ou la définition du domaine et du co-domaine du rôle *âge* à l'aide des propriétés respectives rdfs:domain et rdfs:range. On constate que le rôle *nom* ne précise pas quel est son domaine, dans ce cas il s'agit par défaut de la classe prédéfinie rdfs:Thing.

Au niveau des faits, en plus du typage des individus, comme par exemple l'individu alphand instance de la classe Membre, toute valeur de rôle lorsque celui-ci est un rôle de types de données est explicitement rattachée à un datatype. Par exemple la valeur de rôle 38 liée à l'individu alphand par une assertion du rôle âge est considérée comme de type xsd:integer. L'usage de l'attribut rdf:datatype est nécessaire ici pour indiquer qu'il s'agit par modélisation de l'entier 38 et non de la chaîne de caractères "38" par exemple. Ceci permet de lever toute ambiguïté sur le type de la donnée et s'assurer que l'assertion de rôle de type âge est conforme au co-domaine de ce rôle.

Notons qu'il n'est pas nécessaire que la modélisation XML/RDF(S) de cet exemple soit décrite dans un seul document. Elle aurait pu être répartie sur plusieurs documents, eux-mêmes disponibles en plusieurs endroits accessibles sur le Web.

Comme tout document écrit sur une syntaxe XML, le document RDF(S) en Figure 25 occupe un grand nombre de lignes. Afin de soulager la lisibilité de cette figure, l'en-tête et la balise racine <rdf:RDF> n'y sont que mentionnés en commentaire. Ces en-tête et balise racine sont les suivants :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE raccourcis [
    <!ENTITY xsd  "http://www.w3.org/2001/XMLSchema#" >
```

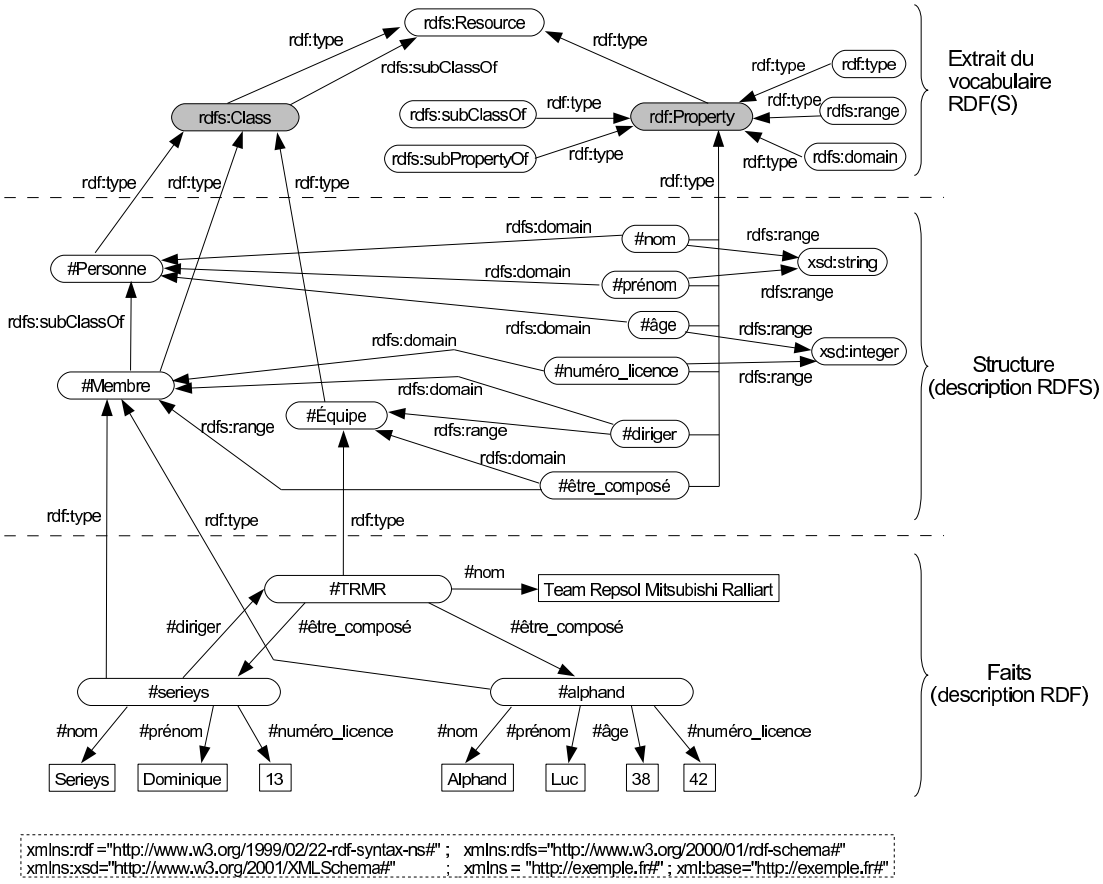


Figure 24 – Une description RDF basée sur une structure décrite en RDFS, elle-même définie sur des éléments RDF(S) prédéfinis.

```
1 <!-- En-tete : encodage et version XML,
2                doctrine DTD et
3                espaces de nommages. -->
4
5 <rdf:Description rdf:about = "#Équipe">
6   <rdf:type rdf:resource = "&rdfs;Class" />
7 </rdf:Description>
8 <rdfs:Class rdf:about = "#Personne" />
9 <rdfs:Class rdf:about = "#Membre">
10  <rdfs:subClassOf rdf:resource = "#Personne" />
11 </rdfs:Class>
12
13 <rdf:Description rdf:about = "#nom">
14  <rdf:type rdf:resource = "&rdf;Property" />
15  <rdfs:range  rdf:resource = "&xsd;string" />
16 </rdf:Description>
17 <rdf:Property rdf:about = "#prénom">
18  <rdfs:domain rdf:resource = "#Personne" />
19  <rdfs:range  rdf:resource = "&xsd;string" />
20 </rdf:Property>
21 <rdf:Property rdf:about = "#âge">
22  <rdfs:domain rdf:resource = "#Personne" />
23  <rdfs:range  rdf:resource = "&xsd;integer" />
24 </rdf:Property>
25 <rdf:Property rdf:about = "#numéro_licence">
26  <rdfs:domain rdf:resource = "#Membre" />
27  <rdfs:range  rdf:resource = "&xsd;integer" />
28 </rdf:Property>
29 <rdf:Property rdf:about = "#diriger">
30  <rdfs:domain rdf:resource = "#Membre" />
31  <rdfs:range  rdf:resource = "#Équipe" />
32 </rdf:Property>
33
34 <rdf:Description rdf:about = "#TMRM">
35  <rdf:type>
36    <rdf:Description rdf:about = "#Équipe" />
37  </rdf:type>
38  <nom>Team Repsol Mitsubishi Ralliart</nom>
39  <être_composé>
40    <rdf:Description rdf:about = "#serieys">
41      <rdf:type rdf:resource = "#Membre" />
42      <nom rdf:datatype="&xsd;string">Serieys</nom>
43      <prénom rdf:datatype="&xsd;string">Dominique</prénom>
44      <numéro_licence rdf:datatype="&xsd;integer">13</numéro_licence>
45      <diriger rdf:resource="#TMRM"/>
46    </rdf:Description>
47  </être_composé>
48  <être_composé>
49    <Membre rdf:about = "#alphand">
50      <nom rdf:datatype="&xsd;string">Alphand</nom>
51      <prénom rdf:datatype="&xsd;string">Luc</prénom>
52      <âge rdf:datatype="&xsd;integer"> 38 </âge>
53      <numéro_licence rdf:datatype="&xsd;integer">42</numéro_licence>
54    </Membre>
55  </être_composé>
56 </rdf:Description>
```

Figure 25 – Une description RDF (lignes 34 à 56) et sa structure RDFS (lignes 5 à 32), sous la forme d'un document XML, reprenant les informations en Figure 24.

```
    <!ENTITY rdf  "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY ex   "http://exemple.fr#" >
  ]>
<rdf:RDF
  xmlns:rdfs="&rdfs;" xmlns:rdf="&rdf;" xmlns:xsd="&xsd;"
  xmlns="&ex;" xml:base="http://exemple.fr#">

  <!-- ici le contenu XML/RDF(S) -->
</rdf:RDF>
```

On y trouve la déclaration des espaces de nommages utilisés dans le document, comme lors de l'utilisation des ressources rdfs:Class ou xsd:integer faisant ainsi référence aux ressources http://www.w3.org/2000/01/rdf-schema#Class et http://www.w3.org/2001/XMLSchema#integer. L'utilisation d'une DTD pour définir certaines constantes correspondant à des URIs peut sembler redondante avec la déclaration des espaces de nommages. Il n'en est cependant rien car l'utilisation d'un nom qualifié n'est syntaxiquement pas possible à l'intérieur d'une valeur d'un attribut XML, comme on serait tenté de la faire avec xsd:string en ligne 15 pour renseigner le co-domaine du rôle nom. Ici, c'est une entité de la DTD qui est utilisée et est interprétée pour pouvoir faciliter - ou désambiguïser - l'écriture de la valeur de l'attribut rdf:resource. Notons qu'une entité d'une DTD est encadrée entre le caractère « et commercial » '&' et un point virgule ';' lorsqu'elle est utilisée entre des guillemets, ici comme valeur d'un attribut.

2.3 Le langage OWL

Le langange OWL - pseudo acronyme de *Web Ontology Language* - [Dean *et al.*, 2004] doit son nom au terme *ontologie*. Ce mot tire son origine du grec (*onto* signifiant « étude » et *logos* le participe présent du verbe « être »), et est emprunté à la philosophie même s'il ne fut manifestement créé qu'au XVII^{e} siècle. L'Ontologie constitue, selon Aristote, aux « Discours sur l'être en tant qu'être »,. L'Ontologie est considérée comme une branche de la *métaphysique* [15]. Dans son traité « Catégories » [Aristote, 350 av JC], Aristote indique la nécessité de bien distinguer le sens des mots, et précise que les termes d'une proposition (c'est-à-dire une expression) n'ont de sens qu'une fois liés entre eux. En effet, dans une « expression sans liaison », aucun de ses termes, en lui-même et par lui-même, n'affirme ni ne nie, c'est-à-dire ne sont ni vraies ni fausses. Aristote définit une liste de dix catégories, comme étant les genres les plus généraux de l'être : essence, quantifié, qualifié, relatif, quelque part, à un moment, se trouver dans une position, avoir, agir, pâtir. À cette liste s'ajoutent les opposés, les contraires, l'antérieur, le simultané et la mobilité. La métaphysique (*meta* signifie « après » ou « au-delà ») est l'étude des *Choses* après la physique, c'est-à-dire après leurs études par la physique. Elle étudie les types des Choses qu'il y a dans le monde et quelles relations ces Choses entretiennent les unes avec les autres. Les critères d'étude sont : l'existence, l'objet, la propriété (d'une chose), l'existence de Dieu, l'espace, le temps, la causalité, la possibilité. L'*ontologie* prend un tout autre sens en in-

[15] Bien que la *métaphysique* soit attribuée à Aristote, il n'est pas l'auteur de ce terme qui apparu au Moyen-Âge (scolastique médiévale).

formatique, où le terme désigne un ensemble structuré de savoirs dans un domaine de connaissance particulier. Le sens du terme ontologie s'approche alors de celui donné par Emmanuel Kant [Kant, 1997], qui critique les « Catégories » d'Aristote en définissant une ontologie comme suit : « *Ontology is a pure doctrine of elements of all our a priori cognitions* ». C'est-à-dire que l'ontologie n'est plus l'étude d'une catégorisation universelle de toutes choses, mais une conception de l'esprit portant uniquement sur un domaine donné.

Pour représenter des ontologies et des faits, le langage OWL enrichit le langage RDF(S) en fournissant un vocabulaire riche qui permet principalement de modéliser en classes, relations et individus un domaine donné. Dans le « jargon » OWL, une relation est appelée une *propriété* de par son origine RDF(S) (voir la Section 2.2.3) ou bien un *rôle* en correspondance avec les logiques de descriptions (voir la Section 2.3.4). Dans la suite de ce chapitre, nous emploierons le mot *propriété* pour parler des relations prédéfinies dans le vocabulaire RDF(S) et OWL, et nous utiliserons le mot *rôle* pour parler des autres relations, c'est-à-dire celles créées par l'utilisateur pour modéliser un domaine donné.

Le langage OWL offre une vision ensembliste de modélisation des connaissances d'un domaine. Il s'agit de regrouper des ensembles de faits au sein de structrures génériques. En effet, OWL fournit un niveau d'abstraction de modélisation permettant d'une part de structurer et regrouper en *classes* des ressources ayant des caractéristiques communes. Ces ressources, appartenant au niveau factuel d'une modélisation, sont appelées des *assertions* ou encore *instances* ou *individus*. Les classes appartiennent au niveau structurel de la modélisation. Une classe constitue donc un *type* pour un ensemble d'individus ; cet ensemble constitue à son tour l'*extension* de la classe. Notons la prise en compte de *datatypes* en OWL, qui - comme avec RDF(S) - constituent les types d'ensembles de *données*. D'autre part, OWL permet de définir des *rôles*, au niveau structurel d'une modélisation. Un rôle associe une classe, appelée le *domaine*, à une autre classe ou à un datatype, appelé(e) le *co-domaine*. Au niveau factuel d'une modélisation, une *assertion* d'un rôle constitue une association binaire entre un individu dans l'extension du domaine et individu ou une donnée dans l'extension du co-domaine du rôle.

Cette présentation du langage OWL est inspirée de celle donnée par le W3C [Dean *et al.*, 2004].

2.3.1 Connaissances structurelles d'un domaine

2.3.1.1 Classe

Une classe constitue un *type* pour un ensemble d'individus, qui partagent des caractéristiques communes. Par exemple, la classe Conducteur représente l'ensemble des individus qui conduisent un véhicule, c'est-à-dire qui ont pour caractéristique commune de conduire un véhicule. Il existe deux classes prédéfinies en OWL, à savoir les classes owl:Thing et owl:Nothing. L'extension de classe de owl:Thing est l'ensemble de tous les individus du domaine modélisé. L'extension de classe de owl:Nothing est l'ensemble vide.

Les classes d'une modélisation sont élaborées par des *descriptions de classes* ou des *axiomes de classes*. Chaque description de classe constitue la définition d'une classe. Ces

classes sont considérées comme « élémentaires », dans la mesure où elles peuvent être combinées entre elles par des axiomes de classes pour caractériser des classes plus « élaborées ». Une classe est dite *définie* ou complète, si les axiomes de la classe constituent des caractéristiques nécessaires et suffisantes pour la définition d'une classe, ou plus exactement pour l'appartenance d'un individu à l'extension de la classe. Une classe est dite *décrite* ou incomplète, si les axiomes de la classe en constituent des caractéristiques nécessaires mais non suffisantes. Notons que le terme « description de classe » peut prêter à confusion et laisser croire qu'une description de classe constitue toujours une condition nécessaire à la définition d'une classe, or ce peut être la cas ou non.

Nous présentons d'abord les descriptions de classes, et nous aborderons ensuite ce que sont les axiomes de classes.

■ Description de classe

Une *description de classe* constitue la définition d'une classe à part entière, contrairement à ce que cette désignation pourrait laisser croire [16]. On assimile aussi une description de classe à la désignation de « bloc de construction » de classe, dans la mesure où une description de classe est utilisée - généralement avec d'autres - au travers d'axiomes de classes (voir ci-après), afin de caractériser des classes considérées comme « élaborées ». C'est dans l'utilisation d'une description de classe qui est rarement utilisée seule pour caractériser une classe élaborée, que le terme *description* trouve sont origine : la description de classe ne constitue alors qu'une condition nécessaire mais non suffisante pour caractériser la classe élaborée.

Le langage OWL distingue cinq types de descriptions de classes : les restrictions de rôles - elles mêmes divisées en deux catégories, les contraintes de cardinalité et les contraintes de valeurs -, l'intersection de deux descriptions de classe ou plus, l'union de deux descriptions de classe ou plus, le complémentaire d'une description de classe, et l'énumération exhaustive des individus formant collectivement l'extension et l'intension de la classe. Ces cinq types définissent des classes en exerçant des contraintes sur les extensions de ces classes. Ces classes sont généralement *anonymes*, car elles ne sont pas identifiées. Une classe anonyme est syntaxiquement représentée par un nœud blanc (dont l'étiquette est vide).

■■ *Restriction de rôle*

Une *restriction de rôle* consiste à restreindre l'utilisation d'un rôle entre deux classes par des contraintes, ou plus exactement restreindre l'utilisation d'une assertion de rôle entre deux individus. Une restriction de rôle constitue la définition d'une classe dont l'extension est l'ensemble des individus situés au domaine du rôle et qui respectent une restriction (c'est-à-dire une contrainte) donnée.

Le langage OWL distingue deux types de restrictions de rôles : celles contraignant sa valeur et celles contraignant sa cardinalité. Une *contrainte de cardinalité* exerce une limi-

[16] Dans le monde du Web sémantique (de par sa liaison étroite avec les logiques de descriptions), le terme *description* est utilisé pour désigner une condition nécessaire à la caractérisation d'une ressource - une classe par exemple -, tandis que le terme *définition* est employé pour désigner une condition nécessaire et suffisante à cette caractérisation.

tation sur le nombre des valeurs prises par un rôle dans le contexte de cette description de classe particulière. Une *contrainte de valeur* exerce une limitation sur le co-domaine du rôle lorsqu'il s'applique à cette description de classe particulière. Le terme « valeur » peut préter à confusion ici ; il désigne en fait l'*objet* du triplet représentant la contrainte au niveau assertionnel. Il s'agira donc soit d'un individu, soit d'une donnée. Les restrictions de rôles ont la forme générale suivante :

```
<owl:Restriction>
  <owl:onProperty rdf:resource="une propriété donnée" />
  <!--une valeur ou bien une contrainte de cardinalité exactement-->
</owl:Restriction>
```

La *classe de restriction* owl:Restriction se définit comme une sous-classe de owl:Class. Une classe de restriction est un cas particulier de classe anonyme, qui doit comporter exactement un seul triplet reliant la restriction à un rôle particulier au moyen de la propriété owl:onProperty.

Dans le langage OWL, toute instance d'une classe peut avoir via des assertions d'un rôle particulier un nombre arbitraire de valeurs (individus ou données). Les contraintes de cardinalités owl:maxCardinality, owl:minCardinality ou owl:cardinality sont des propriétés qui relient une classe de restriction à une donnée du type xsd:nonNegativeInteger du Schéma XML d'un rôle donné. Une restriction avec une contrainte owl:maxCardinality (respectivement owl:minCardinality) décrit la classe de tous les individus ayant au plus (respectivement au moins) n valeurs sémantiquement [17] distinctes pour le rôle concerné, où n est la cardinalité. Une restriction avec une contrainte owl:cardinality décrit la classe de tous les individus ayant exactement n valeurs. La contrainte de cardinalité est représentée par un élément de propriété RDF avec l'attribut rdf:datatype correspondant. L'exemple suivant décrit la classe des individus ayant au plus trois prénoms :

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#prénom" />
  <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">3</owl:maxCardinality>
</owl:Restriction>
```

La contrainte de valeur owl:allValuesFrom ou owl:someValuesFrom est une propriété qui relie une classe de restriction à une description de classe ou à un datatype. On utilise une restriction avec une contrainte owl:allValuesFrom (respectivement owl:someValuesFrom) pour décrire la classe de tous les individus pour lesquels toutes les valeurs (respectivement *au moins une* valeur) du rôle sont soit des instances de la description de classe indiquée ou des données du datatype indiqué. En d'autres termes, owl:allValuesFrom définit une classe d'individus x telle que, si le couple (x, y) est une instance du rôle concerné R, alors y doit être une instance de la description de classe ou bien une valeur du datatype donné. Tandis que owl:someValuesFrom définit une classe d'individus x pour lesquels il existe au moins un y (soit une instance de la description de classe, soit une valeur du datatype donné) telle que le couple (x, y) soit une instance du rôle R. Cela n'exclut pas le fait qu'il puisse y avoir d'autres instances (x, y') de R pour lesquelles y' n'appartient pas à la description de classe ou du datatype.

[17] Le terme *sémantiquement distinct* signifie pour des individus qu'ils soient définis comme différents (*cf.* la propriété owl:differentFrom), et pour des données qu'elles soient lexicographiquement différentes.

La contrainte owl:allValuesFrom s'apparente au *quantificateur universel* de la logique des prédicats, et la contrainte owl:someValuesFrom s'apparente au *quantificateur existentiel*.

L'exemple suivant décrit la classe OWL anonyme de tous les individus pour lesquels le rôle aPourParent a seulement des valeurs de la classe Humain.

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#aPourParent" />
  <owl:allValuesFrom rdf:resource="#Humain"  />
</owl:Restriction>
```

L'exemple suivant définit la classe des individus dont au moins un parent est un pilote :

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#aPourParent" />
  <owl:someValuesFrom rdf:resource="#Pilote" />
</owl:Restriction>
```

La contrainte de valeur owl:hasValue est une propriété qui relie, par un rôle donné, une classe de restriction à une valeur v (individu ou donnée). Une restriction contenant une contrainte owl:hasValue décrit la classe de tous les individus pour lesquels le rôle concerné a au moins une valeur sémantiquement égale à la valeur v. L'exemple suivant décrit la classe des individus qui ont pour parent l'individu #claudeDurand :

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#aPourParent" />
  <owl:hasValue rdf:resource="#claudeDurand" />
</owl:Restriction>
```

Remarquons que les restrictions de rôles ont une portée locale. Par exemple, la restirction owl:allValuesFrom ne déclare pas que le rôle a toujours des valeurs de cette classe, mais juste qu'elle est vérifiée pour les individus appartenant à l'extension de classe de la classe de restriction anonyme. Il existe certaines restrictions ayant une portée globale pour la cardinalité, à savoir les propriétés owl:FunctionalProperty et owl:InverseFunctionalProperty ; elles sont présentées ci-dessous. Quant aux propriétés rdfs:domain et rdfs:range, elles précisent une portée globale sur les valeurs des domaine et co-domaine d'un rôle.

■■ *Énumération*

Une description de classe du type *énumération* se définit au moyen de la propriété owl:oneOf. L'objet de cette propriété est la liste exhaustive des individus instances de la classe. Syntaxiquement, la liste des individus est représentée par la structure RDF rdf:parseType='Collection'. L'exemple suivant définit la classe de tous les continents. La syntaxe XML/OWL [18] <owl:Thing rdf:about="..."/> désigne un individu, car tous les individus sont des instances de la classe owl:Thing.

```
<owl:Class>
  <owl:oneOf rdf:parseType="Collection">
    <owl:Thing rdf:about="#Eurasie"/>
    <owl:Thing rdf:about="#Afrique"/>
    <owl:Thing rdf:about="#Amérique"/>
    <owl:Thing rdf:about="#Australie"/>
    <owl:Thing rdf:about="#Antarctique"/>
  </owl:oneOf>
</owl:Class>
```

[18] On note XML/OWL l'utilisation de XML/RDF comme « version » de RDF pour OWL.

■■ ***Intersection, Union, Complémentarité***

Les trois types de descriptions de classes *intersection*, *union* et *complémentarité* permettent de combiner des descriptions de classe entre elles. Ils peuvent être assimilés aux opérateurs ensemblistes $\cap$, $\cup$ et $\setminus$ sur les extensions de ces descriptions de classes. Ces structures du langage OWL peuvent contenir des descriptions de classes imbriquées, que ce soit plusieurs pour l'union et l'intersection ou une seule pour la complémentarité.

La propriété owl:intersectionOf relie une classe à une liste de descriptions de classes. Une déclaration owl:intersectionOf décrit la classe anonyme dont l'extension de classe contient précisément les individus membres de l'extension de classe de toutes les descriptions de classes dans la liste. Dans l'exemple suivant, la valeur du rôle owl:intersectionOf est une liste composée de deux descriptions de classes, à savoir deux énumérations décrivant toutes deux une classe de deux individus. L'intersection résultante est une classe d'un seul individu, à savoir alphand, car c'est le seul individu commun aux deux énumérations.

```
<owl:Class>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <owl:Thing rdf:about="#serieys" />
        <owl:Thing rdf:about="#alphand" />
      </owl:oneOf>
    </owl:Class>
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <owl:Thing rdf:about="#alphand" />
        <owl:Thing rdf:about="#picard" />
      </owl:oneOf>
    </owl:Class>
  </owl:intersectionOf>
</owl:Class>
```

La propriété owl:unionOf relie une classe à une liste de descriptions de classes. Une déclaration owl:unionOf décrit une classe anonyme dont l'extension de classe contient les individus apparaissant dans au moins une des extensions de classe des descriptions de classes de la liste. La description de classe suivante définit une classe dont l'extension de classe contient trois individus, à savoir serieys, alphand et picard (en supposant qu'ils soient tous différents).

```
<owl:Class>
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <owl:Thing rdf:about="#serieys" />
        <owl:Thing rdf:about="#alphand" />
      </owl:oneOf>
    </owl:Class>
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <owl:Thing rdf:about="#alphand" />
        <owl:Thing rdf:about="#picard" />
      </owl:oneOf>
    </owl:Class>
  </owl:intersectionOf>
</owl:Class>
```

Une propriété owl:complementOf relie une classe à une seule description de classe. Une déclaration owl:complementOf décrit la classe dont l'extension de classe contient exactement

les individus n'appartenant pas à l'extension de classe de la description de classe faisant l'objet de la déclaration. L'extension de la description de classe suivante contient tous les individus n'appartenant pas à la classe Personne.

```
<owl:Class>
  <owl:complementOf>
    <owl:Class rdf:about="#Personne" />
  </owl:complementOf>
</owl:Class>
```

Notons Le langage OWL tolère de nommer les descriptions de classes de type énumération ou opérateur d'ensembles ; elles sont appelées des *descriptions de classes nommées*. En effet, nous avons vu que les descriptions de classes sont des classes anonymes. Pour nommer ces classes, c'est-à-dire pour attribuer un identifiant à chacune d'elle, il est normalement nécessaire de passer par l'axiome de classe *équivalence* (voir la section ci-dessous). Voici un exemple où la description de classe vue ci-dessus est identifiée ici par Inhumain :

```
<owl:Class rdf:about="Inhumain">
  <owl:complementOf rdf:resource="#Personne" />
</owl:Class>
```

équivaut à :

```
<owl:Class rdf:about="Inhumain">
  <owl:equivalentClass>
    <owl:Class>
      <owl:complementOf rdf:resource="#Personne" />
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

■ Axiome de classe

Les axiomes de classes permettent, en combinant des descriptions de classes, de définir partiellement ou totalement une classe.

L'axiome de classe le plus élémentaire déclare juste l'existence de la classe au moyen de l'élément syntaxique owl:Class et d'un identifiant pour cette classe. L'exemple suivant nous indique de l'existence de la classe Homme.

```
<owl:Class rdf:about="Homme" />
```

Cependant cette déclaration d'existence ne nous apprend pas grand chose de la classe Homme ; les axiomes de classes *sous-classe*, *disjonction* et *équivalence* permettent d'affiner la caractérisation des classes.

■■ *Sous-classe*

Si une classe C_1 est définie (partiellement) comme sous-classe de la classe C_2, alors l'ensemble des individus dans l'extension de C_1 devrait être un sous-ensemble de l'ensemble des individus dans l'extension de C_2. Une classe peut avoir un nombre quelconque d'axiomes rdfs:subClassOf. L'exemple suivant indique que la classe Homme est une sous-classe de la classe Personne.

```
<owl:Class rdf:about="Homme">
  <rdfs:subClassOf rdf:resource="#Personne" />
</owl:Class>
```

Les axiomes de classes peuvent devenir plus complexes lorsque les descriptions de classes s'imbriquent. Par exemple, l'axiome de classe suivant définit l'opéra italien traditionnel comme étant une sous-classe des opéras, et ayant pour genre opera seria, ou bien opera buffa (sans contrainte de cardinalité, il pourrait en fait avoir les deux valeurs).

```
<owl:Class rdf:about="OpéraItalienTraditionnel">
  <rdfs:subClassOf rdf:resource="#Opéra"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#aPourGenre"/>
      <owl:someValuesFrom>
        <owl:Class>
          <owl:oneOf rdf:parseType="Collection">
            <owl:Thing rdf:about="#OperaSeria"/>
            <owl:Thing rdf:about="#OperaBuffa"/>
          </owl:oneOf>
        </owl:Class>
      </owl:someValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

On notera que l'utilisation de rdfs:subClassOf fournit une définition partielle de la classe enfant : une classe C_1 déclarée comme sous-classe de C_2 est une « sorte-de » C_1.

■■ *Disjonction*

Le domaine et le co-domaine de la propriété owl:disjointWith sont des descriptions de classes. Chaque déclaration owl:disjointWith fait valoir que les deux descriptions de classes concernées n'ont aucun individu commun. Voici un exemple de disjonction de classe, indiquant qu'un pilote ne peut être un directeur :

```
<owl:Class rdf:about="#Pilote">
  <owl:disjointWith rdf:resource="#Directeur"/>
</owl:Class>
```

On notera que l'utilisation de owl:disjointWith pour déclarer une classe C_1 disjointe de la classe C_2 donne une définition partielle de C_1 ; l'utilisation de l'axiome owl:subClassOf y est implicite : C_2 est une sous-classe du complément de C_1.

■■ *Équivalence*

La propriété owl:equivalentClass relie une classe à une autre. Un tel axiome de classe signifie que les deux classes concernées possèdent la même extension de classe. Un axiome de classe peut contenir plusieurs déclarations owl:equivalentClass. L'exemple suivant indique que la classe Personne est équivalente à la classe Humain.

```
<owl:Class rdf:about="#Personne">
  <equivalentClass rdf:resource="#Humain"/>
</owl:Class>
```

On rappelle que l'utilisation de owl:intersectionOf, owl:unionOf, owl:complementOf et owl:oneOf définissent les conditions nécessaires et suffisantes pour établir l'appartenance à une classe ; et n'ont donc pas besoin de l'utilisation de l'axiome owl:equivalentClass qui est dans ces quatre cas-là implicite.

2.3.1.2 Datatype

Le langage XML/OWL utilise le système XML/RDF de typage de données, qui fournit un mécanisme pour utiliser les datatypes du Schéma XML (voir Section 2.2.3.1). Ainsi tout datatype est une sous-classe de la classe rdf:XMLLiteral. Une donnée peut ne pas être typée, dans ce cas elle est considérée comme un littéral, c'est-à-dire une instance de la classe rdfs:Literal. Cette classe prédéfinie RDFS est une classe parent de la classe rdf:XMLLiteral.

La classe rdf:XMLLiteral et ses sous-classes sont des instances de la classe rdfs:Datatypes.

■ Autres Datatypes

Il est possible de définir d'autres datatypes que ceux fournis par RDF. Pour cela, le nouveau datatype doit être une instance de la classe prédéfinie RDFS rdfs:Datatype. Par exemple, le type entierPair peut être déclaré de la manière suivante.

```
<rdfs:Datatype rdf:about="entierPair" />
```

Il peut ensuite être décrit comme un sous-type du datatype xsd:integer :

```
<rdf:Description rdf:about="entierPair">
  <rdfs:subClassOf rdf:resource="&xsd;integer" />
</rdf:Description>
```

■ Datatype énuméré

Outre les types de données RDF, le langage OWL fournit une autre structure pour définir un type de donnée *énuméré*. Ce format de type de donnée, ennXML/OWL, utilise la structure owl:oneOf (la même qui sert aussi à décrire une classe énumérée). Pour un datatype énuméré, le sujet de la propriété owl:oneOf est un nœud blanc de la classe owl:DataRange et son objet une liste de littéraux. Notons, qu'il n'est pas possible d'utiliser l'attribut et la valeur rdf:parseType='Collection' pour définir la liste des littéraux, car XML/RDF impose à une telle collection d'être une rdf:List dont les nœuds sont formés par enchaînement à l'aide des éléments RDF rdf:first, rdf:rest et rdf:nil. Ci-dessous est présenté un exemple des trois données possibles d'un datatype énuméré représentant les valeurs possibles que peut prendre un feu tricolore. Une liste exhaustive de données d'un datatype énuméré (anonyme ci-dessous) est définie en XML/OWL dans une balise <owl:DataRange> car ces données se trouveront nécessairement dans le co-domaine d'un rôle (voir la section suivante pour les termes *co-domaine* et *rôle*).

```
<owl:DataRange>
  <owl:oneOf>
    <rdf:List>
      <rdf:first rdf:datatype="&xsd;string"> vert </rdf:first>
      <rdf:rest>
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;string"> orange </rdf:first>
          <rdf:rest>
            <rdf:List>
              <rdf:first rdf:datatype="&xsd;string"> rouge </rdf:first>
              <rdf:rest rdf:resource="&rdf;nil" />
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </rdf:rest>
    </rdf:List>
  </owl:oneOf>
</owl:DataRange>
```

2.3.1.3 Rôle

Un *rôle* permet de mettre en relation une classe avec une autre classe ou bien une classe avec un datatype. Rappelons qu'en RDF(S) un rôle est appelé une *propriété*, mais nous réservons cette appelation pour les rôles prédéfinis par les langages RDF(S) et OWL.

Le langage OWL distingue deux catégories de rôles :

- Un *rôle d'objet* (ou *object property*) relie une classe à une autre ; ainsi une assertion d'un rôle d'objet liera un individu à un autre individu. Un rôle d'objet est de type owl:ObjectProperty.
- Un *rôle de type de donnée* (ou *Datatype property*) relie une classe à un datatype ; ainsi une assertion de rôle de type de donnée liera un individu à une (valeur de) donnée. Un rôle de type de donnée est de type owl:DatatypeProperty.

■ Domaine et co-domaine

Les rôles sont binaires en OWL et ont une direction d'application, d'un *domaine* (*domain* en anglais) à un *co-domaine* (ou *image*, ou *range* en anglais). Le domaine et le co-domaine ont une portée globale [19] dans l'utilisation du rôle. Par défaut, le domaine et le co-domaine d'un rôle sont la classe owl:Thing.

La propriété rdfs:domain relie un rôle à une description de classe. Cette propriété fait valoir que les *sujets* de la propriété doivent appartenir à l'extension de la classe indiquée. L'utilisation multiple de rdfs:domain est permise, et est interprétée comme une intersection des extensions de classe des classes impliquées.

La propriété rdfs:range relie un rôle soit à une description de classe soit à un datatype (généralement un datatype définit dans le Schéma XML). Cette propriété fait valoir que les *objets* d'une telle déclaration de rôle doivent appartenir à l'extension de la classe indiquée ou à une valeur du datatype. L'utilisation multiple de rdfs:range est permise, et est interprétée comme une intersection des extensions de classe des classes impliquées ou une intersection des données (s'il s'agit de nombres, de dates ou de durées).

L'exemple suivant introduit le rôle d'objet conduire_quad, dont ses assertions auront pour domaine un individu de la classe Pilote et pour co-domaine une instance à la fois de la classe Auto et Moto.

```
<owl:ObjectProperty rdf:about = "#conduire_quad">
  <rdfs:domain rdf:resource="#Pilote"/>
  <rdfs:range rdf:resource="#Auto"/>
  <rdfs:range rdf:resource="#Moto"/>
</owl:ObjectProperty>
```

Dans ce deuxième exemple, le rôle de type de donnée nom a pour co-domaine le datatype xsd:string constituant le type des chaînes de caractères. Le domaine de ce rôle n'étant pas renseigné, il s'agit donc par défaut de la classe Thing. Ainsi, sauf restriction de rôle éventuelle, les assertions de ce rôle nom pourront lier tout individu à une chaîne de caractères.

```
<owl:DatatypeProperty rdf:about = "#nom">
  <rdfs:range rdf:datatype="&xsd;string"/>
</owl:DatatypeProperty>
```

[19] Par opposition aux restrictions de rôles par *contraintes de valeurs*, vues en Section 2.3.1.1.

■ **Transitivité, symétrie et inverse**

Les propriétés owl:TransitiveProperty, owl:SymmetricProperty et owl:inverseOf sont applicables sur des rôles d'objets (domaine et co-domaine sont des classes, pas de datatype).

La structure owl:TransitiveProperty intégrée à OWL permet de spécifier qu'un rôle est transitif. Cela signifie que si le couple (x, y) et (y, z) sont des assertions d'un rôle R transitif, alors le couple (x, z) l'est aussi. Par exemple, pour décrire que le rôle aPourDescendance est transitif la syntaxe XML/OWL est la suivante :

```
<owl:ObjectProperty rdf:about="aPourDescendance">
  <rdf:type    rdf:resource="&owl;TransitiveProperty" />
  <rdfs:domain rdf:resource="#Humain"/>
  <rdfs:range  rdf:resource="#Humain"/>
</owl:ObjectProperty>
```

ou bien de façon raccourcie :

```
<owl:TransitiveProperty rdf:about="aPourDescendance">
  <rdfs:domain rdf:resource="#Humain"/>
  <rdfs:range  rdf:resource="#Humain"/>
</owl:TransitiveProperty>
```

La structure owl:SymmetricProperty intégrée à OWL permet de spécifier qu'un rôle est symétrique. Cela signifie que si le couple (x, y) est une assertion d'un rôle R transitif, alors le couple (y, x) l'est aussi. Par exemple, pour décrire que le rôle estAmiDe est symétrique la syntaxe XML/OWL est la suivante :

```
<owl:SymmetricProperty rdf:about="estAmiDe">
  <rdfs:domain rdf:resource="#Humain"/>
  <rdfs:range  rdf:resource="#Humain"/>
</owl:SymmetricProperty>
```

La propriété owl:inverseOf permet de spécifier un rôle est inverse d'un autre. Ainsi, un rôle R_1 déclaré comme inverse du rôle R_2 fait valoir que pour chaque couple (x, y) dans l'extension de propriété de la propriété R_1, il existe un couple (y, x) dans l'extension de propriété de R_2, et inversemement. Notons qu'avec l'utilisation de la propriété owl:inverseOf, R_2 constitue une conditon nécessaire et suffisante pour définir R_1.

```
<owl:ObjectProperty rdf:ID="aPourEnfant">
  <owl:inverseOf rdf:resource="#aPourParent"/>
</owl:ObjectProperty>
```

■ **Cardinalité globale sur un rôle**

Les propriétés owl:FunctionalProperty et owl:InverseFunctionalProperty permettent détablir des contraintes de cardinalités globales, agissant respectivement sur le domaine et sur le co-domaine du rôle. C'est-à-dire, une contrainte de cardinalité qui est vérifiée quelle que soit la classe sur laquelle le rôle s'applique, en accord avec le domaine du rôle toutefois. C'est une différence par rapport aux contraintes de cardinalités contenues dans les restrictions de rôles (mise à part qu'une restriction de rôle concerne la mise en œuvre d'une classe, alors qu'ici il s'agit d'un rôle) : une restriction de rôle constitue une contrainte qui n'a qu'une portée locale vis-à-vis des individus dans l'extension de la description de classe.

Un rôle *fonctionnel* (owl:FunctionalProperty) permet de spécifier que pour un individu x du type du domaine du rôle ne peut être le sujet, au plus, que d'une unique assertion du rôle. En d'autres termes, il ne peut y avoir de valeurs (individus ou données) distinctes

y_1 et y_2 telles que les couples (x, y_1) et (x, y_2) soient tous deux des assertions de ce rôle. L'exemple suivant affirme que le rôle de type de donnée ci-dessous est fonctionnel, c'est-à-dire qu'un individu (le domaine est par défaut Thing) ne peut avoir qu'un nom au plus :

```
<owl:DatatypeProperty rdf:about = "#nom">
  <rdf:type   rdf:resource="&owl;FunctionalProperty" />
  <rdfs:range rdf:datatype="&xsd;string"/>
</owl:DatatypeProperty>
```

Un rôle *fonctionnel inverse* (owl:InverseFunctionalProperty) permet, à partir d'un rôle donné, de spécifier que pour un individu y du type du co-domaine du rôle ne peut être l'objet que d'une unique assertion du rôle. Ainsi, il ne peut y avoir d'individus distincts x_1 et x_2 tels que les couples (x_1, y) et (x_2, y) soient tous deux des assertions de ce rôle. On notera que cette caractéristique de fonctionalité inverse ne s'applique que sur des rôles d'objets. Notons que tel que nous avons présenté un rôle en OWL, reliant soit deux classes, soit une classe à un datatype, un rôle symétrique ne peut donc que s'appliquer sur des rôles d'objets. Nous verrons en fin de ce chapitre, que le sous-langage *OWL Full* autorise le modélisation de rôles de types de données symétriques. Prenons l'exemple où à partir de l'objet d'une assertion estLePèreDe il est possible d'identifier de manière unique le père (une instance de Homme) :

```
<owl:ObjectProperty rdf:about="#estLePèreDe">
  <rdf:type    rdf:resource="&owl;InverseFunctionalProperty" />
  <rdfs:domain rdf:resource="#Homme"/>
  <rdfs:range  rdf:resource="#Personne"/>
</owl:ObjectProperty>
```

■ Axiome de rôle

■■ *Sous-rôle*

La notion d'héritage entre rôles est exprimée par la propriété rdfs:subPropertyOf. Sa signification dans OWL est la même que celle dans RDFS (voir Section 2.2.3) : si un rôle R_1 est un sous-rôle de R_2 alors l'extension de propriété de R_1 (un ensemble de couples) devrait être un sous-ensemble de l'extension de propriété de R_2 (un ensemble de couples aussi). Par exemple, le rôle piloter est un sous-rôle (une sorte-de) de rôle conduire :

```
<owl:ObjectProperty rdf:about="piloter">
  <rdfs:subPropertyOf rdf:resource="#conduire" />
</owl:ObjectProperty>
```

Les axiomes de sous-rôle peuvent s'appliquer aux rôles d'objets comme aux rôles de types de données, même si ce dernier cas est généralement moins employé puisque souvent seul les datatypes de RDFS sont utilisés.

■■ *Équivalence*

L'équivalence entre rôles se définit en OWL par la propriété rdfs:equivalentProperty. Elle permet de déclarer que deux rôles ont la même extension.

Notons que l'utilisation de la propriété owl:inverseOf pour déclarer deux rôles inverses, comme avec le triplet $< R_1$,owl:inverseOf,$R_2 >$, constitue une condition nécessaire et suffisante pour définir le rôle inverse, ici R_2.

2.3.2 Connaissances factuelles d'un domaine

Les individus (ou instances de classes), les assertions de rôles et leurs valeurs constituent les connaissances factuelles d'une modélisation d'un domaine donné.

2.3.2.1 Individu

Un individu est l'intance d'une ou plusieurs classes, qui en sont ses *types*.

En XML/OWL, pour déclarer qu'un individu a pour type une classe donnée, une balise ayant pour nom la classe en question est utilisée. Le nom de l'individu est renseigné à l'aide d'un attribut rdf:about (ou rdf:ID). Par exemple, définir un individu de type Directeur et identifié par serieys, cela s'écrit :

```
<Directeur rdf:about="#serieys" />
```

D'une façon plus générale, cette même déclaration peut s'écrire de la façon suivante, car tout individu est nécessairement du type le plus général owl:Thing auquel on précise un type plus spécifique.

```
<owl:Thing rdf:about="#serieys">
  <rdf:type rdf:resource="#Directeur" />
</owl:Thing>
```

■ Identité de l'individu

Le langage OWL ne fait pas l'hypothèse du nom unique - Unique Name Assomption - (UNA), ainsi des URIs différentes peuvent identifier ou non le même individu. Le langage OWL dispose de trois propriétés pour différencier ou non les individus :

- La propriété owl:sameAs sert à déclarer que deux identifiants (deux URIs) se rapportent au même individu.
- Inversement, la propriété owl:differentFrom sert à déclarer que deux identifiants ne font pas référence au même individu.
- La structure owl:AllDifferent couplée à la propriété owl:distinctMembers constituent un raccourci d'écriture de la propriété owl:differentFrom pour différencier deux à deux les éléments d'un ensemble d'individus.

L'exemple suivant indique que les individus pierre et pierrot font référence à un même individu, mais qu'il ne s'agit pas de l'individu pilou. La seconde partie de cet exemple stipule que les individus serieys, alphand et picard sont deux à deux différents.

```
<owl:Thing rdf:about="pierre">
  <owl:sameAs rdf:resource="#pierrot" />
  <owl:differentFrom rdf:resource="#pilou" />
</owl:Thing>

<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <Thing rdf:about="#serieys" />
    <Thing rdf:about="#alphand" />
    <Thing rdf:about="#picard" />
  </owl:distinctMembers>
</owl:AllDifferent>
```

Il est possible qu'un individu ne soit pas identifié (absence d'attribut rdf:about ou rdf:ID) ; il est dit dans ce cas *anonyme*. Un individu anonyme constitue généralement une *valeur de rôle*, c'est-à-dire le second argument d'une assertion de rôle (voir ci-après).

2.3.2.2 Assertion de rôle et valeur de rôle

Un individu peut être lié à d'autres individus ou à des données par des assertions de rôles. Dans ce cas, l'individu est nécessairement dans l'extension du domaine de chacun de ces rôles (rôle d'objets ou rôle de types de données) ; c'est-à-dire qu'il est le premier argument des assertions des rôles. De plus les différentes contraintes liées à la définition de la (des) classe(s) dont l'individu est l'instance (contrainte de valeur par exemple) ou liées aux rôles concernées (cardinalité globale par exemple) doivent être respectées.

Le second argument d'une assertion d'un rôle est appelé *valeur de rôle* (ou valeur de propriété, suivant le vocabulaire RDF/OWL). Une valeur de rôle est un individu ou une donnée, selon que l'assertion de rôle employée ait pour type un rôle d'objets ou un rôle de types de données. Les différentes valeurs de rôles liées à un même individu, par un ensemble d'assertions de rôles dont il est le premier argument, contribuent à caractériser cet individu.

L'exemple suivant déclare l'individu serieys comme étant un Directeur, qui a pour nom Serieys, pour prénom Dominique et qui dirige l'équipe #TRMR. C'est-à-dire que l'individu serieys est lié par une assertion du rôle de types de données nom à la chaîne de caractères Serieys, par une assertion du rôle de types de données prénom à la chaîne de caractères Dominique et par une assertion du rôle d'objets diriger à l'individu #TRMR. Cette équipe possède aussi un nom dont la valeur est Team Repsol Mitsubishi Ralliart.

```
<Directeur rdf:about="serieys">
  <nom rdf:datatype="&xsd;string"> Serieys </nom>
  <prénom rdf:datatype="&xsd;string"> Dominique </prénom>
  <diriger rdf:resource="#TRMR" />
</Directeur>

<Équipe rdf:about="TRMR">
  <nom rdf:datatype="&xsd;string"> Team Repsol Mitsubishi Ralliart </nom>
</Équipe>
```

Comme dit précédement, un individu n'est pas nécessairement nommé. Syntaxiquement, lors de sa déclaration comme instance d'une classe, aucun identifiant n'est associé à l'individu anonyme. Reprenons l'exemple précédent, mais cette fois-ci l'équipe dirigée par serieys est une instance anonyme de la classe Équipe.

```
<Directeur rdf:about="serieys">
  <nom rdf:datatype="&xsd;string"> Serieys </nom>
  <prénom rdf:datatype="&xsd;string"> Dominique </prénom>
  <diriger>
    <Équipe>
      <nom rdf:datatype="&xsd;string"> Team Repsol Mitsubishi Ralliart </nom>
    </Équipe>
  <diriger>
</Directeur>
```

2.3.3 Exemple de document OWL

Les Figures 26, 27 et 28 constituent un exemple d'une modélisation OWL, reprenant et complétant la modélisation en Figure 24 ou 25 mais aussi celle en Section 1.1 (en Figures 3, 4 et 5) avec le modèle des GCs. Les deux premières figures de cet exemple modélisent les connaissances structurelles du domaine, avec respectivement la modélisation de classes et de rôles. La troisième figure contient la modélisation des faits.

Dans un souci de lisibilité des trois fichiers XML/OWL constituant cet exemple, l'entête contenant la version XML ainsi que l'encodage utilisés, un ensemble de constantes définies par DTD, et la balise racine <rdf:RDF> déclarant les espaces de nommages utilisés y ont été omis. Il s'agit des lignes suivantes :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE raccourcis [
    <!ENTITY xsd  "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY rdf  "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY owl  "http://www.w3.org/2002/07/owl#" >
    <!ENTITY ex   "http://exemple.fr#" >
  ]>
<rdf:RDF
  xmlns:owl="&owl;" xmlns:rdfs="&rdfs;" xmlns:rdf="&rdf;" xmlns:xsd="&xsd;"
  xmlns="&ex;" xml:base="http://exemple.fr#">

  <!-- ici le contenu XML/OWL -->
</rdf:RDF>
```

Le langage OWL étant plus expressif que RDF(S) ou que le modèle des GCs, les connaissances d'un domaine peuvent donc être modélisées avec plus de finesse. Par exemple, par rapport à la modélisation en Figures 24 et 25 ou celle en Figures 3, 4 et 5, ici il est précisé qu'une Personne possède entre un et trois prénom(s), qu'une Équipe est composée de Membres tandis qu'un Rallye est composé d'Étapes, ou encore que si une instance possède un nom c'est au nombre de zéro ou un (le rôle nom est owl:FunctionalProperty).

2.3.4 Logiques de descriptions

le langage OWL repose sur la syntaxe RDF [Klyne et Carroll, 2004], étend le vocabulaire de RDFS [Brickley et Guha, 2004], et possède une sémantique logique issue des logiques de descriptions [Patel-Schneider *et al.*, 2004]. OWL étant une quasi-réécriture de certaines logiques de descriptions, nous commençons par en donner une courte introduction.

Les logiques de descriptions forment une famille de modèles, structurés et formels, pour représenter des connaissances et raisonner sur ces connaissances représentées [Baader *et al.*, 2003]. Ces logiques ont été conçues à partir des réseaux sémantiques [Lehmann, 1992] et des frames [Minsky, 1980]. Une solide introduction aux logiques de descriptions peut être trouvée dans [Napoli, 1997].

Concrètement, les connaissances d'un domaine sont divisées en deux niveaux. D'un côté la *terminologie*, appelée T-Box, qui « décrit » des concepts et des rôles. Elle spécifie les connaissances structurelles. De l'autre, le niveau *assertionnel*, appelé A-Box, qui constitue les connaissances factuelles par un ensemble d'assertions. On appellera une *base de connaissances*, notée $\mathcal{K}$, un couple (T-Box, A-Box). Une sémantique est associée à $\mathcal{K}$ par une *interprétation*.

2.3.4.1 Connaissances terminologiques

Définition 2.1 (Terminologie) *Une terminologie, ou T-Box, est l'ensemble des formules qui décrivent les concepts et les rôles. L'introduction d'un concept se fait soit par une description*

```
<!--En-tete : encodage et version XML, DTD et espaces de nommages.-->

<owl:Class rdf:about = "Personne">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#prénom" />
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
        1 </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#prénom" />
      <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">
        3 </owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about = "Membre">
  <rdfs:subClassOf> <owl:Restriction>
      <owl:onProperty rdf:resource="#être_composé" />
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
        1 </owl:minCardinality>
  </owl:Restriction> </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about = "Directeur">
  <rdfs:subClassOf rdf:resource = "#Membre" />
  <rdfs:subClassOf> <owl:Restriction>
      <owl:onProperty rdf:resource="#diriger" />
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
        1 </owl:minCardinality>
  </owl:Restriction> </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about = "Pilote">
  <rdfs:subClassOf rdf:resource = "#Membre" />
  <rdfs:subClassOf> <owl:Restriction>
      <owl:onProperty rdf:resource="#piloter" />
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
        1 </owl:minCardinality>
  </owl:Restriction> </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about = "Équipe">
  <rdfs:subClassOf> <owl:Restriction>
      <owl:onProperty rdf:resource="#être_composé" />
      <owl:allValuesFrom rdf:resource="#Membre" />
  </owl:Restriction> </rdfs:subClassOf>
  <rdfs:subClassOf> <owl:Restriction>
      <owl:onProperty rdf:resource="#être_composé" />
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
        2 </owl:minCardinality>
  </owl:Restriction> </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about = "Rallye" />
  <rdfs:subClassOf> <owl:Restriction>
      <owl:onProperty rdf:resource="#être_composé" />
      <owl:allValuesFrom rdf:resource="#Étape" />
  </owl:Restriction> </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about = "Étape" />
<owl:Class rdf:about = "Auto">
  <rdfs:subClassOf rdf:resource = "Véhicule" />
</owl:Class>
<owl:Class rdf:about = "Moto">
  <rdfs:subClassOf rdf:resource = "#Véhicule" />
</owl:Class>
```


Figure 26 – Ontologie en OWL (ici les classes), reprenant et complétant la taxonomie en Figure 3 ; suite de l'ontologie en Figure 27.

```
<!-- En-tête : encodage et version XML,
               doctrine DTD et
               espaces de nommages. -->

<owl:DatatypeProperty rdf:about = "nom">
  <rdf:type rdf:resource="&owl;FunctionalProperty" />
  <rdfs:range rdf:resource = "&xsd;string" />
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about = "prénom">
  <rdfs:domain rdf:resource = "#Personne" />
  <rdfs:range  rdf:resource = "&xsd;string" />
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about = "âge">
  <rdfs:domain rdf:resource = "#Personne" />
  <rdfs:range  rdf:resource = "&xsd;integer" />
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about = "numéro_licence">
  <rdfs:domain rdf:resource = "#Membre" />
  <rdfs:range  rdf:resource = "&xsd;integer" />
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about = "nbr_km">
  <rdfs:domain rdf:resource = "#Étape" />
  <rdfs:range  rdf:resource = "&xsd;integer" />
</owl:DatatypeProperty>


<owl:ObjectProperty rdf:about = "composer" />
<owl:ObjectProperty rdf:about = "être_composé">
  <owl:inverseOf rdf:resource="#composer" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about = "diriger">
  <rdf:type rdf:resource="&owl;FunctionalProperty" />
  <rdfs:domain rdf:resource = "#Directeur" />
  <rdfs:range  rdf:resource = "#Équipe" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about = "être_dirigé">
  <owl:inverseOf rdf:resource="#diriger" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about = "participer">
  <rdfs:domain rdf:resource = "#Équipe" />
  <rdfs:range  rdf:resource = "#Rallye" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about = "conduire">
  <rdfs:domain rdf:resource = "#Personne" />
  <rdfs:range  rdf:resource = "#Véhicule" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about = "piloter">
  <rdfs:subPropertyOf rdf:resource = "#conduire" />
  <rdfs:domain rdf:resource = "#Pilote" />
</owl:ObjectProperty>
```

Figure 27 – Ontologie en OWL (ici les rôles), reprenant et complétant la taxonomie en Figure 4 ; suite de l'ontologie en Figure 26.

```
<!-- En-tête : encodage et version XML,
               doctrine DTD et
               espaces de nommages. -->

<Équipe rdf:about = "TMRM">
  <nom rdf:datatype="&xsd;string">Team Repsol Mitsubishi Ralliart</nom>
  <être_composé rdf:resource="#serieys" />
  <être_composé rdf:resource="#alphand" />
  <participer rdf:resource="#dakar2007" />
</Équipe>
<Directeur rdf:about = "serieys">
  <nom rdf:datatype="&xsd;string">Serieys</nom>
  <prénom rdf:datatype="&xsd;string">Dominique</prénom>
  <âge rdf:datatype="&xsd;integer"> 42 </âge>
  <numéro_licence rdf:datatype="&xsd;integer">13</numéro_licence>
  <diriger rdf:resource="#TMRM" />
</Directeur>
<Membre rdf:about = "alphand">
  <nom rdf:datatype="&xsd;string">Alphand</nom>
  <prénom rdf:datatype="&xsd;string">Luc</prénom>
  <âge rdf:datatype="&xsd;integer"> 38 </âge>
  <numéro_licence rdf:datatype="&xsd;integer">33</numéro_licence>
  <piloter>
    <Auto /> <!--instance anonyme-->
  </piloter>
</Membre>
<Rallye rdf:about = "dakar2007">
  <être_composé>
    <Étape rdf:about = "AtârTichit">
      <nbr_km rdf:datatype="&xsd;integer"> 589 </âge>
    </Étape>
  </être_composé>
  <être_composé>
    <Étape rdf:about = "TichitNéma">
      <nbr_km rdf:datatype="&xsd;integer"> 494 </âge>
    </Étape>
  </être_composé>
</Rallye>
```

Figure 28 – Connaissances factuelles en OWL, basées sur l'ontologie en Figures 26 et 27, et reprenant les faits en Figure 5.

($\sqsubseteq$), on parle de concept primitif, *soit par une définition ($\equiv$), on parle de* concept défini. *Un concept primitif est introduit en tant qu'une dérivation d'un concept ou d'une construction de plusieurs concepts, qui en est son subsumant. Il existe un concept de plus haut niveau, noté $\top$, et un de plus bas niveau, noté $\bot$. Une construction se fait au moyen de* constructeurs *qui agissent comme des opérateurs de combinaison de concepts, suivant la syntaxe donnée au Tableau 4. Un concept défini est introduit par une construction de concepts qui le définit. De même que pour les concepts, un rôle peut être introduit dans la terminologie par définition ou par description, avec des constructeurs propres aux rôles et* toprole *comme rôle de plus haut niveau.*

La signification d'une description se traduit par « sorte-de », et ordonne en hiérarchie les concepts et en hiérarchie les rôles.

Remarquons que les notions de *domaine* et de *co-domaine* dans la caractérisation d'un rôle ne font pas explicitement partie des logiques de descriptions. Cependant, l'utilisation d'un quantificateur universel $\forall r.C$ ou existence $\exists r.C$ dans la construction d'un concept provoque le typage implicite du co-domaine du rôle r par C. Ainsi, [Buchheit *et al.*, 1993] propose de préciser dans l'absolu le domaine et co-domaine d'un rôle, sans les « découvrir » à travers les descriptions et définitions de concepts. Pour cela, le co-domaine C d'un rôle r sera explicitement connu par la description $\top \sqsubseteq \forall r.C$, et son domaine D par la description $\top \sqsubseteq \forall r^{-}.D$. Cette dernière description exige que la logique de description utilisée accepte le constructeur de *rôle inverse*. À défaut, la description suivante pourra être utilisée $\exists r.\top \sqsubseteq D$. Cependant, bien que le domaine du rôle soit renseigné, il est en plus assujetti à l'exigence qu'il existe au moins une assertion du rôle r portée par l'existentiel ($\exists$).

■ **Exemple**

La T-Box en Figure 29 reprend d'une part la hiérarchie des concepts et la hiérarchie des relations (ici les rôles) en Figures 3 et 4, et d'autre part complète la spécification de ce niveau structurel. Par exemple, le concept Moto est décrit comme étant disjoint de Auto ; le concept Directeur est défini comme une spécialisation de Membre qui dirige au moins une (*cf.* quantificateur existentiel) Équipe ; le rôle composer est transitif ; le rôle être_dirigé est l'inverse de diriger. Deux remarques peuvent être faites.

La première est que d'après les descriptions et définitions des concepts, on peut déduire les domaines et co-domaines des rôles diriger, être_dirigé, piloter et participer. Par exemple en ligne (t13), diriger a pour domaine Directeur et pour co-domaine Équipe. Par contre, le domaine et co-domaine de composer n'ont qu'une portée locale, car (i) ce rôle est employé de deux manières différentes aux lignes (t12) pour lier Membre et Équipe et (t15) pour lier Étape et Rallye, et (ii) il n'existe ni de lien hiérarchique entre Membre et Équipe ni entre Étape et Rallye. Le domaine et le co-domaine de conduire ne peuvent être déduits. La description suivante aurait pu être ajoutée : Personne $\sqsubseteq \forall$conduire.Véhicule.

La seconde remarque est qu'il n'y a pas qu'une façon de spécifier une T-Box. Par exemple, la ligne (t4) peut être remplacée par les deux descriptions : Moto $\sqsubseteq$ Véhicule puis Auto $\sqcap$ Moto $\sqsubseteq \bot$. En l'absence d'expressivité qualifiée pour la restriction de nombre ($\mathcal{Q}$), l'expression ($\geq$ 2 composer^{-}.Membre) peut être de façon quasi-équivalente remplacée par

Constructeur (lettre associée)		**Syntaxe**	**Sémantique**
		$\top$	$\top^{\mathcal{I}} = \Delta_{\mathcal{I}}$
		$\bot$	$\bot^{\mathcal{I}} = \emptyset$
	hiérarchie de concepts	$C_1 \sqsubseteq C_2$	$(C_1 \sqsubseteq C_2)^{\mathcal{I}} = \{x \in \Delta_{\mathcal{I}} \mid x \in C_1^{\mathcal{I}} \rightarrow x \in C_2^{\mathcal{I}}\}$
	conjonction	$C_1 \sqcap C_2$	$(C_1 \sqcap C_2)^{\mathcal{I}} = \{x \in \Delta_{\mathcal{I}} \mid x \in C_1^{\mathcal{I}} \wedge x \in C_2^{\mathcal{I}}\}$
	quantificateur universel	$\forall r.C$	$(\forall r.C)^{\mathcal{I}} = \{x \in \Delta_{\mathcal{I}} \mid \forall y \in \Delta_{\mathcal{I}}, (x,y) \in r^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
$\mathcal{E}$	quantificateur existentiel	$\exists r.C$	$(\exists r.C)^{\mathcal{I}} = \{x \in \Delta_{\mathcal{I}} \mid \exists y \in \Delta_{\mathcal{I}}, (x,y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
$\mathcal{C}$	négation	$\neg C$	$(\neg C)^{\mathcal{I}} = \{x \in \Delta_{\mathcal{I}} \mid x \notin C^{\mathcal{I}}\}$
$\mathcal{U}$	disjonction	$C_1 \sqcup C_2$	$(C_1 \sqcup C_2)^{\mathcal{I}} = \{x \in \Delta_{\mathcal{I}} \mid x \in C_1^{\mathcal{I}} \vee x \in C_2^{\mathcal{I}}\}$
$\mathcal{N}$	restriction de nombre	$\geq n\ r$	$(\geq n\ r)^{\mathcal{I}} = \{x \in \Delta_{\mathcal{I}} \mid \text{card}(\{y \in \Delta_{\mathcal{I}} \mid (x,y) \in r^{\mathcal{I}}\}) \geq n\}$
	(au-plus et au-moins)	$\leq n\ r$	$(\leq n\ r)^{\mathcal{I}} = \{x \in \Delta_{\mathcal{I}} \mid \text{card}(\{y \in \Delta_{\mathcal{I}} \mid (x,y) \in r^{\mathcal{I}}\}) \leq n\}$
$\mathcal{Q}$	restriction de nombre	$\geq n\ r.C$	$(\geq n\ r.C)^{\mathcal{I}} = \{x \in \Delta_{\mathcal{I}} \mid \text{card}(\{y \in \Delta_{\mathcal{I}} \mid (x,y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}) \geq n\}$
	qualifié	$\leq n\ r.C$	$(\leq n\ r.C)^{\mathcal{I}} = \{x \in \Delta_{\mathcal{I}} \mid \text{card}(\{y \in \Delta_{\mathcal{I}} \mid (x,y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}) \leq n\}$
$\mathcal{O}$	type énuméré	$\{a_1, ..., a_n\}$	$\{a_1, ..., a_n\}^{\mathcal{I}} = \{a_1^{\mathcal{I}}, ..., a_n^{\mathcal{I}}\}$
$\mathcal{H}$	*hiérarchie de rôles*	$r_1 \sqsubseteq r_2$	$(r_1 \sqsubseteq r_2)^{\mathcal{I}} = \{(x,y) \in \Delta_{\mathcal{I}} \times \Delta_{\mathcal{I}} \mid (x,y) \in r_1^{\mathcal{I}} \rightarrow (x,y) \in r_2^{\mathcal{I}}\}$
$\mathcal{R}$	conjonction de rôles	$r_1 \sqcap r_2$	$(r_1 \sqcap r_2)^{\mathcal{I}} = \{(x,y) \in \Delta_{\mathcal{I}} \times \Delta_{\mathcal{I}} \mid (x,y) \in r_1^{\mathcal{I}} \wedge (x,y) \in r_2^{\mathcal{I}}\}$
$\mathcal{I}$	rôle inverse	r^-	$(r^-)^{\mathcal{I}} = \{(y,x) \in \Delta_{\mathcal{I}} \times \Delta_{\mathcal{I}} \mid (x,y) \in r^{\mathcal{I}}\} = (r^{\mathcal{I}})^-$
$\mathcal{R}^+$	transitivité des rôles	r^+	$(r^+)^{\mathcal{I}} = \{(x,z) \in \Delta_{\mathcal{I}} \times \Delta_{\mathcal{I}} \mid \forall y \in \Delta_{\mathcal{I}}, (x,y) \in r^{\mathcal{I}} \wedge (y,z) \in r^{\mathcal{I}} \rightarrow (x,z) \in r^{\mathcal{I}}\}$

C, C_1 et C_2 sont des noms de concepts, r, r_1 et r_2 sont des noms de rôles, et a_1 à a_n sont des individus.

Tableau 4 – Les constructeurs du langage $\mathcal{AL}$ et de ses extensions les plus courantes des logiques de descriptions

$(\geq 2\ \text{composer}^-) \sqcap \exists \text{composer}^-.\text{Membre}$ (d'expressivité moindre $\mathcal{N}$) où seulement un élément dans le co-domaine du rôle composer^- est contraint à être de type Membre.

2.3.4.2 Connaissances assertionnelles

Définition 2.2 (A-Box) *Une A-Box est l'ensemble des formules qui décrivent les* assertions de concepts, *appelés individus, et les* assertions de rôles, *suivant la syntaxe donnée au Tableau 5.*

Assertion	**Syntaxe**	**Sémantique**
inclusion d'assertion de concept (d'individu)	$C(a)$	$C(a)^{\mathcal{I}} = a^{\mathcal{I}} \in C^{\mathcal{I}}$
inclusion d'assertion de rôle	$r(a,b)$	$r(a,b)^{\mathcal{I}} = (a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$
égalité d'individus	$a = b$	$(a{=}b)^{\mathcal{I}} = (a^{\mathcal{I}}{=}b^{\mathcal{I}})$
inégalité d'individus	$a \neq b$	$(a{\neq}b)^{\mathcal{I}} = (a^{\mathcal{I}}{\neq}b^{\mathcal{I}})$

C est un nom de concept, r est un nom de rôle, et a ainsi que b identifient des individus.

Tableau 5 – Les assertions de concepts (individus) et les assertions de rôles en logiques de descriptions.

■ **Exemple**

(t1) Personne $\sqsubseteq \top$	(t6) composer$^{+} \sqsubseteq$ toprole
(t2) Véhicule $\sqsubseteq \top$	(t7) diriger $\sqsubseteq$ toprole
(t3) Auto $\sqsubseteq$ Véhicule	(t8) participer $\sqsubseteq$ toprole
(t4) Moto $\sqsubseteq$ Véhicule $\sqcap \neg$Auto	(t9) conduire $\sqsubseteq$ toprole
(t5) Étape $\sqsubseteq \top$	(t10) piloter $\sqsubseteq$ conduire
(t12) Membre $\equiv$ Personne $\sqcap \exists$composer.Équipe	
(t13) Directeur $\equiv$ Membre $\sqcap \exists$diriger.Équipe	(t11) être_dirigé $\equiv$ diriger^{-}
(t14) Pilote $\equiv$ Membre $\sqcap \exists$piloter.Véhicule	
(t15) Rallye $\equiv \exists$composer^{-}.Étape	
(t16) Équipe $\equiv \forall$participer.Rallye $\sqcap \exists$être_dirigé.Directeur $\sqcap$ (≥ 2 composer^{-}.Membre)	

Figure 29 – Une T-Box, représentant des connaissances structurelles, aussi appelées *terminologiques*.

(a1) Pilote(Alphand)	(a7) piloter(Alphand,pajero)
(a2) Auto(pajero)	(a8) composer(Alphand,Mitsubishi)
(a3) Équipe(Mitsubishi)	(a9) diriger(Serieys,Mitsubishi)
(a4) Rallye(dakar2007)	(a10) participer(Mitsubishi,dakar2007)
(a5) Étape(AtârTichit)	(a11) composer(AtârTichit,dakar2007)
(a6) Étape(TichitNéma)	(a12) composer(TichitNéma,dakar2007)

Figure 30 – Une A-Box, représentant des connaissances factuelles, aussi appelées *assertionnelles*.

La A-Box en Figure 30 est définie sur la T-Box en Figure 29, et reprend les faits du graphe conceptuel en Figure 5. Remarquons l'instance particulière pajero du concept Voiture par rapport au graphe conceptuel.

2.3.4.3 Interprétation et modèle

Définition 2.3 (Interprétation) *Une interprétation $\mathcal{I} = (\Delta_\mathcal{I}, .^\mathcal{I})$ consiste en un ensemble non vide $\Delta_\mathcal{I}$, appelé* domaine de l'interprétation, *et une* fonction d'interprétation $.^\mathcal{I}$ *qui fait correspondre à un concept un sous-ensemble de $\Delta_\mathcal{I}$, à un rôle un sous-ensemble de $\Delta_\mathcal{I} \times \Delta_\mathcal{I}$, et à un individu un élément de $\Delta_\mathcal{I}$ et à une assertion de rôle un élément de $\Delta_\mathcal{I} \times \Delta_\mathcal{I}$, de telle sorte que les expressions en colonne « Sémantique » des Tableaux 4 et 5 soient satisfaites.*

Définition 2.4 (Modèle) *Une interprétation $\mathcal{I}$ est dite un* modèle *pour une T-Box $\mathcal{T}$ si et seulement si $\mathcal{I}$ satisfait toutes les expressions de $\mathcal{T}$. Pour une A-Box $\mathcal{A}$, $\mathcal{I}$ est un* modèle *si et seulement si $\mathcal{I}$ satisfait toutes les assertions de $\mathcal{A}$. $\mathcal{I}$ est un* modèle *pour $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ si et seulement si $\mathcal{I}$ est un modèle pour $\mathcal{T}$ et $\mathcal{A}$ à la fois.*

2.3.4.4 Classification et instanciation

Les logiques de descriptions permettent de faire l'hypothèse d'un *monde ouvert*, où les connaissances d'un domaine peuvent à tout moment être complétées. C'est-à-dire que contrairement à l'hypothèse d'un *monde fermé*[20] où ce qui n'est pas actuellement connu est considéré comme *faux* (aussi appelé *negation as failure*), ici ce qui n'est pas connu reste

[20] L'expression *monde fermé* fut introduite par [Reiter, 1980] dans le domaine des bases de données ; par opposition l'expression *monde ouvert* a vu le jour.

incertain. Cette souplesse de modélisation nécessite de vérifier que l'ensemble des descriptions localement structurées et « éparpillées » ont un sens dans leur globalité. La *classification* et l'*instanciation* des connaissances sont à la base des raisonnements en logiques de descriptions pour répondre à ce besoin de cohérence [Baader *et al.*, 2003]. Le langage OWL étant basé sur les logiques de descriptions, il bénéficie de ces raisonnements.

Définition 2.5 (Classification et Instanciation) *La classification est un processus qui consiste à placer un concept ou un rôle dans leurs hiérarchies respectives, on parle de* classification de concepts *et de* classification de rôles. *L'instanciation*[21] *est un processus qui consiste à déterminer les concepts (notamment le ou les plus spécifiques) dont un individu donné peut être une instance.*

Les quatre principales opérations d'inférence pour la classification et l'instanciation sont le *test de subsomption*, le *test de satisfiabilité de concept*, le *test d'instanciation* et le *test de satisfiabilité d'une base de connaissances*. Des deux premiers tests dérivent le *test d'équivalence de concepts* et le *test de disjonction de concepts* (voir Figure 31).

Définition 2.6 (Test de subsomption) *Pour une base de connaissances $\mathcal{K}$ = (T-Box,A-Box), un concept C_1 (respectivement un rôle r_1) subsume un concept C_2 (respectivement un rôle r_2) par rapport à $\mathcal{K}$, si et seulement si pour tout modèle $\mathcal{I}$ de $\mathcal{K}$, $C_2^{\mathcal{I}} \subseteq C_1^{\mathcal{I}}$ (respectivement $r_2^{\mathcal{I}} \subseteq r_1^{\mathcal{I}}$).*

Définition 2.7 (Test de satisfiabilité (de concept)) *Pour $\mathcal{K}$ = (T-Box,A-Box), un concept C est* satisfiable *si et seulement si il existe au moins un modèle $\mathcal{I}$ de $\mathcal{K}$ tel que $C^{\mathcal{I}} \neq \emptyset$, c-à-d C admet des instances.*

Définition 2.8 (Test d'instanciation) *Pour $\mathcal{K}$ = (T-Box,A-Box), une assertion de concept a est instance d'un concept C (notamment pour C le plus spécifique), notée $\mathcal{K} \models C(a)$, si et seulement si pour tout modèle $\mathcal{I}$ de $\mathcal{K}$, $a^{\mathcal{I}} \in C^{\mathcal{I}}$. On dit que $C(a)$ est* satisfiable *par $\mathcal{I}$.*

Du test d'instanciation découlent le *test de vérification d'instance* et le *test de vérification de rôle* pour vérifier que les assertions définies sont cohérentes avec la T-Box. Le test de vérification d'instance (respectivement de rôle) vérifie que chaque assertion de concept (respectivement assertion de rôle) respecte la définition de ses concepts (respectivement rôles) associés.

Définition 2.9 (Test de satisfiabilité (de $\mathcal{K}$)) *$\mathcal{K}$ = (T-Box,A-Box) est satisfiable si et seulement si il existe au moins un modèle $\mathcal{I}$ tel que $\mathcal{I}$ satisfasse tous les concepts et toutes les assertions de $\mathcal{K}$.*

Remarquons que ces tests à la base des raisonnements en logiques de descriptions sont dépendants les uns des autres. La Figure 31 présente les dépendances entres ces tests.

■ **Exemple**

[21] *On parle parfois de* classification d'instance.

Il est possible d'exprimer, avec le test de subsomption :	**avec le test de satisfiabilité :**	**avec le test d'instanciation :**
C est satisfiable $\Longleftrightarrow$ $C \not\sqsubseteq \bot$	$C_1 \sqsubseteq C_2 \Longleftrightarrow C_1 \sqcap \neg C_2$ insatisfiable	$\mathcal{K}$ est satisfiable $\Longleftrightarrow$ $\mathcal{K} \not\models \bot(a)$
C_1 et C_2 sont équivalents $\Longleftrightarrow C_1 \sqsubseteq C_2$ et $C_2 \sqsubseteq C_1$	C_1 et C_2 sont équivalents $\Longleftrightarrow$ $\neg C_1 \sqcap C_2$ et $C_1 \sqcap \neg C_2$ sont insatisfiables	C est satisfiable $\Longleftrightarrow \{C(a)\}$ est satisfiable
C_1 et C_2 sont disjoints $\Longleftrightarrow (C_1 \sqcap C_2) \sqsubseteq \bot$	C_1 et C_2 sont disjoints $\Longleftrightarrow C_1 \sqcap C_2$ est insatisfiable	$C_1 \sqsubseteq C_2 \Longleftrightarrow \{C_1(a)\} \models C_2(a)$
	$\mathcal{K} \models C(a) \Longleftrightarrow \mathcal{K} \cup \{\neg C(a)\}$ est insatisfiable	

Figure 31 – Dépendances entre les tests de subsomption, de satisfiabilité et d'instanciation

$\Delta_{\mathcal{I}}$ = {Serieys, Alphand, Mitsubishi, pajero, dakar2007, AtârTichit, TichitNéma}

Serieys$^{\mathcal{I}}$ = Serieys	Pilote$^{\mathcal{I}}$ = {Alphand}
Alphan$^{\mathcal{I}}$ = Alphand	Directeur$^{\mathcal{I}}$ = {**Serieys**}
Mitsubishi$^{\mathcal{I}}$ = Mitsubishi	Membre$^{\mathcal{I}}$ = {**Serieys, Alphand**}
pajero$^{\mathcal{I}}$ = pajero	Personne$^{\mathcal{I}}$ = {**Serieys, Alphand**}
dakar2007$^{\mathcal{I}}$ = dakar2007	Auto$^{\mathcal{I}}$ = {pajero}
AtârTichit$^{\mathcal{I}}$ = AtârTichit	Moto$^{\mathcal{I}}$ = $\varnothing$
participer$^{\mathcal{I}}$ = {(Mitsubishi,dakar2007)}	Véhicule$^{\mathcal{I}}$ = {**pajero**}
piloter$^{\mathcal{I}}$ = {(Alphand,pajero)}	Équipe$^{\mathcal{I}}$ = {Mitsubishi}
conduire$^{\mathcal{I}}$ = {**(Alphand,pajero)**}	Rallye$^{\mathcal{I}}$ = {dakar2007}
diriger$^{\mathcal{I}}$ = {(Serieys,Mitsubishi)}	Étape$^{\mathcal{I}}$ = {AtârTichit, TichitNéma}
être_dirigé$^{\mathcal{I}}$ = {**(Mitsubishi,Serieys)**}	

composer$^{\mathcal{I}}$ = {(Alphand,Mitsubishi), (AtârTichit,dakar2007), (TichitNéma,dakar2007)}

Figure 32 – Un modèle $\mathcal{I}$ pour la base de connaissances (T-Box,A-Box) des Fig. 29 et 30

Après les processus de classification et d'instanciation appliqués sur le couple (T-Box,A-box) en Figures 29 et 30, il existe une interprétation qui satisfasse ce couple. La Figure 32 présente un modèle pour cette T-Box et cette A-Box. Les éléments en gras sont ceux déduits par les processus de classification et d'instanciation. En effet, les opérations d'inférence au niveau terminologique ont conclu que Membre $\sqsubseteq$ Personne, que Directeur $\sqsubseteq$ Membre et que Pilote $\sqsubseteq$ Membre (*cf.* (t12) à (t14)). Les opérations d'inférence au niveau assertionnel ont conclu : être_dirigé(Mitsubishi, Serieys) (*cf.* (t11) et (a8)), Directeur(Serieys) [22] puisque Serieys et le premier argument d'une assertion du rôle diriger (*cf.* (t13) et (a9)), et Véhicule(pajero) (*cf.* (t14) et (a7)).

Remarquons que ce modèle est satisfaisant bien que l'équipe Mitsubishi ne soit composée que du seul membre Alphand (*cf.* (a8)) et non de deux comme cela devrait être la cas (*cf.* (t16)). Il ne s'agit cependant pas d'une erreur d'interprétation. En effet, rappelons que les raisonnements en logiques de descriptions s'effectuent sous l'hypothèse d'un monde ouvert, où ce qui n'est pas connu reste incertain. Ici, si seulement un membre constitue cette équipe, c'est « tout simplement » que le ou les autres menbres ne sont pas encore connus.

[22] ainsi que Membre(Serieys) et Personne(Serieys), tout comme Membre(Alphand) et Personne(Alphand)

Notons que certains raisonneurs en logiques de descriptions (voir la Section 2.4.1) travaillent uniquement sous l'hypothèse d'un monde fermé, comme CWM (*Closed World Machine*), et que d'autres peuvent ou non travailler sous cette hypothèse, comme l'outil RacerPro.

2.3.4.5 Expressivité des logiques de descriptions

Les langages des logiques de descriptions se distinguent les uns des autres par l'ensemble des constructeurs qu'ils offrent.

L'une des premières logiques de description est le langage $\mathcal{FL}^-$ - pour *Frame Logic* - [Brachman et Levesque, 1984], qui a été proposé comme un formalisme pour la sémantique des *frames* de Minsky, et pour lequel ont été établis des résultats théoriques de complexité sur la subsomption par Brachman et Levesque. Le langage $\mathcal{FL}^-$ offre les constructeurs de conjonction ($\sqcap$), de quantificateur universel ($\forall$), de quantificateur existentiel non typé ($\exists r$, c-à-d $\exists r.\top$), et la restriction du co-domaine d'un rôle ($r|C$). Ce constructeur $r|C$ peut être rapproché du constructeur existentiel typé ($\exists r.C$), mais avec deux différences : il s'applique aux rôles et n'a aucune contrainte d'existence ; $(r|C)^{\mathcal{I}} = \{(x,y) \in \Delta_{\mathcal{I}} \times \Delta_{\mathcal{I}} \mid (x,y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$.

Le langage $\mathcal{FL}$ est une extension de $\mathcal{FL}^-$ où le quantifieur existentiel est typé ($\exists r.C$). Le langage $\mathcal{AL} = \{\top, \bot, C_1 \sqcap C_2, \forall r.C, \exists r, \neg A\}$ [Schmidt-Schauss et Smolka, 1991] étend le langage $\mathcal{FL}^-$ en y ajoutant la négation, sur des concepts primitifs uniquement. Ce langage peut-être considéré comme la logique de base des autres logiques de descriptions. En effet, les logiques de descriptions sont des combinaisons des différents éléments du Tableau 4. Par exemple si on ajoute la négation complète (étendue aux concepts définis), repérée par la lettre $\mathcal{C}$, à la logique $\mathcal{AL}$ on obtient la logique $\mathcal{ALC}$. On peut remarquer que le langage $\mathcal{ALC}$ équivaut à $\mathcal{ALUE}$, puisque l'union ($\mathcal{U}$) et la quantification existentielle typée ($\mathcal{E}$) peuvent s'exprimer par la négation complète et inversement : $C_1 \sqcup C_2$ correspond à $\neg(\neg C_1 \sqcap \neg C_2)$ et $\exists r.C$ correspond à $\neg\forall r.\neg C$ [Baader *et al.*, 2003].

2.3.5 Sémantique logique de OWL

Définition 2.10 (Sémantique logique) *À un document OWL est associée une formule logique dans le langage $\mathcal{SHIF}$(D)*[23] *pour OWL Lite et dans le langage $\mathcal{SHOIN}$(D) pour OWL DL [Horrocks et Patel-Schneider, 2003]. Une* classe *en OWL correspond à un* concept *en logique de description, une* propriété concept *en OWL correspond à un* rôle *en DL. Les* datatypes *et* propriétés dadatypes *sont nouvellement introduits ; une propriété datatype lie une classe à un datatype. L'interprétation $\mathcal{I}_{OWL}$, qui associe une sémantique à un document OWL, étend l'interprétation $\mathcal{I}$ définie en Section 2.3, de façon à prendre en compte les datatypes. L'interprétation $\mathcal{I}_{OWL} = (\Delta_{\mathcal{I}}, \Delta_D, .^{\mathcal{I}})$ consiste en un ensemble non vide $\Delta_{\mathcal{I}}$, appelé cette fois* domaine de l'interprétation des instances, *en un ensemble Δ_D disjoint de $\Delta_{\mathcal{I}}$, appelé* domaine de l'interprétation des datatypes, *et une* fonction d'interprétation[24] *$.^{\mathcal{I}}$ qui fait correspondre : à une classe*

[23] *$\mathcal{S}$ désigne le langage $\mathcal{ALCR}^+$, $\mathcal{F}$ la restriction de nombre fonctionnelle (0 ou 1), et (D) fait référence à* datatype.

[24] *Certains auteurs dissocient cette fonction d'interprétation en deux sous-fonctions : $.^{\mathcal{I}}$ pour interpréter les concepts, les propriétés concepts et les individus, et $.^{D}$ pour interpréter les datatypes, les propriétés datatypes et les valeurs.*

Constructeur	Syntaxe	Sémantique
hiérarchie de datatypes	$D_1 \sqsubseteq D_2$	$(D_1 \sqsubseteq D_2)^{\mathcal{I}} = \{d \in \Delta_D \mid d \in D_1^{\mathcal{I}} \rightarrow d \in D_2^{\mathcal{I}}\}$
quantificateur universel	$\forall u.D$	$(\forall u.D)^{\mathcal{I}} = \{x \in \Delta_{\mathcal{I}} \mid \forall d \in \Delta_D, (x,d) \in u^{\mathcal{I}} \rightarrow d \in D^{\mathcal{I}}\}$
quantificateur existentiel	$\exists u.D$	$(\exists u.D)^{\mathcal{I}} = \{x \in \Delta_{\mathcal{I}} \mid \exists d \in \Delta_D, (x,d) \in u^{\mathcal{I}} \wedge d \in D^{\mathcal{I}}\}$
restrictions de nombre	$\geq n\ u$	$(\geq n\ u)^{\mathcal{I}} = \{x \in \Delta_{\mathcal{I}} \mid \text{card}(\{d \in \Delta_D \mid (x,d) \in u^{\mathcal{I}}\}) \geq n\}$
(au-plus et au-moins)	$\leq n\ u$	$(\leq n\ u)^{\mathcal{I}} = \{x \in \Delta_{\mathcal{I}} \mid \text{card}(\{d \in \Delta_D \mid (x,d) \in u^{\mathcal{I}}\}) \leq n\}$
type énuméré	$\{v_1, ..., v_n\}$	$\{v_1, ..., v_n\}^{\mathcal{I}} = \{v_1^{\mathcal{I}}, ..., v_n^{\mathcal{I}}\}$
Axiome	**Syntaxe**	**Sémantique**
définition de valeur	$D(v)$	$D(v)^{\mathcal{I}} = v^{\mathcal{I}} \in D^{\mathcal{I}}$
affectation de valeur	$u(a,v)$	$u(a,v)^{\mathcal{I}} = (a^{\mathcal{I}}, v^{\mathcal{I}}) \in u^{\mathcal{I}}$

D, D_1 et D_2 sont des datatypes, u, u_1 et u_2 des propriétés datatypes, a un individu, et v, v_1 à v_n des valeurs.

Tableau 6 – Les datatypes et leurs valeurs en OWL

un sous-ensemble de $\Delta_{\mathcal{I}}$, à une propriété concept un sous-ensemble de $\Delta_{\mathcal{I}} \times \Delta_{\mathcal{I}}$, à un individu un élément de $\Delta_{\mathcal{I}}$, à un datatype un sous-ensemble Δ_D, à une propriété datatype un sous-ensemble de $\Delta_{\mathcal{I}} \times \Delta_D$, et à une valeur un élément de Δ_D, de telle sorte que les expressions en colonne « Sémantique » des Tableaux 4, 6 et 5 soient satisfaites.

Le langage OWL est en réalité divisé en trois sous-langages d'expressivité croissante [Patel-Schneider *et al.*, 2004] : *OWL Lite*, *OWL DL* et *OWL Full*, de sorte que tout document OWL Lite soit aussi un document OWL DL, et que tout document OWL DL soit un document OWL Full. OWL Lite permet principalement de modéliser en hiérarchies les concepts et les rôles, et possède des mécanismes simples de construction de modélisation. Par exemple, seules les cardinalités 0 ou 1 sont gérées par OWL Lite. OWL DL comprend toutes les structures du langage OWL, avec certaines restrictions d'utilisation, de telle sorte qu'il y ait le meilleur compromis entre expressivité d'une part, et complétude du calcul et décidabilité d'autre part. OWL Full offre une expressivité maximale et la liberté syntaxique de RDF.

Notons que les opérations de classification et d'instanciation sont dans la classe de complexité NEXPTIME-Complet[25] pour OWL DL mais sont indécidables en OWL Full [Horrocks et Patel-Schneider, 2003; Nebel, 1990b].

2.3.5.1 OWL Full

Le langage OWL Full n'est en réalité pas un sous-langage, mais contient l'intégralité des structures du langage OWL. Il offre donc une expressivité maximale et la liberté syntaxique de RDF :

[25][Cook, 1972] montre que P $\subset$ EXP et NP $\subset$ NEXPTIME.

- La ressource owl:Class est équivalente (*cf.* propriété owl:equivalentClass) à rdfs:Class.
- Un individu peut être considéré comme une classe à son tour. Par exemple, l'individu 2cv peut être une instance de la classe Voiture, mais aussi la classe de toutes les 2cv (et 2583RT49 en serait une instance).
- La ressource owl:Thing est identique (*cf.* propriété owl:sameAs) à rdfs:Resource. Ceci a pour conséquence que l'univers des individus se compose de toutes les ressources. En d'autres termes, les individus et les données font partie du même ensemble. Ainsi, les rôles d'objets et les rôles de types de données ne sont pas disjoints.
- La classe owl:ObjectProperty équivaut à rdf:Property, et owl:DatatypeProperty est une sous-classe de owl:ObjectProperty.

Le langage OWL Full servira typiquement aux utilisateurs voulant combiner la capacité d'expression du langage OWL à la flexibilité et aux caractéristiques de méta-modélisation de RDF. En contrepartie, l'efficacité de calcul n'est pas garantie. On entend par *efficacité*, d'une part la complétude du calcul, c'est-à-dire toutes les inférences sont garanties calculables, et d'autre part la décidabilité, c'est-à-dire tous les calculs s'achèveront dans un intervalle de temps fini. Nous ne considérerons pas ce sous-langage par la suite du fait de sa non efficacité, et nous nous en tiendrons au sous-langages OWL DL et OWL Lite.

2.3.5.2 OWL DL

Le langage OWL DL est un sous-langage de OWL (Full) qui impose certaines contraintes dans l'utilisation des structures du langage OWL. Ces contraintes sont principalement les suivantes :

- Il y a séparation deux à deux entre les classes, les types de données, les rôles d'objets, les rôles de types de données, les individus, les données, ainsi que les propriétés d'annotation, les propriétés d'ontologies (c'est-à-dire celles concernant l'importation et le versionnage) et le vocabulaire intégré. Cela signifie, par exemple, qu'une classe d'une modélisation en OWL DL ne peut pas être en même temps un individu d'une modélisation en OWL DL.
- Le ressource owl:Class est une sous-classe de rdfs:Class. Ainsi les classes RDF ne sont pas toutes des classes OWL DL.
- Le langage OWL DL interdit de placer une contrainte de cardinalité (ni locale ni globale) sur les rôles transitifs, les rôles symétriques ou leurs sous-rôles.
- Toutes les classes et tous les rôles référencés doivent être explicitement typés comme respectivement de type owl:Class ou de type owl:ObjectProperty ou owl:DatatypeProperty. Par exemple, $C2$ doit être explicitement de type owl:Class dans la description suivante :

```
<owl:Class rdf:about = "C1">
  <rdfs:subClassOf rdf:resource="#C2" />
</owl:Class>
```

OWL DL tire son appellation « DL » des travaux réalisés dans le domaine des moteurs de raisonnement en logiques de descriptions (Description Logics), lesquels imposent ces contraintes de modélisation ci-dessus afin de garantir l'efficacité des raisonnements.

Le langage OWL DL servira typiquement aux utilisateurs cherchant le meilleur compromis entre expressivité et efficacité.

2.3.5.3 OWL Lite

Le langage OWL Lite est un sous-langage de OWL DL. Outre les contraintes d'utilisation des structures du langage OWL imposées par OWL DL, OWL Lite ajoute d'autres contraintes d'utilisation qui sont :

- Les propriétés suivantes sont interdites d'utilisation : owl:unionOf, owl:complementOf, owl:disjointWith, owl:oneOf, owl:DataRange et owl:hasValue.
- Les valeurs de cardinalité doivent être soit 0 (zéro), soit 1. Ainsi, les rôles ont pour caractéristique résultante d'être soit optionnels, soit obligatoires.
- Le sujet d'un triplet de prédicat owl:equivalentClass ou rdfs:subClassOf doit être un nom de classe (pas de classe anonyme ni de restriction de classe), et son objet doit être un nom de classe ou une restriction (pas de classe anonyme).
- Le sujet d'un triplet de prédicat rdfs:subClassOf doit être un nom de classe, et son objet doit être un nom de classe ou une restriction.
- L'objet des triplets de prédicat owl:allValuesFrom et owl:someValuesFrom doit être un nom de classe ou un nom de type de donnée.
- L'objet des triplets de prédicat rdfs:domain doit être un nom de classe.
- L'objet des triplets de prédicat rdfs:range doit être un nom de classe ou un nom de datatype.
- L'objet des triplets de prédicat rdf:type doit être un nom de classe ou une restriction.

L'idée derrière les limitations d'expressivité avec OWL Lite est de produire un sous-ensemble minimal du langage OWL, relativement simple à mettre en œuvre par les utilisateurs n'ayant pas besoin de toute la richesse expressive de OWL. Les structures de langage OWL Lite fournissent essentiellement la construction d'une hiérarchie de classes, par l'emploi de rdfs:subClassOf ou par des restrictions de rôles.

Les limitations du langage OWL Lite le placent à un niveau de complexité inférieur à celui du langage OWL DL. Cela peut avoir un impact positif sur l'efficacité des moteurs de raisonnement complets OWL Lite.

2.4 Outils

2.4.1 Raisonneurs en logiques de descriptions

Les raisonneurs [26] en logiques de descriptions profitent des dépendances entre les tests de subsomption et de satisfiabilité (voir Figure 31). Deux familles d'algorithmes de raisonnement pour les logiques de descriptions cohabitent.

La première famille, issue du système KL-ONE [Brachman et Schmolze, 1985], utilise des algorithmes de type *normalisation/comparaison* (abrégé par NC), qui sont basés sur la subsomption (voir [Nebel, 1990a] pour un exemple d'un tel algorithme). Ces algorithmes ont l'avantage d'être simples à mettre en œuvre, mais sont efficaces seulement pour des

[26] Un raisonneur est une application logicielle qui peut inférer des conséquences logiques issues d'un ensemble de connaissances établies.

logiques de descriptions très simples comme $\mathcal{FL}^-$ ou $\mathcal{AL}$. Des systèmes comme CLASSIC [Patel-Schneider, 1991] ont fait le choix de ne pas tout représenter, afin d'adopter un comportement déductif contrôlable. À l'inverse, des systèmes comme LOOM [MacGregor et Bates, 1987] et BACK [Peltason, 1991] offrent plus d'expressivité.

La seconde famille utilise des algorithmes basés sur la satisfiabilité, avec l'utilisation d'algorithmes appelés *tableaux* [27] [Baader et Sattler, 2001]. Les raisonneurs actuels font partie de cette seconde famille. Ces algorithmes peuvent s'appliquer sur des logiques de descriptions plus expressives, comme celles basées sur $\mathcal{SH}$. Des systèmes comme KRIS [Baader et Hollunder, 1991], FaCT - *Fast Classification of Terminologies* - [Horrocks, 1998], ou d'autres plus récents comme FaCT++ (version en C++ de FaCT), Pellet [Sirin *et al.*, 2004] et Racer - *Renamed ABox and Concept Expression Reasoner* - [Haarslev et Müller, 2001] se basent sur de tels algorithmes, tout en possédant des optimisations sophistiquées. FaCT++, Pellet et Racer sont orientés pour la manipulation de documents OWL DL, et sont capables de raisonner sur la T-Box et la A-Box (uniquement la T-Box pour FaCT++). Nous invitons le lecteur en guise d'exemple mettant en œuvre les processus de classification et d'instanciation sur une modélisation en logique de description selon un algorithme des tableaux de se reporter au chapitre *II* du cours très pédagogique de Serge Haddad [28] intitulé « *Cours de logique pour l'IA* ».

2.4.2 Autres langages et outils pour le Web sémantique

Contrairement au modèle des GCs et UML basé sur OCL, OWL ne dispose pas de *variable* au sens des variables en logique classique [29] du fait qu'il soit basé sur les logiques de descriptions [Borgida, 1996]. Cela peut être expliqué de manière intuitive comme suit : après les processus de classification et d'instanciation, tous les individus doivent au final avoir « trouvé leurs places » parmi les ensembles de concepts (de classes, en OWL). L'interrogation de documents OWL ou l'application de règles s'avèrent donc limitées d'une façon générale. La prise en compte des règles est d'ailleurs considérée comme « la prochaine étape du Web Sémantique » par ses auteurs [Berners-Lee *et al.*, 2001]. Cependant, si l'acceptation de RDF et OWL font plus ou moins l'unanimité comme recommandations pour représenter des connaissances sur le Web, la situation pour l'établissement de recommandations pour un langage [30] de règles est plus contreversé (voir par exemple la dynamique sur ce sujet à l'adresse suivante : `http://www.w3.org/2005/rules/wg/wiki/List_of_Rule_Systems`).

Actuellement, en complément des raisonnements en logiques des descriptions, plusieurs langages et outils proposent l'application de règles et l'interrogation de documents OWL. Ces travaux sont définis soit en complément du modèle OWL, soit comme une

[27] Se reporter notamment à la conférence annuelle internationnale traitant des modes de raisonnements selon des méthodes dites des tableaux ou apparentées ; `http://i12www.ira.uka.de/TABLEAUX/`.

[28] `http://www.lsv.ens-cachan.fr/~haddad/`

[29] Une variable est une entité, d'un type donné, qui peut référencer successivement - c'est-à-dire de manière variable - différentes entités de même type.

[30] Il est d'ailleurs étonnant de voir que la « bataille » sur le choix d'un format pour *représenter* des règles se fait au détriment d'une discussion sur comment *exploiter* ces règles.

« sur-couche » de OWL (voir la Figure 17). Différentes approches y sont proposées : en évoluant dans un cadre comparable aux traitements faits en logique classique, en combinant logique classique et logiques de descriptions, ou à un niveau logiciel par des méthodes ad-hoc. Cependant, les travaux manipulant des règles ou requêtes sur des documents OWL sont récents et demandent encore à être bien formalisés et étudiés [31].

2.4.2.1 Éditeurs

Protégé [32] est à la base un éditeur d'ontologies développé en Java. Soutenu par une forte communauté, Protégé possède de très nombreux plugins [33] tant pour l'amélioration de la visualisation des connaissances modélisées que pour effectuer des raisonnements sur ces connaissances (essentiellement par la mise en relation avec des raisonneurs). Il est actuellement l'outil incontrounable de modélisation dans l'environnement du Web Sémantique.

L'environnement de développement Eclipe [34] dispose récemment d'un éditeur d'ontologie OWL (*OWL Visual Editor* [35]), ainsi qu'un ensemble d'outils pour l'exploitation de documents OWL. Ces outils sont regroupés au sein du projet MDT [36] (*Model Development Tools*) sous l'appelation EODM offrant la possibilité de parser un document OWL, de transformer les connaissances modélisées vers d'autres langages comme UML, ou encore d'effectuer quelques raisonnements exploitant l'organisation taxonomique des classes et rôles.

2.4.2.2 Langages et outils d'applications de règles

SWRL - *Semantic Web Rule Language* - [Horrocks *et al.*, 2004] est un langage pour appliquer des règles sur des documents OWL. SWRL tend actuellement à devenir une recommandation W3C, et est parfois vue comme une extension de OWL avec prise en compte des variables.

La manipulation de OWL conjointement avec des règles SWRL peut par exemple se faire avec :

– la plateforme SweetRules [37] qui regroupe plusieurs outils Java. Ces outils sont les moteurs d'inférence Jena2 [38] ou Jess [39], et l'éditeur SWRL Editor [40] constituant une extension du plugin OWL [Knublauch *et al.*, 2004] de Protégé.

[31] D'autres travaux comme Carin [Levy et Rousset, 1998], combinant règles de Horn et logiques de descriptions (pas spécialement OWL donc) sont quant à eux plus anciens et bien formalisés.

[32] `http://protege.stanford.edu/`

[33] Un plugin, aussi appelé greffon, est un programme qui interagit avec le logiciel principal pour lui apporter de nouvelles fonctionnalités.

[34] `http://www.eclipse.org/`

[35] `http://owlve.sourceforge.net/`

[36] `http://www.eclipse.org/modeling/mdt/`

[37] `http://sweetrules.projects.semwebcentral.org/`

[38] `http://jena.sourceforge.net/`

[39] `http://herzberg.ca.sandia.gov/jess/`

[40] `http://protege.stanford.edu/plugins/owl/swrl/`

– la bibliothèque KAON2 [41] qui utilise des raisonneurs en logiques de descriptions, tels que FaCT++, Pellet ou Racer[42], et intègre en plus un moteur d'inférence de règles [Motik *et al.*, 2004]. KAON2 est implémenté en Java, et peut communiquer avec Protégé via une interface DIG [43]. Le groupe de travail [44], de l'université de Manchester, sur cette interface vise à élaborer un standard écrit en XML pour l'échange d'informations entre des outils du Web sémantique qui sont les éditeurs et les raisonneurs en logiques de descriptions.

2.4.2.3 Langages et outils d'interrogation

Parmi les langages d'interrogation en passe de devenir des recommandations W3C ou des standards de fait, nous pouvons citer :

– RQL - *Rdf Query Language* - [Karvounarakis *et al.*, 2002] pour l'interrogation de documents RDF(S) avec une syntaxe « à la SQL » ;
– SPARQL - acronyme récursif de *Sparql Protocol and Rdf Query Language* - [Prudhommeaux et Seaborne, 2008] pour l'interrogation de documents RDF(S) suivant une syntaxe « SQL-like » ;
– OWL-QL - *OWL Query Language* - [Fikes *et al.*, 2004] pour l'interrogation de documents OWL, avec une syntaxe qui étend OWL.

L'excécution de requêtes écrites dans ces langages d'interrogation peut se faire avec les outils suivants. Sesame [45] et Jena2 proposent l'interprétation de requêtes en RQL. CWM [Berners-Lee, 2000], Euler [46] et Jena2 permettent l'interprétation de SPARQL. Racer constitue une application pour utiliser OWL-QL.

Notons que les langages d'interrogation de données XML, comme XQuery, peuvent servir pour l'interrogation de documents XML/RDF(S) ou XML/OWL mais leurs utilisations en restent difficiles puisque les concepts propres à RDF(S) ou OWL (comme les notions de classe, de rôle ou d'héritage) n'y sont pas incorporés et doivent donc être exploités « à la main ».

2.4.2.4 Langages et outils de vérification

La notion de *contraintes*, au sens de celles présentées avec le modèle GCs et non de celles venant des restrictions d'utilisation des constructeurs en logiques de descriptions, ne semble pas actuellement préoccuper la communauté du Web Sémantique. Citons le travail [McKenzie *et al.*, 2004] qui étend SWRL pour exprimer des contraintes, et le langage PAL [47] - *Protégé Axiom Language* - dédié à Protégé pour définir contraintes et requêtes sous forme de règles.

[41] `http://kaon2.semanticweb.org/`
[42] dans sa version Pro, `http://www.racer-systems.com/products/racerpro/`
[43] `http://dig.sourceforge.net/` et `http://dig.cs.manchester.ac.uk/`
[44] `http://http ://dl.kr.org/dig/`
[45] `http://sourceforge.net/projects/sesame/`
[46] `http://www.agfa.com/w3c/euler/`
[47] `http://protege.stanford.edu/plugins/paltabs/`

2.5 Conclusion

Basé sur le langage RDF formalisant les connaissances sous forme de triplets <*sujet, prédicat, donnée*> et le langage RDFS organisant en taxonomie les connaissances structurelles d'une modélisation, OWL est un langage très expressif et formel. Le langage OWL a été conçu par le W3C en trois sous-familles d'expressivité croissante et de manière à ce que chaque sous-famille dispose d'une sémantique en une logique de description. Le modèle OWL dispose ainsi d'une capacité expressive riche et d'une capacité de raisonnement pour classifier et instancier les connaissances.

Un des avantages du modèle OWL est d'offrir une grande souplesse de modélisation des connaissances, notamment pour une phase initiale de capitalisation des connaissances lors d'une modélisation. En effet, le choix de faire l'hypothèse d'un monde ouvert et la non supposition du nom unique influence sur la façon de modéliser les connaissances d'un domaine. Contrairement au modèle des GCs où la démarche de modélisation est *intensionnelle*, le modèle OWL peut - et c'est généralement le cas - fournir une démarche *extensionnelle* de modélisation. Une démarche intensionnelle de modélisation consiste à définir les connaissances structurelles puis, à partir de la structure, les connaissances factuelles du domaine. Une démarche extensionnelle de modélisation consiste littéralement à s'intéresser d'abord aux faits, puis à les regrouper selon des caractéristiques communes et ainsi définir la structure d'un domaine. On entend aussi et surtout par cette démarche extensionnelle et propre aux langages du Web sémantique le fait de décrire de façon « éparpillée » des éléments factuels et structurels, sans une véritable vue globale et organisée de l'ensemble de la modélisation. L'idée est que des raisonneurs seront là pour réorganiser les connaissances structurelles par classification et affiner les connaissances factuelles par instanciation.

Cette souplesse de modélisation a cependant pour conséquence une exploitation limitée des connaissances modélisées par d'autres formes de raisonnements (que ceux issus des logiques des descriptions), comme la recherche d'information. Ainsi, des langages et outils récents, s'appuyant généralement sur la logique classique, ont été développés ou sont en cours d'élaboration pour l'interrogation et l'application de règles sur des connaissances modélisées en RDF(S) ou OWL.

Chapitre 3

Langage UML

UNIFIED Modeling Language (UML) est un langage essentiellement visuel de modélisation dite orientée objet. Standardisé en 1997 par l'OMG [1], UML est aujourd'hui l'environnement quasi-incontournable et de référence dans l'industrie en terme de modélisation orientée objet.

L'approche objet n'est pas apparue dans la dernière décennie du XX^{e} siècle. Elle tire son origine des *frames* de M. Minsky [1980], elles-mêmes en partie inspirées des réseaux sémantiques [Quillian, 1969]. Simula, premier langage de programmation ayant implémenté le concept de type abstrait à l'aide de classes, date de 1967. En 1976, Smalltalk implémente les concepts fondateurs de l'approche objet : encapsulation, agrégation, héritage. Les premiers compilateurs C++ datent du début des années 80 et de nombreux langages orientés objets « académiques » ont commencé à démocratiser les concepts objets (Eiffel, EiffelObjective C, EiffelLoops, *etc.*). Cependant, dans les années 80, l'approche objet souffre de méthodes de conception inadaptées. Les méthodes utilisées dans ces années-là étaient conçues pour la gestion de bases de données relationnelles voire la programmation procédurale. Ces méthodes étaient fondées sur une modélisation séparant les données des traitements, ce qui n'est pas le cas en objet. Lorsque la programmation objet prend de l'importance au début des années 90, l'existence d'une méthode qui lui soit adaptée devient nécessaire. Des dizaines de méthodes objet sont apparues durant le milieu des années 90 (*OMT, OOD, OOSE, Fusion, Classe-Relation, HOOD, OOA, OOM, etc.*), mais aucune ne s'est réellement imposée jusqu'à UML.

Le succès de UML vient d'un part de son aspect très largement visuel, sa notation graphique permet d'exprimer visuellement les connaissances tant structurelles que factuelles d'un domaine à représenter. UML offre aussi une notation permettant une représentation standard et stricte de concepts abstraits que sont les classes et objets, afin de constituer un langage commun de modélisation. UML n'est cependant pas une méthode

[1] L'*OMG* est un organisme à but non lucratif, créé en 1989 à l'initiative de grandes sociétés (*HP, Sun, Unisys, American Airlines, Philips, ...*), dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes. L'OMG est notamment à l'origine des standards UML, MOF (Meta-Object Facility), CORBA (COmmon Request Broker Architecture) et IDL (Interface Definition Language) ; ainsi que la recommandation MDA (Model Driven Architecture) ou le langage standardisé de transformation de modèles QVT (Query/View/Transformation). `http://www.omg.org/`.

de conception objet, bien qu'il donne une dimension méthodologique à l'approche objet. Le but poursuivi par ses créateurs est qu'il serve de support à une analyse basée sur les concepts objet ; en faire une méthode n'est pas jugé opportun au vue de la diversité des cas particuliers en modélisation. UML comble une lacune importante des technologies objet : il permet d'exprimer et d'élaborer des modèles objet, indépendamment de tout langage de programmation.

Concrètement, le langage UML définit plusieurs types de diagrammes [OMG, 2003a; OMG, 2003b] pour représenter différents aspects et vues d'un domaine. Les différents diagrammes sont complémentaires, car chacun s'intéresse à un aspect précis de la représentation. La combinaison des différents types de diagrammes UML offre une vue complète du domaine modélisé. Dans la version 1.4 de UML, les neuf [2] types de diagrammes sont regroupés en deux grandes catégories : ceux modélisant les aspects *statiques* du domaine et ceux en modélisant les aspects *dynamiques*.

Le comportement dynamique étant très lié au domaine informatique, et donc pas assez général d'un point de vue représentation des connaissances, il ne sera pas traité dans le cadre de cette thèse.

Les diagrammes modélisant les aspects statiques d'un domaine sont, quant à eux, suffisamment génériques pour aussi représenter des connaissances qui ne soient pas d'un domaine informatique. Les trois principaux diagrammes statiques sont : le *diagramme de cas d'utilisations*, le *diagramme de classes* et le *diagramme d'objets*. Si le diagramme de classes est le diagramme principal de toute modélisation UML, celui des cas d'utilisations sert surtout à « dégrossir » la phase d'analyse lors d'une modélisation. Ce dernier diagramme n'offre pas une vision précise des connaissances, et n'est pas utilisé dans cette thèse. Un diagramme de classes spécifie les connaissances structurelles du domaine à représenter. Un diagramme d'objets modélise des connaissances factuelles, et souvent partielles, du domaine.

Ce chapitre fournit dans les deux premières sections une introduction au diagramme de classes et au diagramme d'objets du langage UML. Une troisième section se penche sur des extensions du langage UML : les *stéréotypes* [OMG, 2003a; OMG, 2003b] qui permettent de modifier la signification d'une notion et accroître ainsi l'expressivité du langage, OCL [OMG, 2003c] - *Object Constraint Language* - qui d'une part dote UML d'une sémantique formelle proche de la logique classique et d'autre part permet d'exprimer des contraintes dites d'invariance sur les connaissances modélisées, et le format *XMI* - basé sur XML - qui offre un standard de stockage et d'échange de connaissances modélisées dans le modèle UML. Dans une quatrième section sont présentés différents outils logiciels pour l'édition de diagrammes UML, ainsi que pour la prise en compte des contraintes OCL et la génération ou l'importation de fichiers XMI [OMG, 2002].

[2] Dans sa version 2.0, UML dispose de trois nouveaux types de diagrammes, regroupés dans une nouvelle catégorie pour organiser et gérer les modules qui composent un programme informatique objet.

Sommaire

3.1 Diagramme de classes

Un diagramme de classes [OMG, 2003b; OMG, 2003a] est une collection d'éléments statiques (classes, associations, paquetages, interfaces, ...), qui modélisent la structure d'un domaine. Un diagramme de classes fait abstraction des aspects dynamiques du domaine qu'il modélise.

Les principaux éléments présents dans un diagramme de classes sont : les classes et les relations entre ces classes. Ces relations peuvent-être des associations, des liens de généralisation ou des liens de dépendance.

3.1.1 Classe

En modélisation objet, une *classe* est un élément structurel du domaine modélisé. Il s'agit d'un « moule » (un gabarit) à partir duquel sont créés des *objets* ayant certaines caractéristiques communes. Ces objets s'appellent des *instances*, et seront abordés dans la Section 3.2. La relation qui existe entre un objet et la classe est la *relation d'instanciation*. Cette relation d'instanciation est représentée en UML au niveau du diagramme d'objets, où à un objet est associé le nom de la classe. Le diagramme d'objets est présenté en Section 3.2 . On dit que l'objet est une « instance-de » la classe, ou encore qu'il « a pour type » la classe. Par exemple, si la classe Personne désigne les personnes en général, Jean en est une instance : un individu particulier. Une classe est donc un type abstrait commun à un ensemble d'objets, dont leurs caractéristiques communes sont définies au niveau de la classe par des propriétés que sont les *attributs* et les *opérations*.

Un attribut définit une information que tout objet du type de la classe doit renseigner d'une valeur qui lui est propre. Un attribut est un triplet [3] (*nom, type, multiplicité*), où le *nom* identifie l'attribut, *type* est celui associé à la valeur, et la *multiplicité* indique combien de fois une valeur de ce type doit être renseignée sous ce même nom d'attribut. Généralement, un attribut fait référence à un type prédéfini. L'ensemble des types prédéfinis inclut les types primitifs, aussi appelés *datatypes*, tels que le type booléen, entier ou chaîne de caractères, et le type énumération. Il est aussi possible que le type d'un attribut soit une classe. Cependant si cette classe est aussi définie dans le diagramme de classes, il sera préféré en terme de modélisation d'utiliser une association plutôt qu'un attribut ; l'association ayant dans ce cas le nom que l'attribut aurait eu et ayant pour co-domaine le type en question (voir la Section 3.1.3 pour les notions d'association et de co-domaine).

Une opération est une caractéristique qui décrit un comportement donné que peuvent avoir les objets de la classe. Cette caractéristique est très liée à l'aspect dynamique de la modélisation, et ne sera que survolée.

3.1.1.1 Représentation d'une classe

Une classe est caractérisée par son nom, un ensemble d'attributs, un ensemble d'opérations et par ses relations avec les autres classes.

[3] La visibilité des attributs n'est pas présentée ici.

Syntaxe	Signification
$0..1$	Zéro ou un
$1..1$ ou 1	Un et un seul
$0..*$ ou $*$	De zéro à plusieurs
$1..*$	De un à plusieurs
$m..n$	De m à n inclus (m et $n \in \mathbb{N}$, $m \leq n$). Il s'agit du cas général.

Tableau 7 – Les différentes plages de multiplicités en UML.

Dans le formalisme UML, une classe est représentée par un rectangle qui l'identifie de façon unique. Ce rectangle est divisé en trois compartiments verticaux. Le compartiment du haut contient le nom de la classe, le compartiment du milieu contient les attributs et celui du bas les opérations. Chaque attribut est représenté par une ligne de texte selon la syntaxe suivante (les éléments entre crochets sont optionnels) :

```
<attribut> := nom_attribut [<multiplicité>] : type_attribut [= valeur_défaut]
<multiplicité> := '['multiplicité_min .. multiplicité_max']'
```

Une multiplicité [4] est composée d'un nombre entier positif dit minimal et d'un autre dit maximal, qui indiquent le nombre d'éléments différents minimal et maximal qui peuvent être associés à un objet suivant une caractérisation particulière. Dans le cas d'un attribut, la multiplicité précise le nombre de valeurs différentes possibles pour l'attribut donné que peut posséder l'objet. Le Tableau 7 précise les différentes plages de multiplicités utilisables en UML.

Notons que le compartiment des attributs, celui des opérations ou les deux peuvent être omis. Ceci ne veut pas nécessairement dire que la classe ne possède d'attributs et/ou d'opérations, même si c'est généralement le cas. Par la suite, comme nous n'utilisons pas celui des opérations, il ne sera pas représenté.

3.1.1.2 Exemple

La Figure 33 représente un diagramme de classes UML. La classe Personne est caractérisée par son nom 'Personne', par les attributs nom, prénom et âge, et par sa relation (éventuelle) conduire avec la classe Véhicule. La multiplicité [1..3] de l'attribut prénom indique qu'une instance de Personne peut posséder entre un et trois prénoms. Pour les autres attributs, la multiplicité est (par défaut) exactement de un. On rappelle que le compartiment des opérations (vide) n'est pas représenté. La plupart des autres classes sur cet exemple, comme la classe Moto, ne possèdent pas d'attribut (autres que ceux hérités, voir ci-après en Section 3.1.2). À cette classe est associé un commentaire, situé dans un rectangle au coin supérieur-droit replié. Un *commentaire* permet d'apporter, à tout élément UML, une précision qui ne peut s'exprimer avec le vocabulaire graphique UML. Dans le cas de ce commentaire, il s'agit d'une expression *OCL*, qui est discutée en Section 3.3.2. Les classes conduire, piloter, diriger et être_dirigé, sont plus précisement des *classes d'association*. Ceci est

[4] Le terme *multiplicité* a une signification proche du terme *cardinalité* employé avec *Merise*.

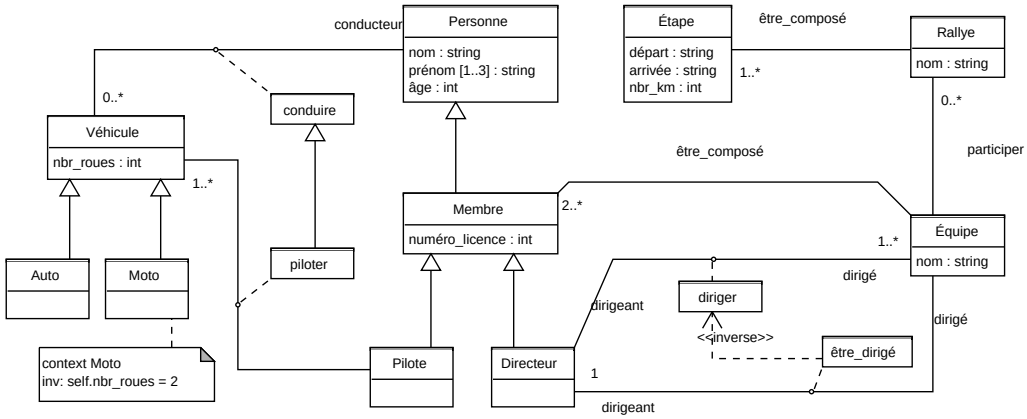


Figure 33 – Un diagramme de classes UML.

présenté dans la Section 3.1.3.

3.1.2 Généralisation

Il existe principalement deux types de relations entre classes, le lien de *généralisation* et le lien d'*association* (voir la Section 3.1.3), auxquels s'ajoute le lien de *dépendance* (voir la Section 3.1.4).

La généralisation indique une relation entre une classe dite générale (ou parent) et une classe dite spécialisée (ou sous-classe ou enfant). Dans le langage UML, comme dans la plupart des langages objets, ce lien de généralisation se traduit par le concept d'héritage qui permet la classification des classes. La généralisation est transitive. La *sous-classe* possède toutes les caractéristiques de la *classe parent* auxquelles s'ajoutent ses propres caractéristiques. Notons qu'un lien de généralisation n'est pas instanciable. En d'autres termes, une généralisation est un élément du niveau structurel d'une modélisation qui ne peut pas être le type d'un élément du niveau factuel de la modélisation.

3.1.2.1 Représentation d'une généralisation

Le symbole utilisé en UML pour représenter un lien de généralisation est une flèche en trait plein partant de la sous-classe et se terminant sur la classe parent par un triangle fermé vide.

3.1.2.2 Exemple

Sur le diagramme de classes en Figure 33, les classes Directeur et Pilote ont pour classe parent la classe Membre, qui elle-même est une sous-classe de Personne. Ainsi, les classes Directeur et Pilote possèdent quatre attributs : ceux hérités de la classe Personne auxquels s'ajoute l'attribut numéro_licence hérité de la classe Membre.

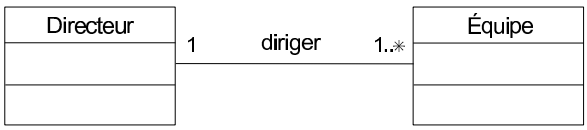


Figure 34 – Une association binaire, exprimée dans sa forme la plus simple.

3.1.3 Association

Une association est un élément structurel d'une modélisation. Elle exprime une relation multi-directionnelle entre plusieurs classes. Le plus souvent, une association est binaire et bidirectionnelle. C'est-à-dire qu'il s'agit d'une relation entre deux classes pouvant être « lue » dans les deux sens. Notons qu'une association binaire, dont les deux extrémités lient la même classe, est dite réflexive.

De même qu'une classe est une entité générique d'un ensemble d'objets, une association est une relation générique d'un ensemble de relations entre objets. Une association est instanciable dans un diagramme d'objets (voir la Section 3.2), sous forme de *liens* entre instances ayant pour type les classes participant à l'association.

Une association est une relation définie par un nom ou une classe d'association, par les classes concernées dans la relation et par les multiplicités aux extrémités de la relation. Les classes participant à une association sont appelées *classes d'extrémité* ; les extrémités d'une association sont appelées des *terminaisons d'association*. Pour une terminaison d'association donnée, on parle de *classe cible* pour désigner la classe d'extrémité située du même côté que cette terminaison, et *classe source* pour la classe d'extrémité située à une autre terminaison de la même association (à l'autre terminaison, pour une association binaire).

3.1.3.1 Représentation d'une association

Une association peut-être représentée en UML de différentes façons suivant qu'elle soit binaire ou n-aire ($n > 2$) et/ou que sa définition soit plus ou moins complète.

Dans sa version la plus simple, une *association binaire* est représentée par un trait plein entre les deux classes qu'elle lie. Ce trait est orné d'un nom qui identifie l'association de manière unique. À chaque terminaison de l'association est donnée la multiplicité, selon la même syntaxe que pour les attributs, qui indique comment la classe d'extrémité participe à l'association. Cette multiplicité précise pour un objet du type de la classe source le nombre d'objets du type de la classe cible succeptibles d'être liés à cet objet via des assertions de l'association. Par exemple, en Figure 34, les classes Directeur et Équipe sont associées entre elles par l'association diriger. Au niveau de la terminaison d'association « du coté » de la classe Équipe, la multiplicité $0..*$ indique qu'un directeur (instance de la classe source) peut diriger de une à plusieurs équipes (instances de la classe cible). Du coté de l'autre terminaison d'association, il est précisé qu'une équipe doit être dirigée par un seul directeur.

Comme il a déjà été mentionné plus haut, une association binaire est bi-directionnelle.

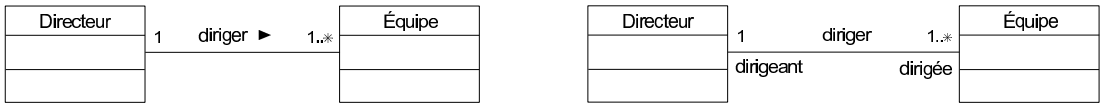


Figure 35 – Une association binaire dont le sens de lecture (à gauche) ou l'orientation sémantique (à droite) sont explicitement donnés.

Or, de par la sémantique du nom donné à l'association, une association est généralement (pour ne pas dire toujours) orientée. Si par exemple il existe une association diriger entre Équipe et Directeur, cette association n'a de sens que si elle est uni-directionnelle : si c'est « un directeur qui dirige une équipe », et non le contraire. La définition d'une association en UML peut-être complétée en l'orientant. Ceci peut-être fait de deux manières [5].

La première consiste à faire suivre (respectivement à faire précéder) le nom de l'association du symbole ► (respectivement ◄), ce qui donne un sens de lecture à l'association. Par exemple, sur la partie gauche de la Figure 35, la relation diriger se lit : « un directeur dirige une équipe » dans le sens direct (celui donné par le symbole ►), ou « une équipe est dirigée par un directeur » dans le sens indirect. Cependant l'utilisation de ce symbole reste ambigu dès lors que le trait qui représente l'association s'écarte trop d'une position horizontale ou si l'association à plus de deux classes d'extrémité.

La seconde manière pour donner une orientation à une association est de nommer les terminaisons de l'association. Ainsi, le sens (direction et sémantique) d'une association est porté par les noms de ses terminaisons. Le nom d'une terminaison d'association est appelé un *rôle*. Par exemple, en partie droite de la Figure 35 le rôle de l'association côté Directeur est dirigeant et celui côté Équipe est dirigée. Cette association, dont le nom est mis sous une forme verbale, est orientée par le genre des noms des rôles : la classe Directeur du côté du rôle dirigeant (nom commun) en est l'agent, et Équipe attenant au rôle dirigée (participe passé) en est l'objet.

Sous la forme la plus complète, une association utilise une *classe d'association*, comme avec les associations conduire et piloter en Figure 33. Une classe d'association est littéralement à la fois une classe et une association. Une classe d'association doit être vue comme une association, qui dans sa définition, bénéficie de tout le vocabulaire spécifique propre à une classe. Pour caractériser une association, en plus d'un nom, une classe d'association peut posséder des attributs ou des opérations. Elle peut aussi être en relation avec d'autres classes (d'association), par exemple avec un lien de généralisation.

Une classe d'association est représentée par une classe et par un trait plein d'association, tous deux reliés par une ligne pointillée. Par exemple, l'association diriger entre Directeur et Équipe en Figure 36, est représentée par une classe d'association. Le nom de l'association est le nom de la « partie classe » de la classe d'association. En plus de ce qui est présentée en Figure 35, la définition de l'association est complétée par l'attribut depuis de la « partie classe » de la classe d'association. Cette attribut permet de préciser depuis

[5] Les flèches de *navigation* (non présentées ici) pouvant être placées aux extrémités du trait de l'association ne signifient pas d'une orientation de l'association, mais font référence à une limitation en terme de *visibilité* (non plus présentée ici) entre classes d'extrémités.

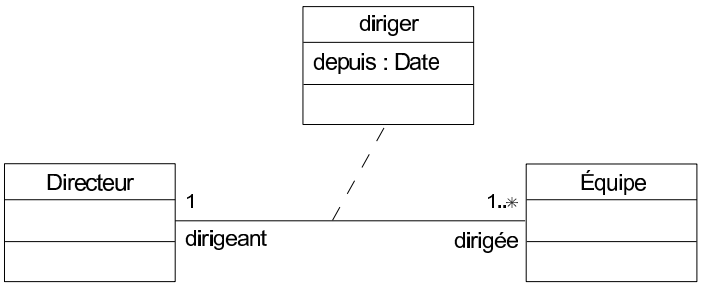


Figure 36 – Une classe d'association.

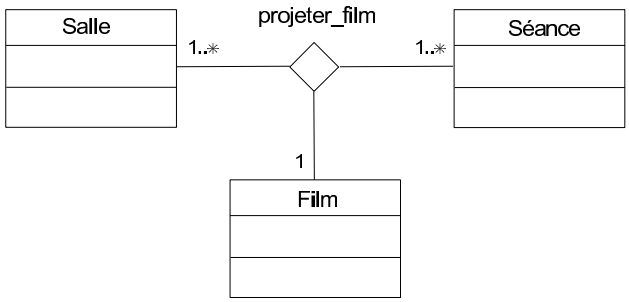


Figure 37 – Une association ternaire.

quelle date un directeur dirige une équipe. Avec une classe d'association, les multiplicités ou les rôles des terminaisons d'association de la « partie association » de la classe d'association fonctionnent bien sûr comme pour une association simple.

D'une façon générale, UML permet de représenter une association n-aire ($n \geq 2$). Dans le cas où le nombre de classes participant à l'association est strictement supérieur à deux, l'association est représentée par un losange vide d'où partent des traits pleins reliés aux classes d'extrémité. Avec une association n-aire pour $n > 2$, la multiplicité [6] à une terminaison d'association précise pour un objet du type de la classe cible le nombre d'ensembles d'objets succeptibles d'être liés à cet objet via des assertions de l'association ; un ensemble étant constitué d'une instance de chaque classe source.

Sur la Figure 37 est représentée l'association ternaire projeter_film qui associe les classes Film, Séance et Salle entre elles. La multiplicité de 1 précise que pour une salle et une séance données (instances des classes sources pour cette terminaison d'association) ne peut être projeté qu'un seul film (instance de la classe cible). Notons qu'une classe d'association peut aussi être utilisée pour une association n-aire, avec $n > 2$. Dans ce cas la ligne pointillée est entre le losange et la « partie classe » de la classe d'association.

[6] Pour les habitués du modèle *entité/relation*, notez qu'entre *UML* et *Merise*, les positions des multiplicités (appelées cardinalités en *Merise*) sont inversées de part et d'autre du nom de l'association dans le cas des associations binaires.

3.1.3.2 Association *versus* attribut

Il est intéressant de constater qu'*attributs* et *associations binaires* sont en fait très proches d'un point de vue conceptuel. Il s'agit pour un attribut de lier la classe à un ou plusieurs types primitifs, et pour une association (vue comme une relation uni-directionnelle) de la lier à une ou plusieurs autres classes. Ils sont tous deux caractérisés par un nom et une multiplicité sur le co-domaine. La Figure 38 pourrait être une formulation de la classe Personne ayant pour attributs un nom, un à trois prénom(s) et un âge, et conduisant éventuellement une voiture, et où :

- Un attribut est lié à la classe en tant qu'une association uni-directionnelle orientée de la classe vers le type de l'attribut. Le nom de cette association est celui de l'attribut.
- Le type d'un attribut est représenté de manière analogue à une classe. Ici, nous l'avons représenté par le symbole d'une classe stéréotypée par ≪datatype≫ (voir la Section 3.3.1), dont le nom est celui du type de l'attribut. Ce symbole n'est doté que d'un compartiment, car un type primitif ne possède ni d'attribut ni d'opération.

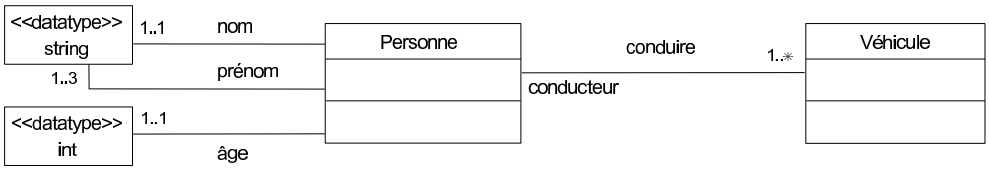


Figure 38 – Un diagramme de classes en langage, où attributs et associations sont vus de manière analogue.

Cependant, ne pas représenter les attributs et les associations de manière analogue présente un avantage majeur en terme de lisibilité du diagramme de classes. En effet, les attributs sont des caractéristiques internes à la classes, et sont « emboîtés » dans cette dernière. Ainsi, attributs et associations n'occupent pas les mêmes positions dans un diagramme de classes. De plus, le diagramme de classes peut être visualisé selon deux niveaux de « zoom » : soit on s'intéresse aux relations entre les classes et auquel cas on peut masquer visuellement les caractéristiques internes des classes (ne montrer que le compartiment supérieur des classes), soit on s'intéresse à une classe particulière et on visualise sa structure interne (tous les compartiments de la classe sont montrés).

3.1.4 Dépendance

Un lien de dépendance entre deux éléments UML de même nature (par exemple deux classes) stipule que la définition d'un élément, dit cible, dépend de l'existence d'un autre élément, dit source. Il s'agit d'une dépendance sémantique unidirectionnelle où toute modification de l'élément cible est succeptible d'entraîner une modification de l'élément source.

Figure 39 – Un lien de dépendance entre la classe PlanningSaison et la classe Rallye.

3.1.4.1 Représentation d'une dépendance

Un lien de dépendance entre deux éléments est représenté par un trait discontinu partant de l'élément source et se terminant par une pointe de flèche ouverte sur l'élément cible. Pour indiquer plus précisément de quelle manière un élément dépend d'un autre, le lien de dépendance est stéréotypé (voir la Section 3.3.1).

3.1.4.2 Exemple

Sur la Figure 39 est représenté un lien de dépendance qui part de la classe source PlanningSaison et pointe sur la classe cible Rallye. Le planning d'une saison de rallyes donnée est dépendant des rallyes qui le composent. Toute modification d'un rallye, par exemple du jour d'une étape, est succeptible de modifier le planning.

3.2 Diagramme d'objets

Un diagramme d'objets [OMG, 2003b; OMG, 2003a] est une collection d'éléments statiques, qui modélisent un fait particulier du domaine modélisé. Un diagramme d'objets fait abstraction des aspects dynamiques des connaissances factuelles modélisées sur un domaine. Cependant un diagramme d'objets peut servir, en modélisation objet, à représenter un état du domaine avant ou après un comportement dynamique de celui-ci (une interaction entre objets par exemple).

Un diagramme d'objets n'a de sens que part rapport à un diagramme de classes donné, et en respecte ses contraintes. Il est composé d'objets et de liens entre ces objets ; les *objets* sont des instances des classes du diagramme de classes associé, et les *liens* sont des instances des associations du diagramme de classes. Un diagramme d'objets peut être considéré comme une « instance » d'un diagramme de classes.

Concrètement, un *objet* est l'instance d'une ou de plusieurs classes. Il s'agit d'une entité spécifique qui est dotée d'un état et d'un nom éventuel qui la caractérisent. Un *lien* est une instance d'une association. Il a pour but de relier un objet à un autre objet, et contribue ainsi à caractériser ce premier objet. L'état d'un objet est donné par les valeurs de ses attributs. Pour chaque attribut, il peut s'agir d'une valeur de donnée, dans le cas où l'attribut est un type primitif, ou d'un autre objet, dans le cas où l'attribut est une instance d'une classe. Dans ce dernier cas la valeur de l'attribut est assimilable à un lien, dont le type est une association entre les deux classes.

3.2.1 Représentations d'un objet, d'une valeur d'attribut et d'un lien

Graphiquement, en notation UML, un objet est représenté par un rectangle qui l'identifie de façon unique. Ce rectangle est divisé en deux compartiments verticaux.

Le compartiment du haut contient le nom et le type de l'objet, selon la syntaxe suivante :

```
<compartiment_haut> := [nom_attribut : ] <type>
<type> := nom_classe (, nom_autre_classe)*
```

Le nom de l'objet n'est pas obligatoire, car c'est le rectangle associé à l'objet - le symbole visuel - qui identifie l'objet. L'objet peut être l'instance de plusieurs classes, c'est pourquoi le *type* de l'objet est donné par une liste contenant le nom de ces classes séparées par une virgule. L'ensemble du label contenu dans le compartiment du haut est souligné.

Le compartiment du bas renseigne les *valeurs des attributs* que possède l'objet et qui participe à le caractériser. Pour les différencier, chaque valeur est précédée du nom de l'attribut auquel elle fait référence, c'est-à-dire son type. Le caractère '=' permet de séparer le type de la valeur. La syntaxe du compartiement du bas est la suivante :

```
<compartiment_bas> := (<valeur_attribut>)*
<valeur_attribut> := type_attribut = valeur
```

Un *lien* est une instanciation d'une association. Un lien binaire entre deux objets se représente par un trait plein entre les deux rectangles qui représentent les deux objets. À ce trait plein est adjoint le nom de l'association dont le lien est instance ; ce nom est écrit en caractères soulignés. Un lien a pour but de caractériser en partie l'objet situé à sa première extrémité.

Contrairement à la définition d'une association, un lien ne dispose ni de multiplicité ni de rôle. Rappelons qu'une multiplicité est une contrainte définie au niveau structurel qui doit être respectée au niveau des faits, c'est-à-dire au niveau des liens entre plusieurs objets ou entre objets et valeurs. Cette contrainte indique, on le rappelle, le nombre d'instances possibles d'une même association auquel un objet peut être connecté. Les rôles ne figurent pas non plus au niveau d'un lien, dans la mesure où ils sont facilement déductibles en fonction du type des objets se situant aux extrémités du lien et en se reportant aux rôles de l'association sur le diagramme de classes.

3.2.2 Exemple

Le diagramme en Figure 40 est un exemple de diagramme d'objets défini sur le diagramme de classes en Figure 33. L'équipe TMRM est composée du pilote Alphand et du directeur Serieys. Ici, à titre d'exemple on a présenté l'individu Serieys comme étant aussi [7] une instance de la classe Pilote. Cette équipe, ayant pour nom "Team Repsol Mitsubishi Ralliart", participe au rallye dakar2007. Ce rallye est composé des étapes etap08 et etap09. L'objet

[7] Cette information n'est pas totalement erronée dans la mesure ou D. Serieys fût co-pilote de rallyes de 1978 jusqu'en 1989 (mais pas dans cette équipe bien sûr).

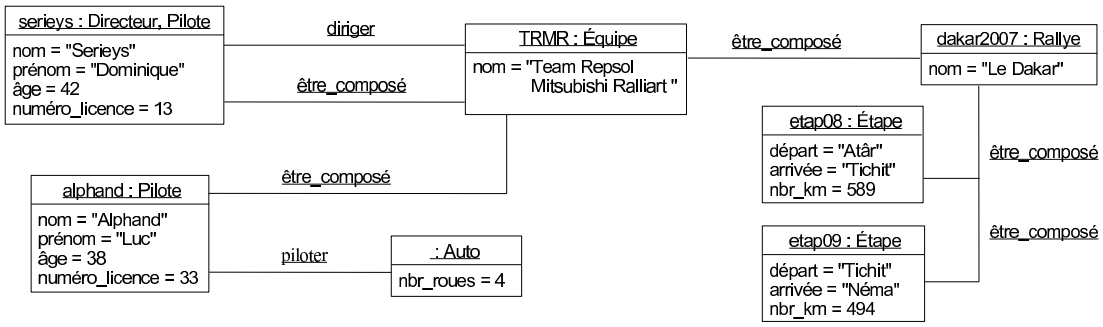


Figure 40 – Un diagramme d'objets, défini sur le diagramme de classes en Figure 33.

etap08 par exemple, est caractérisé par la valeur des attributs départ, arrivée et nbr_km qui sont respectivement "Atâr", "Tichit" et 589.

Il est intéressant de constater sur cet exemple, qu'un diagramme d'objets exprime des connaissances factuelles d'un domaine qui sont partielles. En effet, les faits peuvent être décrits au sein de plusieurs diagrammes d'objets, tous définis sur le même diagramme de classes. Ceci permet à l'utilisateur de ne représenter qu'une partie des connaissances selon ses besoins. Ici, le rallye du nom "Le Dakar" est composé d'au moins deux étapes, et l'équipe TMRM est certainement constituée de d'autres membres, comme le co-pilote d'Alphand.

Le diagramme d'objets reste en accord avec les contraintes exprimées au niveau du diagramme de classes, à savoir les types des objets aux extrémités des liens, les multiplicités et les contraintes OCL. Par exemple, le lien instance de l'association diriger est bien connecté entre un individu de type Directeur et un autre de type Équipe. De plus, cette équipe est composée d'au moins deux membres, et respecte ainsi la multiplicité $[2..*]$ exigée entre les classes Équipe et Membre.

3.3 Extensions du langage

3.3.1 Stéréotype

L'usage de stéréotypes est un mécanisme d'extensibilité du langage UML [OMG, 2003a; OMG, 2003b] . Un *stéréotype* permet à l'utilisateur de créer de nouvelles notations UML à partir de celles existantes. Ceci dans le but de mieux adapter le langage UML pour modéliser un domaine donné. Un stéréotype, appliqué à un élément UML, permet donc à l'utilisateur d'utiliser cet élément d'une manière différente. Généralement, un stéréotype permet de spécialiser l'élément de base (sans stéréotype) auquel il est appliqué.

Les éléments UML dotés de stéréotypes sont appelés des *éléments stéréotypés*. Concrètement, un stéréotype n'est autre qu'un mot, habituellement un nom commun, qui donne la signification dont l'élément stéréotypé est utilisé dans la modélisation. Un même stéréotype peut s'appliquer à un ou plusieurs types d'éléments UML, par exemple un stéréotype pourrait être utilisé sur les classes ou les associations. Un stéréotype est repré-

senté comme son élément de base, auquel est ajouté à coté du nom de l'élément le nom du stéréotype entre '≪' et '≫'.

Exemple

Prenons la Figure 38 sur laquelle on représente les datatypes, normalement présents dans la définition des attributs d'une classe, comme des classes. Ce choix est légitime dans la mesure où une classe représente un *type* pour des individus, là où un datatype est un *type* pour des valeurs de données. Ainsi, pour ce choix de modélisation, la notation pour représenter un datatype est celle d'une classe dont on a spécialisé son utilisation (en tant que *type*) avec l'emploi du stéréotype ≪Datatype≫ : string et int en sont deux exemples. La notation UML pour représenter une classe reste inchangée bien sûr.

Sur la Figure 33, le lien de dépendance entre la classe d'association diriger et la classe d'association être_dirigé est muni du stéréotype ≪inverse≫. Ce dernier permet de préciser la manière dont être_dirigé dépend de diriger : être_dirigé est l'association inverse de diriger.

Un autre exemple pourrait être d'utiliser un stéréotype ≪Table≫ sur les classes d'un diagramme de classes dans le cadre d'une modélisation de base de données.

3.3.2 Object Constraint Language

Le langage OCL - *Object Constraint Language* - constitue une extension [Booch *et al.*, 1998; Warmer et Kleppe, 1998; OMG, 2003c] du langage UML réalisée en collaboration avec la société IBM. Apparu à la version 1.1 de UML (première version UML officielle reconnue), OCL en fait partie intégrante depuis la version 1.3 en juin 1999. Une contrainte OCL permet d'affiner [8] les spécifications d'une modélisation là où le vocabulaire visuel de UML est impuissant en terme de représentation. Il s'agit d'un langage d'expression de contraintes qui est adapté aux diagrammes UML, en particulier au diagramme de classes. Ce langage a initialement été défini suivant une approche semi-formelle utilisant des descriptions textuelles en anglais, une grammaire munie d'une syntaxe souple et en s'appuyant sur de nombreux exemples afin d'illustrer le sens porté par les expressions OCL. Depuis le travail de thèse de M. Richters [2002], repris en annexe A dans [OMG, 2003c], OCL dispose d'une sémantique formelle proche de la logique classique.

Concrètement, OCL décrit textuellement des contraintes sous forme de pseudo-codes. Ces contraintes s'expriment essentiellement en terme d'*invariants* sur les attributs, et de *pré et post-conditions* pour les opérations. Un constat immédiat peut être fait : OCL ajoute certes de l'expressivité au langage UML, mais il le fait de façon textuelle alors que la force de UML réside dans sa faculté à représenter des connaissances de façon visuelle. OCL a donc était conçu avec une syntaxe relativement simple pour rester accessible à l'utilisateur et être interprété par des outils logiciels (voir la Section 3.4).

En UML, une contrainte OCL est insérée dans un commentaire UML, éventuellement lié à l'élément concerné par la contrainte (auquel cas, le mot-clé context est optionnel).

[8] Depuis la version 2.0 de UML, OCL permet aussi de définir le méta-modèle de UML [OMG, 2003a].

Notons bien cependant que OCL est conçu pour spécifier des contraintes mais non pour calculer leurs vérifications. D'une manière générale, les expressions OCL sont difficilement exploitables par des outils du fait de la grande expressivité de ces expressions au détriment de l'aspect temps calculatoire, donnant aux raisonnements sur ces contraintes un caractère indécidable [Berardi *et al.*, 2005].

Exemple

Sur la Figure 33, une contrainte OCL complète la définition de la classe Moto. Cette contrainte est la suivante :

```
context Auto
inv: self.nombre_roues = 2
```

Elle précise que pour la classe Auto (*cf.* le mot clé context), l'attribut nombre_roues de toute instance de cette classe (*cf.* le mot clé self désignant l'individu courant) doit toujours avoir la valeur 2, c'est-à-dire de manière *invariable* (*cf.* le mot clé inv).

3.3.3 Sérialisation en XMI

Le format XMI - *XML Metadata Interchange* - [OMG, 2002] est un standard [9], créé par l'OMG, pour l'échange d'informations de métadonnées basé sur XML. Il s'agit concrètement d'un procédé de sérialisation d'éléments MOF [10].

Le langage MOF - *Meta-Object Facility* - (un autre standard [11] de l'OMG) est un langage auto-défini pour représenter et manipuler des méta-modèles. En fait, MOF sert de socle commun pour définir les standards de l'OMG : « *The MetaObject Facility Specification is the foundation of OMG's industry-standard environment* ».

Le langage UML étant conforme au standard MOF, il en résulte qu'une modélisation en UML peut être sérialisée en XMI. La sérialisation d'éléments UML en XMI est définie depuis la version 1.3 de UML.

Le format XMI ne constitue pas à proprement parler d'une extension de UML, mais offre un format standard de stockage et d'échange de connaissances modélisées dans le modèle UML.

3.4 Outils

Il existe de nombreux outils logiciels de modélisation UML, mais aucun ne respecte strictement une version de UML - particulièrement UML 2.0 - et beaucoup de ces outils introduisent des notations particulières non conformes. Il en est de même pour la gestion du format XMI par les outils UML. Cependant, tous ces outils supportent le diagramme de classes qui est le cœur d'une modélisation UML ; le diagramme d'objets n'est souvent supporté que partiellement.

[9] XMI est normalisé ISO/IEC 19503 :2005 (`http://www.iso.org`)

[10] « MOF 2.0/XMI Mapping, version 2.1.1 », `http://www.omg.org/spec/XMI/`

[11] `http://www.omg.org/mof/` ; MOF est normalisé ISO/IEC 19502 :2005 (`http://www.iso.org`)

Si UML propose différents types de diagrammes pour représenter différents aspects et vues des connaissances d'un domaine, rappelons qu'il ne constitue - dès son origine - ni une méthodologie de modélisation ni un environnement pour raisonner sur les connaissances modélisées. C'est pourquoi la majorité des outils UML se consacrent « simplement » à l'édition de diagrammes.

Depuis les débuts de UML, les besoins des utilisateurs, notamment les programmeurs dans des langages objets, ont évolué. Ces développeurs ont souhaité que le temps consacré à éditer un diagramme UML, essentiellement un diagramme de classes, ne soit pas « du temps perdu ». Ainsi, en plus de son aspect visuel pour concevoir une modélisation, des éditeurs UML disposent de mécanismes permettant d'une part la génération de code et d'autre part la génération de documentation dans ces langages (principalement en C++ et en Java) à partir de diagramme de classes.

Concernant OCL, le fait qu'il ne soit pas prévu pour l'écriture de code objet et que l'accent soit actuellement mis sur cet aspect, fait que très peu d'outils UML sont capables de vérifier des contraintes OCL sur des connaissances modélisées en UML.

Rational Rose et Together

Les outils les plus connus et aboutis dans le monde du développement objet sont les deux outils commerciaux Rational Rose [12] de la société IBM et Together [13] de Borland.

Ces deux outils prennent en charge les différents types de diagrammes UML 1.x pour Rational Rose et 2.0 pour Together. La division « Rational Software » de IBM fournit désormais l'environnement Rational Software Architect (RSA) qui supporte les diagrammes UML 2.0 et est basé sur Eclipse (voir ci-dessous). Depuis 2006, Together est lui aussi basé sur le projet GMF de Eclipse.

En plus de la génération de code dans des langages de programmation objet, de la conception inverse (reverse engineering) de modélisation UML à partir de code et de l'import/export en XMI, ces deux outils font partie des rares à prendre en compte des contraintes OCL. Ces dernières sont exploitées en terme d'audit sur les diagrammes (diagramme de classes essentiellement) en cours d'édition. Sans totalement être conformes aux spécifications OCL, ces outils permettent certaines vérifications comme la détection de cycles d'héritage dans une hiérarchie de classes.

ArgoUML et Poseidon/Apollo

L'outil ArgoUML [Boger *et al.*, 2008] est un des outils libres (sous licence BSD) les plus aboutis. Il supporte notamment les neuf types de diagrammes UML 1.4, l'import/export en XMI, l'export des diagrammes sous forme d'images (GIF, PNG, EPS, ...), la génération de code dans des langages objets, de la conception inverse (reverse engineering) de modélisation UML à partir de code et de fichiers binaires exécutables. ArgoUML supporte

[12] `http://www-306.ibm.com/software/rational/uml/`
[13] `http://wwww.borland.com/together/`

aussi OCL. Pour cela, le Dresden OCL toolkit de l'université de Dresde en Allemagne [Bräuer et Demuth, 2008] est utilisé conjointement avec ArgoUML.

L'outil Poseidon [14] est un dérivé commercial de ArgoUML. Cependant, certaines divergences de fonctionnalités, de plus en plus grandes, existent actuellement entre les deux outils. La société Gentleware qui édite Poseidon produit aussi l'outil Apollo [15] pour une compatibilité de son environnement UML avec Eclipse.

Eclipse

L'environnement bien connu de développement Eclipe [16] propose un ensemble de bibliothèques Java, dont EMF - *Eclipse Modeling Framework* - [Eclipse community, 2008]. EMF est composé de bibliothèques de modélisation et de génération de code Java. Depuis un modèle de spécifications décrit en XMI, EMF fournit un moteur de génération et de vérification de classes Java. Notons que EMF fournit un fondement à l'interopérabilité entre outils UML. En effet de plus en plus d'outils sont conforme à EMF, comme l'outil RSA et Together.

Le projet GMF - *Graphical Modeling Framework* -, basé sur EMF, est dédiée au développement des éditeurs graphiques. Il est donc adapté à la mise en place de logiciels pour l'édition de diagrammes UML.

Autres outils

Citons quelques autres outils pour l'édition et l'exploitation de diagrammes UML.

- BOUML [17] est une suite d'outils libres (licence GNU GPL) et multi-plateformes (Unix/ Linux/ Solaris, MacOS X et Windows) pour UML 2. Codé en C++, cet outil est très réactif dans l'édition de diagrammes UML et très rapide dans la génération de code.
- Umbrello UML Modeller [18] est un outil KDE libre et multi-plateformes d'édition de diagrammes UML et de génération de code. On peut cependant regretter la non prise en charge des diagramme d'objets par Umbrello.
- La plateforme académique NEPTUNE [Cruellas *et al.*, 2003], basée sur les standards d'échange XMI et de transformation XSL, a pour but de valider des modélisations UML 1.x à partir de spécifications en OCL.
- L'environnent de validation OCL HOL-OCL [Brucker et Wolff, 2002] utilise une instance du démonstrateur logique - Higher-order Logic (HOL) - Isabelle [19] pour vérifier des contraintes OCL.

[14] `http://www.gentleware.com/fileadmin/media/archives/userguides/`
[15] `http://www.gentleware.com/apollo.html`
[16] `http://www.eclipse.org/`
[17] `http://bouml.free.fr`
[18] `http://uml.sourceforge.net/documentation.php`
[19] développé à l'université de Cambridge et l'université technique de Munich ; `http://isabelle.in.tum.de/`

– Le travail présenté dans [Akehurst et Bordbar, 2001] propose d'utiliser OCL pour interroger des éléments de diagrammes de classes UML.

3.5 Conclusion

UML est le langage de modélisation objet de référence. UML définit des notations, essentiellement graphiques, pour définir différents types de diagrammes qui représentent différentes vues et aspects d'un domaine à modéliser. Tout comme le modèle des GCs, la modélisation de connaissances en UML se fait selon une démarche intensionnelle. Le diagramme de classes modélise les connaissances structurelles d'un domaine, il est de plus le cœur de la modélisation objet vis-à-vis des autres types de diagrammes. Le diagramme d'objets est composé d'instances des éléments du diagramme de classes et constitue les connaissances factuelles du domaine modélisé.

Le principal avantage de UML est de disposer de notations très largement visuelles, intuitives à manipuler pour l'humain, et très connues. Initialement, UML ne dispose pas de capacité d'inférence. Ce n'est d'ailleurs toujours pas un objectif recherché par ses créateurs. Cependant, couplé à UML depuis la version 1.3, OCL propose une sémantique formalisant logiquement UML. OCL fournit aussi un mécanisme pour exprimer des contraintes d'invariance et des règles sur les modélisations UML. Cependant, les expressions OCL sont difficilement exploitables en général par des outils du fait de la grande expressivité de ces expressions au détriment de l'aspect temps calculatoire.

Chapitre 4

Transition de modèles appliquée aux notions de *classe* et de *datatype*

Afin d'utiliser conjointement les modèles OWL, GCs et UML, nous établissons au cours des Chapitres 4, 5 et 6 l'ensemble des notions considérées dans notre système de connaissance ainsi que les règles indiquant comment sont instanciées ces notions dans chacun de ces trois modèles pour modéliser des connaissances.

Ces notions sont issues de celles présentes dans les modèles de connaissances OWL, GCs et UML. Elles ont été choisies parmi les notions communes à ces trois modèles autorisant ainsi leur utilisation conjointe auxquelles s'ajoutent des notions propres à seulement un ou deux modèles venant en complémentarité de ces notions communes. L'expressivité de notre système de connaissance n'est cependant pas la somme des expressivités des trois modèles. En effet, nous ne considérons pas certaines notions jugées trop « éloignées » des autres notions pour leur mise en synergie car très spécifiques à un modèle donné, comme par exemple en UML les notions d'interface ou de méthode propres à la programmation orientée objet.

Lors de leurs instanciations, certaines notions peuvent être considérées dans une modélisation soit comme des connaissances à déduire soit comme des connaissances à vérifier. Par exemple, pour exprimer la connaissance « tout directeur sportif dirige une équipe », le système face à un directeur sportif peut soit déduire qu'il dirige une équipe soit vérifier qu'il dirige bien une équipe. Ce choix dépend de l'utilisateur, qui agit en tant qu'expert pour la modélisation d'un domaine donné. Cette dimension utilisateur dans le système de connaissance sera par la suite mentionnée sous l'appellation *besoins utilisateur*, prenant ainsi en compte les exigences ou l'expérience de l'utilisateur.

Dans ce chapitre, nous présentons les *règles d'instanciation de notions* concernant les notions de *classe* et de *datatype*. Ces notions constituent, lors de leurs instanciations, les entités génériques d'une modélisation. Les instances de ces notions appartiennent donc au niveau structurel de la modélisation. La première section définit d'une part les notions de *classe* et de *datatype* telles qu'elles sont considérées au sein du système de connaissance. D'autre part, cette section présente les règles d'instanciation pour la notion de datatype. La deuxième section présente les sous-notions d'héritage, de disjonction et d'équiva-

lence, qui constituent des *axiomes de classes* permettant la construction de classes à partir de classes existantes. La troisième section s'interesse à la définition de classe dites *matrices*. Ces classes matrices peuvent être définies indépendamment d'autres classes ; elles peuvent aussi être utilisées pour la construction d'autres classes en se servant d'axiomes de classes.

Notons que dans la suite de cette thèse, nous utiliserons par abus de langage le terme associé à une notion pour faire référence à une instance de cette notion. Par exemple, nous parlerons d'une « classe » pour désigner une « instance de la notion de classe ». En contrepartie et afin d'éviter toute ambiguïté, nous emploierons explicitement le mot « notion » lorsque nous souhaiterons parler d'une notion et non d'une de ses instances.

Sommaire

4.1 Les notions de *classe* et *datatype* en OWL, Graphes Conceptuels et UML

4.1.1 Classe

Les *classes* - ou plus exactement des instances de la notion de classe - sont définies au niveau structurel d'une modélisation. Cette notion est considérée dans notre système de connaissance de la manière suivante.

Définition 4.1 (classe) *Une* classe *est une entité générique possédant des caractéristiques communes à un ensemble d'entités spécifiques, appelées* individus*. Ces caractéristiques, dites caractéristiques de la classe, constituent l'*intension *de la classe, et ces individus constituent l'*extension *de la classe.*

Une classe forme le (ou un [1]) *type* des individus dans son extension, et chacun de ces individus *a pour type* la classe. Ces individus sont aussi appelés les *instances* ou encore les *assertions* de la classe. D'après la définition ci-dessus, l'élaboration d'une classe correspond ainsi soit à la caractérisation de son intension, soit à l'énumération exhaustive des individus qui composent son extension, et éventuellement au deux. Dans le premier cas, la classe agit comme un « moule » ou « gabarit » permettant la création des instances de la classe .

La Figure 41 indique comment est identifiée une classe dans chacun des trois modèles du système de connaissance. En OWL [2], nous considérons une instance de la notion de classe par une instance de la ressource owl:Class (à gauche de la figure). La classe peut alors être identifiée à l'aide d'un attribut rdf:about (ou rdf:ID) qui a pour valeur l'identifiant de la classe, c'est-à-dire son nom. Dans le modèle des GCs, une classe est modélisée par un type de concept, dont l'identifiant est celui qui identifie la classe (à droite de la figure). Nous verrons en Section 4.1.2 que les types de concepts ne correspondent pas tous à des classes. Dans le modèle UML, une instance de la notion de classe est modélisée par une classe UML (en haut de la figure). Cependant, nous ne considérons que l'aspect statique d'un diagramme de classes UML. Ainsi, une classe UML dans notre système de connaissance ne sera pas caractérisée par des méthodes. C'est pourquoi la modélisation UML d'une classe est un rectangle composé de deux compartiments, celui du haut avec le nom de la classe qui l'identifie et celui avec les attributs - *slots* - de la classe, mais dépourvu du compartiment du bas des méthodes. De plus, certaines notions représentables en UML mais propres à la programmation orientée objet, comme la visibilité ou les classes abstraites, ne sont pas prises en compte.

Notons qu'une classe qui ne possède pas de nom est dite *anonyme*. Nous verrons au Chapitre 8 que d'un point de vue implémentation du système de connaissance, tout

[1] Dans le cas d'un individu multi-typé, c'est-à-dire un individu qui est l'instance de plusieurs classes.

[2] Rappelons que nous utilisons la « version » XML de RDF (voir Chapitre 2) et de façon étendue de OWL aussi ; de ce fait nous faisons l'amalgame entre la syntaxe d'une balise XML et sa sémantique RDF sous-jacente.

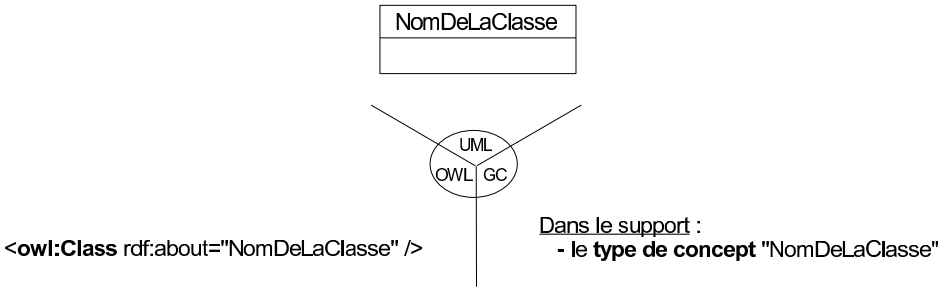


Figure 41 – Une classe - ou instance de la notion de classe - en OWL, GCs et UML.

élément d'une modélisation possède un identifiant même si celui-ci peut être masqué à l'utilisateur.

4.1.2 Datatype

Les *datatypes* sont des éléments du niveau structurel d'une modélisation.

Définition 4.2 (Datatype) *Un* datatype *est une entité générique désignant un ensemble de* données. *Cet ensemble, non nécessairement fini et établi de manière axiomatique, constitue l'extension de ce datatype.*

Un datatype, littéralement un « type de données » ; il constitue donc un *type* pour un ensemble de données, et chacune de ces données *a pour type* le datatype. La définition d'un datatype est généralement liée à l'encodage des données dans son extension. Le datatype permet ainsi de déterminer de quelle « nature » est une donnée. Par exemple, la donnée 42 sera interprétée selon son type comme un entier ou bien une chaîne de caractères.

Les classes et les datatypes peuvent sembler relativement proches d'un point de vue conceptuel. En effet, classes et datatypes sont tous deux des types d'ensembles de ressources. De plus, certains mécanismes de définitions de classes peuvent être adaptés à la définition de datatypes, comme l'organisation en hiérarchies (du moins en UML et XML Schema). La différence est faite d'un point de vue sémantique au niveau assertionnel des connaissances. Une donnée est considérée comme une simple information brute, tandis qu'un individu est détenteur d'une caractérisation intrinsèque (portée par des attributions) et peut être en liaison avec d'autres individus (voir le Chapitre 6 pour les notions de *lien* et d'*attribution*).

C'est cette différence sémantique qui nous a amenés à faire une séparation entre l'ensemble des classes et l'ensemble des datatypes au sein du système de connaissance. Cette séparation existe d'ailleurs en UML, et est nécessaire en OWL DL. La plupart des outils logiciels, dédiés à ces modèles, possèdent un ensemble de datatypes prédéfinis. S'il est possible d'ajouter de nouveaux datatypes à ces derniers, comme indiqué en Figure 42 pour UML et OWL, cela n'est généralement pas préconisé car leur interprétation n'est pas toujours prise en compte. Un nouveau datatype peut être ajouté en OWL avec le mot clé rdfs:Datatype, et en UML avec une classe stéréotypée par le stéréotype ≪datatype≫ prédéfini

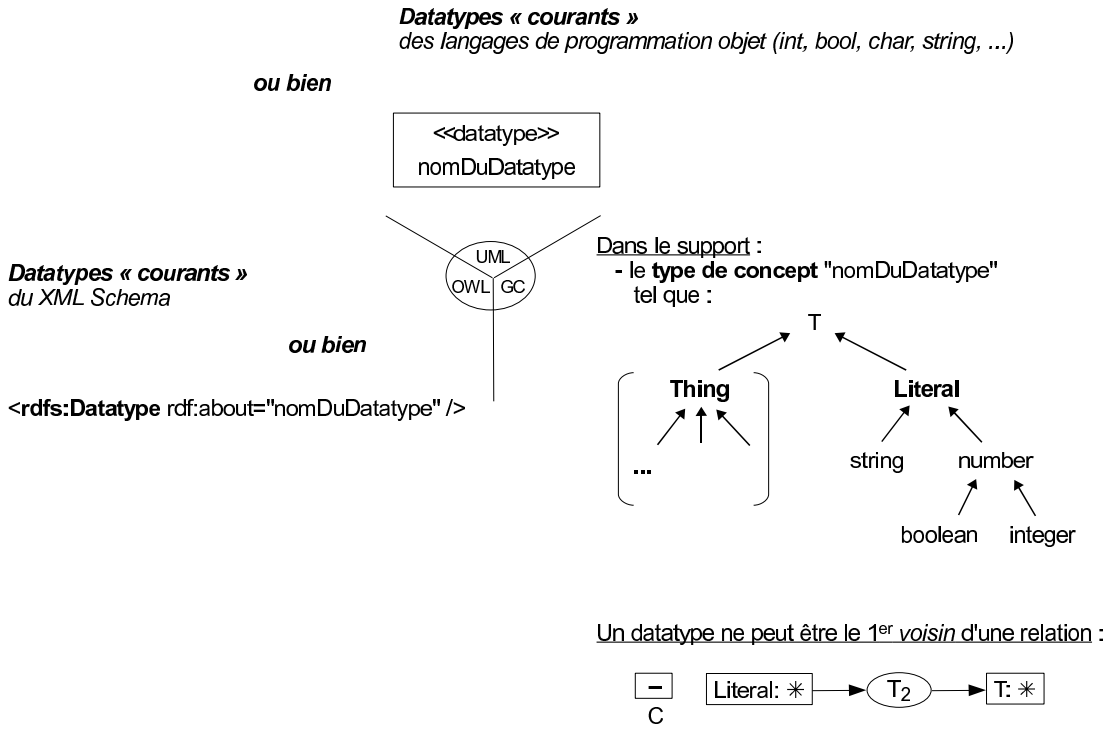


Figure 42 – Gestion des *datatypes*.

en UML ([OMG, 2003a], sections 10.1 et 11.5). Dans un environnement dit « tout objet », comme Smalltalk[3], les datatypes n'existent pas ; seules les classes existent. Ils peuvent cependant être simulés par des classes, simplifiées à l'extrême, portant uniquement le nom de ces datatypes.

Si le modèle de base des GCs ne gère pas les datatypes, le modèle des GCs considéré dans cette thèse et que nous avons présenté en Section 1.2.4 remédie à cette limite. Rappelons que pour séparer de façon explicite les types de concepts « à valeur » de *classes* de ceux « à valeur » de *datatypes*, nous avons introduit deux types de concepts particuliers : le type de concept Thing et le type de concept Literal. Le type de concept Thing représente la classe la plus générale, tandis que Literal représente le datatype le plus général. La Figure 42, en partie droite supérieure, reprend cette séparation des types de concepts représentant des classes (Thing et ses sous-types) de ceux représentant des datatypes (Literal et ses sous-types). Notons que ces deux types de concepts particuliers sont des sous-types du type de concept $\top$, le plus général en graphes conceptuels. Ceci nous permettra par la suite de pouvoir traiter si nécessaire classes et datatypes conjointement ; un peu à la manière de RDF où tout est une « sorte-de » rdf:Resource. Les datatypes font partie intégrante du modèle OWL et du modèle UML, ainsi l'utilisateur ne pourra pas, dans une modélisation, représenter un datatype comme étant le domaine d'une relation ou d'un slot (voir le Chapitre 5 pour les notions de *relation* et *slot*). Cette interdiction générique

[3] `http://www.smalltalk.org/main/`

au niveau structurel de la modélisation stipule au niveau factuel qu'une donnée ne peut être le premier argument d'une assertion de relation ou d'une assertion de slot (voir le Chapitre 5 pour les notions d'assertions de relation ou de slot). Cette interdiction est définie pour le modèle des GCs par la contrainte négative C en partie droite inférieure de la Figure 42. Cette contrainte a bien une portée générique dans la mesure où elle possède uniquement des sommets concepts génériques, ce qui l'élève au niveau structurel d'une modélisation.

■ **Datatype énuméré**

Les modèles UML et OWL offrent la possibilité de définir des datatypes dits *énumérés*. Un datatype énuméré est un type pour un ensemble fini de données prédéfinies. Ces données constituent de manière exhaustive celles composant l'extension du datatype en question ; elles définissent donc le datatype. La Figure 43 indique comment introduire le datatype énuméré feuTricolore, qui se définit comme étant le type des données vert, orange ou rouge uniquement.

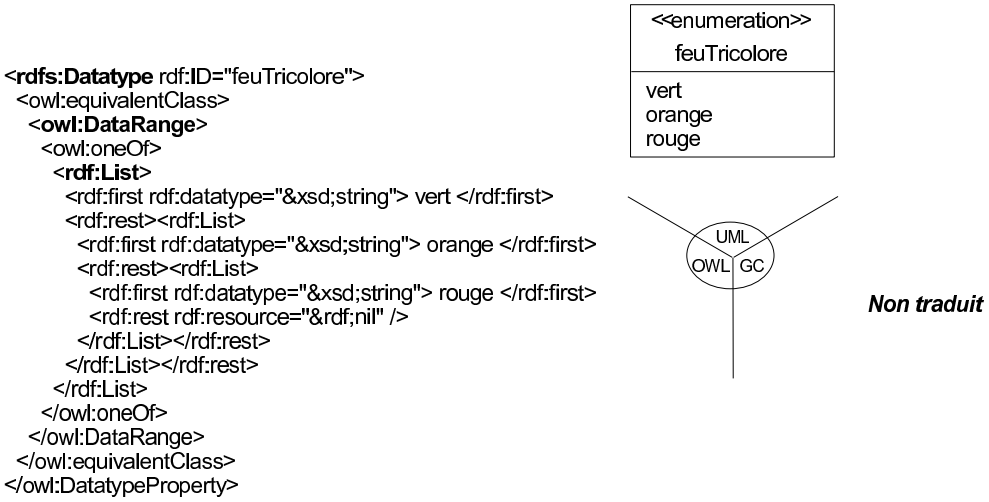


Figure 43 – Datatype *énuméré*.

Le modèle UML utilise pour définir un datatype énuméré une classe stéréotypée par le stéréotype UML prédéfini ≪enumeration≫ ([OMG, 2003a], sections 10.3 et 11.5). Le compartiment des « attributs » de cette classe stéréotypée regroupe les données possibles du datatype. Remarquons que cette localisation des données (appartenant au niveau factuel d'une modélisation) dans le compartiment des attributs (issus du niveau structurel de la modélisation) peut préter à confusion dans l'interprétation du datatype énuméré. Cependant, comme un datatype ne possède pas d'attribut, la convention est de se servir de ce compartiment de la sorte.

Le modèle OWL se sert de certains mécanismes de construction de classe pour définir des datatypes énumérés, comme les propriétés owl:equivalentClass et owl:oneOf. Cependant, d'un point de vue purement syntaxique, l'élément rdf:List est utilisé afin d'établir l'exten-

sion d'un datatype énuméré plutôt que l'idiome rdf:parseType='Collection' employé notamment avec les classes (énumérées).

Le modèle des GCs ne permet pas de modéliser partiellement des datatypes énumérés. En effet, dans le modèle des GCs considéré (Chapitre 1) une donnée ne peut être définie qu'au sein d'une graphe conceptuel. Au niveau du support, un sommet concept représentant un datatype ne peut donc pas être défini par énumération.

4.2 Axiomes de classes

Nous désignons par *axiomes de classes* des mécanismes pour l'élaboration - on parlera aussi de construction - d'une classe à partir d'une ou plusieurs autres classes. Selon l'axiome de classse utilisé, il est dit *partiel* si cette (ces) autre(s) classe(s) constitue(nt) une condition nécessaire pour *décrire* C, ou dit *complet* s'il s'agit d'une condition nécessaire et suffisante pour *définir* C. Plusieurs axiomes de classes peuvent être combinés dans la construction d'une classe (voir par exemple la Section 4.3.2.2).

4.2.1 Axiomes partiels de classes

4.2.1.1 Héritage

Les trois modèles de connaissances, OWL, GCs et UML, offrent la possibilité d'organiser de manière taxonomique les classes d'une modélisation, et ainsi de former une *hiérarchie* de classes.

Définition 4.3 (Héritage) *La relation d'*héritage *(ou de* subsomption*) est une relation de spécialisation. On dit d'une classe C_1 qu'elle hérite d'une classe C_2 si C_1 est plus spécialisée que C_2.*

La classe la plus spécifique est appelée la *sous-classe* ou le *subsumé*, et la classe la plus générale est appelée la *sur-classe* ou le *subsumant* ; la sous-classe *est subsumée* par la sur-classe et que la sur-classe *subsume* la sous-classe. Une relation d'héritage entre deux classes indique donc que les caractéristiques entrant dans la définition de la sur-classe font partie des caractéristiques de la sous-classe, ainsi que l'extension de la sous-classe est un sous ensemble de l'extension de la sur-classe. Il en résulte d'une part que la sous-classe C_2 est interprétée comme une « sorte-de » la sur-classe C_1. Il s'agit d'un mécanisme de construction de classe permettant de décrire une classe en fonction d'une autre classe : les caractéristiques de C_1 constituent des conditions nécessaires mais non suffisantes pour définir C_2.

Propriété 1 *Une classe C_2 hérite d'une classe C_1 si les individus dans l'extension de C_2 sont aussi dans l'extension de C_1.*

Propriété 2 *La relation d'héritage entre deux classes est transitive.*

La Figure 44 précise sur un exemple comment décrire la classe Auto comme étant une sous-classe de la classe Véhicule.

En OWL, nous faisons correspondre à la notion de sous-classe la propriété rdfs:subClassOf. Ainsi, pour modéliser une subsomption entre deux classes, le modèle OWL utilise une instance de la propriété rdfs:subClassOf qui constitue le prédicat d'un triplet RDF dont le sujet est la sous-classe et l'objet est la sur-classe.

Dans le modèle des GCs, selon les besoins de l'utilisateur, la connaissance « Auto est une sous-classe de Véhicule » est modélisée et donc interprétable soit par une connaissance à déduire soit par une connaissance à vérifier. Dans le premier cas, il s'agit de déduire à partir d'un individu de type Auto qu'il est aussi de type Véhicule. L'ordre partiel entre les types de concepts dans le support répond à ce cas. Notons qu'une règle dans le modèle des GCs aurait pu être utilisée pour arriver à l'expression de ce premier cas, mais nous préférons tirer profit des relations de subsomption présentes dans le support. Dans le second cas, il s'agit pour tout individu de type Auto de vérifier qu'il est aussi de type Véhicule. La contrainte positive C répond à ce second cas de modélisation.

Pour le modèle UML, le lien de généralisation permet l'organisation hiérarchique des classes.

Nous disposons dans notre système de connaissance de deux classes prédéfinies. L'une représente la classe la plus générale parmi l'ensemble des classes existantes dans toute modélisation. L'autre représente la classe la plus spécifique ; elle correspondra à un type « absurde » pour les individus d'une modélisation. Dans le modèle OWL, ces deux classes sont respectivement assimilées à la classe owl:Thing et à la classe owl:Nothing. Pour le modèle des GCs, nous définisons le type de concept Thing comme cela a déjà été mentionné en Figure 42, et le type de concepts Nothing. Ainsi, Thing (respectivement Nothing) sera le type de concept le plus général (respectivement le plus spécifique) parmi les types de concepts représentant des classes. Sur le même principe, nous introduisons et considérons dans modèle UML le la classe Thing comme étant la plus générale de toutes les classes d'une modélisation et la classe Nothing comme étant la plus spécifique par héritage direct ou indirect.

4.2.1.2 Classes disjointes

La notion disjoint, tout comme l'héritage, est un mécanisme de construction de classes permettant de décrire une classe en fonction d'une autre classe.

Définition 4.4 (Classes disjointes) *Deux classes* disjointes *ne peuvent pas avoir d'instance commune.*

Propriété 3 *Une classe C_1 est disjointe d'une classe C_2 si les individus dans l'extension de C_1 ne sont pas dans l'extension de C_2.*

Propriété 4 *La relation de disjonction entre deux classes est symétrique.*

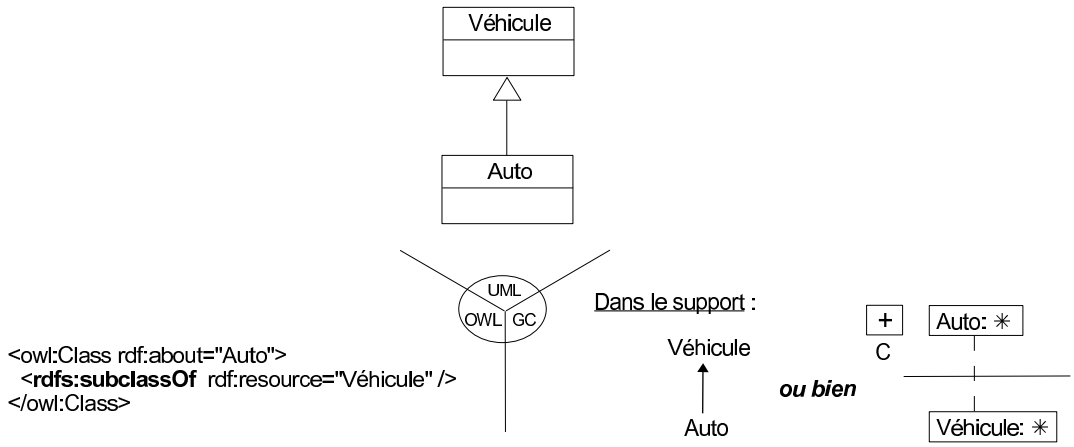


Figure 44 – La classe Auto est une sous-classe de la classe Véhicule.

Prenons l'exemple en Figure 45 qui précise que la classe Directeur est disjointe de la classe Pilote. Ainsi, d'après la Propriété 3, la non appartenance d'un individu à l'extension de Pilote est une condition nécessaire - mais non suffisante - à son appartenance à l'extension de Directeur.

Avec le modèle OWL, pour spécifier que la classe Directeur est disjointe de la classe Pilote, une instance de la propriété owl:disjointWith est utilisée comme prédicat d'un triplet RDF entre le sujet Directeur et l'objet Pilote.

Dans le modèle des GCs, nous utilisons l'extension des *types concepts bannis* pour indiquer que deux classes sont disjointes l'une de l'autre. Le type de concept conjonctif {Directeur,Pilote} est donc un type de concept conjonctif banni.

En UML, nous avons introduit [4] le stéréotype ≪disjointWith≫ que nous appliquons sur un lien de dépendance. Ce lien de dépendance stéréotypé indique que la classe source de ce lien dépend de la classe destination pour se décrire, et ceci par disjonction. Selon les besoins utilisateur, la classe Directeur disjointe de la classe Pilote peut être considérée comme une connaissance à vérifier. Dans ce cas, la contrainte OCL suivante est introduite :

```
Context Directeur
inv: not self.oclIsKindOf(Pilote)  ou bien  inv: Pilote.allInstances().excludes(self)
```

4.2.2 Axiomes complets de classes

4.2.2.1 Équivalence

Une classe peut être définie comme étant *équivalente* à une autre classe.

Définition 4.5 (Équivalence) *Deux classes* équivalentes *constituent simultanément un type pour chacune de leurs instances.*

[4] Tous les stéréotypes utilisés dans la Partie 2 de cette thèse (à l'exception de «datatype» et «enumeration») sont des propositions que nous faisons pour représenter en UML certains notions de modélisation. Ils ne font donc pas partie du langage UML, mais en constituent une extension.

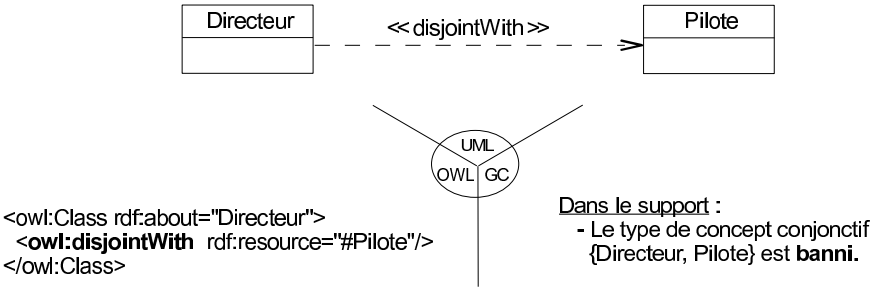


Figure 45 – La classe Directeur est disjointe de la classe Pilote.

Propriété 5 *Une classe C_1 est* équivalente *à une classe C_2 si et seulement si les individus dans l'extension de C_2 sont dans l'extension de C_1 et les individus dans l'extension de C_1 sont dans l'extension de C_2.*

En d'autres termes, les classes C_1 et C_2 partagent la même extension.

Propriété 6 *La relation d'équivalence entre deux classes est symétrique.*

L'équivalence constitue une condition nécessaire et suffisante pour définir une classe. Cependant, dire que deux classes sont équivalentes ne veut pas dire qu'elle sont identiques du point de vue de leurs intensions, mais qu'elles ont la même extension. En d'autres termes, deux classes équivalentes partagent le même ensemble d'individus, mais pas nécessairement les mêmes caractéristiques pour se définir.

L'exemple en Figure 46 définit la classe Auto comme étant équivalente à la classe Voiture.

En OWL, une définition par équivalence se fait à l'aide de la propriété owl:equivalentClass associant les deux classes équivalentes.

Avec le modèle des GCs, selon les besoins de l'utilisateur, cette équivalence est modélisée soit par l'ordre partiel [5] Auto $\leq$ Voiture dans le support et par la règle R_2, soit par les contraintes positives C_1 et C_2. Le premier choix permet de déduire que toute Auto est un Voiture (ordre partiel), et que tout Voiture est une Auto (règle R_2). Le second choix permet de vérifier que toute Auto doit aussi être un Voiture (contrainte C_1), et que tout Voiture doit aussi être une Auto (contrainte C_2). Le nouveaux symbole introduit $\boxed{\Rightarrow\!\!/_{+}}$ signifie que le couple de graphes associé est - en fonction des besoins utilisateur - soit interprété comme une règle (symbolisée par $\boxed{\Rightarrow}$) soit comme une contrainte positive (symbolisée par $\boxed{+}$) dans le modèle des GCs.

Dans le modèle UML, nous utilisons un lien de dépendance stéréotypé par ≪equivalentClass≫ entre les deux classes équivalentes. Selon les besoins utilisateur, l'équivalence entre les deux classes peut être considérée comme une connaissance à vérifier. Dans ce cas, la contrainte OCL suivante est ajoutée :

```
(=>)  Context Auto                          (<=)  Context Voiture
      inv: self.oclIsKindOf(Voiture)              inv: self.oclIsKindOf(Auto)
```

[5] Une règle aurait pu être utilisée pour exprimer le fait que « toute Auto est une Voiture », mais là encore nous exploitons les relations de subsomption présentes dans le support.

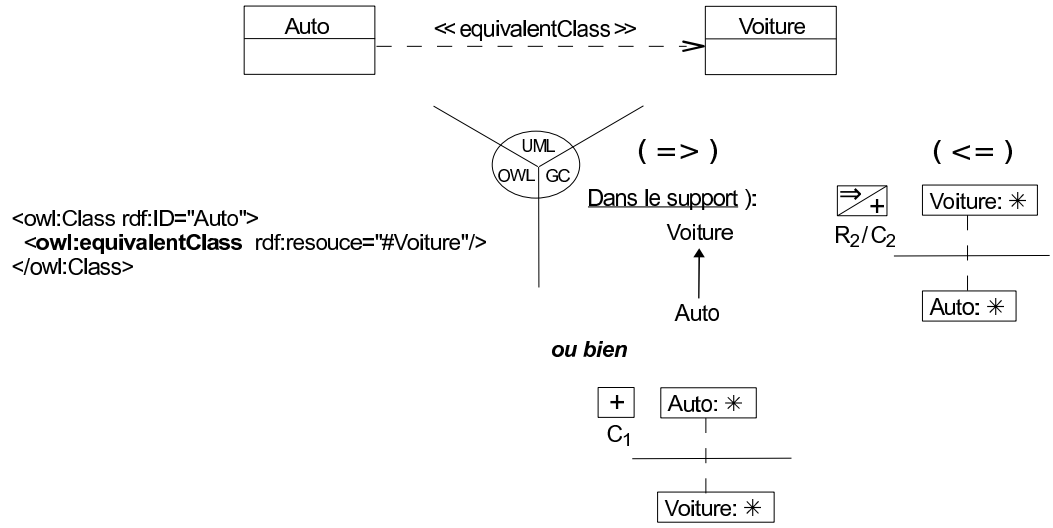


Figure 46 – Classes Auto et Voiture équivalentes.

Les notations (=>) et (<=) n'appartiennent ni au modèle des GCs ni au modèle UML, nous les utilisons pour exprimer l'aspect nécessaire et suffisant associé à la définition de la classe Auto par équivalence (*cf.* Propriété 5).

Remarquons, qu'avec le modèle UML ou le modèle des GCs, on aurait pu être tenté d'utiliser un « double héritage ». C'est-à-dire de spécifier que Voiture est une sous-classe de Auto et inversement. Mais une telle utilisation de l'héritage crée un cycle d'héritage, ce qui n'est généralement pas souhaité.

4.2.2.2 Intersection

L'intersection tout comme l'union permet de définir une classe en la construisant à partir de plusieurs autres classes.

Définition 4.6 (Intersection) *Une classe C est formée de l'*intersection *des classes C_1 à C_n si et seulement si les individus dans l'extension de C sont exactement les individus à la fois dans l'extension de C_1, ..., et dans l'extension de C_n.*

Prenons l'exemple en Figure 47 où la classe Quadricycle est définie comme étant l'intersection de la classe Auto et de la classe Moto.

En OWL, l'intersection de plusieurs classes s'exprime par la propriété owl:intersectionOf qui contient l'ensemble - ou collection - de classes participant à l'intersection.

Avec le modèle des GCs, l'utilisateur peut d'une part souhaiter qu'une classe définie par intersection soit interprétable comme une connaissance inférentielle. Ainsi, une telle classe est modélisée par un type de concept héritant de tous les types de concepts concernés par l'intersection (cf. le support). Cette représentation n'étant que partielle nous la complétons par une règle (cf. R_2), qui indique que tout individu qui est instance de toutes les classes participant à l'intersection est une instance de la classe définie par intersection.

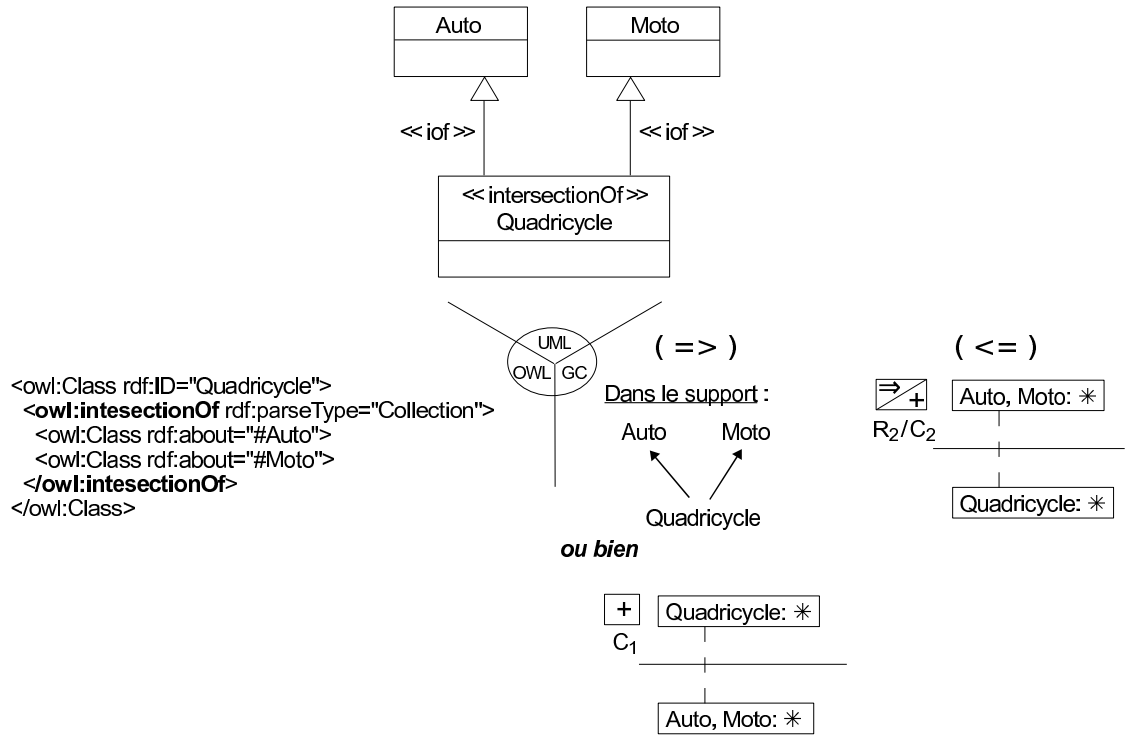


Figure 47 – La classe Quadricycle est l'intersection des classes Auto et Moto.

D'autre part, la connaissance d'une classe définie par intersection peut amener l'utilisateur à vérifier le respect de cette définition au niveau factuel d'une modélisation. Les contraintes positives C_1 et C_2 répondent à cette exigence utilisateur.

En UML, une classe définie par intersection est modélisée dans le système par une classe stéréotypée ; le stéréotype que nous proposons est ≪intersectionOf≫. Cette classe stéréotypée hérite des classes qui en constituent son intersection, comme indiquée en partie haute sur la Figure 47. De façon à distinguer par la suite les sur-classes appartenant à la définition de la classe stéréotypée d'éventuelles autres sur-classes, l'héritage portant sur les classes de l'intersection est représenté par des liens de généralisation stéréotypés par ≪iof≫. Selon les besoins utilisateur, les contraintes OCL suivantes peuvent être considérées :

```
(=>) Context Quadricycle
     inv: self.oclIsKindOf(Auto)
     inv: self.oclIsKindOf(Moto)
```

```
(<=) Context Auto
     inv: if self.oclIsKindOf(Moto) then
          then self.oclIsKindOf(Quadricycle) endif
     Context Moto
     inv: if self.oclIsKindOf(Auto) then
          then self.oclIsKindOf(Quadricycle) endif
```

4.2.2.3 Union

Définition 4.7 (Union) *Une classe C est formée de l'*union *des classes C_1 à C_n si et seulement si les individus dans l'extension de C sont exactement les individus dans l'extension de C_1, ... ou dans l'extension de C_n.*

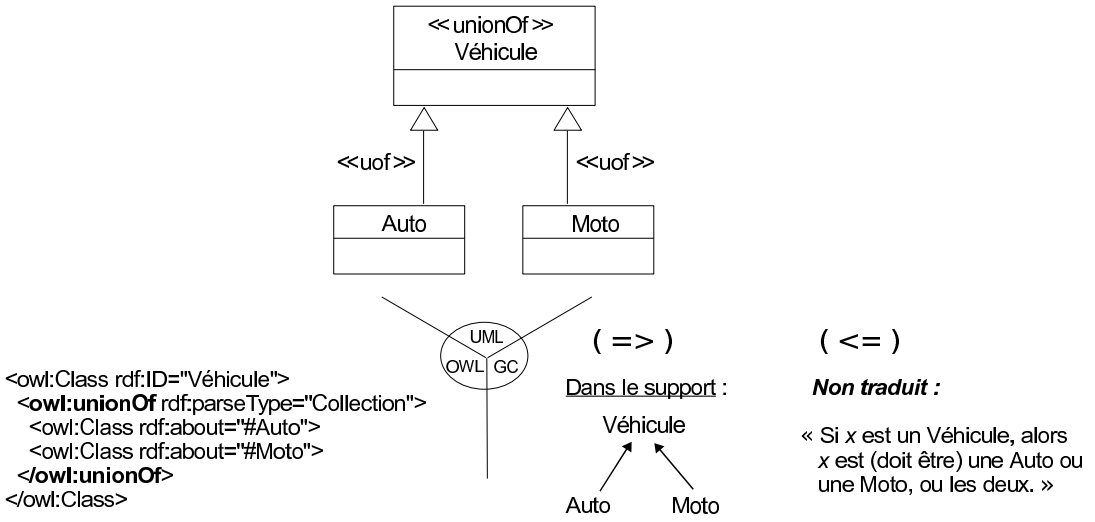


Figure 48 – La classe Véhicule est l'union des classes Auto et Moto.

La Figure 48 constitue un exemple d'union de classes, où la classe Véhicule est définie comme étant l'union de la classe Auto et de la classe Moto.

En OWL, l'union de plusieurs classes s'exprime par la propriété owl:unionOf qui contient un ensemble des classes concernées pour définir la classe union.

Avec le modèle des GCs, nous définissons une telle classe par un type de concept dont tous les types de concepts représentant les classes concernées dans l'union en héritent (cf. le support). Cette modélisation n'est que partielle, et ne peut être complétée dans le modèle actuel des GCs. En effet, il n'est pas possible d'exprimer le fait que si un individu est une instance de la classe union alors il est l'instance d'au moins une des classes participant à cet union.

En UML, nous modélisons une classe définie par union par une classe stéréotypée par ≪unionOf≫. Cette classe stéréotypée est une sur-classe des classes qui en constituent son union, comme présenté en partie haute de la Figure 48. De façon à distinguer par la suite les sous-classes appartenant à la définition de la classe stéréotypée de d'autres sous-classes éventuelles, l'héritage portant sur les classes de l'union est représenté par des liens de généralisation stéréotypé par ≪uof≫. Selon les besoins utilisateur, les contraintes OCL suivantes peuvent être considérées :

```
(=>) Context Auto
     inv: self.oclIsKindOf(Véhicule)

     Context Moto
     inv: self.oclIsKindOf(Véhicule)
```

```
(<=) Context Véhicule
     inv: self.oclIsKindOf(Auto)
          or self.oclIsKindOf(Moto)
```

4.2.2.4 Complément

Définition 4.8 (Complément) *Une classe C_1 est* le complément *d'une classe C_2 si et seulement si les individus qui ne sont pas dans l'extension de C_2 sont dans l'extension de C_1.*

Propriété 7 *Une relation de complémentarité entre deux classes est symétrique.*

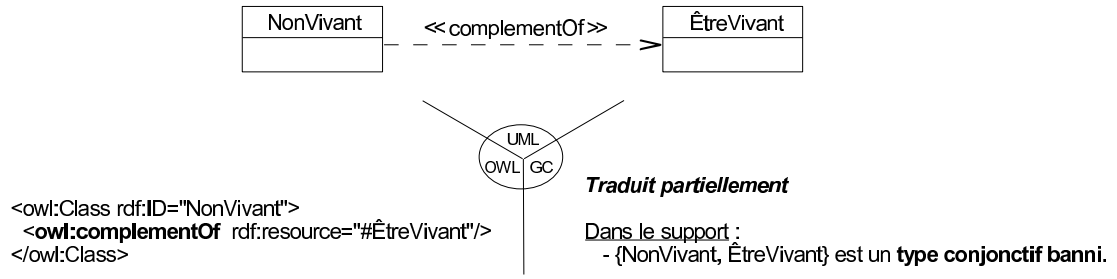


Figure 49 – La classe NonVivant est le complément de la classe ÊtreVivant.

Contrairement à la disjonction, le complément d'une classe constitue une condition nécessaire et suffisante pour définir une classe.

La Figure 49 présente un exemple de complément, où la classe NonVivant est définie comme étant le complément de la classe ÊtreVivant.

En OWL, le complément d'une classe s'exprime par la propriété owl:complementOf. Rappelons qu'en OWL, la définition d'une classe d'intersection, d'union ou de complément est théoriquement une classe anonyme, mais peut cependant être directement nommée dans les dernières recommendations [Dean *et al.*, 2004] (sans nécessiter le recours à une propriété owl:equivalentClass, voir la Section 2.3.1.1) ; il en sera de même pour les classes énumérées (ci-après).

Avec le modèle des GCs, la notion de complément n'est pas modélisable, mais peut l'être partiellement avec la notion de disjonction : les deux classes sont des types incompatibles pour un même individu.

En UML, nous introduisons le stéréotype ≪complementOf≫ applicable sur un lien de dépendance, de sorte que la classe NonVivant dépende pour se définir de la classe ÊtreVivant selon ce lien stéréotypé. Selon les besoins utilisateur, la contrainte OCL suivante peut être ajoutée :

```
    Context NonVivant.commonSuperType(ÊtreVivant)
(=>) inv: NonVivant.commonSuperType(ÊtreVivant).allInstances()
       ->select(i | not i.oclIsKindOf(ÊtreVivant))->forAll(h | h.oclIsKindOf(NonVivant))
(<=) inv: NonVivant.commonSuperType(ÊtreVivant).allInstances()
       ->select(i | not i.oclIsKindOf(NonVivant))->forAll(h | h.oclIsKindOf(ÊtreVivant))
```

Elle indique pour les deux invariances, que si un individu n'est pas de type ÊtreVivant alors il est de type NonVivant, et inversement. Cette vérification devant être faite sur un ensemble d'individus, celui concerné est donc celui des instances de la sur-classe commune la plus spécifique à ÊtreVivant et NonVivant. Cet ensemble existe nécessairement, car par construction la super-classe commune par défaut sera Thing.

Les axiomes complets de classes *intersection*, *union* et *complément* offrent une vision ensembliste de construction de classes en agissant sur les extensions des classes impliquées. Ils peuvent être respectivement assimilés aux opérateurs ensemblistes $\cap$, $\cup$ et $\setminus$.

4.3 Classes matrices

Nous avons vu dans la section précédente qu'une classe pouvait être décrite (axiomes partiels) ou définie (axiomes complets) à partir d'autres classes. Voyons maintenant comment définir les *classes matrices* qui serviront de composants fondamentaux dans une modélisation. Ces classes peuvent être définies indépendament des autres classes d'une modélisation, ou servir à la construction d'autres classes en se servant d'axiomes de classes.

Il existe deux catégories de classes matrices : les classes énumérées et les restrictions d'associations.

4.3.1 Énumération

Définition 4.9 (Classe énumérée) *Une classe est dite* énumérée *lorsqu'elle est définie par un ensemble fini d'individus qui constituent exhaustivement l'extension de la classe.*

En OWL, une classe énumérée est définie par la propriété owl:oneOf. La partie gauche de la Figure 50 présente la classe énumérée Continent qui possèdent exactement pour instances les cinq individus suivants : Eurasie, Afrique, Amérique, Australie et Antartique.

Dans le modèle des GCs, la classe énumérée est partiellement modélisable. En effet, la relation de conformité τ permet au niveau du support de typer les (marqueurs individuels associés aux) individus. Ainsi, il est possible de lister l'ensemble des individus qui définissent la classe énumérée. Cependant, il n'est pas possible d'indiquer que cette énumération est exhaustive.

Une définition de classe énumérée en UML ne semble pas exprimable, d'un point de vue concret. En effet, il y a séparation entre la représentation de la structure et la représentation des faits d'un domaine donné dans une modélisation UML. Notons qu'une tentative a été proposée dans [Brockmans *et al.*, 2004] [6] pour représenter une classe énumérée OWL en UML. Cependant, cette proposition n'est pas utilisable en pratique dans les outils UML pour la raison évoquée précédemment : diagrammes de classes et diagrammes d'objets sont distincts.

4.3.2 Restriction d'associations

Une restriction d'association définit une classe, telle que toutes les instances de cette classe satisfassent à une certaine contrainte quant à la manière dont elles sont associées à d'autres individus par des assertions d'une relation ou d'un slot donné(e). Pour une meilleure lecture de cette section, les notions de *relation* et de *slot* sont abordées au Chapitre 5, celles de *lien* (assertion de relation) ou d'*attribution* (assertion de slot) sont présentées au Chapitre 6.

Les restrictions d'associations sont de deux sortes : les *restrictions de co-domaines* et les *contraintes de cardinalités*.

[6] Travail inspiré d'une proposition inachevée de Schreiber [Schreiber, 2002], reprise aussi dans [Martin, 2007].

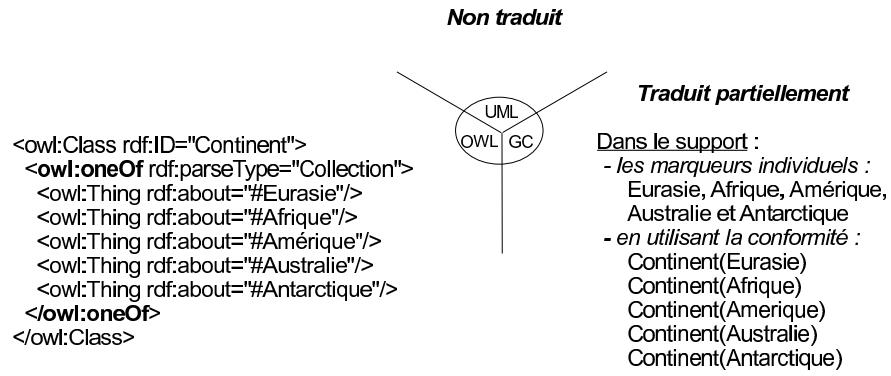


Figure 50 – La classe Continent est définie comme ayant pour extension la liste exhaustive des individus suivants : Eurasie, Afrique, Amérique, Australie et Antarctique.

4.3.2.1 Restriction de co-domaine

Une *restriction de co-domaine* constitue une définition de la classe située au domaine d'une relation ou d'un slot. Cette contrainte agit sur le co-domaine de la relation ou du slot, en fonction de son domaine (c'est-à-dire la classe définie). Les contraintes de co-domaine sont de trois types : celles contraignant le type de tous les individus dans le co-domaine d'une relation ou d'un slot, celles contraignant le type d'au moins un individu dans le co-domaine, et celles restreignant le co-domaine à un unique individu ou une unique donnée.

■ **Restriction universelle de co-domaine**

Définition 4.10 (Restriction universelle de co-domaine) *Soient A une relation binaire (respectivement un slot), D le domaine de A et C le co-domaine. Soit C' une classe (respectivement un datatype) plus spécifique que C. Une* restriction universelle de co-domaine à valeur dans *C' définit une sous-classe D' de D, telle que l'extension de D' est constituée de chaque instance de D qui si elle est le premier argument d'une assertion a de type A alors le second argument de a est une assertion de type C'.*

Cette définition ne précise pas que chaque instance de D' est liée - en tant que premier argument - à une assertion d'une relation ou d'un slot de type A. Par contre, dans l'hypothèse ou une instance de D' le serait, alors le second argument - le premier étant l'instance de D' - de l'assertion de type A est nécessairement une assertion de type C'.

En Figure 51 est présenté un exemple d'utilisation de restriction universelle de co-domaine à valeur dans Homme. Elle précise que pour toute assertion de la relation père dont le premier argument est une instance de la classe Humain, alors le second argument est nécessairement une instance de la classe Homme.

En OWL, une restriction universelle de co-domaine est modélisée par une ressource de type owl:Restriction. Cette restriction est liée par la propriété owl:onProperty à la relation concernée, ici père, ou le slot concerné, et par la propriété owl:allValuesfrom au co-domaine

concerné, ici la classe Homme. Rappelons qu'une owl:Restriction en OWL est une classe anonyme, qui constitue un bloc élémentaire dont le but est de décrire ou définir une autre classe via un axiome de classe. Dans cet exemple en Figure 51, nous supposons que la restriction constitue une condition nécessaire à la définition de la classe Humain, d'où l'utilisation de l'axiome de classe rdfs:subClassOf.

Dans le modèle UML, nous introduisons le stéréotype ≪allValuesFrom≫ et stéréotypons par ce dernier la relation ou le slot pris(e) en compte dans la restriction universelle de co-domaine. En l'état actuel (de l'extrait) du diagramme de classes en partie supérieure de la Figure 51, la restriction universelle de co-domaine constitue la définition de la classe Humain. Cette restriction sera seulement considérée comme une condition nécessaire (mais non suffisante) à la définition de cette classe lorsque d'autres caractéristiques participeront à la définition de Humain (voir par exemple la Section 4.3.2.2). Ce diagramme de classes est donc un raccourci syntaxique du diagramme de classes en Figure 52, où la classe C_1 (les classes anonymes n'existent pas en UML) est celle définie exactement (aucune autre caractéristique ne sera apportée à C_1) par la restriction universelle de co-domaine. La classe Humain est finalement une spécialisation de la C_1, qui en constitue donc une condition nécessaire. À cette représentation graphique et si selon l'utilisateur cette restriction doit être une contrainte à vérifier, nous associons la sémantique OCL suivante :

```
(=>)  Context C_1
      inv: if self.père[range]->notEmpty() then
           self.père[range]->forAll(ev | ev.oclIsKindOf(Homme))[7] endif

(<=)  Context ÊtreVivant
      inv: if self.père[range]->exists(ev | ev.oclIsKindOf(Homme))[8] then
           self.oclIsKindOf(C_1) endif
```

Avec le modèle des GCs et suivant les besoins utilisateurs, une restriction universelle de co-domaine est modélisée soit par les règles R_1 et R_2 soit par les contraintes positives C_1 et C_2. Il s'agit d'une part de déduire ou de vérifier (R_1 ou C_1) la nécessité que tout père d'un Humain - sorte-de C_1 avec cet exemple - est un Homme. D'autre part, il est suffisant de déduire ou de vérifier (R_2 ou C_2) que tout ÊtreVivant (la relation père étant définie entre deux ÊtreVivants) qui a pour père un Homme est un C_1. De manière à ce que la restriction universelle de co-domaine C_1 constitue une condition nécessaire à la définition de la classe Humain, et corresponde à la modélisation OWL en Figure 51 ou UML en Figure 52, l'ordre partiel suivant est introduit dans le support : Humain $\leq C\_1$. Notons, que dans le cas comme ici, où la restriction ne constitue qu'une condition nécessaire à la définition d'une classe, il est possible en graphes conceptuels de ne retenir que la règle R_1 ou la contrainte positive C_1 en remplaçant directement le type de concept C_1 par Humain.

■ **Restriction existentielle de co-domaine**

Définition 4.11 (Restriction existentielle de co-domaine) *Soient A une relation binaire (respectivement un slot), D le domaine de A et C le co-domaine. Soit C' une classe (respectivement un datatype) plus spécifique que C. Une* restriction existentielle de co-domaine à valeur dans

[7] L'expression `Homme.allInstances()->includesAll(self.père[range])` peut aussi être utilisée.
[8] Peut aussi s'écrire `self.père[range]->select(ev | ev.oclIsKindOf(Homme))->notEmpty()`

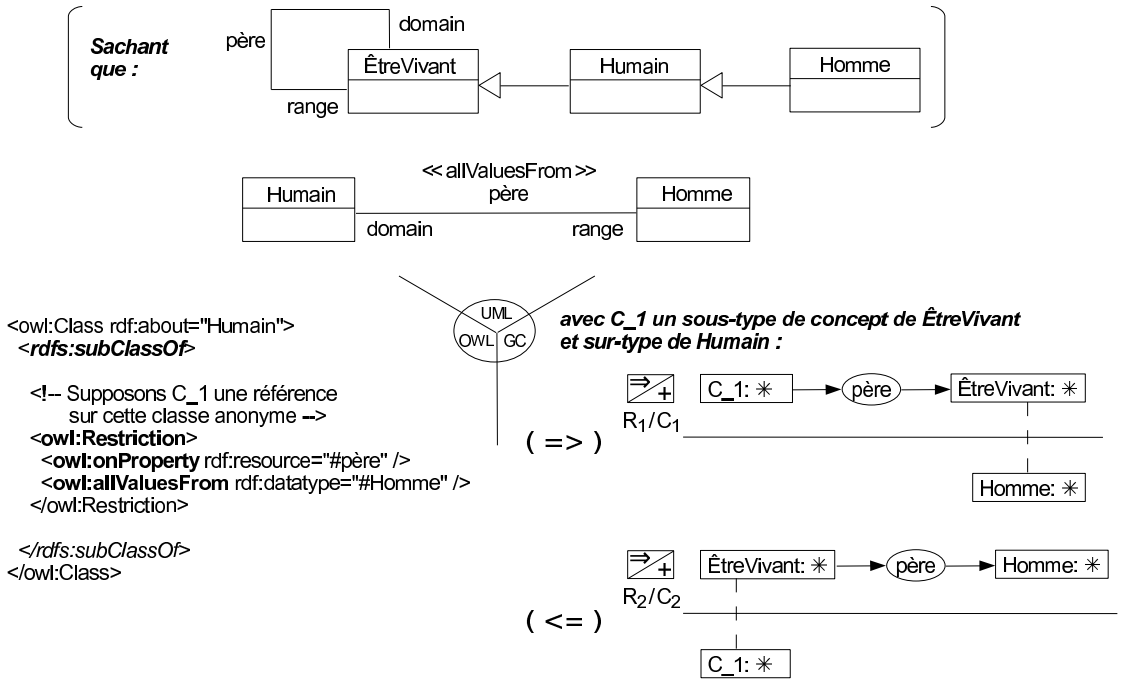


Figure 51 – Restriction universelle de co-domaine.

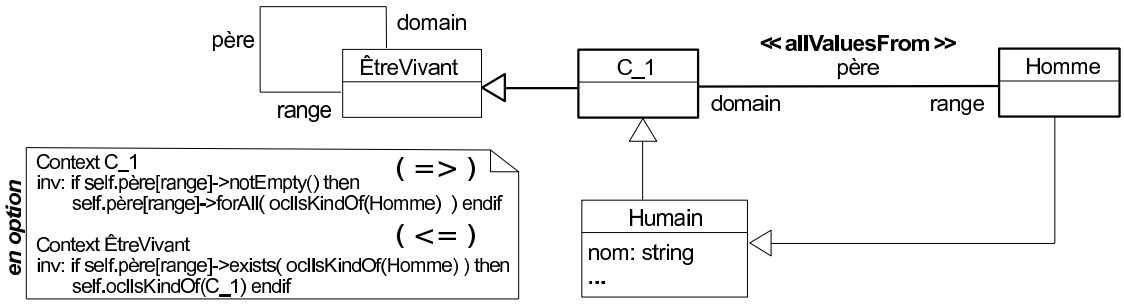


Figure 52 – Précision en UML sur la restriction universelle de co-domaine en Figure 51.

C' définit une sous-classe D' de D, telle que tout individu dans l'extension de D' est le premier argument d'au moins une assertion a de type A tel que le second argument de a est une assertion de type C'.

La Figure 53 présente un exemple d'utilisation de restriction existentielle de co-domaine à valeur dans Auto. Elle précise que toute instance de la classe PiloteAuto - car classe équivalente ici de C_2 - constitue le premier argument d'au moins une assertion de la relation piloter dont le second argument est une instance de la classe Auto.

En OWL, une restriction existentielle de co-domaine est modélisée par une owl:Restriction, qui d'une part est liée par la propriété owl:onProperty à la relation concernée, ici piloter, ou au slot concerné, et d'autre part par la propriété owl:someValuesfrom au co-domaine concerné, ici la classe Auto. La owl:Restriction étant une classe anonyme, nous utilisons ici l'axiome de classe owl:equivalentClass pour identifier cette classe définie par restriction existentielle de co-domaine.

Dans le modèle UML, nous stéréotypons par ≪someValuesFrom≫ la relation ou le slot concerné(e) par la restriction existentielle de co-domaine. À cette représentation graphique et si selon l'utilisateur cette restriction doit être une contrainte à vérifier, nous associons la sémantique OCL suivante :

```
(=>)  Context PiloteAuto
      inv: self.piloter[range]->exists(a | a.oclIsKindOf(Auto))

(<=)  Context Pilote
      inv: if self.piloter[range]->exists(a | a.oclIsKindOf(Auto)) then
           self.oclIsKindOf(PiloteAuto) endif
```

Avec le modèle des GCs et suivant les besoins utilisateurs, une restriction existentielle de co-domaine est modélisée soit par les règles R_1 et R_2 soit par les contraintes positives C_1 et C_2. Il est nécessaire d'une part de déduire ou de vérifier (R_1 ou C_1) que tout C_2 pilote(r) au moins une Auto. D'autre part, il est suffisant de déduire ou de vérifier (R_2 ou C_2) que tout Pilote qui pilote(r) une Auto est un C_2 (et donc un PiloteAuto).

■ **Restriction de co-domaine constant**

Les restrictions de co-domaines constants sont des cas particuliers de restrictions de co-domaines existentielles, où l'existence du second argument d'une assertion d'une relation (respectivement d'un slot) n'est pas uniquement fonction de son type mais aussi de son identité (respectivement de sa valeur).

Définition 4.12 (Restriction de co-domaine constant) *Soient A une relation binaire (respectivement un slot), D le domaine de A et C le co-domaine. Soit c un individu (respectivement une donnée) de type C. Une* restriction de co-domaine constant à valeur *c définit une sous-classe D' de D, telle que tout individu dans l'extension de D' est le premier argument d'au moins une assertion a de type A telle que le second argument de a soit c.*

En Figure 54 est présenté un exemple d'utilisation de restriction de co-domaine constant à valeur 4, une donnée de type int. Cette restriction définit la classe Auto - car classe équivalente ici de C_3 - dont tous les individus sont le premier argument de (au moins) une assertion de la relation nbr_roues ayant pour second argument la donnée 4.

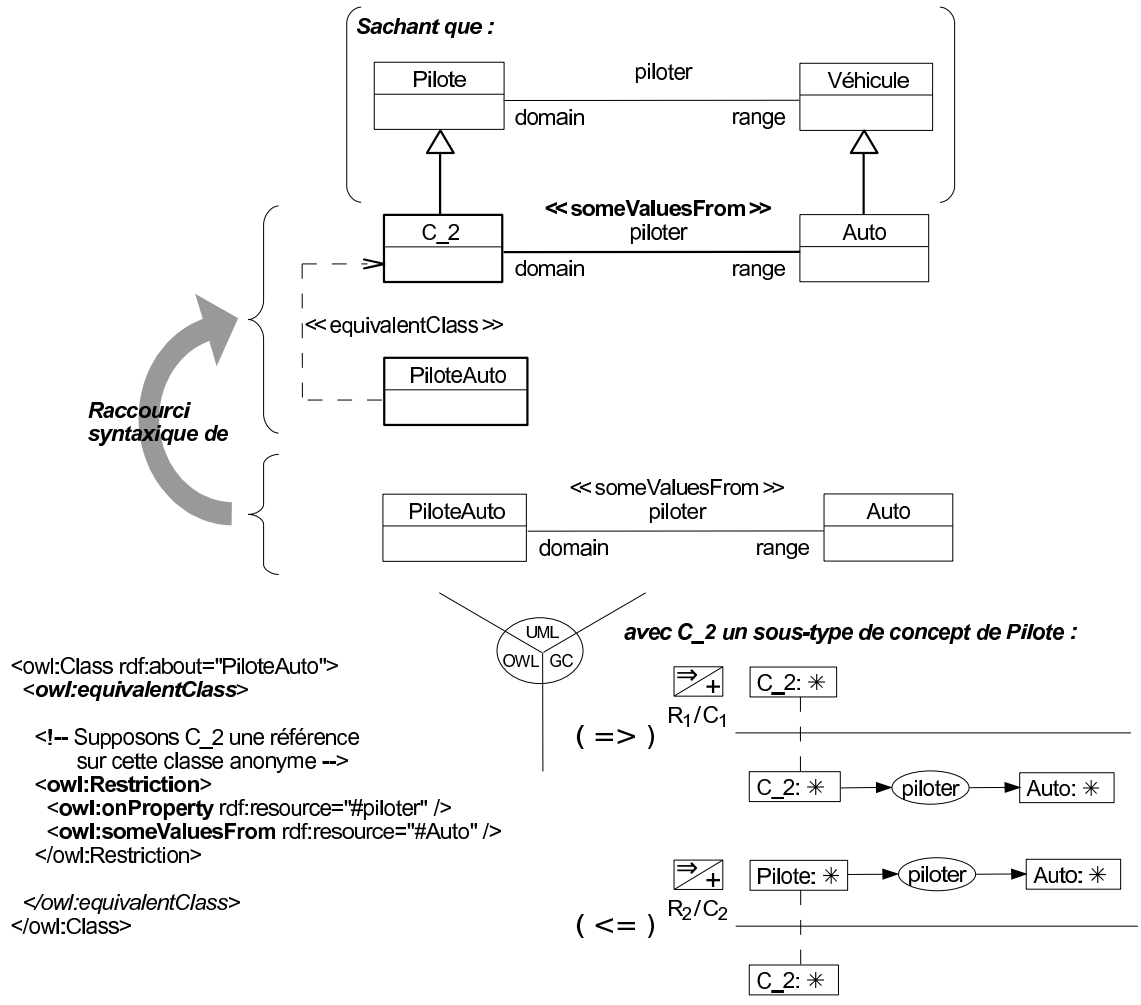


Figure 53 – Restriction existentielle de co-domaine.

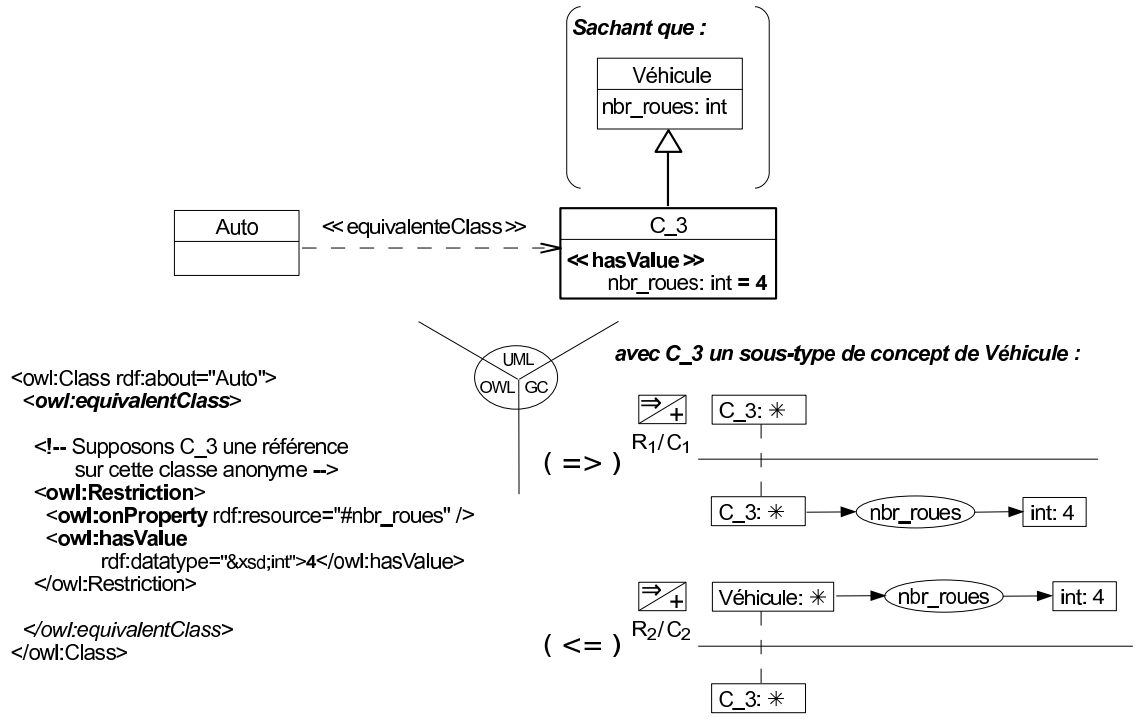


Figure 54 – Restriction de co-domaine constant.

En OWL, une restriction de co-domaine constant définit une owl:Restriction, qui d'une part est liée par la propriété owl:onProperty à la relation concernée ou au slot concerné, ici le slot nbr_roues, et d'autre part par la propriété owl:hasValue à la valeur concernée, ici la donnée 4.

Dans le modèle UML, nous stéréotypons par ≪hasValue≫ le slot concerné par la restriction de co-domaine constant. Remarquons que si la restriction de co-domaine constant concerne une relation et donc un individu comme valeur, alors cette restriction n'est pas représentable concrètement. En effet, il faudrait d'un point de vue logiciel être capable de représenter sur un même diagramme UML une classe (celle définie) et un individu (la valeur constante). Seule la « version » OCL peut dans ce cas figurer sur le diagramme de classe. Selon l'utilisateur, si la restriction en Figure 54 doit être une contrainte à vérifier, nous associons la sémantique OCL suivante :

```
(=>)  Context Auto
      inv: self.nbr_roues=4
```

```
(<=)  Context Véhicule
      inv: if self.nbr_roues=4 then
           self.oclIsKindOf(Auto) endif
```

Avec le modèle des GCs et suivant les besoins utilisateurs, une restriction de co-domaine constant est modélisée soit par les règles R_1 et R_2 soit par les contraintes positives C_1 et C_2. Il est nécessaire d'une part de déduire ou de vérifier (R_1 ou C_1) que tout C_3 ait un nbr_roues qui soit exactement 4 (de type int). D'autre part, il est suffisant de déduire ou de vérifier (R_2 ou C_2) que tout Véhicule qui a pour nbr_roues la valeur 4 est une Auto.

4.3.2.2 Restriction de cardinalité

Définition 4.13 (Restriction de cardinalité) *Soient A une relation binaire (respectivement un slot), et D le domaine de A. Une* restriction de cardinalité sur *A définit une sous-classe D' de D, telle que tout individu dans l'extension de D' soit le premier argument de x assertions différentes (respectivement données différentes) de type A où $x \in [m, n]$. m et n sont deux entiers positifs, respectivement appelés la* cardinalité minimale *et la* cardinalité maximale.

Notons que si la m n'est pas explicitement renseigné, $m = 0$; si n n'est pas explicitement renseigné, $x \in [m, +\infty[$.

L'exemple présenté en Figure 55 indique que la classe Rallye possède - car ici sous-classe de C_4 - une cardinalité de 1 pour le slot nom : toute instance de Rallye doit être liée, en tant que premier argumant, par exactement une assertion du slot nom à une donnée de type string. Il est aussi précisé qu'une instance de Rallye - car ici sous-classe de C_5 - est liée à plusieurs assertions de être_composé. En effet la cardinalité minimale est de 2 et la cardinalité maximale est quant à elle quelconque soit parce qu'elle n'est pas renseignée comme en OWL soit par la présence du caractère '$*$' en UML.

En OWL, une restriction de cardinalité définit une owl:Restriction, qui d'une part est liée par la propriété owl:onProperty à la relation concernée ou au slot concerné, et d'autre part par une des propriétés owl:minCardinality, owl:maxCardinality ou owl:cardinality à la valeur de la cardinalité concernée.

Avec le modèle UML, un restriction de cardinalité est traduite par une multiplicité associée à une relation ou à un slot. Rappelons que par défaut la valeur d'une multiplicité minimale et maximale est de 1 en UML. Selon les besoins de l'utilisateur, une sémantique OCL peut être associée à une restriction de cardinalité (voir la Section 3.3.2). Là encore nous avons fait le choix, d'après l'axiome *sous-classe*, que les restrictions de cardinalités présentées en Figure 55 ne constituent que des conditions nécessaires à la définition de la classe Rallye. En effet, c'est la classe C_4 qui est définie (par restriction de cardinalité) comme ayant pour extension l'ensemble des individus qui ont exactement un nom, et la classe C_5 (par restriction de cardinalité minimale) avec pour extension les individus liés en premiers arguments par des assertions de la relation être_composé à au moins 2 instances différents de la classe Étape. La classe Rallye est finalement décrite comme sous-classe de C_4 et C_5.

Le modèle des GCs ne permet pas de prendre en compte de façon générale la notion de cardinalité. Cependant, une restriction de cardinalité peut partiellement être prise en compte dans le cas où la cardinalité minimum est de 1, et dans le cas où la cardinalité maximum est de 0. En effet, d'une part dire que la cardinalité minimum est (au moins) de 1 revient à exprimer une restriction existentielle de co-domaine sans spécialiser le co-domaine. Sur la Figure 55 avec la classe C_4, seule la condition nécessaire de cette restriction existentielle de co-domaine est précisée pour la relation nom. D'autre part, dire que la cardinalité maximum est de 0 revient à interdire la liaison d'une assertion de relation ou de slot à une instance de la classe concernée comme premier argument. Par exemple, la contrainte négative C_2 modélise une telle interdiction : « un humain ne peut être_composé d'étape(s) ».

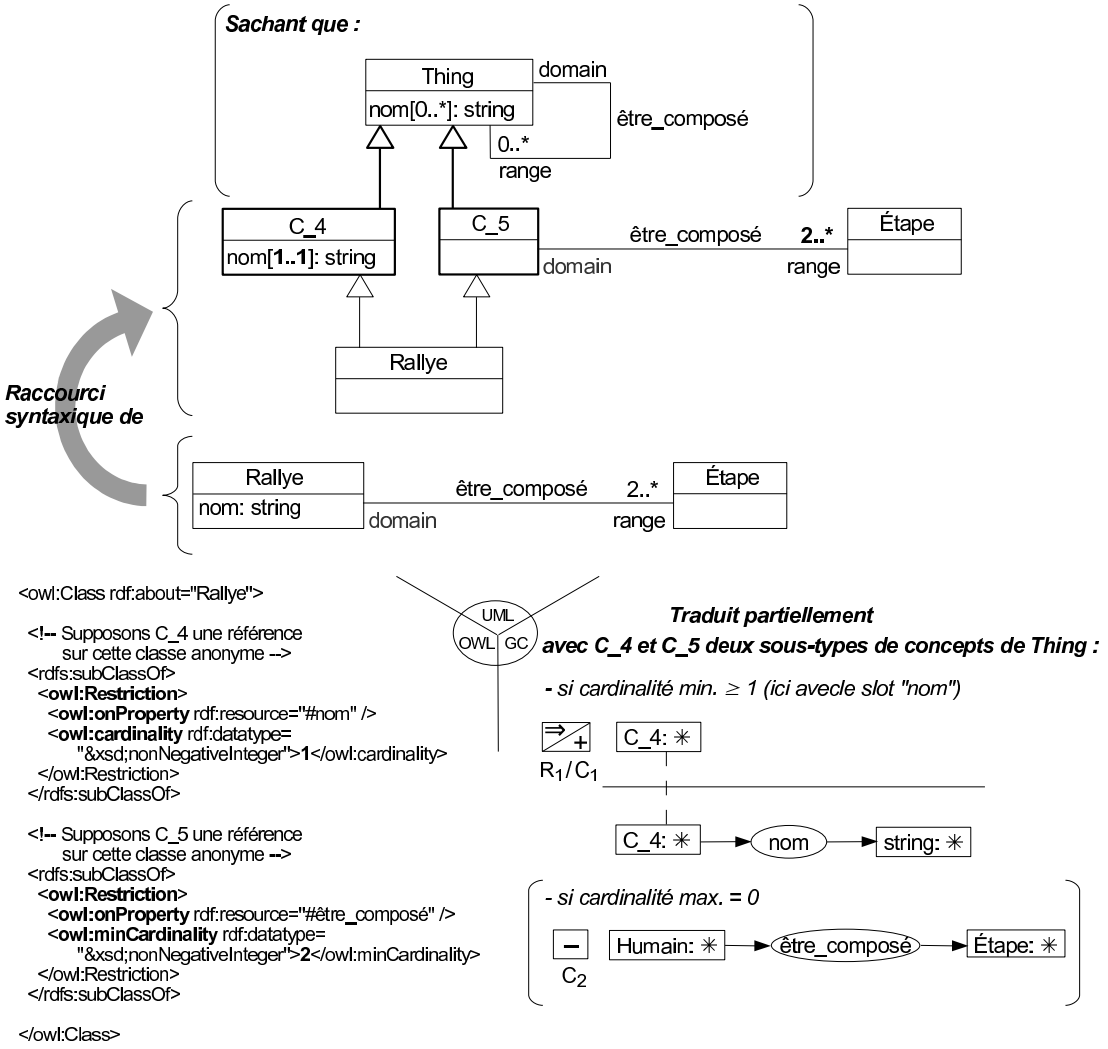


Figure 55 – Restrictions de cardinalités.

4.4 Conclusion

Dans ce chapitre ont été définies les notions de *classe* et de *datatype* ainsi que leurs sous-notions adjacentes. Il a été également présenté les *règles d'instanciation de notions* indiquant comment ces notions sont instanciables dans chacun des modèles OWL, GCs et UML. Le tableau ci-dessous résume l'« instanciabilité » - totale, partielle ou nulle - des notions présentées dans ce chapitre dans les trois modèles de notre système de connaissance.

Notions ╲ Modèles	OWL	GCs	UML
Classe	o	o	o
sous-classe	o	o	o
disjonction	o	o	o
équivalence	o	o	o
intersection	o	o	o
union	o	.	o
complément	o	.	o
énumération	o	.	x
restriction universelle de co-domaine	o	o	o
restriction existentielle de co-domaine	o	o	o
restriction de co-domaine constant	o	o	o
restriction de cardinalité	o	.	o
Datatype	o	o	o
énumération	o	x	o

Connaissance totalement (o), partiellement (.) ou non (x) modélisable

Notons que parmi les trois modèles concernés dans notre système, celui de OWL dispose du langage offrant le vocabulaire le plus riche. Ainsi, la majorité des notions du système sont issues de celles existantes dans OWL et sont donc généralement (totalement) instanciables dans les modélisations basées sur OWL. Pour UML, dans la mesure où le langage dispose de mécanismes permettant d'étendre son vocabulaire, toutes les notions utilisées dans le système seront en principe instanciables dans les modélisations basées sur UML. De ces deux points, il ne faudrait cependant pas arriver à la conclusion rapide que OWL ou bien UML soit le seul modèle « intéressant » à utiliser pour modéliser les connaissances d'un domaine. En effet, la modélisation de connaissances n'a de sens que si elles sont exploitées par la suite ; « *Tant que les inférences à accomplir ne sont pas spécifiées, il est impossible de savoir si les connaissances sont bien ou mal représentées.* » [Kayser, 1997]. Ceci concerne d'autant plus UML, que nous considérons essentiellement pour son aspect représentationnel plutôt qu'inférentiel. Or, si le modèle GCs semble d'un point de vue expressif plus limité que OWL et UML, il a toutefois toute sa place au sein de notre système de connaissance. En effet, il offre des possibilités de raisonnements complémentaires à ceux possibles dans les deux autres modèles, comme cela est présenté au Chapitre 7 par exemple.

Chapitre 5

Transition de modèles appliquée aux notions de *relation* et de *slot*

Les notions de *relation* et de *slot* permettent de modéliser les associations génériques au niveau des connaissances structurelles d'un domaine. Ces deux notions font partie des notions communes aux modèles des GCs, au modèle OWL et au modèle UML.

Ce chapitre définit dans une première section la notion de *relation* et la notion de *slot* telles qu'elles sont considérées au sein du système de connaissance. Dans cette section sont aussi présentées les règles d'instanciation de notions pour la notion de slot. Les deux sections suivantes présentent des sous-notions associées à celle de relation. La seconde section présente les sous-notions de symétrie, de transitivité et de cardinalité ainsi que leurs règles d'instanciation dans les trois modèles. La troisième section présente les sous-notions d'héritage, d'équivalence et d'inverse, constituant des *axiomes de relations* pour la construction de relations.

Sommaire

5.1 Les notions de *relation* et *slot* en OWL, Graphes Conceptuels et UML

5.1.1 Relation

Les *relations* sont des composants du niveau structurel d'une modélisation des connaissances. Elles permettent d'associer des classes entre elles.

Définition 5.1 (Relation) *Une* relation *est une association générique entre plusieurs classes ordonnées, qui constituent les extrémités de la relation. Dans le cas d'une* relation binaire*, les deux extrémités sont appelées le* domaine *et le* co-domaine *de la relation. Les éléments dans l'extension d'une relation sont des* liens *qui associent des individus, tels que le $i^{ième}$ argument d'un lien soit une instance de la classe qui est l'extrémité en position i de la relation ($0 \leq i \leq n$, avec n l'arité de la relation).*

On dit d'une relation qu'elle est le *type* d'un ensemble de liens, ou que ces liens constituent les *assertions* de la relation. Bien que le modèle des GCs et le modèle UML puissent traiter de relations n-aires avec $n \geqslant 2$ ($n \geqslant 1$ pour les graphes conceptuels), le modèle OWL ne peut tenir compte que des relations binaires. Afin de pouvoir utiliser conjointement des relations dans chacun de ces trois modèles de connaissances, nous nous limitons dans ce travail de transition de modèles aux *relations binaires*.

La Figure 56 indique comment est identifiée une relation dans chacun des trois modèles du système de connaissance.

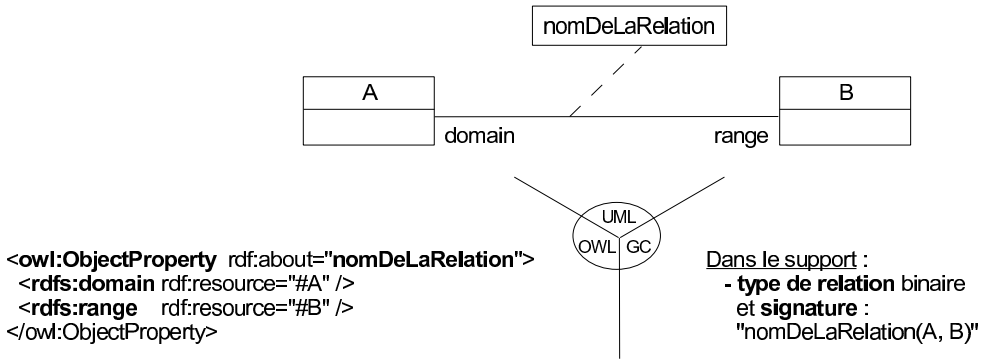


Figure 56 – Une *relation binaire* en OWL, GCs et UML.

En OWL, nous assimilons la notion de relation binaire à la notion de rôle d'objets. Ainsi, une instance de la notion de relation binaire est modélisée en OWL par une instance de la ressource owl:ObjectProperty (à gauche de la figure). La relation est identifiée à l'aide d'un attribut RDF rdf:about (éventuellement rdf:ID) qui a pour valeur l'identifiant de la relation, c'est-à-dire son nom. Le domaine et le co-domaine de la relation sont renseignés à l'aide des propriétés respectives rdfs:domain et rdfs:range. Rappelons que d'une part, le domaine ou le co-domaine d'une relation binaire peuvent ne pas être renseignés en

Figure 57 – Déclaration simplifiée, en UML, d'une *relation binaire* entre deux classes.

OWL. Dans ce cas, la classe constituant le domaine ou le co-domaine est par défaut la classe owl:Thing. D'autre part, le domaine ou le co-domaine en OWL peuvent être multiples, dans ce cas le domaine ou co-domaine est ramené à la classe définie par intersection entre ces multiples classes.

Avec le modèle des GCs, nous modélisons une relation binaire par un type de relation binaire, dont l'identifiant est celui de la relation. La signature du type de relation renseigne le domaine et le co-domaine de la relation. Nous verrons en Section 5.1.2 que tous les types de relations ne correspondent pas à des relations.

Dans le modèle UML, une relation binaire est modélisée d'une façon générale par une classe d'association. Nous nommons les deux extrémités de l'association par domain et range. Nous employons de manière alternative un simple lien d'association UML pour modéliser une relation binaire, comme présenté en Figure 57. Cependant, cette représentation simplifiée ne pourra être utilisée pour définir certaines relations « élaborées ».

5.1.2 Slot

Les *slots* permettent d'associer, au niveau structurel d'une modélisation, des classes à des datatypes.

Définition 5.2 (Slot) *Un* slot *est une association binaire générique entre une classe, appelée le* domaine *et un datatype, appelé le* co-domaine*. Les éléments dans l'extension d'un slot sont des* attributions*, tel qu'une attribution associe un individu instance du domaine à une donnée dans l'extension du co-domaine.*

Un datatype ne peut donc constituer ni le domaine d'une relation ni le domaine d'un slot.

La Figure 58 indique comment est identifié un slot dans chacun des trois modèles du système de connaissance.

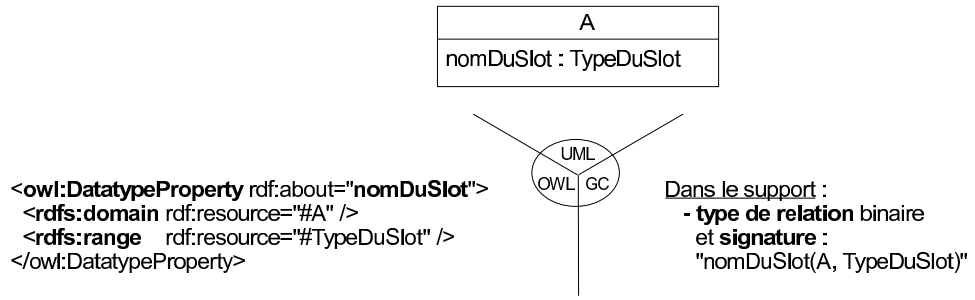


Figure 58 – Un *slot* en OWL, GCs et UML.

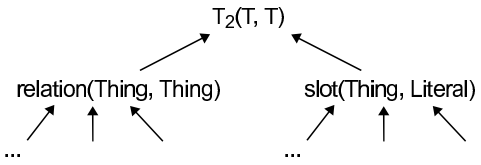


Figure 59 – Séparation, dans le modèle des GCs, des types de relations en deux catégories : ceux représentant les *relations* et ceux représentant les *slots*.

En OWL, nous modélisons un slot par un rôle de types de données. Un slot est donc identifié en OWL comme étant une instance de la ressource owl:DatatypeProperty. Comme pour une relation, le domaine et le co-domaine du slot sont renseignés à l'aide des propriétés respectives rdfs:domain et rdfs:range. Le domaine ou le co-domaine d'un slot peuvent ne pas être renseignés en OWL. Dans ce cas, la classe constituant le domaine est par défaut la classe owl:Thing, tandis que le co-domaine est par défaut est rdf:Literal.

Avec le modèle des GCs, un slot est modélisé par un type de relation binaire, dont la signature renseigne le domaine et le co-domaine du slot. Nous séparons dans le support l'ensemble des types de relations binaires « à valeur » de *relations binaires* de ceux « à valeur » de *slots*. En Figure 59 est présentée cette séparation, où toute relation est un sous-type du type de relation relation et où tout slot est un sous-type du type de relation slot. On rappelle qu'au chapitre précédent, nous avons fait une distinction entre les types de concepts modélisant les classes de ceux modélisant les datatypes. Et qu'à cette occasion nous avons introduit le type de concept Thing pour représenter la classe la plus générale dans toute modélisation et le type de concept Literal pour représenter le datatype le plus général.

Dans le modèle UML, nous modélisons un slot par un attribut tel que le domaine du slot soit la classe contenant l'attribut et le co-domaine du slot soit le type de l'attribut.

5.2 Caractéristiques de relations

Le domaine et le co-domaine d'une relation binaire (abordés en section précédente) constituent deux caractéristiques indispensables à la définition de cette relation. Les notions de *symétrie* ou de *transitivité* peuvent être utilisées, de manière optionnelle, pour venir compléter les caractéristiques d'une relation binaire. Notons que la symétrie ou la transitivité ne peuvent cependant s'appliquer qu'à des relations binaires où le domaine et le co-domaine correspondent à la même classe. Elles ne peuvent être utiliseées avec des slots puisque le domaine (une classe) et le co-domaine (un datatype) d'un slot ne sont pas comparables.

5.2.1 Symétrie

Définition 5.3 (Symétrie) *Soit R une relation binaire ayant pour domaine et co-domaine la classe C. R est une relation binaire* symétrique *si pour toute assertion $r_1 : R(a, b)$ de type R*

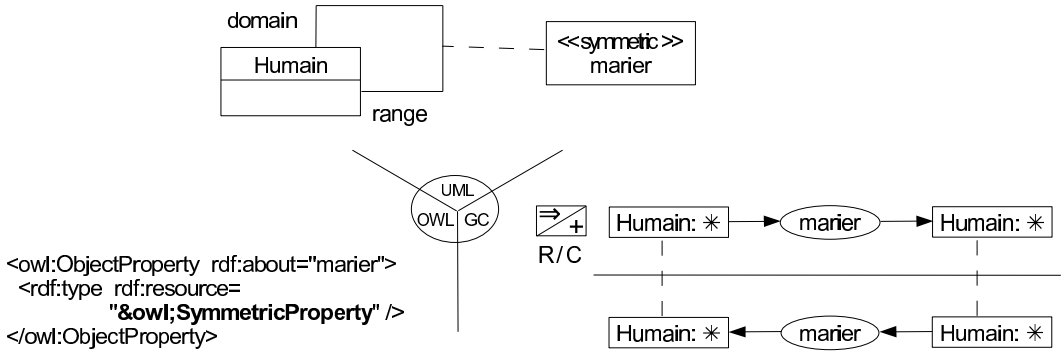


Figure 60 – La relation marier est une relation *symétrique*.

entre deux instances a et b de type C, il existe une assertion $r_2 : R(b, a)$ de type R entre b et a.

En Figure 60 est présentée la relation binaire marier qui a pour caractéristique d'être une relation symétrique.

Avec le modèle OWL, une relation symétrique est modélisée comme étant une instance de la ressource owl:SymmetricProperty. On rappelle que owl:SymmetricProperty est une sous-classe de la classe OWL owl:ObjectProperty.

Dans le modèle des GCs et suivant les besoins utilisateurs, une caractéristique de symétrie est modélisée soit par la règle R soit par la contrainte positive C. La règle modélise le fait que s'il existe un sommet relation marier entre un sommet concept a et un sommet concept b tous deux de type Humain, alors il en résulte qu'un sommet relation marier entre b et a existe. La contrainte positive stipule que cette symétrie est à vérifier dans une modélisation : si un Humain est marier à un second, alors il faut que ce dernier soit marier au premier.

En UML, nous introduisons le stéréotype ≪symmetric≫, que nous employons sur la classe d'association qui représente la relation symétrique. Si l'utilisateur souhaite vérifier qu'une relation binaire est symétrique, la contrainte OCL suivante peut être ajoutée à la modélisation.

```
Context marier
inv: marier.allInstances()->exists(m | m.domain=self.range and  m.domain=self.range)
```

5.2.2 Transitivité

Définition 5.4 (Transitivité) *Soit R une relation binaire ayant pour domaine et co-domaine la classe C. R est une relation binaire* transitive *si pour toutes assertions $r_1 : R(a, b), r_2 : R(b, c)$ de type R entre les instances a et b d'une part et les instances b et c d'autre part, toutes de types C, il existe une assertion $r_3 : R(a, c)$ de type R entre a et c.*

Sur la Figure 61 est présenté un exemple de relation *transitive*, à savoir la relation descendant.

En OWL, une relation transitive est modélisée par une instance de owl:TransitiveProperty, qui est une ressource OWL définie comme une sous-classe de la classe owl:ObjectProperty.

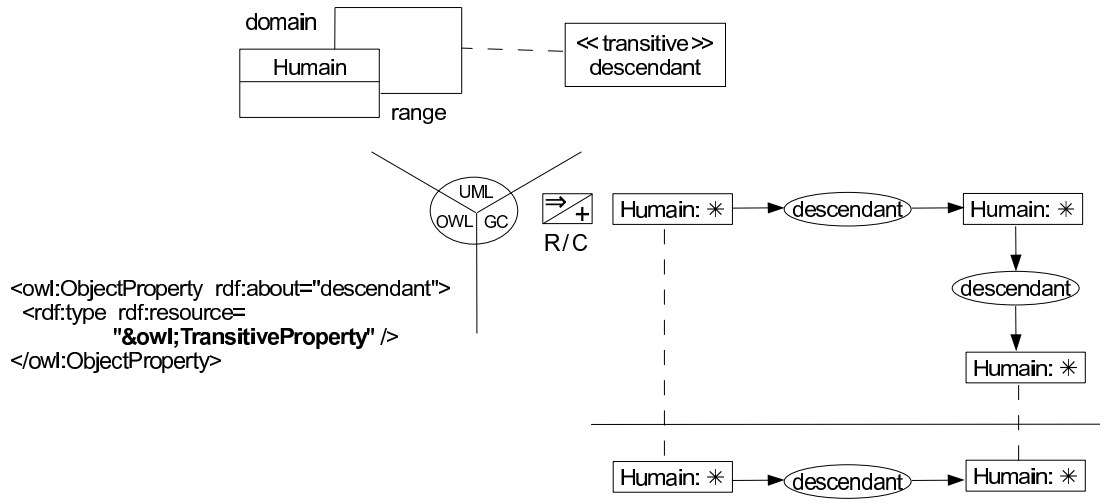


Figure 61 – La relation descendant est une relation *transitive*.

Dans le modèle des GCs et suivant les besoins utilisateurs, une caractéristique de transitivité est modélisée soit par la règle R soit par la contrainte positive C. La règle modélise le fait que si un Humain qui a pour descendant un second Humain qui à son tour a pour descendant à un troisième, alors ce premier Humain a pour descendant ce troisième. La contrainte positive stipule que cette transitivité est à vérifier dans une modélisation.

Pour le modèle UML, nous introduisons le stéréotype ≪transitive≫, que nous employons sur la classe d'association qui représente la relation transitive. Selon l'utilisateur, la contrainte OCL suivante peut venir compléter cette représentation de la transitivité.

```
Context descendant
inv: descendant.allInstances()->forAll(d1 | descendant.allInstances()->forAll(d2 |
     if d1.range = d2.domain then
        descendant.allInstances()->exists(d3 |d3.domain=d1.domain and d3.range=d2.range)
     endif ))
```

5.3 Axiomes de relations et de slots

Les trois axiomes de relations ou de slots considérés dans notre système de connaissance sont l'*héritage*, l'*équivalence* et l'*inverse*.

5.3.1 Héritage

Les trois modèles de connaissances, OWL, graphes conceptuels et UML, offrent la possibilité d'organiser de manière taxonomique les relations d'une modélisation, et ainsi de former une *hiérarchie* de relations. Cependant, l'organisation hiérarchique de slots n'est possible qu'avec le modèle des GCs ou avec le modèle OWL, mais pas avec le modèle UML.

Définition 5.5 (Héritage) *L'*héritage *entre deux relations (respectivement entre deux slots) est une propriété de spécialisation. On dit qu'une relation (respectivement qu'un slot) A_1 hérite d'une relation (respectivement d'un slot) A_2 si A_1 est plus spécialisé(e) que A_2.*

La relation (respectivement le slot) la plus spécifique est appelée la *sous-relation* (respectivement le *sous-slot*) ou le *subsumé*, et la relation (respectivement le slot) la plus générale est appelée la *sur-classe* (respectivement le *sur-slot*) ou le *subsumant*. La sous-relation ou le sous-slot doit être interprété comme une « sorte-de » la sur-relation ou le sur-slot. Il en résulte d'une part que le domaine de la sous-relation (respectivement du sous-slot) peut être une spécialisation du domaine de la sur-relation (respectivement du sur-slot), d'autre part que le co-domaine de la sous-relation (respectivement du sous-slot) peut être une spécialisation du co-domaine de la sur-relation (respectivement du sur-slot).

Propriété 8 *Une relation (respectivement un slot) A_2 hérite d'une relation (respectivement d'un slot) A_1 si les liens (respectivement les attributions) dans l'extension de A_2 sont aussi dans l'extension de A_1.*

Propriété 9 *L'axiome d'héritage entre deux relations ou deux slots est une propriété transitive.*

La Figure 62 décrit la relation piloter comme étant une sous-relation de conduire. Ici, le domaine Pilote de la sous-relation est une spécialisation du domaine Humain de la sur-relation.

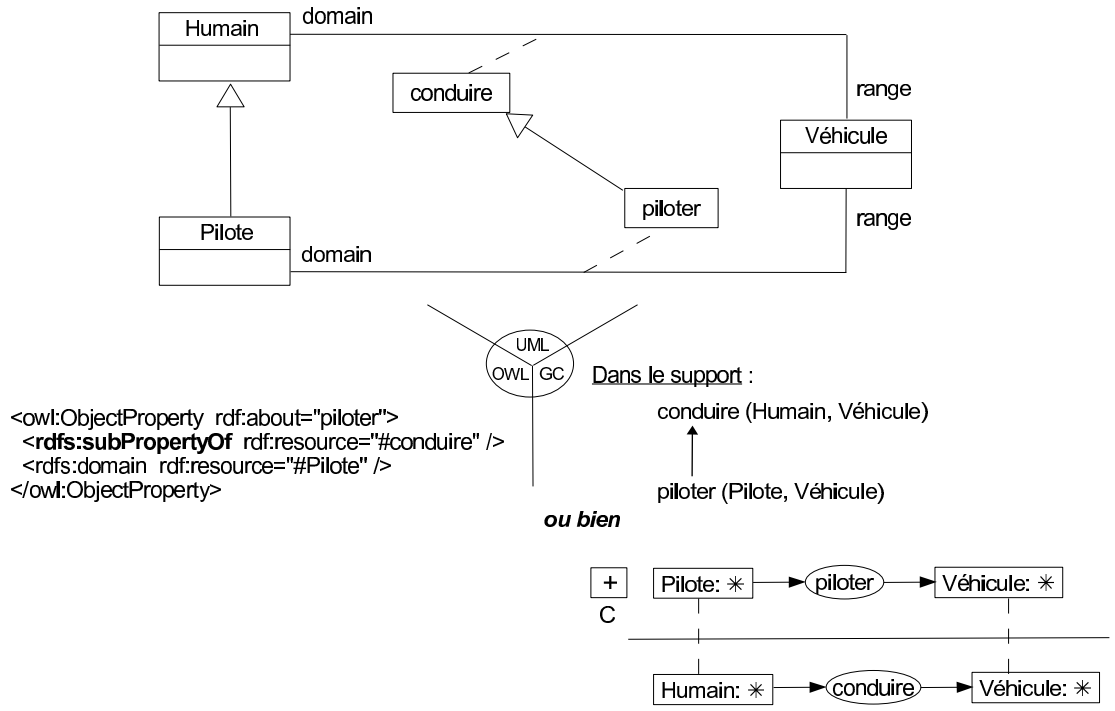


Figure 62 – La relation piloter est une sous-relation de conduire.

En OWL, une relation ou un slot est décrit(e) comme sous-relation ou sous-slot d'un(e) autre à l'aide de la propriété rdfs:subPropertyOf. Pour cela, une instance de rdfs:subPropertyOf constitue le prédicat d'un triplet RDF dont le sujet est la sous-relation ou le sous-slot et l'objet est la sur-relation ou le sur-slot. La description d'un sous-slot se fait en OWL de manière analogue à ce qui est présenté en partie gauche de la Figure 62 pour une sous-relation.

Avec le modèle des GCs, selon les besoins de l'utilisateur, la connaissance « piloter est une sous-relation de conduire » est modélisée et donc interprétable soit par une connaissance à déduire soit par une connaissance à vérifier. Dans le premier cas, l'ordre partiel entre les types de relations dans le support modélise le type de relation piloter comme une « sorte-de » conduire. Notons qu'une règle dans le modèle des GCs aurait pu être utilisée pour arriver à l'expression de ce premier cas, mais nous profitons ici de l'organisation hiérarchique entre types de relations présente dans le support. Dans le second cas, la contrainte positive C stipule que pour tout sommet relation de type piloter entre deux sommet concept a et b, il faut vérifier l'existence d'un sommet relation de type conduire entre a et b. La description d'un sous-slot se fait de manière similaire à la la description en graphes conceptuels d'une sous-relation (en partie droite de la Figure 62).

Dans le modèle UML, nous modélisons par un lien de généralisation la subsomption entre deux relations. L'utilisation de la généralisation est applicable ici pour les relations dans la mesure où nous avons fait le choix de modéliser une relation par une classe d'association (voir la Section 5.1.1). Si l'utilisateur souhaite que la subsomption entre les deux relations soit considérée comme une connaissance à vérifier, la contrainte OCL suivante y répond :

```
Context piloter
inv: self.oclIsKindOf(conduire)
```

Il n'est cependant pas possible de modéliser une hiérarchie entre slots. En effet, nous modélisons en UML un slot par un attribut (voir la Section 5.1.2) ; mais en modélisation orientée objet - ce pour quoi UML est dédié - il n'est pas possible de spécialiser un attribut.

5.3.2 Équivalence

Définition 5.6 (Équivalence) *Deux relations (respectivement deux slots)* équivalent(e)s *constituent simultanément un type pour chacune de lerus assertions (respectivement pour chaque attribution de leurs extensions).*

Propriété 10 *Une relation (respectivement un slot) A_1 est* équivalent(e) *à une relation (respectivement un slot) A_2 si et seulement si les liens (respectivement les attributions) dans l'extension de A_2 sont dans l'extension de A_1 et les liens (respectivement les attributions) dans l'extension de A_1 sont dans l'extension de A_2.*

Propriété 11 *L'axiome d'équivalence entre deux relations ou deux slots est une propriété symétrique.*

L'équivalence constitue une condition nécessaire et suffisante pour définir une relation ou un slot. L'exemple en Figure 63 définit la relation papa comme étant équivalente à la classe père.

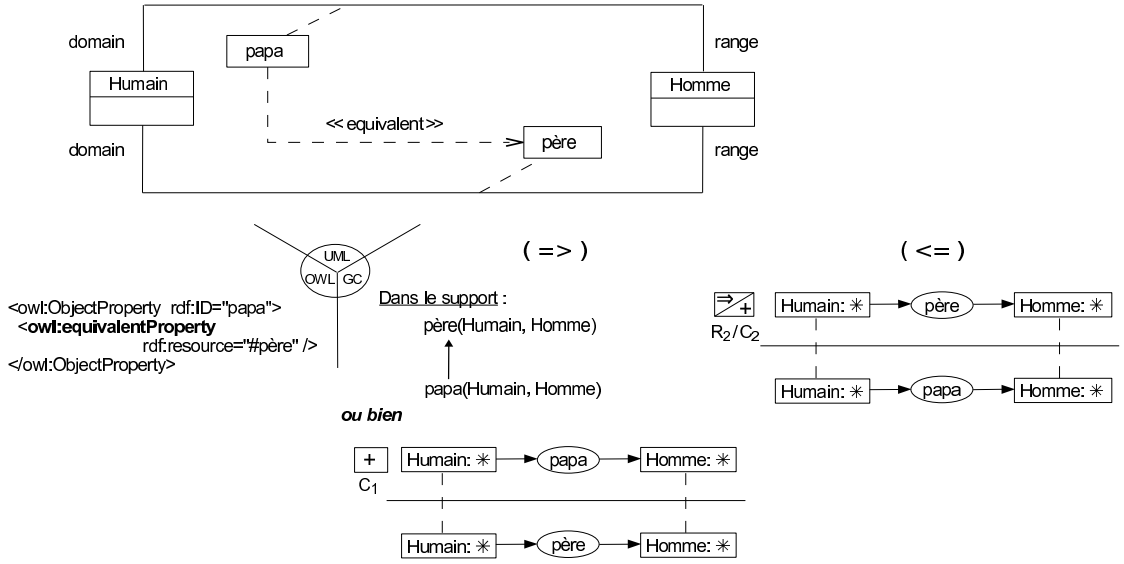


Figure 63 – Relations père et papa équivalentes.

En OWL, une équivalence entre deux relations ou deux slots est modélisée par la propriété rdfs:equivalentProperty. Une instance de rdfs:equivalentProperty constitue le prédicat d'un triplet RDF dont le sujet est relation ou le slot défini(e) et l'objet est la relation ou le slot équivalent(e). La définition d'un slot équivalent se fait en OWL de manière analogue à ce qui est présenté en partie gauche de la Figure 63 pour une relation équivalente.

Dans le modèle des GCs et suivant les besoins utilisateurs, une équivalence est modélisée soit comme une connaissance permettant la déduction d'une nouvelle connaissance soit comme une connaissance nécessitant d'être vérifiée. Le premier choix consiste à déduire que tout sommet relation de type papa est de type père (ordre partiel du support), et que tout sommet relation de type père entre deux sommets concepts entraîne l'existence d'un sommet relation de type papa entre ces deux sommets concepts (règle R_2). Le second choix est de vérifier que s'il existe un sommet relation de type papa entre deux sommets concepts il doit exister un sommet relation de type père entre ces mêmes sommets concepts (contrainte C_1), et inversement (contrainte C_2). Comme en partie droite de la Figure 63 pour la définition en graphes conceptuels d'une relation équivalente, la description d'un slot équivalent se fait de manière similaire.

Avec le modèle UML nous introduisons le stéréotype ≪equivalent≫ que nous appliquons à un lien de dépendance entre les deux relations équivalentes ; la relation source de ce lien étant celle définie comme équivalente de l'autre. Cette équivalence peut, suivant le souhait utilisateur, être exprimée par les deux contraintes OCL suivantes :

```
(=>)  Context papa
      inv: self.oclIsKindOf(père)
```

```
(<=)  Context père
      inv: self.oclIsKindOf(papa)
```

Pour la définition de slots équivalents, nous assimilons cette notion à celle d'*attributs dérivés* en UML. D'un point de vue représentation, seul le fait de savoir que l'attribut est équivalent à un autre est visualisable (cf. la caractère '/' devant le nom de l'attribut) mais pas de quel attribut. En Figure 64 est présenté un exemple d'attribut équivalent : l'attribut name. Il est possible de préciser avec OCL, mais uniquement sous forme d'une contrainte donc, de quel attribut est équivalent le slot défini par équivalence. La contrainte OCL s'exprime sous la forme d'une dérivation ; en Figure 64 le slot name dérive du slot nom comme lui étant identique.

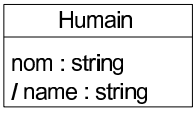


Figure 64 – Cas de slots équivalents en UML, ici les slots nom et name.

5.3.3 Inverse

Définition 5.7 (Inverse) *Soient R_1 et R_2 deux relations binaires ayant pour domaine la classe D et co-domaine la classe C. R_1 est l'*inverse *de R_2 si et seulement si pour toutes assertions $r_1 : R_1(a, b)$ de type R_1 entre une instance a de D et une instance b de C, il existe une assertion $r_2 : R_2(b, a)$ de type R_2 entre b et a.*

Remarquons que cet axiome ne peut pas s'appliquer sur des slots, car le domaine (une classe) et le co-domaine (un datatype) d'un slot ne sont pas comparables.

Propriété 12 *L'axiome inverse entre deux relations est une propriété symétrique.*

L'axiome inverse constitue donc une condition nécessaire et suffisante pour définir une relation. La Figure 65 présente un exemple de relations inverses, où la relation être_dirigé est définie comme étant l'inverse de la relation diriger.

Avec le modèle OWL, l'inverse entre deux relations est modélisé à l'aide de la propriété owl:inverseOf, telle qu'une instance de cette propriété constitue le prédicat d'un triplet RDF dont le sujet est la relation défini(e) et l'objet est la relation inverse.

Dans le modèle des GCs, selon les besoins de l'utilisateur, l'inverse est modélisé soit par les règles R_1 et R_2, soit par les contraintes positives C_1 et C_2. D'une part il s'agit, sur cet exemple, de modéliser le fait que l'existence d'un sommet relation de type être_dirigé (respectivement diriger) entre deux sommets concepts entraîne l'existence d'un sommet relation de type diriger (respectivement être_dirigé) entre ces deux sommets concepts dont l'ordre des voisins est inversé. D'autre part, il s'agit de modéliser la contrainte de l'existence d'un sommet relation de type être_dirigé (respectivement diriger) entraîne l'obligation l'existence d'un sommet relation de type diriger (respectivement être_dirigé) entre ces deux sommets concepts dont l'ordre des voisins est inversé.

Pour le modèle UML, nous introduisons le stéréotype ≪inverseOf≫, que nous employons sur un lien de dépendance entre les deux relations, partant de celle définie vers

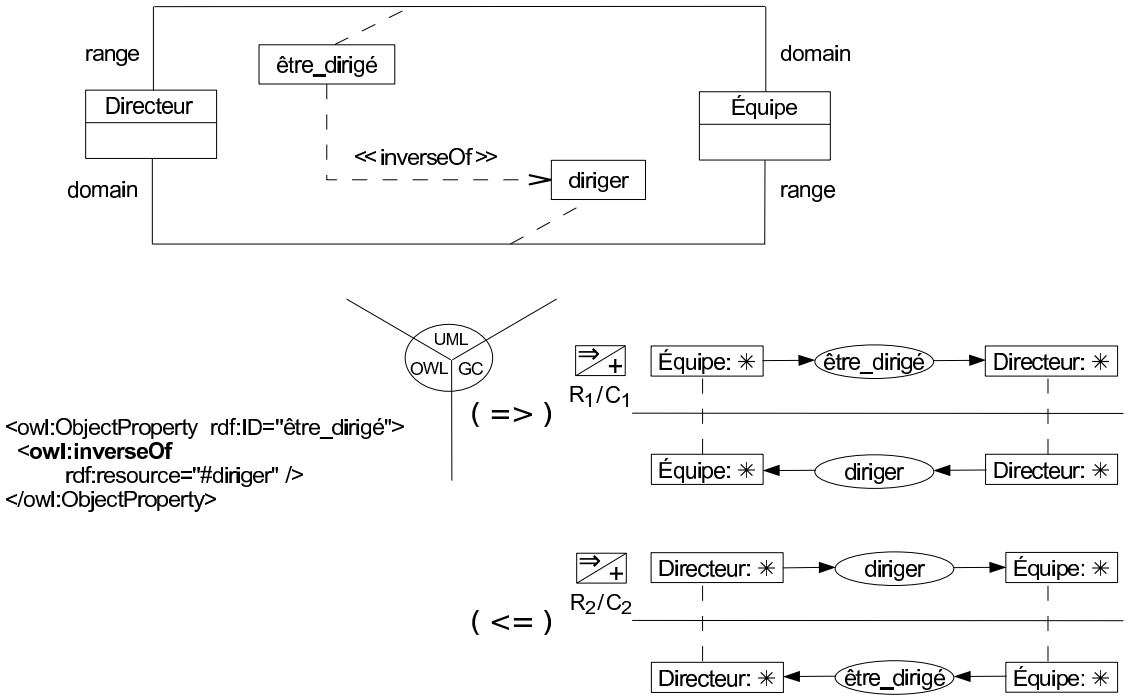


Figure 65 – Relation être_dirigé inverse de la relation diriger.

son inverse. Selon l'utilisateur, la contrainte OCL suivante peut venir compléter cette représentation de l'inverse.

```
(=>)  Context diriger
      inv: être_dirigé.allInstances()->exists(e | e.domain=self.range
                                                  and e.domain=self.range)

(=>)  Context être_dirigé
      inv: diriger.allInstances()->exists(d | d.domain=self.range and d.domain=self.range)
```

5.4 Conclusion

Dans ce chapitre ont été définies les notions, et sous-notions adjacentes, de *relation* et de *slot* ainsi que leurs *règles d'instanciation de notions* dans chacun des modèles OWL, GCs et UML. Le tableau ci-dessous résume l'« instanciabilité » - totale, partielle ou nulle - de ces notions dans les trois modèles de notre système de connaissance.

Notions ╲ Modèles	OWL	GCs	UML
Relation	o	o	o
symétrie	o	o	o
transitivité	o	o	o
héritage	o	o	o
équivalence	o	o	o
inverse	o	o	o
Slot	o	o	o
héritage	o	o	x
équivalence	o	o	o

Connaissance totalement (o), partiellement (.) ou non (x) modélisable

Chapitre 6

Transition de modèles appliquée aux connaissances factuelles

LES notions d'*individu*, de *donnée*, de *lien* et d'*attribution* permettent de modéliser les assertions qui composent les connaissances factuelles d'un domaine. Ces dernières n'ont de sens qu'en rapport aux connaissances structurelles de la modélisation du domaine. Ainsi, pour utiliser conjointement les modèles OWL, GCs et UML dans notre système de connaissance, ces quatre notions sont instanciables dans chacun de ces modèles afin de modéliser les assertions des différents éléments génériques composant les connaissances structurelles d'une modélisation.

Ce chapitre définit dans une première section les notions d'*individu* et de *donnée* au sein de notre système de connaissance, et présente les règles d'instanciation de ces notions dans les trois modèles. Une seconde section définit les notions de *lien* et d'*attribution* telles qu'elles sont considérées dans notre système de connaissance, et présente les règles d'instanciation de ces deux notions .

Notons que les connaissances modélisées d'un domaine ont pour vocation à être exploitées par la suite. « *Representation is a stylised version of the world. Depending on how to be used, a representation can be quite simple, or quite complex.* » [Charniak et McDermott, 1985], ou encore « *représenter, c'est coder en prenant en compte les utilisations* » [Kayser, 1997]. Ainsi, nous utiliserons par la suite de manière alternative l'expression *base de faits* pour désigner les connaissances factuelles modélisées du domaine. Par extension, nous utiliserons l'expression *base de connaissances* pour désiger à la fois les connaissances structurelles et factuelles du domaine.

Sommaire

6.1 Les notions d'*individu* et de *donnée* en OWL, Graphes Conceptuels et UML

6.1.1 Individu

Les *individus* sont des composants du niveau factuel d'une modélisation des connaissances. Il s'agit d'entités spécifiques se reportant aux classes du niveau structurel de la modélisation.

Définition 6.1 (Individu) *Un* individu *est une assertion d'une ou plusieurs classes. Un individu possède des* caractéristiques propres *qui sont conformes aux intensions de ces classes ; il fait partie des extensions de ces classes.*

Chaque classe, dont un individu est une assertion (on dira aussi une instance), constitue un *type* pour cet individu. Dans le cas où un individu est l'assertion de plusieurs classes, on dira qu'il est *multi-typé*. L'ensemble des types d'un individu peut être ramené à un unique « type conjonctif » (terme emprunté au modèle des GCs) pour l'individu. En effet, dans ce cas l'individu est dans l'extension de la classe d'intersection de ces différents types. Dans le cas particulier où les différents types d'un individu font partie d'une même hiérarchie de classes, le type retenu pour l'individu est le type le plus spécialisé puisque, par héritage, la classe la plus spécialisée est aussi une sorte-de la plus générale (voir la Section 4.2.1.1).

La Figure 66 présente un exemple déclarant l'individu alphand de type la classe Pilote ; la Figure 67 présente l'individu serieys comme une instance des classes Directeur et Pilote.

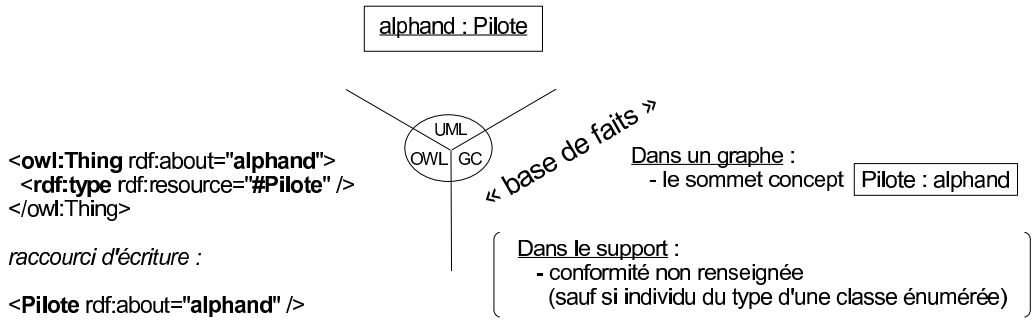


Figure 66 – L'individu alphand, instance de la classe Pilote.

En OWL, un individu est nécessairement une instance de la classe owl:Thing, qui par définition est la classe (la plus générale) de tous les individus. La propriété rdf:type permet de préciser la ou les classes les plus spécifiques dont l'individu est une assertion. L'individu est identifié à l'aide d'un attribut RDF rdf:about, éventuellement rdf:ID.

Avec le modèle UML, nous assimilons un individu à un objet. Cet objet est donc déclaré et typé au sein d'un diagramme d'objets. Le label de cet objet renseigne son ou ses types ainsi que son identifiant.

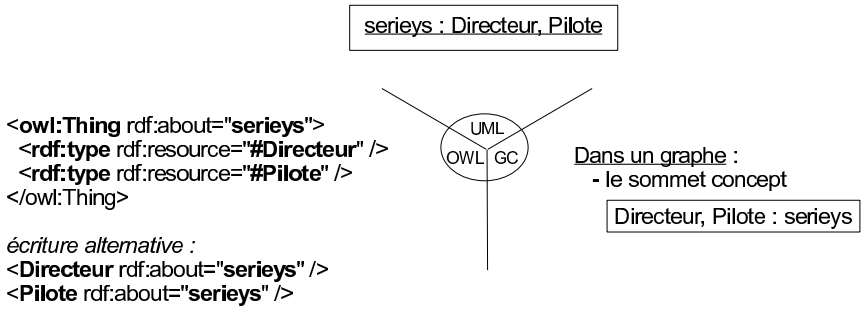


Figure 67 – L'individu serieys, instance des classes Directeur et Pilote.

Dans le modèle des GCs, un individu est modélisé par un sommet concept individuel au sein d'un graphe conceptuel. L'étiquette de ce sommet concept renseigne d'une part son type et d'autre part l'identifiant associé à l'individu. Dans le cas d'un individu multi-typé, le type associé au sommet concept est un type conjonctif qui est formé de la conjonction de ses types.

6.1.2 Identité d'un individu

Sur les trois modèles du système de connaissance, seul le modèle OWL ne fait pas l'hypothèse du nom (à valeur d'identifiant) unique pour les individus, tandis que le modèle UML et le modèle des GCs le font[1]. Ainsi, deux noms différents pour identifier des individus devraient faire référence à deux individus différents en UML et avec les graphes conceptuels, tandis qu'avec OWL ils peuvent l'être ou non.

Nous souhaitons conserver une partie de cette souplesse de modélisation proposée sans l'hypothèse du nom unique pour identifier des individus. Pour cela nous faisons l'hypothèse que des individus munis d'identifiants différents sont par défaut deux à deux différents, mais qu'il est possible de préciser explicitement que certains individus sont identiques.

6.1.2.1 Individus identiques

Définition 6.2 (Individus identiques) *Un individu i_1 est dit* identique *à un autre individu i_2 si et seulement si l'identifiant de l'individu i_2 identifie aussi i_1.*

Nous considérons que cette notion *identique* est assimilée à une idée de synonymie ou d'alias, au niveau des identifiants et non des individus eux-mêmes. En effet, un individu doit être interprété comme une entité spécifique *unique* : parler de deux individus identiques est dans l'absolu un non sens. Il s'agit donc pour cette notion de définir deux ou plusieurs identifiants qui permettent de référencer un (unique) individu.

Propriété 13 *L'axiome identique entre deux individus est symétrique.*

[1] On parle en anglais de *Unique Name Assumption* (UNA).

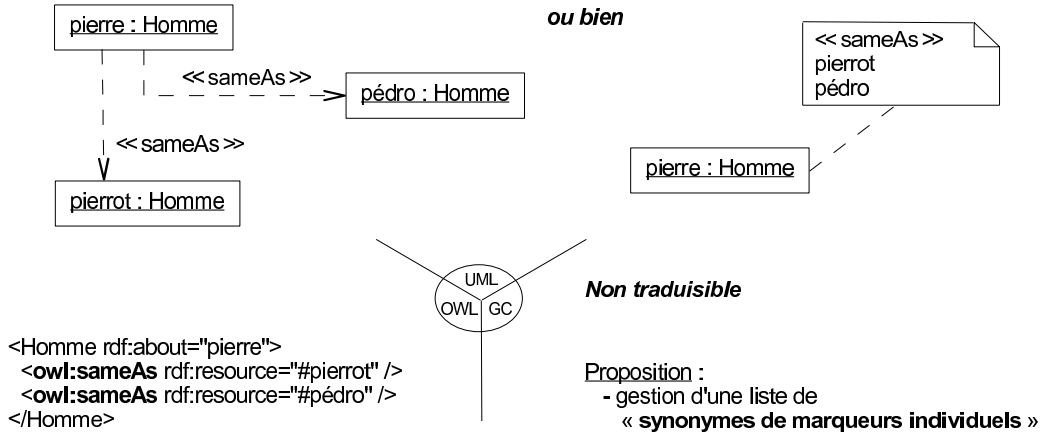


Figure 68 – Les individus pierre, pierrot et pédro sont *identiques*.

L'exemple donné en Figure 68 présente la modélisation de plusieurs (identifiants d') individus qui ne constituent en fait qu'un seul et même individu. Ici, les individus pierre, pierrot et pédro sont *identiques*.

Avec le modèle OWL, un individu identique à un ou plusieurs autres est modélisé par la propriété owl:sameAs. Une assertion de owl:sameAs constitue le prédicat d'un triplet RDF dont le sujet est l'individu concerné et l'objet est l'individu identique (à gauche sur la figure).

Dans le modèle des GCs, la notion d'individus identiques ne semble pas exprimable. Il pourrait être envisager d'utiliser explicitement un type de relation, identique par exemple. Cependant, cette relation serait mise « au même niveau » que les relations d'une modélisation et non comme une notion de modélisation à part entière. Notons qu'un lien de coréférence ne peut être utilisé, car il ne peut exister entre deux sommets concepts individuels avec des marqueurs individuels différents. Nous suggérons la création de *listes de synonymes* de marqueurs individuels, externe au modèle des GCs, qui peuvent être prises en compte à un niveau logiciel [2] lors de l'exploitation de graphes conceptuels contenant de tels marqueurs individuels.

Pour le langage UML, nous proposons deux solutions pour modéliser cette notion. Dans une première, en partie supérieure gauche de la Figure 68, nous introduisons le stéréotype ≪sameAs≫ à utiliser sur un lien de dépendance entre deux individus identiques. Une seconde solution, en partie supérieure droite de la Figure 68, est d'utiliser un commentaire stéréotypé par ≪sameAs≫, qui contient la liste des individus qui lui sont identiques : une liste de synonymes. Remarquons que les commentaires n'étant pas pris en compte au niveau logiciel lors d'importation/exportation en XMI des diagrammes UML, nous évitons d'utiliser en pratique cette proposition de modélisation.

[2] comme dans notre implémentation du système de connaissance au Chapitre 8

6.1.2.2 Individus différents

Définition 6.3 (Individus différents) *Un individu i_1 est dit* différent *d'un autre individu i_2 si et seulement si l'identifiant de l'individu i_2 n'identifie pas i_1.*

Cette notion *différent* agit, comme pour la notion *identique*, au niveau des identifiants d'individus. Par définition, un lien différent entre deux individus est symétrique.

La Figure 69 présente l'individu identifié par pierre, qui est considéré comme différent de l'individu identifié par pilou.

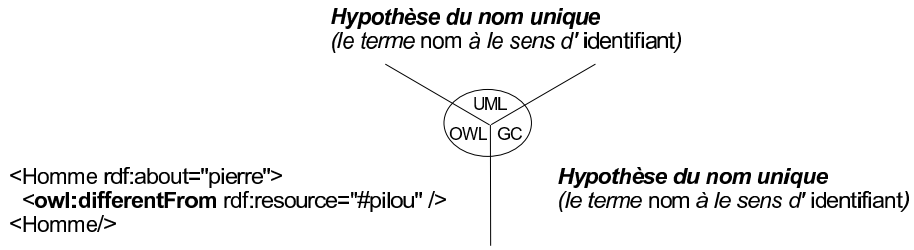


Figure 69 – Les individus pierre et pilou sont *différents*.

En OWL, un individu différent d'un autre est modélisé par la propriété owl:differentFrom, dont une assertion constitue le prédicat d'un triplet RDF ayant pour sujet l'individu concerné et pour objet l'individu différent (à gauche sur la figure).

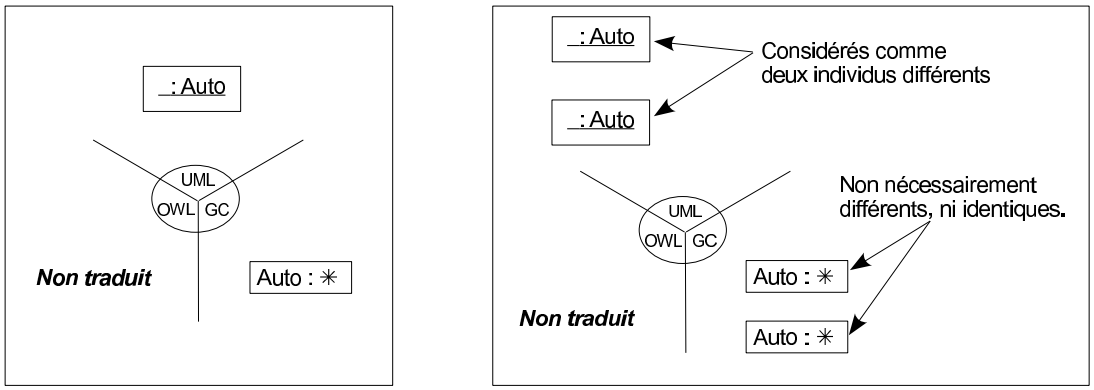
Pour le modèle UML ou le modèle des GCs, cette notion de différence entre individus est implicite. Cela est dû à l'hypothèse du nom unique qui est faite avec ses deux approches de modélisation. Le terme « nom » est à interpréter par *identifiant*.

6.1.2.3 Individu anonyme

En modélisation des connaissances, il peut être intéressant de spécifier dans une base de faits l'existence d'un individu sans pour autant en connaître son identité. Il ne s'agit cependant pas de générer un base de connaissances « floue », mais incomplète. Le travail d'un raisonneur pourrait par exemple aider l'utilisateur à identifier cet individu en le considérant comme identique à un individu non anonyme.

Définition 6.4 (Individu anonyme) *Un individu est dit* anonyme *si et seulement si il ne possède pas d'identifiant et n'est identique à aucun individu non anonyme.*

Notons qu'un individu anonyme dispose d'un type, a minima Thing, et que dans l'absolu il ne peut être ni considéré comme identique ni comme différent d'un ou plusieurs autres individus. Nous considérons un individu anonyme comme une *variable* au sens de la logique classique : une variable, d'un type donné, peut référencer successivement - c'est-à-dire de manière variable - différents individus de même type. L'exemple en Figure 70(a) présente un individu anonyme de type Auto.



(a) Individu anonyme. **(b)** Gestion de plusieurs individus anonymes.

Figure 70 – La notion d'*individu anonyme*.

Avec le modèle OWL, un individu anonyme n'est pas modélisable car toute assertion de classe doit posséder un identifiant (via un attribut rdf:about ou rdf:ID). En effet, tout document OWL doit respecter la structuration RDF en triplets <*sujet, prédicat, objet*>, avec ici <individu, type, nom_classe> (c'est-à-dire l'individu, nécessairement identifié pour constituer le sujet, est une instance de la classe mentionnée).

Pour le modèle UML, un individu anonyme est modélisé par un objet qui n'est pas nommé. C'est-à-dire que le label associé au rectangle représentant l'individu anonyme est formé directement du caractère ':' suivi de son ou ses types ; le champ correspondant au nom est laissé vide.

Il existe cependant une distinction notoire d'interprétation entre un individu anonyme modélisé dans le modèle des GCs et un individu anonyme modélisé en UML. Alors que, *par défaut*, dans le modèle des GCs deux individus anonymes ne peuvent être interprétés ni comme différents ni comme identiques, dans le modèle UML ils sont nécessairement considérés comme différents. Cette distinction, illustrée en Figure 70(b), devra être prise en compte lors de raisonnements sur les connaissances modélisées dans le système.

6.1.3 Donnée

Une donnée est un élément « brut » du niveau factuel d'une modélisation, comme l'entier 42 ou la chaîne de caractère "Hello world".

Définition 6.5 (Donnée) *Une* donnée *est un élément dans l'extension d'un ou plusieurs datatypes. Chacun de ces datatypes constituent un* type *pour la donnée. Une donnée est caractérisée par une* valeur.

Avec le modèle OWL et le modèle UML, nous associons à la notion de donnée la notion de valeur de datatype, issue de ces deux modèles.

Comme mentionné au Chapitre 4, classe et datatype sont relativement proches d'un point de vue conceptuel ; ces deux ressources sont modélisées de manière analogue par

un type de concept lors de l'application des règles d'instanciation de notions dans le modèle des GCs. Il en résulte que dans ce modèle nous modélisons une donnée par un sommet concept, à la manière d'un individu. L'analogie au niveau assertionnel entre individu et donnée s'arrête là, car un marqueur individuel d'un sommet concept ne peut cependant pas constituer la valeur d'une donnée. En effet, le nommage d'un marqueur individuel n'a aucune importance - sauf peut être à un niveau implémentation -, seul son aspect unique compte, tandis que par exemple 42 peut aussi bien identifier un entier, un nombre réel ou encore une chaîne de caractères.

Notons qu'une donnée générique, c'est-à-dire ne disposant pas de valeur, ne peut être modélisée en OWL ni en UML car lors de la déclaration et du typage d'une donnée la valeur qui caractérise cette donnée doit nécessairement être précisée. Le modèle des GCs considéré (Section 1.2.4) offre par contre la possibilité de modéliser une donnée générique par un sommet concept particulier appelé un sommet concept donnée non-valuée.

6.2 Les notions de *lien* et d'*attribution* en OWL, Graphes Conceptuels et UML

Les liens et attributions - ou plus exactement les instances de ces deux notions - permettent dans une base de faits de caractériser des individus en les liant à d'autres individus ou à des données.

6.2.1 Lien

Un lien est un composant du niveau factuel d'une modélisation des connaissances qui associe deux individus entre eux.

Définition 6.6 (Lien) *Un* lien *entre deux individus est une assertion d'une relation binaire. Cette relation constitue le* type *du lien. Un lien obéit à la définition de son type, c'est-à-dire que le* premier argument *du lien est un individu dans l'extension du domaine de la relation et le* second argument *un individu dans l'extension du co-domaine.*

La Figure 71 présente un exemple de lien : une assertion de la relation diriger entre les individus serieys et TRMR ; la relation diriger est présentée en Figure 65.

En OWL, nous faisons correspondre à la notion de lien celle de rôle d'objet. Ainsi, l'identifiant de la relation, qui est le type du lien, est utilisé comme prédicat d'un triplet RDF où le sujet est le premier argument du lien et l'objet le second argument.

Dans le modèle des GCs, un lien est modélisé par un sommet relation qui est typé par un type de relation représentant une *relation* (voir la Section 5.1). Ce sommet relation a donc pour premier voisin un sommet concept dont le type est le premier argument du type de relation ou de tout subsumé et pour second voisin un sommet concept dont le type est le second argument ou de tout subsumé.

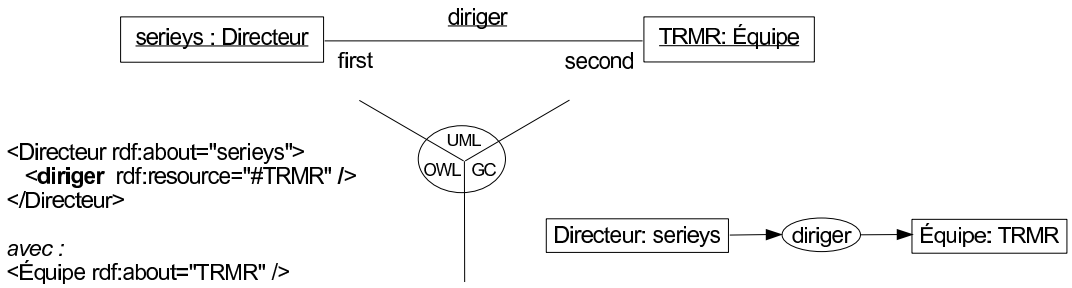


Figure 71 – Lien entre deux individus.

Avec le modèle UML, un lien est modélisé par un trait continu entre deux objets (c'est-à-dire deux individus). Ce trait possède un label formé d'un texte souligné avec l'identifiant du type du lien. Il n'y a pas d'ambiguïté quant à l'ordre des arguments du lien dans cet exemple, car il suffit de « regarder » leurs types et se reporter à la définition de la relation qui est le type du lien. La relation diriger ayant pour domaine et co-domaine respectivement les classes Directeur et Équipe, l'individu serieys étant une instance de Directeur est interprété comme le premier argument du lien en Figure 71. Il en va du même constat pour pour le second argument. De manière générale, nous représentons explicitement en UML l'ordre des arguments d'un lien en nommant les extrémités de ce lien par first et second. Ceci nous permet de traiter cet ordre dans le cas général, notamment pour les assertions d'une relation dont le domaine et le co-domaine sont la même classe (voir par exemple la relation descendant en Section 5.2), ou plus généralement sont des classes issues d'une même hiérarchie et que le le premier et second arguments du lien sont dans l'extension de la classe la plus spécialisée.

6.2.2 Attribution

Une attribution est un composant du niveau factuel d'une modélisation des connaissances qui relie un individu à une donnée.

Définition 6.7 (Attribution) *Une* attribution *est une assertion d'un slot qui lie un individu à une donnée. Ce slot constitue le* type *de l'attribution. Une attribution obéit à la définition de son type, c'est-à-dire que le* premier argument *de l'attribution est un individu dans l'extension du domaine du slot et le* second argument *une donnée dans l'extension du co-domaine.*

On dira pour une attribution de type S de premier argument i et de second argument d, que l'individu i a pour attribution de (type) S (la donnée ou valeur) d.

La Figure 72 présente trois attributions ayant toutes pour premier argument l'individu serieys. Une première attribution de type nom lie serieys à la donnée "Serieys", ce même individu a pour attribution de type prénom la valeur "Dominique", et a pour attribution de âge 42.

En OWL, nous faisons correspondre à la notion d'attribution celle de rôle de type de donnée. Ainsi, l'identifiant du slot, qui est le type de l'attribution, est utilisé comme

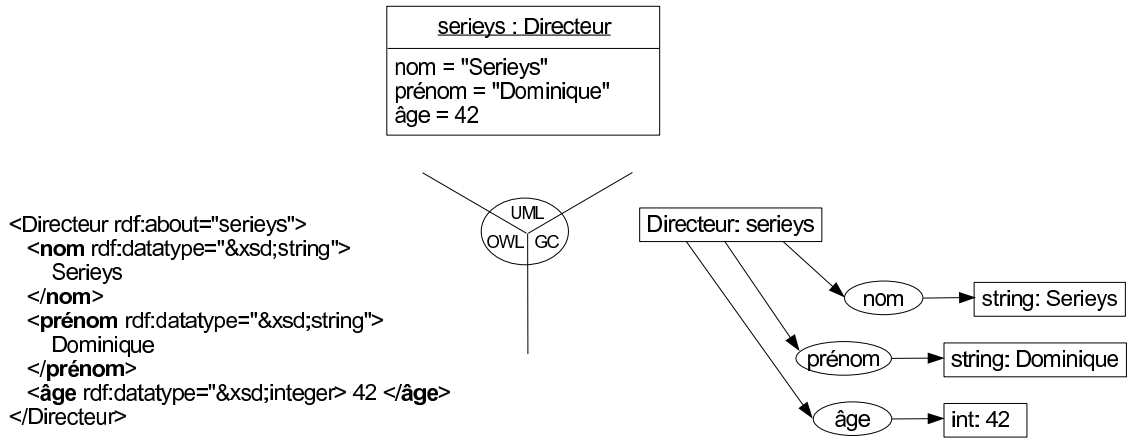


Figure 72 – Attributions de types nom, prénom et âge caractérisant l'individu serieys.

prédicat d'un triplet RDF où le sujet est le premier argument de l'attribution (un individu) et l'objet le second argument (une donnée).

Dans le modèle des GCs, une attribution est modélisée par un sommet relation qui est typé par un type de relation représentant un *slot* (voir la Section 5.1). Ce sommet relation a donc pour premier voisin un sommet concept dont le type est le premier argument du type de relation ou de tout subsumé et pour second voisin un sommet concept dont le type est le second argument ou de tout subsumé.

Avec le modèle UML, une attribution est modélisée par une valeur d'un attribut. Les attributions qui caractérisent un même individu sont regroupées dans le compartiment des valeurs d'attributs de l'objet correspondant à l'individu.

6.2.3 Lien *versus* Attribution

D'un point de vue conceptuel, les liens et les attributions sont proches. Un lien permet d'associer un individu à un autre, tandis qu'une attribution permet d'associer une donnée à un individu. Avec le modèle OWL, la modélisation d'un lien ou d'une attribution est d'ailleurs très proche syntaxiquement en XML/RDF. De plus, dans le modèle des GCs les modélisations choisies précédemment pour modéliser un lien ou une attribution sont là aussi très proches syntaxiquement. La Figure 73 illustre ces deux remarques, en regroupant sur une même figure la modélisation du lien de la Figure 71 et la modélisation des attributions de la Figure 72.

Cependant, même si lien et attribution sont proches conceptuellement, ne pas les représenter de manière analogue offre une facilité de lecture de la modélisation. C'est le cas de UML, comme présenté en Figure 73, où les caractéristiques internes de l'individu, que sont les attributions, et les caractéristiques externes, que sont les liens avec d'autres individus, sont facilement distinguables d'un point de vue représentation.

Avec l'extension du modèle des GCs *emboîtés typés* [Chein et Mugnier, 1997; Chein *et al.*, 1998], il est possible dans un graphe conceptuel d'*emboîter* à l'intérieur du sommet

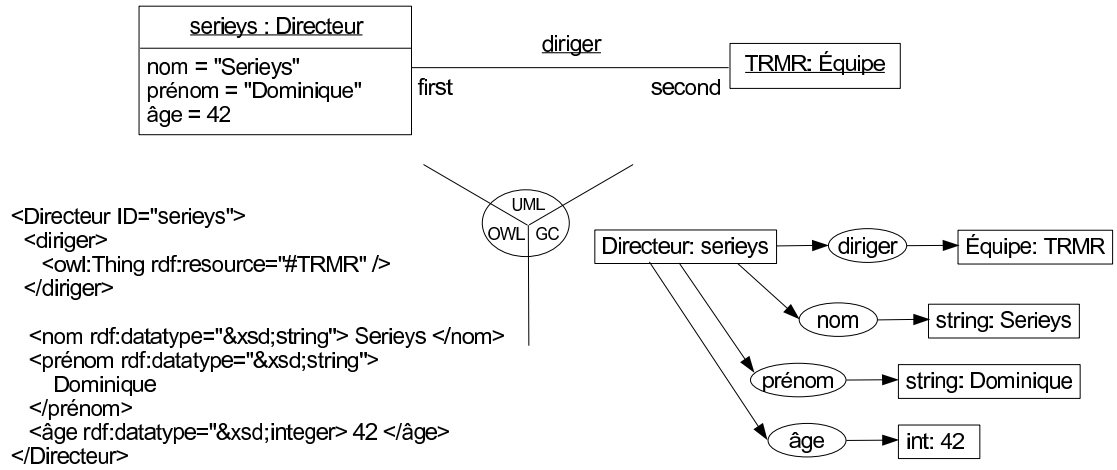


Figure 73 – Lien *versus* attribution.

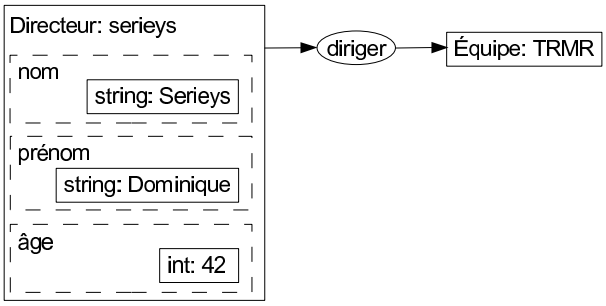


Figure 74 – Attributions et modèle des GCs *emboîtés typés*.

concept modélisant un individu les attributions qui le caractérisent. La Figure 74 reprend la représentation faite en graphes conceptuels sur la Figure 73, et la met sous la forme d'un graphe emboîté typé. À chaque slot correspond un *type d'emboîtement*, ainsi pour une attribution a d'un type S caractérisant un individu i, le sommet concept représentant a est emboîté dans un emboîtement de type S à l'intérieur du sommet concept représentant l'individu i.

Rappelons que l'utilisation d'emboîtement ne rend pas le modèle des GCs plus expressif (ni moins expressif d'ailleurs), mais offre une lisibilité accrue des graphes conceptuels.

6.3 Conclusion

Dans ce chapitre ont été définies les notions, et sous-notions adjacentes, d'*individu*, de *donnée*, de *lien* et d'*attribution* ainsi que leurs *règles d'instanciation de notions* dans chacun des modèles OWL, GCs et UML. Le tableau ci-dessous résume l'« instanciabilité » - totale, partielle ou nulle - de ces notions dans les trois modèles de notre système de connaissance.

Notions ╲ Modèles	OWL	GCs	UML
Individu	o	o	o
multi-typage	o	o	o
identique	o	o	o
différent	o	o	o
anonyme	x	o[a]	o[b]
Donnée	o	o	o
multi-typage	o	o	o
anonyme	x	o	x
Lien	o	o	o
Attribution	o	o	o

Connaissance totalement (o), partiellement (.) ou non (x) modélisable

[a],[b] modélisation sensiblement différente (entre OWL et GCs d'une part, et UML d'autre part)

Les modèles OWL, GCs et UML de par les transitions de modèles définies aux Chapitres 4, 5 et 6 partagent un ensemble conséquent de notions communes, notamment au niveau des connaissances factuelles d'une modélisation. Ceci nous permet dans les chapitres suivants de profiter, en plus d'une complémentarité de modélisation induite par les transitions de modèles, d'une complémentarité de raisonnement entre les modèles OWL, GCs et UML issue de leur utilisation conjointe.

Chapitre 7

De la transition de modèles à l'*interrogation*, la *dédution* et la *vérification* de connaissances

UNE application directe des règles d'instanciation de notions dans les modèles OWL, GCs et UML, présentées dans les trois chapitres précédents, consiste à utiliser conjointement les capacités de raisonnements de ces différents modèles sur des connaissances modélisées dans notre système de connaissance.

L'idée développée dans ce chapitre en terme de raisonnement est d'être capable d'utiliser « simplement » chaque modèle dans son cadre de conception et obtenir du système de connaissance des raisonnements globaux aux capacités inférentielles supérieures à ce qui pourrait être obtenu en n'utilisant qu'un seul modèle. Cet objectif est motivé par le fait que chaque modèle du système de connaissance offre, selon ses spécificités, des possibilités de raisonnements qui lui sont propres pour un ensemble de types de raisonnements. Par conséquent, des raisonneurs conçus sur et pour un modèle donné s'acquittent de ces raisonnements de manière efficace.

Nous regroupons en trois catégories les différents types de raisonnements des différents modèles exploitables au sein du système de connaissance. Une première catégorie concerne la possibilité d'*interroger* les connaissances modélisées, une deuxième permettant de *déduire* de nouvelles connaissances, et une troisième de *vérifier* des connaissances modélisées.

La Figure 75 donne le schéma de principe d'une utilisation conjointe et complémentaire des trois modèles OWL, GCs et UML pour raisonner sur les connaissances modélisées dans le système de connaissance. Le principe est le suivant. Pour un type de raisonnement donné à effectuer sur les connaissances modélisées dans le système de connaissance, il s'agit d'utiliser le modèle du système adapté ou le plus adapté. Une fois un raisonnement effectué dans un modèle, la conséquence résultante - comme par exemple la déduction d'une nouvelle connaissance - une fois intégrée dans le système de connaissance est répercutée dans les autres modèles du système par le biais des *règles d'instanciation de notions* développées aux Chapitres 4, 5 et 6. Suivant une telle approche, une

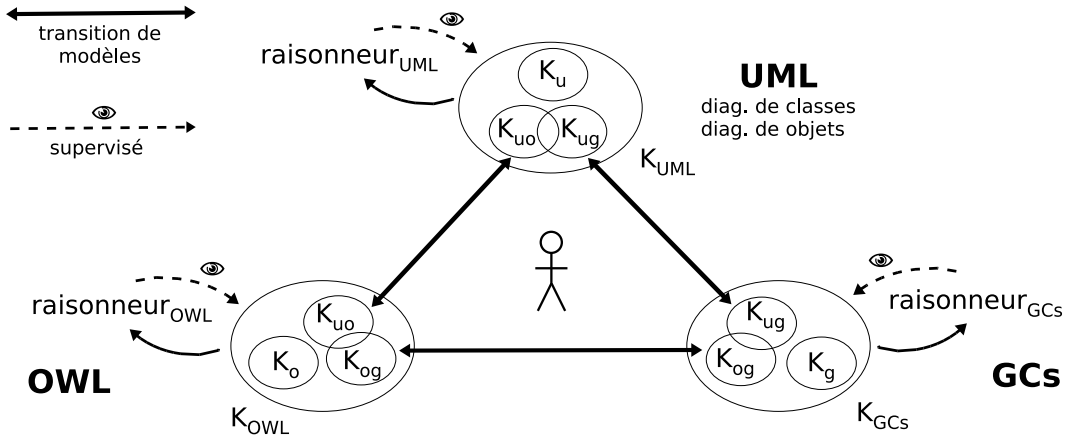


Figure 75 – Schéma de principe : complémentarité des représentations et des raisonnements avec les modèles OWL, GCs et UML.

interaction est alors possible entre les différents types de raisonnements réalisés par des raisonneurs dans les différents modèles. Le Chapitre 8 présente une implémentation du système de connaissance basé sur un choix de raisonneurs pour effectuer concrètement les différents raisonnements possibles présentés dans ce chapitre.

L'acceptation de mettre à jour les connaissances modélisées dans le système de connaissance par des conclusions produites lors de raisonnements dans un modèle peut demander à être supervisée. Étant donné que chaque modèle ne perçoit qu'un sous-ensemble des connaissances modélisées dans le système de connaissance, l'utilisateur peut être amené à juger de la crédibilité des raisonnements effectués au sein du système de connaissance pris dans sa globalité. En effet, bien que les raisonnements dans chaque modèle aient un comportement connu, l'adéquation et la complétude des raisonnements dans le système de connaissance ne peuvent être garanties. L'intervention de l'utilisateur, faisant du système de connaissance un système anthropocentré, a donc comme premier objectif d'harmoniser et éventuellement d'enchaîner des raisonnements. Cette « inclusion » de l'utilisateur dans le système de connaissance a comme second objectif de prendre en considération ses exigences et expériences, de sorte que le système constitue un environnement d'aide à la gestion et l'exploitation des connaissances plutôt qu'une « boîte noire » rendant l'utilisateur passif.

Ce chapitre s'articule de la manière suivante. Dans un premier temps, nous présentons d'un point de vue méthodologique différentes phases d'utilisation du système de connaissance pour obtenir de ce dernier une synergie entre les modèles OWL, GCs et UML en terme de raisonnements. Dans un second temps, en s'appuyant sur un exemple, nous présentons les trois catégories de raisonnements, interrogation, déduction et vérification, qui sont réalisables dans le système de connaissance en mettant à profit la complémentarité inférentielle des modèles OWL, GCs et UML.

Sommaire

7.1 Méthodologie d'exploitation des connaissances

L'approche méthodologique d'utilisation des modèles OWL, GCs et UML a pour objectif de constituer un environnement pour modéliser les connaissances d'un domaine et raisonner sur ces connaissances.

Les connaissances sont modélisées au sein du système de connaissance composé de ces trois modèles, et sont exploitables dans chaque modèle par application des règles d'instanciation de notions proposées aux Chapitres 4, 5 et 6. Du point de vue du système de connaissance, ces règles ont pour but de faire transiter les connaissances qui y sont modélisées d'un modèle à un autre. Il est alors possible de profiter dans chacun des modèles de ses capacités de raisonnements, créant ainsi une synergie entre différents raisonneurs associés aux différents modèles.

La Figure 76 présente les différentes phases d'utilisation du système de connaissance pour arriver à cet objectif.

Le « point d'entrée » (étape 0) dans le système de connaissance peut se faire dans l'un des trois modèles. Il peut s'agir de la création d'une modélisation, auquel cas l'utilisateur privilégiera certainement UML, ou de l'importation d'une modélisation déjà existant en OWL, GCs ou UML. Cette étape 0 consiste à intégrer dans le système les connaissances modélisées en provenance d'un modèle initial choisi. On parle d'*intégration* de connaissances dans le système, pour une application « inverse » des règles d'instanciation de notions. C'est-à-dire en considérant les connaissances K_m déjà modélisées, et donc considérées comme instanciées, dans un modèle m (ici initial), il s'agit de les incorporer au système de connaissance de manière à ce qu'en partant d'une modélisation vierge dans m et par application des règles d'instanciation de notions les connaissances modélisées dans m soient K_m.

Passée cette étape, les connaissances modélisées sont réparties [1] au sein des trois modèles par applications des règles d'instanciation de notions (étape 1). Chaque modèle accueille donc, en fonction de ses capacités de représentation, une modélisation plus ou moins complète qui peut avoir des conséquences ultérieures quant à la crédibilité et son acceptation par l'utilisateur d'un raisonnement produit sur ces connaissances partielles. Il est par conséquent nécessaire pour cette étape 1 de prendre en considération les connaissances modélisées dans le système mais non prises en comptes, car non modélisables et donc non accessibles, dans chaque modèle. L'utilisateur est informé de la liste de ces connaissances non prises en compte pour chacun des outils ; il pourra donc en tenir compte lors de l'étape 3 par exemple.

À partir des connaissances qui lui sont accessibles, un raisonneur d'un modèle peut, en étape 2, effectuer dans son modèle des raisonnements qui lui sont propres (voir les Sections 7.2, 7.3 et 7.4). En parallèle ou en complément de cette étape, l'utilisateur peut éventuellement compléter la modélisation depuis un modèle (étape $2'$), avec lequel par exemple un type de connaissance est modélisable en comparaison du modèle initial.

[1] Il ne s'agit pas bien sûr d'une répartition disjointe : une connaissance donnée peut être modélisée dans un seul, dans deux ou dans les trois modèles du système de connaissance (en fonction des règles d'instanciation de notions) ; le terme « répartition » est là pour mettre en avant la complémentarité représentationnelle.

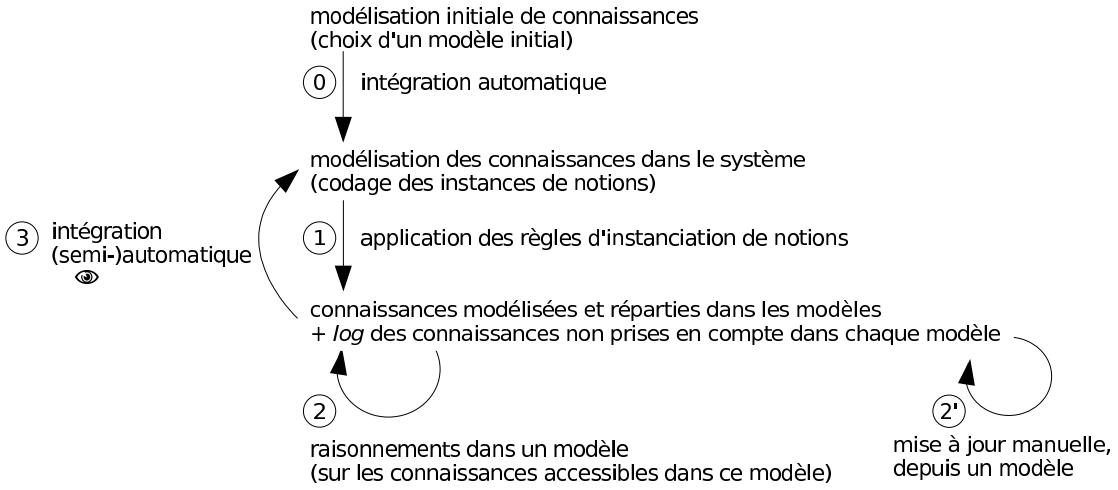


Figure 76 – Les différentes phases d'utilisation du système de connaissance.

L'étape 3 consiste à mettre à jour, s'il y a lieu, les connaissances modélisées dans le système en fonction des conclusions produites lors de raisonnements effectués par les raisonneurs. Bien que l'intégration des connaissances résultantes à des raisonnements puisse à cette étape être automatisée, nous la considérons comme *supervisée*. En effet, comme mentionné après l'étape 1 un raisonnement peut être, du point de vue de l'ensemble des connaissances modélisées dans tout le système, incomplet ou inconsistant du fait de connaissances non accessibles. Il est donc nécessaire pour un raisonnement produit qu'il soit accepté par l'utilisateur et/ou les autres modèles avant d'être intégré dans le système de connaissance. L'acceptation par l'utilisateur se fera notamment après consultation des connaissances non prises en compte par le modèle effectuant le raisonnement. L'acceptation par les autres modèles dépendra des conclusions résultantes de ce raisonnement, à savoir si les connaissances modélisées dans les autres modèles resteront valides sous l'hypothèse d'intégrer cette conclusion (voir la Section 7.4).

Exemple d'utilisation

Nous nous proposons de présenter à travers un exemple de modélisation en Figure 77 (structure) et en Figure 78 (faits) des modes d'utilisation possibles de notre système de connaissance. Il s'agit d'illustrer la complémentarité inférencielle des modèles OWL, GCs et UML ; la complémentarité représentationnelle de ces modèles étant immédiate du fait des règles d'instanciation de notions vues aux Chapitres 4, 5 et 6.

Cette synergie en modélisation et raisonnement entre les différents modèles du système de connaissance, obtenue par le biais des transitions de modèles, peut de façon non exhaustive être vue de la manière suivante.

Le modèle UML dispose d'un langage très largement visuel et intuitif pour l'utilisateur de par ses schématisations sous forme de diagrammes. Il peut être privilégié par

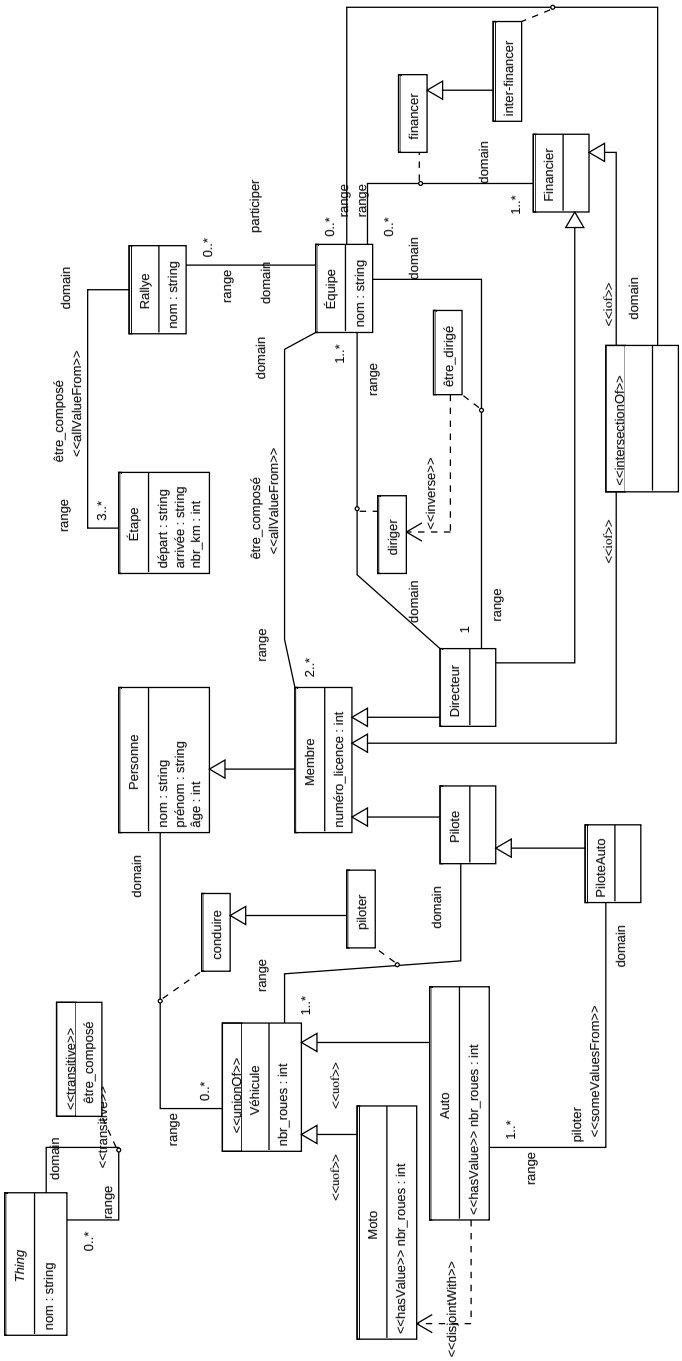


Figure 77 – Exemple de modélisation (version initiale UML), connaissances structurelles.

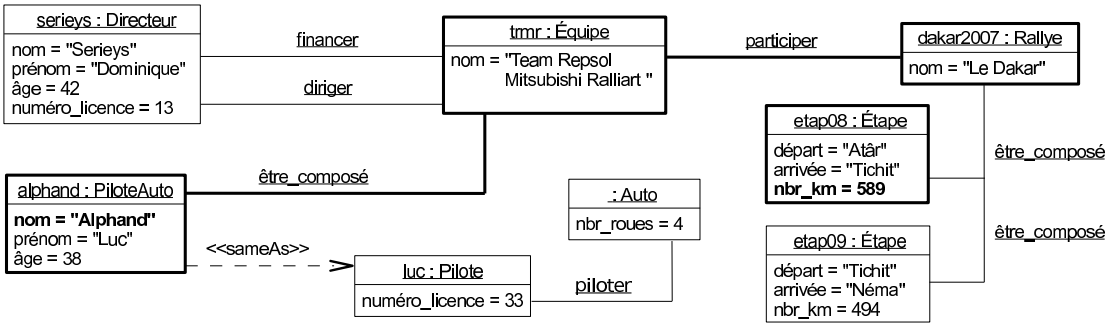


Figure 78 – Exemple de modélisation (version initiale UML), connaissances factuelles.

l'utilisateur, comme sur cet exemple en Figures 77 et 78, lors de la création ou la mise à jour « manuelle » d'une modélisation des connaissances d'un domaine, ou pour interpréter des conclusions produites lors de raisonnements. Bien que certains types de raisonnements soient possibles à partir de OCL, peu d'outils logiciels en permettent l'exploitation.

Le modèle OWL basé sur des logiques de descriptions offre la possibilité d'aider l'utilisateur, par des mécanismes de *classification* et d'*instanciation*, à (ré-)organiser en une ontologie la structure d'une modélisation au cours de son cycle de vie, ainsi qu'à vérifier la validité de la structure ou des faits d'une modélisation.

Le modèle des GCs offre quant à lui la possibilité, via l'opérateur de *projection*, de rechercher des connaissances parmi celles modélisées. De plus et de façon complémentaire aux deux autres modèles, notamment le modèle OWL, ce modèle permet aussi l'application de règles (au sens de celles en logique classique) et/ou la vérification de contraintes.

Les différents types de raisonnements réalisables par les modèles du système sont regroupées en trois catégories *interrogation*, *déduction* et *vérification*, présentées respectivement dans les Sections 7.2, 7.3 et 7.4.

7.2 Interrogation

Il est intéressant de constater que si les modèles basés sur des logiques de descriptions - notamment OWL - peuvent disposer d'une grande richesse expressive pour modéliser avec finesse un domaine donné et que des raisonneurs permettent de vérifier et/ou d'aider l'utilisateur à affiner une modélisation dans ce modèle, il peut sembler surprenant que ces modèles ne disposent pas de moyen intrinsèque d'interrogation sur la base de connaissances que constitue une telle modélisation. L'article [Rosati, 2007] fait un état de l'art intéressant sur la difficulté de pouvoir questionner des connaissances modélisées dans un modèle basé sur des logiques de descriptions, incluant OWL. C'est pourquoi, des modèles et outils s'appuyant sur la logique classique ont été développés en sus de ces modèles en logiques de descriptions afin de pouvoir profiter de « nouveaux » raisonnements, comme ici l'interrogation. Citons par exemple les outils et langages RQL,

SparQL ou OWL-QL (voir Section 2.4), ou encore le modèle CARIN [Levy et Rousset, 1998] combinant règles de Horn et la logique de description $\mathcal{ALCNR}$.

OCL dispose de mécanismes en logique classique offrant la possibilité, via certaines primitives pour l'expression de contraintes comme select, reject ou collect, d'interroger des connaissances modélisées en UML. Afin de rendre plus accessible l'utilisation de OCL, dont la syntaxe est peu intuitive, le travail théorique [Warmer et Kleppe, 2003] propose un syntaxe « SQL-like » pour OCL. Cependant comme mentionné précédemment, peu d'outils logiciels permettent l'exploitation de formules OCL. D'une part car UML est généralement utilisé dans son but original, à savoir l'aspect purement représentatif. Ainsi les outils UML sont essentiellement des éditeurs de diagrammes. D'autre part des raisonnements effectués via OCL sont en général indécidables, car basés sur l'ensemble de la logique du premier ordre.

Au sein de notre système de connaissance, nous nous tournons donc essentiellement vers le modèle des GCs pour interroger les connaissances modélisées d'un domaine, grâce à l'opération de projection.

7.2.1 Interrogation par graphe conceptuel requête

L'interrogation dans le modèle des GCs s'opère par la définition d'un graphe conceptuel, dit requête, qui représente un ensemble de connaissances à rechercher au sein d'un graphe conceptuel factuel, c'est-à-dire au sein des connaissances factuelles d'une modélisation (voir Chapitre 1). L'ensemble des projections du graphe conceptuel requête sur un graphe conceptuel factuel constitue l'ensemble des réponses à la question.

La Figure 79 présente un exemple de graphe conceptuel requête (Section 1.2.4.3) pour interroger les connaissances factuelles en Figure 78 définies sur les connaissances structurelles en Figure 77. Cette requête exprime l'interrogation suivante : « Existe-t-il un pilote nommé approximativement Alphan membre d'une équipe qui participe à un rallye dont au moins une étape est de plus de 500 km ? ». On rappelle que l'extension du modèle des GCs en Section 1.2.4 permet de prendre en compte des données en graphes conceptuels.

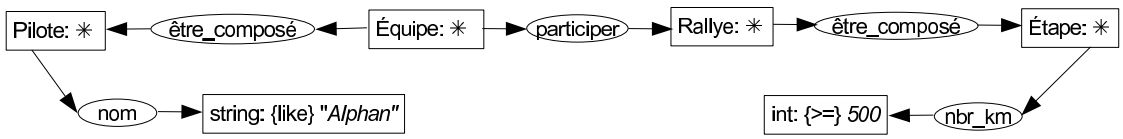


Figure 79 – Exemple d'interrogation, écrite dans le modèle des GCs par un *graphe conceptuel requête*.

La Figure 80 représente la modélisation des connaissances factuelles en Figure 78 dans le modèle des GCs cette fois-ci. Cette modélisation en Figure 80 est obtenue par transition de modèles du modèle UML vers le modèle des GCs [2].

[2]Notons l'unique sommet concept individuel pour représenté l'individu alphand et luc puisqu'il s'agit de « deux » individus identiques. Comme mentionné au Chapitre 6, un seul marqueur individuel est utilisé et une *liste de synonymes* est créée pour cet individu ; liste qui est prise en compte à un niveau implémentation au Chapitre 8.

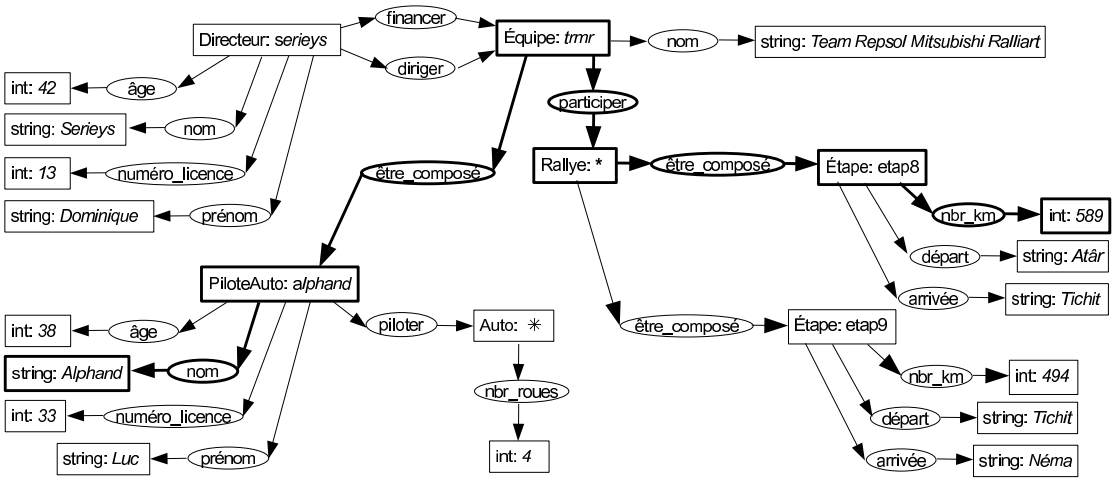


Figure 80 – Résultat (en gras) de la requête en Figure 79 sur les connaissances factuelles en Figure 78 ici modélisées par transition de modèles dans le modèle des GCs.

L'unique résultat de l'interrogation, dans le modèle des GCs, des connaissances factuelles sur la Figure 80 par la requête en Figure 79 est mis en gras sur la Figure 80. On constate notamment la projection du sommet concept [string: {like} "Alphan"] dans le sommet concept [string: "Alphand"] où le type string coïncide avec string et la chaîne de caractères "Alphan" est une sous-chaîne de "Alphand", ainsi que la projection de [int: {>=} 500] dans [int: 589]. La projection des autres sommets du graphe conceptuel dans le graphe conceptuel factuel reste équivalente à celle faite dans le modèle « classique » des GCs (Section 1.1).

7.2.2 UML pour la représentation d'interrogations

Bien que les raisonnements dans le modèle UML soient d'une manière générale indécidables [Berardi *et al.*, 2005] et en pratique intraitabilité par les outils UML, nous avons souhaiter cependant pouvoir « interfacer » une requête en UML. Ayant privilégié UML pour son aspect très visuel et intuitif d'utilisation pour l'humain, il est possible d'exprimer dans notre système de connaissance une requête en UML pour l'interrogation des connaissances factuelles d'une modélisation. Il suffit pour cela qu'un diagramme d'objets soit considéré comme une requête. C'est-à-dire que ce diagramme d'objets représente un sous-ensemble possible (à rechercher donc) des connaissances factuelles modélisées parmi l'ensemble des connaissances factuelles modélisées dans le système de connaissances. Cette considération peut être complétée en autorisant au sein du diagramme d'objets constituant un question la représentation de connaissances approchées, come l'existance d'un nombre supérieur ou égal à 500. Dans ce cas, la valeur d'un attribut peut être complétée si besoin d'un opérateur - parmi ceux du Tableau 3 - mis entre accolades pour le distinguer du reste de la valeur à proprement parlée.

L'interrogation selon UML en Figure 81 exprime la même requête que celle en Fi-

gure 79 dans le modèle des GCs. Pour ce qui est de la recherche des connaissances fac-

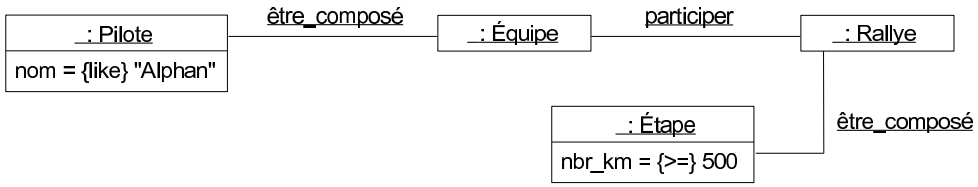


Figure 81 – Exemple d'un diagramme d'objets considéré comme une interrogation, reprenant la requête en Figure 79.

tuelles dans la modélisation répondant à la requête UML, elle ne s'effectue pas dans le modèle UML pour les raisons évoquées ci-dessus concernant OCL. Cependant, tout comme pour les connaissances structurelles et factuelles d'une modélisation, une requête est modélisée dans le système de connaissance et non seulement dans un modèle. C'est donc ensuite par transition de modèles que la requête en Figure 81 va pouvoir être effectuée dans le modèle des GCs (voir le Chapitre 8). Ensuite, par transition « inverse » de modèles, le résultat de cette requête peut être visualisé dans le modèle UML (partie en gras sur la Figure 78).

7.3 Déduction

Avant d'interroger ou de vérifier (voir la section suivante) les connaissances d'une modélisation, il peut être souhaitable en amont de faire ressortir explicitement certaines connaissances implicitement présentes dans le modélisation. Des raisonnements complémentaires dans les modèles GCs et OWL répondent à cette exigence que nous illustrons par des exemples.

7.3.1 Le modèle des GCs et l'application de règles

Pour la déduction de connaissances, le modèle des GCs dispose de règles d'inférence permettant à partir de l'existence de certaines connaissances, appelées hypothèse, de déduire d'autres connaissances, appelées conclusion. L'hypothèse d'une règle est un graphe conceptuel requête et la conclusion est un graphe conceptuel factuel (Chapitre 1).

Si certaines règles du modèle des GCs sont obtenues lors de l'application des règles d'instanciation de notions dans le système de connaissance, il est possible d'enrichir la modélisation par d'autres règles. Cet enrichissement est d'autant plus souhaitable qu'en dehors de quelques cas particuliers, comme la propriété de transitivité ou d'inverse caractérisant une relation, les règles d'inférences au sens de la logique classique ne sont en général pas modélisables en OWL. Prenons par exemple la règle R_1 en Figure 82 qui stipule que « si un directeur dirige une équipe, alors cette équipe compte parmi ses membres (c'est-à-dire est composée de) ce directeur ». La règle R_2 sur cette même figure permet de déduire que « si un directeur dirige et finance une même équipe, alors ce financement du

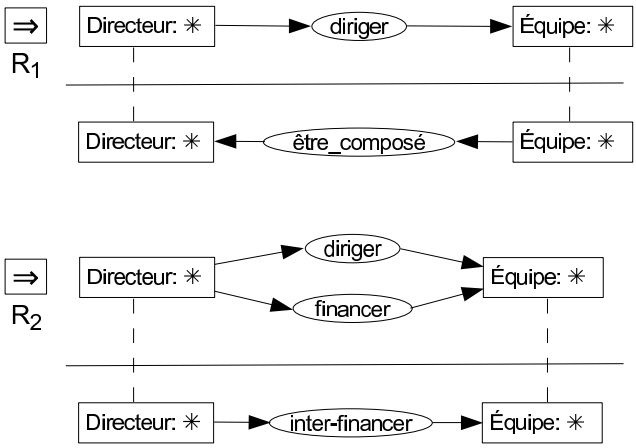


Figure 82 – Exemples de règles, écrites dans le modèle des GCs.

point de vue de cette équipe correspond à un financement interne ». Ainsi, l'application de ces règles sur les connaissances factuelles en Figure 80 ajoutent comme connaissances d'une part que l'équipe Mitsubishi est composée du Directeur de nom Serieys puisqu'il la dirige, et d'autre part que ce directeur a pour action d'inter-financer cette qu'il dirige et finance simultanément.

S'il est raisonnable de penser que l'utilisateur souhaite - sur cet exemple - créer cette règle R_2, dont l'application qui conclue sur une « spécialisation » en inter-financer la relation financer entre serieys et trmr a un sens, il n'en demeure pas moins que la définition de cette règle ne peut exister de la sorte. En effet, le relation inter-financer a été définie en Figure 77 comme ayant pour domaine la classe anonyme intersection entre Membre et Financier. Cependant, l'assertion de la relation - le sommet relation - inter-financer a pour premier argument ici une instance de la classe Directeur qui n'est pas comparable à cette classe d'intersection. La section suivante va pouvoir « débloquer » la situation.

7.3.2 Classification et instanciation de connaissances en OWL

Au cours de la modélisation des connaissances d'un domaine, des raisonnements dans le modèle OWL peuvent venir en aide à l'utilisateur.

Dans l'exemple de la section précédente, l'application du processus de classification en logiques de descriptions offre une réorganisation de la hiérarchie de classes : la classe Directeur - sous classe de Membre et de Financier - se retrouve logiquement comme une spécialisation de la classe d'intersection entre Membre et Financier. La Figure 83 constitue une extrait de cette réorganisation hiérarchique (en « version UML », visuellement plus lisible). Ainsi, après intégration de la nouvelle hiérarchie de classes de la modélisation au sein du système de connaissance, la règle R en Figure 82 est conforme à la signature de la relation inter-financer et peut être appliquée sur les connaissances factuelles de l'exemple du chapitre.

Le modèle OWL a pour vocation de traiter des connaissances incomplètes car évo-

luant sous l'hypothèse d'un *monde ouvert* lors de ses raisonnements, où les connaissances d'un domaine peuvent à tout moment être complétées. Reprenons la modélisation OWL fournie en Section 2.3.3 (très proche de celle de l'exemple de ce chapitre). Bien que certaines connaissances ne soient pas explicitement modélisées ou soient omises, un raisonneur en logiques de descriptions permet d'effectuer plusieurs déductions, comme par exemple les trois suivantes. Premièrement, la ressource Véhicule est assimilée à une classe. Ceci est justifié par le fait que Véhicule est considérée comme une sur-classe des classes Auto et Moto, il ne peut s'agir que d'une classe en OWL DL. Deuxièmement, un tel raisonneur permet de déduire que la classe Membre est une sous-classe de Personne. En effet, les instances serieys et alphand de type Membre possèdent une attribution de type prénom, or la relation prénom a pour domaine la classe Personne, donc Membre est nécessairement une sorte-de Personne. Troisièmement, il est déduit que l'individu alphand est plus particulièrement considéré comme un pilote. En effet, comme cet individu de type Membre est lié en premier argument à une assertion de la relation piloter et comme le domaine de cette relation est la classe Pilote, il en résulte que le type de alphand est spécialisé en Pilote.

Bien que les raisonnements en OWL soient effectués sous l'hypothèse d'un monde ouvert, où tout ce qui n'est pas connu reste incertain, il est possible au niveau factuel d'une modélisation OWL de « fermer »[3] en partie ce monde. En effet, les individus modélisés peuvent être supposés comme les seuls possibles ; de sorte que les individus actuellement existants sont considérés comme complets et « vrais », ainsi tout autre individu - non dérivé - sera considéré comme « faux ». Cette fermeture n'est que partielle car il n'est pas possible de préciser que l'ensemble des liens et l'ensemble des attributions sont complets et immuables ; en effet, les assertions de relations n'étant pas identifiées, elles ne peuvent être énumérées par le bais de la propriété owl:oneOf appliquée sur la ressource rdf:Property. L'optique de fermeture, appelée aussi *negation as failure*, est légitime ici dans la mesure où les connaissances factuelles sont composées de la *conjonction* d'un ensemble

[3] Ce procédé est souvent intuitivement recherché par l'utilisateur ; à l'image du raisonneur CWM [Berners-Lee, 2000] pour le Web sémantique, dont le nom signifie *Closed World Machine*.

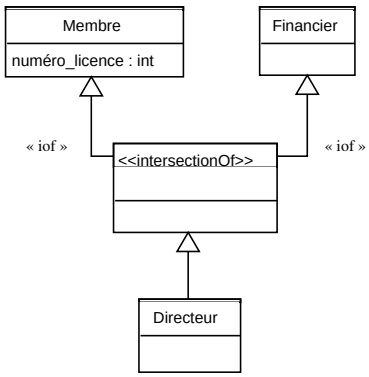


Figure 83 – Organisation modifiée de la hiérarchie des classes (extrait) en Figure 77 : Directeur est maintenant une sous-classe de l'intersection entre Membre et Financier.

de faits [Reiter, 1980]. La fermeture sur l'ensemble des individus peut s'écrire en OWL en utilisant la structure owl:oneOf de construction de classe, appliquée à la classe Thing puisqu'il s'agit de la classe la plus générale de toute modélisation. Par exemple, les individus appartenant aux connaissances factuelles de l'exemple considéré dans ce chapitre sont considérés comme les seuls avec la construction suivante :

```
<owl:Class rdf:about="#Thing">
  <owl:oneOf rdf:parseType="Collection">
    <owl:Thing rdf:about="#serieys/>
    <owl:Thing rdf:about="#alphand"/>
    <owl:Thing rdf:about="#luc"/>
    <owl:Thing rdf:about="#trmr"/>
    <owl:Thing rdf:about="#dakar2007"/>
    <owl:Thing rdf:about="#etap08"/>
    <owl:Thing rdf:about="#etap09"/>
  </owl:oneOf>
</owl:Class>
```

Cette action simple à mettre en œuvre dans un document OWL peut permettre ici d'utiliser les raisonnements terminologiques comme une aide à la modélisation, où cette fois-ci seuls les individus actuellement modélisés sont pris en compte. En effet, après les processus de classification et d'instanciation effectués sur les connaissances initialement introduites dans le système en Figure 77 et 78, on obtient par exemple comme résultat sous l'hypothèse d'un monde partiellement fermé que les classes Personne et Membre sont équivalentes.

Dans cette vision ensembliste des raisonnements terminologiques, ceci se devine assez instinctivement sur cet exemple très académique puisque Personne et Membre partagent exactement les mêmes individus serieys et alphand (luc et alphand étant identiques). Un tel résultat peut amener le concepteur de la modélisation à reconsidérer éventuellement sa modélisation du domaine concerné.

7.4 Vérification

La notion de vérification ne revêt pas la même signification dans le modèle OWL ou le modèle des GCs. Il s'agit dans le premier modèle de vérifier qu'il existe une interprétation qui satisfasse la modélisation, tandis que dans le modèle des GCs il s'agit de vérifier un ensemble de contraintes positives et/ou négatives.

7.4.1 Le modèle OWL et la validation des connaissances

Dans le modèle OWL, lors des processus de classification et d'instanciation, un certain nombre de contraintes liées à la modélisation doivent être vérifiées de sorte qu'il existe un *modèle* - au sens logiques de description - valide sur les connaissances modélisées. Il s'agit par exemple de vérifier si au niveau factuel d'une modélisation les arguments d'une assertion d'une relation ou d'un slot respectent les domaine et co-domaine de l'association en question, ou encore si les restrictions de rôles sont respectées. La création par exemple d'une classe Quadricycle définie comme l'intersection des classes Auto et Moto conduira un raisonneur en logiques de descriptions à vérifier que l'extension de cette nouvelle classe

est toujours vide ; du fait que Auto et Moto soient disjointes et que par déduction Quadricycle est équivalente à la classe owl:Nothing.

Il peut être intéressant, comme mentionné dans la section précédente, de « fermer » le monde pour effectuer de telles vérifications, afin de vérifier l'état actuel d'une modélisation. Cette action peut de plus être complétée d'une supposition du nom unique (UNA, *Unique Name Assuption*) pour les individus comme pour les classes. OWL fournit d'ailleurs la structure owl:AllDifferent permettant de différencier facilement deux à deux les éléments d'un ensemble d'individus. En mettant de coté les individus identiques (via la propriété owl:sameAs), l'hypothèse du nom unique sur les connaissances factuelles de l'exemple de ce chapitre est obtenue avec :

```
<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <owl:Thing rdf:about="#serieys/>
    <owl:Thing rdf:about="#alphand"/> <!-- #luc identique à #alphand -->
    <owl:Thing rdf:about="#trmr"/>
    <owl:Thing rdf:about="#dakar2007"/>
    <owl:Thing rdf:about="#etap08"/>
    <owl:Thing rdf:about="#etap09"/>
  </owl:distinctMembers>
</owl:AllDifferent>
```

Pour les classes, en OWL DL il faut utiliser la propriété owl:disjointWith pour explicitement différencier les classes entre elles, non issues d'une même hiérarchie. Notons que malgré cette double hypothèse de monde fermé et de nom unique, les raisonneurs en logiques de descriptions sont en général [4] pris en défaut lorsqu'il s'agit de vérifier si une contrainte de cardinalité minimale est bien respectée, comme ici sur la cardinalité minimale du nombre d'étapes dont un rallye est composé. Dans le système de connaissance, la recherche de l'existence d'« au moins un » est par contre vérifiable dans le modèle des GCs (voir Section 4.3.2.2), comme mentionnée par la contrainte positive C_1 en Figure 84 où « tout rallye doit être composé d'au moins une étape » (à défaut de pouvoir vérifier plus).

7.4.2 Le modèle des GCs et la vérification de contraintes

En complément des vérifications réalisables dans le modèle OWL et afin d'affiner la modélisation des connaissances d'un domaine, certaines contraintes positives et négatives peuvent être vérifiées dans le modèle des GCs. Une contrainte positive permet à partir de l'existence de certaines connaissances, appelées condition, de vérifier l'existence d'autres connaissances, appelées obligation. Une contrainte négative permet à partir de l'existence de certaines connaissances, appelées condition, de vérifier la non existence d'autres connaissances, appelées interdiction (ou plus simplement de vérifier la non existence de l'interdiction, sans préciser de condition). La condition d'une contrainte (positive ou négative) et l'obligation ou l'interdiction sont des graphes conceptuels requêtes (Chapitre 1).

La contrainte positive C_1 en Figure 84 est par exemple issue des règles d'instanciation de notions, de la modélisation en UML en Figure 77 vers le modèle des GCs, concernant la

[4] exception faite de Racer Pro

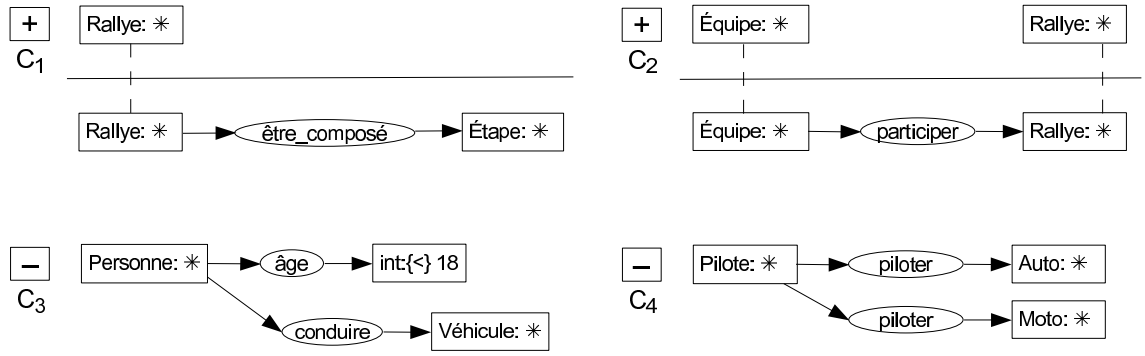


Figure 84 – Exemples de contraintes positives (C_1 et C_2) et négatives (C_3 et C_4), écrites dans le modèle des GCs.

cardinalité minimale sur la relation être_composé entre la classe domaine Rallye et la classe co-domaine Étape.

Ce complément de vérifications, en plus des containtes générées par transition, est par exemple exprimable avec la contrainte positive C_2 en Figure 84. Cette dernière permet de contraindre que « toutes les équipes participent à tous les rallyes ». Ce type de contrainte, consistant à exprimer une qualification sur une relation (ici « pour tout »), est un cas bien connu en logiques de descriptions pour ne pas être exprimable. La contrainte négative C_3 exprime que « une personne de moins de 18 ans ne peut conduire un véhicule ». Remarquons qu'une telle contrainte n'est généralement pas prise en compte par les modèles OWL ou GCs qui souffrent de ne pouvoir traiter des valeurs numériques. Ceci s'explique par le choix logique de ces modèles qui sont dépourvues de symboles fonctionnels (logique de description pour OWL et un sous-ensemble de la logique classique pour GCs). Cependant l'extension du modèles des GCs présentée en Section 1.2.4, d'un point de vue opérations de graphes [5], permet un tel traitement sur des données numériques. La contrainte négative C_4 précise que « un pilote ne peut à la fois piloter une Auto et une Moto ». Notons que cette dernière contrainte est aussi exprimable en OWL en définissant la classe - probablement anonyme - de tous les individus qui conduisent au moins une [6] Auto comme disjointe de la classe de tous les individus qui conduisent au moins une Moto.

7.5 Conclusion

Nous avons présenté dans ce chapitre de quelle manière conjointe et complémentaire les modèles OWL, GCs et UML peuvent être utilisés au sein de notre système de connaissance pour modéliser les connaissances d'un domaine et raisonner sur ces connaissances.

[5] On rappelle que les raisonnements dans le modèle des GCs peuvent être vus soit comme des inférences logiques soit comme ici des opérations de graphes (voir Chapitre 1).

[6] à la difficulté près d'interprétation des cardinalités minimales avec la plupart des raisonneurs en logiques des descriptions

Pour la modélisation de connaissances, UML est priviligié pour modéliser des connaissances de par son environnement visuel et connu ; les autres modèles sont utilisés en complément ou pour la prise en compte d'une modélisation existante. Les raisonnements réalisables dans le système de connaissance sont regroupés en trois catégories : pour interroger, pour déduire et pour vérifier des connaissances. L'objectif principal était de montrer que des raisonnements globaux supervisés au sein du système de connaissances peuvent surclasser ceux effectués localement dans un seul modèle. En effet, le système de connaissance dispose des capacités inférentielles des différents modèles - ici OWL et GCs sont exploités - et une interaction est possible entre des raisonnements issus de ces modèles par le biais de transitions de connaissances entre les modèles du système.

Chapitre 8

KR-suite : implémentation d'un système de connaissance fondé sur les modèles OWL, GCs et UML

NOTRE objectif d'utilisation conjointe et complémentaire des modèles OWL, GCs et UML en termes de modélisation et de raisonnement au sein de notre système de connaissance a fait l'objet d'une implémentation en C++ nommée KR-suite, pour « *Knowledge Representation and Reasoning suite* ». Cette implémentation est basée sur les outils logiciels suivants : Pellet pour le modèle OWL, Cogitant pour le modèle des GCs et ArgoUML pour le modèle UML.

Les différentes connaissances modélisables au sein de KR-suite sont celles issues des notions présentées aux Chapitres 4, 5 et 6, et qui sont instanciables dans ces trois outils. Les différents types de raisonnements considérés sont issus des possibilités conjointes et complémentaires de raisonnements réalisables par Cogitant et Pellet. ArgoUML, bénéficiant d'un environnement largement visuel, est uniquement utilisé pour modéliser des connaissances selon la vocation première de UML ; les raisonnements OCL dans ce modèle n'étant pas pris en charge par ArgoUML ni d'une manière générale par les autres outils UML (voir Chapitre 3).

Ce chapitre est organisé de la façon suivante. Après une présentation générale de KR-suite, nous présentons dans un premier temps la manière dont sont codées les connaissances et la structure interne de l'application. Ce codage assure notamment à ce que les connaissances restent synchronisées entre les représentations partielles - non nécessairement disjointes - dans les différents outils au cours des transitions de modèles. Dans un second temps, en s'appuyant sur un exemple, nous présentons de quelle manière sont prises en compte les différentes catégories de raisonnements dans KR-suite pour interroger, vérifier et déduire les connaissances modélisées.

Sommaire

8.1 Présentation générale de KR-suite

KR-suite est une application logicielle ayant pour but d'exploiter la complémentarité de représentation et de raisonnement des modèles de connaissances OWL, GCs et UML. Il ne s'agit pas d'un « super-modèle » de connaissance, mais d'une plateforme visant d'une part à modéliser les connaissances selon une grande expressivité et d'autre part à exploiter conjointement les capacités de raisonnements de ces modèles.

KR-suite repose sur un éditeur et/ou raisonneur particulier de chaque modèle de système de connaissance, à savoir la bibliothèque Pellet [Sirin *et al.*, 2004] pour le modèle OWL (voir Section 2.4) que nous limitons à la sous famille OWL DL, la bibliothèque Cogitant [Genest, 2007] pour le modèle des GCs (voir Section 1.3) et l'outil ArgoUML [Boger *et al.*, 2008] pour le modèle UML (voir Section 3.4). Ces outils logiciels ont été choisis suivant le meilleur compromis, selon nous, entre *liberté du code* (open source) [1], *fonctionnalités* disponibles et *efficacité*.

La Figure 85 expose le schéma du système de connaissance intégrant ces logiciels autour de l'implémentation KR-suite. Il s'agit pour l'implémentation KR-suite, codée en

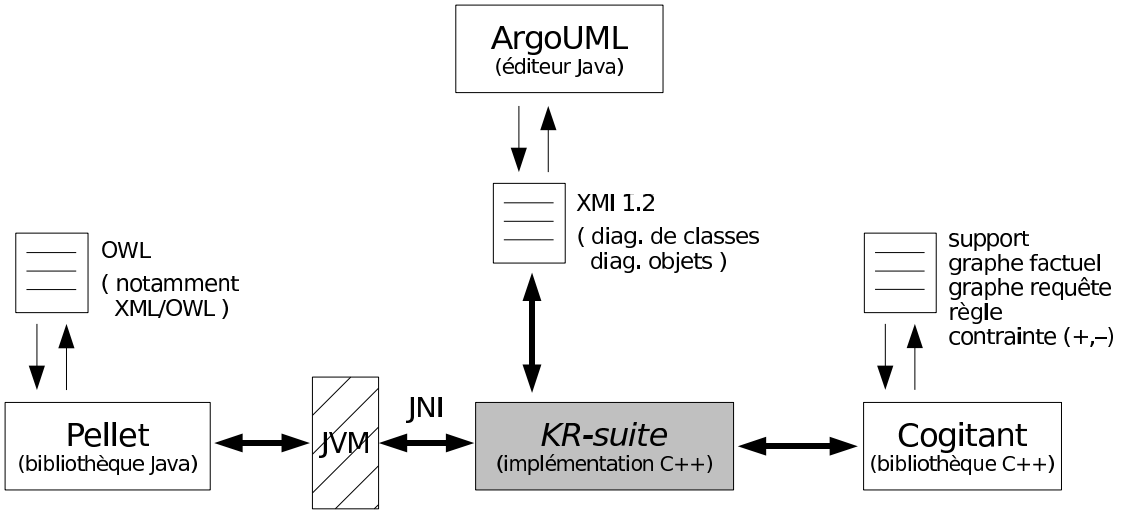


Figure 85 – Schéma d'intégration du système de connaissance basé sur les outils Pellet, Cogitant et ArgoUML.

C++, d'accéder directement aux structures de données de la bibliothèque Cogitant (codée aussi en C++) pour manipuler les connaissances modélisées selon le modèle des GCs. Cet accès « direct » permet aussi d'utiliser Cogitant dans l'import/export de fichiers codant les connaissances dans cet environnement de graphes conceptuels. L'intention est la même avec la bibliothèque Pellet, à l'obstacle près que Pellet est écrite en Java. Afin de manipuler les structures de données de Pellet nous avons eu recours à JNI. La JNI - *Java Native Interface* - est une bibliothèque qui permet à du code Java s'exécutant à l'intérieur d'une JVM [2] d'appeler et d'être appelé par des applications natives (c'est-à-dire des pro-

[1] Cogitant est sous licence *GNU GPL*, Pellet sous licence *MIT* et ArgoUML sous licence *BSD*.

[2] abréviation de *Java Virtual Machine*. Une JVM est donc une *machine virtuelle* pour l'interprétation et l'exé-

grammes spécifiques au matériel et/ou au système d'exploitation concernés) ou avec des bibliothèques logicielles basées sur d'autres langages (comme ici le langage C++). Dans la mesure où nous percevons UML plus comme un langage qu'un modèle de connaissance, KR-suite se « contente » de dialoguer avec ArgoUML par l'intermédiaire de fichiers XMI en version 1.2 [OMG, 2002].

Lors de l'étape 0 (le « point d'entrée ») dans le système de connaissance, illustré en Figure 76, l'importation d'une modélisation existante peut se faire selon les différents formats de fichiers gérés par les trois outils considérés. Il peut s'agir d'un fichier au format XML/OWL pour une importation dans l'outil Pellet, des formats BCGCT, CogXML ou CGIF [3] dans Cogitant et des formats XMI 1.2 ou éventuellement *zargo* [4] pour ArgoUML.

8.2 Structure interne de KR-suite

L'application KR-suite possède son propre *codage* des connaissances, dans la mesure où les connaissances d'un domaine sont modélisées au niveau du système de connaissance lui-même. En effet, le pouvoir expressif du système de connaissance étant supérieur à celui d'un seul modèle du système, toutes les connaissances modélisables dans le système ne peuvent être modélisées par un seul des trois modèles OWL, GCs ou UML ; et donc ne peuvent être codées par un seul des outils Pellet, Cogitant ou ArgoUML.

De façon à faire transiter d'un modèle à un autre les connaissances modélisées (et centralisées) dans notre système de connaissance, KR-suite dispose pour chaque modèle d'une *interface* permettant l'échange de connaissances entre celles codées dans le système et celles dans le modèle.

8.2.1 Codage des connaissances

L'ensemble des notions pris en compte dans KR-suite sont regroupées en *catégories*, dont celle des connaissances structurelles et celle des connaissances factuelles d'une modélisation (les autres catégories sont abordées en Section 8.3). Pour des notions d'une même catégorie, des instances de ces notions - c'est-à-dire des connaissances modélisées - sont regroupées dans une ou plusieurs instances de cette catégorie. Notons que par abus de langage, nous emploierons par la suite le terme « catégorie » pour désigner « une instance de cette catégorie ».

Le codage des connaissances dans KR-suite consite à énumérer et à identifier l'ensemble des connaissances modélisées dans le système de connaissances.

Concrètement, les connaissances sont codées au sein d'un ensemble où un élément

cution de code Java ; l'intérêt d'une machine virtuelle étant l'indépendance du code vis-à-vis du couple machine/système d'exploitation.

[3] voir la documentation de Cogitant pour ces trois formats [Genest, 2007] ; CGIF est normalisé ISO/IEC 24707:2007, `http://www.iso.org`

[4] format spécifique à ArgoUML [Boger *et al.*, 2008]

code une connaissance, c'est-à-dire une instance d'une notion. Chaque élément est caractérisé par un *contexte*, un *identifiant système*, un *prédicat* et un ensemble d'*arguments*.

Le contexte permet d'indiquer à quelle (instance de) catégorie se rattache une connaissance, on parle de *contexte global*, ou à quelle connaissance se rattache une autre connaissance, on parle de *contexte local*. Par exemple, la déclaration d'une classe, c'est-à-dire l'instanciation de la notion de classe, est effectuée dans le contexte global des connaissances structurelles. Autre exemple, une connaissance portant sur une cardinalité minimale est modélisée dans le contexte d'une classe donnée.

Un identifiant système permet de référencer de manière unique au sein de KR-suite une *ressource* modélisée, c'est-à-dire une classe, une relation, un slot, un individu, un lien ou une attribution. Un identifiant système est indépendant - mais peut être identique - d'un nom (on dit aussi référence ou identifiant[5]) attribué à une ressource dans la modélisation. En effet, une classe peut être nommée ex:Personne dans une modélisation, mais être identifiée par l'entier 6743 dans Cogitant et par un autre identifiant dans Pellet. Un identifiant système est donc là pour fournir un identifiant à utiliser au sein des structures de données de Cogitant ou Pellet ou dans un fichier XMI pour ArgoUML, afin que les connaissances modélisées dans les trois outils restent synchronisées. Le nom de la ressource (*cf.* prédicat named) est nécessairement une URI dans KR-suite, de façon à ce qu'il puisse être pris en compte sur une ressource XML/OWL dans Pellet ; les deux autres outils n'ayant pas de contrainte particulière à ce niveau. Si ce n'est pas le cas, KR-suite se charge d'ajouter devant les noms de ressources une « base » (comme http://exemple.fr#) à la manière d'un espace de nommage en XML.

Le prédicat indique la notion instanciée pour modéliser la connaissance de l'élément en question. Par exemple, le prédicat minCardinality indique que la connaissance concerne une restriction de cardinalité minimale (sur la ressource identifiée par l'identifiant système).

Les arguments apportent la caractérisation de la connaissance selon le prédicat concerné, comme par exemple la valeur d'une cardinalité.

Nous présentons dans cette section le codage des connaissances dans KR-suite par catégorie. L'exemple de modélisation en Figure 86 pour la structure et en Figure 91 pour les faits reprend en partie l'exemple illustratif du Chapitre 7.

8.2.1.1 Codage des connaissances structurelles

La Figure 86 présente un exemple de codage des connaissances structurelles d'une modélisation. Ces connaissances sont toutes regroupées dans le contexte particulier SK1 de type la catégorie des connaissances structurelles (symbole STRUCTURAL_K) ; SK1 est lui même dans le contexte de plus haut niveau, noté 0, englobant la modélisation dans son entier au sein KR-suite.

[5]Pour éviter toute ambiguïté entre un identifiant attribué à une ressource au niveau de la modélisation et un identifiant (système) attribué par KR-suite à cette ressource, nous utilisons le terme « nom » (ou « nom de référence ») dans le premier cas et le terme « identifiant système » dans le second cas.

contexte	identifiant système	connaissance (une instanciation de notion)	argument(s)		
0	SK1	*type*	*STRUCTURAL_K*		
SK1	c_anony_1	*type*	*Class*		
SK1	Personne	*type*	*Class*		
SK1	Personne	*named*	**ex:Personne**		
SK1	Personne	*subClassOf*	c_anony_1		
SK1	Pilote	*type*	*Class*		
SK1	Pilote	*named*	**ex:Pilote**		
SK1	Pilote	*subClassOf*	Personne		
SK1	c_anony_2	*type*	*Class*		
SK1	c_anony_2	*subClassOf*	Pilote		
SK1	PiloteAuto	*type*	*Class*		
SK1	PiloteAuto	*named*	**ex:PiloteAuto**		
SK1	PiloteAuto	*equivalentClass*	c_anony_2		
SK1	Véhicule	*type*	*Class*		
SK1	Véhicule	*named*	**ex:Véhicule**		
SK1	Auto	*type*	*Class*		
SK1	Auto	*named*	**ex:Auto**		
SK1	Moto	*type*	*Class*		
SK1	Moto	*named*	**ex:Moto**		
SK1	Véhicule	*unionOf*	Auto	Moto	
SK1	Véhicule	*disjointWith*	Moto		
c_anony_1	nom	*minCardinality*	*1*		
c_anony_1	nom	*maxCardinality*	*1*		
c_anony_2	piloter	*someValuesFrom*	*Auto*		
SK1	nom	*type*	slot		
SK1	nom	*named*	**ex:nom**		
SK1	nom	*signature*	*Thing*	*string*	
SK1	conduire	*type*	*relation*		
SK1	conduire	*named*	**ex:conduire**		
SK1	conduire	*signature*	Personne	Véhicule	
SK1	piloter	*type*	*relation*		
SK1	piloter	*named*	**ex:piloter**		
SK1	piloter	*signature*	Pilote	Véhicule	
SK1	piloter	*subRelationOf*	conduire		

Classes

Slots et Relations

avec « http://exemple.fr# » noté « ex: »

Figure 86 – Codage dans KR-suite des connaissances structurelles d'une modélisation.

contexte	identifiant système	connaissance (une instanciation de notion)	argument(s)
SKx	**Thing**	*type*	*Class*
SKx	Thing	*named*	*http://kr-suite.com#Thing*
SKx	**string**	*type*	*http://kr-suite.com#Datatype*
SKx	string	*named*	*http://www.w3.org/2001/XMLSchema#string*
SKx	**int**	*type*	*http://kr-suite.com#Datatype*
SKx	int	*named*	*http://www.w3.org/2001/XMLSchema#int*
SKx	**boolean**	*type*	*http://kr-suite.com#Datatype*
SKx	boolean	*named*	*http://www.w3.org/2001/XMLSchema#boolean*

Figure 87 – Classe et datatypes prédéfinis et leurs *identifiants systèmes* associés, pour toute catégorie de connaissances structurelles (désignée ici sous SKx).

Prenons la ressource identifiée par l'identifiant système Personne. Cette ressource est caractérisée comme étant de type Class, étant une sous-classe de c_anony_1 et étant nommée ex:Personne (ou http://exemple.fr#Personne, avec ex: un raccourci d'écriture de http://exemple.fr#) Ici, c_anony_1 ne dispose pas de nom d'un point de vue modélisation dans les modèles du système de connaissance (absence d'un élément du codage ayant pour prédicat named et pour identifiant système c_anony_1). Il s'agit donc d'une classe anonyme. Remarquons que la restriction de cardinalité minimale concernant l'emploi du slot ex:nom est dans le contexte de la classe c_anony_1, qui est toujours dans le contexte SK1.

Notons que les mots en italique dans l'ensemble des éléments du codage sont des symboles dans KR-suite, qui correspondent à l'ensemble des notions prises en compte dans KR-suite.

Notons aussi que les identifiants systèmes sont des entiers, mais qu'il sont ici des chaînes de caractères de façon à rendre la lecture des exemples plus facile. Cette table de codage étant relativement intuitive, nous ne la détaillerons pas plus. Cependant, nous présentons ci-après comment ces connaissances codées dans KR-suite se retrouvent modélisées dans les trois modèles du système de connaissance par le biais des transitions de modèles. L'interprétation de cette table de codage pourra ainsi être abordée par le lecteur à partir d'un des trois modèles OWL, GCs ou UML.

Un ensemble de datatypes est prédéfini dans KR-suite. De façon à pouvoir les utiliser dans Pellet, manipulant des document XML/OWL, nous les assimilons par le prédicat named à ceux du Schéma XML (voir la Section 2.2.3).

■ **Transition de modèles dans le modèle UML**

Suivant les transitions de modèles définies aux Chapitres 4 et 5, les connaissances structurelles codées dans KR-suite présentées ci-dessus sont modélisées dans le modèle UML en Figure 88.

On constate notamment, que la classe identifiée Personne apparaît sous le nom ex:Personne dans ce diagramme de classes UML, selon l'information codée par le symbole named. On constate aussi que la classe anonyme sur-classe de Personne est tout à fait distinguable de la classe anonyme sous-classe de Pilote, car ne possédant pas le même identifiant système. Pour cela, dans le format de fichier XMI d'import/export pour coder un élément UML,

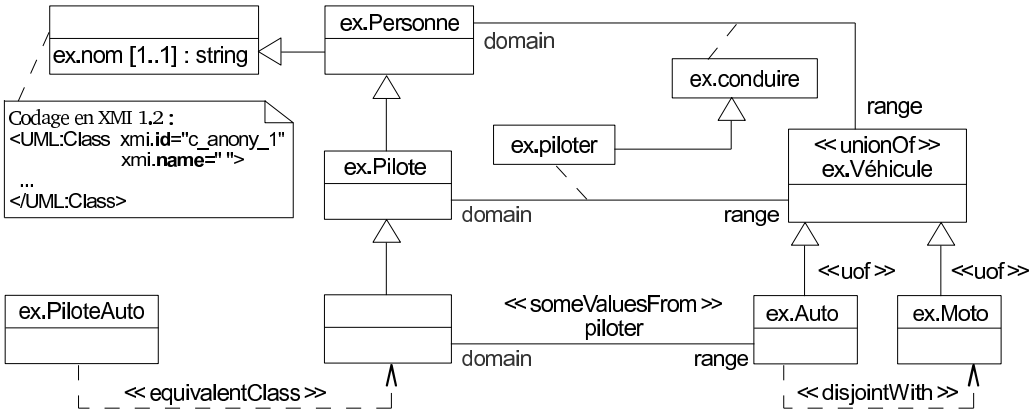


Figure 88 – Modélisation des connaissances structurelles dans le modèle UML par transition de modèles (depuis les connaissances codées en Figure 86).

deux attributs particuliers sont utilisés. Il s'agit de l'attribut xmi.id pour attribuer un identifiant à l'élément et de l'attribut xmi.name fournissant un nom « d'affichage » à l'élément dans l'environnement visuel UML.

Remarquons que ces deux classes anonymes pourraient être issues de la modélisation initiale en Figure 77 (du Chapitre 7) portant sur les classes Personne et PiloteAuto : après intégration dans KR-suite, les classes anonymes sont générées par les règles de transitions de notions sur la notion de restriction de cardinalité portant sur le slot nom et la notion de restriction existentielle de co-domaine agissant sur la classe PiloteAuto par la relation piloter (voir le Chapitre 4.3.2).

Notons que le raccourci d'écriture ex: est remplacé ici par ex. pour ne pas confondre le caractère deux-points venant de ex: avec le séparateur utilisé entre le nom d'un objet et son type dans un diagramme d'objets (comme en Section 8.2.1.2).

■ Transition de modèles dans le modèle OWL

La Figure 89 présente la modélisation de connaissances structurelles dans le modèle OWL, issue des connaissances codées dans KR-suite en Figure 86 et obtenue par transition de modèles dans Pellet.

Notons que ce document ne constitue que l'export en XML des connaissances modélisées dans Pellet. C'est pourquoi seuls les noms (suivant le symbole named en Figure 86) des ressources figurent dans le document ; les identifiants systèmes sont utilisés en interne par Pellet pour identifier les ressources OWL qu'il modélise (le codage interne dans l'outil Pellet n'étant pas visible sur ce document XML/OWL).

■ Transition de modèles dans le modèle des GCs

La Figure 90 présente la modélisation de connaissances structurelles dans le modèle des GCs, issue des connaissances codées dans KR-suite en Figure 86 et obtenue par transition de modèles dans Cogitant.

Les hiérarchies des classes et des dataypes (ceux utilisés ici) sont modélisées par

```
<!-- avec 'http://exemple.fr#',  noté 'ex:' ou '&ex;' si entre guillemets,
     l'espace de nommage par défaut -->

<owl:Class rdf:about = "&ex;Personne">
  <rdfs:subClassOf>
    <owl:Restriction> <!-- identifié par c_anony_1 dans l'outil Pellet -->
      <owl:onProperty rdf:resource="&ex;nom" />
      <owl:minCardinality rdf:datatype="&xsd;int">1</owl:minCardinality>
      <owl:maxCardinality rdf:datatype="&xsd;int">1</owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about = "&ex;Pilote">
  <rdfs:subClassOf rdf:resource="&ex;Personne" />
</owl:Class>
<owl:Class rdf:about = "&ex;PiloteAuto">
  <owl:equivalentClass>
    <owl:Restriction> <!-- identifié par c_anony_1 dans l'outil Pellet -->
      <owl:onProperty rdf:resource="&ex;piloter" />
      <owl:someValuesFrom rdf:resource="&ex;Auto" />
    </owl:Restriction>
  </owl:equivalentClass>
  <rdfs:subClassOf rdf:resource="&ex;Pilote" />
</owl:Class>
<owl:Class rdf:about = "&ex;Véhicule">
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="&ex;Auto" />
    <owl:Class rdf:about="&ex;Moto" />
  </owl:unionOf>
</owl:Class>
<owl:Class rdf:about = "&ex;Auto">
  <rdfs:subClassOf rdf:resource="&ex;Véhicule" />
  <owl:disjointWith rdf:resource="&ex;Moto"/>
</owl:Class>
<owl:Class rdf:about = "&ex;Moto">
  <rdfs:subClassOf rdf:resource="&ex;Véhicule" />
</owl:Class>

<owl:DatatypeProperty rdf:about = "&ex;nom">
  <rdfs:domain rdf:resource="&owl;Thing" />
  <rdfs:range rdf:resource="&xsd;string" />
</owl:DatatypeProperty>
<owl:ObjectProperty rdf:about = "&ex;conduire">
  <rdfs:domain rdf:resource="&ex;Personne" />
  <rdfs:range  rdf:resource="&ex;Véhicule" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about = "&ex;piloter">
  <rdfs:domain rdf:resource="&ex;Pilote" />
  <rdfs:range  rdf:resource="&ex;Auto" />
</owl:ObjectProperty>
```

Figure 89 – Modélisation des connaissances structurelles dans le modèle OWL par transition de modèles (depuis les connaissances codées en Figure 86).

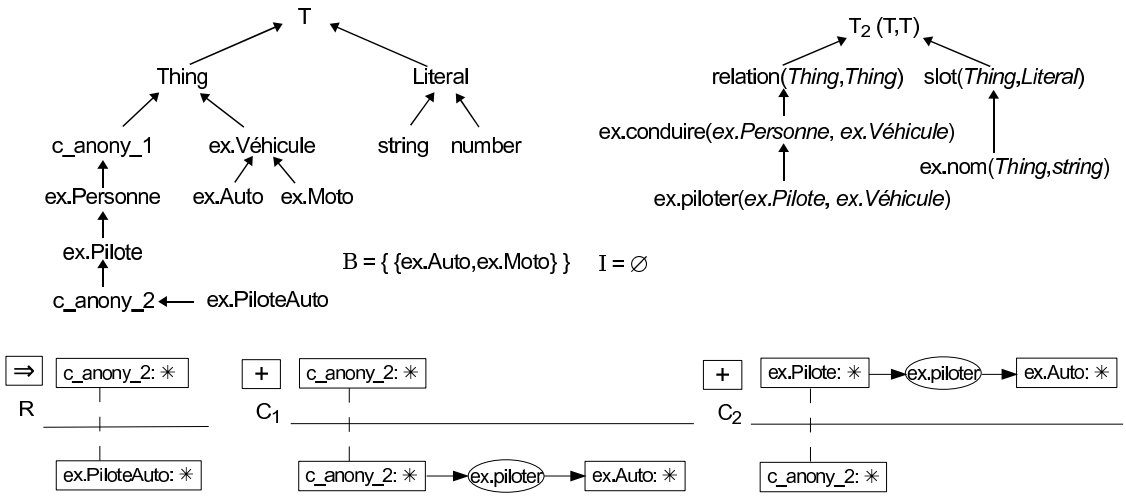


Figure 90 – Modélisation des connaissances structurelles dans le modèle des GCs par transition de modèles (depuis les connaissances codées en Figure 86).

l'ensemble des types de concepts. Les hiérarchies des slots et des relations sont modélisées par l'ensemble des types de relations binaires. Le type de concept conjonctif banni {Auto,Moto} est issu de l'application de la règle d'instanciation de la notion de disjonction entre les classes Auto et Moto.

Concernant l'équivalence qui existe entre la classe anonyme identifié par c_anony_2 et la classe ex:PiloteAuto, l'utilisateur à la possibilité dans KR-suite de configurer le système de connaissance pour que la notion d'équivalence soit interpréter soit comme une connaissance à déduire soit comme une connaissance à vérifier (voir le Chapitre4). Dans cet exemple nous avons supposé le premier choix, c'est-à-dire que cette équivalence permettra à Cogitant de déduire que toute instance de c_anony_2 sera interprétée aussi comme une instance de ex:PiloteAuto et inversement. Par ce choix et selon la règle d'instanciation de ladite notion, ex:PiloteAuto est modélisé comme sous-type de c_anony_2 et la règle R vient compléter cette modélisation d'équivalence.

À propos de la restriction existentielle de co-domaine agissant sur la classe c_anony_2 et portant sur la relation piloter, nous avons considéré cette fois-ci que la configuration de KR-suite vise à vérifier cette connaissance modélisée. C'est-à-dire que cette restriction devra toujours être vérifiée dans les faits. Les contraintes positives C_1 et C_2 modélise cette contrainte, conformément à la règle d'instanciation de cette notion et du choix effectué.

Notons que le raccourci d'écriture ex: est remplacé ici par ex. pour ne pas confondre le caractère deux-points dans ex: avec le séparateur utilisé visuellement entre un type et un marqueur (individuel ou générique) dans l'étiquetage d'un sommet concept d'un graphe conceptuel.

Notons aussi que le support en Figure 90 n'est qu'une représentation visuelle du support codé dans Cogitant. Pour cette raison, seuls les noms des ressources sont représentées, pas leurs identifiants.

contexte	identifiant système	connaissance (une instanciation de notion)	argument(s)		
0	FK1	*type*	*FACTUAL_K*		
0	FK1	*defined-on*	*SK1*		
FK1	alphand	*type*	PiloteAuto		
FK1	alphand	*named*	**ex:alphand**		
FK1	alphand	*caract*	*UNA*		
FK1	luc	*type*	Personne		
FK1	luc	*named*	**ex:luc**		
FK1	luc	*caract*	*UNA*		individus
FK1	luc	*sameAs*	alphand		et données
FK1	i_anony_1	*type*	Auto		
FK1	donnee1	*type*	*string*		
FK1	donnee1	*value*	"L. alphand"		
FK1	donnee2	*type*	*string*		
FK1	donnee2	*value*	"Luc Alphand"		
FK1	attribution1	*type*	nom		
FK1	attribution1	*neighbours*	alphand	donnee1	
FK1	attribution2	*type*	nom		attributions
FK1	attribution2	*neighbours*	luc	donnee2	et liens
FK1	lien1	*type*	conduire		
FK1	lien1	*neighbours*	alphand	i_anony_1	

Figure 91 – Codage dans KR-suite des connaissances factuelles d'une modélisation.

8.2.1.2 Codage des connaissances factuelles

La Figure 91 présente un exemple de codage des connaissances factuelles d'une modélisation. Ces connaissances sont toutes regroupées dans le contexte FK1 de type (le symbole) FACTUAL_K ; FK1 est lui même dans le contexte de plus haut niveau, noté 0.

Prenons l'individu alphand. Il est référencé, par le prédicat named, sous le nom de référence ex:alphand (http://exemple.fr#alphand) au sein de la modélisation. Cet individu est assujetti à une caractéristique de nom unique (caract. UNA). C'est-à-dire que cet individu sera par défaut considéré comme différent de tous les autres individus de la modélisation, sauf indication contraire explicite comme avec l'individu luc. L'individu alphand est lié par une assertion de la relation nom (l'attribution d'identifiant système attribution1) à la donnée (d'identifiant système donnee1) de type string et de valeur "L. Alphand".

L'individu de type Auto est anonyme, car il n'est associé à aucun nom de référence au niveau de la modélisation.

Un individu non contraint à l'hypothèse du non unique, dont le cas particulier d'un individu anonyme, est considéré dans KR-suite comme une variable (voir la Section 6.1.2).

Notons qu'un individu expressement nommé, mais non soumis à l'hypothèse du nom unique, est dans l'absolu comparable à un individu anonyme puisqu'il pourra être - suivant la modélisation - être interprété comme identique ou différent de d'autres individus. Cependant, ce nom peut être porteur d'une signification. C'est pourquoi ce nom est codé dans KR-suite, et est conservé lors de la transition de modèles entre les différents outils

■ **Transition de modèles dans le modèle UML**

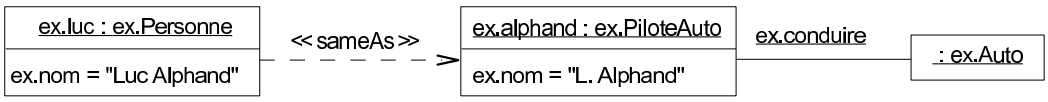


Figure 92 – Modélisation des connaissances factuelles dans le modèle UML par transition de modèles (depuis les connaissances codées en Figure 91).

Suivant les transitions de modèles définies au Chapitre 6, les connaissances factuelles codées dans KR-suite présentées ci-dessus sont modélisées dans le modèle UML en Figure 92.

La modélisation de connaissances dans le modèle UML étant sous l'hypothèse du nom unique (UNA), il n'est pas nécessaire de préciser que l'individu alphand et l'individu anonyme de type Auto sont différents. Par contre, le lien de dépendance stéréotypé par ≪sameAs≫ permet d'indiquer que les individus luc et alphand sont identiques.

■ **Transition de modèles dans le modèle OWL**

La Figure 93 présente la modélisation de connaissances factuelles dans le modèle OWL, issue des connaissances codées dans KR-suite en Figure 91 et obtenue par transition de modèles dans Pellet.

```
<!-- avec 'http://exemple.fr#',  note 'ex:' ou '&ex;' si entre guillemets,
     l'espace de nommage par défaut -->

<ex:PiloteAuto rdf:about = "&ex;alphand">
  <ex:nom rdf:datatype="&xsd;string">L. Alphand</ex:nom>
  <ex:conduire rdf:resource="http://temp.com#i_anony_1"/>
</ex:PiloteAuto>
<Personne rdf:about = "&ex;luc">
  <ex:nom rdf:datatype="&xsd;string">Luc Alphand</ex:nom>
  <owl:sameAs rdf:resource="&ex;alphand" />
</Personne>
<ex:Auto rdf:about = "http://temp.com#i_anony_1">
  <!--individu identifié par i_anony_1 dans l'outil Pellet -->
    <rdfs:label lang="en">unnamed</rdfs:label>
    <!--référence XML temporaire, sans signification de modélisation -->
</ex:Auto>
```

Figure 93 – Modélisation des connaissances factuelles dans le modèle OWL par transition de modèles (depuis les connaissances codées en Figure 91).

La modélisation de connaissances dans le modèle OWL n'évolue pas sous l'hypothèse du nom unique. Ainsi, il est explicitement précisé que alphand et luc sont identiques.

■ **Transition de modèles dans le modèle des GCs**

La Figure 94 présente la modélisation de connaissances factuelles dans le modèle des GCs, issue des connaissances codées dans KR-suite en Figure 91 et obtenue par transition de modèles dans Cogitant.

La modélisation de connaissances dans le modèle des GCs étant sous l'hypothèse du nom unique, il n'est pas nécessaire de préciser que l'individu alphand et l'individu anonyme de type Auto sont différents. La synonymie existante entre les identifiants luc et

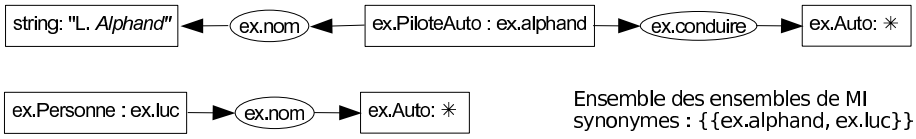


Figure 94 – Modélisation des connaissances factuelles dans le modèle des GCs par transition de modèles (depuis les connaissances codées en Figure 91).

alphand est quant à elle prise en charge à un niveau implémentation de KR-suite visant à gérer une liste d'ensembles de marqueurs individuels synonymes.

8.2.2 Interfaces modèles

L'application KR-suite dispose de trois interfaces modèles, à savoir une interface KR-suite−Pellet, une interface KR-suite−Cogitant et une interface KR-suite−XMI (Figure 95). Une *interface modèle* a pour but de faire transiter les connaissances codées dans KR-suite

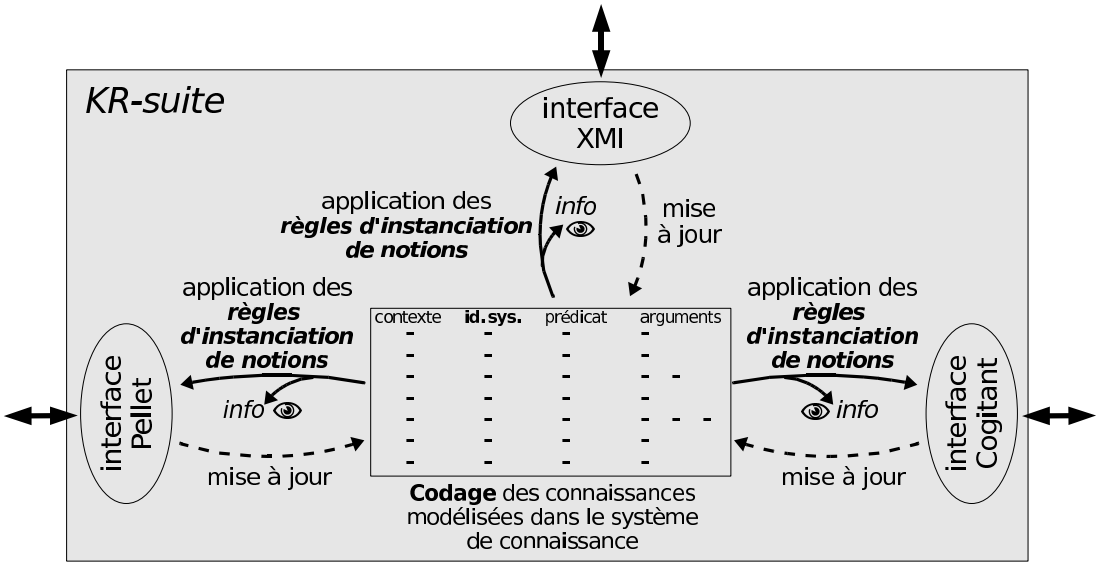


Figure 95 – Fonctionnement interne de KR-suite.

vers un des outils Pellet, Cogitant ou ArgoUML, et inversement. C'est-à-dire d'implémenter les règles d'instanciation de notions définies aux Chapitres 4, 5 et 6.

Les deux interfaces KR-suite−Pellet et KR-suite−Cogitant sont développées de manière à passer d'un codage de connaissances dans KR-suite directement à un codage de connaissances propre à respectivement Pellet et Cogitant. En effet, KR-suite étant codé en C++, il nous a été possible d'utiliser les structures de données C++ de la bibliothèque Cogitant pour implémenter les transitions de modèles entre KR-suite et Cogitant. Dans le même esprit, nous avons pu exploiter les structures de données Java de la bibliothèque Pellet à l'aide de l'outil JNI (*Java Native Interface*), faisant le lien entre du code Java et C++.

L'interface KR-suite−XMI se « contente » de faire transiter le codage des connaissances dans KR-suite vers un codage des connaissances en XMI version 1.2. Rappelons que le modèle UML, à travers l'éditeur ArgoUML, est utilisé pour ses qualités visuelles et non l'inférence. L'aspect rapidité d'échange de connaissances entre ArgoUML et KR-suite est relégué en second plan.

Lors de la transition de connaissances de KR-suite vers un des outils du système de connaissance, selon l'outil - basé sur un modèle - certaines connaissances peuvent ne pas être représentables dans cet outil. Dans ce cas, un message en informe l'utilisateur de façon à ce que ce dernier puisse en tenir compte lors de raisonnements produits par les raisonneurs (Pellet et Cogitant).

Lors de la transition de connaissances d'un des outils du système vers KR-suite, l'utilisateur est amené à superviser la transition de connaissances d'un outil vers KR-suite, notamment en consultant les messages mentionnés ci-avant.

8.3 Interrogation, déduction et vérification de connaissances

Les requêtes, les règles et les contraintes, ainsi que les réponses à une requête, les applications d'une règle et les vérifications d'une contrainte sont autant de catégories de connaissances dans KR-suite.

8.3.1 Interrogation

Les interrogations, ou requêtes, qui peuvent être posées pour questionner les connaissances factuelles d'une modélisation constituent une catégorie de connaissances codée dans KR-suite sous le symbole QUERY.

Le codage d'une interrogation se fait suivant le même principe que pour des connaissances factuelles, puisqu'une requête consite à modéliser des faits qui seront recherchés dans la modélisation ; une réponse étant un sous-ensemble des connaissances factuelles de la modélisation interrogée ou l'ensemble vide. Au niveau de la valeur d'une donnée, une information optionnelle peut être renseignée, à savoir l'opérateur avec lequel devra s'effectuer la correspondance entre une ressource de la requête avec une ressource dans les faits interrogés. L'opérateur par défaut est la stricte correspondance. Sur cet exemple, la chaîne de caractères "Alphand" à chercher peut être une sous-chaîne (*cf.* l'argument like) de la chaîne de caractères dans les connaissances interrogées.

Supposons le scénario suivant. L'utilisateur définit la requête dans le modèle UML qu'il connaît bien. Cette requête, présentée en haut de la Figure 96(a), est la suivante : « Existe-t-il un individu de type Pilote qui conduit un Véhicule et qui a pour nom (ou partie de nom) "Alphand" ? ». Une fois cette requête éditée dans ArgoUML, elle est codée dans KR-suite telle que présentée en Figure 96(b) dans le contexte Q1.

(a) Interrogation dans le modèle UML .

contexte	identifiant système	connaissance (une instanciation de notion)	argument(s)	
0	Q1	*type*	*QUERY*	
0	Q1	*defined-on*	*SK1*	
Q1	individu1	*type*	Pilote	
Q1	individu2	*type*	Véhicule	
Q1	donnee3	*type*	*string*	
Q1	donnee3	*value*	"Alphand"	{like}
Q1	attribution3	*type*	nom	
Q1	attribution3	*neighbours*	individu1	donnee3
Q1	lien2	*type*	conduire	
Q1	lien2	*neighbours*	individu1	individu2
0	SA1	*type*	*SEARCHING_ANSWER*	
0	SA1	*from*	Q1	
0	SA1	*applied-on*	FK1	
0	A1	*type*	*ANSWER_TRUE*	
0	A1	*on*	SA1	
A1	individu1	*image*	alphand	
A1	individu2	*image*	i_anony_1	
A1	donnee3	*image*	donnee1	
A1	attribution3	*image*	attribution1	
A1	lien2	*image*	lien1	
0	A2	*type*	*ANSWER_TRUE*	
0	A2	*on*	SA1	
A2	individu1	*image*	luc	
A2	individu2	*image*	i_anony_1	
A2	donnee3	*image*	donnee2	
A2	attribution3	*image*	attribution2	
A2	lien2	*image*	lien1	

(b) Codage dans KR-suite de l'interrogation et de ses deux réponses.

(c) Interrogation dans le modèle des GCs par un graphe conceptuel requête, issu de la transition de modèles de UML vers GCs.

Figure 96 – Interrogation des connaissances en Figure 91.

Le choix de l'utilisateur d'interroger les connaissances factuelles en Figure 91 par cette requête est codé et identifié par SA1 dans KR-suite (milieu de la Figure 96(b)) ; cette demande d'interrogation fait partie de la catégorie ANSWERING_QUERY.

Pour effectuer la recherche, KR-suite fait *transiter* la requête du modèle UML vers le modèle des GCs par les interfaces modèles ArgoUML−KR-suite puis KR-suite−Cogitant.

Cette requête maintenant représentée dans le modèle des GCs (en Figure 96(c)) peut être traitée par Cogitant. Les réponses à une requête sont dans la catégorie de connaissances ANSWER_TRUE s'il y a une réponse, sinon dans la catégorie ANSWER_FALSE. Sur cet exemple, la requête admet deux réponses, l'une identifiée par A1 et l'autre identifiée par A2. Les correspondances entre les ressources d'une requête et les ressources dans les faits interrogés sont codées à l'aide du prédicat image. De ce codage centralisé dans KR-suite, la réponse peut être visualisée dans UML.

8.3.2 Déduction

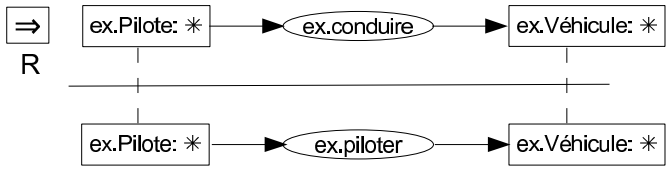
Les règles font partie de la catégorie de connaissances codée dans KR-suite sous le symbole RULE. Le codage de l'hypothèse d'une règle (prédicat hypothesis) se fait comme pour une requête, et le codage de la conclusion d'une règle prédicat conclusion) se fait comme pour des connaissances factuelles.

L'exemple de règle R en Figure 97(a), définie dans le modèle des GCs, stipule que « si un pilote conduit un véhicule, alors il pilote ce même véhicule ». Cette règle est identifiée dans KR-suite par R1 en Figure 97(b).

Une demande d'application de cette règle sur les connaissances factuelles en Figure 91 fait partie de la catégorie APPLYING_RULE. Les applications d'une règle sont dans la catégorie de connaissances RULE_APPLICATION_DONE s'il elles existent, sinon dans la catégorie RULE_APPLICATION_NO. Sur cet exemple, la règle admet une application, identifiée par RAP1. La correspondance entre une ressource de l'hypothèse de la règle et une ressource dans les faits est codée à l'aide du prédicat image. La correspondance entre une ressource de la conclusion de la règle et une ressource dans les faits est codée à l'aide du prédicat image-add, sorte-de image, indiquant que si une connaissance est nouvelle alors elle est ajoutée aux faits.

8.3.3 Vérification

Issues du modèle des GCs, et en complément de contraintes liées aux connaissances structurelles et factuelles d'une modélisation, deux sortes de contraintes peuvent être définies. Les contraintes positives permettent de vérifier l'existence de connaissances sous condition parmi les connaissances factuelles, elles sont dans la catégorie de connaissances codée dans KR-suite sous le symbole POSITIVE_CONSTRAINT. Les contraintes négatives permettent d'interdire l'existence de connaissances, elles sont dans la catégorie de connaissances codée dans KR-suite sous le symbole NEGATIVE_CONSTRAINT. Le codage de la condition d'une contrainte (prédicat condition), d'une obligation (prédicat obligation) ou d'une interdiction (prédicat prohibition) se font comme pour des connaissances factuelles.



(a) Règle définie dans le modèle des GCs.

contexte	identifiant système	connaissance (une instanciation de notion)	argument(s)	
0	R1	*type*	*RULE*	
0	R1	*defined-on*	*SK1*	
0	R1	*hypothesis*	*R1H*	
0	R1	*conclusion*	*R1C*	
R1H	individu3	*type*	Pilote	
R1H	individu4	*type*	Véhicule	
R1H	lien3	*type*	conduire	
R1H	lien3	*neighbours*	individu3	individu4
R1C	lien4	*type*	piloter	
R1C	lien4	*neighbours*	individu3	individu4
0	AR1	*type*	*APPLYING_RULE*	
0	AR1	*from*	R1	
0	AR1	*applied-on*	FK1	
0	RAP1	*type*	*RULE_APPLICATION_DONE*	
0	RAP1	*on*	AR1	
RAP1	individu3	*image*	alphand	
RAP1	individu4	*image*	i_anony_1	
RAP1	lien3	*image*	lien1	
RAP1	lien5	*type*	piloter	
RAP1	lien5	*neighbours*	alphand	i_anony_1
RAP1	lien5	***image-add***	lien4	

(b) Codage dans KR-suite de la règle et de son application.

Figure 97 – Règle d'inférence appliquée sur les connaissances en Figure 91 pour la déduction de connaissances.

contexte	identifiant système	connaissance (une instanciation de notion)	argument(s)
0	PC1	*type*	*P_CONSTRAINTE*
0	PC1	*defined-on*	*SK1*
0	PC1	*condition*	*PC1C*
0	PC1	*obligation*	*PC1O*
PC1C	individu5	*type*	c_anony_2 (équivalent à PiloteAuto)
PC1O	individu6	*type*	Auto
PC1O	lien6	*type*	piloter
PC1O	lien6	*neighbours*	individu5 individu6
0	CC1	*type*	*CHECKING_CONSTRAINT*
0	CC1	*from*	PC1
0	CC1	*applied-on*	FK1
0	PCCF1	*type*	*P_CONSTRAINT_CHECK_FAILED*
0	PCCF1	*on*	CC1
PCCF1	individu5	*image-condition*	alphand
0	PCCP1	*type*	*P_CONSTRAINT_CHECK_PASS*
0	PCCP1	*on*	CC1
PCCP1	individu5	*image-condition*	alphand
PCCP1	individu6	*image-obligation*	i_anony_1
PCCP1	lien6	*image-obligation*	lien5

si vérification **avant** l'application de la règle en Figure précédente

si vérification **après** l'application de la règle en Figure précédente (avec l'ajout du lien5)

Figure 98 – Codage dans KR-suite de la contrainte positive C_1 en Figure 90 et de ses vérifications.

Reprenons pour exemple la contrainte C_1 en Figure 90, qui oblige que « tout individu de type i_anony_2 (type équivalent à PiloteAuto) doit pilote une auto ». Cette contrainte positive est identifiée dans KR-suite par PC1 en Figure 98.

L'action de vérifier une contrainte positive (positive ou négative) sur des connaissances factuelles d'une modélisation fait partie de la catégorie CHECKING_CONSTRAINT. Supposons que le règle ci-dessus n'ait pas été appliquée sur les faits en Figure 91. Il en résulte que la contrainte positive de cet exemple n'est pas vérifiée ; cette connaissance de non succès est identifiée par PCCF1, dans la catégorie P_CONSTRAINT_CHECK_FAILED. En effet, il y a correspondance (*cf.* prédicat image-condition) de la condition de la contrainte sur les faits sans qu'il y ait de correspondance (*cf.* prédicat image-obligation) de l'obligation de la contrainte sur les faits. Par contre, si la règle précédente à bien été appliquée, il en résulte que cette contrainte positive est bien vérifiée (vérification identifiée par PCCP1 dans la catégorie P_CONSTRAINT_CHECK_PASS).

Dans le cas d'une contrainte négative, elle sera vérifiée (catégorie N_CONSTRAINT_CHECK_PASS) si lorsque la condition étant vraie il n'y a pas de correspondance de l'interdiction (*cf.* prédicat image-prohibition) de la contrainte sur les faits. Inversement, cette contrainte négative ne sera pas vérifiée (catégorie N_CONSTRAINT_CHECK_FAILED) si lorsque la condition étant vraie il y a pas correspondance de l'interdiction de la contrainte sur les faits.

8.4 Conclusion

KR-suite est une application logicielle qui permet d'exploiter la complémentarité représentationnelle et inférencielle des modèles de connaissances OWL, GCs et UML par une utilisation conjointe des outils Pellet, Cogitant et ArgoUML.

Afin de pouvoir modéliser des connaissances dans KR-suite et de les faire transiter d'un outil à un autre, les deux principaux points dans l'implémentation de notre système de connaissance KR-suite ont été les suivants. D'une part, il a fallu élaborer un système de codage des connaissances propre à notre application. En effet, les différents outils logiciels ne peuvent, pris séparément, représenter l'ensemble des connaissances modélisables dans KR-suite. En plus de lister l'ensemble des connaissances modélisées, KR-suite attribue à chaque ressource un identifiant unique de façon à ce qu'au gré des transitions de connaissances d'un outil à un autre les modélisations au sein des trois outils restent synchronisées. D'autre part, la mise en relation des différents outils avec KR-suite a nécessité la mise en place d'interfaces pour implémenter les règles de transitions de modèles propres à chaque modèle du système de connaissance et pour palier des différences technologiques entre les outils.

Conclusion générale

Le travail de thèse présenté ici se place dans le cadre de la représentation des connaissances. Il concerne plus particulièrement l'élaboration d'un système de connaissance fondé sur les modèles OWL, GCs et UML visant à exploiter la complémentarité tant de modélisation que de raisonnement de ces trois modèles.

Le cœur de ce travail a consisté d'une part à la définition de notions pour modéliser des connaissances, structurelles et factuelles, d'un domaine. D'autre part, des règles d'instanciation de notions ont été établies afin de spécifier comment chaque notion est instanciable dans chacun des modèles du système de connaissance pour y représenter un type de connaissance spécifique.

L'originalité de cette thèse réside en la modélisation centralisée de connaissances au sein du système de connaissance et à faire transiter ces connaissances d'un modèle à un autre du système au fur et à mesure des besoins exigés par la modélisation et les raisonnements. L'intérêt de cette approche est double. Premièrement, les connaissances d'un domaine sont modélisées suivant une grande expressivité de représentation tout en rendant le système de connaissance en lui-même transparent à l'utilisateur. En effet, l'expressivité du système de connaissance repose sur les expressivités conjuguées des modèles qui le composent, de sorte que l'utilisateur est amené à utiliser de manière complémentaire ces modèles en fonction de leurs différents pouvoirs expressifs et/ou éventuellement en fonction des préférences de l'utilisateur pour certains modèles connus. Deuxièmement, le système de connaissance dispose de capacités de raisonnements, sur les connaissances modélisées, qui surclassent celles qui pourraient être obtenus dans chaque modèle. Pour cela, le système de connaissance par le biais des règles d'instanciation de notions sur les connaissances modélisées dans le système offrent la possibilité de faire transiter d'un modèle à un autre les connaissances. Ainsi, les différents types de raisonnements propres à chaque modèle sont exploitables, et une synergie entre ces derniers est réalisable.

L'harmonisation des raisonnements produits entre les différents modèles nécessite d'être éventuellement orchestrée par l'utilisateur. En effet, certaines connaissances modélisées dans le système ne sont pas représentables dans un modèle donné - du fait de la complémentarité représentationnelle entre les modèles - et donc non prises en compte lors de raisonnements dans ce modèle. L'aspect anthropocentré du système de connaissance n'en constitue cependant pas une limite dans la mesure où il est conçu comme une aide pour l'utilisateur qui apporte son expérience, ses compétences et exigences à la modélisation de connaissances et aux raisonnements effectués.

Une implémentation du système de connaissance a été réalisée. Nommée KR-suite, elle repose sur les outils logiciels Pellet pour le modèle OWL, Cogitant pour le modèle des GCs et ArgoUML. Les principales difficultés ont été de faire travailler de concert des outils aux technologies différentes, et de concevoir un codage des connaissances modélisées au sein du système de façon à ce qu'au cours des transitions de modèles ces connaissances restent synchronisées entre les représentations effectives dans les différents outils.

Plusieurs limites de notre travail amènent à des perspectives de recherches nouvelles. Une première amélioration abordable sur le court terme peut concerner l'arité des relations. Actuellement binaire, elle pourrait être généralisée à des relations n-aires pour n un nombre entier supérieur à 1. Si cette extension est immédiate avec le modèle des GCs et le modèle UML, elle demanderait une certaine adaptation du modèle OWL. Deuxièmement, l'évolution des connaissances modélisées dans le système de connaissances s'effectue actuellement dans un environnement monotome. En effet, qu'il s'agisse d'une mise à jour manuelle ou découlant de raisonnements, seul l'ajout de nouvelles connaissances est réalisable dans le système mais ni la suppression ni la modification des connaissances déjà existantes le sont. La prise en compte d'un environnement non-monotome pour le système de connaissance est envisagable, où l'adéquation des raisonnements pourrait être dans un premier temps supervisé. Troisièmement, le système de connaissance permet, dans une certaine mesure la prise en compte de connaissances incomplètes (comme l'existence d'un individu sans pour autant être capable de le désigner). Cet aspect pourrait évoluer vers une prise en compte plus large, notamment au niveau des données qui pourrait être valuée par un ensemble de littéraux (comme l'âge d'un individu qui serait supérieur à dix-huit, ou son nom que contiendrait la chaîne de caractères "Dupon").

Pour finir, notons bien que notre approche - basée sur des règles d'instanciation pour un ensemble de notions prédéfinies dans les modèles considérés - permet d'être ouvert à la prise en compte de nouveaux modèles dans le système de connaissance. Ce coté « plug-in sémantique » confère d'une part une évolution simple du système de connaissance pour la prise en compte d'expressivité ou de raisonnement nouveaux, et d'autre part une pérennité au système de connaissance au gré des révisions des différents modèles soutenus par leurs communautés respectives.

Liste des figures

Liste des tableaux

Références bibliographiques

[Akehurst et Bordbar, 2001] cité p. 103
D. H. Akehurst and Behzad Bordbar. On Querying UML Data Models with OCL. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 91–103, London, UK, 2001. Springer-Verlag.

[Aristote, 350 av JC] cité p. 56
Aristote. Catégories, 350 av. JC. Traduction partielle disponible sur Wikisource `http://fr.wikisource.org/wiki/Aristote`.

[Baader *et al.*, 2003] cité p. 3, 38, 70, 76, 79
F. Baader, D. Calvanese, D. L. McGuiness, D. Nardi, and P. F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, Applications*. Cambridge University Press, UK, 2003.

[Baader et Hollunder, 1991] cité p. 83
F. Baader and B. Hollunder. KRIS: Knowledge Representation and Inference System. *SIGART Bulletin*, 2(3) :8–15, 1991.

[Baader et Sattler, 2001] cité p. 83
F. Baader and U. Sattler. An Overview of Tableau Algorithms for Description Logics. *Studia Logica*, 69 :5–40, 2001.

[Baget et Mugnier, 2002] cité p. 7, 18, 19, 20, 21
J. F. Baget and M. L. Mugnier. Extensions of Simple Conceptual Graphs: the Complexity of Rules and Constraints. *Journal of Artificial Intelligence Research (JAIR)*, 16(12) :425–465, 2002.

[Baget, 2007] cité p. 25, 25, 25
Jean-François Baget. A Datatype Extension for Simple Conceptual Graphs and Conceptual Graphs Rules. In Uta Priss, Simon Polovina, and Richard Hill, editors, *Conceptual Structures : Knowledge Architectures for Smart Applications, 15th International Conference on Conceptual Structures (ICCS 2007), Sheffield, UK*, volume 4604 of *Lecture Notes in Computer Science (LNCS)*, pages 83–96. Springer, 2007.

[Berardi *et al.*, 2005] cité p. 100, 161
Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1) :70–118, 2005.

[Berners-Lee *et al.*, 2001] cité p. 39, 83
Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5) :34–43, 2001.

[Berners-Lee, 2000] cité p. 85, 164
Tim Berners-Lee. CWM - Closed World Machine, 2000. `http://www.w3.org/2000/10/swap/doc/cwm.html`.

[Boger *et al.*, 2008] cité p. 102, 171, 172
Marko Boger, Jason Robbins, Linus Tolke, and Markus Klink. ArgoUML Website, 2008. `http://argouml.tigris.org/`.

[Booch *et al.*, 1998] cité p. 3, 100
G. Booch, C. Jacobson, and J. Rumbaugh. *The Unified Modeling Language - a reference manual*. Addison Wesley, 1998.

[Borgida, 1996] cité p. 83
Alex Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82(1-2) :353–367, 1996.

[Brachman *et al.*, 1985] cité p. 5
Ronald J. Brachman, Victoria Pigman Gilbert, and Hector J. Levesque. An Essential Hybrid Reasoning System : Knowledge and Symbol Level Accounts of KRYPTON. In *Proceedings of the ninth International Joint Conferences on Artificial Intelligence IJCAI'85*, pages 532–539, 1985.

[Brachman et Levesque, 1984] cité p. 79
R. J. Brachman and H. J. Levesque. The Tractability of Subsumption in Frame-Based Description Languages. In *Proceedings of AAAI'84*, pages 34–37, 1984.

[Brachman et Schmolze, 1985] cité p. 82
R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation syste. *Cognitive Science*, 9(2) :171–216, 1985.

[Bray *et al.*, 2006] cité p. 40, 41
Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible Markup Language (XML) 1.1 (Second Edition). Technical report, 2006. W3C Recommendation `http://www.w3.org/TR/xml11/`.

[Brickley et Guha, 2004] cité p. 38, 50, 70
D. Brickley and R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. Technical report, 2004. W3C Recommendation `http://www.w3.org/TR/rdf-schema/`.

[Brockmans *et al.*, 2004] cité p. 119
Sara Brockmans, Raphael Volz, Andreas Eberhart, and Peter Löffler. Visual Modeling of OWL DL Ontologies Using UML. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *Proceedings of International Semantic Web Conference (ISWC'04)*, volume 3298 of *Lecture Notes in Computer Science (LNCS)*, pages 198–213. Springer, 2004.

[Brucker et Wolff, 2002] cité p. 103
Achim D. Brucker and Burkhart Wolff. A Proposal for a Formal OCL Semantics in Isabelle/HOL. In *Proceedings of the Theorem Proving in Higher Order Logics*, volume 2410 of *Lecture Notes in Computer Science (LNCS)*, pages 99–114. Springer-Verlag, 2002.

[Bräuer et Demuth, 2008] cité p. 102
Mathias Bräuer and Birgit Demuth. Model-Level Integration of the OCL Standard Library Using a Pivot Model with Generics Support. In *Proceedings of the Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science (LNCS)*, pages 182–193. Springer Berlin, 2008.

[Buchheit *et al.*, 1993] cité p. 74
Martin Buchheit, Francesco M. Donini, and Andrea Schaerf. Decidable Reasoning in Terminological Knowledge Representation Systems. *Artificial Intelligence Research*, 1 :109–138, 1993.

[Catarci et Lenzerini, 1993] cité p. 5
T. Catarci and M. Lenzerini. Representing and Using Interschema Knowledge in Cooperative Information Systems. *Journal for Intelligent and Cooperative Information Systems*, 2(4) :375–399, 1993.

[Charniak et McDermott, 1985] cité p. 1, 141
Eugene Charniak and Drew McDermott. *Introduction to artificial intelligence*. Addison-Wesley Longman Publishing Co., Inc., 1985.

[Chein *et al.*, 1998] cité p. 150
M. Chein, M.L. Mugnier, and G. Simonet. Nested graphs : A graph-based knowledge representation model with FOL semantics. In *Proceedings of Knowledge Representation and Reasoning (KR'98)*, pages 524–534. Morgan Kaufmann Publishers, 1998.

[Chein et Genest, 2000] cité p. 33
Michel Chein and David Genest. CGs Applications : Where Are We 7 Years After the First ICCS. In Ganter and Mineau [2000], pages 127–139.

[Chein et Mugnier, 1992] cité p. 5, 7, 10, 12, 12, 13, 17
M. Chein and M. L. Mugnier. Conceptual Graphs: Fundamental Notions. *Revue d'Intelligence Artificielle (RIA)*, 6(4) :365–406, 1992.

[Chein et Mugnier, 1997] cité p. 33, 150
M. Chein and M. L. Mugnier. Positive Nested Conceptual graphs. In Dickson Lukose, Harry Delugach, Mary Keeler, Leroy Searle, and John F. Sowa, editors, *Conceptual Structures : Fulfilling Peirce's Dream, proceedings of the fifth International Conference on Conceptual Structures (ICCS '97), Seattle, USA*, volume 1257 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 95–109. Springer, 1997.

[Chein et Mugnier, 2004] cité p. 7, 16, 22, 25, 27, 35
M. Chein and M. L. Mugnier. Concept types and coreference in simple conceptual graphs. In Karl Erich Wolff, Heather D. Pfeiffer, and Harry S. Delugach, editors, *Conceptual Structures at Work : proceedings of 12th International Conference on Conceptual Structures (ICCS'2004), Huntsville, AL, USA*, volume 3127 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 303–318. Springer, 2004.

[Cook, 1972] cité p. 80
S. A. Cook. A hierarchy for nondeterministic time complexity. In *the fourth annual ACM symposium on Theory of computing*, pages 187–192, New York, USA, 1972. ACM Press.

[Corby *et al.*, 2000] cité p. 5
O. Corby, R. Dieng, and E. Hébert. A Conceptual Graph Model for W3C Resource Description Framework. In Ganter and Mineau [2000], pages 468–482.

[Cruellas *et al.*, 2003] cité p. 103
Juan-Carlos Cruellas, Jean-Paul Bodeveix, Thierry Millan, and Agusti Canals. The NEPTUNE Technology to verify and to Document Software Components. In Franck Barbier, editor, *Business Component-Based Software Engineering*, pages 101–118. Kluwer Academic Publishers, 2003.

[Davey et Priestley, 2002] cité p. 22
B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambride University Press, Cambridge, UK, 2002.

[Dean *et al.*, 2004] cité p. 38, 56, 57, 118
M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language Reference. Technical report, 2004. W3C Recommendation `http://www.w3.org/TR/owl-ref/`.

[Delugach, 2001] cité p. 34
H. S. Delugach. CharGer: A Graphical Conceptual Graph Editor. In *Proceedings of CG Tools Workshop of ICCS'01*, 2001. `http://charger.sourceforge.net/`.

[Dobrev et Toutanova, 2000] cité p. 34
Pavlin Dobrev and Kristina Toutanova. CGWorld - A Web Based Workbench for Conceptual Graphs Management and Applications. In Gerd Stumme, editor, *Proceedings of CG Tools Workshop of ICCS'00*, pages 243–256. Shaker Verlag, 2000.

[Eclipse community, 2008] cité p. 103
Eclipse community. The Eclipse Modeling Framework (EMF), 2008. `http://www.eclipse.org/modeling/emf/`.

[Falkenhainer et Forbus, 1991] cité p. 5
Brian Falkenhainer and Kenneth D. Forbus. Compositional Modeling : Finding the Right Model for the Job. *Artificial Intelligence*, 51(1-3) :95–143, 1991.

[Fikes *et al.*, 2004] cité p. 85
R. Fikes, P. Hayes, and I. Horrocks. OWL-QL - a language for deductive query answering on the Semantic Web. *Journal of Web Semantics*, 2(1) :19–29, 2004.

[Forster *et al.*, 1999] cité p. 34
Michael Forster, Andreas Pick, and Marcus Raitner. Graph template library. University of Passau, 1999. `http://infosun.fmi.uni-passau.de/GTL`.

[Ganter et Mineau, 2000] cité p. 199, 199
Bernhard Ganter and Guy W. Mineau, editors. *Conceptual Structures : Logical, Linguistic and Computational Issues, proceedings of the eighth International Conference on Conceptual Structures (ICCS'2000), Darmstadt, Germany*, volume 1867 of *Lecture Notes in Artificial Intelligence (LNAI)*. Springer, 2000.

[Genest, 2007] cité p. 33, 171, 172
David Genest. CoGITaNT 5.1.9 - Manuel de référence, 2007. `http://cogitant.sourceforge.net`.

[Gruber, 1993] cité p. 40
T. R. Gruber. A translation approach to portable ontologies. *Knowledge Acquisition*, 5(2) :199–220, 1993.

[Gutierez, 2005] cité p. 34
A. Gutierez. In *Proceedings of CG Tools Workshop of ICCS'05*, 2005. `http://www.lirmm.fr/cogui/`.

[Haarslev et Müller, 2001] cité p. 83
V. Haarslev and R. Müller. Racer system description. In *Proceedings of IJCAR'2001*, volume 2083 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 701–705. Springer, 2001.

[Himsolt, 1996] cité p. 34
Michael Himsolt. The Graphlet System. In *Proceedings of Graph Drawing*, volume 1190 of *Lecture Notes in Computer Science (LNCS)*, pages 233–240. Springer-Verlag, 1996.

[Horrocks *et al.*, 2004] cité p. 84
I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML, 2004. `http://www.w3.org/Submission/SWRL/`.

[Horrocks et Patel-Schneider, 2003] cité p. 79, 80
I. Horrocks and F. P. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. In *Proceedings of the second International Semantic Web Conference ISWC'03*, 2003.

[Horrocks, 1998] cité p. 83
I. Horrocks. Using an Expressive Description Logic: FaCT or Fiction ? In *Proceedings of KR'98*, pages 636–649, 1998.

[Kabbaj *et al.*, 2001] cité p. 34
A. Kabbaj, B. Moulin, J. Gancet, D. Nadeau, and O. Rouleau. Uses, Improvements, and Extensions of Prolog+CG: Case Studies. In Harry S. Delugach and Gerd Stumme, editors, *Conceptual Structures : Broadening the Base, proceedings of nineth International Conference on Conceptual Structures (ICCS 2001), Stanford, CA, USA*, volume 2120 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 346–359. Springer, 2001.

[Kabbaj, 1999] cité p. 34
Adil Kabbaj. Synergy as an Hybrid Object-Oriented Conceptual Graph Language. In Tepfenhart and Cyre [1999], pages 198–213.

[Kabbaj, 2006] cité p. 34
Adil Kabbaj. Development of Intelligent Systems and Multi-Agents Systems with Amine Platform. In Henrik Schärfe, Pascal Hitzler, and Peter Øhrstrøm, editors, *Conceptual Structures : Inspiration and Application, proceedings of 14th International Conference on Conceptual Structures (ICCS 2006), Aalborg, Denmark*, volume 4068 of *Lecture Notes in Computer Science (LNCS)*, pages 286–299. Springer, 2006.

[Kant, 1997] cité p. 56
Immanuel Kant. *Lectures on metaphysics*. Translated and edited by Karl Ameriks and Steve Naragon. Cambridge University Press, 1997.

[Karvounarakis *et al.*, 2002] cité p. 85
G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In Peter Gray, Peter King, and Alexandra Poulovassilis, editors, *Proceedings of the eleventh International World Wide Web Conference WWW'02*, pages 592–603. ACM, 2002.

[Kayser, 1997] cité p. 128, 141
D. Kayser. *La représentation des connaissances*. Collection informatique, Hermès edition, 1997.

[Klyne et Carroll, 2004] cité p. 38, 46, 70
G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. Technical report, 2004. W3C Recommendation `http://www.w3.org/TR/rdf-concepts/`.

[Knublauch *et al.*, 2004] cité p. 84
H. Knublauch, R. W. Fergerson, N. F. Noy, and M. A. Musen. The Protege OWL plugin: An Open Development Environment for Semantic Web Applications. In *Proceedings of the 3rd International Semantic Web Conference ISWC'04*, 2004.

[Lehmann, 1992] cité p. 7, 70
F. Lehmann. *Semantic Networks in Artificial Intelligence*. Pergamon Press, 1992.

[Levy et Rousset, 1998] cité p. 5, 83, 159
Alon Y. Levy and Marie-Christine Rousset. Combining Horn rules and description logics in CARIN. *Artificial Intelligence*, 104(1-2) :165–209, 1998.

[MacGregor et Bates, 1987] cité p. 82
R. M. MacGregor and R. Bates. The LOOM knowledge representation language. In *Proceedings of Knowledge-Based Systems Workshop*, 1987.

[Martin, 2007] cité p. 119
Philippe Martin. Knowledge Representation/Translation in RDF+OWL, N3, KIF, UML and the WebKB-2 languages (For-Links, Frame-CG, Formalized English), 2007. `http://www.webkb.org/doc/model/comparisons.html`.

[McKenzie *et al.*, 2004] cité p. 85
C. McKenzie, P. Gray, and A. Preece. Extending SWRL to Express Fully-Quantified Constraints. In *Proceedings of RuleML'04*, 2004.

[Mehlhorn et Näher, 1999] cité p. 34
Kurt Mehlhorn and Stefan Näher. *LEDA - A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999. `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[Minsky, 1980] cité p. 70, 87
M. Minsky. A Framework for Representing Knowledge. In D. Metzing, editor, *Frame Conceptions and Text Understanding*, pages 1–25. de Gruyter, Berlin, 1980.

[Moor, 2003] cité p. 1
James H. Moor, editor. *The Turing Test : The Elusive Standard of Artificial Intelligence*. 2003. ISBN: 1-4020-1205-5.

[Motik *et al.*, 2004] cité p. 84
B. Motik, U. Sattler, and R. Studer. Query Answering for OWL-DL with Rules. In *Proceedings of the 3rd International Semantic Web Conference ISWC'04*, 2004.

[Mugnier et Chein, 1996] cité p. 3, 5, 7, 9, 9, 10, 11, 12, 13, 25, 26, 35
M. L. Mugnier and M. Chein. Représenter des connaissances et raisonner avec des graphes. *Revue d'Intelligence Artificielle (RIA)*, 10(1) :7–56, 1996.

[Napoli, 1997] cité p. 70
A. Napoli. Une introduction aux logiques de descriptions, 1997. Tech. Report 3314, INRIA.

[Nebel, 1990a] cité p. 82
Bernhard Nebel. *Reasoning and Revision in Hybrid Representation Systems*. Lecture Notes in Artificial Intelligence (LNAI). Springer, 1990.

[Nebel, 1990b] cité p. 80
Bernhard Nebel. Terminological Reasoning is Inherently Intractable. *Artificial Intelligence*, 43 :235–249, 1990.

[OMG, 2002] cité p. 88, 101, 171
OMG. XML Metadata Interchange (XMI) Specification, 2002. `http://www.omg.org/cgi-bin/doc?formal/2002-01-01`.

[OMG, 2003a] cité p. 88, 88, 90, 97, 99, 100, 108, 110
OMG. Documentation de l'OMG, *Unified Modeling Language (UML) Specification: Infrastructure, version 2.0*, 2003. `http://www.omg.org/docs/ptc/03-09-15.pdf`.

[OMG, 2003b] cité p. 88, 88, 90, 97, 99
OMG. Documentation de l'OMG, *Unified Modeling Language (UML) Specification: Superstructure, version 2.0*, 2003. `http://www.omg.org/docs/ptc/03-08-02.pdf`.

[OMG, 2003c] cité p. 88, 100, 100
OMG. UML 2.0 OCL Specification, 2003. `http://www.omg.org/cgi-bin/doc?ptc/2003-10-14`.

[Patel-Schneider *et al.*, 2004] cité p. 70, 80
Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL Web Ontology Language; Semantics and Abstract Syntax. Technical report, 2004. W3C Recommendation `http://www.w3.org/TR/owl-semantics/`.

[Patel-Schneider, 1991] cité p. 82
P. F. Patel-Schneider. The CLASSIC knowledge representation system: Guiding principals and implementation rationale. *SIGART Bulletin*, 2(3) :108–113, 1991.

[Peltason, 1991] cité p. 82
C. Peltason. The BACK system - an overview. *SIGART Bulletin*, 2(3) :114–119, 1991.

[Peterson *et al.*, 2006] cité p. 41, 41, 50, 51, 52
David Peterson, Paul V. Biron, Ashok Malhotra, and C. M. Sperberg-McQueen. XML Schema 1.1 Part 2: Datatypes. Technical report, 2006. W3C Working Draft `http://www.w3.org/TR/xmlschema11-2/`.

[Prudhommeaux et Seaborne, 2008] cité p. 85
E. Prudhommeaux and A. Seaborne. SPARQL Query Language for RDF. Technical report, 2008. W3C Recommendation `http://www.w3.org/TR/rdf-sparql-query/`.

[Quillian, 1969] cité p. 7, 87
R. Quillian. Semantic Memory. In Marvin Minsky, editor, *Semantic Information Processing*. MIT Press, 1969.

[Raggett *et al.*, 1999] cité p. 37
Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 Specification. Technical report, 1999. W3C Recommendation `http://www.w3.org/TR/html401/`.

[Raimbault *et al.*, 2005] cité p. 5
Thomas Raimbault, David Genest, and Stéphane Loiseau. A new method to interrogate and check UML class diagrams. In Frithjof Dau, Marie-Laure Mugnier, and Gerd Stumme, editors, *Proceedings of the 13th International Conference on Conceptual Structures (ICCS'2005)*, volume 3596 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 353–366, Kassel, Germany, 2005. Springer.

[Raimbault *et al.*, 2006a] cité p. 5
Thomas Raimbault, Henri Briand, Rémi Lehn, and Stéphane Loiseau. Interrogation et Vérification de documents OWL dans le modèle des Graphes Conceptuels. In Gilbert Ritschard and Chabane Djeraba, editor, *Actes des 6èmes journées Extraction et Gestion des Connaissances (EGC 2006)*, volume RNTI-E-6 of *Revue des Nouvelles Technologies de l'Information (RNTI) - Extraction et gestion des connaissances*, pages 187–192, Lille, France, 2006. Cépaduès Éditions.

[Raimbault *et al.*, 2006b] cité p. 5
Thomas Raimbault, David Genest, and Stéphane Loiseau. Interrogation et vérification des diagrammes de classes UML : une approche fondée sur le modèle des graphes conceptuels. In *Actes du 15 ème Congrès Francophone Reconnaissance des Formes et Intelligence Artificielle (RFIA'2006)*, Tours, France, 2006. Presses Univeritaires François-Rabelais.

[Reiter, 1980] cité p. 76, 164
Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13 :81–132, 1980.

[Richters, 2002] cité p. 100
M. Richters. A Precise Approach to Validating UML Models and OCL Constraints, 2002. Ph.D. thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14.

[Rosati, 2007] cité p. 159
Riccardo Rosati. The Limits of Querying Ontologies. In *Proceedings of 11th International Conference in Database Theory (ICDT'07)*, volume 4353 of *Lecture Notes in Computer Science (LNCS)*, pages 164–178. Springer, 2007.

[Roth et Schmitt, 2007] cité p. 5
Andreas Roth and Peter H. Schmitt. *Formal Specification*, volume 4334 of *Lecture Notes in Computer Science (LNCS)*, chapter 5, pages 245–294. Springer, 2007.

[Salvat et Mugnier, 1996] cité p. 16
Éric Salvat and Marie-Laure Mugnier. Sound and Complete Forward and Backward Chaining of Graph Rules. In Peter W. Eklund, Gerard Ellis, and Graham Mann, editors, *Conceptual Structures : Knowledge Representation as Interlingua, proceedings of the fourth International Conference on Conceptual Structures (ICCS'96), Sydney, Australia*, volume 1115 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 248–262. Springer, 1996.

[Salvat, 1997] cité p. 16
Éric Salvat. *Raisonner avec des opérations de graphes : Graphes conceptuels et règles d'inférence*. PhD thesis, Université Montpellier II, France, 1997.

[Salvat, 1998] cité p. 7, 16
Éric Salvat. Theorem Proving Using Graph Operations in the Conceptual Graph Formalism. In *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI'98), Brighton, UK*, 1998.

[Schmidt-Schauss et Smolka, 1991] cité p. 79
M. Schmidt-Schauss and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1) :1–26, 1991.

[Schreiber, 2002] cité p. 119
Guus Schreiber. A UML Presentation Syntax for OWL Lite, 2002. Incomplete Draft `http://hcs.science.uva.nl/usr/Schreiber/docs/owl-uml/owl-uml.html`.

[Sirin *et al.*, 2004] cité p. 83, 171
E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner, 2004. Submitted for publication to Journal of Web Semantics.

[Southey et Linders, 1999] cité p. 34
F. Southey and J. G. Linders. Notio - A Java API for Developing CG Tools. In Tepfenhart and Cyre [1999], pages 262–271.

[Sowa, 1984] cité p. 3, 7, 13, 13, 16
J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison Wesley, 1984.

[Studer *et al.*, 1998] cité p. 40
R. Studer, R. Benjamins, and D. Fensel. Knowledge Engineering: Principles and Methods. *Data Knowledge Engineering*, 25(1-2) :161–197, 1998.

[Tepfenhart et Cyre, 1999] cité p. 201, 204
William Tepfenhart and Walling Cyre, editors. *Conceptual Structures : Standards and Practices, proceedings of the seventh International Conference on Conceptual Structures (ICCS'99), Blacksburg, USA*, volume 1640 of *Lecture Notes in Artificial Intelligence (LNAI)*. Springer, 1999.

[Thompson *et al.*, 2006] cité p. 41
Henry S. Thompson, C. M. Sperberg-McQueen, Shudi (Sandy) Gao, Noah Mendelsohn, David Beech, and Murray Maloney. XML Schema 1.1 Part 1: Structures. Technical report, 2006. W3C Working Draft `http://www.w3.org/TR/xmlschema11-1/`.

[W3C, 2002] cité p. 41
HTML Working Group W3C. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). Technical report, 2002. W3C Recommendation `http://www.w3.org/TR/xhtml1/`.

[Warmer et Kleppe, 1998] cité p. 3, 100
J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, 1998.

[Warmer et Kleppe, 2003] cité p. 160
J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley, second edition, 2003.

Liste des publications personnelles

Revue francophone avec comité de sélection

[1] Thomas Raimbault, Henri Briand, David Genest, Rémi Lehn et Stéphane Loiseau. Une synthèse des modèles de représentation des connaissances à base de Graphes Conceptuels et OWL. Cépaduès Éditions, numéro spécial *Modélisation des Connaissances* de la Revue des Nouvelles Technologies de l'Information (RNTI), 2008.

Conférence internationale avec comité de sélection

[2] Thomas Raimbault, David Genest et Stéphane Loiseau. A New Method to Interrogate and Check UML Class Diagrams. In Frithjof Dau, Marie-Laure Mugnier and Gerd Stumme editors, *Proceedings of the 13th International Conference on Conceptual Structures (ICCS'05)*, volume 3596 of Lecture Notes in Artificial Intelligence (LNAI), pages 353–366, Kassel, Germany, 2005. Springer.

Conférences francophones avec comité de sélection

[3] Thomas Raimbault, David Genest et Stéphane Loiseau. Interroger et vérifier des diagrammes de classes UML : une approche fondée sur le modèle des graphes conceptuels. Dans les *Actes du 15ème congrès francophone Reconnaissance des Formes et Intelligence Artificielle (RFIA'06)*, Tours, France. 2006.

[4] Thomas Raimbault, Henri Briand, Rémi Lehn et Stéphane Loiseau. Interrogation et Vérification de documents OWL dans le modèle des Graphes Conceptuels. Dans les *Actes des 6èmes journées francophones Extraction et Gestion des Connaissances (EGC'06)*, volume RNTI-E-6 de la Revue des Nouvelles Technologies de l'Information (RNTI), pages 187–19, Lille, France, 2006. Cépadués Éditions.

Ateliers francophones avec comité de sélection

[5] Thomas Raimbault, David Genest Stéphane Loiseau et Henri Briand. Modélisation des connaissances : une confrontation entre le modèle des Graphes Concep-

tuels et le langage OWL. Dans *Atelier Modélisation des Connaissances, 7ème journées francophones Extraction et Gestion des Connaissances (EGC'07)*, pages 36–45, Namur, Belgique, 2007.

[6] Thomas Raimbault. Une nouvelle méthode graphique pour interroger et vérifier des diagrammes de classes UML. Dans *Atelier Modélisation des Connaissances, 5ème journées francophones Extraction et Gestion des Connaissances (EGC'05)*, volume RNTI-E-5 de la Revue des Nouvelles Technologies de l'Information (RNTI), pages 7–11, Paris, France, 2005. Cépadués Éditions.

Mémoire de DEA

[7] Thomas Raimbault. Une nouvelle méthode à base de graphes pour interroger et vérifier des diagrammes de classes UML. Mémoire de DEA, Université d'Angers, Laboratoire d'Études et de Recherche en Informatique d'Angers (LERIA), 2004.