UNIVERSITÉ PIERRE ET MARIE CURIE - PARIS VI

ÉCOLE DOCTORALE 393

THÈSE

pour obtenir le titre de

Docteur en Sciences

de l'Université Pierre et Marie Curie - Paris VI

Mention : Informatique

Présentée et soutenue par

Amina Chniti

Gestion des dépendances et des interactions entre Ontologies et Règles Métier

Thèse dirigée par Patrick Albert et Jean Charlet

Jury

M. Aldo Gangemi, Professeur, Université Paris 13 (*Rapporteur*)
M. Régis Duvauferrier, PU-PH, Université Rennes 1 (*Rapporteur*)
M. Jean Charlet, AP-HP, INSERM (*Directeur*)
M. Patrick Albert, IBM, CAS France (*Directeur*)
Mme. Adeline Nazarenko, Professeur, Université de Paris 13 (*Examinateur*)
M. Khaled Ghedira, Professeur, Université de Tunis (*Examinateur*)
M. Patrick Galinari, Professeur, Université de Paris 6 (*Examinateur*)

Remerciements

Ce mémoire de thèse est le résultat de trois ans de travail acharné qui n'aurait pas pu être achevé sans le soutien, les conseils et l'aide de nombreuses personnes.

Je tiens ainsi à remercier,

M. Patrick Albert, mon mentor. Je ne le remercierai jamais assez pour m'avoir offert l'opportunité d'effectuer cette thèse. Son soutien inconditionnel, ses encouragements sans limite et ses conseils très précieux, tout au long de ma thèse et plus encore, m'ont permis d'évoluer sur le plan professionnel et personnel. Je le remercie énormément de m'avoir mis, à plusieurs reprises, au devant de la scène, en me confiant des tâches importantes qui m'ont permis d'améliorer et d'avoir confiance en mes compétences. Mis à part sa personnalité hors norme, travailler avec lui est une véritable aventure.

M. Jean Charlet, mon directeur de thèse, pour avoir accepté de diriger ma thèse et pour m'avoir accueillie dans son équipe. Je le remercie pour ses conseils et ses recommandations, pour m'avoir guidé tout au long de cette thèse tout en m'accordant la liberté et la confiance nécessaire pour faire évoluer mon travail. Je le remercie énormément pour les relectures multiples de ce mémoire et pour ces corrections minutieuses. C'est vraiment un honneur et un véritable plaisir de l'avoir comme directeur de thèse.

Les membres du jury qui m'ont fait l'honneur d'accepter de juger ce travail.

Adil El Ghali, pour ses conseils mais surtout pour ses critiques pertinentes, constructives et sans pitié qui m'ont permis de toujours donner le meilleur de moi même et de me surpasser.

Abdellali Boussadi, pour sa collaboration et son implication pour l'élaboration des expérimentations dans le domaine médical. Je le remercie de m'avoir fournit les ressources nécessaires, dans la limite du possible, pour mener à terme ces expérimentations.

Les membres de IBM France Lab et en particulier ceux du CAS avec qui j'ai énormément appris. Je les remercie pour cette ambiance chaleureuse et conviviale et pour leur bonne humeur. Je remercie particulièrement Hugues Citeau, Hassan Aït-Kaci, Thomas Baudel, Christian de Sainte Marie et les futurs Docteur du CAS, Bertjan Broeksema, Valerio Cosentino et Penelope Agular Melgareja.

Les membres de l'équipe BRMS et en particulier Bruno Berstel-Da Silva, Changhai Ke, Philippe Bonnard et Michel Leconte et Daniel Selman qui m'ont beaucoup aidé à résoudre les différentes problématiques auxquelles j'ai été confrontée durant mes recherches.

Les membres de l'équipe 20 de l'INSERM. Je les remercie pour leur bonne humeur et pour les moments inoubliables que nous avons passé ensemble.

Mes merveilleux amis que j'adore. Je les remercie pour m'avoir soutenu, encouragé et surtout supporté particulièrement pendant la période de rédaction de ma thèse. Je remercie Yosra Ben Mustapha, Amel Haji, Emna Trabelsi, Wael Kouni et Mahdi Aouadi et je leur souhaite plein de bonheur, d'amour et de réussite. Je remercie aussi Nouha Omrane pour son coup de main.

Last but not least, ma Famille. Je ne remercierai jamais assez mes parents pour tout ce qu'ils m'ont donné. Je ne serai jamais arrivée là où je suis aujourd'hui sans leur soutien et leurs encouragements. Je remercie au même titre Tannour et Hamadi pour avoir toujours été présents. Je remercie ma petite sœur Noussa, Likou, M'nawar et Ali et j'espère qu'ils trouveront enfin dans ce travail la réponse à leur question « mais qu'est ce que tu cherches ? ! ». Je remercie énormément le meilleur papi au monde, Babouti, que j'adore. Enfin, je dédie ce travail à Mami Wassila. J'ai toujours pensé qu'elle serait la première personne à m'appeler pour me féliciter après ma soutenance, mais malheureusement la vie en a voulu autrement.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction Générale

Sommaire

1.1 Contexte général

1.1.1 Contexte industriel

Cette thèse a été effectuée au sein du *Center for Advanced Studies* (CAS). Le CAS est une entité appartenant au France Lab de IBM et travaille en étroite collaboration avec des universitaires. Cette collaboration est effectuée via des projets collaboratifs européens ou nationaux (*e.g.* ONTORULE[1], Optimod[2], Rider[3]...) qui impliquent des clients et partenaires stratégiques de IBM mais aussi des chercheurs universitaires, doctorants et stagiaires. Parmi les universités collaborant avec le CAS, nous pouvons citer l'Université de Paris 13 (LIPN), l'Université de Montpellier (LIRMM), l'Université de Toulouse (IRIT), l'Université de Lyon (LIRIS), ...

1. http ://ontorule-project.eu/
2. http ://www.optimodlyon.com/
3. http ://www.rider-project.com

Les principales thématiques traitées par ces projets sont : l'automatisation des décision, l'optimisation, les *Smarter Cities*...

1.1.2 Contexte universitaire

Ce travail a été effectué au sein de l'INSERM U872, équipe 20 [4] spécialisée dans l'ingénierie de la connaissance dans le domaine de la santé. Le projet de l'équipe est de proposer des méthodes innovantes de gestion informatisée des données et des connaissances complexes en médecine à des fins d'amélioration de la qualité des soins et de la sécurité des patients. Ainsi, les principales thématiques traitées dans cette équipe sont : la mise en œuvre informatique de systèmes d'aide à la décision, la construction d'ontologies pour l'aide au codage médical du dossier patient, l'informatisation des guides de bonnes pratiques, l'évaluation de l'impact des aides informatisées sur la pratique médicale...

1.1.3 Contexte de recherche

Cette thèse s'est déroulée dans le contexte du projet Européen ONTORULE (*ONTOlogies meet Business RULEs*). Ce projet a commencé en 2009 pour une durée de trois ans et compte un certain nombre de partenaires universitaires (*e.g.* l'Université de Paris 13, l'Université de Vienne et l'Université de Bolzano), industriels (*e.g.* IBM, Ontoprise) et des clients (Audi et Arcelor Mittal).

ONTORULE propose une approche orientée experts/utilisateurs métier dont l'objectif principal est de permettre l'automatisation des décisions métier écrites dans des documents textuels. Pour cela, il a fallu développer des méthodes permettant de formaliser des politiques métier (*Business policies*) afin de pouvoir en extraire des règles métiers exécutables par des machines. Dans ce contexte, deux axes ont été suivis :

- le premier axe consiste à extraire des ontologies à partir de sources textuelles, les formaliser en OWL (Web Ontology Language) pour ensuite être exploitées par des systèmes de gestion de règles métier ;
- le deuxième axe consiste à extraire des règles métier à partir des documents textuels, les formaliser en RIF (Rule Interchange Format) pour

4. http ://ics.upmc.fr/

ensuite les exploiter via des systèmes de gestion de règles métier (Guisse, 2013) .

1.2 Objectifs de la thèse

Dans la majorité des systèmes d'information, les connaissances du métier sont toujours codées dans un langage de programmation tel que Java, C, C++ ...De ce fait et vue l'évolution quotidienne des domaines métier, il est toujours nécessaire d'avoir l'intervention d'un technicien, qui n'a pas les connaissances nécessaires du domaine métier, pour mettre à jour le système d'information. Ceci devient de plus en plus compliqué à gérer vue la rapidité de ces évolutions.

L'objectif principal de la thèse est de permettre aux experts métier de gérer eux même ces évolutions sans avoir recourt à un technicien, et de leur permettre d'être impliqués et de participer au développement de l'application qu'ils utilisent pour accomplir les tâches quotidiennes de leur travail. Ceci revient à séparer la logique métier de l'application codée dans un langage de programmation et d'avoir un moyen qui permet aux experts métier de modéliser eux même cette logique dans un langage facile à comprendre et à manipuler. D'où l'idée de s'orienter vers les règles métier et donc les Systèmes de Gestion des Règles Métier (SGRM).

L'utilisation des SGRM s'est généralisée depuis une dizaine d'années et gagne de plus en plus de succès grâce aux règles métier qui permettent d'exprimer une logique métier dans un langage pseudo-naturel, le langage naturel contrôlé (Controlled Natural Language). Ce genre de système nécessite une modélisation des connaissances du domaine métier et utilise donc des modèles orientés objet pour modéliser les entités et les relations entre les entités d'un domaine donné.

Étant donnée les limites du pouvoir d'expressivité des modèles orientés objets, l'avènement des ontologies et particulièrement l'avènement de OWL (Web Ontology Language) et son pouvoir d'expressivité, il nous est donc apparu intéressant et innovant de trouver un moyen d'apporter l'expressivité de OWL aux experts métier, sans qu'ils aient connaissances des ontologies, et ce au moyen des règles métier. D'où l'idée d'intégrer les ontologies et les règles.

1.3 Problématique

Notre approche consiste donc à intégrer des ontologies et des règles au moyen des SGRMs. Avec la puissance d'expressivité de OWL, de plus en plus d'entreprise s'orientent vers la modélisation à base d'ontologies. La question de fond est alors : comment permettre aux experts métier de manipuler les connaissances de leur domaine dont la sémantique est formalisée dans un langage formel, quasiment incompréhensible par les non spécialistes ?

Les SGRM sont fondés sur des modèles orientés objets. Le problème majeur rencontré est la gestion de la différence de représentation des connaissances entre les modèles orientés objets et OWL. Les modèles orientés objets sont déclaratifs et toutes les connaissances doivent explicitement être modélisées alors qu'avec OWL la majorité des connaissances sont inférées grâce aux axiomes et aux raisonneurs. Comment intégrer des connaissances qui ne sont pas explicitement modélisées dans un système orienté objets où les connaissances doivent être explicitement déclarées ?

Ensuite, dans le cas où les ontologies et les règles sont intégrées, une nouvelle problématique se pose. Les règles dépendent des entités modélisées dans l'ontologie. De ce fait, l'évolution des ontologies a un impact sur les règles et peut causer des problèmes de cohérence. Étant donné l'aspect orienté experts et utilisateurs métier de notre travail, comment permettre à ces deux catégories d'intervenants de gérer eux-mêmes l'évolution des ontologies et leur impact sur les règles ?

1.4 Approches proposées

Afin d'atteindre notre objectif, nous proposons deux approches complémentaires. La première approche permet aux experts métier de développer leurs règles métier à partir d'ontologies OWL et la deuxième permet aux experts métier de gérer l'impact de l'évolution des ontologies sur les règles.

Pour la première approche, l'idée est d'importer des ontologies dans les SGRM pour ensuite utiliser toutes les fonctionnalités offertes par ces systèmes afin d'éditer et exécuter des règles métier écrites en langage naturel contrôlé. Pour la deuxième approche, sujet principal de cette thèse, l'idée est de développer une méthode, que nous avons appelé $\mathcal{MDR}$ ($\mathcal{M}$odéliser - $\mathcal{D}$étecter -

$\mathcal{R}$éparer) qui permet de détecter des changements d'ontologies, analyser leur impact sur les règles afin de détecter d'éventuels problèmes de cohérence entre les deux modèles et de proposer des solutions, appelées réparations pour réparer les incohérences causées. L'approche proposée est une approche orientée utilisateurs métier et est fondée sur les systèmes de gestion des règles métier.

1.5 Plan du mémoire

Ce mémoire est constitué de 5 chapitres. Après ce premier chapitre d'introduction, nous présentons le chapitre sur l'état de l'art (Chapitre 2) décrivant les principaux champs de recherche concernés par nos travaux. Ensuite, nous poursuivrons avec le chapitre décrivant notre contribution (Chapitre 3), puis le chapitre présentant les expérimentations effectuées pour tester les solutions proposées (Chapitre 4). Enfin, une conclusion (Chapitre 5) qui fait le point sur les principales contributions de la thèse, les limites de l'approche proposée et les perspectives.

Chapitre 2 : État de l'art – ce chapitre présente une vue globale des différents travaux existant dans des champs relatifs au sujet de cette thèse à savoir la représentation des connaissances en particulier avec des ontologies et règles, la gestion des évolutions et de la cohérence des ontologies, l'intégration des ontologies et des règles...

Chapitre 3 : $\mathcal{MDR}$: Approche de gestion de l'évolution de modèles – ce chapitre présente l'approche $\mathcal{MDR}$ (Model-Detect-Repair) proposée pour résoudre la problématique de gestion de l'impact de l'évolution des ontologies sur les règles. Ce chapitre présente le framework général de notre approche en utilisant les formalismes proposés par la méthode *CommonKADS*. Il décrit son architecture et donne une description détaillée de ses différents composants, leur spécification ainsi que le processus défini pour la gestion de l'impact de l'évolution des ontologies.

Chapitre 4 : Implémentation et expérimentations – en première partie, ce chapitre décrit la méthode implémentée pour permettre l'édition et l'exécution des règles métier à partir d'ontologies OWL et l'outil utilisé pour implémenter notre approche. En deuxième partie, il décrit les

différents cas d'études mis en œuvre pour tester et valider notre approche.

Chapitre 2

État de l'art

Sommaire

2.1 Introduction

L'objectif de ce chapitre est de présenter les différents axes sur lesquels repose notre travail. Ce chapitre est composé de quatre parties principales.

Nous commençons d'abord par présenter les différentes techniques de représentation des connaissances ainsi que les langages existant pour les formaliser. L'objectif n'est pas de fournir un état de l'art complet sur les méthodes de représentation des connaissances mais de faire une synthèse qui permet de déterminer quelle est la méthode de représentation à utiliser en fonction du besoin, de l'objectif et des types de connaissances.

Ensuite, nous présentons un état de l'art sur les évolutions des ontologies : les opérations de changements, les principales classifications de changements

dans la littérature ainsi que les approches de représentation de ces changements et les approches de gestion des évolutions d'ontologies.

La troisième partie présente les approches d'intégration des ontologies et des règles ainsi qu'une vue générale sur les systèmes de gestion des règles métier.

La quatrième partie positionne notre problématique par rapport aux différents axes décrit précédemment et discute des différents choix que nous avons effectués ainsi que des hypothèses élaborées.

2.2 Les techniques de représentation des connaissances

La représentation des connaissances est une question majeure à laquelle des chercheurs en sciences cognitives et en intelligence artificielle accordent une très grande importance. En sciences cognitives, les chercheurs se focalisent sur les méthodes de mémorisation et de traitement de l'information chez l'être humain. En intelligence artificielle, l'objectif est d'arriver à simuler le raisonnement humain. La représentation des connaissances consiste ici à modéliser les connaissances de manière à ce qu'un logiciel informatique puisse utiliser cette modélisation et la traiter afin d'efféctuer des raisonnements et inférer de nouvelles connaissances.

Il existe différentes manières pour représenter des connaissances. Pour choisir la bonne technique de représentation, il faut d'abord savoir deux choses essentielles : pourquoi représenter des connaissances et quel est le type de la connaissance à représenter.

Ceci étant, il n'existe pas de technique de représentation spécifique pour chaque catégorie de connaissance. Comme nous l'avons déjà dit, la technique de représentation dépend des besoins et du but visé par le système à développer. Dans ce qui suit, et en se fondant sur (Gasevic *et al.*, 2000), nous présentons une breve description des techniques les plus utilisées en IA, tels que les triplets, les réseaux sémantique, les graphes conceptuel,... (voir Section 2.2.1) et nous nous focalisons principalement sur les ontologies (voir Section 2.2.3) et les règles (voir Section 2.2.2).

2.2.1 Quelques techniques de représentation des connaissances

triplets Objet - Attribut - Valeur Cette technique est utilisée pour représenter des faits sur des objets et leurs attributs. Elle permet d'assigner une ou plusieurs valeurs à un attribut d'un objet. Par exemple, la phrase «la balle est de couleur jaune» peut être représentée sous forme de triplet de la manière suivante : <Balle - couleur - jaune>. Il est aussi possible d'avoir une représentation graphique du triplet (voir Figure 2.1).

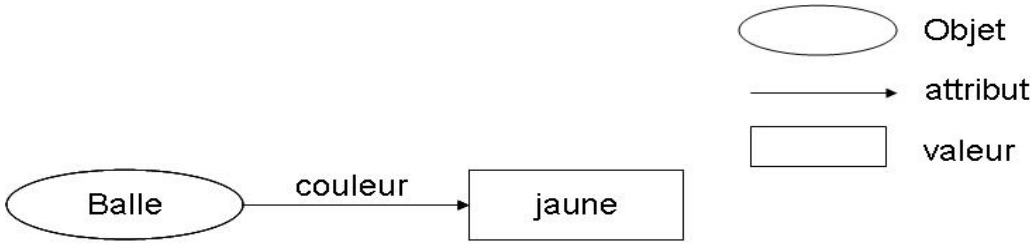

FIGURE 2.1 – Exemple d'un triplet <Objet - Attribut - Valeur>.

Généralement, un objet a plusieurs attributs. Un attribut peut être mono-valué comme décrit dans la Figure 2.1, ou bien multi-valué (*e.g.* <Voyage - catégorie - business, économique>). Dans le cas où un objet possède plusieurs attributs, la représentation graphique consiste à ajouter une flèche pour chaque attribut, dans le cas où un attribut est multi-valué, la représentation graphique consiste à ajouter un cadre pour chaque valeur de l'attribut.

Les réseaux sémantiques Les réseaux sémantiques sont une technique de représentation des connaissances qui reflètent la sémantique de la mémoire humaine. Il sont représentés sous forme de graphe dont les nœuds représentent les objets et les concepts d'un domaine donné et les arcs représentent les différents liens entre les nœuds. Les arcs sont munis d'un label qui dénote les relations entre les concepts (voir Figure 2.2).

Il n'existe pas une représentation graphique standard des réseaux sémantique, néanmoins ils permettent toujours de représenter

- les concepts (classes), les objets (instances), les attributs d'un concept ;
- les relations entre les classes et les objets grâce à la relation *is-a* ;

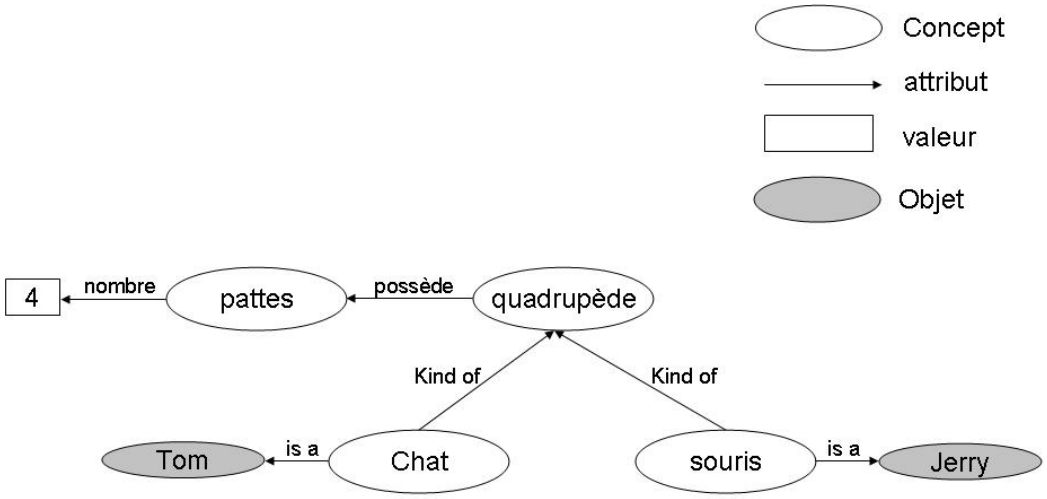

FIGURE 2.2 – Exemple d'un réseau sémantique.

- la valeur d'un attribut. Il est aussi possible de représenter une valeur par défaut (*e.g.* «un quadrupède à quatre pattes») ;
- la relation d'héritage grâce à la relation *Kind-of*.

Il est aussi possible de représenter les parties d'un réseau sémantique sous forme de triplets (*e.g.* < quadrupède - possède - pattes>, < pattes - nombre - 4>). Les réseaux sémantiques sont connus pour être intuitifs et faciles à comprendre. Néanmoins, ils sont aussi connus pour avoir un faible pouvoir d'expressivité pour la formalisation des exceptions et leur sémantique reste floue.

Par exemple, avec le triplet <Balle, couleur, jaune>, nous ne pouvons pas savoir si chaque balle a une couleur différente ou si toutes les balles sont de couleur jaune. Aussi, il est difficile avec cette technique de représenter l'inférence. Même si la subsomption est bien gérée par les réseaux sémantiques, il est possible d'avoir des transitivités qui ne sont pas correctes. Par exemple, pour les triplets <Voiture - est un - Véhicule>, <Siège - partie de - Voiture> il est possible d'avoir par transitivité <Siège - est un - Véhicule> assertion qui n'est évidemment pas vraie.

Les graphes conceptuels Les graphes conceptuels, créés par John Sowa, sont un formalisme de représentation des connaissances qui allient le pouvoir d'expression du langage naturel avec le formalisme de la logique de premier

ordre (Sowa, 2000). Ils ont été conçus pour représenter la sémantique du langage naturel et ont évolué pour devenir des systèmes complets au sens de la logique. Ils sont fondés sur les graphes existentiels de Charles Sanders Peirce et les réseaux sémantiques. Les graphes conceptuels sont le résultat d'une synthèse de plusieurs représentations sémantiques des connaissances d'où la flexibilité et le pouvoir d'expressivité de ce formalisme. Ainsi, ils sont utilisés dans différents domaines d'applications tels que le traitement du langage naturel, la spécification des moteurs d'inférences, l'acquisition et extraction des connaissances,...

Généralement, un graphe conceptuel est défini comme étant un graphe orienté, fini, connexe et bipartie. Ainsi, il ne possède que deux types de nœuds :

- les nœuds concepts : ils représentent les objets du domaine et sont représentés par un rectangle dans la notation graphique ou bien entre crochet. Un concept est composé de deux parties ; le type d'un objet représenté et un marqueur représentant une instance de cet objet. Par exemple, le nœud concept étiqueté par `Vol:*` représente un vol en général et est qualifié de *nœud général*. Le nœud étiqueté par `Vol:AF447` représente un vol spécifique ayant le numéro AF447, ce nœud est qualifié de *nœud spécifique*.
- les nœuds relations : ils représentent les relations entre les différents concepts et sont généralement représentés pas un cercle ovale ou bien entre parenthèse et possèdent un arc entrant qui relie un concept à une relation et arc sortant qui relie la relation à un concept.

  `[Vol]` ⟶ `(effectué par)` ⟶ `[Compagnie aérienne]`

La représentation des graphes conceptuels peut être formalisée en logique de premier ordre ce qui offre une sémantique opérationnelle permettant la cohérence des inférences et des interprétations, deux choses qu'on ne peut avoir avec les réseaux sémantiques.

2.2.2 Les règles

Les règles sont une technique de représentation des connaissances constituées d'une ou plusieurs *prémisses* (appelée aussi *conditions* ou *antécédents*) reliées à une ou plusieurs *conclusions* (appelée aussi *conséquences* ou *action*). Une règle est généralement écrite sous la forme `SI - ALORS` (voir exemple

ci-dessous), les *prémisses* sont décrites dans la partie `SI` de la règle et représentent des faits ou des situations et les *conclusions* sont décrites dans la partie `ALORS` et représentent soit de nouveaux faits soit des actions à appliquer. Les règles peuvent être catégorisées en fonction de l'impact des conclusions modélisées (*i.e.* règles logiques, règles de production) ou en fonction des connaissances modélisées (*i.e.* règles probabilistes, règles floues, règles métier).

- **Les règles logiques.** Les règles logiques permettent d'inférer de nouveaux faits lorsque les prémisses sont vérifiées. Une des règles logique la plus connue est :

  ```
  SI    tout homme est mortel
  ET    Socrate est un homme
  ALORS Socrate est mortel
  ```

  Dans ce cas, un nouveau fait *Socrate est mortel* sera ajouté à la base des faits.
- **Les règles de production.** Les règles de production représentent les conditions nécessaires pour effectuer une transition d'état ou une action. Dans ce cas, la règle décrit dans sa prémisse un ensemble d'états - situations initiales qui peuvent se produire et la partie conclusion modifie cet ensemble en exécutant une opération (*e.g.* procédure ou méthode). Par exemple :

  ```
  SI    tout homme est mortel
  ET    Socrate est un homme
  ALORS ajouter Socrate à l'ensemble des mortels
  ```

  Dans ce cas, l'ensemble représentant le commun des mortels sera modifié en y ajoutant l'individu Socrate.
- **Les règles probabilistes.**

  Les règles probabilistes permettent d'attribuer un *facteur de certitude* aux prémisses et aux conclusions d'une règle et permettent d'inférer des conclusions incertaines.

  ```
  SI    le projet est accepté (FC = 0.5)
  ET   le financement est accordé
  ALORS le projet peut démarrer (FC = -0.5)
  ```

  Il est possible que le *facteur de certitude* ne soit pas spécifié, dans ce cas il est soit égal à celui inféré dans la prémisse ou conclusion précédente soit il faut prendre celui défini par défaut. Les conclusions inférées à partir d'une prémisse probabiliste ne peuvent être considérées comme

«certainement vrai» et les prémisses qui sont «absolument certaines» peuvent conduire à des conclusions incertaines.

– **Les règles floues.** Les règles floues permettent de représenter des ensembles flous en utilisant des termes qui peuvent être interprétés différemment selon le contexte. Par exemple

```
SI    le repas est chaud
ALORS je dois attendre un peu avant de manger
```

Dans cet exemple, la règle représente un test sur la chaleur d'un repas, qui peut être interprété différemment en fonction de ce qu'une personne considère comme chaud, et conclut qu'il faut attendre un peu, de même le terme «peu» peut avoir différentes interprétations.

– **les règles métier.** Les règles métier sont des déclarations qui définissent un aspect métier. Elles permettent de décrire précisément les critères de décision et les actions à mener pour un processus donné. Ces règles, qu'elles soient formalisées ou non, existent dans chaque cœur de métier. Elles capitalisent la connaissance, le savoir-faire d'une entreprise et traduisent sa stratégie (Diouf *et al.*, 2007).

Les règles métier peuvent être vues de deux points de vues différents. D'un point de vue système d'information, «les règles métier sont des formulations qui définissent ou contraignent certains aspects d'un métier. Elles permettent de structurer un métier (politique, savoir-faire), de le contrôler ou d'en influencer le comportement.» (Ross, 2003). D'un point de vue métier, «une règle métier est une directive qui est censée influencer ou guider le comportement d'un métier dans le but de mettre en œuvre une politique métier qui est formulée en vue d'une réponse à une opportunité ou un risque» (Hay & Healy, 2000). Une fois définies, ces règles peuvent être interprétées par les moteurs de règles.

Le principe des règles métier est de séparer la logique métier (le comportement) et la logique système d'une application (le comment), ainsi le comportement d'un système d'information pourra être modifié par l'expert métier sans avoir recours au service informatique.

Avec l'approche par règles métier, dans chaque domaine, les applications sont dirigées et gérées par les experts métier du domaine : pour une application financière c'est l'expert financier, pour une application bancaire c'est le banquier, etc. et pour une application de validation ergonomique,

c'est l'expert ergonome qui modifie les règles ergonomiques (Diouf *et al.*, 2006).

2.2.3 Les ontologies

Le terme ontologie est issu de la philosophie au temps d'Aristote [1]. L'ontologie est une branche de la philosophie qui a pour objet l'étude de l'être en tant qu'être, c'est-à-dire l'étude des propriétés les plus générales [2] telles que l'existence, la possibilité, la durée, le devenir (Wikipedia, 2012b).

Par analogie, le terme est repris en informatique particulièrement en Ingénierie des Connaissances dans le but de modéliser des connaissances pour les systèmes à base de connaissances (SBC) (Charlet *et al.*, 2004). La première définition des ontologies informatique a été donnée par (Neches *et al.*, 1991) ; «*Une ontologie définie les termes et les relations de base comportant le vocabulaire d'un domaine aussi bien que les règles pour combiner des termes et les relations afin de définir des extensions du vocabulaire*». Dans (Paolo *et al.*, 2003), les ontologies sont définies comme étant «*des modèles partagés d'un domaine encodant une vue qui est commune à un ensemble de différentes parties*». Néanmoins, la définition la plus citée est celle donnée par Gruber en 1993 (Gruber, 1993), «*Une ontologie est une spécification explicite et formelle d'une conceptualisation d'un domaine de connaissance*». La conceptualisation étant une modélisation abstraite des concepts, permettant de décrire un domaine de connaissance, ainsi que leurs propriétés, organisés en hiérarchie. La spécification est la représentation, dans un langage formel muni d'une syntaxe et d'une sémantique, du modèle obtenu lors de la conceptualisation.

2.2.3.1 Les connaissances représentées dans les ontologies

Une ontologie représente généralement les concepts d'un domaine ainsi que les propriétés et les relations entre ces concepts. (Charlet *et al.*, 2004) définit quatre caractéristiques pour déterminer qu'est ce qui peut être représenté dans une ontologie.

le type de l'ontologie – le type de l'ontologie dépend de l'ensemble des objets conceptualisés et manipulés au sein du SBC. Ainsi, 4 principales

1. http ://fr.wikipedia.org/wiki/Aristote
2. http ://www.cnrtl.fr/lexicographie/ontologie

types d'ontologie sont distingués : (1) l'ontologie de domaine représente les concepts et propriétés d'un domaine particulier, (2) l'ontologie générique repère et organise les concepts les plus abstraits du domaine, (3) l'ontologie d'une méthode de résolution de problème où le rôle joué par chaque concept dans le raisonnement est rendu explicite et (4) l'ontologie de représentation qui repère et organise les primitives de la théorie logique permettant de représenter l'ontologie ;

les propriétés – ce sont les caractéristiques rattachées aux concepts représentés par l'ontologie ;

la relation *is-a* – la relation de subsomption qui permet de structurer l'arborescence de l'ontologie et de représenter la notion d'héritage de propriétés. Elle doit être complétée par d'autres relations pour exprimer la sémantique du domaine ;

les autres relations – ce sont les relations qui relient les concepts entre eux afin de former un modèle conceptuel plus complexe que celui formé par la relation *is-a*. Si la connaissance représentée correspond à un concept dans le monde modélisé, celui-ci est dit *défini*, à l'opposé des concepts insérés dans l'arborescence de l'ontologie qui sont dits *primitifs*.

2.2.3.2 Pourquoi utiliser les ontologies ?

Les ontologies fournissent des fonctionnalités utiles pour la représentation des connaissances. Dans cette section nous présentons quatre fonctionnalités les plus utiles en se fondant sur (Gasevic *et al.*, 2000).

Définition du vocabulaire – les ontologies permettent de définir un modèle de connaissance utilisant la terminologie du domaine modélisé et fournissent la possibilité de présenter les relations entre les différents termes du vocabulaire. Elles permettent aussi de définir des règles pour combiner les différents termes afin d'étendre le vocabulaire. Il est important de signaler que ce n'est pas le vocabulaire lui même qui permet de définir une ontologie mais la conceptualisation dénotée par les différents termes et leurs relations.

Définition des taxonomies – Chaque ontologie fournit une taxonomie dans un format qui peut être exploité par une machine. Cependant, une ontologie est plus riche qu'une taxonomie puisqu'elle permet de définir la

spécification complète d'un domaine. Le vocabulaire et la taxonomie de l'ontologie fournissent ensemble une conceptualisation permettant l'analyse et la recherche d'information. Il est à noter que les taxonomies ne fournissent pas nécessairement une spécialisation complète des hiérarchies alors qu'avec les ontologies la spécialisation est formellement spécifiée et assure ainsi la cohérence du raisonnement.

Partage et réutilisation des connaissances – L'objectif principal des ontologies est le partage et la réutilisation des connaissances par différentes applications. Les ontologies fournissent des descriptions de concepts et les relations entre les concepts d'un domaine qui sont partagées et utilisées par des agents intelligents et des applications. Le partage et la réutilisation des connaissances exigent une interprétation commune de la sémantique des termes du domaine, la compatibilité des différentes modélisations utilisées par les différents agents et applications. Tout ceci est possible grâce aux ontologies partagées qui permettent de créer un modèle de connaissances homogènes modélisant des connaissances hétérogènes. Néanmoins, en pratique, le partage et la réutilisation des connaissances n'est pas chose facile. Même s'il existe différentes ontologies dans différents domaines et même s'il existe beaucoup de langages de représentation des ontologies, ces langages ne sont pas supportés par toutes les applications et les systèmes à base de connaissances.

Spécification du contenu – Développer une ontologie ne consiste pas uniquement à identifier les concepts d'un domaine et les relations entre eux et de les définir sous forme d'une taxonomie. Il faut aussi les représenter dans un langage formel (voir Section 2.3.3) de manière à ce qu'elles puissent être traitées par des machines. Les ontologies bien développées permettent l'interopérabilité entre différentes applications et facilitent la gestion de la cohérence (*e.g.* vérification des types de propriétés et des restrictions). Il est nécessaire d'effectuer une analyse ontologique du domaine à modéliser avant de commencer le développement d'une ontologie. Une analyse ontologique consiste à déterminer les différentes entités du domaine, leur taxonomie et leurs caractéristiques. Cette analyse est nécessaire pour avoir un modèle complet qui regroupe l'ensemble des connaissances du domaine.

2.2.4 Synthèse

Les ontologies sont de plus en plus utilisées pour la représentation des connaissances, vu la flexibilité et l'interopérabilité qu'elles fournissent. D'un autre côté, les règles et principalement les règles métier permettent aux utilisateurs métier d'automatiser les décisions et les actions à mener pour un processus donné et sont généralement écrites dans un langage naturel contrôlé. Pour cela, nous nous intéressons aux règles métier qui sont écrites à partir d'ontologies de domaine.

2.3 Les langages de représentation des connaissances

Dans la section précédente, nous avons présenté quelques techniques utilisées pour la représentation des connaissances dans les systèmes à base de connaissances. Pour que ces représentations puissent être traitées et exploitées en outre par des algorithmes d'inférences, il est nécessaire qu'elles soient représentées par des langages formels définis par une syntaxe (*i.e.* ensemble des éléments du langage), d'une sémantique (*i.e.* comment interpréter les expressions du langage) et d'un raisonnement (*i.e.* comment inférer de nouveaux faits à partir de ceux qui existent). Un langage de représentation des connaissances doit permettre de représenter les différentes entités, actions, événements, procédures... d'un domaine donné. Néanmoins, la majorité des langages qui existent dans la littérature ne permettent pas de représenter toutes ces connaissances et, de plus, ils ne permettent de formaliser qu'un certain nombre des techniques de représentation précédemment citées. En d'autres termes, chaque technique de représentation des connaissances ne peut être formalisée qu'avec certains langages.

Il existe beaucoup de langages de représentation des connaissances dans la littérature (Gasevic *et al.*, 2000). Dans cette section nous nous intéressons principalement aux langages fondés sur la logique qui sont utilisés pour représenter les ontologies et les règles.

2.3.1 Langages de représentation fondés sur la logique

2.3.1.1 Logique propositionnelle

La logique propositionnelle est une théorie logique qui définit les lois formelles du raisonnement (Wikipedia, 2012a). Une proposition est une déclaration logique qui peut être soit vraie soit fausse (*i.e.* tout homme est mortel). En logique des propositions, une variable symbolique est attribuée à chaque proposition (*i.e.* "A = tout_homme_est_mortel") et une valeur de vérité (vrai ou faux) est attribuée à chaque variable. Les propositions peuvent ensuite être reliées via des connecteurs logiques afin de pouvoir former des règles et des propositions plus complexes (voir Table 2.1).

Connecteur	Symbole
AND	$\wedge$
OR	$\vee$
Implies	$\rightarrow$ ou $\Rightarrow$
not	$\neg$

TABLE 2.1 – Les connecteurs logiques.

Par exemple, soit la variable `A` précédemment définie et les variables,

`"B = Socrate_est_un_homme"` et `"C = Socrate_est_mortel"`

La représentation formelle et symbolique de la règle

```
SI    tout homme est mortel
ET    Socrate est un homme
ALORS Socrate est mortel
```

est comme suit : `A` $\wedge$ `B` $\rightarrow$ `C`

La logique propositionnelle permet d'effectuer un raisonnement formel, et ce au moyen de règles, afin de calculer la valeur de vérité d'un ensemble de propositions. Le calcul de la valeur de vérité d'un ensemble de propositions se fait en fonction de la valeur de vérité de chaque proposition et de la table de vérité des opérateurs logiques. Néanmoins, cette logique présente quelques inconvénients vu qu'il est parfois difficile d'assigner une seule variable à une proposition et que la valeur de vérité d'une proposition peut souvent être

subjective et sujette à interprétation, et son calcul est généralement de complexité exponentielle. La logique du premier ordre offre une approche plus fine qui permet de représenter et de raisonner sur chaque partie d'une proposition.

2.3.1.2 Logique du premier ordre

La logique du premier ordre appelée aussi la logique des prédicats est une extension de la logique propositionnelle. Elle introduit le quantificateur universel, $\forall$, et le quantificateur existentiel, $\exists$. Elle introduit aussi la notion de symboles pour représenter des connaissances et les opérateurs logiques. Ces symboles peuvent être des *constantes*, des *variables*, des *prédicats* ou des *fonctions*. Généralement, les *constantes* sont des symboles qui commencent par une lettre minuscule, les *variables* commencent par une lettre majuscule. Les propositions sont représentées aux moyens d'*arguments* (*i.e.* les objets de la proposition) et de *prédicats* qui sont des assertions concernant les objets. Par exemple, en logique de premier ordre les propositions «Socrate est un homme» et «Platon est un disciple de Socrate» sont respectivement représentées par le prédicats :

`Homme(socrate)` et `discipleDe(socrate, platon)`

Ces prédicats peuvent aussi être représentés avec des *variables* pour représenter des propositions plus général qui peuvent ensuite être instanciées :

`Homme(X)` et `discipleDe(X, Y)`

Ce prédicat peut être instancié par des propositions telle que «Socrate est un homme» (`Homme(socrate)`) ou «Platon est un homme» (`Homme(platon)`).

Les *fonctions* permettent de relier les entités d'un ensemble à un élément unique d'un autre ensemble.

Il est aussi possible d'utiliser les fonctions, les opérateurs, les quantificateurs pour définir des règles. Par example : $\forall$ `X Homme(X)` $\rightarrow$ `Mortel(X)`, ainsi nous pouvons déduire que $\forall$ `Homme(scorate)` $\rightarrow$ `Mortel(socrate)`

La représentation des connaissances en logique du premier ordre est effectuée à l'aide de prédicats et de règles. De même pour le raisonnement qui est effectué à l'aide de prédicats et de règles d'inférences. Beaucoup de langages utilisés en intelligence artificielle sont fondés sur la logique des prédicats (*e.g.* Prolog). Le pouvoir d'expressivité de la logique de premier ordre est très

puissant ; elle peut être utilisée comme un méta-langage pour définir d'autres types de logique (Gasevic *et al.*, 2000).

2.3.1.3 Logique de description

La logique de description est reconnue comme étant un fragment décidable de la logique de premier ordre. Le développement d'une base de connaissance en utilisant la logique de description revient à d'abord définir les entités du domaine à modéliser en utilisant la *terminologie* (*i.e.* le vocabulaire) du domaine, ce qui représente la *TBox* (Terminological Box), et ensuite introduire les assertions, la *ABox* (Assertional Box), qui représentent les individus des concepts définis dans la *TBox*. Les entités du domaine consistent en l'ensemble des concepts, qui dénotent des *individus*, et des rôles qui représentent des relations binaires entre les *individus*. Ceci étant, au niveau terminologique, il existe deux types d'entités :

- Les concepts qui représentent un ensemble d'individus. Deux types de concepts peuvent être distingués : les concepts *atomiques* ou *primitifs* qui sont représentés par le nom du concept et les concepts *complexes* ou *définis* qui sont représentés par une expression constituée de la connexion, via des connecteurs logiques, d'un ensemble de concepts primitifs et de rôles.
- Les rôles : représentent des relations entre les concepts. Des restrictions et des cardinalités peuvent être affectées aux rôles.

Par exemple, soit les concepts atomiques `Person` et `Femal` et le rôle `hasChild`, une partie de la *TBox* représentant les familles sera comme suit (adaptation à partir de (Gasevic *et al.*, 2000)) :

$$\texttt{Woman} \equiv \texttt{Person} \sqcap \texttt{Female}$$
$$\texttt{Man} \equiv \texttt{Person} \sqcap \neg\ \texttt{Woman}$$
$$\texttt{Parent} \equiv \texttt{Mother} \sqcup \texttt{Father}$$
$$\texttt{hasChild (Person, Person)}$$
$$\texttt{Mother} \equiv \texttt{Woman} \sqcap \exists\ \texttt{hasChild.Person}$$
$$\texttt{MotherWithManyChildren} \equiv \texttt{Mother} \sqcap \geq 3\ \texttt{hasChild.Preson}$$

Généralement, la *TBox* définit les relations entre les individus introduits dans la *ABox* et leurs propriétés. Ainsi la *ABox* décrit un état spécifique du

monde en utilisant les concepts et rôles de la *TBox*. La *ABox* correspondant à la *TBox* représentée ci dessus peut être comme suit :

```
Woman (Olive)
Man (Popey)
hasChild(Olive, Mimosa)
```

Les systèmes à base de connaissances composés d'une *TBox* et d'une *ABox* permettent d'effectuer des raisonnements et d'inférer de nouvelles connaissances. Deux types de raisonnements sur les assertions de la *ABox* sont importants ; (1) déterminer si les assertions impliquent qu'un individu est une instance d'un concept donné, et (2) si l'ensemble des assertions est cohérent (*i.e.* conforme à un modèle).

2.3.2 Langages de représentation fondés sur les règles

2.3.2.1 Rule Interchange Format

Rule Interchange Format (RIF), est un standard du W3C [3] permettant de fournir l'interopérabilité entre les langages de règles en général et ceux utilisés dans le web sémantique en particulier. L'objectif de RIF n'est pas de fournir un langage de règle unique car il est quasiment impossible de trouver un seul langage de règles qui permet de satisfaire tous les besoin de représentation des connaissances à base de règles. Le but de RIF est de permettre la traduction de règles d'un langage de règles à un autre et ainsi de pouvoir les transférer d'un système de règles à un autre .

Le formalisme de RIF a une architecture en couches qui s'organise autour des dialectes qui permettent d'étendre son noyau central (RIF Core). Un dialecte est un langage de règles qui a une syntaxe et une sémantique bien définie. Les dialectes sont proposés pour représenter différents type de règles utilisés dans les systèmes à base de règles qui, selon le groupe de travail du W3C, sont classés en trois catégories : les systèmes fondés sur la logique de premier ordre, les systèmes fondés sur la logique de programmation et les systèmes fondés sur les règles de production.

Deux dialectes de RIF ont été développés :

1. Basic Logic Dialect (RIF-BLD) – RIF-DLB est un format d'échange des

3. http ://www.w3.org/TR/2010/NOTE-rif-overview-20100622/

règles logique. Ce dialecte correspond à une extension syntaxique et sémantique de la logique de Horn. Ces extensions consistent en l'ajout de caractéristiques permettant de représenter des objets, des frames, des *Internationalized Resource Identifiers* (IRI) pour identifier les concepts et des type de données de XML schema[4]. Ci-dessous un exemple d'utilisation de RIF-BLD :

soit la règle suivante permettant d'inférer les relations *achète* à partir des relations *vend* représentées par des faits de la base de connaissances :

Un acheteur achète un article si le vendeur vend l'article à l'acheteur.
John vend LeRif à Mary.

Ainsi, le fait que Marie achète le livre à John est logiquement inféré à partir de ces deux prémisses, qui sont représentées en RIF-BLD comme suit :

```
Document(
  Base(<http://example.com/people#>)
  Prefix(cpt <http://example.com/concepts#>)
  Prefix(bks <http://example.com/books#>)

  Group
  (
    Forall ?Buyer ?Item ?Seller (
        cpt:buy(?Buyer ?Item ?Seller) :- cpt:sell(?Seller ?Item ?Buyer)
    )
    cpt:sell(<John> bks:LeRif "Mary"^^rif:iri)
  )
)
```

2. Production Rule Dialect (RIF-PRD) – RIF-PRD est un format d'échange des règles de production. Un règle de production est représentée par une partie condition, comme celle représentée par les règles logiques, et une partie action qui permet de spécifier une ou plusieurs actions quand la partie condition est vérifiée. Une action consiste en, soit l'ajout d'un fait, soit la modification d'un fait ou soit la suppression d'un fait. Ci-dessous, quelques exemples de règles de production :

   A customer becomes a "Gold" customer when his cumulative purchases

4. http ://www.w3.org/TR/2009/CR-xmlschema11-2-20090430/

during the current year reach $5000.

Le représentation de cette règle en RIF-PRD peut être comme suit :

```
Prefix(ex <http://example.com/2008/prd1#>)
Forall ?customer ?purchasesYTD (
 If   And( ?customer#ex:Customer
           ?customer[ex:purchasesYTD->?purchasesYTD]
           External(pred:numeric-greater-than(?purchasesYTD 5000)) )
 Then Do( Modify(?customer[ex:status->"Gold"]) ) )
```

2.3.2.2 Rule Markup Language

L'initiative RuleML vise à développer un langage de règles fondé sur une syntaxe XML/RDF afin d'assurer l'interopérabilité des règles entre différents systèmes à base de règles. RuleML est fondé sur la logique de premier ordre [5] et sa spécification consiste en une famille modulaire de sous-langages du web. Chaque sous-langage est défini en XML Schema et possède une URI ce qui permet de déterminer la spécification requise pour un problème donné. La structure en famille modulaire offre une hiérarchie d'inclusion expressive pour les sous-langages (Boley, 2006).

2.3.2.3 Semantics of Business Vocabulary and Rules

SBVR est un standard de l'OMG [6] qui regroupe les aspects de modélisation à base de règles et d'ontologies. Il permet de structurer le vocabulaire métier utilisé par les experts métier afin de formuler un ensemble de règles métier, dans un langage naturel, et de définir une sémantique formelle de cette structure afin d'assurer une interprétation cohérente du vocabulaire et des règles. Les principaux composants d'un méta-modèle en SBVR sont le vocabulaire métier et les règles métier (Linehan, 2008).

Le vocabulaire métier – un vocabulaire métier SBVR définit principalement les noms des concepts (*e.g.* les concepts OWL), les *facts types* (*i.e.* les propriétés OWL), les instances des noms des concepts et des *fact types*. SBVR

5. http ://www.w3.org/Submission/FOL-RuleML/
6. http ://www.omg.org/spec/SBVR/1.0

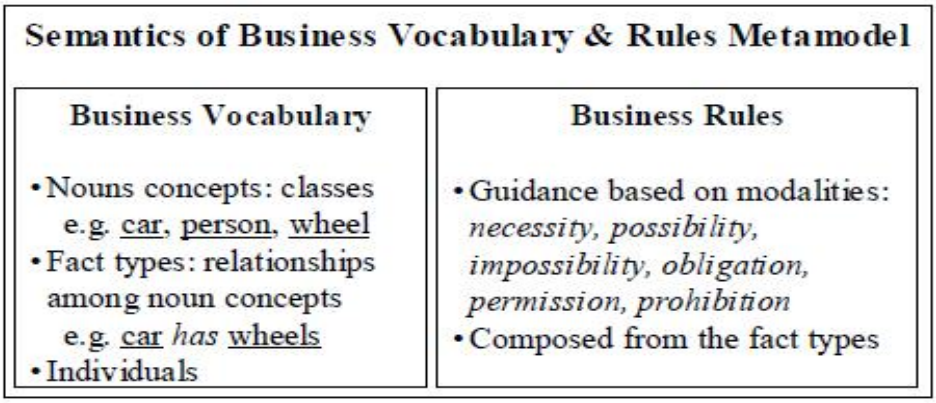

FIGURE 2.3 – Les composants d"un méta-modèle SBVR (Linehan, 2008).

permet de définir une hiérarchie des noms de concepts ce qui permet d'effectuer un raisonnement de subsomption. Les *facts types* identifient des relations entre un ou plusieurs rôles, ainsi nous pouvons avoir des *fact types* unaires, binaires, ou plus, mais chaque *fact type* a un nombre fixe de rôles. Un rôle représente les entités intervenant dans une relation.

Les termes d'un vocabulaire peuvent être définis par des règles de dérivation. Par exemple, «le grand-père d'une personne» peut être défini par «le parent du parent d'une personne» ce qui permet de spécifier formellement la dérivation de concepts à partir d'autres concepts. Il est aussi possible de définir des contraintes sur des concepts ainsi que des synonymes d'un concept.

Les règles métier – Les règles SBVR peuvent être soit des règles structurelles soit des règles de comportement. Les règles structurelles définissent les caractéristiques d'un modèle et doivent toujours être vérifiées [7].

Exemple : **It is possible that** an order has **more than** one line item.

Les règles de comportement modélisent les attentes d'une personne ou d'un système qui ne sont pas toujours vérifiées.

Exemple : **Each** order **must** be paid within 24 hours

7. Le texte écrit en gras représente les mots clés de SBVR

2.3.3 Représentation des ontologies

2.3.3.1 Resource Description Framework

RDF est un modèle de représentation des données, définit par le W3C, permettant de décrire des ressources indépendamment des applications. Les connaissances décrites en RDF peuvent être manipulées en dehors de l'environnement dans lequel elles ont été créées, ce qui permet de combiner les données de plusieurs applications pour générer de nouvelles informations. Les connaissances en RDF sont représentées sous forme de triplets <sujet, prédicat, objet>. Le sujet étant la ressource à décrire, chaque ressource possède un identifiant unique Universal Ressource Identifier (URI). Le prédicat représente la relation entre le sujet et l'objet et est aussi une ressource. L'objet est la valeur de ce prédicat et peut être soit une ressource soit un littéral (Manola & Miller, 2004). Une description en RDF peut être représentée sous forme d'un graphe orienté étiqueté (*e.g.* réseau sémantique) où le sujet et l'objet représentent les sommets de ce graphe, et chaque triplet est représenté par un arc dont l'origine est son sujet et l'extrémité est son objet. Un exemple d'un graphe RDF est décrit ci-dessous (voir figure 2.4).

FIGURE 2.4 – Exemple d'un graphe RDF.

Ce graphe permet de représenter que Amina Chniti, représentée par l'URI `http://dblp.L3S.de/Authors/Amina_Chniti` est l'auteur d'une référence bibliographique représentée par l'URI `http://dblp.l3s.de/d2r/resource/publications/conf/ruleml/ChnitiDAC10`. Le triplet représentant le graphe décrit dans la figure 2.4 est

```
<http://dblp.L3S.de/Authors/Amina_Chniti, estAuteurDe,
 http://dblp.l3s.de/d2r/resource/publications/conf/ruleml/ChnitiDAC10>
```

RDF fournit un moyen de modéliser des individus mais il ne permet pas de définir la sémantique d'un domaine d'application. RDF est utilisé pour décrire

les individus d'une ontologie, alors que pour décrire des ontologies, il faudra un langage plus expressif, RDFS.

2.3.3.2 Resource Description Framework Schema

RDFS est une extension de RDF. Il définit un vocabulaire de méta-données pour la description d'une ressource. RDFS permet de définir des groupes de ressources similaires (*i.e.* les classes), une hiérarchie de spécialisation sur les classes, les relations entre les ressources (*i.e.* les propriétés), une hiérarchie de spécialisation sur les propriétés, des instances des classes (voir Figure 2.5). Les descriptions de vocabulaire de RDFS sont écrites en RDF en utilisant les termes (primitives) décrits dans la spécification du schéma RDF (Brickley & Guha, 2004).

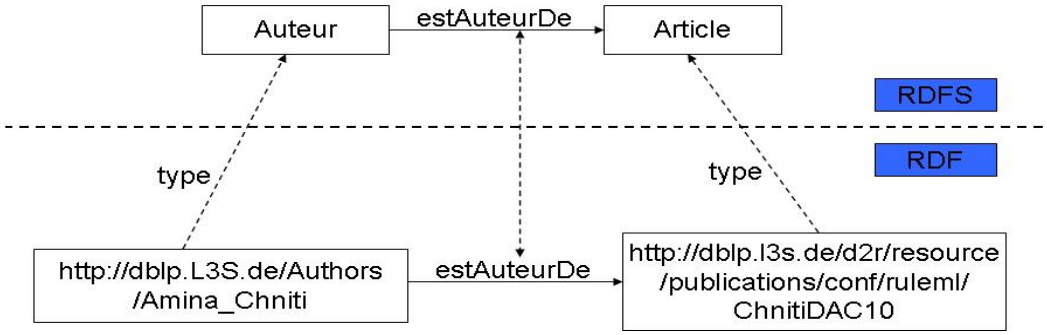

FIGURE 2.5 – Exemple d'une modélisation RDFS.

Ceci étant, malgré les avantages de RDFS, ce dernier représente tout de même un pouvoir d'expressivité limité. Ainsi RDFS ne permet pas d'exprimer la disjonction, l'union ou l'intersection. Aussi certaines caractéristiques des propriétés ne sont pas prises en charge par RDFS telles que la transitivité, l'unicité ou les propriétés inverses et il ne permet pas d'exprimer des restrictions de valeur ou cardinalité. Et afin de pouvoir exprimer ces caractéristiques dans une ontologie, il faut alors utiliser OWL.

2.3.3.3 Web Ontology Language

OWL, est un langage défini par le W3C qui permet la spécification d'ontologies. Tout comme RDF et RDFS, la syntaxe de OWL est fondée sur XML et

hérite donc de son universalité syntaxique. En plus de la possibilité de décrire des classes et des propriétés, OWL permet aussi d'exprimer les relations entre les classes, la cardinalité et les caractéristiques des propriétés (symétrie, transitivité, disjonction,. . .). Dans ce qui suit, nous commençons par introduire la première version de ce langage, OWL 1, les limites de cette première version et enfin la deuxième version OWL2.

OWL 1 Le langage d'ontologie OWL possède trois sous langage offrant des capacités d'expression croissantes, OWL Lite, OWL DL et OWL Full.

- **OWL Lite :** C'est une version de OWL assez réduite destinée aux utilisateurs ayant besoin d'une hiérarchie de concepts et d'un mécanisme de contraintes simples. Par exemple, bien qu'il supporte les contraintes de cardinalité, il permet uniquement les valeurs de cardinalité 0 ou 1.
- **OWL DL :** Plus complexe que OWL lite, OWL DL permet de fournir une expressivité maximum tout en garantissant la complétude des raisonnements (toutes les inférences sont calculables) et la décidabilité (tous les calculs doivent finir dans un temps fini). OWL DL contient tout les constructeurs du langage OWL auxquels il ajoute des restrictions sur le type des ressources décrites, ainsi une classe ne peut pas être un individu ou une propriété et une propriété ne peut pas être une classe ou un individu.
- **OWL Full :** c'est la version la plus complexe de OWL. Il permet d'avoir un très haut niveau d'expressivité et la liberté syntaxique de RDF, mais ne garantie pas la calculabilité. Par exemple, une classe peut être traitée comme une collection d'individus et en même temps peut être vue comme un seul individu. OWL Full permet aussi à une ontologie d'augmenter le sens du vocabulaire prédéfini (RDF et OWL).

Une ontologie OWL se compose principalement des entités suivantes :

les concepts– ils regroupent des ressources ayant des caractéristiques communes. Toute classe OWL est associée à un ensemble d'individus appelé l'extension de la classe ;

les propriétés – elles relient l'ensemble des classes deux à deux ;

les individus– ils représentent sont les instances des classes.

Le vocabulaire OWL utilisé pour décrire ces ressources est principalement basé sur RDF. Les préfixes `rdf:` et `rdfs:` sont employés pour écrire une ontologies OWL. Ces préfixes sont utilisés lorsque les termes sont déjà présents dans RDF sinon les termes sont introduits par le préfixe `owl:`.

Le pouvoir d'expressivité fournit par OWL est dû à la diversité de ses constructeurs qui permettent de définir des expressions de classes très complexes (voir table 2.2) et qui permettent de raisonner et d'inférer de nouvelles connaissances grâces aux différents axiomes (voir Table 2.3).

Constructeur	**Exemple**
`owl:intersectionOf`	Man $\equiv$ Human $\sqcap$ Male
`owl:unionOf`	Human $\equiv$ Male $\sqcup$ Female
`owl:complementOf`	$\neg$Male
`owl:oneOf`	Days $\equiv$ Monday $\sqcup$ Tuesday $\sqcup \ldots \sqcup$ Saturday
`owl:allValuesFrom`	$\forall$ hasFather.Man
`owl:someValuesFrom`	$\exists$ hasDrivingLicense. CarDrivingLicense
`owl:maxCardinality`	$\leqslant$ 4 hasWheel
`owl:minCardinality`	$\geqslant$ 1 hasChild

TABLE 2.2 – Les expressions de classes OWL.

Constructeur	**Exemple**
`owl:subClassOf`	Man $\sqsubseteq$ Human
`owl:equivalentClass`	Man $\equiv$ Human $\sqcap$ Male
`owl:disjointWith`	Male $\sqsubseteq \neg$ Female
`rdfs:domain`, `rdfs:range`	hasFather(Child, Father)
`rdfs:subProperty`	hasDauther $\sqsubseteq$ hasChild
`owl:equivalentProperty`	cost $\equiv$ price
`owl:inverseOf`	hasChild $\equiv$ hasParent^{-}
`owl:transitiveProperty`	ancestor^{+} $\sqsubseteq$ ancestor
`owl:symmetricProperty`	hasFriend
`owl:functionalProperty`	$\top \sqsubseteq \leqslant 1$ hasMother
`owl:inverseFunctionalProperty`	$\top \sqsubseteq \leqslant 1$ hasISBN $^{-}$

TABLE 2.3 – Les axiomes OWL.

Limites du pouvoir d'expressivité de OWL 1 Malgré le nombre de constructeurs offert par OWL 1 pour la modélisation des connaissances, il existe quelques limitations de son pouvoir d'expressivité et de raisonnement de cette version (Grau *et al.*, 2008). Parmi ces limitations nous pouvons citer :

Restriction des cardinalités « qualifié » : la restriction existentielle fournit par OWL 1 DL est une restriction utilisée pour la définition des classes ; par exemple, il est possible de définir un concept correspondant à des personnes ayant au minimum un enfant qui est mâle. Par contre, les restrictions de cardinalié ne peuvent être définies sur des classes. Ainsi, il est possible dans OWL 1 DL de définir le concept des personnes ayant au minimum 3 enfants mais il n'est pas possible de définir le concept des personnes ayant au minimum 3 enfants qui sont mâles. Pour pouvoir définir cette expression, il faut que la restriction soit établie sur la classe (*i.e.* Mâle) à laquelle les objets qui doivent être comptés appartiennent (*i.e.* Enfant). Cette caractéristique est appelée *qualified cardinality restriction* (QCR).

Expressivité des relations : OWL 1 fournit plusieurs constructeurs pour définir des expressions de classes complexes mais on ne peut pas en dire autant pour les expressions des propriétés. L'absence de cette expressivité au niveau des propriétés a été reconnue comme l'obstacle majeur à l'adoption de OWL 1. Ainsi, OWL 1 ne supporte pas la propagation transitive des rôles, chose qui est nécessaire dans certains domaines tels que la médecine. Par exemple, on veut affirmer qu'une anomalie d'une partie d'une structure anatomique constitue une anomalie de la structure dans son ensemble. Cela permettrait d'inférer de nombreuses conclusions utiles, telles que déduire que d'une fracture du col du fémur est une sorte de fracture du fémur, ou qu'un ulcère situé dans la muqueuse gastrique est une sorte d'ulcère de l'estomac. Certaines ontologies de domaine médicale tel que la SNOMED [8] ou OBO [9] permettent de représenter ce type connaissances malgré que l'ensemble de constructeurs utilisés dans ces modélisations est beaucoup moins important de celui proposé par OWL1.

8. http ://www.ihtsdo.org/snomed-ct/
9. http ://www.obofoundry.org/

Expressivité des types de données : Le pouvoir d'expressivité de OWL 1 pour représenter les instances de classes qui ont des valeurs de type concret (string, integer, float...) est très limité. Il est possible d'exprimer que le nom d'une personne est de type `xsd:int` ou bien que la matricule d'une voiture est de type `xsd:float`. Par contre, il n'est pas possible de restreindre la valeur d'une propriété à un sous ensemble de valeur, par exemple l'âge d'un enfant est inférieur à 18 ou bien la vitesse d'une voiture dans une ville doit être entre 20 et 50. Aussi ; il n'est pas possible de représenter les relations entre les valeurs des propriétés de données sur un objet, par exemple, une table carrée est une table dont la largeur est égale à sa profondeur. Autre limite du même type, il n'est pas possible d'exprimer les relations entre les valeurs des propriétés de données sur différents objets, par exemple les gens qui sont plus âgés que leur patron. Aussi, il n'est pas possible avec OWL 1 de représenter l'agrégation comme par exemple la durée d'un processus est la somme des durées de ses sous-processus.

OWL 2 OWL 2 a été développé pour répondre aux besoins de modélisation qui ne peuvent être exprimés par OWL 1. Pour cela, il propose trois profils qui permettent de répondre aux nécessités de modélisation en fonction des besoins. Ces trois profils sont :

1. OWL 2 EL – basé sur la logique de description EL++, il offre une performance de raisonnement particulièrement pour la classification. Il permet le calcul de toutes les relations de sous-classe existantes dans une ontologie dans un temps minimal ; par exemple, pour la classification des concepts de la SNOMED CT, le temps de calcul est inférieur à 1 minute (Hitzler *et al.*, 2009). Les principales fonctionnalités de modélisation utilisées par ce profil sont la conjonction de classes et la restriction `SomeValuesfrom`. Afin d'assurer la traçabilité, OWL 2 EL ne permet pas l'utilisation de la négation, la restriction sur la cardinalité et la restriction ; `AllValuesFrom`
2. OWL 2 QL – basé sur la famille DL-Lite de la logique de description, il est destinée aux applications qui utilisent de très gros volumes de données et qui nécessitent une performance de raisonnement pour effectuer des requêtes sur ces données. Les requêtes OWL 2 QL peuvent être implémentées en utilisant SQL et la performance du raisonnement

est due principalement à la conjonction de ces requêtes. Afin de s'assurer que chaque ontologie peut être réécrite en une union de requêtes conjonctives, ce profil ne permet d'utiliser la disjonction et la restriction `AllValuesFrom` ;

3. OWL 2 RL – est destinée aux applications qui nécessitent un raisonnement évolutif sans pour autant sacrifier le pouvoir d'expressivité. Il est conçu pour les applications qui utilisent toute l'expressivité de OWL 2 ainsi que pour les applications RDF(S) qui nécessitent l'utilisation de plus d'expressivité. Les raisonneurs OWL 2 RL sont implémentés à base de règles, d'où l'acronyme RL qui reflète le fait que le raisonnement sur ce profil est implémenté par des langages de règles standards.

Il est donc nécessaire de connaitre les besoin de l'application du domaine à modéliser pour savoir quel profil utiliser. Néanmoins, ces profils permettent d'avoir des systèmes d'inférences très puissants ainsi qu'une vérification de la cohérence des ontologies plus poussée.

D'un autre côté, OWL 2 propose de nouveaux constructeurs qui permettent de raffiner la représentation des connaissances d'un domaine. Parmi ces constructeurs, nous pouvons citer : `owl:hasSelf` qui permet de représenter la réflexivité locale (*e.g.* ∃killed.Self), `owl:propertyChainAxiom` qui permet de chainage de propriétés (*e.g.* hasEnemy *circ* hasFriend ⊑ hasEnemy), ainsi que de nouvelles caractéristiques de propriétés tels que la disjonction, l'asymétrie, la réflexivité...

2.3.4 Synthèse

Vu le pouvoir d'expressivité et la capacité de raisonnement offert par OWL, nous nous intéressons principalement sur les ontologies écrites en OWL 1 DL et ce pour le caractère décidable de ce profil et le nombre d'outils qui permettent de raisonner dessus. D'un autre côté, et vu le caractère orienté utilisateur métier de notre travail, nous nous intéressons aux règles métier écrites dans un langage naturel contrôlé de la famille de SBVR .

2.4 Évolution des ontologies

2.4.1 Les activités de changements d'ontologies

Différentes terminologies sont utilisées dans la littérature pour définir les différentes activités de changements d'une ontologie.

Ces terminologies, inspirées de la communauté travaillant sur l'évolution des schémas dans les bases de données (Roddick, 1995), ont été principalement reprit par (Stojanovic, 2004) et (O'Brien & Abidi, 2006). Dans (Stojanovic, 2004), nous pouvons distinguer entre la gestion de l'ontologie et la modification de l'ontologie. Dans (O'Brien & Abidi, 2006), une terminologie est proposée en se basant sur le rôle de l'évolution dans le cycle de vie de l'ontologie, nous pouvons distinguer entre la révision d'ontologie et l'adaptation d'ontologie. Néanmoins, il existe deux termes communs, qui sont le plus souvent utilisés pour parler des changement d'ontologies que sont la révision d'ontologie et la gestion des versions d'ontologie ou le *versioning*.

Dans ce qui suit, nous donnons un bref descriptif des différentes activités de changements précédemment citées.

la gestion d'ontologie – consiste à élaborer un ensemble de méthodes pour gérer les différentes variantes des ontologies en vue de résoudre des tâches différentes. Un système de gestion d'ontologies permet donc de créer, modifier, gérer des versions, faire des requêtes et sauvegarder des ontologies (Luong, 2007) ;

la modification d'ontologie – consiste à effectuer des changements sur une ontologie sans tenir compte de la cohérence (Luong, 2007) ;

l'adaptation d'ontologie – consiste à adapter la conceptualisation d'une ontologie pour une application spécifique en évitant la contradiction avec la version précédente. L'adaptation peut être considérée comme une révision ;

la révision d'ontologie – consiste en l'extension d'une ontologie à travers le raffinement, l'abstraction ou l'ajout de concepts, de relations, ou prédicats, entre autres (Foo, 1995). La révision est effectué lorsque la conceptualisation actuelle d'une ontologie n'est plus adaptée aux caractéristiques du domaine modélisé ;

la gestion des versions d'ontologie ou *versoning* – consiste à appliquer une série de changements à une ontologie constituant alors une nouvelle copie de l'ontologie, distincte et indépendante des versions précédentes. D'un autre côté, une ontologie peut être développée séparément par différentes personnes, ce qui permet d'avoir différentes versions d'une ontologie. Il est ainsi nécessaire de pouvoir identifier les différentes versions afin de pouvoir les différencier (Klein & Fensel, 2001). Néanmoins, (Klein & Fensel, 2001) affirme que tout changement dans la représentation d'une ontologie consiste en une révision et qu'il faut laisser le choix à l'ingénieur d'ontologie pour décider s'il faut créer une nouvelle version.

l'évolution d'ontologies : consiste à adapter l'ontologie aux différents changements du domaine modélisé. Elle prend en compte les nouveaux besoins du domaine et produit de nouvelles versions de l'ontologie tout en assurant sa cohérence. La cohérence de l'ontologie peut être assurée grâce à la traçabilité des changements afin de fournir les correspondances entre les versions (Stojanovic & Motik, 2002) et en contrôlant l'utilisation des instances (Stojanovic *et al.*, 2002a).

2.4.2 Les opérations de changement d'ontologies

Pour déterminer les opérations de changement qui peuvent être appliquées sur les ontologies, il faut tenir compte des impacts de ces changements sur les différents composants d'une ontologie. D'après (Klein, 2004), nous pouvons distinguer trois types d'opérations de changement :

- les changements conceptuels : représentent les changements de la conceptualisation de l'ontologie (*e.g.* changer les relations entre les concepts) ;
- les changements de la spécification : représentent les changements de la spécification de la conceptualisation (*e.g.* ajouter une nouvelle propriété) ;
- les changements de la représentation : représentent les changements de la représentation de la spécification (*e.g.* changer le langage de la représentation de l'ontologie).

(Noy & Klein, 2004) définissent un ensemble d'opérations de changement en se fondant sur l'impact d'un changement sur la préservation des individus de l'ontologie (voir Table 2.4). Plus précisément, ils tiennent compte de la

possibilité d'accéder aux individus après l'application du changement. Les impacts de changements sont classifiés en trois catégories :

- les changements qui préservent l'information : aucune instance n'est perdue (représentés par «+» dans la Table 2.4) ;
- les changements qui nécessitent des modifications : il est nécessaire de faire quelques modifications sur les données pour les préserver (représentés par «∼» dans la Table 2.4) ;
- les changements qui causent une perte des données : il n'est pas possible de garantir la préservation de toutes les données (représentés par «-» dans la Table 2.4).

2.4.3 Classification des changements d'ontologies

L'objectif de la classification des changements est de spécifier une taxonomie des changements qui peuvent être appliqués à une ontologie. Les classifications les plus citées dans la littérature sont celle présentées par (Stojanovic, 2004) et (Klein, 2004). Ces deux classifications modélisent des classes de changement en fonction du langage de représentation de l'ontologie. Par exemple, la classification décrite par (Stojanovic, 2004) est proposée pour décrire des classes de changements d'ontologies écrites en KAON [10] alors que celle proposée par (Klein, 2004) décrit des classes de changement sd'ontologies écrites en OWL.

L'ontologie de changement proposée par (Stojanovic, 2004) classifie les changements en trois catégories :

les changement élémentaires – c'est un changement primitif qui impacte une seule entité de l'ontologie (*e.g.* ajouter un concept, supprimer une propriété) ;

les changements composites – c'est un changement qui impacte le voisinage direct d'une entité de l'ontologie (*e.g.* modifier une relation de sous classe) ;

les changements complexes – c'est un changement appliqué d'une manière arbitraire à des entités. Il peut être composé d'au moins deux changements élémentaires ou composites (*e.g.* rattaché un concept à un super-concept qui n'est pas nécessairement dans son voisinage direct).

10. http ://kaon2.semanticweb.org/

Opération de changement	Impact	Commentaire
Créer une classe C	+	Pas de perte d'instance.
Supprimer une classe C	-	Le type des instances de C devient plus général (*i.e.* elles seront affectées à la super-classe de C).
Ajouter une propriété P	+	Pas de perte d'instance.
Supprimer une propriété P	-	La valeur de P dans toutes les instances est nulle.
Attacher une propriété P à une classe C	+	Pas de perte d'instance.
Supprimer une propriété P à une classe C	-	La valeur de la propriété P dans les instances de la classe C devient nulle.
Ajouter une relation de sous-classe entre la super-classe C et la sous-classe D	+	D héritent des propriétés de C. Changement équivalent à l'ajout de propriétés.
Supprimer une relation de sous-classe entre C et D	-	D perd les propriétés héritées de C. La valeur des propriétés héritées devient nulle dans les instances de D.
Re-classifier une instance I entant de classe	+	Pas de perte de données.
Re-classifier une classe C entant qu'instance	-	Le type des instances de C devient moins spécifique.
Déclarer que les classes C et D sont disjointes	-	Les instances de type C et D deviennent invalides.
Définir une propriété comme étant transitive ou symétrique	-	Les valeurs de la propriété P qui ne respectent ces caractéristiques deviennent invalides.
Modifier le domaine d'une propriété P de la classe D vers sa super-classe C	+	D héritera toujours de P.
Modifier le domaine d'une propriété P de la classe C vers sa sous-classe D	-	La valeur de la propriété P dans les instances de la classe C deviennent nulle.
Déplacer une propriété P de la classe C vers une classe R qu'elle référence	∼	Pas de perte de données si les valeurs de P sont déplacées.
Regrouper un ensemble de propriétés dans une seule classe	∼	Pas de perte de données si les valeurs de P sont déplacées.
Modifier la super-classe C de la classe D en un classe qui est plus général	-	D n'a plus les propriétés héritées de C. Les valeurs de P dans les instances de D devient nulle.
Modifier la super-classe C de la classe D en un classe qui est plus spécifique	+	Pas de perte de données. C hérite de nouvelles propriétés.
Généraliser la restriction d'une propriété P	+	Les valeurs de P restent toujours valides
Spécifier la restriction d'une propriété P	-	Les valeurs de P qui ne respectent la restriction deviennent invalides
Fusionner des classes	∼	Pas de perte de données si les valeurs des propriétés sont déplacées
Éclater une classe en un ensemble de classes	∼	Pas de perte de données si les valeurs des propriétés sont déplacées

TABLE 2.4 – Les opérations de changement d'ontologies et leurs impacts sur les individus (Noy & Klein, 2004).

Ceci étant, malgré la définition des trois catégories de changement, l'auteur se limite au développement du niveau le plus élémentaire, qui couvre des changements additifs (*i.e.* ajouter des éléments à l'ontologie) ou des changements soustractifs (*i.e.* supprimer des éléments de l'ontologie) (voir Table 2.5). Ces changements sont applicables à des ontologies KAON.

Élément ontologique	**Ajouter**	**Supprimer**
Concept	Ajouter un concept	Supprimer un concept
Hiérarchie de concept	Ajouter une sous classe	Supprimer une sous classe
Propriété	Ajouter une propriété	Supprimer une propriété
Hiérarchie de propriété	Ajouter une sous propriété	Supprimer une sous propriété
Domaine de propriété	Ajouter un domaine	Supprimer un domaine
Range de propriété	Ajouter un range	Supprimer un range
Propriété transitive	Ajouter une propriété transitive	Supprimer une propriété transitive
Propriété symétrique	Ajouter une propriété symétrique	Supprimer une propriété symétrique
Propriété inverse	Ajouter une propriété inverse	Supprimer une propriété inverse
Cardinalité maximale	Ajouter une cardinalité max	Supprimer une cardinalité max
Cardinalité minimale	Ajouter une cardinalité min	Supprimer une cardinalité min
Instance	Ajouter une instance	Supprimer une instance
InstanceOf	Ajouter une instanceOf	Supprimer une instanceOf
Instance de propriété	Ajouter une instance de propriété	Supprimer une instance de propriété
Modèle OI	Ajouter un modèle OI	Supprimer un modèle OI

TABLE 2.5 – Taxonomie des changements définie dans (Stojanovic, 2004).

L'ontologie de changement proposée par (Klein, 2004), définit des changements d'ontologies écrites en OWL. Cette définition, fondée sur celle donnée par (Stojanovic, 2004) propose d'autres types de changements tels que `modify` ainsi que `set` et `unset` pour les caractéristiques des propriétés (voir Table 2.6).

Élément ontologique	Opération de changement
Opérations sur les concepts	
Concept	add, remove
Cardinality min/max	add, remove, modify
super class relation	add, remove
disjoint class	add, remove
equivalent class	add, remove
Opérations sur les propriétés	
property	add, remove
domain, range	add, remove
equivalent property	add, remove
inverse property	add, remove
symmetric property	set, unset
transitive property	set, unset
fuctionnal property	set, unset
inverse-functionnal	set, unset

TABLE 2.6 – Taxonomie des changements définie dans (Klein, 2004).

2.4.4 Représentation des changements d'ontologies

Une des questions les plus importantes pour la gestion des changements d'ontologies est la représentation des changements. Le plus simple serait de garder la trace des changements dans un journal des modifications contenant la séquence exacte des changements appliqués à une ontologie. (Klein, 2004) propose un langage basé sur RDF pour spécifier des opérations de changement. Selon lui, la spécification d'un changement doit contenir les informations suivantes :

des meta-données descriptives – c'est un ensemble d'informations sur quand et qui a appliqué le changement et aussi sur quelle version de l'ontologie le changement a été appliqué ;

un ensemble minimal des transformations – c'est un descriptif complet de l'ensemble des changements appliqués ;

des relations conceptuelles – c'est une spécification des relations entre les concepts des différentes versions de l'ontologie. Cela facilite l'accès aux données par l'amélioration de l'interprétation et de l'interrogation des

sources de données qui ont été décrites dans les différentes versions d'ontologies ;

des changements complexes – ils consistent en une description de haut niveau des changements. L'ensemble des changements complexes ainsi que l'ensemble des transformations peuvent être utilisés pour générer des scripts pour la transformation des données. De plus, les changements complexes permettent de déterminer plus en détail l'impact des changements sur l'accessibilité aux données et sur le résultat des requêtes spécifiques ;

une explication/justification du changement – elle décrit l'intention derrière ce changement. Cette description indique si le changement est une correction d'une erreur, une conceptualisation plus précise ou une mise à jour du domaine modélisé.

Dans (Plessers & Troyer, 2005) les auteurs proposent un langage de définition des changements. Ils définissent un changement en fonction (1) du moment de son application, (2) d'une propriété *causes* qui permet de relier la nouvelle version de l'ontologie à celle qui l'a «causée», (3) d'une propriété *hasState* qui détermine l'état de la version de l'ontologie (*i.e.* confirmé ou en cours) et s'il existe un ID du concept à modifier.

2.4.5 Approches de gestion des évolutions d'ontologies

Beaucoup de travaux traitant la gestion des évolutions d'ontologies existent dans la littérature. Une analyse détaillée de ces travaux a été effectuée par (Djedidi & Aufaure, 2010) classifiant ces différentes approches selon la problématique traitée pour gérer ces évolutions. Certaines approches traitent la problématique de :

- la gestion des évolutions en se fondant sur l'identification des besoins de changements (Stojanovic *et al.*, 2003b; Cimiano & Volker, 2005; Bloehdorn *et al.*, 2006),
- la détection des changements et sauvegarde des versions (Klein *et al.*, 2002a; Noy & Klein, 2004; Plessers & Troyer, 2005; Eder & Wiggisser, 2007) ;
- la spécification formelle des changements (Stojanovic *et al.*, 2002b; Klein, 2004; Plessers *et al.*, 2007; Djedidi & Aufaure, 2010) ;

- l' implémentation des changements (Stojanovic *et al.*, 2003a; Stojanovic, 2004; Flouris, 2006) ;
- la maintenance de la cohérence (Stojanovic, 2004; Haase & Stojanovic, 2005; Plessers & De Troyer, 2006; Djedidi & Aufaure, 2010) ;
- la gestion des versions d'ontologies (Klein & Fensel, 2001; Klein *et al.*, 2002a; Klein, 2004).

Dans ce qui suit, nous nous intéressons principalement à deux approches, BOEMIE et ONTO-EVO^{A}L qui sont des approches de gestion d'évolution de la cohérence d'ontologies, développées à base de patrons.

Approche BOEMIE

L'approche BOEMIE (*Bootstrapping Ontology Evolution with Multimedia Information Extraction*) est une approche à base de patrons de peuplement et d'enrichissement d'ontologies,. Elle a pour objectif d'automatiser le processus d'acquisition des connaissances à partir de contenus multimédia pour l'évolution d'ontologies multimédia. Les patrons sont utilisés pour spécifier les connaissances extraites sous forme de $ABox$ et pour déterminer l'opération d'évolution à effectuer : peuplement (*i.e.* ajout de nouvelles instances) ou enrichissement (extension par de nouveaux concepts ou relations). Le peuplement d'ontologie consiste à identifier les individus se référant au même concept en se fondant sur des techniques de clustering (Castano *et al.*, 2007). Les patrons de peuplement spécifient les activités de changement sous forme de mise en correspondance des individus. L'enrichissement d'ontologie est appliqué pour acquérir les connaissances manquantes lorsqu'il n'y a pas de concept dans l'ontologie auxquels on peut rattacher les individus. Les patrons d'enrichissement spécifient les activités de changement sous forme d'apprentissage de nouveaux concepts ou sous forme d'enrichissement des concepts existants.

L'approche de gestion de la cohérence de l'ontologie utilisée dans BOEMIE est plutôt minimaliste et dépend de l'activité d'évolution effectuée (*i.e.* peuplement ou enrichissement). Dans le cas du peuplement de l'ontologie, la vérification de la cohérence est déléguée à un raisonneur standard qui permet d'éliminer les informations redondantes et de vérifier que les individus ne causent pas de contradictions. Dans le cas de l'enrichissement d'ontologie, la gestion de la cohérence se limite à détecter les incohérences causées par les

modifications.

Approche ONTO-EVO^{A}L

ONTO-EVO^{A}L (*Ontology Evolution-Evaluation*) (Djedidi & Aufaure, 2010) est une approche de gestion de l'évolution d'ontologies à base de patrons de gestion de changement (*Change Management Pattern*) (CMP). L'objectif de cette approche est de guider de manière automatique le processus d'application des changements tout en assurant la cohérence de l'ontologie. Les CMPs, catégorisés en trois catégories de patrons ; les patrons de changements, les patrons d'incohérences et les patrons d'alternatives. Ils permettent respectivement de classer les types de changements en se fondant sur le modèle OWL, classer les types d'incohérences logiques en se référant aux contraintes de OWL DL et de ressortir des types d'alternatives de résolution des contraintes. Le processus de gestion des changements est effectué en quatre étapes (Djedidi, 2009) :

1. la spécification du changement – elle consiste à représenter le changement à appliquer dans le modèle OWL. Une fois la représentation effectuée, le changement est classé selon les types prédéfinis par les patrons de changement et enrichi par un ensemble d'information qui permettront d'instancier un patron de changement spécifique.
2. l'analyse du changement – elle consiste à détecter les incohérences causées par l'application du changement. Il est à noter que le changement est appliqué à une version temporaire de l'ontologie. Les incohérences détectées sont classées en fonction des types d'incohérences définies par les patrons d'incohérences.
3. la résolution du changement – cette étape est réalisée en deux sous-étapes : la proposition des résolutions et l'évaluation des résolutions. En fonction des incohérences détectées, les patrons d'alternatives sont instanciés pour proposer des résolutions potentielles. Les alternatives proposées sont considérées comme des changements et sont donc analysées, selon un mécanisme pour vérifier qu'elles sont cohérentes. Seule les alternatives cohérentes sont proposées. L'évaluation des résolutions consiste à évaluer l'impact des alternatives proposées sur la qualité de l'ontologie en considérant les aspects structure et usage de l'ontologie,

et un ensemble de critères et de métriques (Djedidi & Aufaure, 2008). L'alternative préservant le mieux la qualité de l'ontologie peut être appliquée automatiquement.

4. l'application du changement – elle consiste en l'application finale du changement. Si les résolutions proposées préservent la qualité de l'ontologie, alors le changement est validé et appliqué directement.Sinon, si les résolutions ont un impact négatif sur l'ontologie, les résultats des différents étapes du processus sont présentés à l'ingénieur d'ontologie qui décidera du changement à appliquer.

2.5 Modélisation à base de patrons

La modélisation à base de patrons a été adoptée depuis plus d'une dizaine années dans différents domaines tels que l'ingénierie logicielle (Gamma *et al.*, 1995; Buschmann *et al.*, 1996; Kolp *et al.*, 2003), modélisation des données (Hay, 1996), linguistique principalement pour l'extraction de connaissances à partir de texte (Aussenac-Gilles & Jacques, 2006; Chagnoux *et al.*, 2008), et aussi en gestion et représentation des connaissances (Stirna & Persson, 2002; Staab *et al.*, 2001). Ces dernières années, l'utilisation des patrons a même été adoptée pour la conception d'ontologies pour le web, c'est ce qu'on appelle les *Ontology Design Pattern* (ODP) [11].

Il existe différentes catégories de patrons. Ainsi nous pouvons trouver les patrons d'architecture qui modélisent des schémas d'organisation structurelle de logiciels, les *idioms* ou bien les *coding patterns* qui modélisent des solutions liées à un langage particulier, les patrons d'organisation qui représentent des schémas d'organisation de tout ce qui entoure le développement d'un logiciel, les anti-patrons qui modélisent les erreurs détectées pendant la phase de conception et enfin les patrons de conception qui sont utilisés pour modéliser des caractéristiques de conception communes à une application. C'est à cette catégorie de patrons que nous allons nous intéresser.

Dans ce qui suit, nous commencerons par introduire les patrons de conception d'ontologies (voir Section 2.5.1) ensuite nous présenterons les patrons de gestion des changements (voir section 3.4).

11. http ://ontologydesignpatterns.org

2.5.1 Les patrons de conception d'ontologies

Les patrons de conception d'ontologies, plus connus sous leur appellation anglaise *Ontology Design Pattern* (ODP) ont été introduit par, entre autres, (Clark *et al.*, 2000; Gangemi *et al.*, 2004) en tant que guides de bonnes pratiques partagées dans le contexte de la conception collaborative d'ontologies. Ils fournissent des catalogues et des composants ontologiques réutilisables. Ainsi, ils apportent des solutions réutilisables liées aux problèmes de construction d'ontologies dans différents domaines tels que la conception des processus métier (Damjanovic, 2009), la conception d'ontologies biologiques (Aranguren *et al.*, 2008)...

Les ODPs sont classés en 6 catégories (Presutti & Gangemi, 2008) :

Les patrons de structure – comportent les patrons logiques et les patrons d'architecture. Les patrons logiques permettent de résoudre les problèmes d'expressivité. Il sont indépendants du domaine d'application modélisé par l'ontologie mais dépendent du langage utilisé pour la spécification de l'ontologie. Les patrons d'architecture sont définis en termes d'une composition de patrons logiques, ils permettent d'identifier une composition de patrons logiques nécessaire dans la conception d'une ontologie et ainsi guider les choix de conception en fonction de besoins spécifiques.

Les patrons de correspondance – consistent en des patrons de ré-ingénierie et les patrons d'alignement. Les patrons de ré-ingénierie sont un ensemble de règles de transformation qui permettent de construire une ontologie (modèle cible) à partir des éléments d'autres modèles (modèles sources) (*i.e.* un ontologie, un thésaurus, une structure linguistique, un modèle UML...) (Garcia-Silvia *et al.*, 2008). Les patrons d'alignement permettent de créer des associations sémantiques entre deux ontologies existantes (Scharffe, 2009).

Les patrons de raisonnement – correspondent à une implémentation orientée des patrons logiques ciblant les résultats d'un raisonnement particulier, se basant sur la comportement implémenté dans un raisonneur. Quelques exemples de raisonnements décrit par les patrons de raisonnement : la classification, la subsomption, l'héritage,...

Les patrons de présentation – traitent du problème de la lisibilité et de l'utilisation de l'ontologie de point de vue de l'utilisateur. Ils sont considérés

comme des guides de bonne pratique qui favorisent la réutilisation des ontologies. Il y a différentes catégories de patrons de présentation ; les patrons de nommage définissent des conventions pour nommer les entités d'une ontologie, les fichiers d'ontologies, les namespaces. . . et les patrons d'annotation fournissent les propriétés d'annotation qui permettent la bonne compréhension de l'ontologie et de ses éléments.

Les patrons lexico-syntaxique – peuvent être définis comme étant une structure linguistique qui consiste en un ensemble de mots ordonnés dans un ordre spécifique et qui permettent de généraliser et d'extraire des conclusions sur leur sémantique.

Les patrons de contenu – contrairement aux patrons logiques qui permettent de résoudre les problèmes de spécification, les patrons de contenu se focalisent sur les problèmes de conceptualisation liés au concepts et aux propriétés du domaine modélisé (Presutti & Gangemi, 2008). Ils permettent de guider la construction d'ontologies valides, par rapport au domaine modélisé, et facilement exploitable.

2.5.2 Les patrons de gestion de changement

Les patrons de gestion de changement (CMP) ont été proposés par (Djedidi & Aufaure, 2010) pour gérer l'évolution des ontologies tout en maintenant leur cohérence. Les CMP consistent en trois catégories de patrons : les patrons de changements, les patrons d'incohérences et les patrons d'alternatives. L'intérêt de ces patrons est d'offrir une modélisation abstraite de ces trois dimensions en établissant un lien entre les incohérences qui peuvent potentiellement être causées par un type de changement et les alternatives pour ces incohérences. Ceci permet d'automatiser le processus de gestion des changements.

les patrons de changements – définissent une catégorisation des patrons de changements d'ontologies et permettent de spécifier formellement leur signification, leur portée et leurs impacts. La modélisation des patrons de changements proposée suit la classification des changements proposée dans (Klein, 2004) qui définit deux classes de changement : des changements basiques et des changements complexes. Ainsi, il existe deux catégories de patrons de changements : les patrons de changements basiques qui représentent des changements modifiant une seule caractéristique

de l'ontologie (*i.e.* suppression de la relation de sous-classe) et les patrons de changements complexes qui représentent des changements qui correspondent à une séquence logique de changements basiques et/ou complexes.

les patrons d'incohérences – définissent une catégorisation des incohérences logiques qui peuvent être causées par les changements ce qui permet par l'interprétation des résultats du raisonneurs, de détecter l'axiome responsable de cette incohérence et ainsi préparer leur résolution.

les patrons d'alternatives – définissent une catégorisation des alternatives qui permettent de résoudre une incohérence. Un patron d'alternative modélise donc un changement qui peut être, soit un changement additionnel et donc appliqué conjointement au changement modélisé par le patron de changement, soit un changement substitutif qui remplace le changement initial.

Les CMP peuvent être considérés comme un type particulier des ODP. Un patron de changement peut être utilisé comme un guide de bonne pratique pour guider la modélisation des changements. Ainsi, un patron de changement peut correspondre à un patron de contenu (voir Section 2.5.1). D'un autre côté, un patron d'alternative permet de résoudre des problèmes de cohérences logiques dans la conception d'une ontologie. Ainsi, ils peuvent être considérés comme un type particulier de patron logique dont le rôle est de proposer des solutions liées à des problèmes d'expressivité liés au langage utilisé pour la spécification d'une ontologie.

2.6 Intégration des ontologies et des règles

L'intégration des règles et des ontologies est de plus en plus utilisé dans différents domaine et spécialement dans le web sémantique. En effet l'idée principale du web sémantique est d'étendre le web actuel en intégrant de la sémantique dans la description des informations, ce qui permettra la coopération entre les différents agents du web. Afin d'aboutir à ces objectifs, les recherches actuelles sont effectuées principalement sur deux couches de l'architecture du web sémantique qui sont les couches Ontologie et Règles (Antoniou *et al.*, 2004).

2.6.1 Problématique

Les ontologies sont fondées sur la logique de description (OWL) alors que les règles sont basées sur un fragment de la logique de premier ordre qui correspond à la logique déclarative de programmation (clauses de Horn par exemple) ce qui fait que l'intégration des règles et des ontologies est un travail délicat. En fait, la logique de description est fondée sur l'hypothèse du monde ouvert (tout ce qui n'est pas connu n'est pas défini) alors que la logique déclarative de programmation est fondée sur l'hypothèse du monde fermé (tout ce qui n'est pas connu est faux) (Rosati, 2005). L'intégration d'un langage de règles avec un langage d'ontologies nécessite la définition d'un nouveau langage, muni d'une syntaxe et d'une sémantique ainsi que de nouveaux algorithmes pour pouvoir raisonner sur ce nouveau langage. Les propositions existantes sur l'intégration des ontologies et des règles peuvent être classées en deux catégories (Antoniou *et al.*, 2004) :

- les approches hybrides : elles définissent une séparation entre les prédicats des règles et les ressources des ontologies, qui sont utilisées uniquement dans l'antécédent des règles . Le raisonnement est effectué en interférant les raisonneurs d'ontologies et les raisonneurs de règles. (*i.e.* ASP (Answer Set Programming)) ;
- les approches homogènes : elles définissent un langage logique (L) qui décrit les ontologies et les règles sans aucune distinction entre les ressources ontologiques et les prédicats des règles. Dans ce cas un sous ensemble (R) de (L) est défini. Pour raisonner sur cet ensemble il est possible soit d'utiliser un raisonneur de L soit de construire un raisonneur sur (R). (*i.e.* SWRL (Semantic Web Rule Language), DLP (Description Logic Propgramming)).

2.6.2 CWM et Euler

RDF (et RDFS) permet de décrire des connaissances du web sémantique mais ne permet pas d'extraire de nouvelles connaissances à partir de celle déjà décrites. Pour cela il est nécessaire d'intégrer les règles. Par exemple, en se fondant sur une ontologie décrivant les différents concepts de la coupe du monde, il est impossible de déduire que :

- Si une équipe représente un pays et qu'une personne est un joueur de

cette équipe alors ce joueur est un citoyen de ce pays.

ou bien,

– Si une personne supporte une équipe et que cette équipe va jouer un match alors cette personne achètera un ticket pour ce match .

CWM est un système développé en Python et spécialement conçu pour le web sémantique. Il supporte N-TRIPLES, RDF/XML et Notation 3 (N3) et permet d'effectuer des transformations entre ces différents formats de modélisation. CWM est implémenté par un ensemble de méthodes qui supportent les représentations logiques et les mécanismes d'inférences. La règles décrites ci-dessus sont définies avec CWM comme suit :

```
@prefix euro: <http://www.eurocup.org#> .
@prefix travel: <http://www.travelagency.com#> .
### Rule R1:
{ ?T euro:represents ?C. ?T euro:hasPlayer ?P. }
             => { ?P euro:citizenOf ?C. }.
### Rule R2:
{ ?S euro:supports ?T . ?M euro:matchTeam ?T. }
             => { ?S euro:buyTicket ?M. }.
```

Les antécédents et conséquences des règles sont des ensembles de triplets implicitement liés et toutes les variables sont universellement quantifiées.

RDFS n'est pas supporté par CWM mais il peut être implémenté partiellement en recourant à un ensemble de règles écrites en N3. Par exemple, les règles décrites ci-dessous sont implémentées pour la gestion des règles en RDFS, et utilisées dans l'un des module `rdfs-rules.n3` du système Euler.

```
{?S ?P ?O} => {?P a rdf:Property}.
{?P a rdf:Property} => {?P rdfs:subPropertyOf ?P}.
{?Q rdfs:subPropertyOf ?R. ?P rdfs:subPropertyOf ?Q}
=> {?P rdfs:subPropertyOf ?R}.
{?P has rdfs:subPropertyOf ?R. ?S ?P ?O} => {?S ?R ?O}.
```

Ceci étant CWM n'est pas un système complet pour la gestion de RDF et RDFS car il ne permet pas de gérer tous les littéraux XML.

Euler est un moteur codé en Java et C♯ basé sur des techniques de *loop-checking* pour garantir le traitement de toutes les données. Euler est un système qui supporte la majorité des types de données XML schema et permet de

détecter les incohérences dans le raisonnement effectué sur RDFS. En outre il fournit plusieurs modules capables de raisonner sur la sémantique décrite par RDF et aussi par le langage OWL. Ceci étant, les *loop-checking* ne sont pas efficaces car ils ne supportent pas les termes complexes définis dans la logique de programmation (Antoniou *et al.*, 2004).

2.6.3 ASP, Answer Set Programming

La majorité des approches combinant ontologies et règles sont face à un problème de décidabilité à cause de l'interaction entre deux différentes méthodes de représentation des connaissances. Aussi, elles sont incapables d'interagir avec différents formats de représentation de connaissances. Ce problème peut être résolu en traitant les ontologies et les règles de manière indépendante. C'est l'idée principale de cette approche. Dans cette approche, les ontologies sont considérées comme une source externe d'information. Les principaux avantages de cette approches sont :

- ASP est décidable : Les applications ASP sont décidables. Aucune restriction n'est nécessaire pour garder cette propriété.
- ASP est déclaratif : l'ordre des règles et des atomes n'est pas important et aucune connaissance sur la sémantique définie n'est requise.
- ASP est non déterministe : La définition des concepts n'est pas soumise à des contraintes, elle peut varier.
- ASP est *scalable* : la représentation des connaissances varie en fonction des besoins.

ASP a été utilisé comme un outil de spécification sûr dans un certain nombre d'applications. Par exemple, plusieurs tâches nécessitant des capacités de raisonnement complexes, tels que la gestion des connaissances, l'intégration des information, la gestion de la sécurité des informations ont été traité avec succès en utilisant ASP.

2.6.4 SWRL, Semantic Web Rule Language

SWRL est un langage de règle pour le web sémantique. Il est le résultat de la combinaison de OWL DL et OWL Lite et de RuleML. Il étend l'ensemble des axioms de OWL en y intégrant des clauses de Horn.

```
   a(x, y) ⋀ b(y, z) ⋀ c(x) ⋀ ... → n(x, z)
a, b, n: prédicats binaires (rôles)
c: prédicat unaire (concept atomique)
x, y : variables, instances ou littéraux.
```

SWRL fournit une syntaxe abstraite de haut niveau en incluant dans les ontologies OWL des règles ayant un syntaxe abstraite. Il permet de garder la puissance d'OWL DL, mais au prix de la décidabilité (Parsia *et al.*, 2005). Ainsi SWRL est un langage indécidable, à cause de l'interaction entre certaines caractéristiques de OWL-DL et de RuleML (Motik *et al.*, 2005). Les règles SWRL sont appliquées même si les individus ne sont pas présents dans la base. Une restriction de SWRL, appelée DL-safe rules, a été conçue pour conserver la décidabilité (Motik *et al.*, 2005). Cette restriction ne porte pas sur les composants du langage, mais sur leurs interactions. Les règles écrites ne peuvent porter que sur des individus explicitement présents dans la base. Autrement dit, les règles ne peuvent s'appliquer que si le type de toutes les instances présentes est connu.

De nombreux moteurs d'inférences commencent à supporter SWRL : Bossam, Hoolet, KAON2, Pellet, RacerPro, R2ML (REWERSE Rule Markup Language) et Sesame. Ils suivent trois types d'approche (Antoniou *et al.*, 2004) :

1. Traduire SWRL en logique du premier ordre (Hoolet).
2. Traduire OWL-DL en règles et appliquer un algorithme de chaînage avant (Bossam).
3. Intégrer les règles SWRL dans le moteur d'inférences OWL-DL fondé sur les algorithmes des tableaux sémantiques (Pellet).

Les règles définies par SWRL sont composées d'un antécédent (corp) et de la conséquence (en-tête), et si les conditions définies dans l'antécédent sont vérifiées alors les conditions définies dans la conséquence sont exécutées. Les deux parties constituant une règle (antécédent et conséquence) sont composées d'un ensemble d'atomes. Un antécédent ne contenant aucun atome est considéré comme vrai, c'est-à-dire qu'il est satisfait par toute interprétation, dans ce cas la conséquence est toujours satisfaite. Par contre

une conséquence ne contenant aucun atome est considérée comme fausse et dans ce cas l'antécédent n'est vérifié par aucune interprétation.
Une manière informelle pour écrire une règle est
`parent(?x,?y)` $\bigwedge$ `frère(?y,?z)` $\Rightarrow$ `oncle(?x,?z)`

En se basant sur une syntaxe abstraite, la règle est définie comme suit :

```
Implies(Antecedent(hasParent(I-variable(x1) I-variable(x2))
   hasBrother(I-variable(x2) I-variable(x3)))
Consequent(hasUncle(I-variable(x1) I-variable(x3))))
```

La definition des règles avec SWRL est faite en XML, en se basant sur la syntaxe de OWL et de RuleML.

```
<swrlx:classAtom>
  <owlx:Class owlx:name="Person" />
  <ruleml:var>x1</ruleml:var>
</swrlx:classAtom>

<swrlx:classAtom>
  <owlx:IntersectionOf>
    <owlx:Class owlx:name="Person" />
    <owlx:ObjectRestriction owlx:property="hasParent">
      <owlx:someValuesFrom owlx:class="Physician" />
    </owlx:ObjectRestriction>
  </owlx:IntersectionOf>
  <ruleml:var>x2</ruleml:var>
</swrlx:classAtom>
```

2.6.5 DLP, Description Logic Programs

L'une des motivations pour combiner les règles et les ontologies est de concevoir des Services Web Sémantiques (SWS). DLP est un langage qui permet d'intégrer des bases de connaissance décrites dans la logique de description avec la logique de programmation (Antoniou *et al.*, 2004). DLP fournit un lien formel entre les deux modèles de représentation de connaissance qui sont la logique de description et la logique de programmation. DLP n'est pas un langage de représentation de connaissances, puisqu'il ne définit pas une nouvelle syntaxe ou une nouvelle sémantique. DLP définit un ensemble de contraintes sur OWL DL afin de garantir une transformation efficace de tous les axiomes

de OWL en des clauses de Horn. DLP a été conçu en se basant principalement sur ces trois concepts (Grosof & Horrocks., 2003) :

1. DLP est syntaxiquement un fragment de OWL dans le sens que chaque base de connaissances DLP est une base de connaissances de OWL syntaxiquement valide.
2. DLP hérite de la sémantique de OWL.
3. Chaque base de connaissances DLP est sémantiquement équivalente à un ensemble de clauses de Horn.

La transformation des axiomes OWL en clauses de Horn est une source de confusion puisque les clauses de Horn sont utilisées dans la logique de programmation qui est basée sur la théorie du monde fermé alors que la logique de description est basée sur la théorie du monde ouvert.

2.6.6 Synthèse

Les règles décrites ci-dessus sont des règles techniques. En fait ces règles ne peuvent être définies et assimilées que par un expert du domaine qui a une parfaite connaissance de la logique de programmation. Or, notre travail vise principalement des personnes qui n'ont aucune notion sur la logique de description ou la logique de programmation. C'est pour cela que nous nous sommes intéressés aux règles métier. Les règles métier permettent de définir des règles dans un langage naturel assimilable par tout type d'utilisateurs.

2.7 Les systèmes de gestion des règles métier

Cette section est fondée sur une étude effectuée par (Legendre *et al.*, 2010). Les systèmes de gestion de règles métier (SGRM), plus connus sous l'acronyme anglophone BRMS qui signifie "Business Rules Management Systems", sont une descendance des systèmes experts. Le développement des systèmes experts consiste à acquérir des connaissances d'un expert métier et de les formaliser sous forme de faits et de règles qui sont ensuite exploités par un moteur d'inférences pour inférer de nouvelles connaissances. Néanmoins, vu la quantité énorme des connaissances à gérer et leur évolution constante, le

développement et la maintenance de ce type de système sont devenus de plus en plus difficile, d'où l'utilisation des SGRM.

2.7.1 Objectifs des systèmes de gestion de règles métier

Comme nous l'avons décrit dans la Section 2.2.2, le principe des règles métier est de séparer la logique métier (*i.e.* formalisation des règles en utilisant le vocabulaire métier) de la logique système (*i.e.* la mise en œuvre technique). Ainsi, l'objectif principal d'un SGRM est de fournir un environnement d'édition de règles, dans un langage compréhensible et intuitif et qui peut facilement être utilisé par les experts métier, ce qui faciliterait la maintenance et la gestion de l'évolution de l'application métier par l'expert.

Aussi, comme nous l'avons déjà décrit dans la Section 2.2.2, un des objectifs des règles métier est d'automatiser les décisions et le savoir faire de l'expert, pour cela un SGRM doit fournir une solution pour exécuter et vérifier la cohérence des règles éditées et ce de manière transparente pour l'utilisateur.

2.7.2 Architecture et composants

Un SGRM offre généralement deux types de composants, les composants dédiés aux informaticiens et les composants dédiés aux experts métier.

Composants dédiés aux informaticiens – (1) un environnement de développement qui permet de modéliser une couche métier, *i.e.* un modèle représentant toutes les entités du métier ainsi que leurs propriétés et relations et qui fournit un éditeur de règles, (2) un moteur d'inférence (API java, C++, .Net, ...) qui permet d'exécuter les règles ; qui sont écrites dans un langage propre au produit. L'exécution est généralement basées sur l'algorithme RETE qui est adapté selon le produit.

Composants dédiés aux experts métier – une interface web d'édition des règles, orientée experts métier, permettant le partage des règles sur un serveur et la gestion des versions des règles de manière collaborative.

L'architecture classique d'un SGRM est représentée dans la Figure 2.6. Tout application à base de règles commence généralement dans l'environnement de développement, avec une collaboration entre le développeur et l'expert métier. Cette étape consiste à définir le modèle métier et le modèle de

règles ainsi que quelques fonctions nécessaires au développement de l'application. Ces éléments sont ensuite déployés dans le *repository* de manière à ce que les utilisateurs métier puissent y accéder via leur interface web. D'autres règles peuvent ensuite être éditées via cette interface et chaque mise à jour doit être sauvegardée dans le *repository* afin que tous les utilisateurs puissent y accéder. Enfin, l'exécution des règles est effectuée à l'aide du moteur d'inférences après le déploiement des règles sauvegardées dans le *repository*.

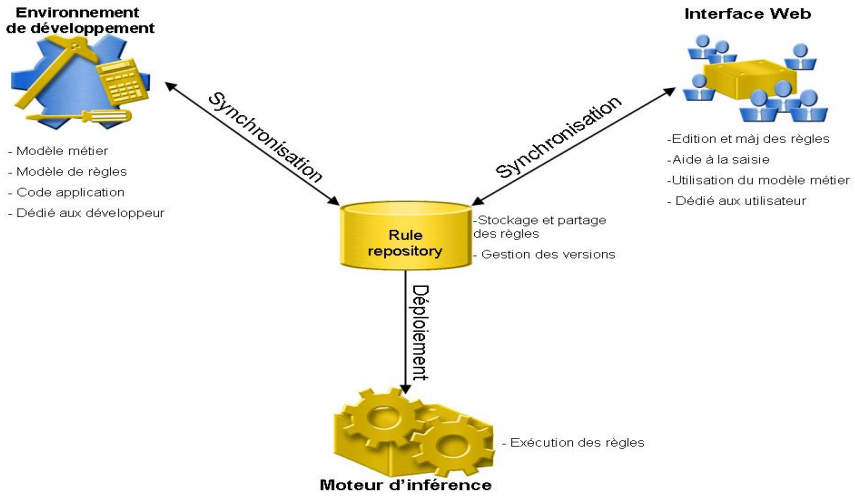

FIGURE 2.6 – Architecture classique des systèmes de gestion des règles métier.

2.7.3 Fonctionnement des systèmes de gestion des règles métier

Définition du langage métier

Afin de permettre aux experts métier d'implémenter et de maintenir à jour leurs systèmes à base de règles, il est nécessaire de fournir un langage de règles intuitif facile à manipuler qui utilise le vocabulaire du métier. Un langage métier doit donc permettre de représenter les entités du domaine métier, les actions à mener et les stratégies à suivre.

Pour ce faire, les SGRM se fondent sur un modèle physique de données (MPD), représentant toutes les entités du métier ainsi que leurs propriétés et relations, pour générer un modèle conceptuel de données (MCD) représentant

le vocabulaire à utiliser pour l'écriture des règles. Ainsi le langage métier est une mise en correspondance du MPD et du MCD (voir Figure 2.7) .

Il est à noter que la définition du langage métier, *i.e.* la définition du MPD et la génération du MCD est effectuée par un technicien en utilisant les composants qui lui sont dédiés (voir Section 2.7.2), et donc de manière transparente à l'utilisateur.

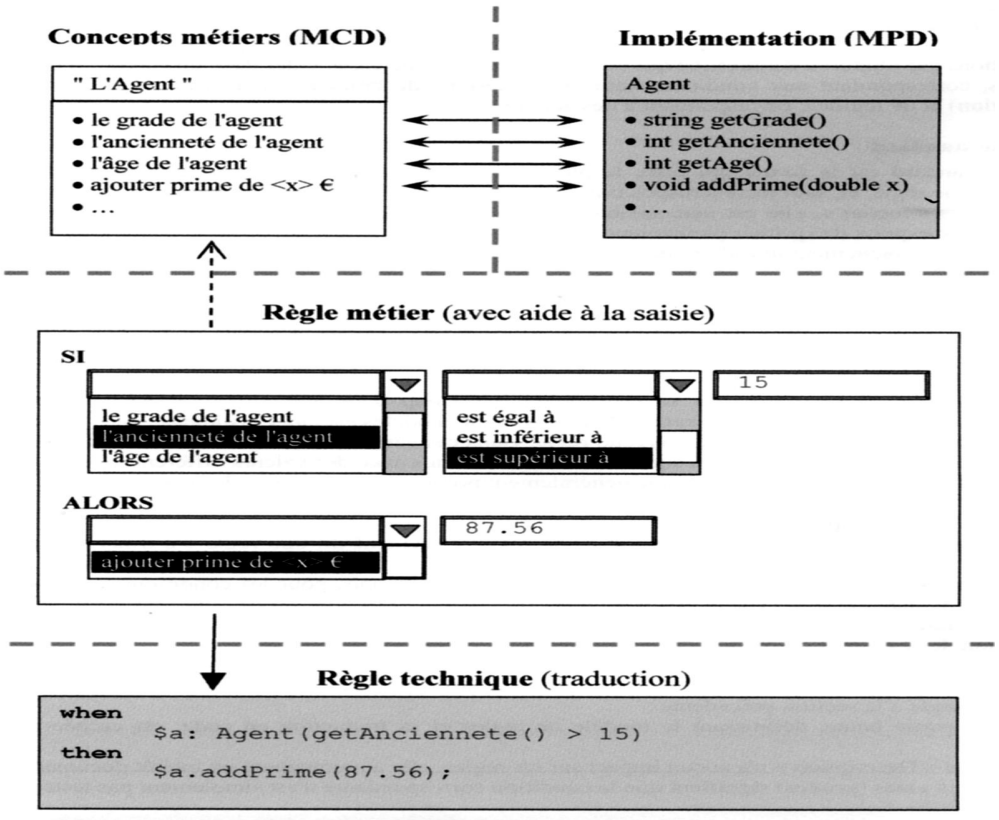

FIGURE 2.7 – Définition du langage métier et des règles (Legendre *et al.*, 2010).

Écriture des règles

Après la génération du langage métier, l'expert métier peut écrire les règles de manière autonome. La majorité des SGRM proposent trois formats pour définir les règles :

les règles SI - ALORS – c'est le format qui offre le plus de possibilité pour décrire des conditions et des actions complexes. Néanmoins, la souplesse

et la flexibilité de ce format peut entraîner des erreurs qui rendent le système inutilisable, bien que la plupart des SGRM fournissent une aide à la saisie pour guider les utilisateurs lors de l'édition des règles. Une solution pour éviter ces erreurs et de définir des modèles de règles qui permettent de figer certaines parties.

les tables de décision – elles permettent de regrouper un ensemble de règles qui ont le même modèle avec seulement des différences sur les valeurs testées et/ou assignées. Ce format permet d'éviter de réécrire plusieurs fois les mêmes règles avec à chaque fois un jeu de valeurs qui change. Une table de décision est composée de colonnes conditions et de colonnes actions représentant les paramètres de la règle. Chaque ligne de la table représente une règle en fonction des valeurs de chaque cellule (voir Figure 2.8). Les tables de décisions sont très utilisées par les utilisateurs métier parce qu'elles sont faciles à manipuler. Néanmoins, le fait qu'elles représentent un modèle de règles figé les rend peu flexibles. En effet, dans la majorité des SGRM la modification d'une table de décision (*i.e.* ajout/suppression de condition ou action) ne peut être effectuée par les utilisateurs, il est nécessaire d'avoir l'intervention d'un technicien.

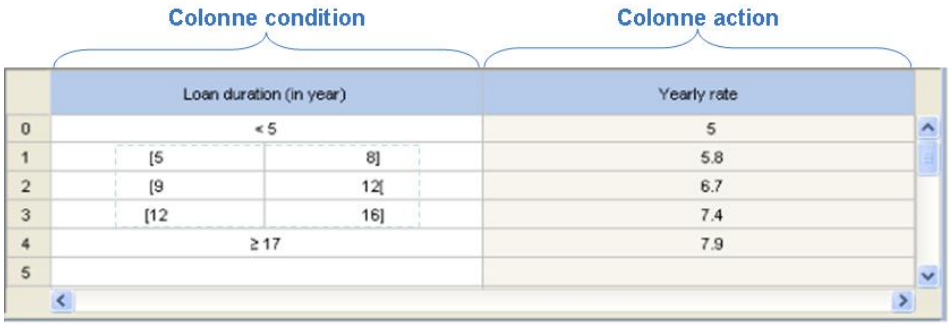

FIGURE 2.8 – Exemple d'une table de décision réalisée avec IBM ODM.

les arbres de décision – elles permettent d'optimiser le processus d'exécution des règles. Les nœuds d'un arbre de décision représentent les tests, les arcs représentent les valeurs des tests et les feuilles représentent les actions. Ainsi, la vérification de certaines conditions ne peut être effectuée que si certaines conditions sont vraies (voir figure 2.9).

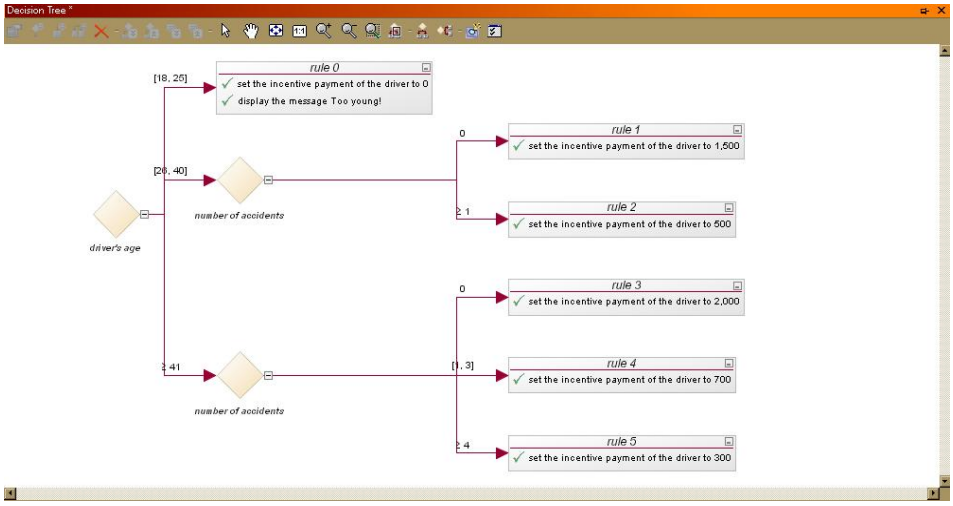

FIGURE 2.9 – Exemple d'un arbre de décision réalisée avec IBM ODM.

Exécution des règles

Lors du lancement du processus d'exécution, les règles éditées par les utilisateurs métier sont traduites dans un langage technique qui dépend du SGRM, capable d'être traité par un moteur de règles. Un moteur de règle est généralement constitué de trois partie ; le *rule set* contenant l'ensemble des règles représentées dans un langage techniques, la *working memory* ou mémoire de travail contenant l'ensemble des objets (appelés aussi des faits) de l'application qui permettront de déclencher l'exécution des règles et l'agenda qui contient l'ensemble des règles éligibles *i.e.* les règles dont la partie condition est vérifiable (évaluée à vrai) sur un ensemble d'objet de la mémoire de travail.

Le processus d'exécution est effectué en trois étapes indépendamment du mode d'exécution [12] (voir Figure 2.10) :

1. détermination les règles éligibles et les objets associés : cette association règles-objets consiste en une instanciation de règles. L'ensemble des règles instanciées constitue l'agenda ;
2. organisation de l'exécution des règles : consiste à définir l'ordre d'exécution des règles. Pour cela, certains SGRM tels que IBM ODM offrent la possibilité d'orchestrer l'exécution des règles via un *rule flow*. La règle

12. Il existe différents algorithmes pour l'exécution des règles : RETE, séquentiel, fast path,...

la plus prioritaire est exécutée en premier ;

3. exécution des règles instanciées et mise à jour de la base des faits en fonction de la modification apportée aux objets. Cette mise à jour modifie l'ensemble des règles éligibles et ainsi modifie l'état de l'agenda. Cette opération continue jusqu'à ce que plus aucune règle n'est éligible.

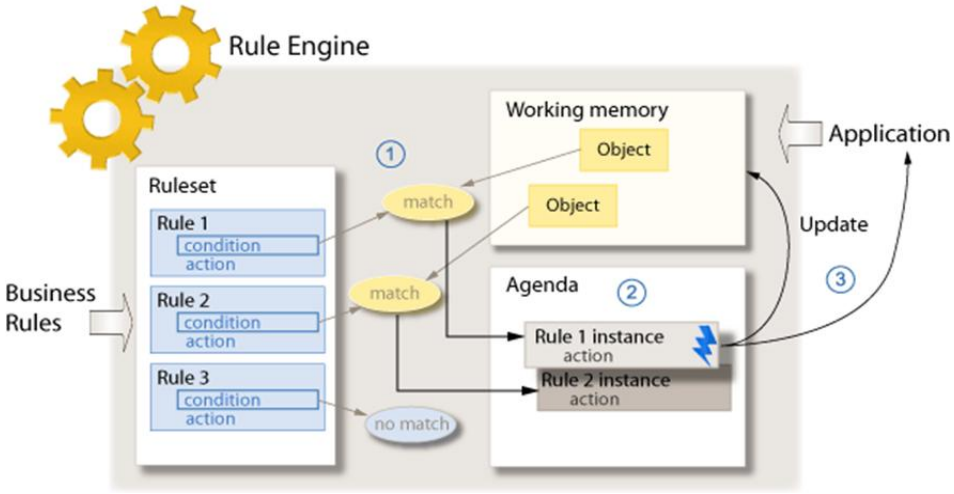

FIGURE 2.10 – Moteur de règles : processus d'exécution.

2.7.4 Les principaux système de gestion de règles métier

Les principaux système de gestion de règles métier sont des systèmes commerciaux : IBM Operational Decision Management (anciennement ILOG JRules) et Blaze Advisor (FICO, Fair Isaac Corporation). Ces deux outils offrent de très bonnes performances pour la gestion des grandes bases de règles et de données et leur fiabilité a été prouvée depuis une dizaine d'années sur plusieurs projets industriels.

Drools (ou JBoss Rules, développée par Red Hat) est un autre système open-source qui contient les fonctionnalités nécessaires qu'un tel système doit fournir. Néanmoins, il reste moins performant que les systèmes commerciaux pour les gros volumes de données et de règles.

2.8 Discussion

La problématique principale de ce travail est la gestion de l'impact de l'évolution des modèles de représentation des connaissances sur les ressources et autres modèles qui en dépendent. Une solution à cette problématique permettrait de faciliter la maintenance et la mise à jour des systèmes d'informations où les connaissances sont codées dans un langage de programmation, quasiment impossible à comprendre par les non-informaticiens et parfois même par les informaticiens.

L'idée que nous avons adoptée pour résoudre ce problème est de séparer les connaissances du métier de l'application codée en bas niveau et de les représenter dans des modèles dont l'évolution est plus ou moins simple à gérer, par un informaticien et par l'expert métier. Pour cela, nous nous sommes focalisés sur les ontologies écrites en OWL et les règles métier.

1. Pourquoi les ontologies OWL ? – Les ontologies écrites en OWL offrent un très grand pouvoir d'expressivité ainsi que la possibilité d'inférer de nouvelles connaissances grâce au nombre de constructeurs de ce langage ainsi qu'au nombre de raisonneurs OWL (Jena, Pellet, racer,...). Sans oublier l'interopérabilité de ce modèle et la possibilité de réutilisation et d'adaptation des ontologies aux besoins du domaine. De plus, depuis que les ontologies sont devenues de plus en plus utilisées, plusieurs approches qui permettent de gérer l'évolution des ontologies tout en maintenant leur cohérence (voir Section 2.4) ont été développées. Néanmoins, il reste toujours difficile de modéliser des actions, des procédures ou des contraintes dans des ontologies, d'autant plus qu'ils n'est pas possible à un expert métier, n'ayant pas des connaissances en logique de premier ordre ou en logique de description, de pouvoir les manipuler.
2. Pourquoi les règles métier ? – Les règles métier permettent de décrire les contraintes et les critères de décision d'un domaine, de définir les actions à mener pour un processus donné et d'automatiser des décisions. Elle permettent de séparer la logique métier (le comportement) de la logique système (le comment) et sont généralement écrites dans un langage pseudo-naturel ce qui rend leur utilisation et leur maintenance faciles pour les experts métier. Les applications à base de règles métier sont de plus en plus utilisées depuis l'avènement des systèmes de gestion

des règles métier, qui permettent l'écriture, l'exécution et la mise à jour de règles. Il existe plusieurs SGRM qui offrent plus ou moins les mêmes fonctionnalités (voir Section 2.7). Néanmoins, certains d'entre eux (principalement les systèmes commerciaux) offrent une meilleure gestion des gros volumes de données et une meilleure gestion des différentes versions des ensembles règles.

3. Pourquoi utiliser IBM ODM ? – Dans notre travail, nous utilisons le système de gestion des règles métier IBM ODM (ex. ILOG JRules) parce qu'il offre une plateforme regroupant les principales fonctionnalités qu'un tel système doit fournir à savoir, une bonne performance pour le traitement des gros volumes de données et de règles, un éditeur de règles dans un langage naturel contrôlé qui permet de guider les utilisateurs lors de l'édition des règles métier, des éditeurs de tables de décision et d'arbres de décision, un *rule flow* qui permet d'orchestrer l'exécution des règles. . . . D'un autre côté, ODM offre une plateforme de travail collaboratif permettant le partage des règles entre les différents collaborateurs et permet aussi la gestion des différentes versions des règles.

Le vocabulaire utilisé pour l'écriture des règles métier est celui utilisé dans le domaine d'application. Pour cela, il est nécessaire de modéliser les différentes entités du domaine et les relations entre eux. Ainsi, vient l'idée d'intégrer les ontologies et les règles métier. L'intégration des ontologies et des règles a été effectuée dans différentes approches orientées web sémantique. Le résultat le plus cité est SWRL (voir Section 2.6). Néanmoins, ce langage est doté d'une syntaxe technique fondée sur celle de OWL et de RuleML et donc impossible à gérer par des experts métier.

L'approche que nous proposons, $\mathcal{MDR}$ (Model - Detect - Repair), pour la gestion de l'impact des évolutions des ontologies sur les règles est une approche à base de patron de gestion des changements. Cette catégorie de patron a été proposée par (Djedidi & Aufaure, 2010) pour gérer l'évolution des ontologies tout en maintenant leur cohérence. $\mathcal{MDR}$ propose une extension de cette approche afin de pouvoir non seulement gérer les évolutions des ontologies mais surtout l'évolution des modèles de représentation des connaissances en général ainsi que leurs impacts sur les ressources et les modèles qui en dépendent.

Deux points importants sont à signaler pour la suite :

1. Dans notre travail, l'intégration des ontologies et des règles n'est pas la problématique principale. Pour gérer l'intégration des ontologies et des règles, nous avons développé une approche hybride qui permet d'écrire et d'exécuter des règles métier à partir d'ontologies OWL et ce en important les ontologies dans ODM pour ensuite exploiter la plateforme offerte par cet outil pour l'édition et l'exécution des règles. Les règles sont donc écrites à partir des entités modélisées par l'ontologie, d'où la dépendance entre ces deux modèles.
2. Nous ne traitons pas du problème de gestion de la cohérence des ontologies lors de leur évolution. Nous supposons que la cohérence des ontologies est gérée par un autre système (*e.g.* ONTO-EVO^{A}L), que les régles sont écrites à partir d'ontologies cohérentes et que les évolutions d'ontologies n'impactent pas la cohérence des ontologies mais elles impactent la cohérence des règles.

2.9 Conclusion

Dans ce chapitre, nous avons présenté les différents axes auxquels touche notre problématique de recherche. Nous avons commencé par présenter les différentes techniques de représentation des connaissances ainsi que les langages qui permettent de formaliser les différentes techniques. Ensuite, nous avons présenté une synthèse de l'état de l'art sur l'évolution des ontologies et nous avons présenté les différentes opérations de changement des ontologies, les classifications des changements ainsi que les différentes approches pour les représenter. Nous avons aussi présenté les différentes méthodes de gestion de l'évolution des ontologies. Nous nous sommes aussi intéressés aux méthodes d'intégration des ontologies et des règles et aux différentes approches (hybrides et homogène) qui supportent une telle intégration. Enfin, les fonctionnalités des systèmes de gestion des règles métier, leur architecture ainsi que leur méthode de fonctionnement.

Dans ce qui suit, nous présentons la principale contribution de notre travail, l'approche $\mathcal{MDR}$ qui permet de gérer l'impact de l'évolution des ontologies sur les règles métier écrites à partir d'ontologies.

Chapitre 3

$\mathcal{MDR}$: Approche de gestion de l'évolution de modèles

Sommaire

3.1 Introduction

Comme nous l'avons décrit dans le chapitre 1, l'idée principale de ce travail est l'intégration des ontologies et des règles afin de permettre aux experts métier de décrire des contraintes et d'automatiser des décisions métier dont la sémantique est formalisée avec OWL. Cette intégration crée une dépendance entre les ontologies et les règles due au fait que les règles sont éditées à partir des concepts et des propriétés de l'ontologie et que leur exécution dépend, entre autres, des individus présents dans celle-ci.

Étant donné le caractère évolutif des ontologies, nous pensons qu'il est nécessaire d'analyser l'impact des changements de l'ontologie sur les règles et de mettre en œuvre un approche qui permet (1) de gérer l'évolution des règles en fonction de l'évolution de l'ontologie correspondante et (2) de gérer de la cohérence des règles lors de cette évolution.

Nous supposons ici qu'il y a un système qui gère l'évolution et la cohérence des ontologies et donc que les ontologies sont cohérentes. Nous nous intéressons à la gestion de l'évolution et de la cohérence des règles en fonction de l'évolution de l'ontologie. Pour cela, nous proposons une approche, que nous appelons $\mathcal{MDR}$ ($\mathcal{M}$odéliser - $\mathcal{D}$étecter - $\mathcal{R}$éparer), qui permet de modéliser des changement d'ontologies, détecter les problèmes qu'ils peuvent causés sur les règles et proposer des réparations pour résoudre les problèmes détectés. Cette approche est à base de patron de conception ou plus précisément des Patrons de Gestion des Changements (voir Section **??**) inspirés de l'approche ONTO-EVO^{A}L (Djedidi & Aufaure, 2010) qui permet la gestion de l'évolution des ontologies. Les patrons de gestion des changements consistent en trois catégories de patrons : les patrons de changement, les patrons d'incohérence et les patrons de réparation. Dans notre approche, les patrons de changement sont modélisés au moyen d'une ontologie OWL, appelé l'ontologie $\mathcal{MDR}$, les patrons d'incohérence et de réparation sont modélisés au moyen de règles éditées à partir de l'ontologie $\mathcal{MDR}$.

Nous proposons donc un système implémenté à base d'une ontologie OWL (*i.e.* L'ontologie $\mathcal{MDR}$) et d'un ensemble de règles, qui permet de gérer l'impact de l'évolution d'une ontologie métier sur les règles métier écrites à partir de cette dernière. [1]

Dans ce qui suit, nous appelons ontologie métier l'ontologie qui modélise un domaine métier, à partir de laquelle les règles métier ont été éditées. Le principe de $\mathcal{MDR}$ est de :

1. détecter les changements de l'ontologie métier et de les modéliser au moyen de l'ontologie $\mathcal{MDR}$;
2. détecter les incohérences qui peuvent être causées par les changements au moyen des patrons d'incohérence ;
3. proposer des solutions pour réparer les incohérences détectées au moyen des patrons de réparation.

Dans ce qui suit, la Section 3.2 présente une modélisation de l'approche $\mathcal{MDR}$ en utilisant la méthode CommonKADS qui permet la représentation des systèmes à base de connaissances. Ensuite, la Section 3.3 décrit les différentes catégories de changements d'ontologies que nous avons définies ainsi que leurs impacts sur les règles. La Section 3.4 présente les différents composants de notre approche, à savoir les patrons de gestion des changements, ainsi que leur spécification. Enfin, la Section 3.5 décrit l'architecture du système proposé ainsi que son fonctionnement.

3.2 Modèle conceptuel de $\mathcal{MDR}$: application de la méthode CommonKADS

Afin de donner une vue générale de notre approche, nous proposons dans cette section une modélisation abstraite de notre *framework*. Pour cela, nous avons utilisé la méthode CommonKADS (Schreiber *et al.*, 2000). Cette méthode, utilisée pour la modélisation et le développement des systèmes à base

1. Lors des soumissions de notre approche à des conférences, beaucoup de rapporteurs ont compris que les règles sont écrites par les utilisateurs métier. Il est donc important de signaler que l'ontologie $\mathcal{MDR}$ ainsi que les règles font partie de l'implémentation de notre système et sont complètement transparentes pour les utilisateurs métier.

de connaissance, propose six modèles (voir Figure 3.1) qui permettent de répondre aux questions suivantes :

Pourquoi ?– Pourquoi un système à base de connaissance peut être la solution ? Pour quel problème ? Quel impact peut il avoir ? Ceci permet de définir le contexte.

Quoi ?– Quelle est la nature et la structure des connaissances impliquées dans une tâche ? La conceptualisation des connaissances impliquées dans une tâche est très importante. Ceci permet de définir les concepts nécessaires au système.

Comment ?– Comment devrait-on implémenter cette connaissance ? Ceci permet de définir les artefacts nécessaires à l'implémentation du système.

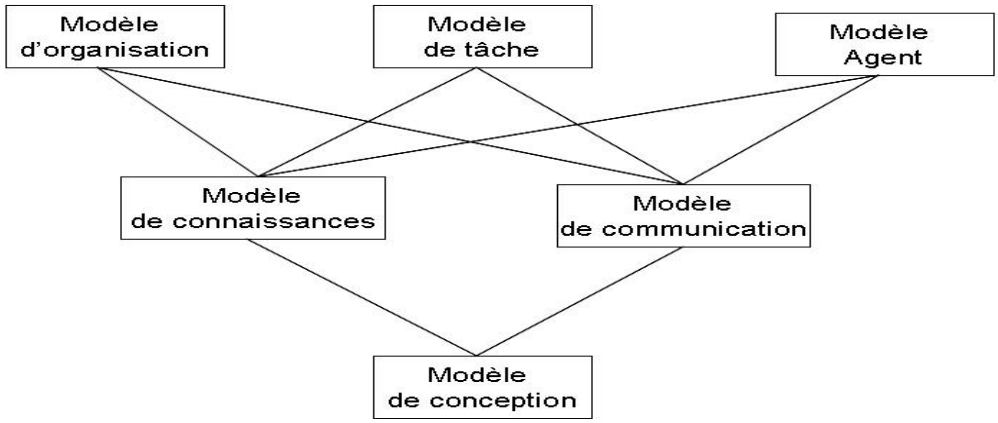

FIGURE 3.1 – Les modèles CommonKADS.

Il n'est évidemment pas toujours nécessaire d'avoir tous les modèles. Tout dépend des besoins du projet et du système à développer. Les modèles d'organisation, des tâches et des agents sont nécessaires pour l'analyse de l'environnement organisationnel ainsi que les facteurs nécessaires au système pour accomplir les objectifs attendus. Les modèles de connaissance et de communication permettent la modélisation des méthodes de résolution des problèmes et des données qui doivent être gérées par le système. Le modèle de conception permet la conversion de ces modèles en des spécifications techniques nécessaires à l'implémentation du système.

Pour modéliser notre système nous avons développé le modèle des connaissances. Ce modèle comprend les trois catégories de connaissances suivantes :

Connaissances du domaine – elles décrivent la structure des connaissances/information du domaine d'application. La description des connaissances du domaine est effectuée sur deux niveaux ; (1) le *schéma du domaine*, une représentation graphique des informations et connaissances du domaine. Un ou plusieurs schémas peuvent être décrit. (2) La *base de connaissances* qui contient les instances des types modélisés dans le *schéma du domaine*.

Connaissances d'inférences – elles décrivent comment utiliser les connaissances du domaine pour raisonner et inférer de nouvelles connaissances.

Connaissances de tâches– elles décrivent le but à atteindre et la stratégie à suivre pour l'atteindre.

Dans ce qui suit, nous utilisons les définitions précédemment données pour modéliser le modèle de connaissance du framework $\mathcal{MDR}$. La notation utilisée dans les figures présentées dans cette section est la notation proposée par la méthode CommonKADS qui est assez similaire à celle proposée dans UML (Unified Modeling Language) [2].

3.2.1 Les connaissances du domaine

L'approche $\mathcal{MDR}$ permet de gérer l'évolution des modèles de représentation des connaissances, d'analyser l'impact de leurs évolutions pour détecter les problèmes potentiels qui peuvent être causés et de proposer des solutions pour les réparer. La Figure 3.2 représente les entités les plus générales de notre approche. La représentation graphique des connaissances du domaine est illustrée à l'aide des concepts du domaine (représentés par des rectangles), des relations entre les concepts (représentées par des flèches annotées) et aussi à l'aide des «Types de règle» (*Rule type*) qui modélisent des expressions du domaine sous forme de règles ayant un antécédent et une conséquence reliés par un symbole de connexion. Les antécédents et conséquences d'une règle sont des concepts du domaine. Chaque règle possède un nom représenté dans une ellipse. L'entité *Model* représente l'ensemble des modèles de représentation des connaissances et l'entité *ModelChange* qui représente les changements de ces modèles. Un ou plusieurs *ModelChange* peuvent être appliqués à un *Model*

2. http ://www.uml.org/

et peuvent causer des *Problem* qui sont résolus par des *Solution*. En tant que modèles de représentation des connaissances, nous prenons en considération les ontologies et les règles (*i.e. Ontology* et *Rule*) et nous nous intéressons donc aux changements des ontologies et des règles (*i.e. OntologyChange* et *RuleChange*). Quant aux problèmes causés par les changement, nous considérons deux types de problème, les *RuleInconsistency* et les *OntologyInconsistency*. Néanmoins dans notre travail nous nous focalisons principalement sur les incohérences des règles et nous supposons que la cohérence de l'ontologie est gérée par un autre système (*e.g.* ONTO-EVO^{AL}). L'entité *Repair* représente l'ensemble des solutions proposées pour réparer les incohérences. Les réparations sont considérées comme des changements.

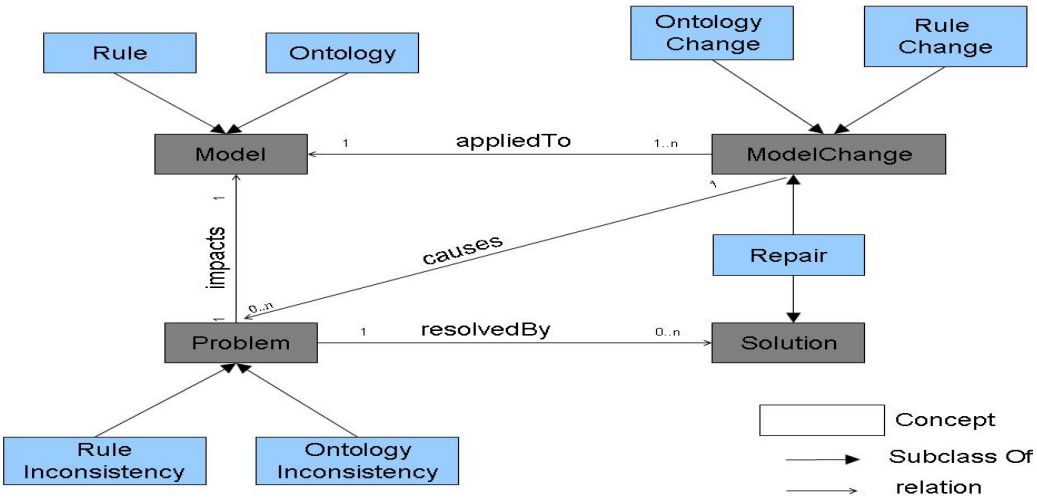

FIGURE 3.2 – $\mathcal{MDR}$ framework : connaissances du domaine.

La Figure 3.3 est une spécification de la figure 3.2. Dans notre approche, les règles dépendent des entités (*i.e.* Concepts et propriétés) de l'ontologie. Ainsi, les changements d'ontologies peuvent causer des problèmes de cohérence qui impactent les règles. Ces problèmes de cohérences sont réparés par des réparations qui peuvent être soit (1) des changements d'ontologies alternatives qui permettent d'éviter le problème de cohérence, soit (2) des changements sur les règles qui permettent de préserver la cohérence de la base de règles.

Dans le domaine de connaissance de notre approche, deux types de règles (*rule type*) peuvent être définies, la première est qu'un changement d'ontologie peut causer zéro ou plusieurs incohérences dans les règles (voir figure 3.4),

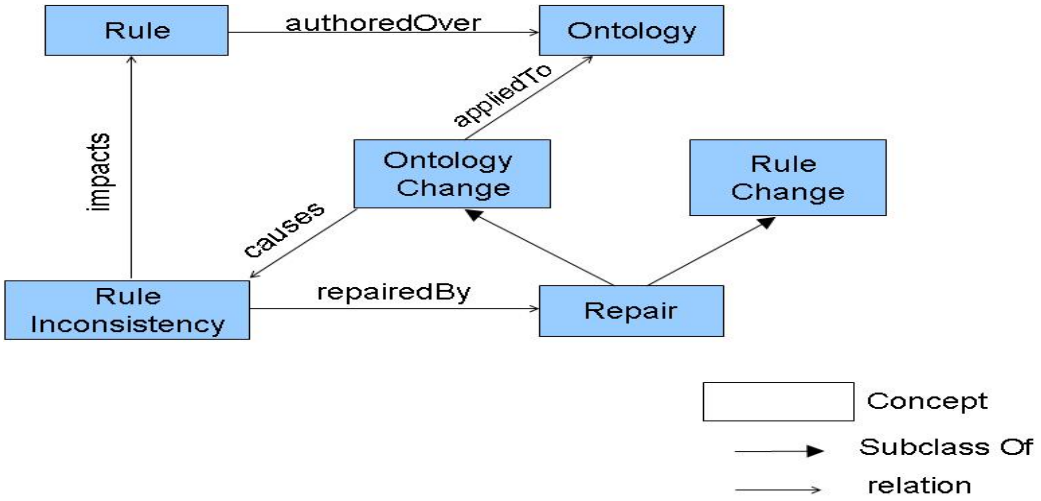

FIGURE 3.3 – $\mathcal{MDR}$ Framework : spécification des connaissances du domaine

la deuxième est qu'une incohérence de règle peut être réparée par zéro ou plusieurs réparations (voir Figure 3.5).

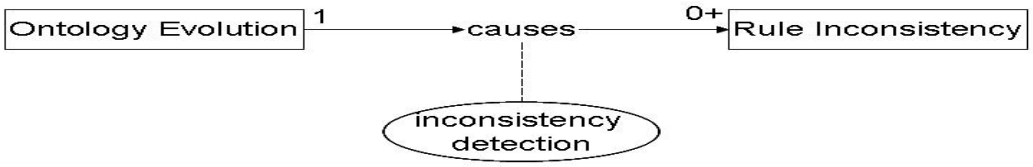

FIGURE 3.4 – Règle de détection des incohérences.

Dans le cas où le système ne propose aucune réparation, l'utilisateur peut choisir entre le fait d'annuler le changement ou de l'appliquer, peut importe les types d'incohérences détectées par le système. Dans le cas où le système propose plusieurs solutions, celle choisie par l'utilisateur sera appliquée. Ceci étant, dans les deux cas l'utilisateur peut choisir de n'appliquer aucune des solutions proposées, auquel cas le changement est appliqué même s'il cause des incohérences dans la base de règles.

3.2.2 Les connaissances d'inférences

Les connaissances d'inférences décrivent comment utiliser les connaissances du domaine afin de pouvoir raisonner et inférer de nouvelles connaissances.

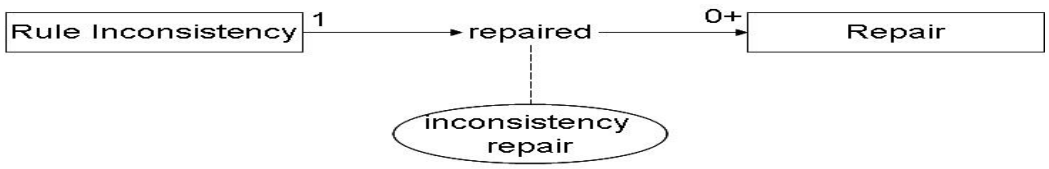

FIGURE 3.5 – Règle de réparation.

Une inférence permet de raisonner sur les connaissances du domaine. Les entrées/sorties d'une inférence sont décrites en termes de *rôles fonctionnels*, un nom abstrait décrivant le rôle d'un ensemble de données dans le processus de raisonnement.

La notation graphique utilisé dans CommonKADS pour représenter les structures d'inférences est la suivante : les rectangles représentent les rôles fonctionnels impliqués dans l'inférence ; les cercles ovales représentent les inférences ; les flèches entre les rectangles et les cercles ovales représentent la relation d'entrée/sortie entre les rôles et les inférences ; les rectangles avec des angles ronds représentent les fonctions de transfert, décrits plus bas dans cette section.

Dans notre système, nous distinguons deux types d'inférences. La première inférence est *causes* qui permet d'inférer le(s) problème(s) causé(s) par un changement (voir Figure 3.6). Cette inférence prend en entrée une description d'un changement, ce qui correspond à une instance de l'entité *OntologyChange* du domaine de connaissance. En sortie, elle détecte le problème causé par ce changement, ce problème correspond à une instance de l'entité *RuleInconsistency* de notre domaine de connaissance.

La deuxième inférence est *repaired*. Elle permet de proposer une ou plusieurs solutions pour réparer un problème (voir Figure 3.7). Cette inférence prend en entrée un problème, ce qui correspond à une instance de l'entité *RuleInconsistency* du domaine de connaissances et produit en sortie un ensemble de réparations, ce qui correspond à des instances de l'entité *Repair*.

Une fois les inférences nécessaires définies, il est possible avec la méthode CommonKADS de modéliser la structure d'inférences. Une structure d'inférences permet de définir le processus de raisonnement ainsi que les interactions avec le monde extérieur (*i.e.* utilisateurs ou autre système). Pour cela, cette méthode propose les fonctions de transfert qui permettent de transférer les

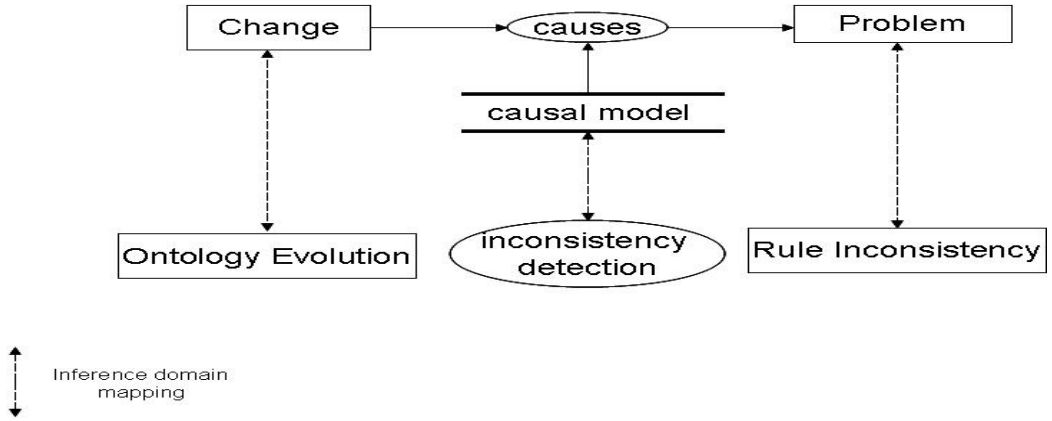

FIGURE 3.6 – Modèle causal.

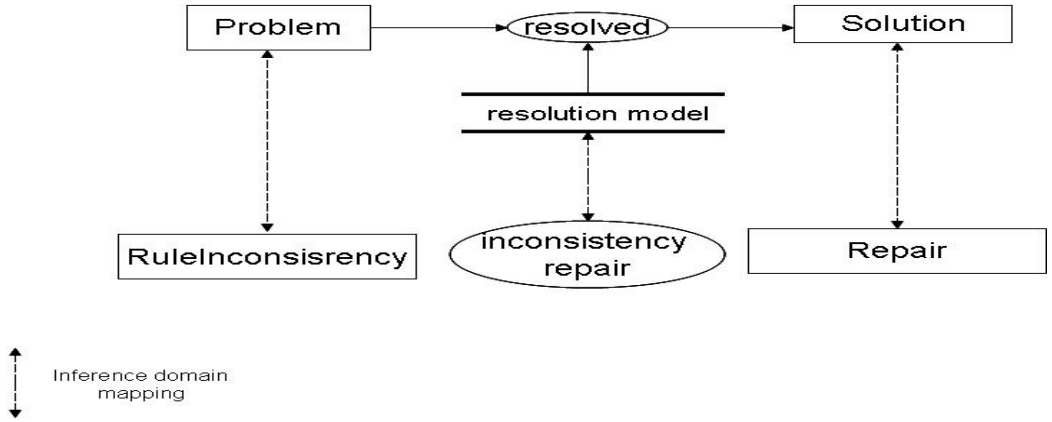

FIGURE 3.7 – Modèle de réparation.

informations entre les agents de raisonnement du domaine de connaissance et le monde externe. Elles sont au nombre de quatre :

Obtain – l'agent de raisonnement demande des informations à un agent externe. L'agent de raisonnement prend l'initiative et l'agent externe détient l'information ;

Receive – l'agent de raisonnement reçoit des informations de l'agent externe. L'agent externe prend l'initiative et détient l'information ;

Present – l'agent de raisonnement présente des informations à l'agent externe. L'agent de raisonnement prend l'initiative et détient l'information ;

Provide – le système fournit les informations à l'agent externe. L'agent externe prend l'initiative et l'agent de raisonnement détient les informations.

La Figure 3.8 décrit la structure d'inférences de notre système. Le système reçoit en entrée une structure de changement, ce qui correspond à l'entité *Change* du modèle de connaissance. L'inférence *cause* est déclenchée pour détecter d'éventuels problèmes. Si aucun problème n'est détecté alors le changement est appliqué à l'ontologie et le processus de raisonnement est terminé. Sinon, si le changement génère des problèmes alors l'inférence *repaired* est déclenchée pour fournir à l'utilisateur un ensemble de solutions. Une solution peut être soit un changement alternatif sur l'ontologie soit un changement sur les règles. Dans les deux cas, la solution choisie par l'utilisateur est considérée comme un changement et le processus de raisonnement est relancé pour vérifier si la solution ne va pas causer d'autres types de problèmes.

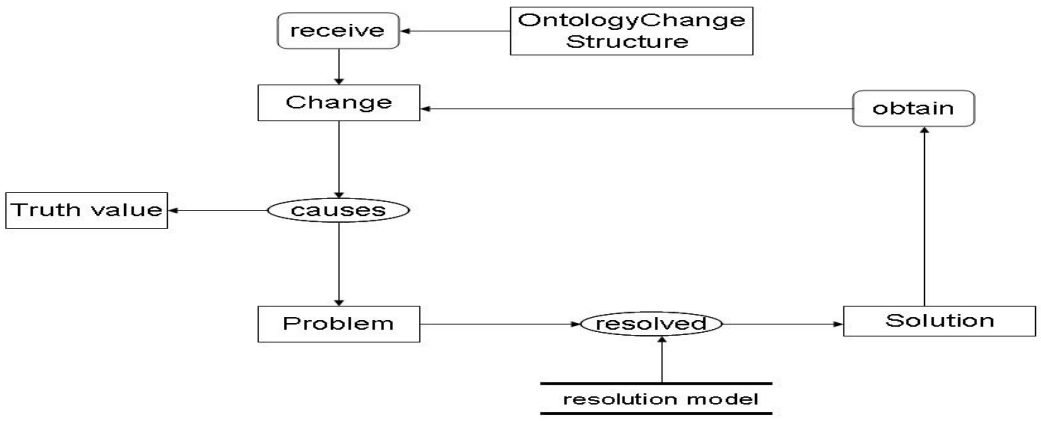

FIGURE 3.8 – Structure d'inférence.

3.2.3 Les connaissances de tâche

Les connaissances de tâche sont la catégorie de connaissance qui permet de décrire les objectifs du système de connaissance et la stratégie à suivre pour atteindre ces objectifs.

La notation graphique utilisé dans CommonKADS pour représenter les connaissances de tâche est la suivante : les rectangles avec les angles arrondis

représentent la tâche à accomplir et les rectangles représentent la méthode utilisée pour l'accomplir. Le niveau le plus bas du graphe représente les inférences et les fonctions de transfert utilisées qui composent cette méthode.

L'objectif principal de notre système est de maintenir la cohérence des règles entre elles et par rapport à l'évolution de l'ontologie. Pour cela, nous avons mis en place la stratégie suivante (voir Figure 3.9) :

1. capture du changement de l'ontologie : le système détecte les changements effectués dans l'ontologie ;
2. détection de son impact sur les règles : l'agent de raisonnement *cause* détecte les problèmes qui peuvent être causés par le changement ;
3. proposition des solutions : l'agent de raisonnement *repair* propose des solutions pour réparer les problèmes détectés.

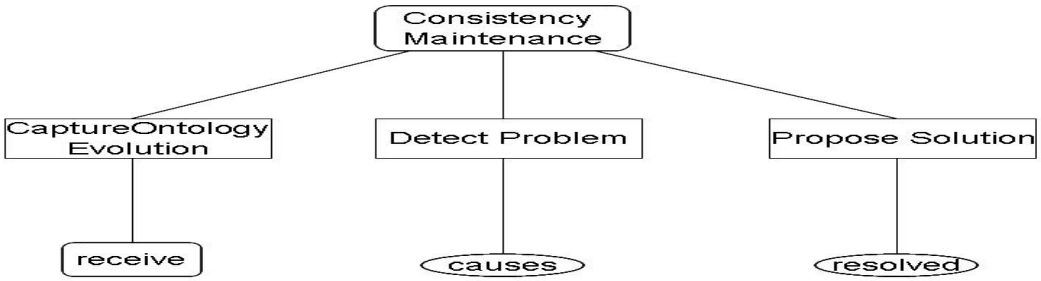

FIGURE 3.9 – Modèle de tâche.

3.3 Les changements d'ontologie et leurs impacts sur les règles

Différents travaux dans la littérature se sont focalisés sur les changements des ontologies dans le but d'analyser leurs impacts sur la cohérence de l'ontologie même (voir Section 2.4). Dans ce qui suit, nous présentons une classification des changements des ontologies en fonction de leur impact sur les règles. Ensuite, nous présentons une analyse de cet impact sur les règles.

3.3.1 Les changements d'ontologies

Dans notre travail, nous distinguons 3 catégories de changements d'ontologies :

- le changement de structure : consiste à ajouter ou supprimer des entités (concepts ou propriétés) de l'ontologie ;
- le renommage des entités : consiste à modifier le nom d'une entité ;
- le changement des définitions : consiste à changer la définition donnée à un constructeur en modifiant ses arguments. Ce type de changement, que nous considérons le plus complexe est défini en six catégories de changement, se fondant sur la classification du W3C [3]. Ces six catégories sont :
  - changement de l'axiomatique des concepts : consiste à modifier les axiomes qui définissent les concepts de l'ontologie ; changer les relations de sous-classe, d'équivalence et de disjonction ;
  - changement de la description des classes : une classe peut être définie soit par son nom soit par une extension de classes regroupant un ensemble de classes définies ou anonymes. Dans le premier cas, le changement est un renommage d'entité (catégorie de changement citée précédemment). Dans le deuxième cas, le changement consiste à modifier les classes énumérées, les intersections et unions de classes ou les classes complémentaires ;
  - changement des caractéristiques des propriétés : une propriété OWL peut être définie comme étant fonctionnelle, inversement fonctionnelle, transitive ou symétrique. Le changement des caractéristiques consiste à modifier les propriétés précédemment citées ;
  - changement des relations entre les propriétés : consiste à modifier les relations d'équivalence ou d'inverse entre les propriétés ;
  - changement des restrictions des propriétés : consiste à modifier les restrictions sur les cardinalités ou les restrictions sur les valeurs de propriétés ;
  - changement des définitions des propriétés : consiste à modifier le domaine ou le co-domaine d'une propriété.

3. http ://www.w3.org/TR/owl-ref/

3.3.2 Problèmes de cohérence des règles

Les travaux traitant des problèmes de cohérence des règles ne sont pas très nombreux dans la littérature. Dans ce qui suit, nous présentons une taxonomie des ces problèmes en se basant sur les travaux effectués dans (Berstel & M.Leconte, 2010) et (Fink *et al.*, 2011). Ainsi, nous définissons trois catégories de problèmes de cohérence des règles :

1. la contradiction – une contradiction est détectée dans un ensemble de règles R, si cet ensemble contient au minimum deux règles qui ont la même partie condition mais qui assignent dans leur partie action deux différentes valeurs à un même attribut.

   Exemple

r_1 : *SI le dosage du médicament est supérieure à 2* *ALORS la prescription est valide*	r_2 : *SI le dosage du médicament est supérieure à 2* *ALORS la prescription n'est pas valide*

2. la pertinence – nous distinguons trois types problèmes de cohérence qui impactent la pertinence d'un ensemble de règles.
   (a) les règles jamais applicables – une règle n'est jamais applicable si sa partie condition ne peut jamais être vérifiée ;

   Exemple

   r_3 : *SI l'âge du patient est supérieur à 200* *ALORS annuler la prescription du médicament*

   (b) la violation de domaine – une violation de domaine est détectée dans un ensemble de règles, si cet ensemble contient au moins une règle qui, dans sa partie action, assigne à un attribut une valeur hors de son domaine ;
   (c) les règles non-valides – une règle est dite non-valide si cette dernière utilise, dans sa prémisse ou dans partie conclusion, un concept ou une propriété qui ne font pas partie du modèle à partir duquel les règles ont été éditées ;

3. la redondance – deux règles sont dites redondantes si la condition d'une des règles est incluse dans la partie condition de l'autre et que les deux règles produisent la même action.

   Exemple

r_1 : *SI le dosage du médicament est compris entre 2 et 10* *ALORS la prescription est valide*	r_2 : *SI le dosage du médicament est compris entre 4 et 9* *ALORS la prescription est valide*

3.3.3 Impacts des changements d'ontologies sur les règles

Afin de détecter l'impact des changements d'ontologies sur les règles, nous avons commencé par définir la liste des problèmes de cohérence qui peuvent impacter les règles. Ensuite, nous avons effectué une étude qui permet de déterminer pour chaque changement d'ontologie quel est le problème de cohérence qu'il peut causer.

Nous avons réalisé que certains changements n'ont aucun impact sur les règles, tels que ajouter ou supprimer une disjonction, le changement de certaines caractéristiques de propriétés (la transitivité, la symétrie), ajout de concepts ou de propriétés,... Ci-dessous, nous présentons les principaux changements d'ontologies sur lesquels nous avons travaillé ainsi que les problèmes de cohérences qui peuvent être générés par chaque changement.

1. changement de la relation de sous-classe – la suppression d'une relation de sous-classe entre deux concepts A et B tels que

   $$\texttt{B} \sqsubseteq \texttt{A et p(A, C)}$$

   dans ce cas, le concept B hérite la propriété p. Ce changement peut causer un problème de type *règles invalides* s'il existe une règle qui utilise, soit dans sa partie condition soit dans sa partie action, le concept B avec la propriété P. Les changements du domaine d'une propriété, d'intersection et de l'union suivent le même raisonnement et peuvent causer le même type de problème (voir Figure 3.10).

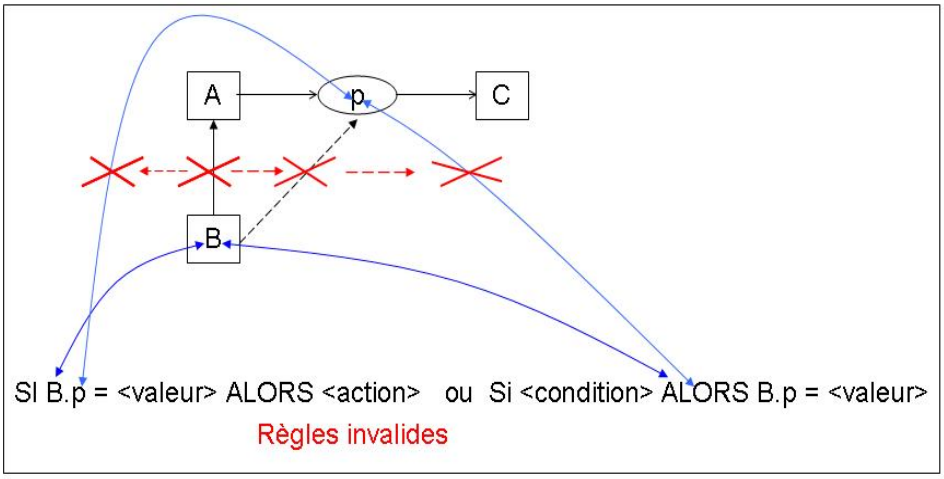

FIGURE 3.10 – Impact du changement de sous-classe.

2. changement d'une énumération – l'ajout d'une énumération ou la suppression d'un élément de l'énumération peut causer deux types de problèmes. Le premier est, si dans l'ensemble des règles, il y a une règle qui teste sur un élément qui n'est plus/pas dans la nouvelle énumération, dans ce cas cette règle ne sera jamais applicable car sa permisse ne peut plus être vérifié après l'application du changement. Le deuxième problème est la *violation de domaine*, qui est détectée si une règle assigne à une propriété, dans sa partie action, une valeur hors du domaine de l'énumération. Le changement de la restriction `owl:hasValue` suit le même raisonnement et peut causer les mêmes types d'incohérences (voir Figure 3.11).

3. changement du co-domaine d'une propriété – la modification du co-domaine d'une propriété peut causer un problème de cohérence dans le cas où le nouveau co-domaine de la propriété n'est pas une sous-classe de l'ancien co-domaine. Ce changement cause une incohérence de type *règle invalide*. La modification d'une restriction `owl:allValuesFrom` suit le même raisonnement et peut causer le même type d'incohérence (voir Figure 3.12).

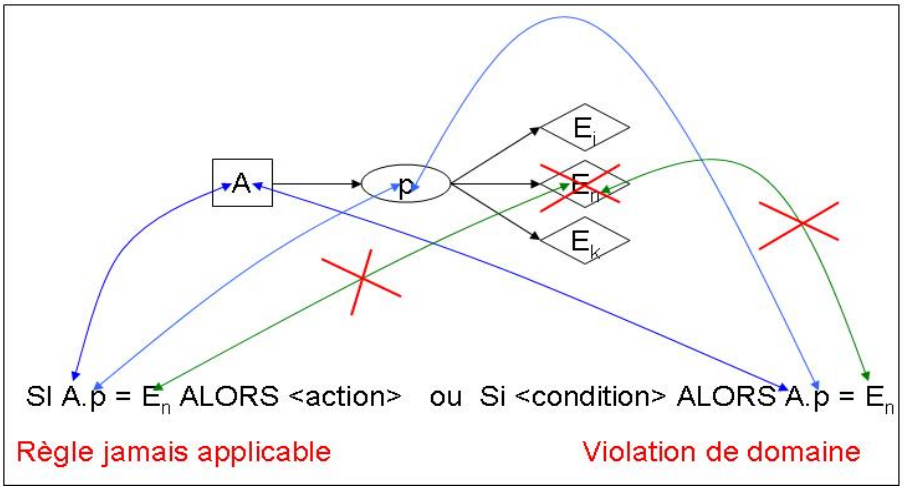

FIGURE 3.11 – Impact du changement d'énumération.

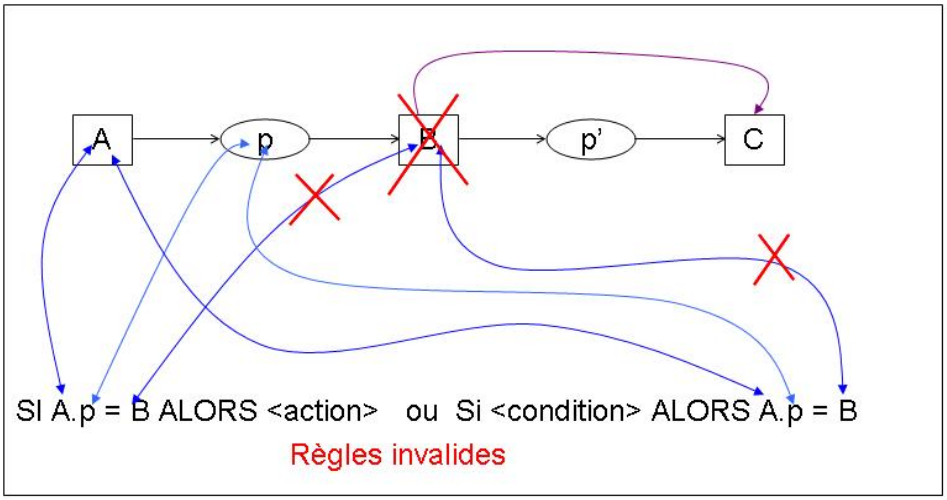

FIGURE 3.12 – Impact du changement du co-domaine.

3.4 Patrons de gestion des changements

Les patrons de gestion des changements (voir Section 3.4) ont été proposés par (Djedidi & Aufaure, 2010) pour guider l'évolution des ontologies tout en maintenant leur cohérence. Dans ce travail, nous proposons une adaptation de ce type de patrons pour gérer l'impact de l'évolution d'une ontologie sur les règles qui en dépendent. L'approche que nous proposons, $\mathcal{MDR}$ ($\mathcal{M}$odéliser - $\mathcal{D}$étecter - $\mathcal{R}$éparer) est une approche à base de patrons de gestion des changements. Ce type de patron consiste en trois catégories de patrons ; les patrons de changement, les patrons d'incohérence et les patrons de réparation. Les patrons de changement permettent de modéliser un changement donné, les patrons d'incohérence permettent de détecter les incohérences et les patrons de réparation permettent de proposer les réparations.

Dans notre approche, les patrons de changement sont modélisés avec une ontologie OWL que nous avons appelé l'ontologie $\mathcal{MDR}$. Cette ontologie modélise les connaissances nécessaires au bon fonctionnement de notre système. Les patrons d'incohérence et de réparation sont modélisés avec des règles. Ces règles sont définies en se fondant sur l'ontologie $\mathcal{MDR}$. En fonction du changement détecté dans l'ontologie métier, elles permettent de détecter des incohérences sur les règles métier et de proposer des solution pour les réparer.

Afin d'éviter toute confusion, il est important de noter que nous parlons de deux types d'ontologies et de deux types de règles. L'approche $\mathcal{MDR}$ est implémentée à base de l'ontologie $\mathcal{MDR}$, des règles de détection des incohérences et des règles de réparation. Cette approche analyse l'impact de l'évolution de l'ontologie métier sur les règles métier. Nous appellerons donc ontologie métier l'ontologie qui modélise les entités et les relations entre les entités du métier.

3.4.1 Patron de changement : l'ontologie $\mathcal{MDR}$

L'ontologie $\mathcal{MDR}$ modélise les connaissances nécessaires pour gérer l'impact des changements d'ontologies sur les règles. Elle comprend 100 concepts et 74 propriétés. Ces entités permettent de modéliser un changement d'ontologie, les règles qui sont impactées, les problèmes causés et les réparations à appliquer.

Il est à noter que la notation graphique utilisée dans les figures de cette

section n'est pas une notation standard. Les concepts de l'ontologie sont représentés par des rectangles, les propriétés par des cercles ovales et les types des propriétés de données par des hexagones irréguliers. Les flèches avec une extrémité épaisse représentent la relation de subsomption et les flèches avec une extrémité fine représentent la relation *propriété (domaine, co-domaine)*. Les valeurs de restriction sur les propriétés de type de données sont représentées par un rectangle aux bouts arrondis.

La définition d'un changement que nous proposons et que nous modélisons dans notre ontologie est la suivante :

Un changement s'applique sur une entité que nous appelons l'entité de changement, a un objet de changement qui est le constructeur OWL impacté par le changement, impacte des règles et a pour impacte une incohérence qui est réparée par une réparation (voir Figure 3.13).

Les principaux concepts pour modéliser un changement sont :

- `OWLConstruct` : décrit l'ensemble des constructeurs OWL ;
- `Change` : décrit l'ensemble des changements qui peuvent impacter une ontologie ;
- `Entity` : décrit l'entité de l'ontologie à laquelle s'applique l'ontologie ;
- `Rule` : décrit les règles métier éditées à partir d'une ontologie ;
- `Inconsistency` : décrit l'ensemble des incohérences qui peuvent impacter des règles ;
- `Repair` : décrit l'ensemble des réparations qui permettent de résoudre une incohérence due à un changement.

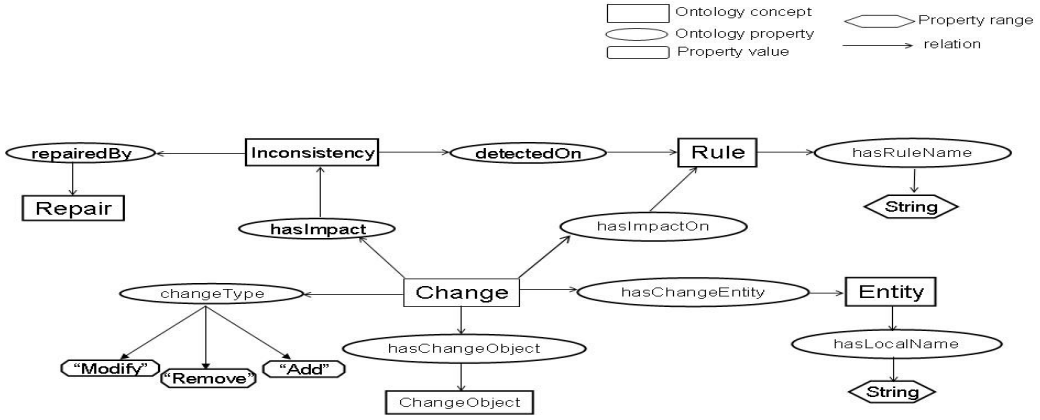

FIGURE 3.13 – Structure des changements.

Dans ce qui suit, nous donnons une description plus détaillée de chacun des concepts présentés ci-dessus.

Le concept `OWLConstruct` – Il représente l'ensemble des constructeurs OWL-DL. La classification des constructeurs est effectuée sur le même principe que la classification des changements des définitions des constructeurs (*i.e.* le concept `OWLConstructChange`) (voir Table 3.1).

Le concept `Change` Les changements d'ontologies sont classifiés en trois catégories représentées par des concepts qui sont définis par d'autres concepts plus spécifiquesvoir Figure 3.14).

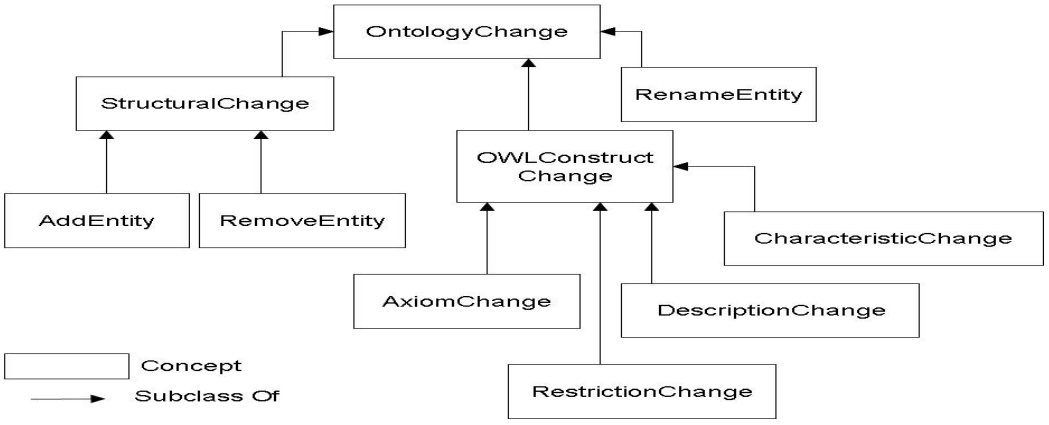

FIGURE 3.14 – Catégories des changements.

Les trois concepts majeurs qui représentent les catégories des changements sont :

- `StructuralChange` : décrivent les changements qui consistent à ajouter ou supprimer des entités de l'ontologie. Ce concept contient donc deux sous concepts, `AddEntity` et `RemoveEntity`, qui eux même ont deux sous concepts, `AddConcept`, `AddProperty`, `RemoveConcept` et `RemoveProperty` (voir Figure 3.15).
- `RenameEntity` : décrit les changements qui consistent à renommer une entité de l'ontologie, ce qui revient à modifier le *fragURI* ;
- `OWLConstructChange` : décrivent les changements qui ont pour objet un constructeur OWL. Ce concept contient les quatre sous-concepts suivants :

- `ClassAxiomChange` : décrit les changements qui impactent les axiomes qui définissent un concept (la subsomption, l'équivalence, la disjonction...) ;
- `ClassDescriptionChange` : décrit des changements qui impactent la description d'un concept (l'énumération, l'intersection, l'union,...) ;
- `PropertyRestrictionChange` : décrit les changements qui impactent les restrictions sur les propriétés (restriction de valeur, restriction du type, restriction de la cardinalité ...) ;
- `PropertyCharacteristicChange` : décrit les changements qui impactent les caractéristiques des propriétés (la transitivité, la symétrie, la fonctionnalité...) ;
- `PropertyRelationChange` : représente les changements qui impactent les relations entre les propriétés (l'équivalent et l'inverse).

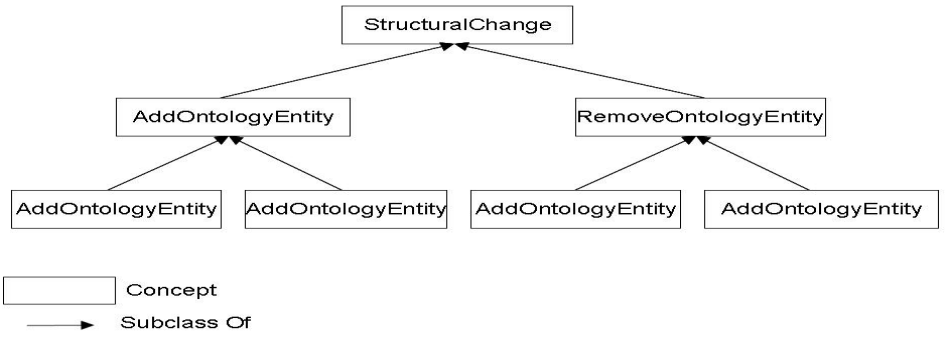

FIGURE 3.15 – Les changements de structure.

Ci-dessous, un tableau récapitulatif des changements qui ont pour objet un constructeur OWL (*i.e.* OWLConstructChange). Il décrit les différentes catégories de ce type de changement ainsi que les constructeurs OWL impactés (voir Table 3.1).

Change Category	Change	Impacted OWL Construct
ClassAxiomChange	SubClassChange	owl:subClassOf
	EquivalentClassChange	owl:equivalentClass
	DisjointClassChange	owl:disjointWith
ClassDescriptionChange	ComplementOfChange	owl:complementOf
	EnumerationChange	owl:oneOf
	IntersectionChange	owl:intersectionOf
	UnionChange	owl:unionOf
PropertyCharacteristicChange	FunctionalPropertyChange	owl:FunctionalProperty
	InverseFunctionalPropertyChange	owl:InverseFunctionalProperty
	SymmetricCaracteristicChange	owl:SymmetricProperty
	TransitiveCaracteristicChange	owl:TransitiveProperty
PropertyRelationChange	EquivalentPropertyChange	owl:equivalentProperty
	InverseOfPropertyChange	owl:inverseOf
PropertyRestrictionChange	CardinalityRestrictionChange	CardinalityChange	owl:cardinality
		MaxCardinalityChange	owl:maxCardinality
		MinCardinalityChange	owl:minCardinality
	ValueConstraintChange	AllValuesFromRestrictionchange	owl:allValuesFrom
		HasValueRestrictionChange	owl:hasValue
		SomeValuesFromRestrictionChange	owl:someValuesFrom

TABLE 3.1 - Les sous-concepts du concept OWLConstructChange.

Le concept `Entity` – Ce concept représente les entités qui composent une ontologie OWL soit `Concept`, `Property` et `Individual`. Comme il existe deux types de propriétés, le concept `Property` possède donc deux sous-concepts qui sont `ObjectProperty` et `DataTypeProperty`(voir Figure 3.16). Une `Entity` est toujours identifiée par son *namespace* et son nom local. Dans les limites de notre travail, un changement s'applique toujours soit à un concept soit à une propriété. Nous ne gérons pas l'impact des changements sur les individus d'une ontologie.

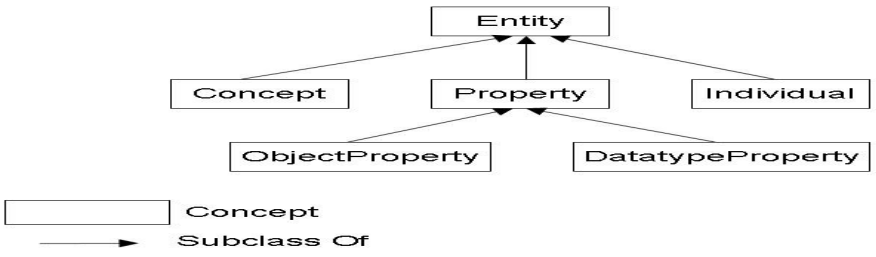

FIGURE 3.16 – Hiérarchie du concept `Entity`.

Le concept `Rule` – Une règle est composée d'une partie condition `ConditionPart`, elle même composée d'un ensemble de conditions `Condition` et une partie action `ActionPart`, composée d'un ensemble d'actions `Action`. Les règles sont écrites à partir des entités `Entity` de l'ontologie (voir Figure 3.17).

le concept `Inconsistency` – Nous distinguons quatre types d'incohérences qui peuvent impacter un ensemble de règles : `DomainViolation`, `RuleNeverApply`, `InvalidRule` and `Contradiction` (voir Section 3.3.2).

le concept `Repair` – Il représente l'ensemble des solutions qui permettent de résoudre une incohérence. Dans l'ontologie $\mathcal{MDR}$ nous modélisons cinq types de *Repair* :

- `AddAttributeToClass` : permet de rajouter une propriété à un concept ;
- `AddValueToEnumeration` : permet d'ajouter une valeur dans une énumération d'un concept de l'ontologie ;

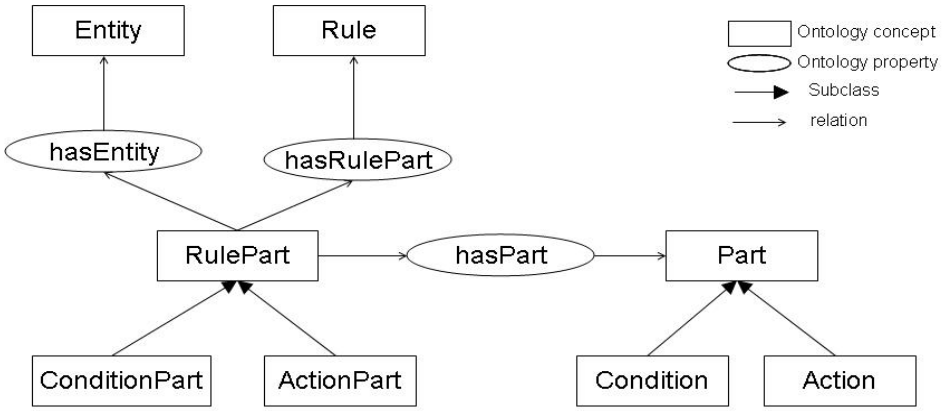

FIGURE 3.17 – Construction du concept `Rule`.

- `ChangePropertyValue` : permet de changer la restriction sur la valeur d'une propriété ;
- `ChangeTestedValue` : permet de changer la valeur de test d'un attribut dans la partie condition d'une règle ;
- `RemoveCondition` : permet de supprimer la condition d'une règle qui cause une incohérence.

3.4.2 Patron d'incohérence : les règles de détection des incohérences

La modélisation de patrons d'incohérence permet de définir les incohérences qu'un changement peut causer. Pour définir l'impact d'un changement d'ontologie sur les règles, trois paramètres sont nécessaires à représenter dans un patron d'incohérence : l'objet du changement, le type du changement (voir Figure 3.13) et la contrainte du changement. Les contraintes de changement définissent les conditions nécessaires et suffisantes qu'un changement doit vérifier afin d'éviter les incohérences. En fonction de ces trois paramètres, le type de l'incohérence est défini.

La détection d'une incohérence consiste en quelque sorte à une décision prise en fonction du changement, de la contrainte et de l'ensemble des incohérences de règles. Pour cela, nous avons modélisé les patrons d'incohérence avec des règles que nous appelons *règles de détection d'incohérences*. Dans la partie condition, ces règles testent l'objet, le type du changement et si la

contrainte que doit vérifier le changement est vérifiée ou pas. Dans le cas où la contrainte n'est pas vérifiée, le type de l'incohérence causée par le changement est assigné.

Les contraintes de changement sont définies principalement par rapport à l'entité de l'ontologie métier impactée par le changement et aux règles métier utilisant cette entité. Dans certains cas et en fonction de l'objet du changement, il est aussi nécessaire de spécifier le partie de la règle qui utilise cette entité.

Ci-dessous, la formalisation générale d'un patron d'incohérence :

```
IF change.changeObject = changeObject
&& change.type = changeType
&& changeConstraint.satisfied = false
THEN change.inconsistency = inconsistency;
```

3.4.3 Patron de réparation : les règles de réparation

La modélisation de patrons de réparation permet de définir les différentes solutions que le système présente à l'utilisateur pour résoudre une incohérence. Pour définir le type de réparation à proposer, deux paramètres sont nécessaires à représenter dans un patron de réparation : l'objet du changement et le type de l'incohérence détectée pour les patrons d'incohérence. Les patrons de réparation sont aussi modélisés avec des règles, les *règles de réparation* qui, dans leur partie condition testent l'objet du changement et l'incohérence détectée et, dans leur partie action, assignent une réparation.

Ci-dessous, la formalisation générale d'un patron de réparation :

```
IF change.changeObject = changeObject
&& change.inconsistency = inconsistency
THEN inconsistency.repair = inconsistencyRepair;
```

3.5 Architecture et fonctionnement

3.5.1 Architecture

Comme nous l'avons décrit dans la section précédente, le composants majeurs de notre approche sont les patrons de gestion de changements. Néan-

moins, jusqu'à maintenant nous n'avons pas explicité le lien entre ces différents patrons ni comment ils sont mis en œuvre pour le fonctionnement du système (voir Figure 3.18).

Un changement est modélisé comme étant un individu de l'ontologie $\mathcal{MDR}$ et représente un changement sur l'ontologie métier. Les patrons d'incohérence et de réparation sont spécifiés avec des règles qui sont écrites en se fondant sur l'ontologie $\mathcal{MDR}$ et qui sont exécutées en fonction des individus de l'ontologie (voir Section 4.2). En fonction du changement modélisé, une ou plusieurs *règles de détection des incohérences* peuvent être déclenchées pour détecter les problèmes de cohérence causés par le changement et impactant les règles métier. De même, en fonction du changement et du problème de cohérence détecté, les *les règles de réparation* sont déclenchées pour proposer des réparations.

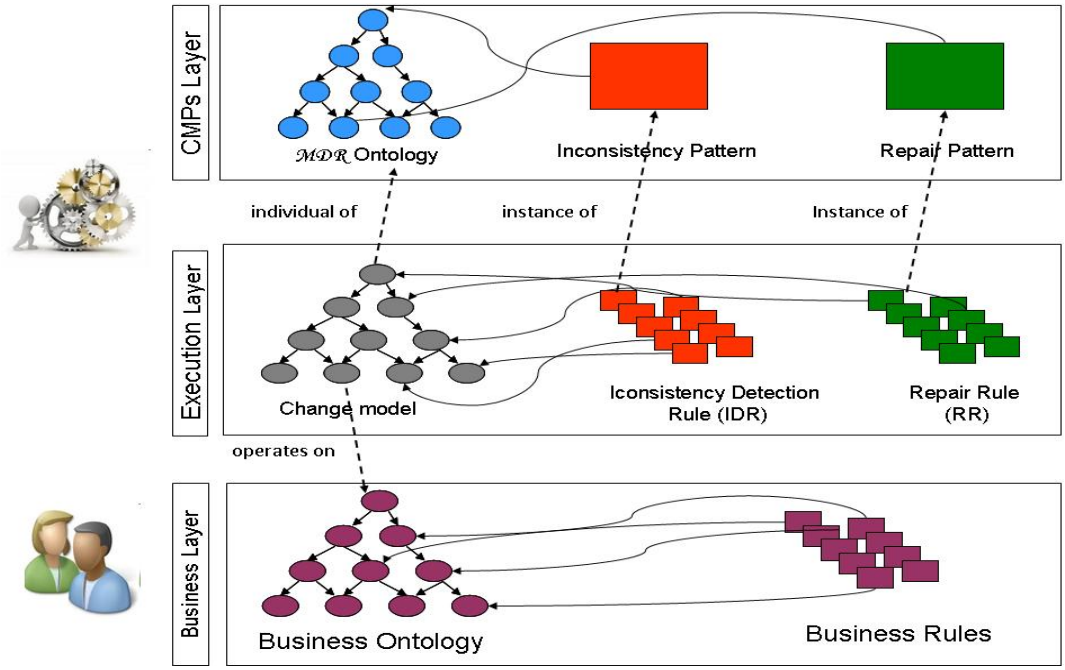

FIGURE 3.18 – Architecture $\mathcal{MDR}$.

3.5.2 Fonctionnement

La construction du modèle de changement présentée dans la Figure 3.13 est effectuée en quatre étapes. Ces étapes représentent le processus de gestion de l'impact des changements d'ontologies sur les règles (voir Figure 3.19).

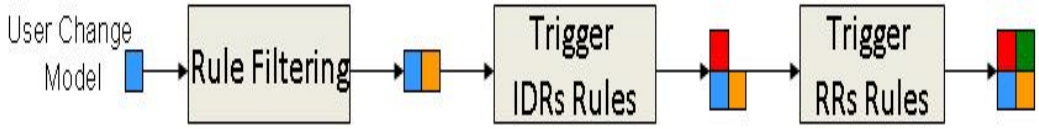

FIGURE 3.19 – Processus de gestion de l'impact des changements d'ontologies sur les règles.

Ces quatre étapes consistent en (1) la spécification du changement à appliquer à l'ontologie métier à l'aide du patron de changement, (2) la détection des règles impactées par le changement, (3) la détection des problèmes de cohérence par l'application des *règles de détection des incohérences* (*i.e.* patrons d'incohérence) et enfin (4) la proposition des réparations par l'application des *règles de réparation* (*i.e.* les patrons de réparation).

La syntaxe utilisée pour décrire les *règles de détection des incohérences* et les *règles de réparation* est fondée sur la syntaxe du langage de règles ILOG Rule Langage (IRL) [4]. La partie `when` décrit la condition de la règle et la partie `then` décrit l'action de la règle.

3.5.2.1 Spécification du changement

C'est la première étape du processus de gestion de l'impact de l'évolution des ontologies sur les règles. Lors de cette étape, l'utilisateur fournit en entrée au système le changement qu'il veut appliquer à l'ontologie métier. Ce changement doit être formalisé comme étant un individu de l'ontologie $\mathcal{MDR}$ et consiste à fournir les informations suivantes (voir Figure 3.20) :

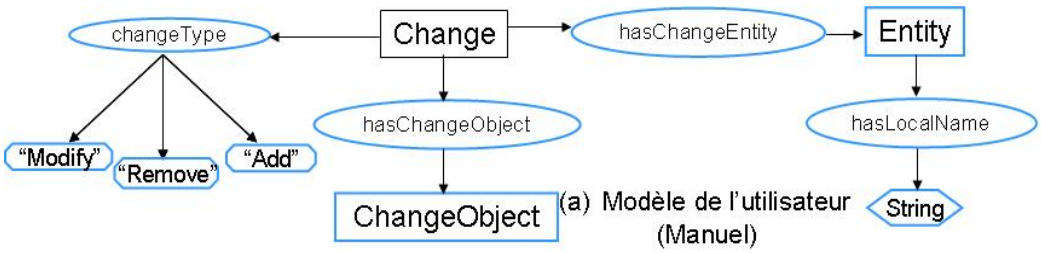

FIGURE 3.20 – Modèle d'entrée du système.

4. http://pic.dhe.ibm.com/infocenter/brjrules/v7r1/index.jsp?topic=%2Fcom.ibm.websphere.ilog.jrules.doc%2Fhtml%2Fapi%2Fhtml%2Filog%2Frules%2Ffactory%2Fpackage-summary.html

- l'entité (*i.e.* concept ou propriété) de l'ontologie métier à laquelle le changement va s'appliquer ainsi que son nom ;
- l'objet du changement, *i.e.* le constructeur OWL impacté par le changement ;
- le type de changement, ce qui consiste en soit ajouter, modifier ou supprimer l'objet du changement.

Le modèle fourni en entrée au système (voir Figure 3.20) représente les informations communes à tous les changements. Néanmoins, en fonction de l'objet du changement, d'autres informations doivent être modélisées afin d'assurer la détection des incohérences. Pour cela, chaque changement modélisé dans l'ontologie $\mathcal{MDR}$ comme sous-classe de `Change` possède son propre patron de changement. Dans ce qui suit, nous donnons la modélisation de chaque patron de changement.

Patron de changement de sous-classe Un changement de sous-classe consiste à ajouter, modifier ou supprimer la relation de sous-classe entre un concept et son super-concept . Ce concept est donc l'entité de changement représenté par le concept `Entity` dans la Figure 3.20. Dans le cas d'un ajout ou de suppression d'une relation de sous-classe, il faut mentionner respectivement le nom du nouveau super-concept et le nom du super-concept à supprimer. La modification d'une relation de sous-classe revient à supprimer cette relation entre le super-concept actuel et l'entité de changement et ensuite à ajouter une nouvelle relation de sous-classe (voir Figure 3.21).

Patron de changement d'énumération Un changement d'énumération consiste à ajouter, modifier ou supprimer une collection ou une liste. Dans OWL, une énumération est définie par le constructeur `owl:oneOf` comme étant une collection [5] ou une liste [6] d'un ensemble de valeurs que peut avoir un concept ou une propriété. Il y a deux possibilités de définir une énumération : lorsque l'énumération définit un concept alors les éléments de la collection définissent l'ensemble des instances de ce concept ; par contre, si l'énumération est défini sur une propriété alors les éléments de la liste définissent l'ensemble des valeurs que peut avoir cette propriété.

5. http ://www.w3.org/TR/owl-ref/#EnumeratedClass
6. http ://www.w3.org/TR/owl-ref/#EnumeratedDatatype

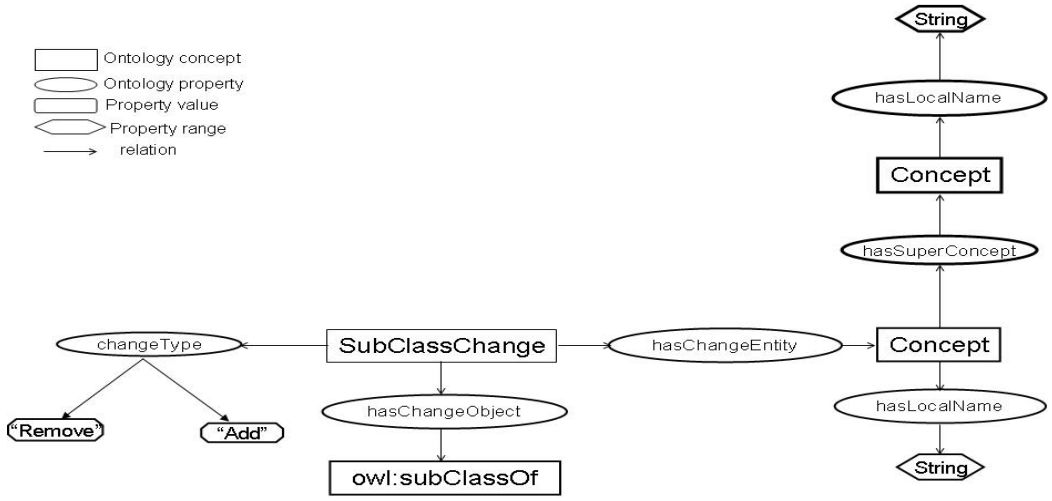

FIGURE 3.21 – Patron de changement de sous-classe.

La modélisation d'un patron de changement d'énumération dépend du type de changement. Si le changement consiste à ajouter une énumération à une entité alors il faut spécifier les éléments de l'énumération (voir Figure 3.22).

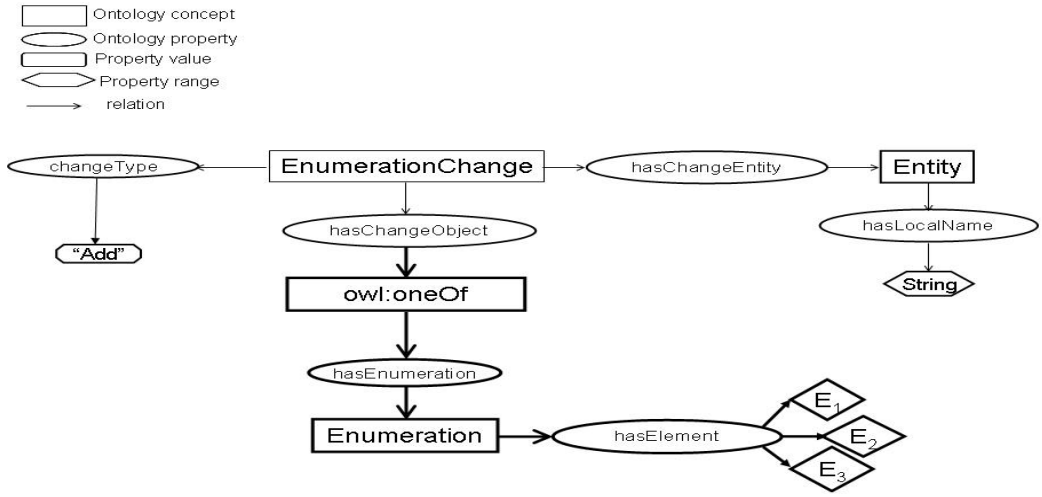

FIGURE 3.22 – Patron de changement d'énumération : ajouter une énumération.

Si le changement consiste à modifier une énumération alors il faut spécifier les éléments à ajouter ou à supprimer de l'énumération (voir Figure 3.23).

Et si le changement consiste à supprimer une énumération alors il suffit de fournir le modèle de changement basique en spécifiant le type de changement (voir

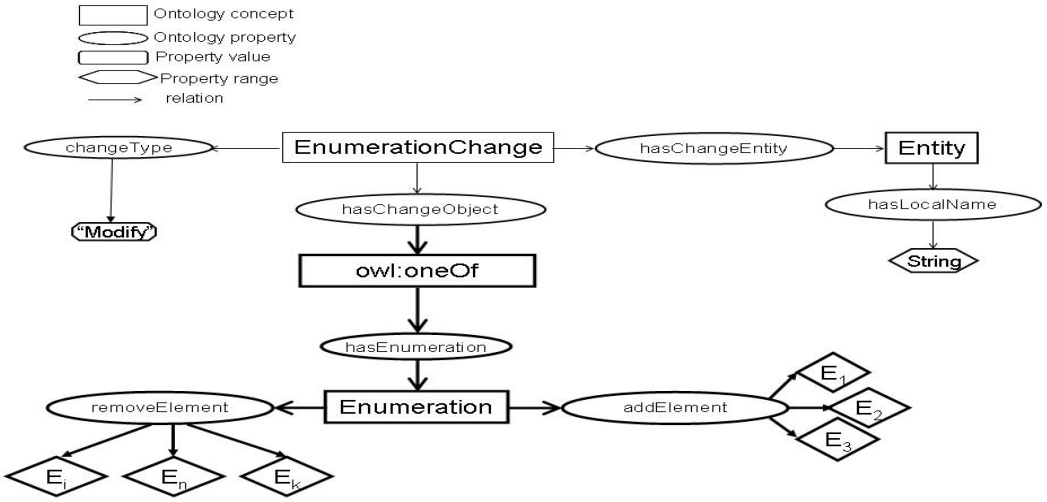

FIGURE 3.23 – Patron de changement d'énumération : modifier une énumération.

Figure 3.24).

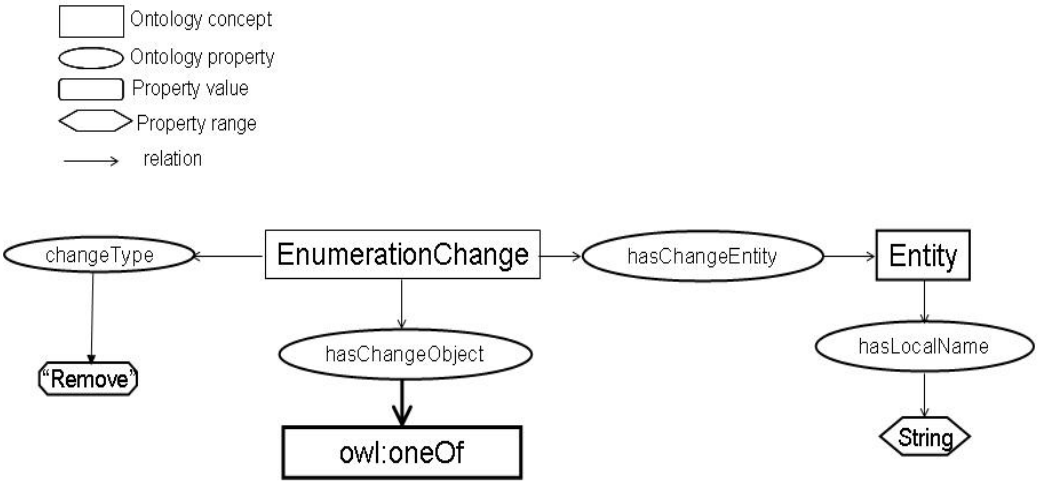

FIGURE 3.24 – Patron de changement d'énumération : supprimer une énumération.

Patron de changement d'ajout/suppression d'un concept La modélisation d'un patron de changement d'ajout/suppression d'un concept est relativement simple. Il suffit de spécifier le nom du concept à ajouter ou à supprimer et l'objet du changement `owl:Class` (voir Figure 3.25).

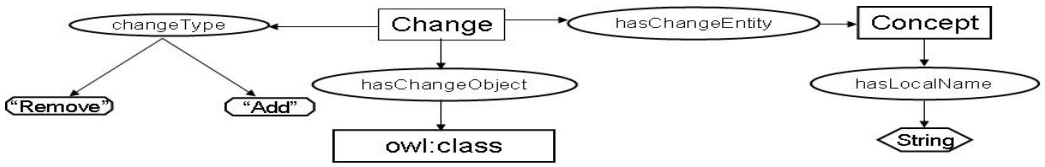

FIGURE 3.25 – Patron de changement de structure : ajouter/supprimer un concept.

Patron de changement de la restriction `AllValuesFrom` d'une propriété La changement de la restriction `AllValuesFrom` d'une propriété consiste à ajouter, supprimer ou modifier le concept de la restriction. L'entité de changement représentée dans le patron est la propriété sur laquelle porte la restriction. Dans le cas d'un ajout de cette restriction, il faut spécifier le concept de la restriction (voir Figure 3.26).

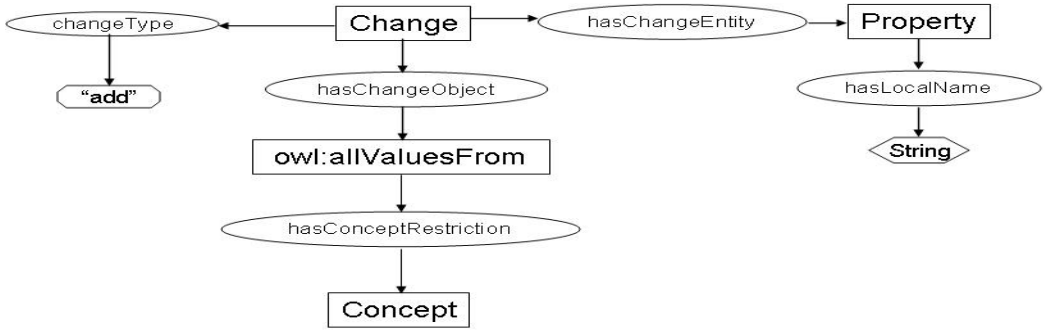

FIGURE 3.26 – Patron de changement de la restriction `AllValuesFrom` : ajouter une restriction `AllValuesFrom`.

Dans le cas de suppression de la restriction, il suffit de spécifier que le type du changement est `remove` et que l'objet du changement est le constructeur `owl:allValuesFrom` (voir Figure 3.27).

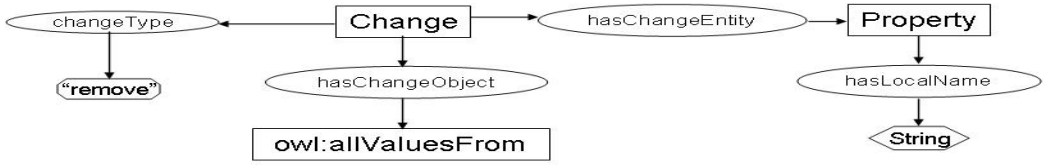

FIGURE 3.27 – Patron de changement de la restriction `AllValuesFrom` : supprimer une restriction `AllValuesFrom`.

Dans le cas d'une modification de cette restriction, il faut spécifier l'ancien et le nouveau concept de la restriction (voir Figure 3.28).

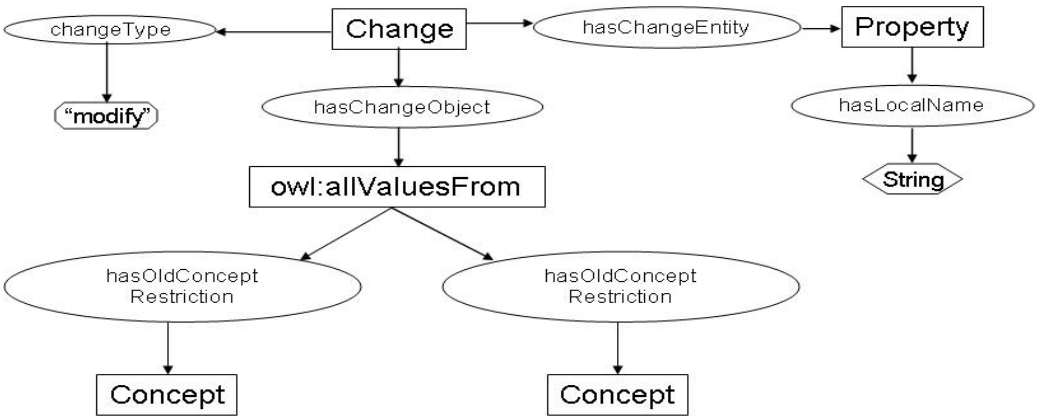

FIGURE 3.28 – Patron de changement de la restriction `AllValuesFrom` : modifier une restriction `AllValuesFrom`.

3.5.2.2 Détection des règles impactées

C'est la deuxième étape du processus de gestion de l'impact de l'évolution des ontologies sur les règles. Elle consiste à détecter l'ensemble des règles impactées par un changement. Généralement, les règles impactées sont détectées en fonction de l'entité et de l'objet du changement. Les règles qui utilisent l'entité du changement ne sont pas forcement impactées tout dépend de l'objet du changement.

La détection des règles impactées est effectuée en analysant l'arbre syntaxique de chaque règle métier écrite à partir de l'ontologie métier. L'ensemble des nœuds de l'arbre syntaxique d'une règle représente les opérateurs du langage de définition des règles et les feuilles représentent les différentes entités sur lesquelles agissent ces opérateurs. Ainsi, en navigant dans l'arbre syntaxique, nous pouvons détecter l'ensemble des entités utilisées dans une règle ainsi que les différents tests et actions effectués sur ces entités.

En fonction du changement à effectuer, certaines informations sont nécessaires pour savoir si une règle sera incohérente ou pas après l'application du changement. Par exemple, dans le cas d'un changement qui consiste à supprimer la relation de sous-classe entre deux concepts A et B (B `owl:subClassOf`A), s'il est nécessaire d'avoir l'ensemble des attributs utilisés dans une règle avec le concept B. Ou bien, dans le cas d'un changement qui consiste à supprimer un élément d'une énumération

alors il est nécessaire d'avoir l'ensemble des valeurs testées ou attribuées à l'entité du changement par une règle.

A la fin de cette étape, le modèle utilisateur, donné en entrée au système, est mis à jour avec l'ensemble des règles impactées (voir Figure 3.29).

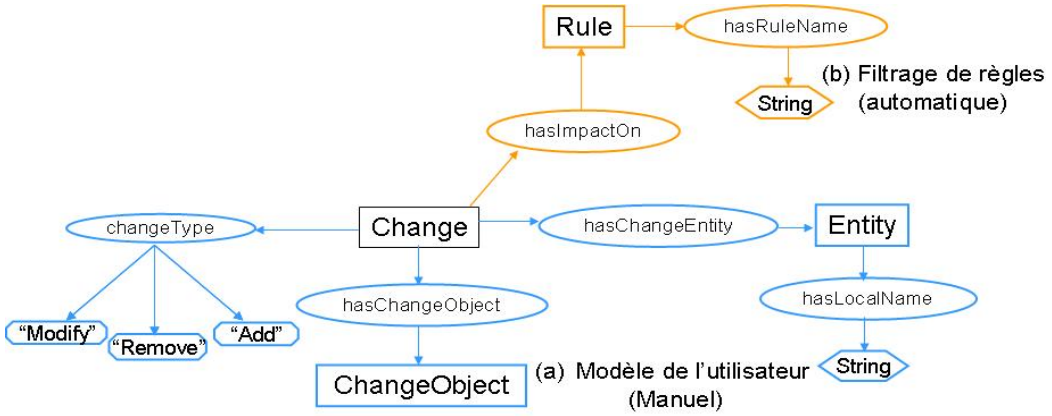

FIGURE 3.29 – Évolution du modèle de changement : détection des règles impactées.

3.5.2.3 Détection des problèmes de cohérence sur les règles

C'est la troisième étape du processus de gestion de l'impact de l'évolution des ontologies sur les règles. Cette étape consiste à détecter, à l'aide des *règles de détection des incohérences*, les problèmes de cohérence sur les règles qui peuvent être causées par un changement. La détection des incohérences dépend de trois facteurs, le changement spécifié dans la première étape, les règles impactées et, dans certains cas, la partie de la règle impactée (*i.e.* la partie condition ou action) et de la contrainte à vérifier. L'idée générale pour détecter les incohérences est : si le changement ne vérifie pas la contrainte et qu'il y a des règles qui sont impactées par ce changement alors les incohérences sont détectées sur les règles en question.

Les contraintes définissent les conditions qu'un changement doit respecter pour éviter les problèmes de cohérence dans l'ensemble des règles. Par example, dans le cas d'un changement qui consiste à supprimer une relation de sous-classe (B `owl:subClassOf` A) , la contrainte à vérifier est que les règles impactées n'utilisent pas l'entité du changement B avec une propriété héritée de la super-classe A car après l'application du changement cette propriété ne fera plus partie de l'ensemble des propriétés de B. Dans le cas où cette contrainte n'est pas vérifiée, une incohérence de type *règle invalide* sera détectée. La règle qui permet de détecter cette incohérence est présentée ci-dessous :

```
when {
subClassChange: SubClassChange();
   changeEntity : Concept() from subClassChange.changeEntity;
   superConcept : Concept() in changeEntity.superConcept ;
   br : subClassChange.impactOn;
   prop : Property() in superConcept.conceptProperty;

 evaluate (  changeEntity.ruleProperty.equals(prop)  );
}
then {
InvalidRule invalidRule = new InvalidRule();
subClassChange.add_hasImpact(invalidRule);
    invalidRule.add_detectedInRule(br);
}
```

Dans le cas d'un changement d'énumération qui consiste à modifier une énumération en supprimant certains de ses éléments, la contrainte que le changement doit vérifier est que les règles impactées n'utilisent pas l'entité de changement avec l'un des éléments à supprimer. Si cette contrainte n'est pas vérifiée alors deux types d'incohérences peuvent être détectées :

- si la règle utilise l'entité du changement dans sa partie condition et qu'elle effectue un test sur l'un des éléments à supprimer de l'énumération alors cette règle ne sera plus exécutée puisque sa partie condition ne pourra plus être vérifiées. La règle qui permet de détecter cette incohérence est la suivante :

```
when {
      enumerationChange : EnumerationChange();
       rule : Rule() in enumerationChange.impactOn;
       ruleProperty : Property();
       enumeration : Enumeration();
       conditionPart : ConditionPart();
       condition : Condition();

       evaluate (enumerationChange.changeType in {"Add", "Modify"}
          && enumerationChange.changeObject.equals(enumeration)
          && ruleProperty.hasOWLConstruct(enumeration)
          && enumerationChange.hasImpactOn(rule)
          && rule.hasRulePart(conditionPart)
          && conditionPart.hasPart(condition)
          && condition.hasEntity(ruleProperty)
          && ! (ruleProperty.testedPropertyValue in
                      enumerationChange.newCollection.element)  );
```

```
}
then {
       RuleNeverApplied rna = new RuleNeverApplied();
   enumerationChange.add_hasImpact(rna);
       rna.detectedOn(rule);
}
```

– si la règle utilise l'entité du changement dans sa partie action et qu'elle lui assigne un des éléments à supprimer de l'énumération, alors une incohérence de type *violation de domaine* sera détectée puisque la règle assigne une valeur, hors du domaine de l'entité du changement. La règle qui permet de détecter cette incohérence est la suivante :

```
 when {
enumerationChange: EnumerationChange();
ruleProperty:Property();
     enumeration: Enumeration();
      actionPart: ActionPart();
      action : Action();
      rule: Rule() in enumerationChange.impactOn;

      evaluate ( enumerationChange.changeType in {"Add", "Modify"}
         && enumerationChange.changeObject.equals(enumeration)
         && ruleProperty.hasOWLConstruct(enumeration)
         && enumerationChange.hasImpactOn(rule)
         && rule.hasRulePart(actionPart)
         && actionPart.hasPart(action)
         && action.hasEntity(ruleProperty)
         && ! (ruleProperty.assignedPropertyValue in
                     enumerationChange.newCollection.element) );
}
then {
       DomainViolation domainViolation = new DomainViolation();
       enumerationChange.add_hasImpact(domainViolation);
       domainViolation.detectedOn(rule);
}
```

Comme nous l'avons décrit précédemment, les *règles de détection des incohérences* sont éditées à partir de la T-Box de l'ontologie $\mathcal{MDR}$. L'exécution de ces règles est déclenchée en fonction de la A-Box qui représente les modèles de changement instanciés dans l'ontologie qui sont donnés en entrée au système. L'exécution des *règles de détection des incohérences* met à jour le modèle du changement en y

ajoutant le type de l'incohérence détectée (voir Figure 3.30).

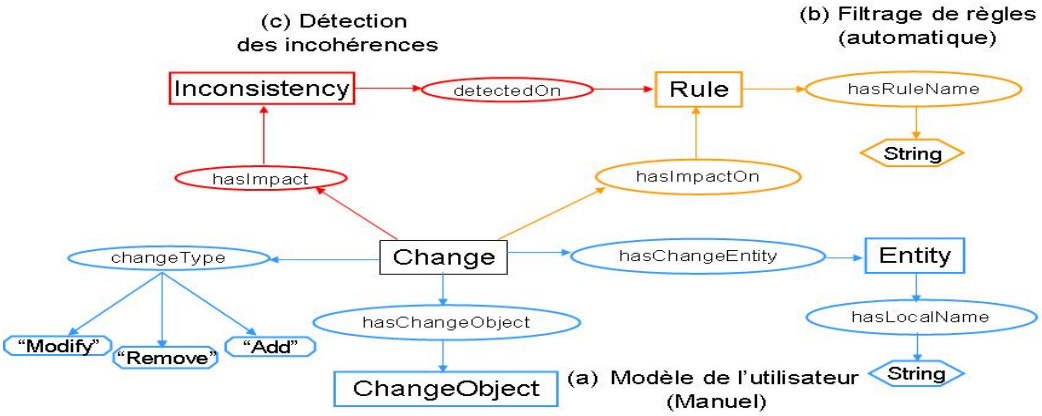

FIGURE 3.30 – Évolution du modèle de changement : détection des incohérences.

3.5.2.4 Proposition des réparations des incohérences

C'est la dernière étape du processus de gestion de l'impact des évolutions des ontologies sur les règles. Cette étape consiste à proposer, à l'aide des *règles de réparation* un ensemble de solutions pour réparer les incohérences détectées. La proposition des réparations est effectuée en fonction du changement à appliquer et de l'incohérence détectée. Néanmoins, il y a toujours des solutions par défaut qui sont proposées. Ces solutions sont soit supprimer la condition ou l'action de la règle où l'incohérence est détectée, soit supprimer la règle qui est incohérente. Par contre, il est toujours possible aux utilisateurs de décider d'appliquer un changement sans tenir compte des incohérences.

Par exemple, dans le cas d'un changement de sous-classe qui cause une incohérence de type *règle invalide*, comme décrit précédemment (voir Section 3.5.2.3), une des solutions proposées par le système est d'ajouter la propriété manquante (*i.e.* la propriété héritée) au concept du changement. Ainsi, l'utilisation de la propriété héritée avec le concept du changement ne sera plus incohérente.

La règle qui permet de proposer cette solution est donnée ci-dessous :

```
when {
   subClassChange: SubClassChange();
   changeEntity : Concept() from subClassChange.changeEntity;
   superConcept : Concept() in changeEntity.superConcept ;
   rule : Rule() in subClassChange.impactOn;
   property : Property() in superConcept.conceptProperty;
```

```
   invalidRule : InvalidRule() in subCassChange.impact ;

 evaluate (  changeEntity.ruleProperty.equals(property)  );
}
then {
   changeEntity.add_hasConceptProperty(property);
 }
```

Dans ce cas, quatre réparations sont proposées : (1) ajouter la propriété héritée au concept de changement, (2) supprimer la partie de la règle qui utilise le concept du changement avec la propriété héritée, (3) supprimer la règle impactée et (4) appliquer le changement sans tenir compte des incohérences. En fonction du choix de l'utilisateur, une *règle de réparation* est exécutée et le modèle de changement est mis à jour avec la réparation choisie par l'utilisateur (voir Figure 3.31).

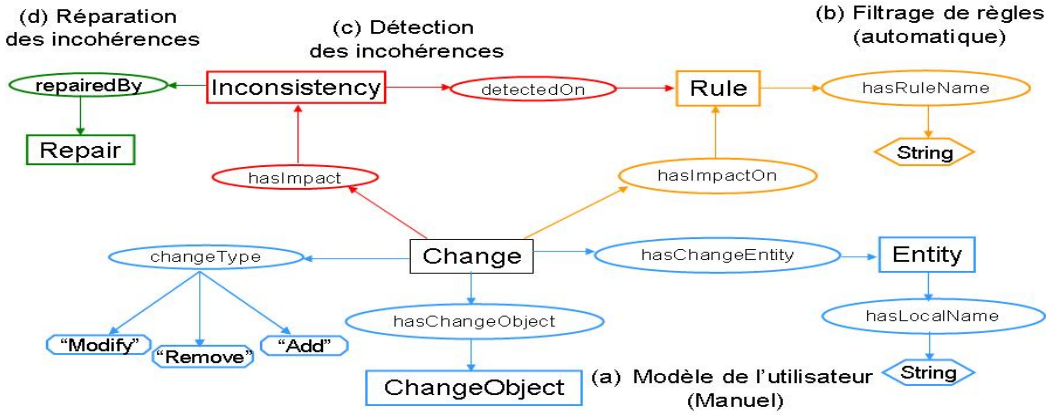

FIGURE 3.31 – Évolution du modèle de changement : proposition des réparations.

3.6 Langage de spécification $\mathcal{MDR}$

L'ontologie $\mathcal{MDR}$ présentée ci-dessus fournit une classification des changements d'ontologies et permet de représenter un changement, les règles qu'il impacte, les problèmes qu'il peut causer ainsi que les solutions pour les résoudre. Néanmoins, comme nous l'avons indiqué dans la Section 3.4, chaque changement doit vérifier des contraintes, si une contrainte n'est pas vérifiée alors le changement cause des problèmes de cohérence. Ces contraintes spécifiées par les patrons d'incohérences via des règles qui testent si les contraintes de changement sont vérifiées et qui détectent

le problème de cohérence causé dans le cas où le changement ne vérifie pas une des contraintes. Ce processus de test et de détection ne peut être modélisé par l'ontologie $\mathcal{MDR}$ et c'est la raison pour laquelle nous avons décidé de le modéliser via des règles. De même pour le processus de réparation, les patrons de réparation sont spécifiés via des règles qui en fonction du changement à appliquer et le problème détecté proposent un ensemble de réparation.

Afin d'avoir une représentation unifiée de l'approche $\mathcal{MDR}$, nous avons développé un langage, le langage $\mathcal{MDR}$ qui permet de représenter les différentes étapes de notre approche, à savoir, la représentation d'un changement, la définition des contraintes d'un changement, la spécification des problèmes causés par la violation d'une contrainte et la représentation des réparations.

Le langage proposé repose sur une syntaxe XML et utilise certains mots-clés de OWL pour représenter les relations ou les caractéristiques des entités. Afin d'éviter toute confusion, les mots-clés du langage proposé ont le préfixe `mdr`. Ainsi, la spécification $\mathcal{MDR}$ d'un changement est comme suit :

```
<mdr:change>
   <mdr:changeEntity  mdr:type = concept|property   mdr:localName = "name" />
   <mdr:changeObject> "owlConstruct" </mdr:changeObject>
  <mdr:changeType> add|remove|modify </mdr:changeType>
   <mdr:changeConstraint> changeConstraintID  </mdr:changeConstraint>
</mdr:change>
```

Les contraintes des changements `mdr:changeConstraint` sont spécifiées en deux niveaux, le premier permet de définir les entités qui entrent en jeu dans la définition des contraintes et le deuxième permet de décrire les relations et les caractéristiques de ces entités.

```
<mdr:changeConstraint mdr:ID = "subClassOfChangeConstraint">
 <mdr:definition>
             ...
 </mdr:definition>
 <mdr:description>
             ...
 </mdr:description>
</mdr:changeConstraint>
```

Ainsi, la spécification $\mathcal{MDR}$ du changement qui consiste à modifier une énumération en supprimant un de ses éléments est définie comme suit :

```
<mdr:enumerationChange>
    <mdr:changeEntity mdr:type = property mdr:localName = "propertyName" />
    <mdr:changeObject> owl:oneOf </mdr:changeObject>
    <mdr:changeType> Modify </mdr:changeType>
    <mdr:oldEnumerationValues> Value1 Value2 Value3 </mdr:oldEnumerationValues>
    <mdr:newEnumerationValues> Value1 Value2 </mdr:newEnumerationValues>
    <mdr:changeConstraint> enumerationChangeConstraint </mdr:changeConstraint>
  </mdr:enumerationChange>
```

La contrainte que ce changement doit vérifier est que les valeurs testées ou assignées à la propriété *propertyName* doivent appartenir à l'ensemble des éléments de la nouvelle énumération. Cette contrainte est définie comme suit :

```
<mdr:changeConstraint mdr:ID = "enumerationChangeConstraint">
<mdr:definition>
  <mdr:enumeration mdr:ID="oldEnumeration">
  <mdr:enumeration mdr:ID="newEnumeration">
  <mdr:property mdr:ID="aproperty"/>
  <mdr:ruleValue mdr:ID="arulevalue">
  <mdr:rule mdr:ID="arule"/>
</mdr:definition>
<mdr:description>
   <mdr:resource="arule" mdr:usedResource="aproperty" mdr:resource="arulevalue"/>
   <mdr:resource="arulevalue" mdr:valueFrom mdr:resource="newEnumeration">
</mdr:description>
</mdr:changeConstraint>
```

L'intérêt d'avoir ce langage, outre le fait d'avoir une représentation unifiée de notre approche, est de :

1. simplifier l'étape de spécification du changement – pour utiliser notre système dans son état actuel, l'utilisateur qui est la personne responsable de la gestion de l'évolution du système d'information de son entreprise, doit créer des individus dans l'ontologie $\mathcal{MDR}$ qui modélisent les changements de l'ontologie métier. Le langage $\mathcal{MDR}$ permettra d'éviter cette tâche qui nécessite d'avoir non seulement les connaissances d'utilisation des éditeurs d'ontologies mais aussi des connaissances dans OWL. Ceci en effectuant un *parsing* de la définition $\mathcal{MDR}$ du changement vers des individus de l'ontologie $\mathcal{MDR}$.

2. permettre la génération automatique des patrons d'incohérences – les patrons d'incohérences présents dans l'implémentation actuelle de notre approche ont été développés manuellement, ce qui représente une tâche très difficile et coûteuse en termes de temps, et n'assurent pas une couverture complète de toutes les contraintes que tous les changements doivent vérifier. Le *parsing* de la définition des contraintes de changement en règles permettra de générer automatiquement les patrons d'incohérences nécessaires pour gérer l'impact d'un changement. Ceci permettra à l'utilisateur de décrire lui-même les contraintes que le changement doit vérifier.

3.7 Conclusion

Dans ce chapitre, nous avons présenté l'approche $\mathcal{MDR}$ qui permet de gérer l'impact des évolutions d'ontologies sur des règles qui en dépendent. Nous avons présenté une modélisation avec la méthode CommonKADS de l'idée générale de notre approche. Nous avons ensuite décrit les composants de $\mathcal{MDR}$, à savoir les patrons de gestion des changements, et leur spécification. Nous avons aussi présenté l'architecture de notre approche ainsi que le processus de gestion des changements.

La principale contribution de notre approche est l'ontologie $\mathcal{MDR}$ qui fournit une spécification du changement qui n'est pas relative à un modèle particulier et qui permet de modéliser et le changement en lui même et son impact sur les modèles qui en dépendent. Nous avons utilisé cette spécification pour modéliser des changements d'ontologies ainsi que leur impact sur les règles. Pour cela, nous avons proposé une classification des changements d'ontologies OWL, une classification des problèmes de cohérence des règles ainsi que des réparations pour les résoudre.

Afin de détecter les problèmes de cohérence causés par les changements d'ontologies, nous avons proposé un ensemble de patrons d'incohérences, implémentés par des règles, les *règles de détection des incohérences* qui définissent les contraintes de changements et détectent les incohérences causées.

La proposition des réparations est effectuée grâce aux patrons de réparation, implémentés par des règles, les *règles de réparation*, qui en fonction d'un changement et de son impact, proposent un ensemble de réparation

Afin d'unifier la spécification des différents composants de notre approche, nous avons proposé le langage de spécification $\mathcal{MDR}$ qui permet de représenter les changements ainsi que les contraintes que chaque changement doit vérifier. En perspective, un *parsing* de ce langage permettrait de générer automatiquement la spécification de changement dans le format $\mathcal{MDR}$ et permettrait aussi de générer automa-

tiquement les règles de détection des incohérences. Mais pour cela, il faudrait que ce langage permette de spécifier les incohérences qui peuvent être causées par un changement ainsi que les réparations.

Chapitre 4

Implémentation et Expérimentations

Sommaire

4.1 Introduction

L'idée principale de ce travail est de permettre à des utilisateurs métier, n'ayant aucune connaissance des ontologies, de pouvoir automatiser des décisions métier

dont la sémantique est formalisée en OWL. Pour cela, nous nous sommes orientés vers les règles métier et nous avons développé un prototype qui permet d'écrire, en langage naturel contrôlé, et exécuter des règles métier à partir d'ontologies OWL. Cette idée nous a mené vers un nouveau type de dépendence, d'où la problématique principale de cette thèse qui est la gestion de l'impact de l'évolution des ontologies sur les règles.

Pour permettre l'intégration des ontologies et des règles, nous nous sommes fondés sur le Système de Gestion des Règles Métier (SGRM) IBM Operational Decision Management (ODM) [1]. Ce système permet l'édition, la maintenance et l'exécution de règles métier. L'idée est donc d'intégrer des ontologies dans ODM et d'utiliser toutes les fonctionnalités offertes par ce système pour écrire et exécuter des règles à partir d'ontologies. Pour cela, nous avons développé le *plug-in OWL* qui permet d'importer des ontologies OWL dans ODM pour ensuite écrire, en langage naturel contrôlé, et d'exécuter des règles à partir d'ontologies.

Pour gérer l'impact de l'évolution des ontologies sur les règles, nous avons développé le *plug-in* $\mathcal{MDR}$ pour ODM. Ce plug-in est une implémentation de l'approche $\mathcal{MDR}$. Ce composant, permet d'analyser les changements des ontologies métier pour détecter les éventuelles incohérences sur les règles qu'ils peuvent causer et de proposer des réparations pour ces incohérences.

Pour définir les utilisateurs ciblés par ces prototypes, nous avons utilisé une approche de *usability* qui permet de définir les profils des utilisateurs qui interagissent dans les processus d'écriture et d'exécution des règles à partir d'ontologies et dans le processus de gestion de l'impact de l'évolution des ontologies sur les règles (de Bonis & Bellino, 2011).

Pour expérimenter les approches proposées, nous avons développé deux applications : (1) une application de validation des prescriptions médicamenteuses à base d'une ontologie OWL et de règles métier. Cette application a été développée en collaboration avec l'Hôpital Européen George Pompidou (HEGP) ; (2) une application, en collaboration avec Audi, un des partenaires du projet ONTORULE, qui permet de vérifier si les tests effectués sur les différents équipements d'un véhicule, lors de son processus de construction, sont conformes aux normes imposées par la réglementation européenne.

1. http ://www-01.ibm.com/software/decision-management/operational-decision-management/websphere-operational-decision-management/

4.2 Intégration des ontologies et des règles : implémentation

4.2.1 IBM Operational Decision Management

IBM Operational Decision Mangement (ODM) [2] est un Système de Gestion des Règles Métier (SGRM) qui permet à des utilisateurs métier d'éditer, d'exécuter et de mettre à jour, de manière collaborative, des règles métier écrites en langage naturel contrôlé. Le processus d'édition des règles métier dans ODM repose principalement sur les composants suivants :

eXecutable Object Model (XOM) – c'est le composant qui permet l'exécution des règles. Il modélise les objets de l'application de règles, et permet de générer le BOM. Le XOM est formalisé soit à l'aide de classe Java soit à l'aide d'un schéma XML. Lors de l'exécution, les règles sont déclenchées soit à travers les instances Java (dans le cas où le XOM est un ensemble de classe Java), soit à travers les données XML (dans le cas où le XOM est un schema XML).

Business Object Model (BOM) – le BOM est un modèle orienté-objet, proche des diagrammes de classe UML. Il décrit les concepts et les propriétés d'un domaine et est représenté sous forme d'un ensemble de classes, correspondant aux concepts du domaine, et chaque classe contient un ensemble d'attributs, correspondant aux propriétés des concepts. Les composants du BOM (classes et attributs) définissent les entités qui composent le texte des règles.

Vocabulaire (VOC) – la couche métier est composée de deux niveaux, les objets métier (*client, âge*), représentés par le BOM, et le vocabulaire qui fournit la terminologie nécessaire pour l'écriture des règles dans un langage naturel contrôlé (*"le client"*, *"l'âge du client"*). Le VOC est ensuite utilisé pour écrire le texte des règles. Exemple :

> `Si` *l'âge du client est inférieur à 30*
> `Alors` *assigner le taux d'intérêt à 1.5*

2. http ://www-01.ibm.com/software/decision-management/operational-decision-management/websphere-operational-decision-management/

4.2.2 Édition des règles métier à partir d'ontologies OWL

Comme nous l'avons décrit dans la section précédente, le BOM est l'entité majeure pour l'édition des règles dans ODM. Pour permettre aux utilisateurs métier, l'édition des règles métier en langage naturel contrôlé, nous avons effectué une transformation du modèle OWL vers le BOM. Ainsi, lorsque l'utilisateur importe une ontologie OWL dans ODM, le BOM est automatiquement généré et toutes les fonctionnalités offertes par le système sont exploitées (voir Figure 4.1).

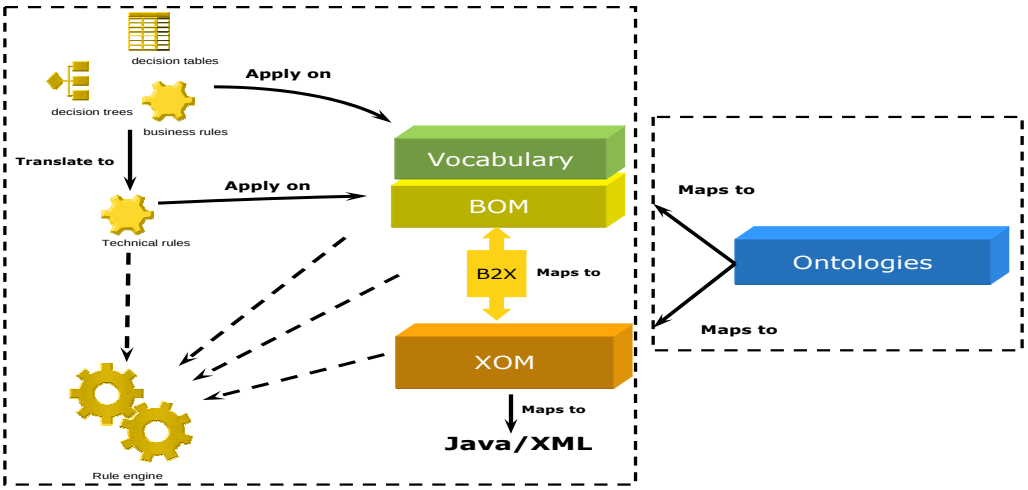

FIGURE 4.1 – Intégration des ontologies dans ODM.

Pour des raisons de différence de pouvoir d'expressivité entre OWL et le BOM, nous nous sommes focalisés sur un sous-ensemble des constructeurs offert par le profil OWL-DL qui peut être représenté dans le BOM. D'un côté, le BOM est un modèle orienté objet, toutes les connaissances sont donc explicitement modélisées sauf la relation de sous classe qui modélise la spécification d'une classe par rapport à une autre. D'un autre côté, dans OWL, beaucoup de connaissances sont inférées à l'aide des raisonneurs grâce à l'utilisation des axiomes qui permettent de raisonner sur le modèle.

Cette différence de pouvoir d'expressivité fait que certains constructeurs OWL tels que `owl:disjointWith`, `owl:complementOf`, `owl:someValuesFrom`... ne peuvent pas être modélisés dans le BOM. Néan-

moins, certains de ces constructeurs sont pris en considération lors de la phase d'exécution pour inférer le types des entités qui déclenchent l'execution des règles (voir Section 4.2.3) Ci-dessous, la description de la transformation de OWL vers le BOM (voir Table 4.1).

Classes OWL : une classe OWL est transformée en une classe BOM. Les relations hiérarchiques OWL sont transformées dans le BOM à l'aide de la relation de sous-classe et comme le BOM supporte la notion de l'héritage multiple, cette information est préservée.

Individus : les individus sont transformés en des instances Java lors de l'exécution des règles (voir Section 4.2.3).

Propriétés : une propriété OWL est transformée en un attribut du BOM. La classe d'un attribut correspond au domaine de la propriété et son type correspond au co-domaine de la propriété. Les propriétés fonctionnelles sont transformées en des attributs mono-valués et les propriétés non fonctionnelles sont transformées en des attributs multi-valués.

Dans OWL, une propriété peut avoir zéro ou plusieurs domaines ou co-domaines. Dans ces cas, la transformation est effectuée comme suit :

- *Domaine null :* l'attribut est ajouté à toutes les classes racines, (ı.e les classes qui héritent directement de `owl:Thing`) ;
- *Domaine multiple :* l'attribut est ajouté à toutes les classes de l'ensemble des domaines.
- *Co-domaine null :* le type de la propriété est inféré comme suit :
  - si la propriété possède une propriété équivalente, le type de l'attribut sera le co-domaine de cette dernière,
  - si la propriété possède une propriété inverse, le type de l'attribut sera le domaine de cette dernière ;
  - sinon le type de l'attribut est Object.

Restrictions :

- `owl:cardinality` and `owl:maxCardinality` restrictions : si la valeur de l'une de ses restrictions est égale à 1 alors l'attribut est mono-valué sinon c'est un attribut multi-valué ;
- `owl:allValuesFrom` restriction : le type de l'attribut est la classe définie par la restriction ;

OWL	BOM
Class ?A	Class A
?B rdfs:subClassOf ?A	Class B extends A
?C owl:intersectionOf(?A,?B)	Class C extends A,B
?C owl:unionOf(?A,?B)	Class A extends C and Class B extends C
?A owl:oneOf {x, y, z}	Class A {domain {'x', 'y', 'z'};}
?A owl:equivalentClass ?B	Keep only A, and references to B are reported to A
P(?A,?B)	Class A {B[ ] P};
P rdfs:subPropertyOf P'(?A,?C)	Class A {B[ ] P; C[ ] P';}
P' owl:equivalentProperty P	Class A {B[ ] P; B[ ] p';}
P' owl:inverseOf P	Class B {A[ ] P;}
P owl:functionalProperty	Class A {B P;}
P.cardinality = 1	Class A {B P;}
P.maxCardinality = 1 on P	Class A {B P;}
P owl:allValuesFrom C	Class A {C[ ] P;}

TABLE 4.1 - OWL to BOM Mapping

– `owl:oneOf` restriction : des valeurs statistiques, correspondant aux éléments de la restriction sont affectées à la classe.

4.2.3 Exécution des règles métier à partir d'ontologies OWL

Pour exécuter les règles éditées à partir d'une ontologie, nous effectuons une deuxième transformation des entités OWL/BOM vers le XOM en utilisant Jena. Jena est une API Java qui permet de générer des objets Java à partir d'ontologies OWL. Ces objets Java constituent donc le XOM. L'utilisation de Jena fournit une couche d'exécution qui gère les mécanismes d'inférence de OWL et qui et qui permet la génération des objets Java à partir des concepts, propriétés et individus OWL.

Quand l'utilisateur lance l'exécution des règles, les individus de l'ontologie (ABox) sont transformés en des instances Java et chargés dans la *working memory* du moteur de règles. Ce dernier évalue la partie condition des règles en fonction des objets de la *working memory* et déclenche l'exécution des règles qui s'apparient avec ces objets. L'exécution des règles modifie l'état de certains objets de la *working memory* ; Etant donné que ces objets correspondent à la ABox de l'ontologie, cette dernière est mise à jour après l'exécution de chaque règle (voir Figure 4.2).

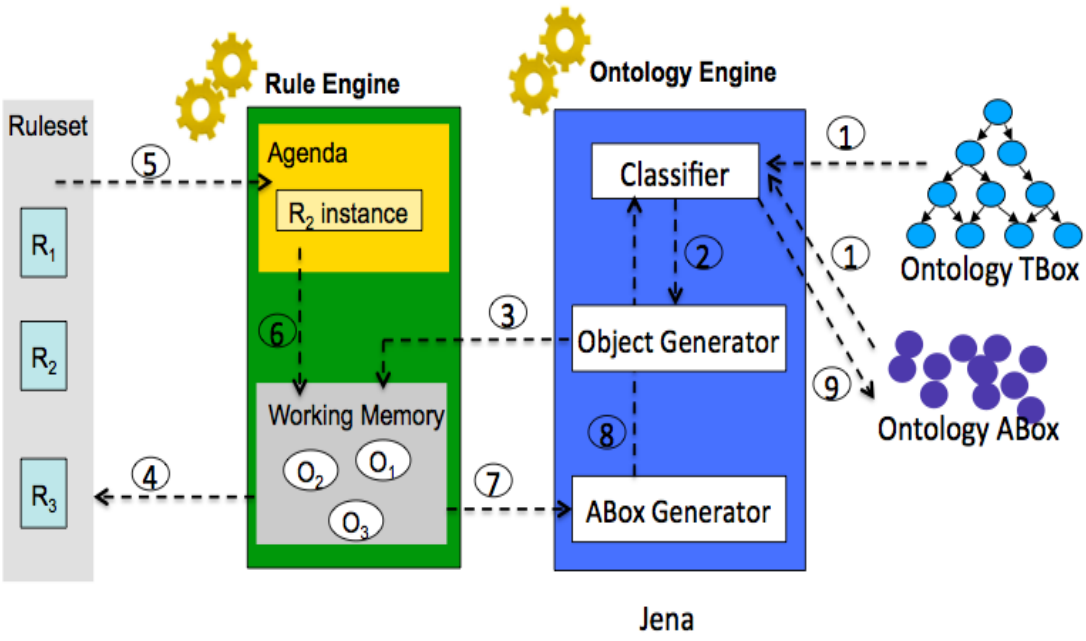

FIGURE 4.2 – Exécution des règles.

Un des points importants lors de l'exécution des règles est l'interaction entre le moteur de règles et le raisonneur. Le raisonneur infère le type d'un individu et le moteur de règle utilise ce type inféré pour exécuter des actions qui ne peuvent être modélisées par une ontologie. En d'autres termes, le moteur de règles demande le type des individus au raisonneur pour ensuite exécuter les règles qui s'apparient avec ces derniers. Ci-dessous, des exemples d'interaction entre le raisonneur et le moteur de règles, fondées sur une ontologie que nous avons développée principalement pour démontrer les avantages ainsi que les limites de cette approche.

Exemple 1 : L'ontologie contient les définitions suivantes :

SupervisedDriver $\sqsubseteq$ **Driver**

CarDriver $\equiv$ *hasDrinvingLicense* `some` **CarDrivingLicense**

numberOfContravention (**Driver**, int)

supervisedDrivingMonth (**SupervisedDriver**, int)

SupervisedDriver(Joe) ; *hasAge*(Joe, 18) ; *hasName*(Joe, "Joe") ; *supervisedDrivingMonth*(Joe, 26) ; *numberOfContravention*(Joe, 3) ;

SupervisedDriver(John) ; *hasAge*(John, 18) ; *hasName*(John, "John") ; *supervisedDrivingMonth*(John, 30) ; *numberOfContravention*(John, 0) ;

A partir de ces définitions, nous avons écrit la règle (*i.e. car driving license*) qui attribut un **CarDrivingLicense** à tous les **SupervisedDriver** ayant 18 ans et ayant fait de la conduite accompagnée pendant 24 mois au minimum et le nombre de contraventions du **SupervisedDriver** est au maximum 1.

car driving license :
IF the age of the supervised driver is 18
and the supervised driving month of the supervised driver is at least 24
and the number of contravention of the supervised driver is at most 1
THEN add the car driving license to the driving licenses of the supervised drivers ;

Après l'exécution de *car driving license*, nous remarquons qu'un **CarDrivingLicense** est attribué à John puisqu'il vérifie les conditions de la règle. Nous exécutons ensuite la règle *car driver* qui liste les noms de tous les **CarDriver**. Le résultat est : *John is a car driver*, ce qui signifie que John est re-classifié en tant que **CarDriver** parce qu'il a un **CarDrivingLicense**.

car driver :
THEN print the name of the car driver + " is a car driver" ;

Exemple 2 : L'ontologie contient les définitions suivantes :

ChildCarDriver $\sqsubseteq$ **Child** $\sqsubseteq$ **Person**

ChildCarDiver $\equiv$ *hasFather* `only` **CarDriver**

Person(Toto) ; *hasAge*(Toto, 8) ; *hasName*(Toto, "Toto")

hasFather(Toto, John)

Après avoir exécuté la règle *car driving license*, nous avons exécuté la règle *child car driver* qui liste le nom de tous les **ChildCarDriver**. Le résultat est *Toto is a child car driver.*

child car driver :
THEN print the name of the child car driver + " is a child car driver";

Exemple 3 : L'ontologie contient le concept **Contravention** avec les définitions suivantes :

hasContraventionAmount(**Contravention**, float)
contraventionAmountToPay(**Driver**, float)
Contravention(c1), **Contravention**(c2)
hasContravention (John, c1), *hasContravention* (John, c2)

A partir de ces définitions, nous écrivons la règle *remove car license* qui retire le **CarDrivingLicense** à chaque **CarDriver** qui a une contravention et calcule le montant de contravention que chaque conducteur doit payer.

Après l'exécution de cette règle, John perd sa **carDrivingLicense** et ne sera donc plus classifié entant que **CarDriver**. Ainsi, lorsque nous re-exécutons les règles *car driver* et *child car driver*, nous obtenons comme résultat : *Joe is a car driver.*

remove car license :
definitions
set 'a contravention' to a contravention ;
set 'a car driver' to a car driver where the contraventions of this car driver contain a contravention ;
IF 'a car driver' is not null
THEN remove the car driving license from the driving licenses of 'a car driver' ;
for each contravention in the contraventions of 'a car driver' :
- set the contravention amount to pay of 'a car driver' to the contravention amount to pay of 'a car driver' + the contravention amount of this contravention ;

Quand on ré-exécute les règles *car driver* et *child car driver* aucun message ne sera affiché, puisque aucun des individus de l'ontologie ne permet de déclencher leur exécution.

Exemple 4 L'ontologie définit le concept **RiskyDriver** comme un **Driver** qui a au minimum 3 contraventions, et nous attribuons à John 4 contraventions

RiskyDiver ≡ *hasContravention* `min` 3

RiskyDriver(Frank) ; *hasName*(Frank, "Frank")

hasContravention (John, c1), *hasContravention* (John, c2), *hasContravention* (John, c3), *hasContravention* (John, c4) tels que **Contravention**(c1), **Contravention**(c2), **Contravention**(c3), **Contravention**(c4).

Quand on exécute la règle *risky driver*, le résultat est *Frank is a risky driver*, ce qui signifie que le raisonneur n'a pas classifié John, qui a 4 conventions, comme un **RiskyDriver**.

risky driver :
THEN print the name of the risky driver + " is a risky driver";

4.3 Scénarios d'utilisation des approches proposées

Pour illustrer les processus d'utilisation des approches proposées, nous avons utilisé une *usability approach* (9241, 2008), fondée sur la *persona development methodology* définie et appliquer à la *Usability Engineering* par Alan Cooper by (Cooper & Reimannn, 2003).

Le but de la *usability approach* est de faciliter le développement, l'utilisation et la gestion des applications métier par les différents acteurs qui auront à les utiliser. La persona development methodology consiste à définir les profiles des utilisateurs impliqués pour guider les décisions concernant la conception visuelle, la fonctionnalité, la navigation et le contenu de l'application qu'ils utiliseront. L'utilisateur défini par un personna caractérise un groupe d'utilisateurs. Persona est une méthode puissante pour évaluer la pertinence d'une application par rapport aux profils des utilisateurs ciblés et pour que ces différents utilisateur puissent participer à la conception et au développement de leurs applications.

4.3.1 Scénario d'utilisation du plug-in OWL

Dans cette section, nous présentons les différents utilisateurs qui interagissent dans le processus d'écriture et d'exécution des règles métier à partir des ontologies (voir Figure 4.3).

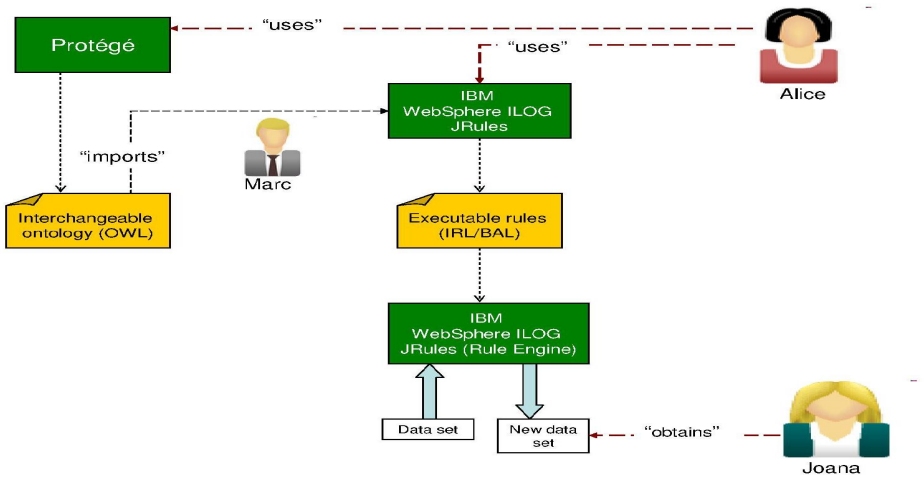

FIGURE 4.3 – Scénario d'utilisation du plug-in OWL.

Lors de ce processus, trois types d'utilisateurs entrent en jeu, Marc, Alice et Joana. Marc est l'analyste métier, il joue le rôle de coordinateur entre les experts métier et le département informatique. Sa mission est de formaliser les connaissances métier, de sorte qu'elles peuvent être transformées en besoins informatiques et de s'assurer que le modèle métier formalisé est correct, complet et valide. Alice est l'expert du domaine. Elle connaît bien les politiques commerciales de son domaine et a été formé pour modifier les règles dans l'application de règles métier. Joana est l'utilisateur opérationnel de l'application de règles. Les règles lui permettent de réaliser des tâches métier, elle voit le résultat de leur exécution mais ne peut pas voir leur structure.

Marc charge l'ontologie métier dans ODM, ce qui génère automatiquement le BOM. Il vérifie la verbalisation générée et peut avoir à modifier certains éléments du vocabulaire (VOC) si nécessaire. Cette terminologie est ensuite utilisée par Alice pour écrire les règles métier. Une fois les règles écrites, Joana peut les utiliser pour exécuter des décisions de son métier.

4.3.2 Scénario d'utilisation du plug-in $\mathcal{MDR}$

Lors du processus de gestion de l'impact de l'évolution des ontologies sur les règles, deux utilisateurs entrent en jeu, Marc et Alice (voir Figure 4.4). Marc, avec l'aide d'Alice modélisent les changements à appliquer à l'ontologie métier. Ensuite, il déclenche le plug-in $\mathcal{MDR}$ et la liste des changements modélisés est affichée. A cette étape, Alice peut choisir le changement qu'elle veut appliquer en le sélectionnant dans la liste des changements, elle ne peut appliquer qu'un changement à la fois. Quand Alice sélectionne un changement, une description du changement est affichée et les *règles de détection des incohérences* sont déclenchées. La description du changement contient l'ensemble des entités qu'il implique ainsi que les règles impactées. Après l'exécution des *règles de détection des incohérences*, une liste des incohérences détectées est affichée et Alice peut choisir les incohérences qu'elle veut réparer ce qui déclenche les *règles de réparation*. Une liste de réparation est proposée à Alice, une seule réparation peut être choisie pour réparer une incohérence.

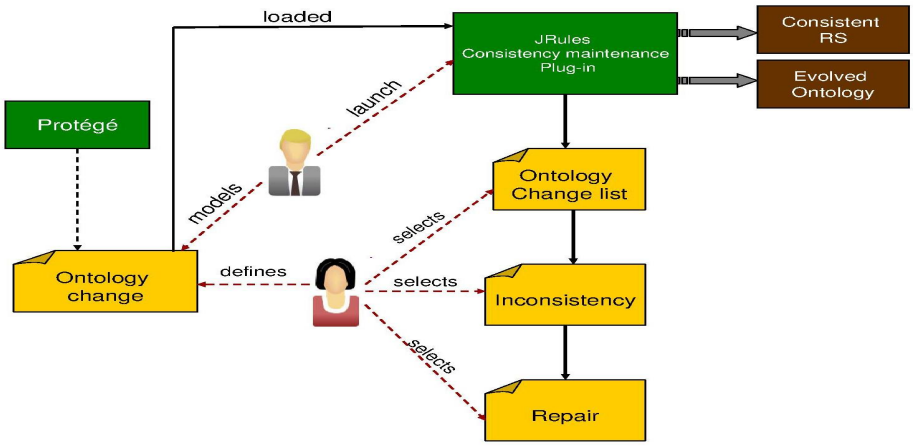

FIGURE 4.4 – Scénario d'utilisation du plug-in $\mathcal{MDR}$.

4.4 Exemples d'application de notre approche

4.4.1 Application de validation des prescriptions médicamenteuses

L'application de validation des prescriptions médicamenteuse a été développée en collaboration avec des pharmaciens de l'HEGP (Hôpital Européen George Pompidou) sur la base du travail présenté dans (Boussadi *et al.*, 2010). Cette application présente un système d'alerte à base d'une ontologie OWL et d'un ensemble de règles métier ou plus précisément des *règles de validation clinique*. Dans cette application, l'ontologie modélise les entités pertinentes dans le processus de validation pharmaceutique des prescriptions alors que les règles, éditées à partir de cette ontologie, testent la validité des prescriptions en fonction des valeurs données aux entités. Quand le système détecte une prescription invalide, une alerte est déclenchée pour informer le pharmacien du problème et un ensemble de recommandations est affiché pour justifier le problème détecté et proposer des prescriptions alternatives.

4.4.1.1 Écriture et exécution des règles métier

L'ontologie est définie sur la base du modèle UML décrit dans (Boussadi *et al.*, 2010) et contient 17 concepts et 25 propriétés. Dans ce qui suit nous nous focalisons principalement sur les concepts et propriétés qui ont été utilisés pour l'écriture des règles présentées ci-dessous [3]. Ces concepts sont **Patient** qui a des **LabResult** et un ensemble de **Prescription**. Une prescription concerne un médicament **Drug** et a un **DosageRegimenPhase**. Un **DosageRegimenPhase** a une unité de dosage, *dosageUnit*, la quantite du dosage, *dosage* et un *relatedTimeUnit* qui représente la fréquence d'administration d'un médicament. Cette propriété ne peut avoir qu'une des valeurs suivantes, "TLJ" [4], "1" ou "2".

Les règles éditées à partir de cette ontologie détectent les prescriptions invalides en testant sur :

– le *presentation name* d'un **Drug** ;

3. Dans ce qui suit, les mots écrit en gras représentent les concepts de l'ontologie métier et ceux écrit en italique représentent les propriétés

4. tous les jours

– le *dosage unit* et le *dosage* du **DosageRegimenPhase** d'une **Prescription** ;
– le **GFR (Glomerular Filtration Rate)** du **LabResult** d'un **Patient**.

En fonction des valeurs assignées à ces propriétés, les règles déclenchent des alertes qui informent le pharmacien des éventuelles prescriptions qui ne sont pas valides et proposent des recommandations pour fixer le problème détecté.

L'ontologie de de validation des prescriptions est créée par Marc, le technicien, avec l'aide de Alice l'expert du domaine. Après la création de l'ontologie, Marc l'importe dans ODM ce qui génère automatiquement le BOM ainsi que le VOC (voir Section 4.2.1) et permet donc à Alice l'écriture des *règles de décision clinique* (voir Figure 4.5).

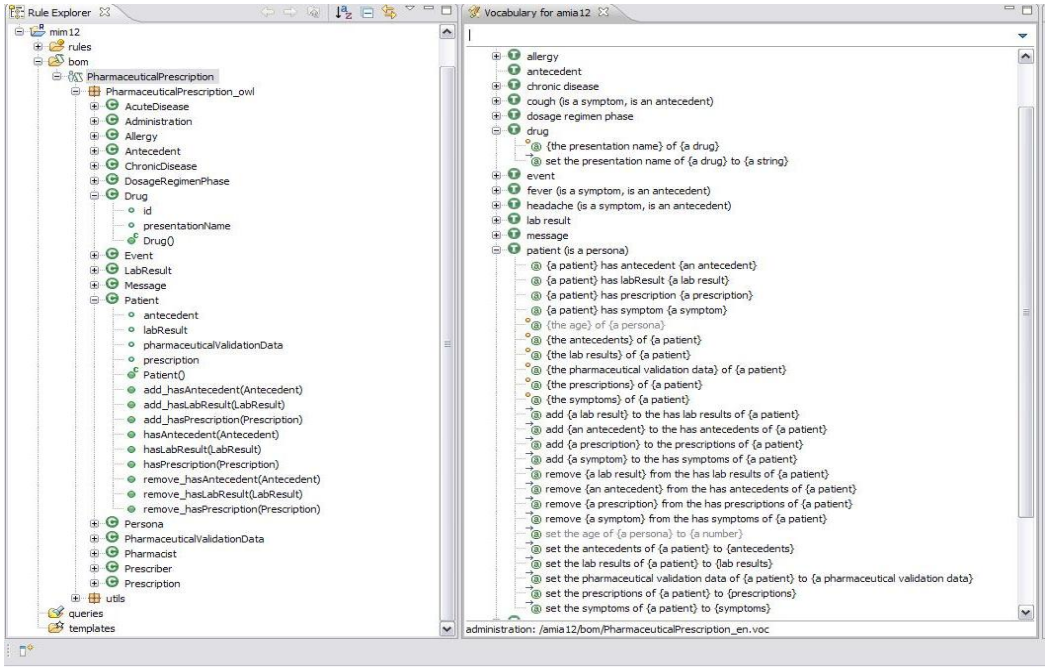

FIGURE 4.5 – BOM et VOC générés à partir de l'ontologie de validation des prescriptions.

Dans ce qui suit, nous présentons trois exemples de règles écrites par Alice. La première règle, appelée `GLUCOPHAGE 1 (Metformin)- 1000 -TABLET` (voir Figure 4.6), teste si :

– le nom du médicament est "GLUCOPHAGE 1000MG TAB" ou "GLU-

COPHAGE 1000MG CPR COATED" ;
- l'unité de dosage est "Tablet" ;
- l'unité de temps pour l'administration du médicament est soit "TLJ" (ce qui signifie que le médicament doit être administré toutes les 24 heures) ;
- le dosage du schéma posologique est supérieur à 3 ;
- le DFG dans les résultats des analyses sanguins du patient est supérieur à 80.

Dans le cas d'une telle prescription, cette règle assigne la valeur `false` à la propriété de validation de la prescription et déclenche un message d'alerte qui contient le nom de la règle qui a été exécutée et une recommandation sur le dosage qui doit être compris entre 1 et 3 grammes toutes les 24 heures dans le cas où la DFG du patient est supérieure à 80. Le dosage recommandé par le système dans ce cas est 3.

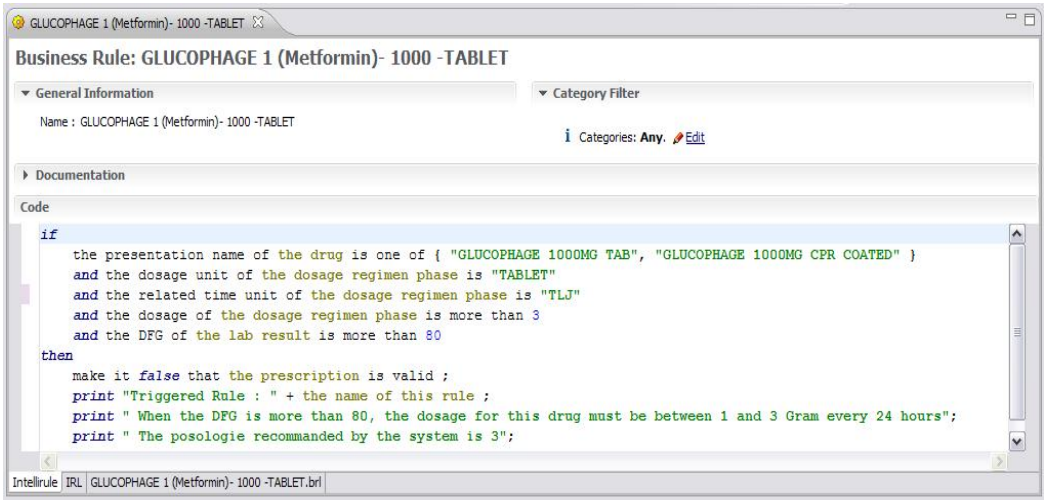

FIGURE 4.6 – Règle métier : GLUCOPHAGE 1 (Metformin)- 1000 -TABLET.

La deuxième règle, appelée `GLUCOPHAGE 2 (Metformin)- 1000 -TABLET` (voir Figure 4.7), teste si :

- le nom du médicament est "GLUCOPHAGE 1000MG TAB" ou "GLUCOPHAGE 1000MG CPR COATED" ;
- l'unité du dosage est "Tablet" ;
- l'unité temps pour l'administration du médicament est soit "TLJ" soit "1" (l'unité de temps "1" est équivalente à "TLJ" les deux notations

peuvent être utilisées pour signaler que le médicament doit être administré toutes les 24 heures) ;
- le dosage du schéma posologique est supérieur à 1 ;
- le DFG dans les résultats des analyses sanguines du patient est compris entre 60 et 80.

L'exécution de cette règle assigne la valeur `false` à la propriété de validation de la prescription et génère un message d'alerte contenant le nom de la règle exécutée et le dosage recommandé qui doit être à 1.5 grammes toutes les 24 heures dans le cas où le DFG du patient est compris entre 60 et 80.

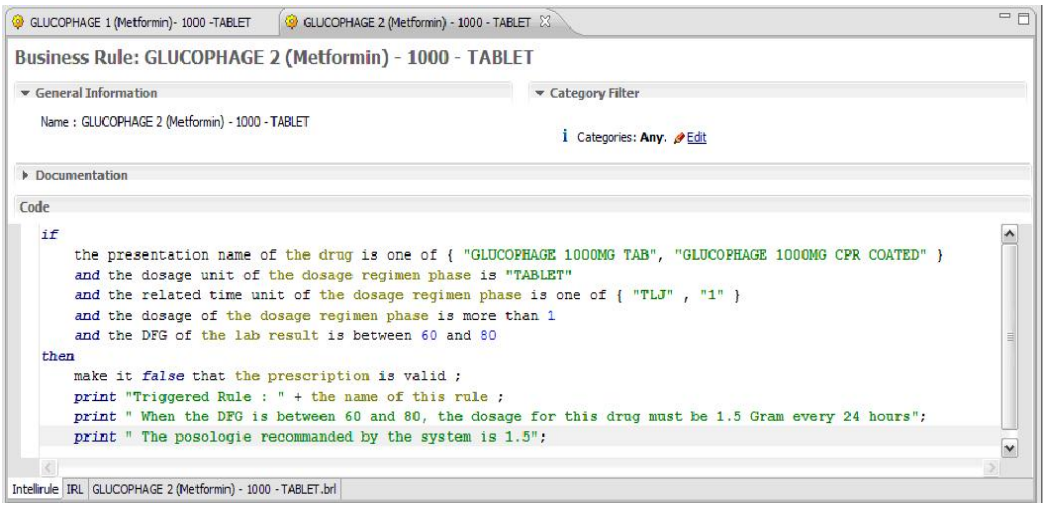

FIGURE 4.7 – GLUCOPHAGE 2 (Metformin)- 1000 -TABLET.

La troisième règle, appelée `TimeUnit-GLUCOPHAGE-1000-TABLET` (voir Figure 4.8), assigne dans sa partie action la valeur "TLJ" à l'unité temps pour l'administration du médicament si :
- le nom du médicament est "GLUCOPHAGE 1000MG TAB" ou "GLUCOPHAGE 1000MG CPR COATED" ;
- l'unité du dosage est "TABLET" ;
- le dosage du schéma posologique est compris entre 1 et 3 ;
- le DFG dans les résultats des analyses sanguines du patient est supérieur à 80.

Pour vérifier la validité d'une prescription, le pharmacien entre les données nécessaires à l'exécution des règles dans le système. Par exemple, Joana entre des données relatives à trois nouvelles prescriptions données à trois patients

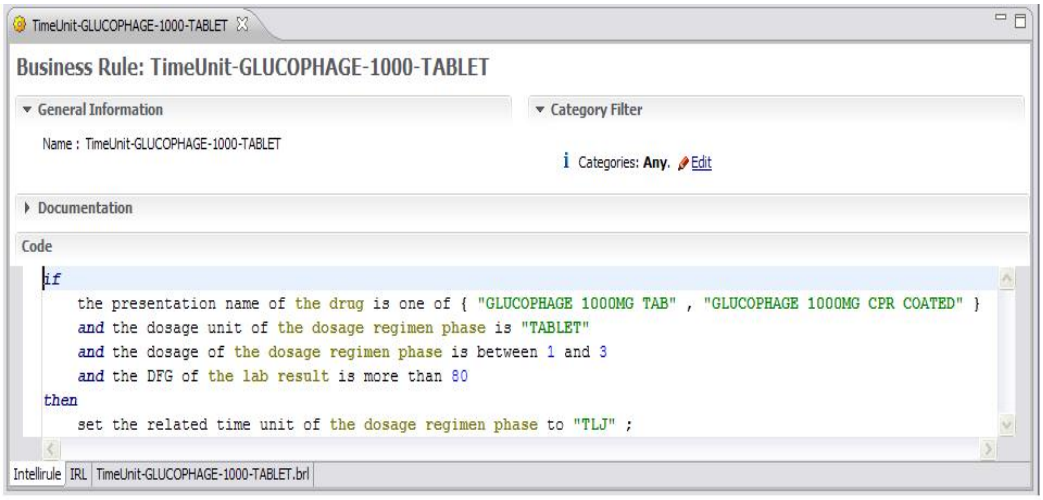

FIGURE 4.8 – TimeUnit-GLUCOPHAGE-1000-TABLET.

différents.

Prescription 1 est prescrite au Patient 1 qui a un DFG égal à 90, l'unité de dosage du schéma posologique est "TABLET", son unité de temps est "1" et un dosage égal à 4. Cette prescription concerne le médicament "GLUCOPHAGE 1000MG TAB". Dans ce cas, la règle "GLUCOPHAGE1 (Metformine)-1000- TABLET" sera exécutée et assignera la valeur `false` à la propriété de validation de la prescription. Le message d'alerte généré par le système concernera la valeur du dosage qui doit être compris entre 1 et 3 dans le cas où le DFG est supérieur à 80. La valeur du dosage recommandée par le système est 3 (voir Figure 4.9).

Prescription 2 est prescrite au Patient 2 qui a un DFG égal à 70. L'unité de dosage du schéma posologique est "TABLET", son unité de temps est "TLJ" et la valeur du dosage est égale à 2. Cette prescription concerne le médicament "GLUCOPHAGE 1000MG CPR COATED". Dans ce cas la règle "GLUCOPHAGE 2 (Metformin)-1000-TABLET" sera exécutée et assignera la valeur `false` à la propriété de validation de la prescription. Le message d'alerte généré par le système concernera la valeur du dosage qui doit être égale à 1.5 Gram dans le cas où le DFG est compris entre 60 et 80 (voir Figure 4.10).

Prescription 3 est prescrite au Patient 3 qui a un DFG égal à 50. L'unité

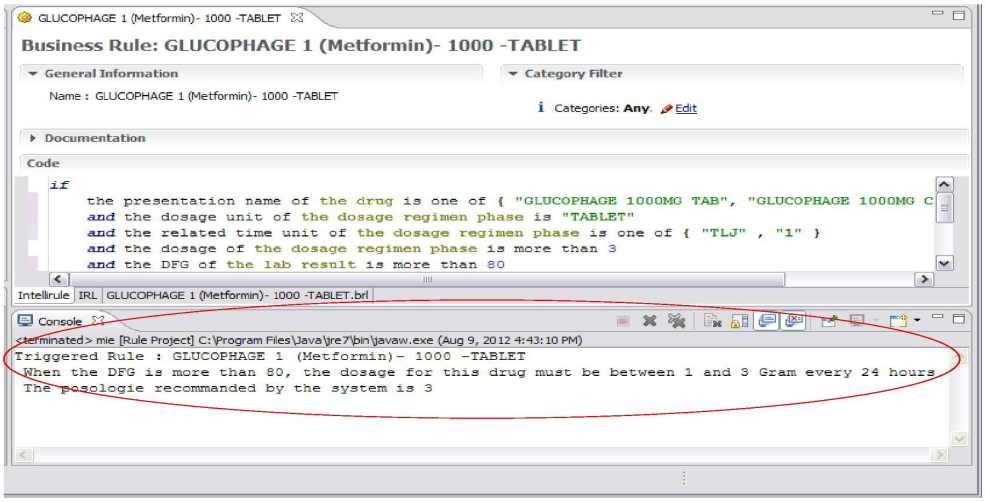

FIGURE 4.9 – execution of GLUCOPHAGE 1 (Metformin)- 1000 -TABLET rule.

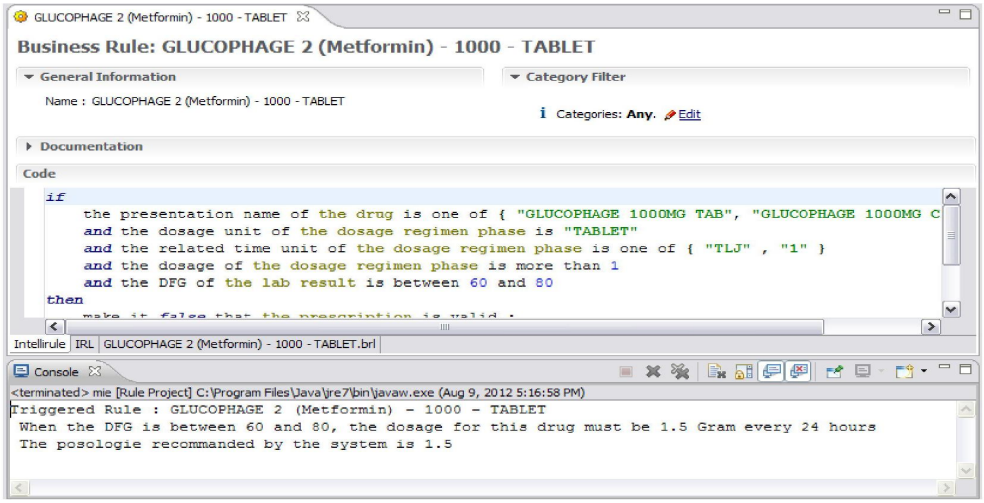

FIGURE 4.10 – execution of GLUCOPHAGE 2 (Metformin)- 1000 -TABLET rule.

de dosage du schéma posologique est “TABLET”, son unité de temps est “TLJ” et la valeur du dosage est égale à 2. Cette prescription concerne le médicament “GLUCOPHAGE 1000MG CPR COATED”. Dans ce cas aucune règle ne sera exécutée ce qui signifie que la prescription est valide.

4.4.1.2 Validation du système proposé

Pour vérifier la validité des résultats obtenus par notre système, nous avons effectué une étude comparative sur les prescriptions validées ou pas par les

	Valide	Pas valide
Accepté	73	8
Rejeté	1	0
Équivalente	14	3
Non soumise	18	5

TABLE 4.2 – Avis pharmacien vs. Validation du système.

pharmaciens de l'HEGP et par notre notre système (voit Table 4.2). Cette étude a été effectuée sur 123 prescriptions. A l'HEGP, les pharmaciens peuvent assigner un des quatre états suivants à une prescription :

- *Accepté* – si la prescription est valide ;
- *Rejeté* – si prescription n'est pas valide ;
- *Non soumise* – si la prescription n'a pas été soumise à la validation du pharmacien ;
- *Équivalente* – si le médicament prescrit n'est pas disponible dans la pharmacie et que le biologiste décide de donner un médicament équivalent.

Parmi les 123 prescriptions utilisées pour valider notre travail il y a :

- 81 prescriptions *acceptées* alors que seulement 73 parmi ces 81 prescriptions ont été validé par notre système et les 8 autres ont généré des alertes ;
- 17 prescriptions *équivalentes* parmi elles seulement 14 ont été validées par notre système et 3 ont généré des alertes ;
- 23 prescriptions *non soumises* parmi elles 18 ont été validées par le système et 5 ont généré des alertes ;
- 1 prescription *rejetée* et qui a été validé par le système.

Parmi les huit prescriptions qui n'ont pas été validées par le système et acceptées par le pharmacien, le système a raison sur deux de ces prescriptions. Concernant la prescription rejetée par le pharmacien et validée par le système, c'est le pharmacien qui a raison. Parmi les trois prescriptions équivalentes qui n'ont pas été validées par le système, le système a raison sur 2 de ces prescription. Enfin, parmi les cinq prescriptions qui n'ont pas été soumises à l'avis du pharmacien et non-validées par le système, le système a raison sur trois de ces prescriptions.

Pour récapituler, parmi les seize prescriptions qui ont été détectées comme des prescriptions non-valides par notre système et qui ont été administrées à des patients, le système a raison sur 7 de ces prescriptions.

4.4.1.3 Gestion de l'évolution de l'ontologie métier

Comme nous l'avons décrit dans les règles présentées ci-dessus, les notations "TLJ" et "1" sont utilisées pour signaler qu'un médicament doit être administré toutes les 24 heures. Afin d'unifier le système de notation, il a été convenu de ne plus utiliser la notation "TLJ" comme unité de temps et de garder la notation numérique "1" . Ceci consiste donc en un changement de l'ensemble de valeur que peut avoir la propriété *relatedTimeUnit*, dans ce qui suit nous allons analyser l'impact d'un tel changement et détecter les incohérences qui peuvent être causées au niveau des règles en appliquant notre approche $\mathcal{MDR}$.

Dans l'ontologie de validation des prescriptions, la propriété *relatedTimeUnit* est définie comme suit :

- domaine (*i.e.* `<rdfs:domain>`) : le concept **DosageRegimenPhase** ;
- range (*i.e.* `<rdfs:range>`) : un concept anonyme définie par une énumération `<owl:oneOf>` ayant les éléments suivants : "TLJ", "1" ou "2".

Lors du chargement de l'ontologie dans ODM, cette propriété est transformée dans le BOM en tant qu'attribut de la classe DosageRegimenPhase ayant un domaine de littéraux composé des éléments de l'énumération (voir Figure 4.11).

Le changement à appliquer est un changement d'énumération qui consiste à supprimer l'élément "TLJ" de l'ensemble des valeurs que peut avoir la propriété *relatedTimeUnit*. En se basant sur l'ensemble des règles métier décrites précédemment, ce changement impliquera que (1) la règle `GLUCOPHAGE 1 (Metformin)- 1000 -TABLET` (voir Figure 4.6) ne sera plus exécutée auquel cas la **Prescription 1** prescrite au `Patient 1` ne déclenchera plus d'alerte et sera donc administrée au patient et (2) l'exécution de la règle `TimeUnit-GLUCOPHAGE-1000-TABLET` (voir Figure 4.8), assignera une valeur hors de son domaine à la propriété *relatedTimeUnit*. Pour réparer ces incohérences il faut modifier les valeurs testées et assignées à la propriété dans les règles impactées. L'application de l'approche $\mathcal{MDR}$ sera comme suit :

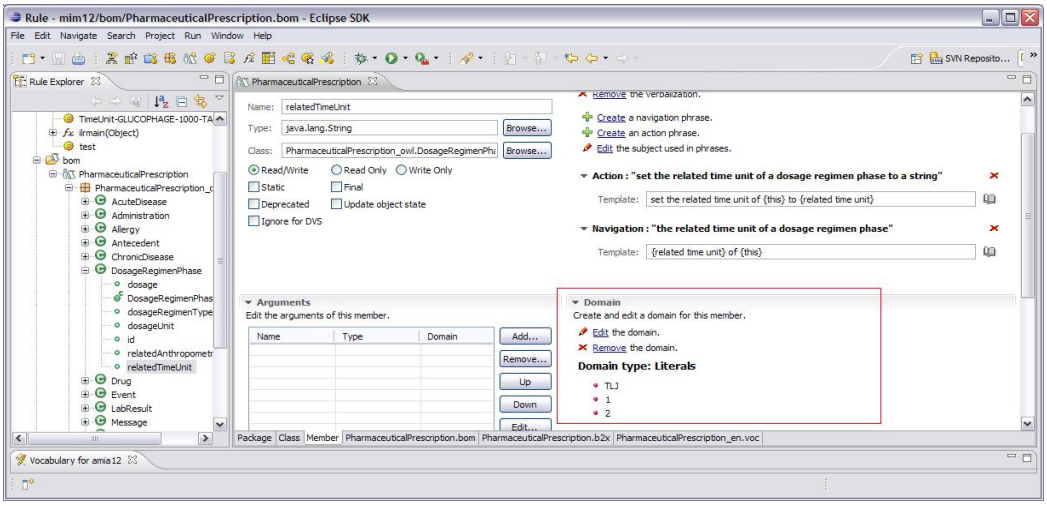

FIGURE 4.11 – Visualisation de la propriété *relatedTimeUnit* dans le BOM.

1. **Spécification du changement**

   Comme nous l'avons décrit dans la Section 3.5.2.1, cette étape consiste à modéliser, à l'aide des patrons de changement, le changement à appliquer à l'ontologie métier. Il s'agit de modifier l'ensemble des élément de l'énumération de la propriété *relatedTimeUnit*. Marc, avec l'aide de Alice, crée dans l'ontologie $\mathcal{MDR}$ une instance du concept `EnumerationChange` qui a pour entité de changement la propriété *relatedTimeUnit*, l'objet du changement est le constructeur OWL, `owl:oneOf` qui a une énumération à laquelle il faut supprimer l'élément "TLJ". Le patron de changement instancié est le patron de changement d'énumération qui modélise la modification d'une énumération (voir Figure 3.23). Cette instance est illustrée ci-dessous (voir Figure 4.12).

2. **Détection des règles impactées**

   Cette étape consiste à détecter les règles impactées par un changement en fonction de l'entité et de l'objet du changement, en analysant l'arbre syntaxique de chaque règle (voir Section 3.5.2.2). Dans cet exemple, deux règles seront impactées, (1) la règle `GLUCOPHAGE 1 (Metformin)- 1000 -TABLET` (voir Figure 4.6) qui teste dans sa partie condition si l'unité de temps est "TLJ" et (2) la règle `TimeUnit-GLUCOPHAGE-1000-TABLET` (voir Figure 4.8) qui assigne dans sa partie action, la valeur "TLJ" à la propriété *relatedTimeUnit*. La règle `GLUCOPHAGE 2 (Metformin)-`

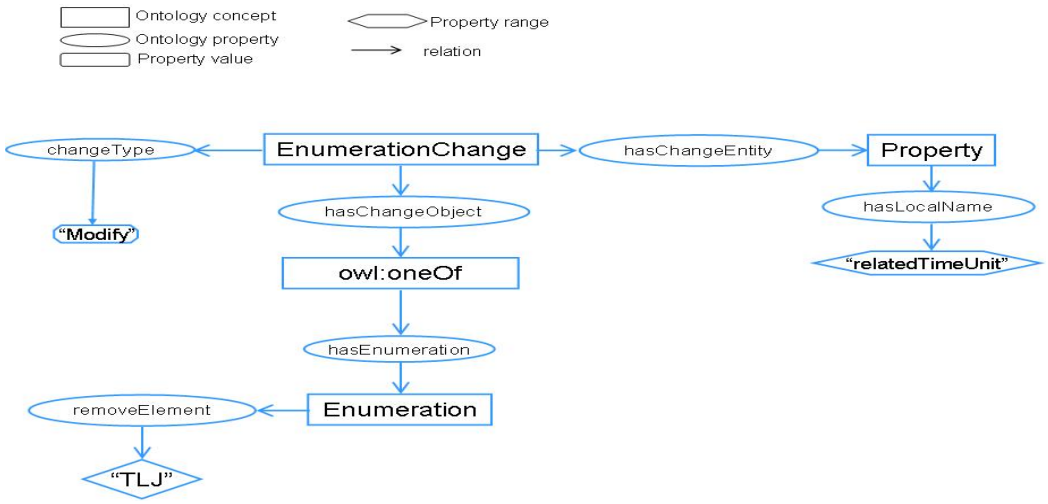

FIGURE 4.12 – Instance du patron de changement d'énumération : modifier une énumération.

`1000 -TABLET` (voir Figure 4.7) ne sera pas impactée puisque dans les valeurs de test sur la propriété *relatedTimeUnit* il y a soit "TLJ" soit "1".

Lors de l'exécution du module de détection des règles impactées, l'instance du patron de changement d'énumération est mise à jour avec l'ensemble des règles impactées (voir Figure 4.13).

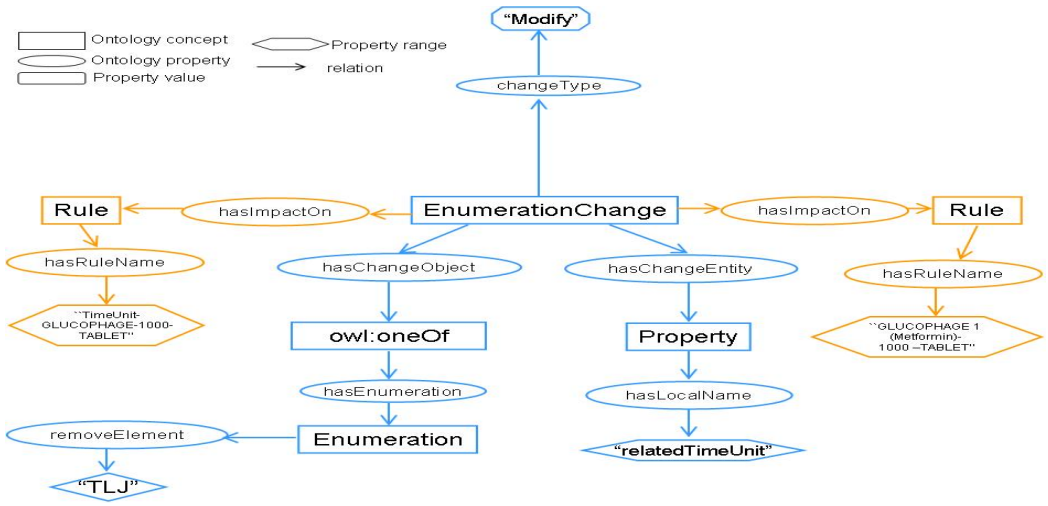

FIGURE 4.13 – Instance du patron de changement d'énumération : détection des règles impactées

3. **Détection des incohérences sur les règles**

   Cette étape consiste à détecter les incohérences à l'aide des *règles de détection des incohérences* (voir Section 3.5.2.3). Dans cet exemple deux incohérences sont détectées. La première est détectée sur la règle `GLUCOPHAGE 1 (Metformin)- 1000 -TABLET` (voir Figure 4.6) qui ne sera plus exécutée car sa partie condition ne pourra plus être vérifiée après l'application du changement, puisque la valeur "TLJ" sera plus utilisée. Cette incohérence *règle jamais applicable* est détectée par la règle suivante :

```
when {
     enumerationChange : EnumerationChange();
      rule : Rule() in enumerationChange.impactOn;
      ruleProperty : Property();
      enumeration : Enumeration();
      conditionPart : ConditionPart();
      condition : Condition();

      evaluate (enumerationChange.changeType in {"Add", "Modify"}
         && enumerationChange.changeObject.equals(enumeration)
         && ruleProperty.hasOWLConstruct(enumeration)
         && enumerationChange.hasImpactOn(rule)
         && rule.hasRulePart(conditionPart)
         && conditionPart.hasPart(condition)
         && condition.hasEntity(ruleProperty)
         && ! (ruleProperty.testedPropertyValue in
                     enumerationChange.newCollection.element)  );
}
then {
       RuleNeverApplied rna = new RuleNeverApplied();
   enumerationChange.add_hasImpact(rna);
       rna.detectedOn(rule);
}
```

La deuxième incohérence est détectée sur la règle `TimeUnit-GLUCOPHAGE-1000-TABLET` (voir Figure 4.8), qui assigne une valeur hors du domaine de la propriété ce qui génère une incohérence de type *Violation de domaine*, détectée par la règle suivante :

```
 when {
```

```
enumerationChange: EnumerationChange();
ruleProperty:Property();
     enumeration: Enumeration();
      actionPart: ActionPart();
      action : Action();
      rule: Rule() in enumerationChange.impactOn;

      evaluate ( enumerationChange.changeType in {"Add", "Modify"}
         && enumerationChange.changeObject.equals(enumeration)
         && ruleProperty.hasOWLConstruct(enumeration)
         && enumerationChange.hasImpactOn(rule)
         && rule.hasRulePart(actionPart)
         && actionPart.hasPart(action)
         && action.hasEntity(ruleProperty)
         && ! (ruleProperty.assignedPropertyValue in
                     enumerationChange.newCollection.element) );
}
then {
       DomainViolation domainViolation = new DomainViolation();
       enumerationChange.add_hasImpact(domainViolation);
       domainViolation.detectedOn(rule);
}
```

L'exécution de ces deux *règles de détection des incohérences* met à jour le modèle de changement avec l'ensemble des incohérences détectées (voir Figure 4.14).

4. **Proposition des réparations des incohérences**

   Cette étape consiste à proposer des solutions, à l'aide des *règles de réparation*, pour résoudre les incohérences détectées (voir Section 3.5.2.4). Comme nous l'avons déjà dans le chapitre précédent, il y a des réparations qui sont proposées par défaut telle que, la suppression de la condition ou l'action de la règle qui cause l'incohérence, la suppression de la règle qui cause l'incohérence ou bien l'application le changement sans tenir compte des incohérences détectées. Néanmoins, pour chaque paire <Changement, Incohérence>, il y a des réparations spécifiques.

   Dans le cas de cet exemple, deux réparations seront proposées. Pour réparer l'incohérence *règle jamais applicable* causé par un changement d'énumération, la réparation proposée est de modifier la valeur testée

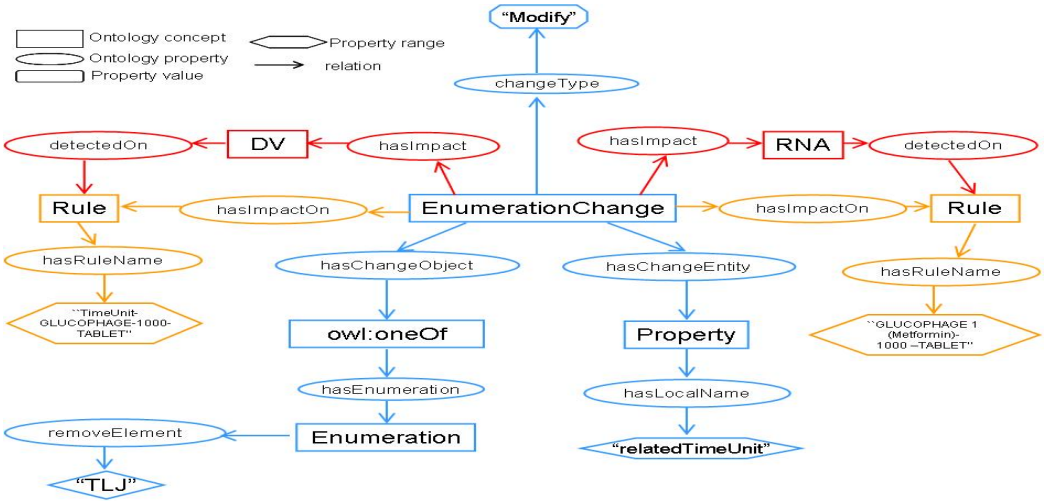

FIGURE 4.14 – Instance du patron de changement d'énumération : détection des incohérences.

sur la propriété *relatedTimeUnit* par une autre valeur appartenant aux éléments de l'énumération. Cette réparation est proposée par la règle suivante :

```
when {
     enumerationChange: EnumerationChange();
     rna : RuleNeverApply() in enumerationChange.impact;
    ruleProperty : Property() from enumerationChange.changeEntity;
    testedValue : TestedValue() from ruleProperty.testValue;

    evaluate ((domain_violation in enum_change.impact
                   || rule_never_apply in enum_change.impact)
                  && change_tested_value.approved);
}
then {
 repair :ReasoningOntology_owl.Repair();
rule_property.testedPropertyValue = change_tested_value.newLitteralValue;
 }
```

Pour réparer l'incohérence *Violation de domaine*, la réparation proposée est de modifier la valeur assignée à la propriété par une autre valeur appartenant à l'ensemble des éléments de l'énumération. Cette réparation est proposée par la *règle de réparation* suivante :

```
when {
     enumerationChange: EnumerationChange();
      rna : RuleNeverApply() in enumerationChange.impact;
    ruleProperty : Property() from enumerationChange.changeEntity;
    testedValue : TestedValue() from ruleProperty.testValue;

    evaluate ((domain_violation in enum_change.impact
                  || rule_never_apply in enum_change.impact)
                && change_tested_value.approved);
}
then {
 repair :ReasoningOntology_owl.Repair();
rule_property.testedPropertyValue = change_tested_value.newLitteralValue;
 }
```

L'exécution des *règles de réparation* met à jour le modèle de changement avec les réparations de chaque incohérence, choisies par l'utilisateur (voir Figure 4.15).

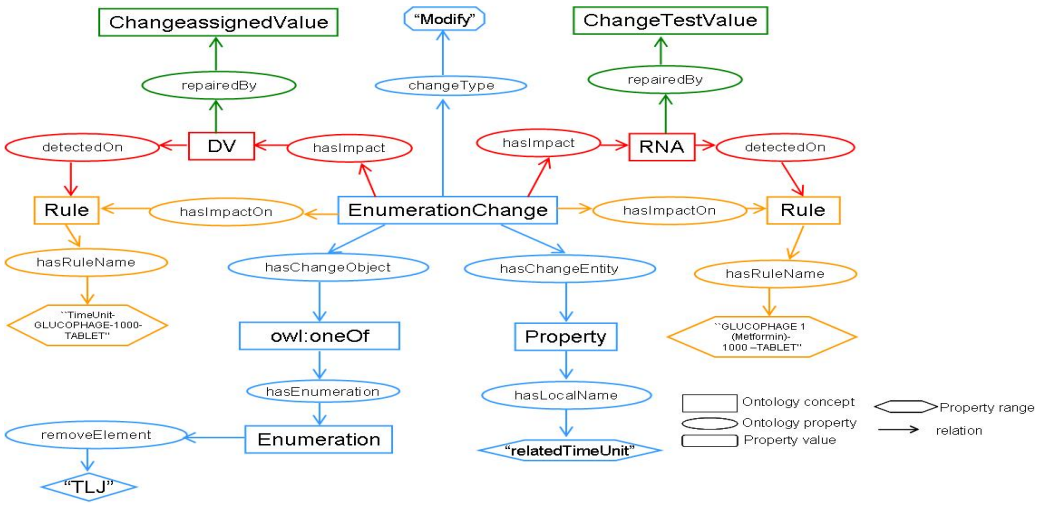

FIGURE 4.15 – Instance du patron de changement d'énumération : proposition des réparations.

4.4.2 Cas d'utilisation de Audi

L'exemple décrit dans cette section est extrait du cas d'utilisation de Audi [5], un des partenaires du projet ONTORULE. Le but de ce cas d'utilisation est de vérifier si les tests effectués sur les différents équipements d'un véhicule, lors de son processus de construction, sont conformes aux normes imposées par la réglementation européenne.

Le texte de la réglementation est organisé en un ensemble de sections qui décrivent les conditions nécessaires pour qu'un équipement soit conforme à la régulation. Nous nous sommes intéressé aux tests effectués sur les ceintures de sécurité, et en particulier au Micro-slip test qui teste sur la rigidité des ceintures.

Parmi les conditions nécessaires pour effectuer le micro-slip test, il faut que l'échantillon sur lequel le test va être effectué soit maintenu, pendant au minimum 24 heures, dans une atmosphère ayant une température comprise entre 15 et 25 degrés et une humidité comprise entre 60% et 70%. Le test doit être effectué à une température comprise entre 15 et 30 degrés. Si les conditions précédemment citées sont réunies alors la ceinture de sécurité est conforme à la régulation.

4.4.2.1 Écriture et exécution des règles métier

Afin de modéliser les conditions décrites ci-dessus avec des règles métier, nous avons utilisé l'ontologie fournit par le partenaire (voir Figure...). Dans cette ontologie, nous avons les définitions suivantes :

Strap $\sqsubseteq$ **BuildOfMaterial** $\sqsubseteq$ **Material** $\sqsubseteq$ **Function**
isConformTo (**Function**, **Section**)
isComposedOf (**Regulation**, **Section**)
approved (**MicroSlipTest**, boolean)
hasAtmosphere (**Strap**, **Atmosphere**)
hasAtmosphere (**MicroSlipTest**, **Atmosphere**)
hasTemperature (**Atmosphere**, float)
hasHumidity (**Atmosphere**, float)

5. http ://www.audi.com

Après avoir importé cette ontologie dans ODM, Alice, l'expert métier, peut éditer les règles qui permettent de tester les conditions de déroulement du **MicroSlipTest**.

La première règle teste si l'échantillon, sur lequel le **MicroSlipTest** est effectué, a été maintenu pendant au minimum 24 heures, dans un atmosphère ayant une température comprise entre 15 et 25 degrés et une humidité comprise entre 60% et 70%. Dans ce cas la ceinture de sécurité est conforme au conditions de la régulation (voir Figure 4.16).

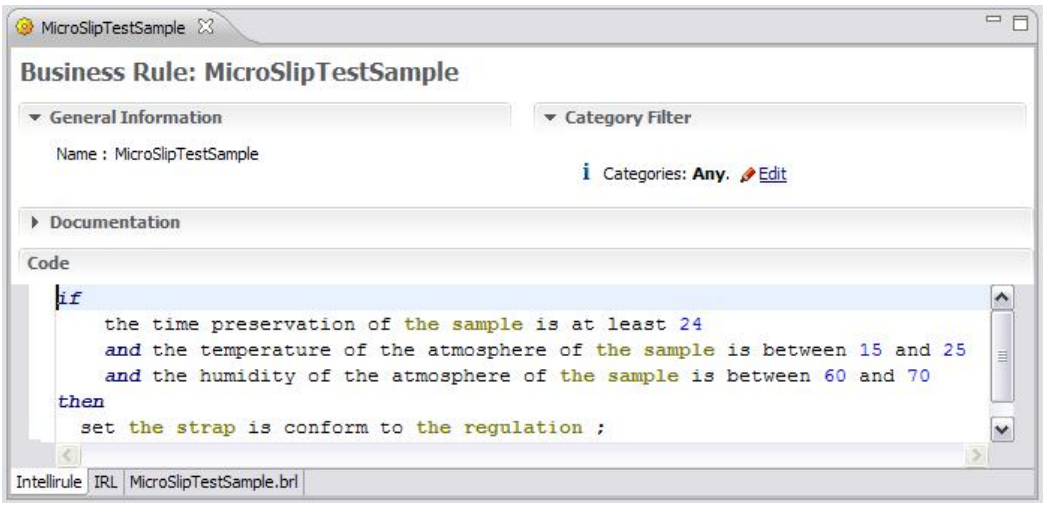

FIGURE 4.16 – Règle : Micro-slip test sample.

La deuxième règle teste si le **MicroSlipTest** est effectué à une température comprise entre 15 et 30 alors il est considéré comme valable (voir Figure 4.17).

4.4.2.2 Gestion de l'évolution de l'ontologie métier

Un des changements qui a été effectué sur l'ontologie métier est de supprimer la relation de sous-classe entre le concept **Function** et le concept **Material**. Dans ce cas, **Material** et ses sous concepts (*i.e.* **BuildOfMaterial** et **Strap**) n'auront plus les propriétés héritées du concept **Function**. Parmi ces propriétés, il y a la propriété *isConformTo* qui est héritée par le concept **Strap** et utilisée dans la règle `Micro-slip test sample` (voir Figure 4.16). Après l'application du changement, le concept **Strap** n'aura plus la propriété *isConformTo* ce qui rendra la règle invalide. L'application de l'approche $\mathcal{MDR}$ sera

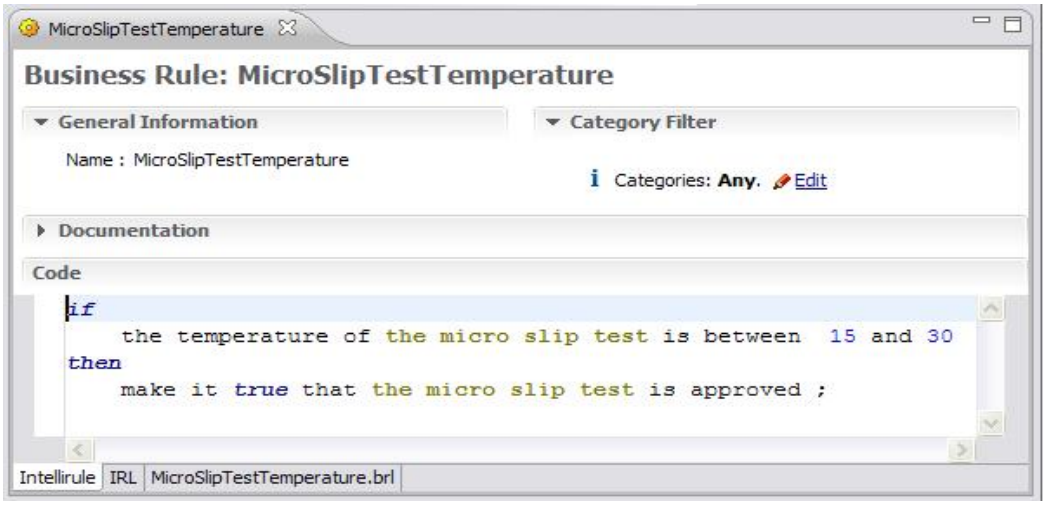

FIGURE 4.17 – Règle : Micro-slip test temperature.

comme suit :

1. **Spécification du changement**

   Le changement consiste à supprimer la relation de sous-classe entre les concepts **Function** et **Material**. Marc, en collaborant avec Alice, crée dans l'ontologie $\mathcal{MDR}$ une instance du concept `SubclassChange` qui a pour entité de changement le concept **Material** qui a le super concept **Function**. L'objet du changement est le constructeur OWL *owl :subClassOf* (voir Figure 4.18).

2. **Détection des règles impactées**

   Dans cet exemple, une seule règle est impactée. Le module de détection des règles impactées cherche toutes les règles qui utilisent l'entité du changement (*i.e.* le concept **Material**) ainsi que ses sous-concepts et détecte les propriétés avec lesquelles ils sont utilisés. Si les propriétés sont héritées alors la règle est impactée. Dans cet exemple, c'est la règle `Micro-slip test sample` qui est impactée (voir Figure 4.19).

3. **Détection des incohérences**

   La règle `Micro-slip test sample` utilise la propriété héritée *isConformTo* avec le concept **Strap**. Comme nous l'avons déjà dit dans la section 3.5.2.3, chaque changement doit vérifier des contraintes pour éviter les incohérence. La contrainte qui doit être vérifiée par le changement

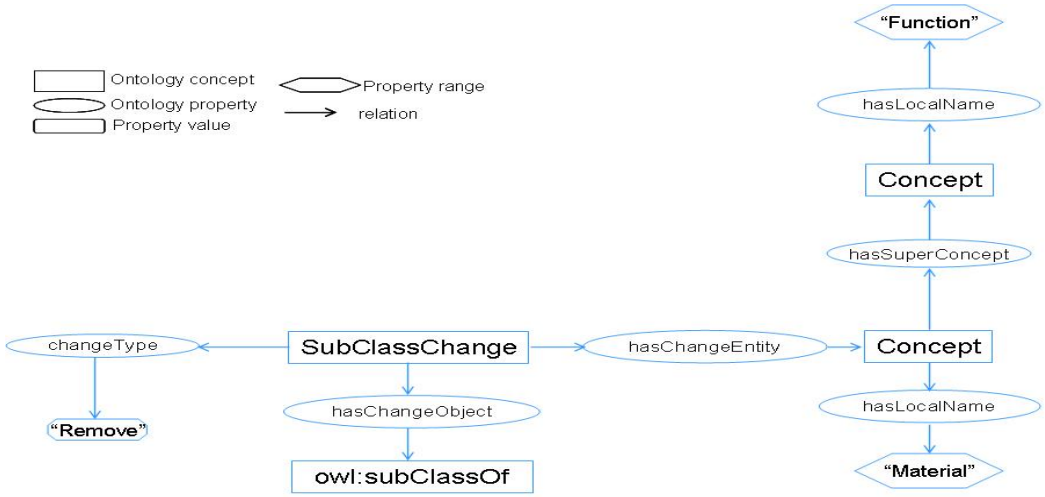

FIGURE 4.18 – Instance du patron de changement de sous classe : supprimer une relation de sous classe.

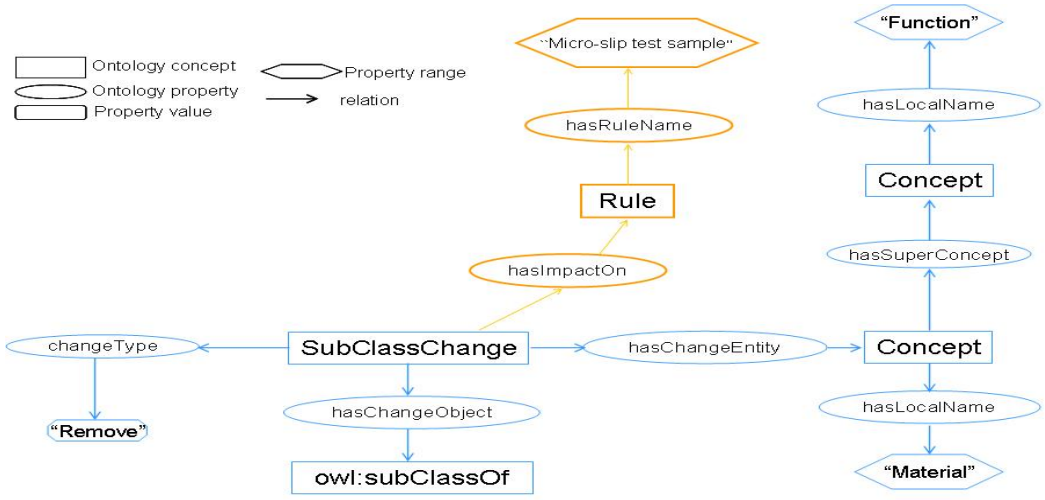

FIGURE 4.19 – Instance du patron de changement de sous classe : détection des règles impactées.

de sous-classe est que le concept de changement et ses sous-concepts n'utilisent pas une des propriétés héritées dans une règle, auquel cas une incohérence de type *règle invalide* sera détectée.

Cette incohérence est détectée par la *règle de détection des incohérences* suivante :

```
when {
```

```
subClassChange: SubClassChange();
   changeEntity : Concept() from subClassChange.changeEntity;
   superConcept : Concept() in changeEntity.superConcept ;
   br : subClassChange.impactOn;
   prop : Property() in superConcept.conceptProperty;

 evaluate (  changeEntity.ruleProperty.equals(prop)  );
}
then {
InvalidRule invalidRule = new InvalidRule();
subClassChange.add_hasImpact(invalidRule);
    invalidRule.add_detectedInRule(br);
}
```

L'exécution de cette règle met à jour le modèle de changement donné en entrée au système avec l'incohérence détectée et la règle sur laquelle elle a été détectée (voir Figure 4.20).

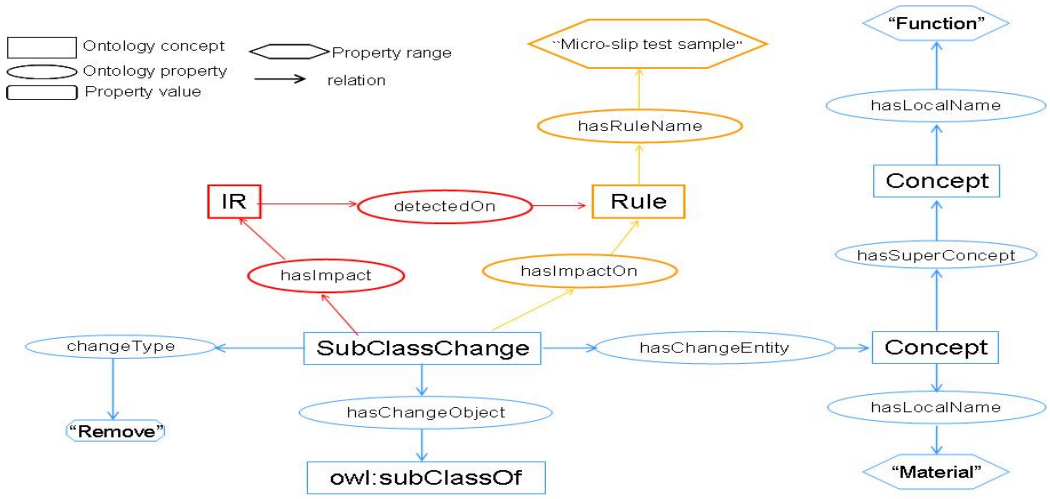

FIGURE 4.20 – Instance du patron de changement de sous-classe : détection des incohérences.

4. **Proposition des réparations**

   Comme nous l'avons décrit dans la Section 3.5.2.4, il y a des réparations par défaut qui consistent soit à supprimer l'action ou la condition qui cause l'incohérence, soit à supprimer la règles incohérente, soit à appliquer le changement sans tenir compte des incohérences.

Le changement spécifique dans ce cas est de rajouter la propriété *isConformTo* au concept **Material**, ainsi le domaine de la propriété *isConformTo* sera l'union des concepts **Function** et **Material**. Cette réparation est proposée par la *règle de réparation* suivante :

```
when {
   subClassChange: SubClassChange();
   changeEntity : Concept() from subClassChange.changeEntity;
   superConcept : Concept() in changeEntity.superConcept ;
   rule : Rule() in subClassChange.impactOn;
   property : Property() in superConcept.conceptProperty;
   invalidRule : InvalidRule() in subCassChange.impact ;

 evaluate (  changeEntity.ruleProperty.equals(property)  );
}
then {
   changeEntity.add_hasConceptProperty(property);
 }
```

L'exécution de cette règle met à jour le modèle du changement avec la réparation choisi par l'utilisateur (voir Figure 4.21).

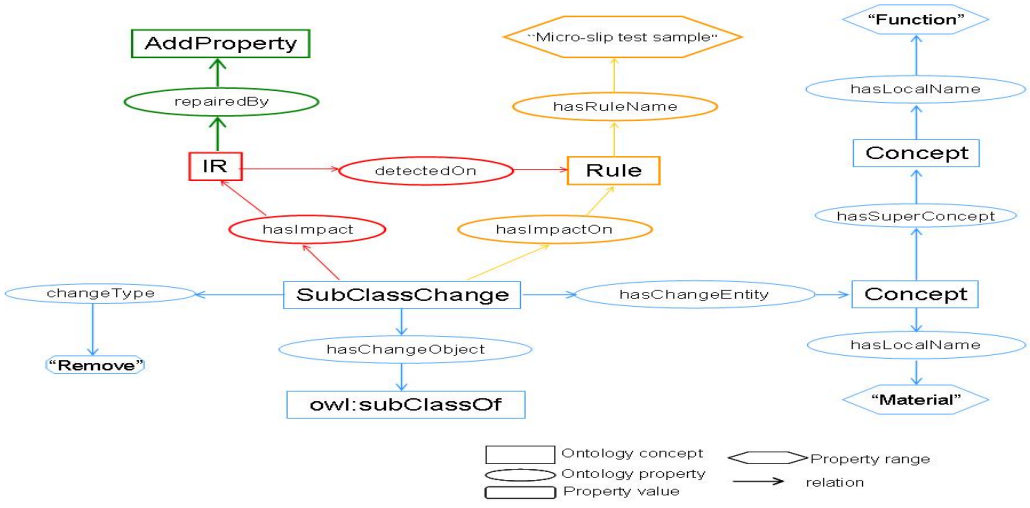

FIGURE 4.21 – Instance du patron de changement de sous classe : proposition des réparations.

4.5 Conclusion

Dans ce chapitre, nous avons présenté les expérimentations effectuées pour tester notre approche. Ces expérimentations ont été élaborées dans différents domaines d'application à savoir (1) la validation clinique des prescriptions médicamenteuses, un cas d'application effectué en collaboration avec l'Hôpital Européen George Pompidou, (2) la validation selon les réglementations européennes, des tests effectués sur les ceintures de sécurité des voitures, un cas d'application effectué en collaboration avec Audi. De plus, vu que $\mathcal{MDR}$ gère l'évolution d'applications développées à base d'ontologies et de règles et qu'elle même est développée à base d'une ontologie et d'un ensemble de règles, il est possible d'utiliser l'approche $\mathcal{MDR}$ pour gérer ses évolutions. Ceci reviendrait à modéliser les changements de l'ontologie $\mathcal{MDR}$ et de déclencher les règles de détection des incohérences et les règles de réparation pour détecter et réparer les problèmes de cohérence causés par le changement.

Chapitre 5

Conclusion et perspectives

Sommaire

5.1 Synthèse du travail

Vu la rapidité de l'évolution des connaissances, la maintenance des systèmes d'information est devenue de plus en plus difficile à gérer. Afin d'assurer une flexibilité de ces systèmes, nous proposons une approche qui permet de représenter les connaissances des domaines par des modèles de représentation des connaissances plutôt que de les coder en dure dans l'application du domaine. Ceci serait de nature à assurer une meilleure flexibilité des systèmes d'information, à faciliter leur maintenance et permettre aux experts métier de gérer eux même l'évolution des connaissances de leur domaine.

5.2 Contributions

Le travail proposé permet, à partir d'une modélisation OWL des connaissances d'un domaine, de modéliser les contraintes et d'automatiser des décisions au moyen de règles métier et de gérer l'impact de l'évolution des ontologies sur les règles.

Contribution 1– Pour atteindre l'objectif fixé, nous avons démarré notre travail par l'implémentation d'une approche, qui permet d'éditer et d'exécuter des règles métier à partir d'ontologies OWL (voir Section 4.2).

Cette approche implémentée sous forme de plug-in pour ODM, permet d'y importer des ontologies OWL et d'utiliser toutes les fonctionnalités offertes par le système pour l'édition des règles et une interaction entre le moteur de règles et le raisonneur d'ontologie pour l'exécution des règles.

Contribution 2 – Nous avons ensuite implémenté l'approche $\mathcal{MDR}$ sous forme de plug-in ODM qui permet de charger les changements de l'ontologie métier modélisés dans l'ontologie $\mathcal{MDR}$. Quand l'utilisateur sélectionne le changement à appliquer, les règles métier impactées par ce changement son détectées et les *règles de détection des incohérences* sont déclenchées et une liste des incohérences détectées est affichée. Quand l'utilisateur sélectionne les incohérences à réparer les *règles de réparation* sont déclenchées et une liste de réparations est affichée. L'utilisateur peut soit choisir une des réparations proposées soit choisir d'appliquer le changement sans appliquer de réparation.

Contribution 3 – L'ontologie $\mathcal{MDR}$; cette ontologie propose une spécification d'un changement indépendamment des modèles de représentation des connaissances. Cette spécification peut être utilisée pour modéliser des changements de différents modèles de représentation des connaissances pouvant être représentés dans un langage formel. Étant donné que nous travaillons sur la gestion de l'impact de l'évolution d'ontologies sur sur les règles, l'ontologie $\mathcal{MDR}$ propose une classification des constructeurs du langage OWL et une classification des changements d'ontologies OWL qui repose sur la classification des constructeurs. Cette ontologie propose aussi une classification des incohérences qui peuvent impacter des règles et une ensemble de réparations pour les résoudre.

Contribution 4 – Les patrons d'incohérences ; ces patrons permettent de spécifier les contraintes des changements. Dans notre travail, les patrons d'incohérences proposent une spécification des contraintes que chaque changement d'ontologie doit vérifier pour qu'il ne cause pas des problèmes de cohérence sur les règles.

Contribution 5 – Le langage $\mathcal{MDR}$; ce langage, qui repose sur une syntaxe XML, permet d'unifier la spécification des changements et des contraintes de changement. Dans notre travail, nous utilisons ce langage pour spécifier des changements et les contraintes de changements

d'ontologies.

5.3 Limites et perspectives

- Auto-application de l'approche $\mathcal{MDR}$ pour le gestion des évolution de l'approche $\mathcal{MDR}$. Vu que $\mathcal{MDR}$ gère l'évolution d'applications développées à base d'ontologies et de règles et qu'elle même est développée à base d'une ontologie et d'un ensemble de règles, il est possible d'utiliser notre approche pour gérer ses évolutions. Ceci reviendrait à modéliser les changements de l'ontologie $\mathcal{MDR}$ et de déclencher les règles de détection des incohérences et les règles de réparation pour détecter et réparer les problèmes de cohérence causés par le changement.
- Dans ce travail, nous nous sommes focalisés sur l'impact de l'évolution des ontologies sur les règles en négligeant l'impact de l'évolution des ontologies sur les individus (*i.e.* la $ABox$) et l'impact de l'exécution des règles sur les individus. Comme nous l'avons expliqué dans la section 4.2, la $ABox$ est mise à jour en fonction de l'exécution des règles. Une problématique intéressante à creuser est d'analyser la cohérence de la $ABox$ par rapport à l'ontologie après l'exécution des règles. Une autre problématique aussi intéressante est, en supposant que la $ABox$ évolue de manière cohérente, en fonction de l'évolution de la $TBox$, d'assurer que si l'exécution d'une règle r_1 été déclenché par l'individu i_1, cette règle sera toujours déclenchée par le même individu même après l'évolution de l'ontologie. Cette problématique peut conduire à une nouvelle vision pour la gestion de l'impact des changements qui consiste à faire évoluer les règles en fonction de l'évolution de la $ABox$ et de la $TBox$ plutôt qu'en fonction de l'évolution de la $TBox$ seule.
- Comme nous l'avons précisé dans le chapitre 3, $\mathcal{MDR}$a été inspiré de ONTO-EVO^A. ONTO-EVO^A propose une solution pour gérer l'évolution d'une ontologie tout en maintenant sa cohérence. L'idée générale de $\mathcal{MDR}$ est proposer une solution qui permet de gérer l'évolution des modèles de représentation des connaissances en général ainsi que l'impact de cette évolution sur d'autres modèles ou ressources qui en dépendent. Nous avons ensuite appliqué cette idée pour la gestion de l'évolution des ontologies et leurs impacts sur des règles qui en dépendent et nous

avons proposé le langage $\mathcal{MDR}$ qui permet de spécifier les changements d'ontologies, leur impact ainsi que des solutions qui permettent de les réparer. Il serait donc intéressant d'avoir un langage, un *Domain Specific Language*, qui permettrait que spécifier de manière générale, des changements de modèle de représentation des connaissances, les problèmes qu'ils peuvent causer ainsi que les solutions pour les résoudre.

- Dans la section 4.2, nous avons présenté une approche d'intégration d'ontologies et de règles qui repose sur une transformation du modèle OWL vers un modèle orienté objets (*i.e.* le BOM de ODM). Lors de cette transformation, vu la différence de la sémantique de représentation des connaissances entre ces deux modèles, certaines connaissances modélisées dans l'ontologie (*e.g.* que la cardinalité, la complémentarité, la disjonction,...) ne sont pas prises en compte lors de l'édition et de l'exécution des règles métier. Une solution qui permettrait d'éviter cette perte d'information est l'utilisation de *règles métier d'inférence*, qui assureront l'inférence des connaissances «perdues» lors de la transformation (El Ghali *et al.*, 2012). La problématique serait donc la génération automatique des *règles métier d'inférences*, tout en assurant leur cohérence et la cohérence des connaissances qui vont être inférées, en fonction des axiomes existant dans l'ontologie métier et qui ne peuvent être calculés par un moteur de règles. Ces *règles métier d'inférence* devront être exécutées en priorité, ce qui permettra d'inférer toutes les connaissances nécessaires avant l'exécution des règles métier écrites par l'expert métier. Il est à noter que cette problématique n'est pas liée uniquement à l'utilisation de ODM mais qu'elle s'applique à tous les systèmes de gestion des règles métier étant donné qu'ils reposent tous des modèles propriétaires orienté objets, pour la représentation des connaissances métier
- Les changements pris en compte par l'approche $\mathcal{MDR}$ ne couvrent pas tous les changements qui peuvent impacter une ontologie. Les changement traités actuellement sont : la suppression d'une relation de sous-classe, l'ajout et la suppression d'une entité de l'ontologie, le changement de la restriction `owl:allValuesFrom`, le changement du domaine ou du co-domaine d'une propriété et les changements d'énumérations. De même, pour les problèmes causés par les changements, seuls la *violation de domaine*, *règles jamais applicables* et les *règles non-valides* peuvent être détectées. Il serait

donc intéressant d'analyser l'impact des autres changements d'ontologie sur les règles, ce qui reviendrait à (1) définir les patrons de changement qui permettent de les spécifier, (2) définir les contraintes que ces changements doivent vérifier et détecter des incohérences qu'ils peuvent gérer et (3) identifier les réparations qui peuvent résoudre ces incohérences.

- La spécification des contraintes de changement est une tâche très complexe et coûteuse en temps. De même, la spécification d'une instance de changement par l'utilisateur peut s'avérer un peu difficile. L'utilisation du langage $\mathcal{MDR}$, via des interface orienté utilisateurs et un *parsing* de ce langage permettra de générer automatiquement des instances de changement dans le format $\mathcal{MDR}$ et permettra aussi la génération automatique des contraintes de changement.
- Le *parsing* du langage $\mathcal{MDR}$ permettra d'assurer la génération automatique des patrons d'incohérence et de réparation. Mais pour cela, il faudra d'abord définir une spécification des incohérences et des réparations. Cette spécification permettra de définir en fonction des contraintes de changement qui ne sont pas respectées, le problème de cohérence qui va être généré et la réparation pour le résoudre.
- Le fait de modéliser les changements en tant qu'individus de l'ontologie $\mathcal{MDR}$ permet la traçabilité des changements appliqués à l'ontologie métier, les incohérences générées ainsi que les réparations choisies. Ces informations pourront être utilisées pour prédire à chaque nouveau changement quelle incohérences il va générer et quelle réparation l'utilisateur va choisir. Ou encore, en fonction des réparations choisies pour les utilisateurs, le système pourra proposer de nouveaux changements.
- L'étape de réparation permet à l'utilisateur de, soit choisir une des réparations proposées soit appliquer les changements sans réparation ; Il serait intéressant de permettre aussi à l'utilisateur de proposer lui même la réparation qu'il veut appliquer dans le cas où aucune des réparations proposées par le système ne correspond à ses attentes.
- La perception de l'étendue de l'impact des changements d'ontologies sur les règles est assez difficile pour les utilisateurs surtout lorsque la base de règle contient un très grand nombre de règles. L'utilisation de techniques de visualisation permettra de visualiser les dépendances existantes entre l'entité de changement et les règles qui en dépendent.
- Les cas d'application utilisés pour expérimenter notre travail ne fournissent pas de grandes ontologies ou de grandes bases de règles. Il serait donc intéressant de voir l'applicabilité de $\mathcal{MDR}$ sur une plus grande échelle.

Bibliographie

ISO 9241. (2008). Ergonomics of human-system interaction. *Part 151 : Guidance on World Wide Web user interfaces.* 110

Antoniou G., Damasio C. V., Grosof B., Horrocks I., Kifer M., Maluszynski J. & Patel-Schneider. P. F. (2004). Combining rules and ontologies : A survey. *Technical Report IST506779/Linkoping/I3-D3/D/PU/a1, Linkoping University.* http ://rewerse.net/publications/. 45, 46, 48, 49, 50

Aranguren M., Antezana E., Kuiper M. & Stevens R. (2008). Ontology design patterns for bioontologies : a case study on the cell cycle ontology. *BMC Bioinformatics*, **9**. Supplement 5. 43

Aussenac-Gilles N. & Jacques M. (2006). Designing and evaluating patterns for ontology enrichment from texts. In *Managing Knowledge in a World of Networks, EKAW'06*, p. 4248 : Springer Berlin Heidelberg. 42

Bach thanh L. (2006). *Construction d'un Web sémantique multi-points de vue.* PhD thesis, CMA Centre de Mathématiques Appliquées, ENSMP.

Berstel B. & M.Leconte (2010). Using constraints to verify properties of rule programs. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on Software Testing, Verification and Validation*, p. 349 – 354. 73

Berstel Da Silva B. (2012). *Verification of Business Rules Programs.* PhD thesis, University of Freiburg.

Bloehdorn S., Haase P., Sure Y. & Voelker J. (2006). Ontology evolution. In *Semantic Web Technologies, Trends and research in Ontology-based Systems*, J. Davies, R. Studer, & P. Warren (Ed.s), p. 51–70 : John Wiley & Sons Publication. 39

Boley H. (2006). The ruleml family of web rule languages. In H. Springer-Verlag Berlin, Ed., *PPSWR'06 Proceedings of the 4th international*

conference on Principles and Practice of Semantic Web Reasoning, p. 1–17. 24

Boley H., Tabet S. & Wagner G. (2001). Design rationale for ruleml : A markup language for semantic web rules. In *SWWS 01*, p. 381–401.

Boussadi A., Bousquet C., Sabatier B., Caruba T., Durieux P. & Degoulet P. (2010). A business rules design framework for a pharmaceutical validation and alert system. *Methods of Information in Medicine*, **Vol. 50)**, 36–50. 113

Brickley D. & Guha R. (2004). Rdf vocabulary description language 1.0 : Rdf schema, w3c recommendation 10 february 2004. http ://www.w3.org/TR/rdf-schema/. Accessed on October, 2012. 27

Buschmann F., Meunier R., Rohnert H., Sommerlad P. & Stal M. (1996). *Pattern-oriented Software Architecture - A System of Patterns*. John Wiley & Sons, Chichester. 42

Castano S., Espinosa S., Ferrara A., Karkaletsis V., Kaya A. & Melzer S. (2007). Ontology dynamics with multimedia information : The boemie evolution methodology. In *Proceedings of the International Workshop on Ontology Dynamics (IWOD-07) at ESWC 07 Conference*, G. Flouris, & M. d'Aquin (Eds.), p. 41–54. 40

Chagnoux M., Hernandez N. & Aussenac-Gilles N. (2008). An interactive pattern based approach for extracting non-taxonomic relations from texts. In G. P. P. Buitelaar, P. Cimiano & M. S. Eds, Eds., *Proceedings of the workshop on Ontology Learning and Population associated to ECAI 2008 et OLP 2008*, p. 1–6. 42

Charlet J., Bachimont B. & Troncy. R. (2004). Ontologies pour le web sémantique. *Revue 3I : Information – Interaction – Intelligence*. 15

Chniti A., Dehors S., Albert P. & Charlet J. (2010). Authoring business rules grounded in owl ontologies. In M. D. et al. (Eds.), Ed., *RuleML 2010 : The 4th International Web Rule Symposium : Research Based and Industry Focused* : LNCS 6403, Springer-Verlag Berlin Heidelberg 2010.

Cimiano P. & Volker J. (2005). Text2onto - a framework for ontology learning and data-driven change discovery. In *Natural Language Processing and Information Systems*, volume 3513 of *A. Montoyo, R. Munoz, & E. Metais*, p. 227–238, Berlin, Germany : Springer-Verlag. 39

Clark P., Thompson J. & Porter B. (2000). Knowledge patterns. In A. G. Cohn, F. Giunchiglia & B. S. Eds, Eds., *KR2000 : Principles of Knowledge Representation and Reasoning*, p. 591–600. 43

Cooper A. & Reimannn M. (2003). *About Face 2.0 : The Essentials of Interaction Design*. Wiley Publishing, Inc., Indianapolis, Indiana. 110

Damjanovic V. (2009). Reengineering patterns for ontologizing the business processes. In *Proceedings of I-KNOW'09 and I-SEMANTICS'09*, p. 509–517 : Verlag der Technischen Universtät Graz. 43

de Bonis S. & Bellino C. (2011). D2.5 final usability report : evaluation and conclusions. *ONTORULE Delivrable, http ://ontorule-project.eu/deliverables.* 102

de Bruijn J., Eiter T., Polleres A. & Tompits H. (2007). Embedding non-ground logic programs into autoepistemic logic for knowledgebase combination. *IJCAI*.

Degoulet P., Marin L., Lavril M., Bozec C. L., Delbecke E., Meaux J.-J. & Rose L. (2003). The hegp component-based clinical information system. *International Journal of Medical Informatics*, **69**, 115–126.

Diouf M., Maabout S. & Musumbu K. (2007). Génération automatique de règles métier par enrichissement sémantique de modèles. *INFORSID*, p. 453–468. 14

Diouf M., Xiong J., Frenc C. & Winckler M. (2006). Automating guide-lines inspection from wed site specification to deployment. *CADUI*. 15

Djedidi R. (2009). *Approche d'évolution d'ontologie guidée par des patrons de gestion de changement*. PhD thesis, Université Paris-Sud XI Orsay. 41

Djedidi R. & Aufaure M. (2008). Enrichissement d'ontologies : maintenance de la consistance et évaluation de la qualité. In *19èmes journées francophones d'Ingénierie des Connaissances (IC'08)*. 42

Djedidi R. & Aufaure M. (2010). Onto-evO^Al ontology evolution approach guided by pattern modelling and quality evaluation. *Proceedings of the Sixth International Symposium on Foundations of Information and Knowledge Systems (FoIKS 2010)*. 39, 40, 41, 44, 59, 62, 77

Durkin J. (1994). *Expert Systems : Design and Development*. Macmillan New York.

Eder J. & Wiggisser K. (2007). Change detection in ontologies using dag comparison. In *Advanced Information Systems Engineering*, J.Krogstie, A.L. Opdahl, & G. Sindre(Ed.s), p. 21–35 : Springer. 39

Eiter T., Ianni G., Polleres A., Schindlauer R. & Tompits H. (2006). Reasoning with rules and ontologies. *Reasoning Web 2006*, p. 93–127.

Eiter T., Lukasiewicz T., Schindlauer R. & Tompits H. (2004). Combining answer set programming with description logics for the semantic web. *KR*.

El Ghali A., Chniti A. & Citeau H. (2012). Bringing owl ontologies to the business rules users. In L. N. in Computer Science, Ed., *Rules on the Web : Research and Applications*, volume 7438/2012 of *RuleML 2012 : The 6th International Symposium on Rules : Research Based and Industry Focused*, p. 62–76. 138

Fabro M. D., Albert P., Bézivin J. & Jouault F. (2009). Achieving rule interoperability using chains of model transformations. *Intl. Conference on Model Transformation, ICMT*, p. 249–259.

Fink M., Ghali A. E., Chniti A., Korf R., Schwichtenberg A., Lévy F., Pührer J. & Eiter T. (2011). D2.6 consistency maintenance. final report. *ONTORULE Delivrable, http ://ontorule-project.eu/deliverables.* 73

FLOURIS G. (2006). *On belief change and ontology evolution.* PhD thesis, University of Crete, Department of Computer Science, Heraklion, Greece. 40

FOO N. (1995). Ontology revision. In SPRINGER-VERLAG, Ed., *Conceptual Structures ; Third International Conference*, p. 16–31. 33

GAMMA E., HELM R., JOHNSON R. & VLISSIDES J. (1995). *Design Patterns - Elements of Reusable Object- Oriented Software.* Addison-Wesley. 42

GANGEMI A., CATENACCI C. & BATTAGLIA M. (2004). Inflammation ontology design pattern : an exercise in building a core biomedical ontology with descriptions and situations. In D. P. (ED.), Ed., *Ontologies in Medicine. IOS Press, Amsterdam.* 43

GARCIA-SILVIA A., GOMEZ-PEREZ A., SUAREZ-FIGUEROA M. & VILLAZON-TERRAZAS B. (2008). A pattern based approach for re-engineering non-ontological resources into ontologies. In S.-V. B. HEIDELBERG, Ed., *ASWC 2008*, p. 167–181. 43

GASEVIC D., DJURIC D. & DEVEDZIC. V. (2000). *Knowledge Representation : Logical, Philosophical, and Computational Foundations.* Brooks Cole and Pacific Grove (CA). 9, 16, 18, 21

GRAU B., HORROCKS I., MOTIK B., PARSIA B., PATEL-SCHNEIDER P. & SATTLER U. (2008). Owl2 : The next step forowl. *Journal Web Semantics : Science, Services and Agents on the World Wide Web*, **6**, 309–322. 30

GROSOF B. & HORROCKS. I. (2003). Description logic programs : Combining logic programs with description logic. *WWW 2003*, p. 48–57. 51

GROUP T. B. R. (2000). *Defining Business Rules What Are They Really ?* Rapport interne, GUIDE Business Rules Project, Final Report.

GRUBER T. (1993). A translation approach to portable ontologies. *KnowledgeAcquisition*, p. 199–220. 15

GUISSE A. (2013). *Plateforme d'aide à l'acquisition et à la maintenance des règles métiers à partir de textes réglementaires.* PhD thesis, Université Paris 13. 3

Haase P. & Stojanovic L. (2005). Consistent evolution of owl ontologies. In *The Semantic Web : Research and Applications*, volume 3532 of *A. Gomez-Perez, J. Euzenat (Eds.)*, p. 182–197, Berlin, Germany : Springer. 40

Hay D. (1996). *Data Model Patterns*. Dorset House Publishing. 42

Hay D. & Healy K. (2000). Guide business rules project. *Final Report. Revision 1.2, GUIDE International Corporation, Chicago. Now available from the Business Rules Group as Defining Business Rules What Are They Really ? 4th ed. (July 2000)*. 14

Hitzler P., Krotzsch M. & Rudolph S. (2009). Knowledge representation for the semantic web. KI 2009. 31

Kalou A. K., Pomonis T., Koutsomitropoulos D. A. & Papatheodorou T. S. (2010). Intelligent book mashup : Using semantic web ontologies and rules for user personalisation. In *ICSC 2010, Fourth IEEE International Conference on Semantic Computing*, p. 536–541.

Kalyanpur A., Jimenez D., Battle S. & Padget J. (2004). Automatic mapping of owl ontologies into java. *Frank Maurer and Gunther Ruhe (eds) Proc. of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004)*.

Kashyap V., Morales A. & Hongsermeier T. (2006). On implementing clinical decision support : Achieving scalability and maintainability by combining business rules and ontologies. In *AMIA Annual Symposium Proceeding*, p. 414–418.

Klein M. (2004). *Change management for distributed ontologies*. PhD thesis, Dutch Graduate School for Information and Knowledge Systems. xi, 34, 35, 37, 38, 39, 40, 44

Klein M. & Fensel D. (2001). Ontology versioning on the semantic web. In I. F. Cruz, S. Decker, J. Euzenat & D. L. M. (Eds.), Eds., *Proceeding of the first International Semantic Web Working Symposium (SWWS)*, p. 75–91. 34, 40

Klein M., Fensel D., Kiryakov A. & Ognyanov D. (2002a). Ontology versioning and change detection on the web. In *Proc. of the 13th European Conf. on Knowledge Engineering and Knowledge Management (EKAW02) (Siguenza, Spain)*, volume 2473, p. 197–212. 39, 40

Klein M., Kiryakov A., Ognyanov D. & Fensel D. (2002b). Finding and characterizing changes in ontologies. In *Proceedings of the 21st International Conference on Conceptual Modeling*, ER '02, p. 79–89, London, UK, UK : Springer-Verlag.

Klein M. & Noy N. (2003). A component-based framework for ontology evolution. In *Workshop on Ontologies and Distributed Systems at IJCAI-03, Acapulco, Mexico*, volume 71.

Kolp M., Giorgini P. & Mylopoulos J. (2003). Organizational patterns for early requirements analysis. In *In Proceedings of the 15th International Conference on Advanced Information Systems Engineering*. 42

Korf R., Kiss E., Durand F., Dousek M., El Ghali A., A., Viguie S., Berrueta D. & Lévy F. (2010). D2.4 : Extended demonstration prototypes for dissemination, teaching and public experimentation purposes. *ONTORULE Delivrable, http ://ontorule-project.eu/deliverables.*

Leenheer P. D. & Mens T. (2008). Ontology management. In *ONTOLOGY EVOLUTION State of the Art and Future Directions*, p. 131–176 : Springer.

Legendre V., Petitjean G. & Lepatre T. (2010). Gestion des règles « métier ». *Génie logiciel*, **92**, 43–52. vii, 51, 54

Leone N., Gottlob G., Rosati R., Eiter T., Faber W., Fink M., Greco G., Ianni G., Kalka E., Lembo D., Lenzerini M., Lio V., Nowicki B., Ruzzi M., Staniszkis W. & Terracina. G. (2006). The infomix system for advanced integration of incomplete and inconsistent data. *Reasoning Web 2006*, p. 93–127.

Lindgren R., Hardless C., Pessi K. & Nuldén U. (2000). The evolution of knowledge management system need to be managed. *Journal of Knowledge Management Practice*, **3**.

Linehan M. H. (2007). Ontologies and rules in business models. *Enterprise Distributed Object Computing Conference Workshops, IEEE International*, p. 149–156.

Linehan M. H. (2008). Sbvr use cases. *Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML 2008, Orlando, FL, USA*, **5321**, 182–196. vii, 24, 25

Luong P.-H. (2007). *Gestion de l'évolution d'un web sémantique d'entreprise*. PhD thesis, Université De Nice Sophia-Antipolos - Ecole Nationale supérieure des Mines PARIS. 33

Manola F. & Miller E. (2004). Rdf primer w3c recommendation 10 february 2004. http ://www.w3.org/TR/rdf-primer/. Accessed on October, 2012. 26

Motik B. & Rosati R. (2007). A faithful integration of description logics with logic programming. *IJCAI*.

Motik B., Sattler U. & R.Studer (2005). Query answering for owl-dl with rules. *Journal of Web Semantics : Science, Services and Agents on the World Wide Web*, p. 41–60. 49

Nauer E., Richard A., Derriere S., Genova F., Napoli A. & Toussaint Y. (2006). Construction d'une ontologie de descripteurs ucd en astronomie. *Actes d'IC 2006*, p. 21–30.

Neches R., Fikes R., Finin T., Gruber T., Patil R., Senator T. & Wartout W. (1991). Enabling technology for knowledge sharing. *AI Magazine*, p. 36–56. 15

Noy N. & Klein M. (2004). Ontology evolution : Not the same as schema evolution. In S.-V. L. Ltd, Ed., *Knowledge and Information Systems*, volume 6, p. 428–440. xi, 34, 36, 39

O'Brien P. & Abidi S. (2006). Modeling intelligent ontology evolution using biological evolutionary processes. In *Proceedings of the IEEE International Conference on Engineering of Intelligent Systems ICEIS 06. Islamabad, Pakistan*. 33

Paolo B., Fausto G., Frank V., Luciano S. & Heiner S. (2003). C-owl :contextualizing ontologies. *International Semantic Web Conference 2003*, p. 164–179. 15

Parsia B., Sirin E., Grau B. C., E.Ruckhaus & Hewlett D. (2005). Cautiously approaching swrl. *Elsevier Science*. 49

Plessers P. & De Troyer O. (2006). Resolving inconsistencies in evolving ontologies. In *The Semantic Web : Research and Applications, Proceedings of the 3rd European Semantic Web Conference ESWC 2006*, volume 4011, p. 200–214 : Springer. 40

Plessers P., De Troyer O. & Casteleyn S. (2007). Understanding ontology evolution : A change detection approach. In *Journal of Web Semantics : Science, Services and Agents on the World Wide Web*, volume 5, p. 39–49 : Elsevier Publication. 39

Plessers P. & Troyer O. D. (2005). Ontology change detection using a version log. In *In Proceeding of the 4th International Semantic Web Conference*, p. 578–592 : Springer. 39

Presutti V. & Gangemi A. (2008). Content ontology design patterns as practical building blocks for web ontologies. In H. Springer-Verlag Berlin, Ed., *Proceedings of the 27th International Conference on Conceptual Modeling (ER 2008)*, p. 128 – 141. 43, 44

Presutti V., Gangemi A., David S., de Cea G. A., Suárez-Figueroa M., Montiel-Ponsoda E. & Poveda M. (2008). D2.5.1 : A library of ontology design patterns : reusable solutions for collaborative design of networked ontologies. Available at : http ://www.neon-project.org/ (Downloaded 2008-05-23).

Puhrer J., El Ghali A., Chniti A., Korf R., Schwichtenberg A., Lévy F., Heymans S., Xiao G. & Eiter T. (2010). D2.3 consistency maintenance. intermediate report. *ONTORULE Delivrable, http ://ontorule-project.eu/deliverables.*

Roddick J. F. (1995). A survey of schema versioning issues for database systems. *Information and Software Technology*, **37**, 383–393. 33

Rosati R. (2005). Semantic and computational advantages of the safe integration of ontologies and rules. *Reasoning Web, Second International Summer School 2006, Lissabon,*, p. 128–151. 46

Rosati R. (2006). Dl+log : Tight integration of description logics and disjunctive datalog. *KR*.

Ross R. (2003). *Principles of the Business Rule Approach*. Addison-Wesley, Boston, USA. 14

Scharffe F. (2009). *Correspondence Patterns Representation*. PhD thesis, Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck. http ://www.scharffe.fr/pub/phdthesis/manuscript.pdf. 43

Schreiber G., Akkermans H., Anjewierden A., de Hoog R., Shadbolt N., de Velde W. V. & Wielinga B. (2000). *Knowledge Engineering and Management : The CommonKADS Methodology*. A Bradford Book, The MIT Press, Cambridge, Massachusetts London, England. 63

Sowa J. (2000). *Knowledge Engineering and Management : The CommonKADS Methodology*. A Bradford Book, The MIT Press, Cambridge, Massachusetts London, England. 12

Staab S., Erdmann M. & Maedche A. (2001). Engineering ontologies using semantic patterns. In e. In A. Preece, Ed., *Proceedings of the IJCAI-01 Workshop on E-business & The Intelligent Web*. 42

Stirna J. & Persson A. (2002). *Report of Dissemination Activities, deliverable D10, IST Programme project HyperKnowledge – Hypermedia and Pattern Based Knowledge Management for Smart Organisations*. Rapport interne, Department of Computer and Systems Sciences, Royal Institute of Technology, Stockholm, Sweden. project no. IST-2000-28401. 42

Stojanovic L. (2004). *Methods and Tools for Ontology Evolution*. PhD thesis, University of Karlsruhe. xi, 33, 35, 37, 40

Stojanovic L., Maedche A., Motik B. & Stojanovic N. (2002a). User-driven ontology evolution management. In G. S.-V. Berlin, Ed., *Knowledge Engineering and Knowledge Management : Ontologies and the Se-*

mantic Web, Proceedings of the 13th International Conference EKAW2002, volume 2473, p. 285–300. 34

Stojanovic L., Maedche M., Stojanovic N. & Studer R. (2003a). Ontology evolution as reconfiguration-design problem solving. In U. A. New York, Ed., *Proceedings of the 2nd international conference on Knowledge capture K-CAP'03*, p. 162–171. 40

Stojanovic L. & Motik B. (2002). Ontology evolution within ontology editors. In *Proceedings of the OntoWeb-SIG3 Workshop Evaluation of Ontology-based Tools (EON2002) at the 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2002)*, volume 62, p. 53–62. 34

Stojanovic L., Stojanovic N., Gonzalez J. & Studer. R. (2003b). Ontomanager – a system for the usage-based ontology management. In G. S. LNCS, Berlin, Ed., *On The Move to Meaningful Internet Systems 2003 : CoopIS, DOA, and ODBASE*, volume 2888, p. 858–875. 39

Stojanovic N., Stojanovic L. & Volz R. (2002b). A reverse engineering approach for migrating dataintensive web sites to the semantic web. In G. S.-V. Berlin, Ed., *Proceedings of the Conference on Intelligent Information Processing (IIP-2002), Part of the IFIP World Computer Congress WCC2002*, p. 141–154. 39

Wikipedia (2012a). Calcul des propositions. http ://fr.wikipedia.org/wiki/Calcul_des_propositions. Accessed on November, 2012. 19

Wikipedia (2012b). Ontologie (philosophie). http ://fr.wikipedia.org/wiki/Ontologie (philosophie). Accessed on October, 2012. 15

Liste des publications

Publications dans des revues internationales

Chniti A., Boussadi A., Degoulet P., Albert P. & Charlet J. (2012c). Pharmaceutical validation of medication orders using an owl ontology and business rules. In S. H. T. I. Quality of Life through Quality of Information, Ed., *In J. Mantas and C. Mazzoleni.*, volume 180, p. 1224–6.

Publications dans des congrès internationaux

Chniti A., Albert P. & Charlet J. (2012a). A loose coupling approach for combining owl ontologies and business rules. In C. W. proceedings, Ed., *RuleML2012@ECAI Challenge, at the 6th International Symposium on Rules Research Based and Industry Focused 2012*, volume 874, p. 103–110.

El Ghali A., Chniti A. & Citeau H. (2012). Bringing owl ontologies to the business rules users. In L. N. in Computer Science, Ed., *Rules on the Web : Research and Applications*, volume 7438/2012 of *RuleML 2012 : The 6th International Symposium on Rules : Research Based and Industry Focused*, p. 62–76. 138

Chniti A., Dehors S., Albert P. & Charlet J. (2010). Authoring business rules grounded in owl ontologies. In M. D. et al. (Eds.), Ed., *RuleML 2010 : The 4th International Web Rule Symposium : Research Based and Industry Focused* : LNCS 6403, Springer-Verlag Berlin Heidelberg 2010.

Publications dans des ateliers internationaux

Chniti A., Albert P. & Charlet J. (2012b). Mdrontology : An ontology for managing ontology changes impacts on business rules. In C. W. proceedings, Ed., *Joint Workshop on Knowledge Evolution and Ontology*

Dynamics 2012. In conjunction with International Semantic Web Conference (ISWC 2012).

Chniti A., Boussadi A., Degoulet P., Albert P. & Charlet J. (2012d). Pharmaceutical validation of medication orders using an owl ontology and business rules. In C. W. proceedings, Ed., *Joint Workshop on Semantic Technologies Applied to Biomedical Informatics and Individualized Medicine. In conjunction with International Semantic Web Conference (ISWC 2012).*

Publications dans des congrès nationaux

Chniti A., Albert P. & Charlet J. (2011). Gestion de la cohérence des règles métier éditées à partir d'ontologies owl. In P. de l'Université des Antilles et de la Guyane, Ed., *IC'2011 : Sciences exactes et naturelles.*

Publications dans des ateliers nationaux

A.Chniti, Boussadi A., Degoulet P., Albert P. & Charlet J. (2012). Validation pharmaceutique des prescriptions médicamenteuses à base d'une ontologie owl et de règles métier. In *IC pour l'Interopérabilté Sémantique dans les applications en e-Santé, Atelier associé à la conférence IC 2012.*

Livrables de projet

Fink M., Ghali A. E., Chniti A., Korf R., Schwichtenberg A., Lévy F., Pührer J. & Eiter T. (2011). D2.6 consistency maintenance. final report. *ONTORULE Delivrable, http ://ontorule-project.eu/deliverables.* 73

Korf R., Kiss E., Durand F., Dousek M., El Ghali A., Chniti A., Viguie S., Berrueta D. & Lévy F. (2010). D2.4 : Extended demons-

tration prototypes for dissemination, teaching and public experimentation purposes. *ONTORULE Delivrable, http ://ontorule-project.eu/deliverables.*

Puhrer J., El Ghali A., Chniti A., Korf R., Schwichtenberg A., Lévy F., Heymans S., Xiao G. & Eiter T. (2010). D2.3 consistency maintenance. intermediate report. *ONTORULE Delivrable, http ://ontorule-project.eu/deliverables.*

Comités d'organisation

- Membre du comité d'organisation de la conférence IC'2012, http://ic2012.crc.jussieu.fr/ ;
- Membre du comité d'organisation du *IBM Discovery event*@ ECAI 2012, http://www-01.ibm.com/software/fr/ECAI2012/.

Comité de programme

- Membre du comité de programme de OSEMA 2012 : 2nd International Workshop on Ontology and Semantic web for Manufacturing.

Annexe A

Implémentation des approches proposées

Comme nous l'avons décrit dans les chapitres précédents, nous avons implémenté les approches proposées sous forme de plug-ins pour le système de gestion des règles métier IBM ODM. Ainsi, nous avons implémenté :

- le *Plug-in OWL* qui permet d'importer des ontologies OWL dans ODM afin dédite et d'exécuter des règles métier à partir d'ontologies OWL ;
- le *Plug-in* $\mathcal{MDR}$ qui permet de gérer l'impact des évolutions des ontologies sur les règles.

Dans ce qui suit, nous commencerons par présenter les différentes étapes d'utilisation des plug-ins implémentés.

A.1 Utilisation du plug-in OWL

1. Après avoir lancer ODM [1], commencer par créer un projet de règle dans le *Rule Explorer* (voir Figure A.1)
2. Pour importer une ontologie OWL dans ODM, il suffit de faire un clique droit sur le projet de règle et de sélectionner *import*→ *OWL Plugin* → *Import OWL File* (voir Figure A.2).
3. Après avoir choisi le fichier OWL à importer le BOM sera automatiquement généré (voir Figure A.3) et toutes les fonctionnalités du système, à savoir l'édition et l"exécution des règles, peuvent être utilisées (voir Tutorial ODM pour plus d'information sur ses fonctionnalités [2]).

1. pour cela il suffit de lancer Eclipse et de sélectionner la perspective Rule.
2. http ://www-01.ibm.com/software/decision-management/operational-decision-management/websphere-operational-decision-management/

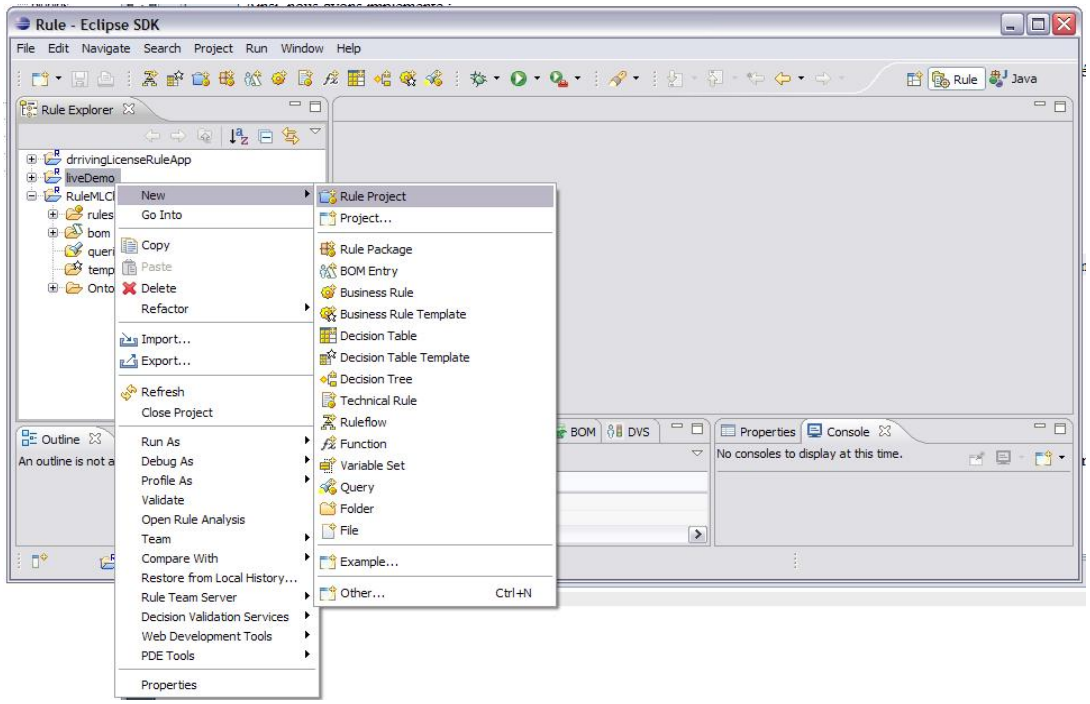

FIGURE A.1 – Création d'un nouveau projet de règles.

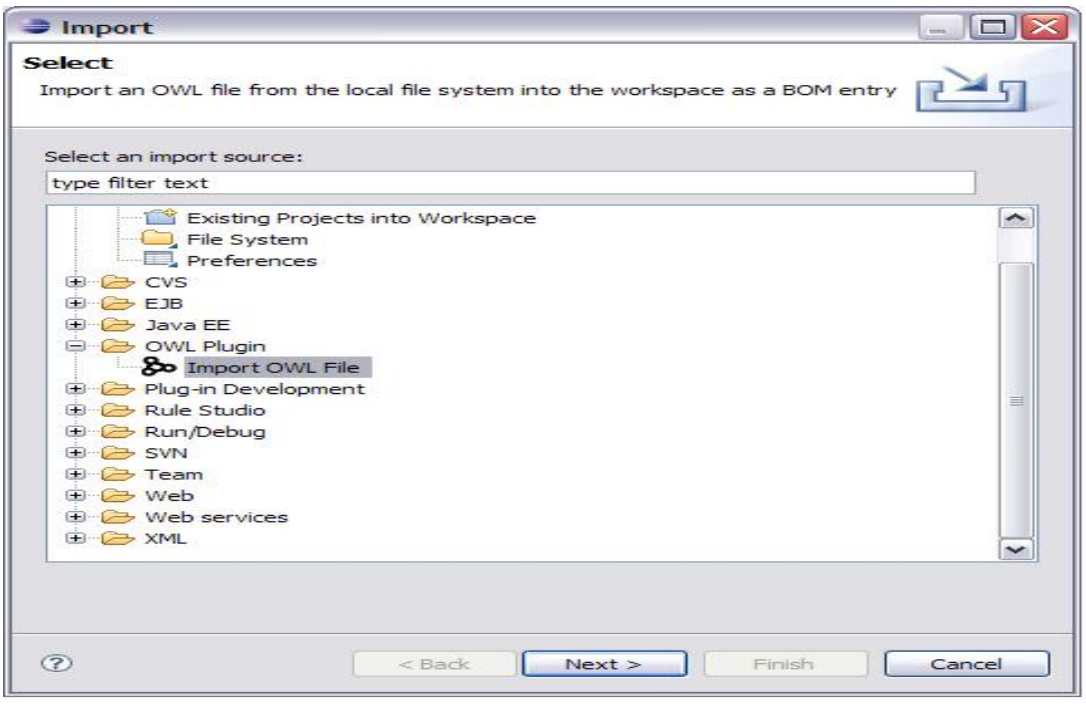

FIGURE A.2 – Importer une ontologie OWL.

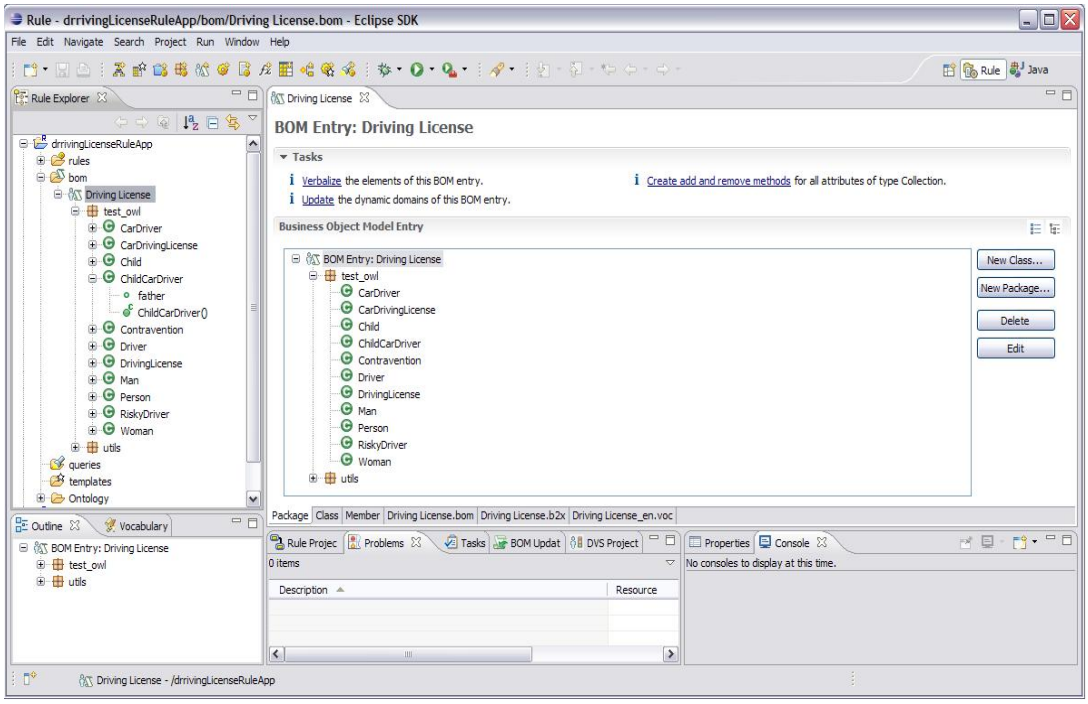

FIGURE A.3 – BOM généré à partie d'une ontologie OWL.

A.2 Utilisation du plug-in $\mathcal{MDR}$

1. Pour gérer l'impact de l'évolution d'une ontologie sur les règles, il faut commencer par lancer le *Plug-in* $\mathcal{MDR}$. Pour cela, il suffit de faire un clique droit sur le projet de règles concerné et de sélectionner *Ontology Change Management* → *Rule Consistency Checking* (voir Figure A.4).
2. Une fois le plug-in lancé, la liste des changements modélisés dans l'ontologie $\mathcal{MDR}$ est affichée. Quand l'utilisateur sélectionne le changement à appliquer, le module de détection des règles impactées est déclenché et une description contenant des informations sur le changement ainsi que le nom des règles impactées est affichée (voir Figure A.5).
3. Quand l'utilisateur clique sur le bouton *Next*, les *règles de détection des incohérences* sont déclenchées et le *Rule Consistency Checking Wizard* est affiché contenant la liste des incohérences détectées. Quand l'utilisateur clique sur une incohérence, des informations concernant cette incohérence sont affichées (voir Figure A.6).
4. Quand l'utilisateur clique sur le bouton *Next*, les *règles de réparation* sont

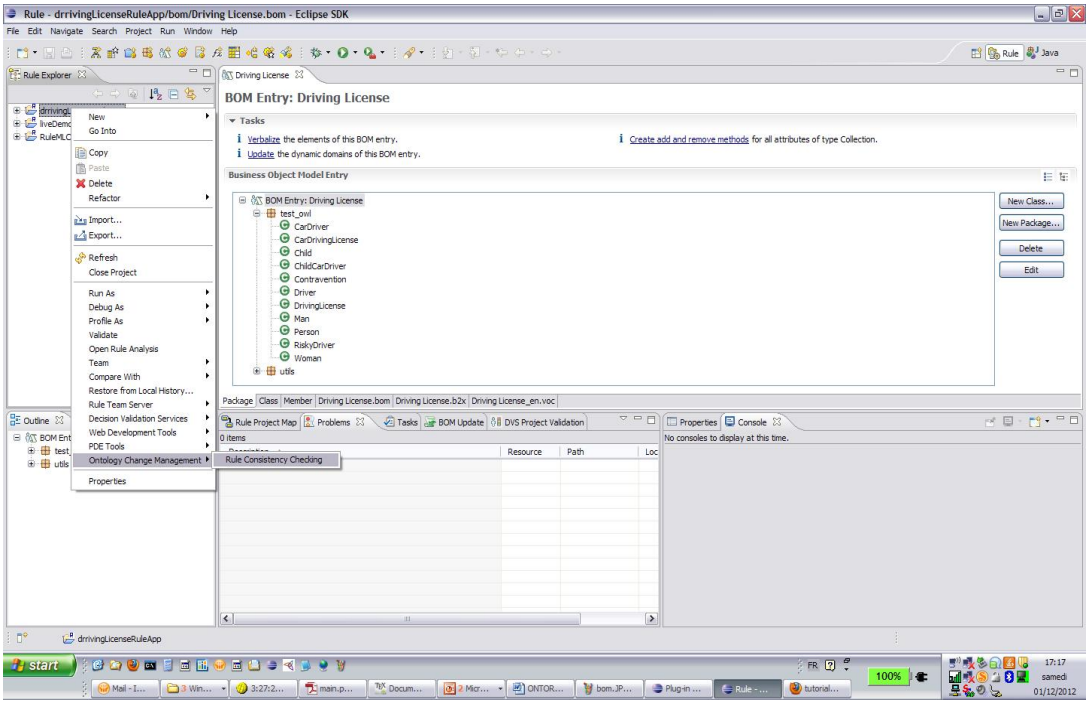

FIGURE A.4 – Lancement du plug-in $\mathcal{MDR}$.

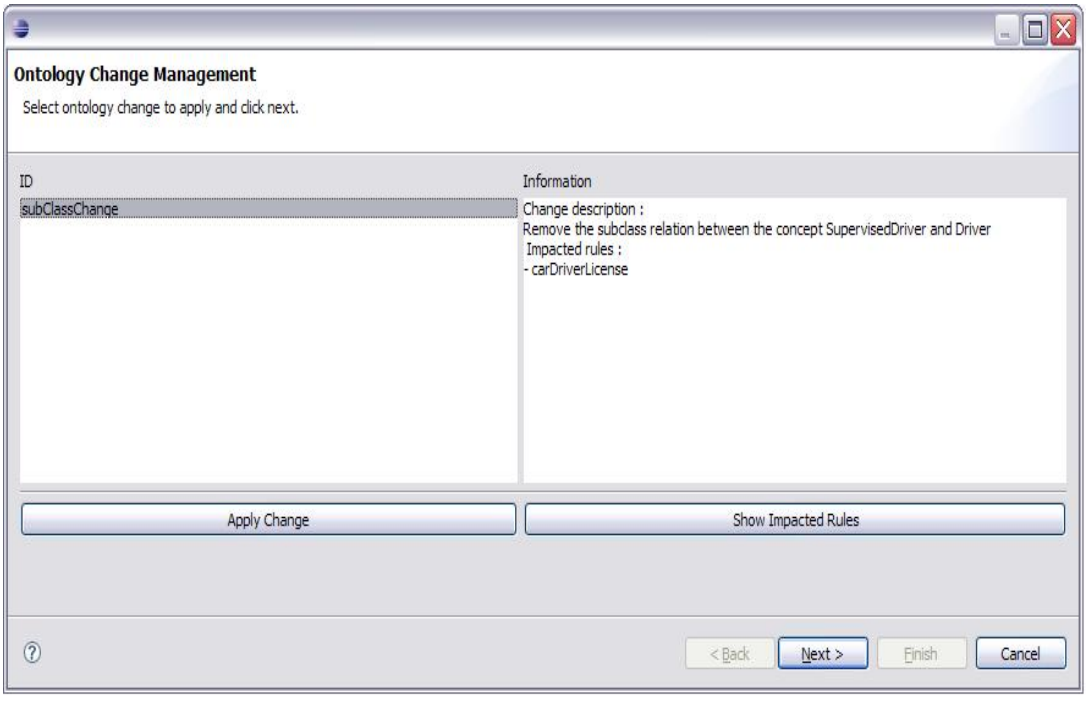

FIGURE A.5 – Liste des changements.

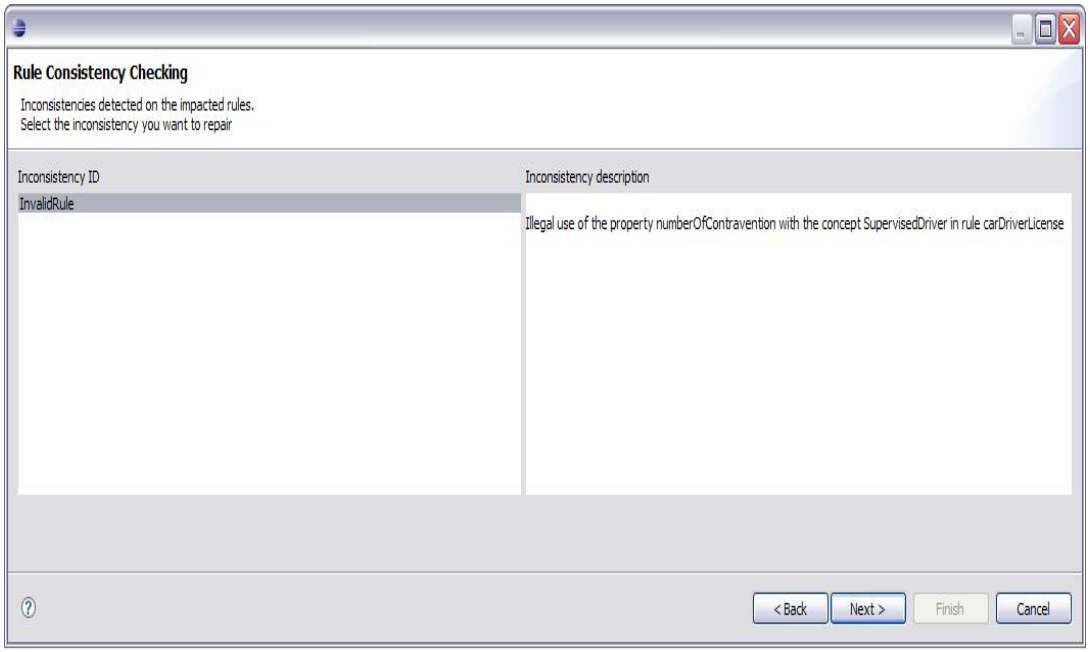

FIGURE A.6 – Détection des réparations.

déclenchées et le *Inconsistency repair wizard* est affiché contenant la liste des réparations choisies. Quand l'utilisateur sélectionne la réparation à effectuer, celle ci est appliquée automatiquement (voir Figure A.7).

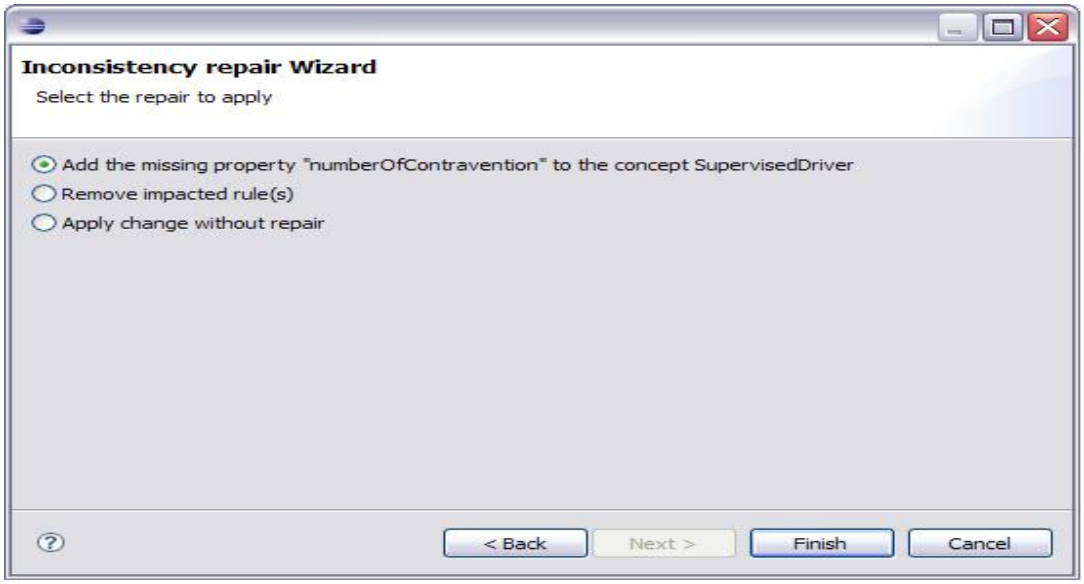

FIGURE A.7 – Proposition des réparation.

Annexe B

Le langage $\mathcal{MDR}$

Dans ce qui suit, nous présentons la spécification des patrons de gestion des changement ainsi que des contraintes de changement avec le langage $\mathcal{MDR}$

```
/*-----------------------------------
 Change patterns
-----------------------------------*/
/*
 * Generic change pattern without impacted rules
 */
 <mdr:change>
   <mdr:changeEntity mdr:type = concept mdr:localName = Name />
   <mdr:changeObject> "changeObject" </mdr:changeObject>
   <mdr:changeType> Remove </mdr:changeType>
   <mdr:changeConstraint> "Constraint to verify" </mdr:changeConstraint>
</mdr:change>

/*
 * Generic change pattern with impacted rule
 */
<mdr:change>
   <mdr:changeEntity mdr:type = property mdr:localName = Name />
  <mdr:changeObject> "changeObject" </mdr:changeObject>
  <mdr:changeType> Add </mdr:changeType>
  <mdr:impactedRule>
    <mdr:ruleName> ruleName </mdr:ruleName>
    <mdr:impactedRulePart> ActionPart </mdr:impactedRulePart>
  </mdr:impactedRule>
  <mdr:changeConstraint> "Constraint to verify" </mdr:changeConstraint>
```

```
</mdr:change>

/*
 * subclass change pattern
 * a subclass change modifies the subClass relation between 2 concepts
 * The rules that reference the sub concept with an inherited property
 * wil be declared as Invalid Rules.
 * Repairs :
 * - Add the inherited property as a property of the sub concept
 * - remove the test/action on this property on the impacted rules
 * - remove the impacted rules
 */
<mdr:subclassChange>
    <mdr:changeEntity mdr:type = concept mdr:localName = Name />
    <mdr:changeObject> subClassOf </mdr:changeObject>
    <mdr:changeType> Modify </mdr:changeType>
       <mdr:changeEntitySuperClass> Name </mdr:changeEntitySuperConcept>
    <mdr:impactedRule>
       <mdr:ruleName> ruleName </mdr:ruleName>
    </mdr:impactedRule>
    <mdr:changeConstraint> subClassOfChangeConstraint </mdr:changeConstraint>
    <mdr:inconsistency> Invalid Rule </mdr:inconsistency>
    <mdr:repair> "Add the inherited property as a property
                    of the change entity" </mdr:repair>
    <mdr:repair> "remove the test/action on this property
                    on the impacted rules" </mdr:repair>
    <mdr:repair> "remove the impacted rules" </mdr:repair>
</mdr:subclassChange>


/*
 * allValuesFrom change pattern
 * an allValuesFrom change modifies the restriction type of a property
 * The rules that reference the property will be declared as invalid rule
 * Repairs:
```

```
 * - Change the new concept restriction to a subclass
            from the old concept restriction subclasses
 * -remove the test/action on this property on the impacted rules
 * - remove the impacted rules
 */

 <mdr:allValuesFromChange>
    <mdr:changeEntity mdr:type = property mdr:localName = Name />
    <mdr:changeObject> allValuesFrom </mdr:changeObject>
    <mdr:changeType> Modify </mdr:changeType>
    <mdr:oldRestrictionValue
             mdr:typeresctriction = concept mdr:localName = oldValue />
    <mdr:newRestrictionValue
             mdr:typerestriction = concept mdr:localName = newValue/>
    <mdr:impactedRule>
        <mdr:ruleName> ruleName </mdr:ruleName>
    </mdr:impactedRule>
    <mdr:changeConstraint> allValuesFromChangeConstraint
                      </mdr:changeConstraint>
    <mdr:inconsistency> Invalid Rule </mdr:inconsistency>
    <mdr:repair>  "Change the type of the property" </mdr:repair>
    <mdr:repair> "remove the test/action on this property
                   on the impacted rules" </mdr:repair>
    <mdr:repair> "remove the impacted rules" </mdr:repair>
</mdr:allValuesFromChange>

/*
 * oneOf Change pattern which impact the rule in the condidition part
 * An oneOf change modifies the list of values that a property could have.
 * The rules that test on this property will not be apllied after
 the change application
 * Repairs :
 * - Change the test value
 * - remove the test on this property in the impacted rules
 * - remove the impacted rules
```

```
 */


<mdr:enumerationChange>
  <mdr:changeEntity mdr:type = property mdr:localName = Name />
  <mdr:changeObject> oneOf </mdr:changeObject>
  <mdr:changeType> Modify </mdr:changeType>
  <mdr:oldEnumerationValues> Value1 Value2 Value3 </mdr:oldEnumerationValues>
  <mdr:newEnumerationValues> enumID enumID </mdr:newEnumerationValues>
  <mdr:impactedRule>
     <mdr:ruleName> ruleName </mdr:ruleName>
     <mdr:impactedRulePart> ConditionPart </mdr:impactedRulePart>
  </mdr:impactedRule>
  <mdr:changeConstraint> enumerationChangeConstrain </mdr:changeConstraint>
  <mdr:inconsistency> Rule Never Apply </mdr:inconsistency>
  <mdr:repair> "Change the test value" </mdr:repair>
  <mdr:repair> "remove the test on this property
                                on the impacted rules" </mdr:repair>
  <mdr:repair> "remove the impacted rules" </mdr:repair>
</mdr:enumerationChange>

/*
 * oneOf Change pattern which impact the rule in the action part
 * An oneOf change modifies the list of values that a property could have.
 * The rules that assign to this property one of the modified value
            will cause a domain violation
 * Repairs :
 * - Change the assigned value
 * - remove the statment (the action) on this property in the impacted rules
 * - remove the impacted rules
 */

<mdr:enumerationChange>
  <mdr:changeEntity mdr:type = property mdr:localName = Name />
  <mdr:changeObject> oneOf </mdr:changeObject>
```

```
  <mdr:changeType> Modify </mdr:changeType>
   <mdr:oldEnumerationValues> Value1 Value2
                          Value3 </mdr:oldEnumerationValues>
  <mdr:newEnumerationValues> Value1 Value2 </mdr:newEnumerationValues>
  <mdr:impactedRule>
     <mdr:ruleName> ruleName </mdr:ruleName>
     <mdr:impactedRulePart> ActionPart </mdr:impactedRulePart>
  </mdr:impactedRule>
  <mdr:changeConstraint> "the rule shoul not assign
              the modified value to the property" </mdr:changeConstraint>
  <mdr:inconsistency> Domain Violation </mdr:inconsistency>
  <mdr:repair> "change the assigned value" </mdr:repair>
  <mdr:repair> "remove the test on this property
                             on the impacted rules" </mdr:repair>
  <mdr:repair> "remove the impacted rules" </mdr:repair>
 </mdr:enumerationChange>


/*-----------------------------------
 Change Constraints
-----------------------------------*/
/* a subClass change constraint
* the change entity (a concept) should not uses an inherited
property in a rule
*/

<mdr:constraint mdr:ID = "subClassOfChangeConstraint">
 <mdr:definition>
   <mdr:concept mdr:ID="aconcept"/>
   <mdr:concept mdr:ID="asuperconcept"/>
   <mdr:rule mdr:ID="arule"/>
   <mdr:property mdr:ID="aproperty"/>
 </mdr:definition>
 <mdr:description>
    <mdr:resource="aconcept" rdfs:subClassOf mdr:resource="asuperconcept"/>
```

```
    <mdr:resource="asuperconcept" rdfs:domain mdr:resource="aproperty"/>
    <mdr:resource="aconcept" mdr:usedResource="aproperty"
              mdr:resource="arule"/> // aconcept uses aproperty in arule
 </mdr:description>
</mdr:constraint>

/* An allValuesFrom change constraint
*  the property type must be a subtype of the new restriction value
*/
<mdr:constraint mdr:ID = "allValuesFromChangeConstraint">
 <mdr:definition>
   <mdr:concept mdr:ID="oldrestriction"/>
   <mdr:property mdr:ID="aproperty"/>
   <mdr:rule mdr:ID="arule"/>
   <mdr:concept mdr:ID="newrestriction">
 </mdr:definition>
 <mdr:description>
    <mdr:resource="aproperty" owl:allValuesFrom
                                    mdr:resource="oldrestriction"/>
    <mdr:resource="arule" mdr:usedResource="aproperty"
                                   mdr:resource="oldrestriction"/>
    <mdr:resource="newrestriction" rdfs:subClassOf
        mdr:resource="oldrestriction"/> // aconcept uses aproperty in arule
 </mdr:description>
</mdr:constraint>

/* An oneOf change constraint
*  the tested/assigned property value must be in the new enumeration
*/
<mdr:constraint mdr:ID = "enumerationChangeConstraint">
 <mdr:definition>
   <mdr:enumeration mdr:ID="oldEnumeration">
   <mdr:enumeration mdr:ID="newEnumeration">
   <mdr:property mdr:ID="aproperty"/>
   <mdr:ruleValue mdr:ID="arulevalue">
```

```
   <mdr:rule mdr:ID="arule"/>
 </mdr:definition>
 <mdr:description>
    <mdr:resource="arule" mdr:usedResource="aproperty"
                                                  mdr:resource="arulevalue"/>
    <mdr:resource="arulevalue" mdr:valueFrom mdr:resource="newEnumeration">
 </mdr:description>
</mdr:constraint>
```

Résumé : Vu la rapidité de l'évolution des connaissances des domaines, la maintenance des systèmes d'information est devenue de plus en plus difficile à gérer. Afin d'assurer une flexibilité de ces systèmes, nous proposons une approche qui permet de représenter les connaissances des domaines dans des modèles de représentation des connaissances plutôt que de les coder, dans un langage de programmation informatique, dans l'application du domaine. Ceci assurerait une meilleure flexibilité des systèmes d'information, faciliterait leur maintenance et permettrait aux experts métier de gérer eux même l'évolution des connaissances de leur domaine.

Pour cela, nous proposons une approche qui permet d'intégrer des ontologies et des règles métier. Les ontologies permettent de modéliser les connaissances d'un domaine. Les règles permettent aux experts métier de définir et d'automatiser, dans un langage naturel contrôlé, des décisions du métier en se fondant sur les connaissances représentées dans l'ontologie. Ainsi, les règles dépendent des entités modélisées dans l'ontologie. Vu cette dépendance, il est nécessaire d'étudier l'impact de l'évolution des ontologies sur les règles. Pour cela, nous proposons l'approche $\mathcal{MDR}$ ($\mathcal{M}$odéliser - $\mathcal{D}$étecter - $\mathcal{R}$éparer) qui permet de modéliser des changements d'ontologies, de détecter les problèmes de cohérence qu'ils peuvent causer sur les règles métier et de proposer des solutions pour réparer ces problèmes.

L'approche proposée est une approche orientée experts métier et est fondée sur les systèmes de gestion des règles métier.

Mots clès : Ontologie, Règle métier, Evolution d'ontologie, Système de gestion de règles métier