N° d'ordre : 3507

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **MOUHAMED DIOUF**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

SPÉCIFICATION ET MISE EN ŒUVRE D'UN FORMALISME DE RÈGLES MÉTIER

Soutenue le : 10 Décembre 2007

Après avis des rapporteurs :

Nicole Bidoit	Professeur
Thérèse Rougé - Libourel	Professeur

Devant la commission d'examen composée de :

Richard Castanet	Professeur	Directeur de Thèse
Sofian Maabout	Maître de Conférences	Encadrant
Nicole Bidoit	Professeur	Rapporteur
Thérèse Rougé - Libourel	Professeur	Rapporteur
Mohamed Mosbah	Professeur	Examinateur
Jean-Paul Pérez	Génigraph	Examinateur

– 2007 –

Remerciements

À Sofian Maabout

À Richard Castanet

À Kaninda Musumbu

À Olivier Nicolas, Jean-Paul Pérez et toute l'équipe de Génigraph Toulouse

Aux membres du jury

À mes frères et sœur, à mes amis

À Ta Mado, tonton Badou ainsi qu'à mes frères et sœurs du 58 rue de Lescure

À Madame et monsieur Ndiaye

À Madame et monsieur Diop

À ma princesse ...

À Tanoor et Assane

À Robert mon ami du banc (que la terre te soit légère)

À ma grand-mère Yaye Seikh (que la terre te soit légère)

À mère et père pour votre amour et votre soutien
je vous dois beaucoup

Table des matières

Table des figures

Glossaire

API = Application Programming Interface
ARM = Approche par règle métier
AS = Action Semantic
BOM = Business Object Model
BPMN = Business Process Modeling Notation
BRMS = Business Rule Management System
BRML = Business Rule Markup Language
CIM = Computation Independent Model
CLP = Courteous Logic Programs
CWM = Closed World Machine
CWM = Common Warehouse Metamodel
DRL = Drool Rule Language
ERML = e-Citiz Rule Markup Language
IDM = Ingénierie Dirigée par les Modèles
IRL = Ilog Rule Language
JAXB = Java XML Binding
JESS = Java Expert System Shell
JVM = Java Virtual Machine
MDA = Model Driven Architecture
MDD = Model Driven Development
MDE = Model Driven Engineering
MOF = Meta Objet Facility
OCL = Object Constraint Language
ODM = Ontology Definition Metamodel
OMG = Object Management Group
OLP = Ordinary Logic Programs
OWL = Web Ontology Language
PIM = Platform Independent Model
PRR = Production Rule Representation
PSM = Platform Specific Model
UML = Unified Modeling Language
URI = Uniforme Resource Identifier
RDF = Resource Description Framework

RIF = Rule Interchange Format
SBVR = Semantics of Business Vocabulary and Business Rules
SGBD = Système de Gestion de Base de Données
SGRM = Système de Gestion de Règle Métier
W3C = World Wide Web Consortium
WM = Working Memory
XML = Extensible Markup Language
XSLT = eXtensible Stylesheet Language Transformation

Chapitre 1

Introduction et contexte

1.1 Introduction

Les systèmes informatique jouent un rôle crucial dans notre vie de tous les jours en automatisant des processus qui gouvernent et régissent nos interactions entre individus ou autres entités. Ces systèmes deviennent de plus en plus complexes et leur mise en œuvre requiert des expertises d'horizons différents. Les techniques permettant d'établir ces systèmes n'ont cessé d'être améliorées afin de les avoir plus rapidement, plus souples, plus évolutifs, etc, de manière plus itérative.
Dès le début de la mise en place d'un système informatique, les experts système ou technique, qui réalisent le développement, sont au centre du cycle de développement. Ils se basent sur un ensemble fonctionnel, qui constitue le cahier des charges, qui est spécifié par l'expert fonctionnel. En effet, ce sont les experts métier ou fonctionnel qui, face à des besoins, les expriment puis en demandent la réalisation aux experts système. Les experts métier ne sont pas, en général, des informaticiens, et étant à l'origine du système informatique, le connaissent mieux que quiconque.
Il existe plusieurs cycles de développement de logiciel dont : les cycles en V, M, W, spirale, chute d'eau, etc. Tous ces cycles de développement ont le même enchaînement, à des degrés de granularité différents. Il consistent à faire une analyse des fonctions demandées, la conception (réalisation et validation de modèles, choix des structures de données, etc.) implémentation, tests, validation par les experts métier et enfin déploiement. Comme nous pouvons le voir, dans ces enchaînements, l'expert métier n'intervient qu'en début (pour la mise en évidence du fonctionnel sous forme d'un cahier des charges), et en fin de cycle (pour la validation de l'application). Une fois que le système est déployé, si la politique de l'entreprise change, c'est-à-dire l'ensemble fonctionnel ayant servi à réaliser le système, dans ce cas, il faut faire intervenir les experts système qui, ré-étudient les besoins fonctionnels, re-conceptualisent, ré-implémentent, re-testent, re-valident et re-deploient. Dans une telle approche, nous pouvons voir que l'expert métier, qui est au centre du métier, ne joue pas pleinement son rôle, ce qui rend la réalisation et la maintenance complexes, influençant

ainsi fortement le cycle de vie de l'application.
L'approche par règles métier (que l'on abrégera ARM) ou the *business rule approach* vient apporter une solution pour des applications orientées métier et par le métier. Le principe de l'ARM est la séparation nette entre la logique métier et la logique système, permettant ainsi aux experts métier de réaliser les exigences du système, en terme métier, dans un environnement zéro développement se basant sur le langage naturel.
L'ARM est complémentaire à l'ingénierie dirigée par les modèles ou Model Driven Architecture (MDA) qui a pour objectif de permettre aux experts technique de se soustraire des détails techniques de l'implémentation en se focalisant dans un premier temps sur un niveau abstrait de modèlisation. Ceci indépendamment de la plateforme, pour ensuite automatiser totalement ou partiellement la génération des détails techniques (code ou transformation de modèles).
D'un autre côté, les opérations sur les modèles se basent uniquement sur la syntaxe, car MDA ne s'intéresse pas au contexte et donc ne dit absolument rien sur la sémantique. La génération de code et, de manière générale, les opérations sur les modèles (validation, simulation, transformation, etc.) permettent un gain considérable de temps en ne se basant que sur la syntaxe. Alors qu'en serait-il si l'on prenait en compte la sémantique dans de telles opérations ? Nous avons le domaine du Web Sémantique qui s'intéresse à la sémantique. Le Web Sémantique n'est pas un autre Web, c'est une extension du Web actuel afin de donner un sens aux ressources et aux liens entre elles. Le Web Sémantique n'est pas une technologie mais plus un concept que l'on tente de mettre en place et qui est de rendre le Web exploitable aussi bien par les humains que par des agents artificiels intelligents (machines).
Dans le cadre de cette thèse, nous nous sommes intéressés à la génération de règles métier suivant l'approche par ingénierie dirigée par des modèles, en partant de la couche d'exigences à la couche de modèles concrets en passant par la couche de modèles abstraits.
L'un des besoins les plus pressants dans le domaine de l'ARM est la mise en place d'un standard de formalisme pour représenter les règles métier. Des travaux dans ce sens sont en cours mais rien de concret n'existe encore. Ainsi notre première problématique était de mettre en place un formalisme indépendant de toute plateforme qui permettrait également de le traduire vers d'autres formalismes plus spécifiques. Pour cela, en plus du formalisme de règles métier, nous avons mis en place un moteur de transformations de ce dernier vers d'autres formalismes.
Notre deuxième problématique était la génération de règles dans un formalisme exécutable depuis les exigences. Nous sommes partis du constat suivant : depuis quelques années, les entreprises ont la bonne pratique de modéliser leurs concepts en utilisant des langages de modélisation comme UML ; alors, comment depuis ces modèles statiques, pouvons-nous générer des règles métier ? Nous allons présenter dans ce document nos réponses à cette question.

1.1.1 Organisation du document

Ce document est composé de 5 parties.
Dans la première partie, au chapitre 2, nous présentons le principe de l'approche par règles métier ainsi que les techniques permettant de la mettre en œuvre. Nous y parlons de problèmes que pose l'approche actuelle qui utilise les cycles de développement que l'on connaît. Nous y parlons également de l'algorithme de RETE qui permet d'améliorer l'exécution des règles dans le contexte d'un chaînage avant et nous présentons également un banc d'essais de moteurs de règles en utilisant des problèmes connus. Nous exposons également le concept de Système de Gestion de Règles Métier (SGRM), et nous finissons en donnant une classification des règles de manière générale.
Au chapitre 3 qui porte sur le formalisme de règles métier, nous parlons du fondement de langages de règles et de tentatives de standardisation. Nous présentons également notre formalisme de règles métier.
Dans la deuxième partie nous présentons l'ingénierie dirigée par les modèles ainsi que les outils permettant sa mise en œuvre.
Dans la troisième partie nous présentons au chapitre 5 le principe du Web Sémantique ainsi que les technologies utilisées pour le mettre en œuvre. Au chapitre 6 nous parlons de logique, d'ontologies et des techniques de raisonnement dans le Web Sémantique.
Dans la quatrième partie, nous présentons au chapitre 7 notre approche concernant la génération automatique de règles métier par enrichissement sémantique de modèles. Nous y montrons comment nous utilisons les domaines de l'ingénierie dirigée par les modèles et du Web Sémantique pour générer des règles métier.
Dans la cinquième et dernière partie nous parlons de l'implémentation de nos différents travaux que nous avons effectués au sein de la plateforme de génération d'e-services *e-Citiz*. Nous parlons de l'API JSR94 au chapitre 10. Cette dernière partie est clôturée par le chapitre 11 contenant la conclusion et les perspectives de la thèse.
Avant tout, nous allons d'abord présenter le contexte industriel de la thèse.

Concernant la lisibilité de ce document, la partie IV dépend des 3 premières, le contexte *e-Citiz* est fondamental pour la compréhension de la partie V concernant l'implémentation, sinon toutes les autres parties sont indépendantes.

1.2 Contexte de la thèse

Cette thèse s'est effectuée dans le cadre d'une convention CIFRE. Génigraph[1], qui est une SSII et un éditeur de logiciels de 150 personnes, est l'entreprise d'accueil, dans son service de recherche et développement du produit *e-Citiz*.

[1] http ://www.genigraph.fr

1.2.1 La Plateforme d'e-services *e-Citiz*

Présentation générale

e-Citiz[2] est une plateforme de spécification, de génération et de gestion de télé-procédures, de télé-service et plus généralement d'e-services. La plateforme *e-Citiz* est composée principalement d'un serveur d'e-services pour leur exécution que l'on appelle moteur (engine), et d'un studio qui permet leur spécification. *e-Citiz* apporte une réponse aux changements fréquents de lois et de réglementation qui impliquent des modifications de la logique métier de la dématérialisation des procédures.
La Figure 1 montre comment les différents acteurs utilisent la plateforme pour réaliser un e-service. Les experts métier ou fonctionnel utilisent le studio pour spécifier les besoins fonctionnels du e-service, en collaboration avec les experts technique (développeurs) qui spécifient les besoins techniques et enfin le moteur *e-Citiz* est utilisé pour exécuter l'e-service

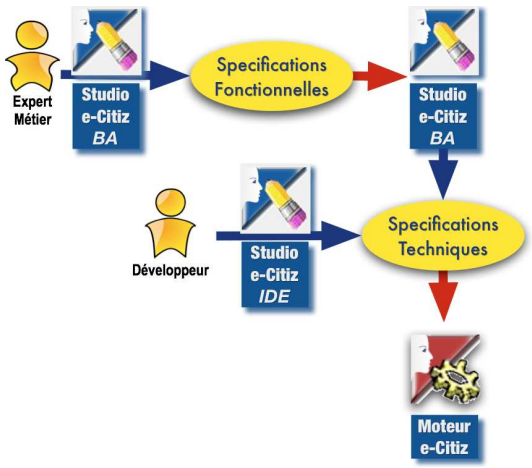

FIG. 1 – Utilisation d'*e-Citiz* pour la réalisation d'un e-service

Approche méthodologique

La démarche de développement proposée par *e-Citiz* est basée sur un processus itératif. La démarche itérative est la mieux adaptée à ce type de projet car elle permet de disposer du système correspondant le mieux à ses besoins en :

- facilitant la compréhension du cahier des charges ;
- affinant certains aspects des spécifications au fur et à mesure de l'avancement du projet ;

[2] http ://www.ecitiz.fr

- améliorant la qualité de l'applicatif par le biais du prototype complété à chaque itération ;
- permettant la validation de l'applicatif et son comportement au fur et à mesure de sa réalisation.

Le produit *e-Citiz* a été spécialement conçu pour faciliter les processus de développement itératif (voir la Figure 2). Il permet de concevoir et réaliser des e-services, en partant d'une maquette, puis en élaborant un prototype jusqu'à l'obtention des e-services mis en production. Le Studio *e-Citiz* permet notamment de :

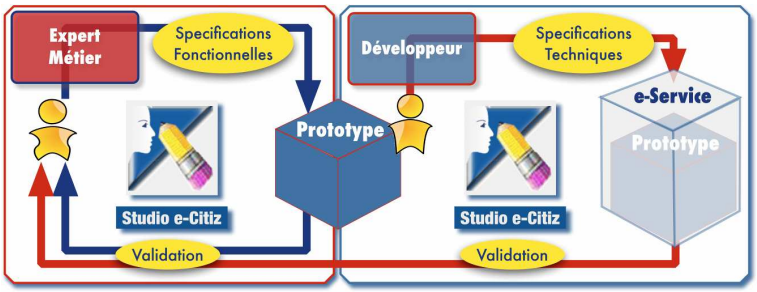

Fig. 2 – Approche itératif pour la réalisation d'un e-service

- définir l'ensemble des acteurs interagissant avec les différents e-services à mettre en œuvre ;
- spécifier, pour chaque acteur, l'ensemble des services auxquels il a le droit d'accéder ;
- définir l'intégralité du modèle de données comme, par exemple, le contenu des dossiers personnels des utilisateurs des e-services ;
- définir les organisations d'appartenance des acteurs quand il s'agit d'agent de traitement Back Office ;
- définir les utilisateurs autorisés à se connecter aux services et les rôles qui leurs sont attribués ;
- gérer les notifications ou alertes entre acteurs et/ou organisations.

Le studio utilise des formalismes graphiques standardisés tels qu'UML (diagrammes de cas d'utilisation ; diagrammes d'états) et BPMN.

Structure générale du studio

Le Studio *e-Citiz* est développé sous la forme d'une extension (plugin) de l'environnement de développement Eclipse[3]. La Figure 3 montre une vision globale du studio *e-Citiz* et les différentes vues qu'utilisent les experts métier pour créer les e-services.

[3] http ://www.eclipse.org

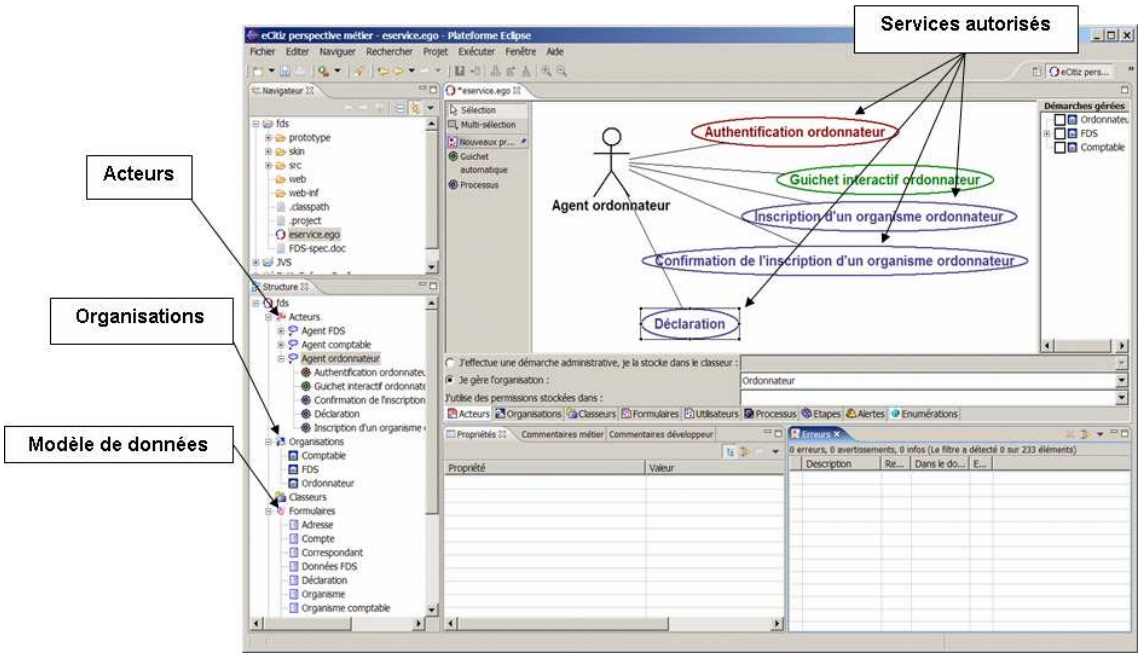

FIG. 3 – Vue générale du studio *e-Citiz*

Elaboration d'un service

Le Studio *e-Citiz* permet, pour chaque service défini, de préciser l'enchaînement des différentes étapes. Un e-service est un ensemble de processus métier et les étapes représentent les tâches qui constituent un processus métier. Il existe plusieurs types d'étapes qui sont enchaînées pour constituer un e-service (authentification de l'usager, affichage d'une page d'information, saisie d'un formulaire, traitement métier spécifique comme l'envoi de mail par exemple) (voir la Figure 4). Une étape a un point d'entrée et un ou plusieurs points de sorties. Les étapes définies pour un service peuvent être réutilisées par d'autres services. Le Studio permet ainsi de mutualiser les travaux effectués suivant le principe simple de réutilisation.

Elaboration du modèle de données

Le Studio *e-Citiz* permet également de définir l'ensemble du modèle de données qui sera ensuite exploité et partagé par les différents e-services du projet.
L'éditeur du modèle de données du Studio permet de spécifier :

- les objets du modèle métier (Compte, Déclaration, Données personnelles, ...) ;
- les attributs des objets du modèle ;
- les types des attributs et la cardinalité correspondante.

Les types des attributs peuvent être primitifs (texte, nombre entier, nombre décimal, valeur logique, ...) ou complexes en faisant référence à d'autres objets du modèle. Par exemple,

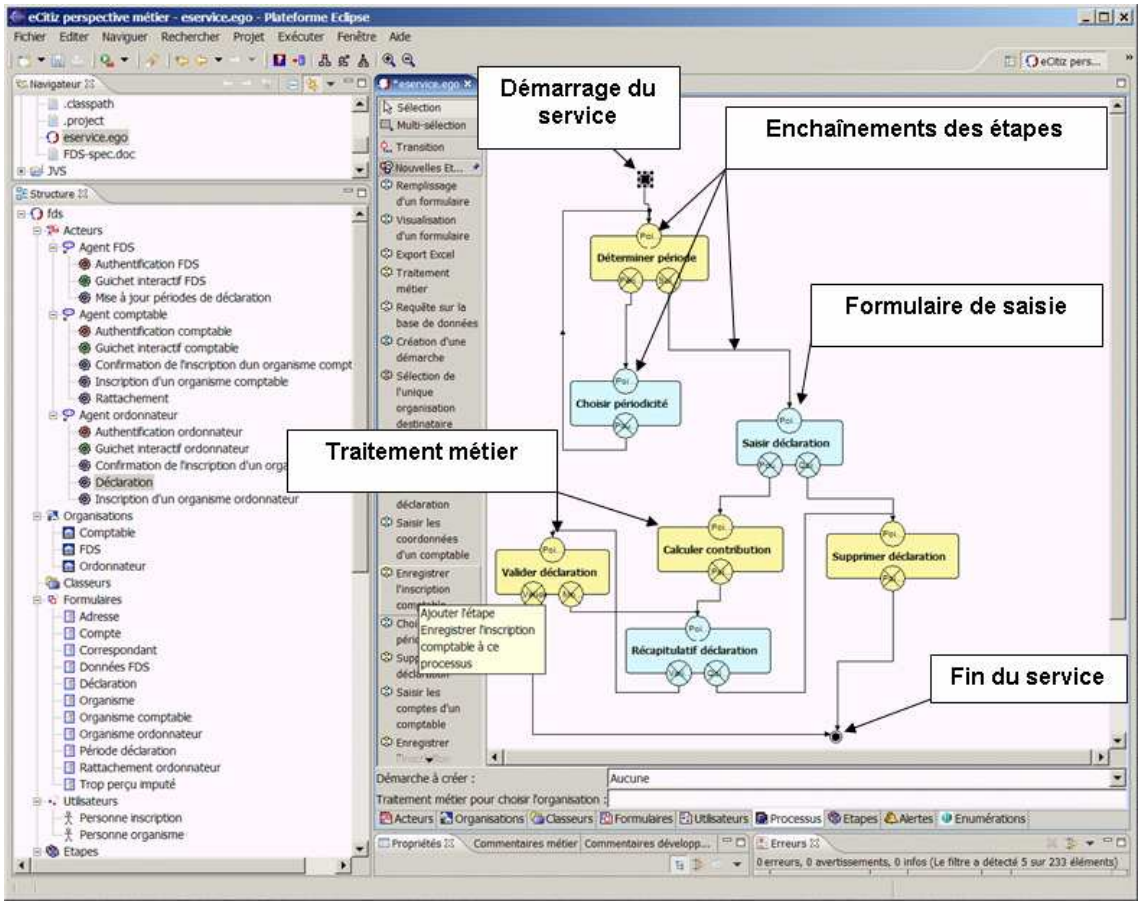

Fig. 4 – Spécification des étapes d'un e-service dans le Studio *e-Citiz*

l'objet de type *Déclaration* pourra être utilisé lors de la définition de l'objet *Ordonnateur* pour spécifier qu'un *Ordonnateur* possède un certain nombre de *Déclarations* (ce qui correspond à l'ensemble des déclarations produites). Ainsi il sera défini, pour un *Ordonnateur*, un attribut de type collection de *Déclarations* faisant référence à l'objet *Déclaration* défini par ailleurs. La Figure 5 montre un modèle de données dans le Studio *e-Citiz*.

Elaboration des formulaires de saisie

Pour élaborer les spécifications d'une étape de saisie de formulaire, qui est assimilable à une page affichée sur un navigateur Internet contenant un ensemble de champs renseignés ou à saisir, le Studio dispose d'un éditeur de page élaboré.
Pour chaque étape de saisie de formulaire, il est possible de préciser :

- les différents points de sortie de l'étape conditionnant le processus d'enchaînement des différentes étapes ;
- l'origine de la donnée à saisir ou à afficher dans le modèle de données.

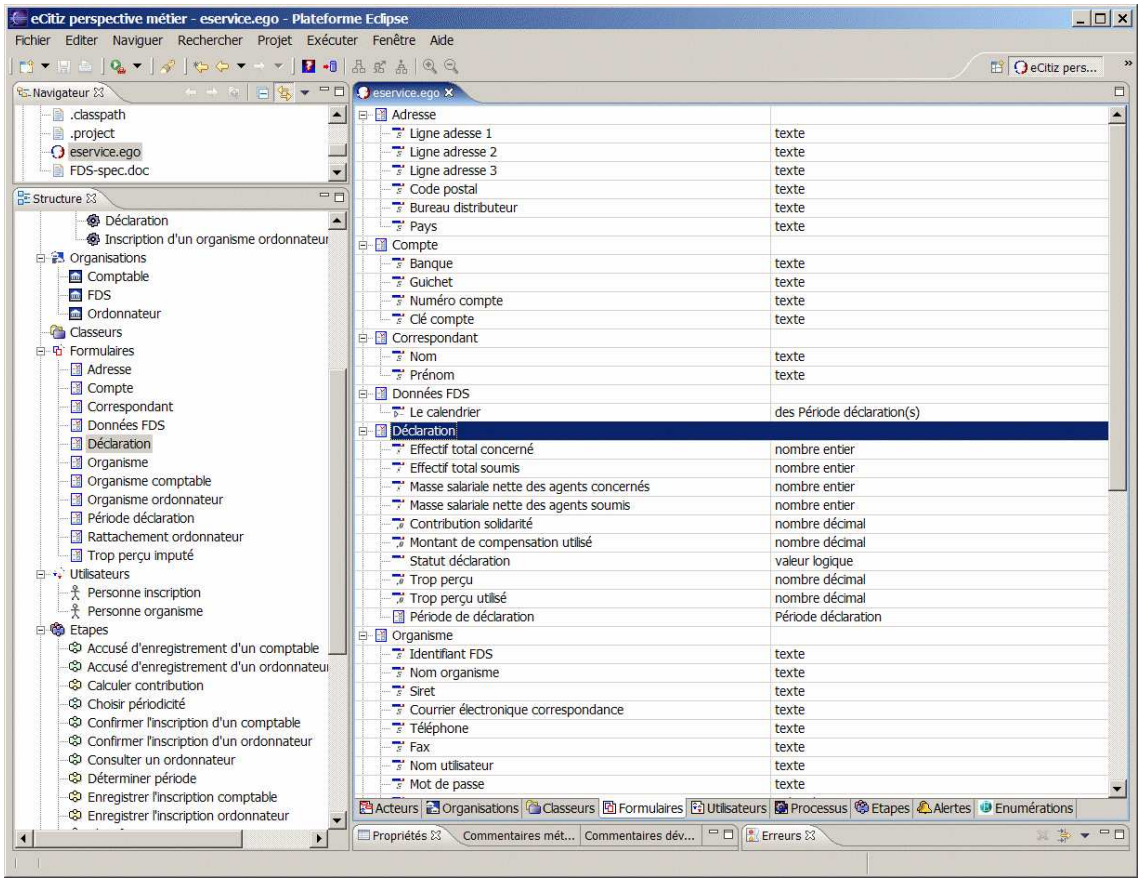

FIG. 5 – Spécification du modèle de données

Pour chaque champ affiché ou à saisir, il est possible de préciser :

- l'origine de la donnée à saisir ou à afficher dans le modèle de données général du projet ;
- le libellé correspondant au champ à saisir ou à afficher ;
- la possibilité de modifier ou non la valeur en question ;
- l'aspect graphique conditionnant la saisie dans le navigateur Internet (champ texte, liste déroulante, bouton radio, case à cocher, zone de texte libre, etc).

1.2.2 Besoin de l'approche par règles métier dans *e-Citiz*

Comme nous venons de le montrer, *e-Citiz* fait des experts métier l'élément central dans la mise en œuvre des systèmes d'information, en leur permettant de spécifier eux-mêmes les besoins fonctionnels. La plateforme d'e-services *e-Citiz* permet aux experts métier et technique de collaborer pour mettre en place plus efficacement un système d'information. En effet, les spécifications fonctionnelles des experts métier sont complétées et facilitées par les spécifications techniques des experts technique.

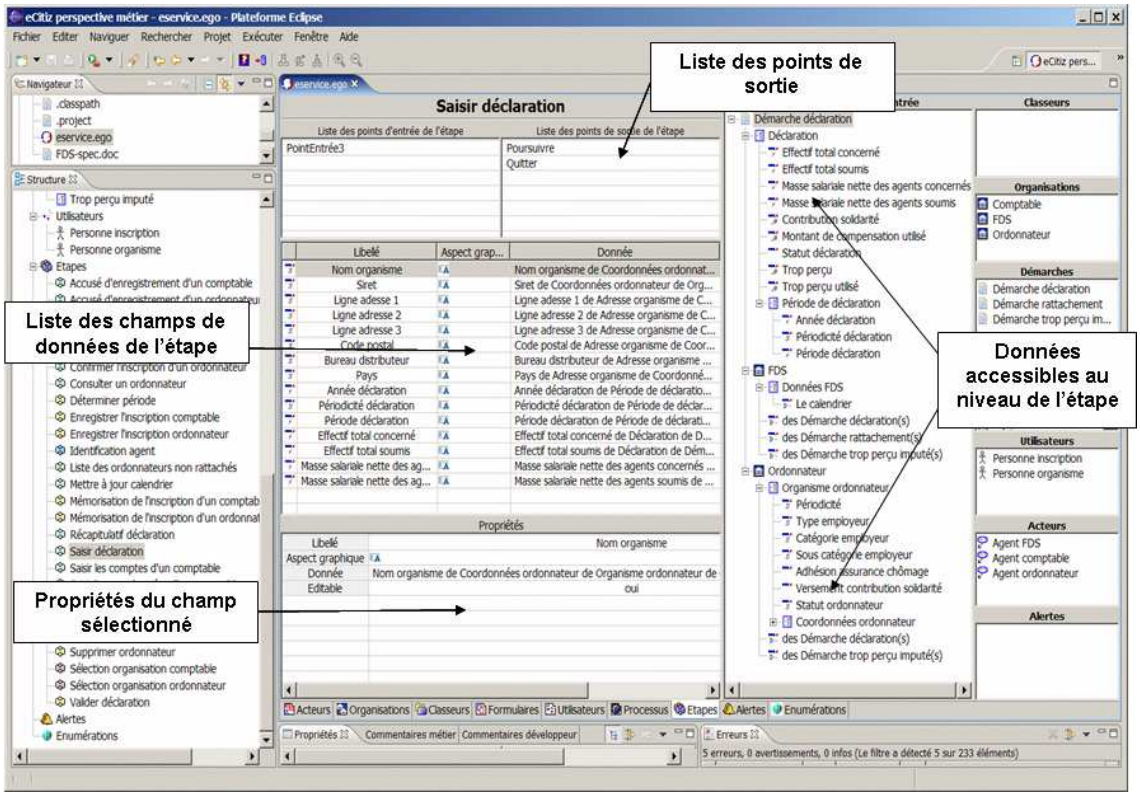

Fig. 6 – Spécification d'une étape de saisie de formulaire

Dans le Studio *e-Citiz*, les experts métiers définissent un modèle de données, des processus, des tâches et des acteurs intervenant dessus. Une fois ces différents éléments définis, les experts techniques définissent la logique nécessaire pour apporter des contraintes ou des amorçages sur ces derniers. Une étape d'amorçage est une étape d'initialisation de données. Cette logique peut être une validation syntaxique et/ou sémantique des éléments d'affichage ou de saisie des étapes correspondantes, de définir de manière dynamique leur visibilité et/ou leur accessibilité, de définir de manière dynamique la navigation (workflow) entre les étapes, d'amorcer des valeurs initiales, etc. Tout ceci se faisait à travers du code métier par les experts techniques depuis un cahier des charges fourni par les experts métier. Une préoccupation majeure de l'équipe R&D d'*e-Citiz* est d'offrir, de plus en plus, aux experts métier la possibilité de mettre en œuvre directement leur logique métier. Cette préoccupation a conduit à l'adoption de l'ARM, qui, comme nous le verrons plus loin dans ce document, a pour objectif de créer un lieu de collaboration entre actifs métier et système. L'ARM fût initialement prévue pour permettre aux experts métier, d'écrire dans un langage naturel, les règles pour la validation sémantique des données des étapes. L'utilisation de l'ARM fût rapidement élargie à d'autres besoins.

Validation métier ou sémantique des données des étapes

Lors de la spécification d'une étape de saisie (saisie d'un formulaire dans un navigateur Internet), l'expert technique peut rajouter des contraintes sur les éléments de l'étape. Il y a deux types de validation : une validation syntaxique et une validation sémantique. La validation syntaxique, qui se fait en amont, consiste à vérifier si les données que l'on a saisies sont correctes ou bien formées au niveau syntaxe. Par exemple vérifier si un email est bien formé, qu'un champ numérique ne contient pas des lettres, etc.
La validation sémantique vérifie si la valeur renseignée a un sens selon des critères bien définies. Par exemple vérifier si le domaine d'email est bien en France ou au Sénégal, ou encore vérifier qu'une date renseignée est bien incluse dans un intervalle défini, etc.
Nous pouvons remarquer qu'une validation syntaxique peut être commune à toutes les applications mais qu'une validation sémantique est propre au métier et donc fournie par les experts métier. Dans *e-Citiz* cette validation sémantique ou métier se faisait de manière statique dans le code par le développeur. Avec les règles métier, il est possible d'externaliser cette validation et permettre aux experts métier de la définir et la maintenir de manière dynamique en utilisant le langage naturel. En même temps le moteur *e-Citiz* bénéficierait d'outils plus performant (basés sur l'algorithme de Rete comme nous le verrons au chapitre suivant) pour l'exécution des règles métier.

Aiguillage de la navigation entres les étapes (workflow)

Un e-service est un ensemble de processus métier, qui à son tour est un enchaînement d'étapes. Comme nous l'avons déjà dit, une étape a un point d'entrée et un ou plusieurs points de sorties. Les branchements des entrées et des sorties dépendent du métier et donc spécifiés par les experts métier en utilisant l'éditeur correspondant dans le Studio *e-Citiz* (voir Figure 4). Cependant parfois, l'expert métier doit pouvoir gérer l'aiguillage selon le contenu des éléments des étapes. Dans ce cas, le développeur, en se basant sur le cahier des charges, faisait l'aiguillage de manière statique dans le code métier qu'il était seul à pouvoir accéder et maintenir. Par exemple l'expert métier peut vouloir définir le comportement métier suivant : "Si l'attribut "a des enfants" est vrai (cela veut dire que dans le formulaire du navigateur le champs "a des enfants" est coché) Alors "aller à la page de remplissage des informations pour les enfants (cela veut dire afficher la page ou il remplit les informations concernant les enfants)". Avec l'ARM un tel comportement peut être spécifié directement par l'expert métier en utilisant le langage naturel.

Visibilité et accessibilité des données dés étapes

Dans les étapes de saisie ou d'affichage, les éléments ont une forme graphique. Par exemple, une valeur booléenne pourra être une case à cocher, une collection pourra être une liste déroulante et ainsi de suite. Les éléments de saisie des étapes ont un état, qui peut être visible ou non. Comme état nous pouvons aussi avoir l'accessibilité qui dira

si dans le formulaire, l'élément est éditable ou non. Des fois l'expert métier a besoin de manipuler dynamiquement l'état d'un élément de l'étape (y compris les points de sortie qui permettent la navigation au sein d'une télé-procédure), par exemple il voudra exprimer : "Si "est marié(e)" est faux Alors rendre non éditable les éléments "prénom et nom de jeune fille de l'épouse"" , ou encore "Si "Date de naissance" est vide alors rendre invisible "le point de sortie *envoyer*" ". Comme dans les autres cas tout ceci se faisait de manière statique dans le code par le développeur et qu'avec les règles métier les experts métier pourraient le faire eux-même.

L'utilisation des règles métier est également envisageable pour l'amorçage de données ou encore pour le traitement métier d'un processus métier (business process) c'est-à-dire le traitement d'un processus dans sa globalité et non en terme d'étapes ponctuelles. L'intérêt d'utiliser l'ARM était réel pour *e-Citiz*.
Comme nous le verrons au chapitre suivant, il y a plusieurs moteurs de règles, aussi bien commerciaux que gratuits. Et ces divers moteurs de règles n'ont pas un formalisme commun pour représenter les règles métier afin de pouvoir passer d'un moteur à l'autre sans pour autant réécrire toutes les règles. Les réflexions autour de ces questions ont abouti au sujet de la thèse présentée dans ce document.
L'objectif de cette thèse est de voir comment proposer un tel formalisme et l'intégrer dans la plateforme *e-Citiz*, aussi bien au niveau Studio que Moteur.

Première partie

Règles métier et formalisme

Chapitre 2

L'approche par règles métier (Business Rule approach)

Sommaire

2.1 Introduction

La mise en place d'un système d'information répond à un besoin. Dans ce sens, les experts métier qui connaissent et définissent les besoins, travaillent avec une équipe système pour mettre en place le système souhaité. Les experts métier expriment leurs besoins sous forme d'un cahier des charges. Ce dernier est remis aux développeurs qui se chargent de concevoir l'application, l'implémenter, la tester, la valider, etc. Les experts métier qui sont à l'origine du système n'interviennent qu'au départ de la conception de l'application et à la fin (validation). Dans une telle approche, ils ne disposent pas d'outils pour faire évoluer le plus petit aspect du système sans l'aide de l'équipe système.
Au niveau des cycles de développement beaucoup de progrès ont été accomplis quant à la rapidité de la mise en place des systèmes d'information et leur vérification/validation, cependant la maintenance reste chère et réservée aux experts techniques.
L'expert métier connaît mieux que quiconque les règles qui régissent une application et donc, du début de la spécification jusqu'aux tests en passant par l'implémentation, son rôle demeure indispensable. Ainsi les experts métier doivent pouvoir, eux-mêmes, créer et faire évoluer leur logique pendant que les experts technique ne s'occupent que des aspects techniques [3].
Dans ce chapitre nous présentons l'approche traditionnelle utilisée pour mettre en place des systèmes informatique et de leur rigidité. Nous montrons comment l'approche par règles métier permet de résoudre les limitations de l'approche traditionnelle et quels sont les outils pour la mettre en oeuvre.

2.2 L'approche actuelle utilisée dans la mise en place des systèmes d'information

2.2.1 Présentation de l'approche actuelle

Un système d'information est l'ensemble des moyens techniques et humains qui permettent le traitement des informations au sein d'une organisation.
L'informatisation d'un système suit un cycle de vie (voir Figure 7) :

1. Une étude préalable : analyse de l'existant, recensement des besoins utilisateurs, étude de faisabilité et rédaction d'un cahier des charges.
2. Une analyse fonctionnelle : élaboration d'une solution conceptuelle (abstraction des moyens informatiques) et rédaction d'un dossier de conception.
3. Une analyse organique : définition des structures de données, choix des matériels et établissement du planning et des budgets de réalisation.
4. Une phase de développement : réalisation effective du système informatique, tests du logiciel réalisé.

5. Mise en service : rédaction d'un manuel, formation des utilisateurs, exploitation et maintenance du système.

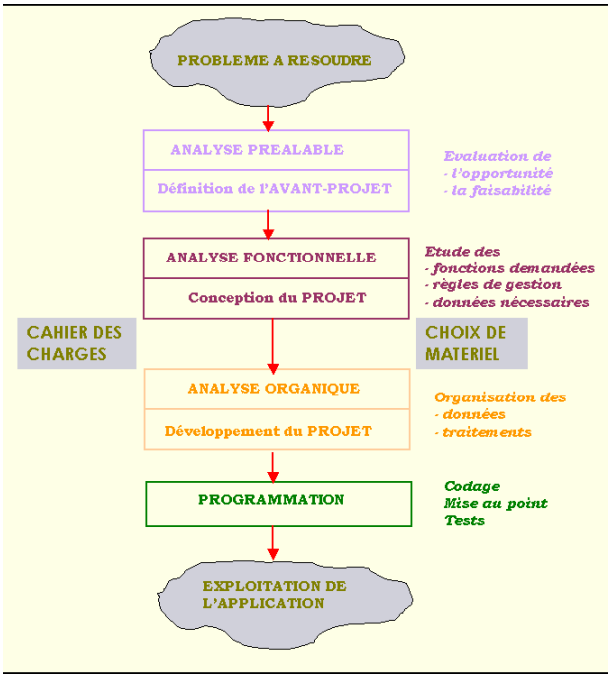

FIG. 7 – Cycle de vie d'un logiciel dans l'approche actuelle

Lors de la phase de développement, on peut recenser différentes étapes :
- Analyse de l'existant et des besoins des utilisateurs.
- Conception : choix des structures de données, des algorithmes, pour le logiciel.
- Réalisation : traduction des algorithmes dans le langage choisi.
- Tests :
  - Vérification : le logiciel répond à la définition des besoins.
  - Validation : le logiciel remplit les fonctionnalités désirées par l'utilisateur.
  - Test de non régression (dans le cas où l'existant est déjà fonctionnel).

2.2.2 Modèles de cycle de vie et de développement

Rappelons quelques modèles classiques de cycle de vie.

Cycle de vie en cascade

Le cycle de vie en cascade consiste en une succession de phases dont chacune est méthodiquement vérifiée avant de passer à l'étape suivante (analyse, conception, réalisation, tests, exploitation et maintenance).

Cycle de vie en spirale

Dans ce cycle on couple de manière itérative le prototypage et les aspects du cycle de vie en cascade. Le prototypage en spirale permet une réduction des risques d'erreurs.

Le cycle de développement classique dans un système se résume comme suit (voir Figure 8) : un expert métier crée un cahier des charges qu'il remet à l'expert technique qui lui, va créer l'application. Une fois l'application créée, l'expert métier peut intervenir pour demander des modifications.

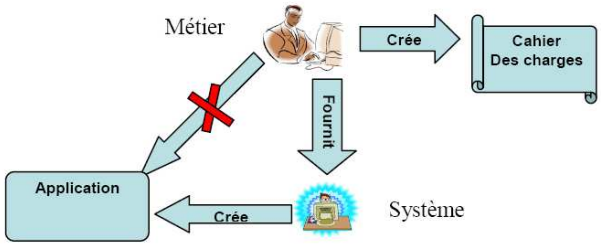

Fig. 8 – Avec l'approche traditionnelle l'expert métier ne peut pas lui-même faire les changements

Dans une telle architecture la coopération entre les deux pôles métier et système est limitée car elle n'a lieu qu'en début et fin de phase.
Une fois le système livré, lorsque la logique métier changera, il faudra à nouveau faire appel à l'équipe système pour traduire ces nouveaux besoins dans l'application. Dans l'approche actuelle beaucoup de progrès ont été accomplis quant à la rapidité de la mise en place des systèmes d'information et leur vérification/validation, cependant la maintenance reste chère et réservée aux experts techniques [4].
Nous allons maintenant introduire l'approche par règles métier afin de montrer comment faire collaborer les acteurs métier (analystes fonctionnels) et les acteurs systèmes (développeurs/analystes).

2.3 Le principe de l'approche par règles métier (ARM)

Dans toute application, nous ne nous posons plus la question de savoir "est-ce que le métier changera ?" mais plutôt "quand ces changements vont-ils intervenir ?". Un système doit être capable de s'adapter aux changements de l'environnement. Mieux encore, il faudrait que le comportement d'un système d'information puisse être modifié par l'expert métier, sans avoir à attendre que le service informatique soit disponible. C'est ce qui fait que l'ARM permette la réalisation de systèmes d'information guidés par le métier, pour le métier et en vue du métier [5].

Les règles métier permettent de séparer (voir la figure 9) la logique métier (*le comportement*) et la logique système d'une application (*le comment*). Ainsi, les experts techniques

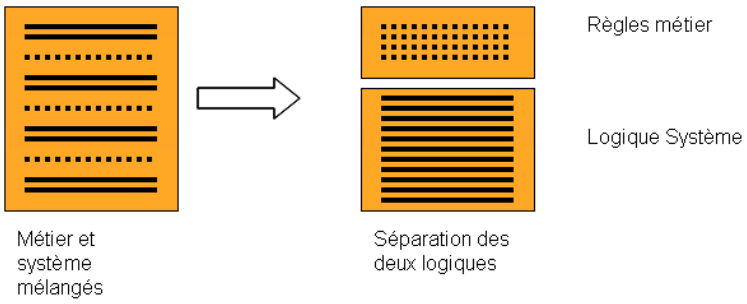

FIG. 9 – Séparation du métier et du système

(développeurs) ne s'occupent que de la logique système applicative tandis que les experts métiers se chargent de la maintenance métier (qui change le plus souvent) dans un environnement "zéro-développement" en utilisant des éditeurs en langage naturel. Le cycle de développement devient plus court car de la conception aux scénarii de tests, en passant par le codage, les experts métiers travaillent en étroite collaboration avec l'équipe technique.
Avec l'ARM, 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 [6].
Les règles sont au cœur de toute application, cependant le plus difficile est de les recenser et de les structurer en une approche de management plus effective [7]. L'utilisation d'un gestionnaire de règles métier ou Business Rules Management System (BRMS) facilite le recensement et la mise en œuvre des règles métier. Le principe des règles métier n'a rien de nouveau, c'est une application directe de théories du domaine de l'intelligence artificielle des années 70 [8] qui, à cette époque, face à un problème de puissance de calcul, passaient pour être utopiques.
La Figure 10 montre l'architecture d'un système orienté règle métier. En 1. les deux acteurs métier et système peuvent se consulter pour spécifier le futur système. Durant cette phase il peut être judicieux de mettre en place une ontologie commune afin qu'il n'y ait pas d'ambiguïté sur les termes [9]. Les experts métier pourront également participer à la conception des modèles, à l'écriture des scénarii de tests et tester eux-mêmes leurs règles. En 2. les experts système mettent en place la logique applicative de l'application qui intègre un moteur de règles. Parallèlement les experts métier utilisent un éditeur de règles. La Figure 11 illustre l'édition d'une règle qui s'énonce ainsi : "pour un client de type platinium qui achète plus de 500 articles lui faire 50% de remise".

Avant de commencer la création de règles métier, l'expert système en collaboration avec l'expert métier fournit pour chaque élément du modèle et son état/comportement intervenant dans l'écriture, l'équivalent en expression métier : par exemple, "*est un client platinium*" pour la méthode *isPlatiniumCustomer()* de "*client* ", etc. Ainsi, avec l'éditeur

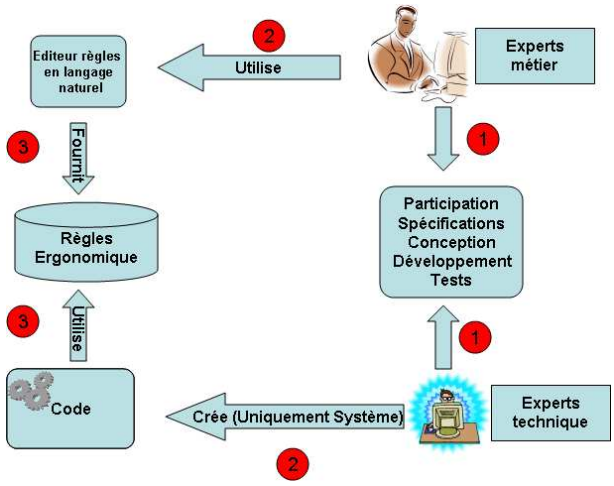

Fig. 10 – Architecture d'un système orienté règle métier

Si
client est de *catégorie platinium*
ET
Quantité achetée >= 500
Alors
Appliquer une remise de 10% au client

Fig. 11 – Exemple de règle en langage naturel dans un éditeur de règles

personnalisé, l'expert fonctionnel peut entrer la règle comme présentée dans la Figure 11. En 3. les règles métier écrites avec un langage naturel sont stockées dans la base de connaissance sous un certain formalisme (à la Figure 12, du XML). Dans cette phase, l'expert système référencie la base de connaissance qui sera utilisée par le moteur de règles. Avec cette architecture, les deux logiques (métier et système) peuvent évoluer séparément, de manière parallélisée ou non.

```
<ruleset name="règles remise noel">
        <rule name="platinium500" prioprity="5">
            <condition>
              <And>
                  customer.isPlatiniumCustomer();
                  customer.getQuantity() >= 500 ;
              </And>
            </condition>
            <action>
                  customer.setDiscount(10);
            <action>
        </rule>
<ruleset>
```

FIG. 12 – Exemple de règle dans un formalisme XML

Le plus important ici est, comme le montre la Figure 13, que les experts métier puissent modifier les règles métier sans que l'équipe système n'intervienne. Ainsi, si l'expert métier juge nécessaire de mettre à jour les règles métier, il lance son éditeur en langage naturel et fait les modifications.

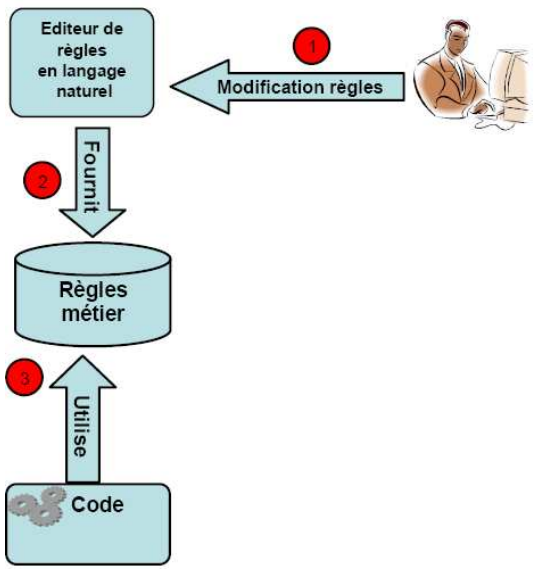

FIG. 13 – Processus de modification des règles

En 2. les modifications sont stockées dans la base de connaissance. En 3. la logique système référencie toujours la base de connaissance mise à jour.

2.4 Règle métier

Les règles métier permettent une encapsulation des besoins de l'utilisateur [3]. Les règles métier sont des règles qui sont sous la juridiction des experts métier. Le terme règle métier a un sens différent selon un point de vue métier ou professionnel des systèmes d'information (SI) :

- D'un point de vue système d'information, "les règles métier sont des formulations qui définissent ou contraignent quelques aspects d'un métier. Son but est de structurer un métier (politique, savoir-faire), de contrôler ou d'influencer le comportement d'un métier" [5, 10].
- 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." [9]

De manière plus basique et simple on peut dire qu'une règle métier est un couple de "SI - ALORS", "SI" étant la partie condition et "ALORS" la partie action qui est exécutée si la partie condition est évaluée à vraie. La partie "SI" est également appelée *left-hand side (LHS)*, *prédicat* ou *prémisse*. La partie "ALORS" est aussi appelée *right-hand side (RHS)*, *actions* ou *conclusions* [8].
Les règles métier sont du ressort des experts fonctionnels du domaine. Les règles métier peuvent provenir des déclarations de la politique de l'entreprise, d'un système d'information existant ou tout simplement de personnes travaillant dans l'entreprise et qui ont su acquérir au fil des années une expérience considérable. Cependant le plus difficile est de les recenser et de les structurer en une approche de management plus effective.
Dans chacun des cas la nécessité d'avoir un système de gestion de règles métier (SGRM) ou Business Rules Management System se fait ressentir.

2.5 Les concepts clés d'un gestionnaire de règles (SGRM)

Un gestionnaire de règles métier doit pouvoir proposer trois (3) fonctionnalités :

- Pouvoir spécifier les règles de telle sorte que cela soit indépendant du mécanisme utilisé pour obtenir et mettre à jour les données ou du mécanisme utilisé pour exécuter les actions. Cette spécification doit pouvoir se faire en utilisant un langage métier de haut niveau compréhensible par les experts fonctionnels et/ou les utilisateurs de l'application (du langage naturel).

- Créer un moyen de pouvoir choisir les bonnes règles au bon moment avec le bon ordre sans que cela ne soit contrôlé par le code de l'application.
- Permettre une organisation et une gestion des règles qui ne dépendent ni n'affectent le reste du code de l'application. Le but étant qu'il n'y ait pas de couplage fort entre l'application au sens applicative et les règles métier.

La plupart des systèmes de règles métier commerciaux incorporent ces trois éléments sous forme d'un langage de règles, d'un ensemble de bibliothèque et d'un moteur de règles (engine). Concernant les BRMS gratuits ou open source, comme nous allons le voir dans la suite de ce document, bien que certains se démarquent, ils ne sont pas très aboutis.

2.5.1 Spécifier les règles métier

La spécification des règles métier doit respecter plusieurs préalables. Le premier et le plus important est d'être complétement indépendant des mécanismes utilisés pour manipuler les données et effectuer les actions. Actuellement l'une des limitations de l'ARM est que, si l'entreprise souhaite utiliser d'autres types de moteurs de règles, les règles doivent être réécrites dans l'implémentation propre au nouveau moteur [11].
Seul un système d'écriture de règles métier standardisé offre la flexibilité et la transparence nécessaire pour fonctionner indépendamment de l'implémentation. Dans le chapitre 3 concernant le formalisme de règles métier nous parlerons plus en détails de ce problème, des solutions en cours de standardisation ainsi que notre apport dans ce sens.

Langage de règle

Quelle que soit la technique utilisée pour représenter une règle métier (visualisation sous forme de langage naturel, de table ou d'arbre), il est nécessaire d'avoir un langage qui permet de spécifier la syntaxe et la sémantique des règles.
Un langage de règle métier se doit d'être intuitif pour les utilisateurs métier, ainsi ils peuvent participer au développement, aux tests et aux processus de modification, réduisant ainsi la durée de l'implémentation et les erreurs d'interprétation entre l'intention (la volonté) du métier et la réalisation de celle-ci.

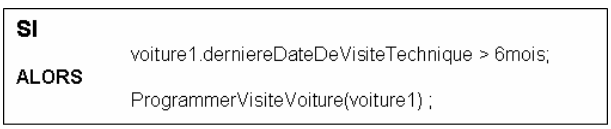

FIG. 14 – Règles en langage technique

Les Figures 14 et 15 montrent la même règles mais écrite différemment suivant la vue technique ou métier de l'utilisateur.
Des fois, présenter les règles sous forme de table ou d'arbre peut être plus intuitif et plus productif pour les experts fonctionnels.

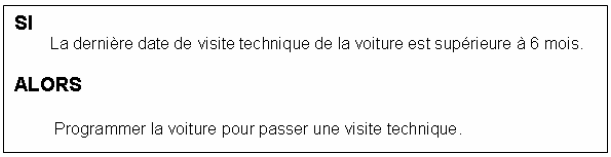

Fig. 15 – Règles en langage naturel

2.5.2 L'exécution des règles métier

L'un des principaux avantages d'avoir un BRMS complet est la réduction de la complexité de la mise en œuvre d'une logique applicative complexe pour permettre l'exécution de la logique métier. Le BRMS fournit un outil plus sophistiqué pour l'exécution des règles : le moteur de règles.
Les moteurs de règles peuvent être subdivisés en deux catégories : ceux qui font un chaînage avant (forward chaining) et ceux qui font le chaînage arrière (backward chaining). Il en existe qui font les deux chaînages en même temps, dans ce cas on les appelle des moteurs hybrides. Comprendre le mode de fonctionnement de ces chaînages est important pour comprendre pourquoi un système de règles de production est différent et comment tirer le meilleur d'eux.
Avant de présenter les chaînages, nous allons d'abord parler des règles de production.

Définition d'une règle de production

Une règle de production est un bloc de logique de programmation qui spécifie l'exécution d'une ou plusieurs actions dans le cas où ses conditions sont satisfaites. Les règles de production sont élaborées sans se préoccuper de l'ordre d'exécution, qui est sous le contrôle du moteur d'inférence. Une règle de production peut avoir plusieurs conditions et actions. Mark Stefik définit une règle de production, dans [12], comme suit : chaque règle a deux parties majeures, qui sont appelées la partie *"Si (if-part)"* et la partie *"Alors (then-part)"*. La partie *"Si"* est composée de conditions qui doivent être testées. Si toutes les conditions de la partie *"Si"* sont vraies, alors les actions dans la parties action sont exécutées. Les parties composant une règle de production peuvent, des fois, avoir d'autres désignations. La partie *"Si"* est connue sous les noms : partie situation, conditionnelle, antécédent ou gauche (Left Hand Side - LHS). La partie *"Alors"* est connue sous les noms : action, conséquence ou droite (Right Hand Side - RHS). La syntaxe d'une règle de production est la suivante :
<marqueur de conditions> <conditions> <marqueur d'actions> <actions> où :

- <marqueur de conditions> indique que les expressions suivantes sont des conditions qui peuvent être évaluées par le moteur d'inférence.
- <conditions>

  1. Cette partie est constituée d'une ou plusieurs expressions booléennes qui sont connectées par les opérateurs logiques ("et", "ou", "non").

2. Cette partie doit être évaluée à "vraie" pour que la partie action s'exécute.

- <marqueur d'actions> indique que les expressions suivantes sont des actions.
- <actions>

1. Cette partie est constituée d'une ou plusieurs expressions d'action.
2. Les actions sont exécutées dans l'ordre où on les rencontre.
3. Les actions sont exécutées si seulement la partie condition est évaluée à "vraie"

Une règle de production est interprétée de la sorte : si les expressions booléennes, qui sont définies dans la partie conditions, sont évaluées à "vraie" alors exécuter les actions de manière séquentielle.
Les règles de productions sont exécutées par, et leur ordre d'exécution contrôlé par un agent logiciel appelé **moteur d'inférence** comme le montre la Figure 16. L'agenda est l'endroit, dans le moteur d'inférence, où l'on stocke les règles durant l'exécution.

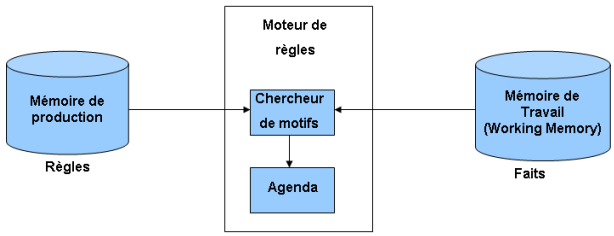

FIG. 16 – Architecture moteur de règles ou moteur d'inférence

Lors d'un contexte d'exécution dans un moteur d'inférence :

- Une règle peut être ré-exécutée si les données utilisées dans ses conditions changent.
- Les règles sont définies dans n'importe quel ordre.
- Les faits peuvent être définies en termes de classes ou collections d'objets. Les faits représentent les données que l'on connaît du problème que l'on cherche à résoudre.
- Les règles sont exécutées en fonction des objets ou autres données injectés dans le moteur, alors on parle de *chaînage avant* ou *forward chaining* ou encore de raisonnement orienté données. Les règles peuvent aussi être orientées résultat ou but, on parle alors de *chaînage arrière* ou *backward chaining* ou encore de raisonnement orienté objectif ou but. Un autre mode possible est le mode séquentiel que nous verrons ci-dessous.

Pour pouvoir être correctement gérée par un moteur d'inférence, une règle de production doit :

- Avoir un identifiant unique permettant de l'adresser de manière unique (cela peut être le nom).
- Etre formulée de telle sorte qu'elle puisse être utilisée lors d'un chaînage avant ou arrière.

La sémantique opérationnelle en mode séquentiel

En mode séquentiel il n'y a pas de ré-évaluation des règles durant l'exécution, c'est-à-dire même si durant cette phase les parties condition de certaines règles deviennent vraies, ces règles ne s'exécutent pas.
En mode séquentiel il n'y a pas d'étape de résolution de conflit car les règles sont exécutées au fur et à mesure qu'on les rencontre. Deux règles R_1 et R_2 sont en conflit si elles sont toutes les deux sélectionnables sur les même données, c'est-à-dire si pour C_R1, condition de R_1, et C_R2 condition de R_2, nous avons $C_R1 \subseteq C_R2$ ou $C_R2 \subseteq C_R1$. La phase de recherche de motifs se divise en deux qui sont les étapes de calcul et d'évaluation. La sémantique opérationnelle en mode séquentiel devient alors :

1. Calcul : calcul des variables suivant l'état des données (faits).
2. Evaluation : évaluer les conditions des règles en se basant sur l'état des données. Chaque instance de règle est traitée séparément.
3. Action : exécution la partie action.

Ce qu'il est important de noter ici est que l'ordre d'exécution des règles en mode séquentiel dépend de leur séquence dans le ruleset.

La sémantique opérationnelle en mode chaînage arrière ou backward chaining

Le chaînage arrière est orienté but ou résultat (goal-driven). Dans ce cas ci, le traitement commence avec une conclusion que l'on cherche à satisfaire (voir Figure 17). Si l'on n'arrive pas à satisfaire la condition initiale alors reprendre la recherche avec une de ses sous-conditions. Et l'on continue ainsi de suite jusqu'à ce que la condition soit satisfaite ou qu'il n'y ait plus de sous-conditions. Prolog est basé sur le chaînage arrière.

La sémantique opérationnelle en mode chaînage avant ou forward chaining

Dans le cadre du chaînage avant, les règles de production sont écrites sans tenir compte de leur ordre d'exécution qui est du ressort du moteur de règles. La sémantique opérationnelle devient dans ce cas :

1. Recherche de motifs (pattern matching) : dans ce cas, en plus de se baser sur les conditions des règles, les déclarations des variables sont considérées comme étant des conditions. Par exemple si on a la déclaration suivante : *var x = Personne(age :28 ; nom :Diouf)* alors si dans les faits injectés dans le moteur de règles il n'y a pas d'instances de *Personne*, la partie condition de la règle est considérée comme fausse.
2. Résolution de conflit : sélection des instances de règles à exécuter selon une valeur définie (priorité, ruleflow ou autre politique).
3. Action : changer l'état des faits en exécutant les actions des instances de règles sélectionnées.

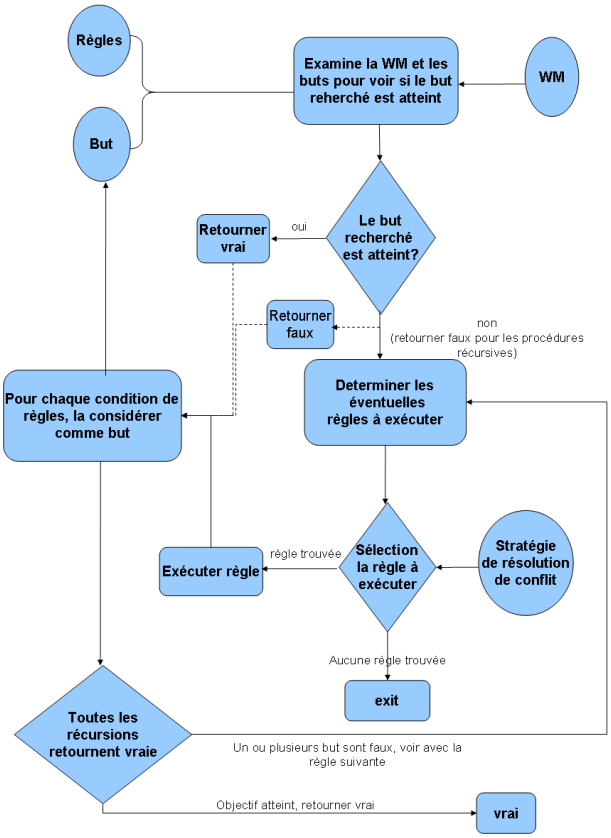

FIG. 17 – Chaînage arrière ou backward chaining

Cette séquence est répétée pour chaque règle jusqu'à ce qu'il n'y ait plus de règles pour faire la recherche de motifs ou qu'un état stable soit atteint (l'état des objets dans le moteur de règles n'évolue plus). Il est important de noter que :

- Une règle peut s'exécuter plusieurs fois.
- Une action peut modifier une donnée, pouvant ainsi entraîner une reconsidération des instances de règles déjà sélectionnées durant l'étape de recherche de motifs. Par exemple une instance de règle peut être supprimée de la sélection à exécuter parce que les conditions ne sont plus respectées, comme encore une instance de règle peut être rajoutée car ses conditions deviennent vraies.

Le chaînage avant est orienté données et est donc "réactif" (voir Figure 18). Les faits (données) sont injectés dans la mémoire de travail ou working memory (WM) entraînant l'élection d'une ou plusieurs règles pouvant être exécutées dans l'agenda.

La performance d'un moteur de règles varie beaucoup avec l'implémentation. Souvent les performances diminuent avec le nombre de règles grandissant dans la base de connaissance ou *repository* de règles. La plupart des moteurs de règles en chaînage avant utilisent un algorithme à base de RETE [13, 14, 15] pour optimiser ces performances qui sont finalement asymptotiquement indépendant du nombre de règles. Le but ici n'est pas une

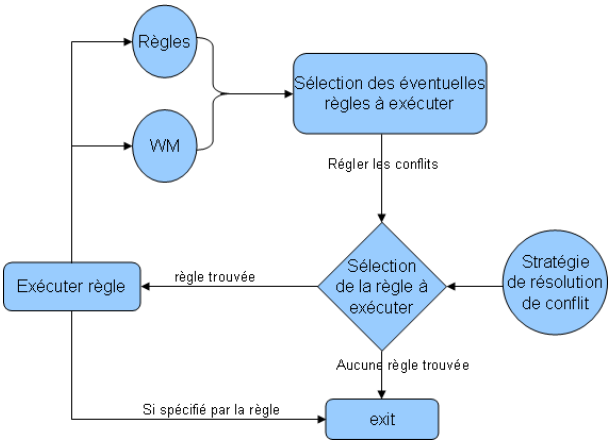

Fig. 18 – Chaînage avant ou forward chaining

étude de RETE mais c'est le seul à fournir des performances en même temps rapide et supportant les montées en charge.

L'algorithme de RETE

La compréhension de l'algorithme de RETE [14, 15, 16] peut être utile dans la mise en œuvre d'un système orienté règle. En des termes simples, le but d'un moteur de règles est de continuellement appliquer un ensemble d'éléments (les règles) sur un ensemble de données (la working memory). La working memory est un ensemble de données formé de connaissances factuelles du monde, du domaine.
Un programme orienté règles a un ensemble fixé de règles tandis que la working memory change continuellement. Cependant il est connu qu'entre deux exécutions d'une règle, la plupart des éléments dans la working memory sont fixés ou évoluent peu. Bien qu'il y ait de nouveaux faits qui sont ajoutés et d'autres supprimés, le pourcentage de changement par unité de temps est assez petit. Pour cette raison, l'implémentation de base des moteurs de règles n'était pas très efficace. Cette implémentation consistait à construire une liste de règles et de continuellement boucler dessus en vérifiant si la partie gauche d'une règle (les conditions) est vérifiée et auquel cas on exécute la partie droite (les actions) [17]. Ceci n'est pas efficace parce que la plupart des vérifications qui ne sont pas vraies lors d'un cycle, ont peu de chance de l'être lors des itérations suivantes. Une fois que la working memory devient stable la plupart des tests seront répétitifs. Ce mécanisme est appelé *la règle de recherche de faits ou mode séquentiel* et sa complexité est de $O(RF^P)$, où R est le nombre de règles, F le nombre de faits et P le nombre moyen de prémisses dans les règles. Quelques moteurs de règles, dans le souci d'être plus performants, utilisent une méthode appelée algorithme de Rete (réseau en latin). Un article classique [14] sur l'algorithme de RETE est la base des moteurs de règles les plus rapides : OPS5 [18, 19], Jrules [20], Drools [21], Jess [8], Blaze [22].

Principes de l'algorithme :

Dans l'algorithme de RETE, la méthode peu efficace décrite plus haut est réduite (conceptuellement) en mettant dans un cache les résultats des tests précédents lors des itérations. Seuls les nouveaux faits sont testés par rapport aux règles qui seront plus à même d'être respectées. Ainsi la complexité devient $O(RFP)$ et linéaire à la taille de la working memory.

Implémentation de l'algorithme :

L'algorithme de RETE est implémenté en construisant un réseau de nœuds. Chaque nœud représente un test de la partie gauche (condition) d'une règle. Les faits qui sont injectés ou retirés de la working memory sont traités par le réseau de faits. Sur la couche la plus basse du réseau on trouve les règles individuelles. Lorsqu'un ensemble de règles parcourt le réseau jusqu'à arriver à la couche basse, cela veut dire que tous les tests correspondant à cette règle ont été réussis et ainsi cette dernière est activée. Cette règle devrait avoir sa partie droite exécutée.
Dans l'arbre de faits il y a deux types de nœuds : *à une entrée* et *à deux entrées*. Les nœuds à une entrée effectuent les tests sur les faits individuels alors que les deux entrées effectuent les tests sur les faits et les fonctions de regroupement.
Exemple :
Dans l'exemple suivant nous avons la définition de deux règles avec les conditions x, y, z pour la règle example-2 et les conditions x, y pour la règle example-3.

```
(defrule example-2        (defrule example-3
(x)                       (x)
(y)                       (y)
(z)                       => Action)
=> Action)
```

Pour les règles ci-dessus on aura l'arbre représenté à la Figure 19.

Amélioration de l'algorithme

Deux optimisations simples peuvent rendre RETE plus performant [23] :

- La première est de permettre un partage des nœuds simples dans l'arbre. Si l'on considère l'exemple en haut, il y a 5 nœuds simples dont trois différents, on peut partager x? et y? alors nous obtenons l'arbre représenté à la Figure 20.

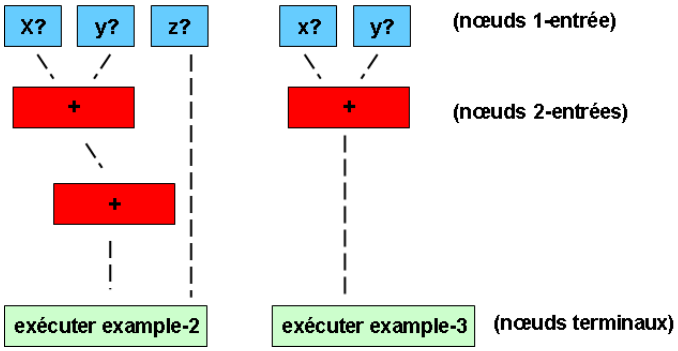

Fig. 19 – Implémentation de base de RETE

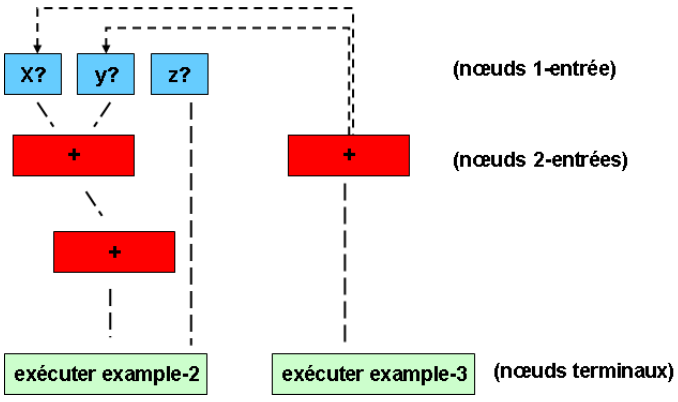

Fig. 20 – Amélioration de RETE par partage des nœuds simples

- La seconde optimisation consiste à partager les nœuds de jonction. Les nœuds de jonction gardent en mémoire tous les faits ou groupe de fait qui arrivent sur l'une des entrées. Ces deux entrées ont des mémoires séparées qui sont traditionnellement appelé mémoires *alpha (ou gauche)* et *beta (ou droite)*. Ces deux mémoires seront consultées aux prochains cycles et mise à jour [8].
  Toujours dans l'exemple précédent on obtiendra le réseau représenté à la Figure 21.

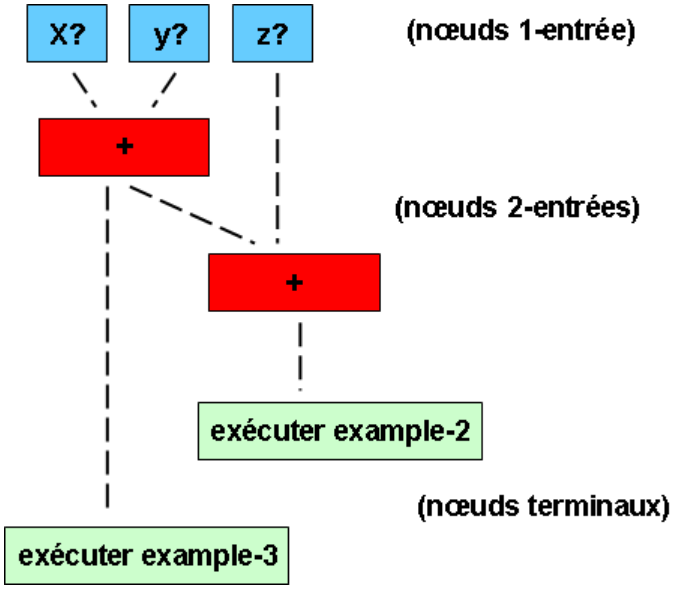

FIG. 21 – Amélioration de RETE par partage des nœuds de jonction

La taille du réseau diminue donc le parcourt plus rapide.
De plus en plus de moteur de règles existent et de plus en plus de travaux portent sur l'optimisation de RETE (actuellement on parle de RETE3).
On ne peut pas dire de manière absolue qu'utiliser le mode séquentiel ou le mode RETE soit mieux ou pas car tout dépend de la nature des règles que l'on a et de comment est-ce qu'on les utilise. Dans le cas où les règles ne sont pas liées entre elles et que leur exécution se fait ponctuellement à un moment bien donné on peut préférer le mode séquentiel au mode RETE.

2.5.3 La gestion des règles métier

Les règles métier constituent la partie la plus prompte à des changements dans un système d'information. La politique de vente d'un produit qui change, l'arrivée d'un nouveau produit, une opération marketing etc., requièrent des changements dans l'aspect comportemental ("métier") de l'application. Dans une organisation traditionnelle, c'est l'expert fonctionnel qui ressent la nécessité de faire des changements, ensuite les spécifie, les transmet à l'équipe informatique. Il attend que ces derniers aient le temps de traiter cette tache,

la réalise, la fait valider par l'expert fonctionnel qui, s'il n'est pas d'accord, un autre cycle est engagé. Ce cycle peut prendre énormément de temps cependant il n'y avait pas d'alternative. Du fait que les règles métier soient complètement indépendantes de la logique système de l'application, elles peuvent alors être changées sans impact sur le reste et avec l'utilisation d'un langage naturel, ceci peut être effectué par un non informaticien.

2.6 Analyse et design d'un système de gestion de règles métier (SGRM)

L'objectif d'un système de gestion de règles métier est de faciliter la mise en place d'un système orienté règles métier en fournissant les outils nécessaires, allant de leur définition à leur exécution en passant par leur stockage. Par analogie, nous pouvons comparer un SGRM à un système de gestion de base de données (SGBD) qui a pour objectif de rendre simple la gestion d'une base de données.
Nous allons maintenant voir ce qu'un SGRM devrait comporter et permettre de faire.

2.6.1 Ce que doit apporter un SGRM

Le principal défi d'un SGRM est de permettre la modification des règles métier par des experts fonctionnels sans aucune aide du service informatique, et ceci aussi souvent que nécessaire.
Un système de gestion de règle doit respecter un minimum de contraintes listées ci-dessous :

- **Permettre la gestion de processus métier :** les entreprises réalisent leurs services en appliquant de manière répétitive des processus métier qui sont définis et orientés par des règles métier générales ou spécifiques suivant une situation ou un client donné. Donc permettre la gestion des besoins d'un processus métier est fondamentale. A leur niveau le plus élémentaire, les règles ne sont que des expressions de la forme "Si X est vrai alors faire Y". Durant leur exécution, les règles peuvent être appelées à interagir pour réaliser un processus métier, donc il est nécessaire de pouvoir les manipuler de manière unitaire et globale.
- **Permettre la maintenance de l'application :** les applications d'entreprise ont besoin, de nos jours, d'un nouveau paradigme de maintenance plus rapide et plus simple. Les besoins de changement des règles métiers sont premièrement ressentis par les utilisateurs (experts fonctionnels et/ou utilisateurs de l'application) du système. La manière la plus simple et la plus sécurisée pour faire ces modifications et de mettre à leur disposition les outils nécessaires pour qu'ils puissent faire les changements eux-même afin de maintenir une quelconque politique, procédure et règle de l'entreprise. En mettant le service informatique (IT) "hors de cette boucle" lors de la maintenance, les technologies à base de règles métier peuvent permettre des applications plus adaptables.

- **Permettre la cohérence de la politique de l'entreprise :** la politique de l'entreprise doit être appliquée partout dans l'entreprise aussi bien de manière transversale que verticale. Plus le nombre d'applications augmente au sein de l'entreprise, plus le nombre de place où la logique métier est utilisée augmente aussi. Du coup on peut avoir différentes applications, avec des buts qui sont similaires ou se chevauchant, être déployées sur des médias de communications différents, par exemple applications sur du mobile, un call center, webs services, etc. Cependant une fois que les règles métier changent, cela doit l'être à tous les niveaux et en même temps quel que soit le support de déploiement, d'où l'intérêt d'avoir un système de gestion centralisé des règles.

Ainsi en séparant la logique métier du code de l'application, le système de gestion de règles permet la création de services de logique métier partagé.

2.6.2 Performance et portabilité

L'efficacité d'un système de gestion de règles peut être évaluée en terme de scalabilité et de performance comme décrit ci-dessous.

Montée en charge

Il est vital qu'un système de gestion de règles métier soit capable de gérer avec efficacité le nombre de règles qui augmente dans le temps.

Performance

Suivant l'application, les besoins de performance peuvent être très importants. La performance d'un moteur de règles peut beaucoup varier suivant l'implémentation (l'éditeur). En général la performance baisse en fonction du nombre de règles. Cependant certains algorithmes utilisés dans les moteurs commerciaux à base de RETE montrent que la performance est asymptotiquement indépendante du nombre de règles (voir 2.5.2).

2.6.3 Les besoins spécifiques d'un moteur de règles

Le moteur de règles doit avoir un certain nombre de fonctionnalités pour un bon fonctionnement du système de gestion de règles. Il y a au moins 6 grandes fonctionnalités :

1. Permettre la modification d'objets et de modèles de données existant.
2. Permettre l'interaction avec d'autres sources de données.
3. Pouvoir gérer simultanément des requêtes provenant de différentes applications ou de différents utilisateurs.
4. Pouvoir réaliser, en se basant sur des conditions, la sélection de la bonne règle.

5. Possibilité d'intégration avec des outils d'audit pour pouvoir déterminer quelle règle a été exécutée pour telle décision.
6. Offrir une possibilité d'optimisation des chaînes de décisions complexes sans que cela ne nécessite un contrôle explicite depuis l'application appelante.

2.6.4 Les besoins spécifiques d'une interface de système de gestion de règles

L'éditeur de règle doit aussi remplir certain critères :

- Permettre la gestion de rulesets, des services de partitionnement (droit d'accès, verrouillage), des services d'assemblage et de déploiement.
- Permettre la définition d'un modèle métier.
- Permettre la validation des règles métier.
- Permettre une représentation des règles qui permet une abstraction complète du langage de règle.

2.6.5 L'architecture d'un système de gestion de règles métier

Pour une utilisation effective du système de gestion de règles métier, la logique métier devrait être indépendante des mécanismes utilisés pour manipuler les données ou implémenter les décisions. Les règles métier devraient être implémentées indépendamment de tout système devant l'utiliser.

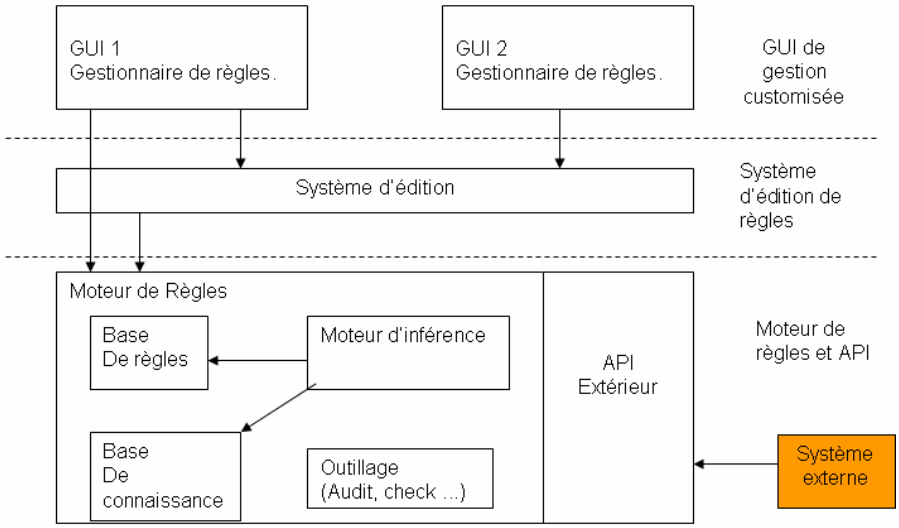

Fig. 22 – Architecture d'un système de gestion de règles

La Figure 22 décrit les concepts identifiés dans un gestionnaire de règles.
Un système de gestion de règles est composé de 3 couches :

- La couche exécution et intégration.
- La couche éditeur de règles générique.
- La couche interface personnalisable.

Ces trois couches s'inter-connectent au travers d'interfaces clairement définies.
La plupart des gestionnaires de règles ne fournissent pas les moyens d'avoir une interface utilisateur personnalisé alors que le plus souvent les besoins des experts métier sont spécifiques à chaque entreprise et chaque domaine.

2.6.6 La couche moteur de règles et API d'intégration

C'est la couche la plus basse d'un système de gestion de règles métier.

L'API extérieur ou d'intégration :

Cette API permet d'interfacer le moteur à une application externe. Elle constitue le point d'ancrage entre le moteur et un système externe. L'API permet la gestion des règles en ces termes :

- Injection des faits dans la mémoire du moteur (working memory).
  Par injection il faut entendre, l'insertion, le retrait et la modification.
- Ajout d'un ensemble de règles (dans l'agenda).
- Interrogation de l'état de la working memory.
- Audit de l'exécution (règles exécutées)

Moteur de règles :

Le moteur de règles est un agent logiciel qui réalise des actions résultant d'un pattern matching (recherche de motif).
En général un moteur de règles est constitué des composants suivants :

- Un langage de règle : un langage de règles définit la syntaxe et la sémantique d'une règle. Nous reviendrons plus en détail sur les langages de règles.
- Une base de règles : Les règles sont stockées dans un endroit commun. On l'appelle souvent *agenda*, c'est la base de connaissance. Le principe d'un moteur de règles est de partir de connaissances apprises ou dictées, encapsulées sous forme de règles métier, et de prendre des décisions suivant les faits (données) fournis.
- Une *working memory* qui est le lieu où l'on stocke tous les faits c'est-à-dire les instances sur lesquelles travailleront les règles.
- *Moteur d'inférence* : il gère les interactions entre les utilisateurs de l'application, la base de règles et la base de connaissance. Il est responsable de l'assortiment (matching), de la sélection, du lancement et de l'exécution des règles.

- Un *ensemble d'outils* : cela peut être soit de l'audit, de la validation (check), du debugage, etc. des règles.

2.6.7 La couche éditeur ou système d'édition

Le but de la couche éditeur est de permettre l'accès (modification) de la logique métier en dehors du système (application), permettant ainsi aux experts métier de pouvoir le faire eux même sans avoir à interrompre le service. Beaucoup de gestionnaires de règles permettent la formulation des règles en langage naturel (nous préférerons plutôt le terme langage métier).

2.6.8 La couche GUI du gestionnaire de règles

Cette couche permet de faire abstraction de tout ce qui est programmation et techniques à l'utilisateur métier, en fournissant un *environnement zéro-programmation*. Grâce à cette couche, l'expert métier pourra définir, supprimer, modifier des données pour la manipulation de règles sans rien connaître à la programmation. Cette couche doit permettre une personnalisation de l'interface graphique pour un besoin spécifique.

2.6.9 Les fonctionnalités que doit avoir un SGRM fini

Actuellement il y a plusieurs systèmes de gestion de règles métier aussi bien propriétaires qu'open source. Cependant les produits commerciaux sont mieux élaborés car ayant beaucoup investi pour avoir de bons algorithmes, de bons langages de règles et des interfaces graphiques très ergonomiques. Dans tout cet éventail de produits il faut avoir des critères pour faire un bon choix technologique.

Ce que doit proposer un bon gestionnaire de règles

Les outils de management

- **Un lieu de stockage pour règles métier** : cet endroit fournit un lieu central pour stocker les règles métier et les définitions métier des données qui sont utilisées (vocabulaire). Ce dépôt peut être persistant dans des fichiers ou bases de données.
- **Archivage (logging)** : la création, la modification et la suppression de règles métier et des propriétés peuvent être archivées dans un fichier historique.
- **Système de verrou** : ceci permettra à plusieurs utilisateurs d'accéder et de modifier les règles. Ce système de verrou permettra d'empêcher à plusieurs utilisateurs de modifier un même élément du repository simultanément.
- **La gestion des permissions** : ce mécanisme permettra de contrôler l'accès au repository, en définissant des droits par utilisateur ou groupe d'utilisateurs. Ce système

pourra être interfacé avec des systèmes externes d'authentification et d'identification comme LDAP, JAAS, UNIX, Domaine Windows, Kerberos, Keystore, etc.

- **Versionnement** : les règles peuvent avoir des versions permettant ainsi une suivie des changements de politique.
  Comme exemple, prenons la règles suivante : "A partir de février 2004, tous les nouveaux pensionnés ont droit à une majoration de pension minimum de 10 %". Dans cet exemple, on aura deux versions d'une même règle : la règle des calculs de droit minimum de pension ne sera pas la même suivant qu'on applique des règles sur des instances datant avant ou après février 2004.
  Le service de gestion de version permet de garder des versions de règles et de les restituer à tout moment.
  Quand on introduit une nouvelle règle, elle est immédiatement introduite par défaut comme la version active, elle peut être éditée, supprimée, ou déplacée. Par défaut, toutes les autres versions ne peuvent pas être modifiées.

Les outils de suivi (profiling)

Le "profiler" doit permettre d'avoir un résumé des statistiques de toutes les règles dans une session donnée :

- Le nombre de fois que chaque règle s'est exécutée.
- Le temps total ou moyen d'exécution d'une règle.
- Etc.

Les outils de debuggage et de traçage

Il est utile d'avoir un mécanisme d'exécution étape par étape dans un moteur de règle. Ces étapes peuvent être règle par règle ou action par action. A chaque étape on doit être en mesure de savoir ce qu'on a dans l'agenda et dans la working memory.

Recherche par requête

Le gestionnaire de règles doit permettre de localiser des règles par requête. En effet si on a plus de 1000 règles, ce mécanisme peut être fort utile et pourra permettre à l'utilisateur de :

- Rechercher et visualiser des règles qui seront affectées par un changement de politique.
- Rechercher et visualiser des règles qui seront affectées par un changement dans le model objet métier.
- Faire des changements globaux sur un ensemble de règles.

Un éditeur de règles

Dans le gestionnaire de règles, ces dernières peuvent être écrites soit en langage naturel ou en une syntaxe WYSIWYG qui, graphiquement, propose la modélisation de toutes les étapes du processus de l'écriture des règles.

2.7 Comparaison de quelques systèmes de gestion de règles métier

Le tableau ci-dessus fait un éventaire (non exhaustif) des SGRM qui existent. Les SGRM commerciaux les plus utilisés dans l'industrie sont JRules [20], Blaze [22, 24], Haley [25] et Jess [8]. Dans les logiciels libres, le moteur qui est de loin le plus utilisé est JBoss Rules qui s'appelait initialement Drools[21].

De manière absolue, nous ne pouvons pas dire qu'un Système de Gestion de Règles Métier est mieux qu'un autre car cela dépend énormément des critères que l'on prend. Par exemple si le principal critère est la rapidité, Jess fait parti des meilleurs (voir le benchmark ci-dessous). Cependant au niveau gestion des règles métier JRules, Blaze et Haley sont de loin mieux que Jess car ils disposent d'un éditeur en langage naturel, de requêtes et de bien d'autres outils qui permettent aux experts métier de mettre en place eux-mêmes leur logique métier.

2.7.1 Banc d'essais de quelques moteurs de règles

Pour mesurer et comparer la performance des moteurs de règles, 5 benchmarks ont été créés, tirés des travaux de Brant, Timothy Grose et Daniel P. Miranker [26]. De ces benchmarks nous en avons utilisé 2 que sont *"Miss manners"* et *"Waltz"* pour faire nos propres comparaisons. Ces deux benchmarks permettent de faire évoluer le nombre de données que manipulent les règles. Les tests ont été exécutés sur la configuration suivante :

Miss manners

"Miss manners" est un programme qui essaie de résoudre le problème de la répartition des invités autour d'une table lors d'un dîner. Cette répartition doit se faire en se basant sur le sexe et les hobbies des invités. Les sexes doivent être alternés et chaque voisin doit avoir au moins un hobbie en commun avec son voisin. Manners est un petit problème avec seulement 8 règles. La complexité du programme augmente en fonction du nombre d'invités et de places.
Nous allons maintenant voir les résultats de nos tests avec la Figure 23 et le tableau 3. *"n/a"* dans les tableaux veut dire que le test correspondant n'a pas pu se terminer dans

SGRM	**Type de licence**	**Editeur en langage naturel**	**Chaînage avant**	**Chaînage arrière**	**Divers outils (versioning, verrou, etc)**
Acquire	commercial	oui	oui	non	non
Blaze advisor	commercial	oui	oui	non	oui
Corticon	commercial	oui	oui	non	oui
Euler	open source	non	non	oui	non
Exsys	commercial	oui	oui	oui	oui
Gensym	commercial	oui	oui	oui	oui
Haley	commercial	oui	oui	oui	oui
InfoSapien	open source	non	oui	non	non
JBoss Rules (Drools)	open source	oui	oui	non	oui
Jena	open source	non	oui	non	oui
Jess	commercial	non	oui	oui	oui
JLisa	open source	non	oui	non	non
JRules, Rules	commercial	oui	oui	déprécié	oui
Mandarax	open source	non	non	oui	non
Nxbre	open source	non	oui	non	non
OpenRules	open source	non	non	oui	non
OpenLexicon	open source	oui	oui	non	oui
OPSJ, OPS/R2, CLIPS/R2	commercial	oui	oui	oui	oui
PegaRules	commercial	oui	oui	oui	oui
Zilonis	open source	non	oui	non	non

Tab. 1 – Tableau comparatif de quelques Systèmes de Gestion de Règles Métier

Système d'exploitation	Microsoft Windows XP SP2
Processeur	Intel Pentium IV
Vitesse du processeur	2 GHz
RAM	2 Go
Version J2SE	1.4

Tab. 2 – Caractéristique du système utilisé pour l'exécution des benchmarks

Nombre invités	Minhobbies	Maxhobbies	Drools (ms)	Jess (ms)
10	2	5	32	20
100	2	5	172	47
1000	2	5	6985	203
10000	2	5	n/a	672

Tab. 3 – Manners pour Drools et Jess

un temps inférieur à 1 heure ou encore la mémoire allouée pour le test a été dépassée occasionnant un plantage de la JVM.

Les Tableaux 4 et 5 et la Figure 24 montrent les tests de performance, de JRules (en mode hash ou non), Blaze Advisor, OPSJ, Jess, CLIPS/R2 et CLIPS, pour Miss manners dans le cas de 128 invités et places dans diverses plateformes. Il faut noter que dans le Tableau 5 les règles de Miss manners ont été optimisées en les réorganisant.

Waltz

Waltz est un système expert qui analyse des lignes dans un système de dessin, ayant un repérage bi-dimensionnels, et les étiquette comme si elles étaient des nœud dans un système de dessin avec un repérage tri-dimensionnels. Comme Manners, Waltz a un nombre de règles fixes (il y en a 33). Nous utilisons un générateur de données pour les tests. Cette génération de données se base sur le nombre de régions que l'on veut donner au système de dessin, sachant qu'une seule région est pré-définie pour pouvoir être la base de dessin de 72 lignes. Les conditions des règles dans Waltz sont plus complexes que dans Manners. Nous pouvons voir les résultats de nos tests avec la Figure 25 et le tableau 6 (dans ce tableau le chiffre après Waltz représente le nombre de régions). Ici aussi *"n/a"* dans les tableaux veut dire que le test correspondant n'a pas pu se terminer dans un temps inférieur à 1 heure ou encore la mémoire allouée pour le test a été dépassée occasionnant un plantage de la JVM.

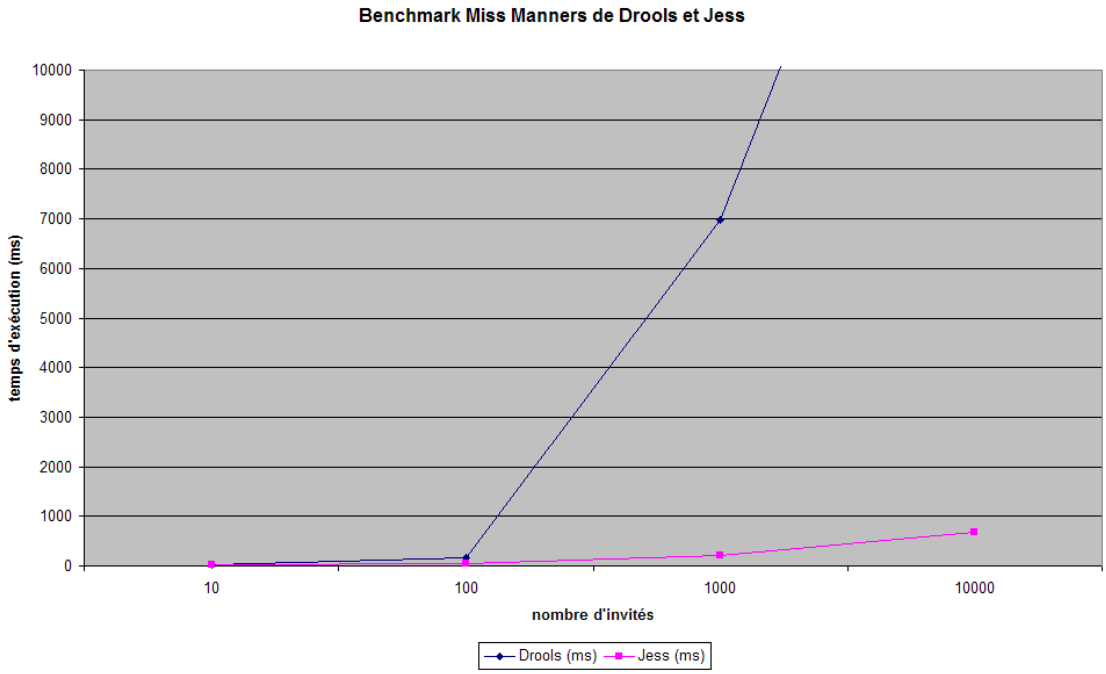

FIG. 23 – Benchmark Miss manners de Drools et Jess

Benchmark	Manners 128 Standard				
Plateforme	Windows XP	Mac 1GHz G4	Dual SPARC	Dual Win Svr	2xMac G5
Vitesse CPU	1.9 GHz	1.0 GHz	1.0 GHz	1.0 GHz	2.0 GHz
JRules 5.0.1 - no hash	89,00	131,00	73,67	73,68	56,80
JRules 5.0.1 - with hash	3,70	8,70	4,66	9,60	3,45
Advisor 5.1	n/a	n/a	n/a	n/a	n/a
OPSJ 6.0	7,93	18,35	9,04	19,58	
Jess 6.1p6,p7	66,50	164,00	73,20		
CLIPS/R2	11,50				
CLIPS 6.21	120,30				

TAB. 4 – Manners 128 standard pour quelques moteur commerciaux

Benchmark	Manners 128 Optimisé				
Plateforme	Win XP	Mac 1GH G4	Dual SPARC	Dual Win Svr	2xMac G5
Vitesse CPU	1.9 GHz	1.0 GHz	1.0 GHz	1.0 GHz	2.0 GHz
JRules 5.0.1 - no hash	32,00	54,00	26,14	47,96	21,60
JRules 5.0.1 - with hash	1,70	8,30	2,83	10,70	3,59
Advisor 5.1	32,83	85,78			
OPSJ 6.0	0,52	1,42	0,75	5,45	
Jess 6.1p6,p7	3,00	7,00	2,00		
CLIPS/R2	7,30				
CLIPS 6.21					

Tab. 5 – Manners 128 optimisé pour quelques moteurs commerciaux

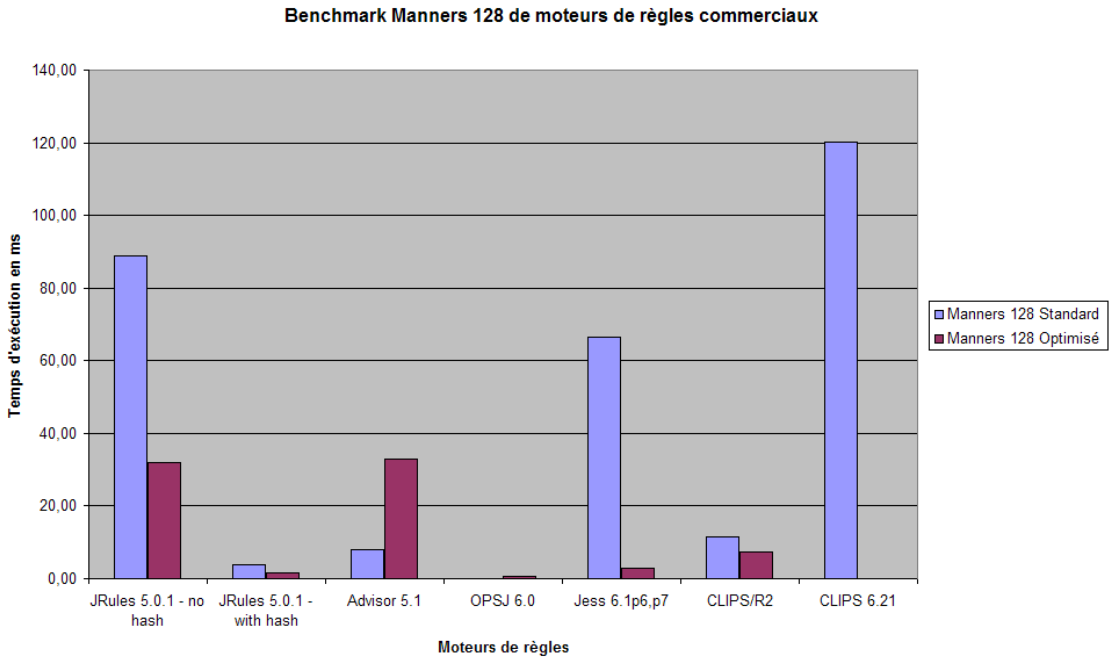

Fig. 24 – Benchmark Miss manners de quelques moteurs de règles commerciaux

	Drools (ms)	Jess (ms)
Waltz0	234	78
Waltz12	93	297
Waltz25	125	453
Waltz37	157	500
Waltz50	n/a	594

TAB. 6 – Waltz50 pour moteurs commerciaux

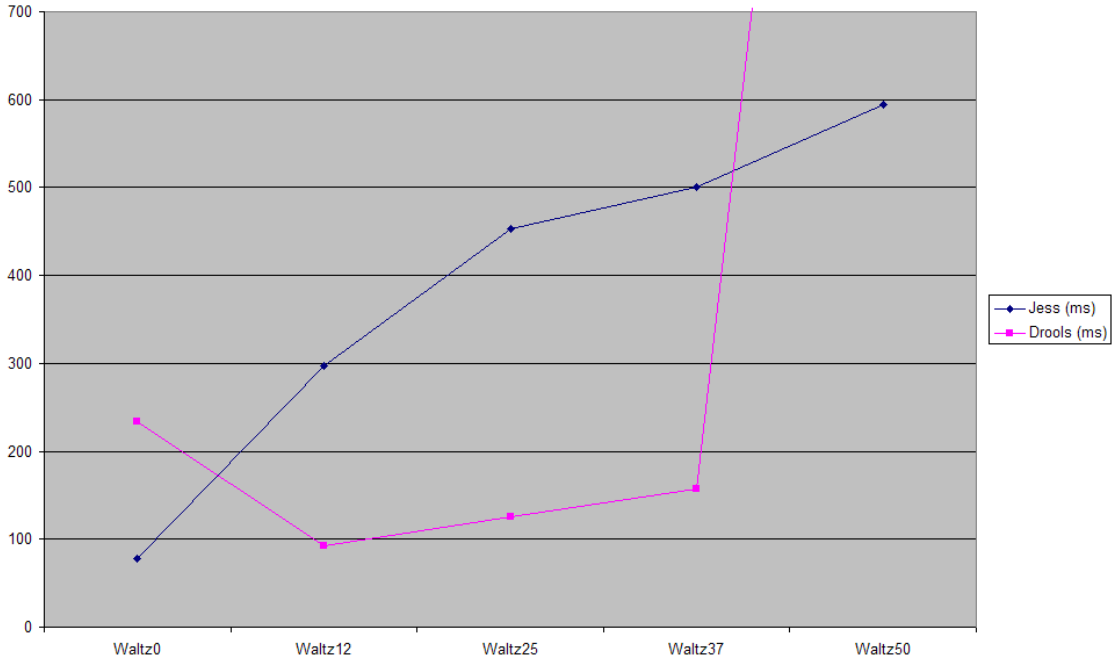

FIG. 25 – Benchmark Waltz de Jess et Drools

Benchmark	Waltz 50 (14066 règles exécutées)				
Plateforme	Win XP	Mac 1GHz G4	Dual SPARC	Dual WinSrv	2xMac G5
Vitesse CPU	1.9 GHz	1.0 GHz	2.0 GHz	1.0 GHz	2.0 GHz
JRules 5.0.1 - no hash	512,00	940,00	459,60	445,50	463,50
JRules 5.0.1 - with hash	256,00	285,00	189,30	193,20	197,50
Advisor 5.1	n/a	n/a			
OPSJ 6.0	1,69	8,00	2,33	2,16	1,51
Jess 6.1p6,p7	18,00	36,00	18,00		
CLIPS/R2					
CLIPS 6.21					

Tab. 7 – Waltz50 pour moteurs commerciaux

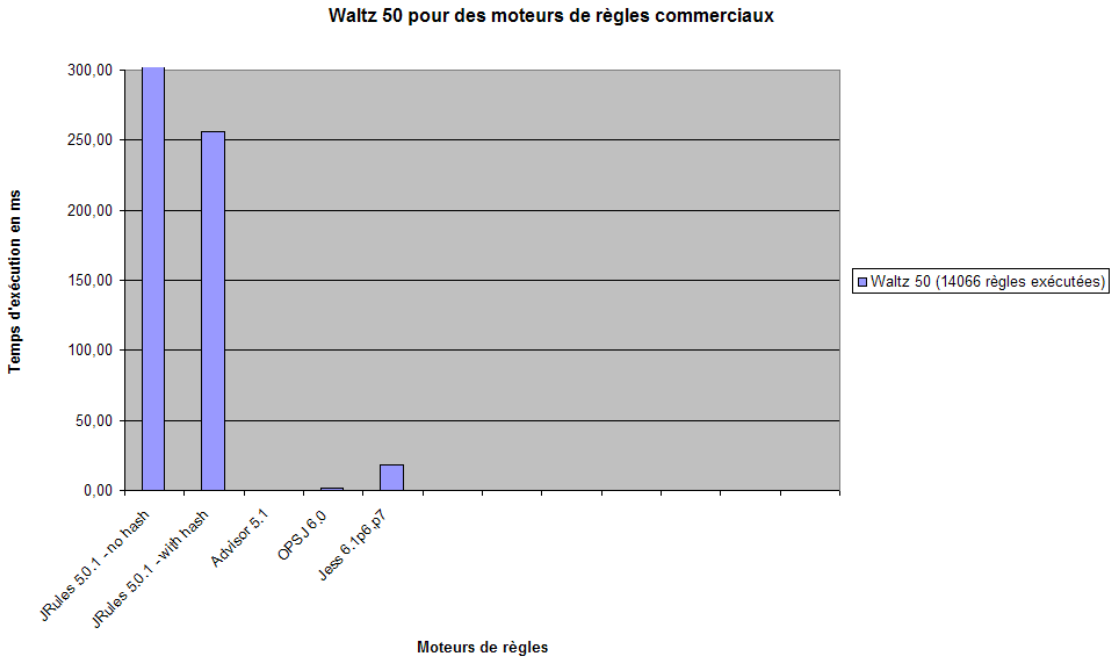

Fig. 26 – Benchmark Waltz de quelques moteurs de règles commerciaux

2.7.2 Etude et analyse des benchmarks

Avant de procéder à l'étude et à l'analyse, nous devons noter que dans le cas des Figures 23 et 25, la version de JESS utilisée est la *70a5* de 2005 et celle de Drools (devenu JBoss Rules) est la *2.0-beta-21* de 2005 également. Plusieurs versions de ces moteurs sont sorties depuis lors avec beaucoup d'avancées et d'améliorations en terme de performance. Une mise à jour des benchmarks est une tâche assez laborieuse car entre temps les formalismes utilisés dans chacun de ces moteurs pour représenter les règles ont évolués et cela nécessiterait de réécrire toutes les règles utilisées (nous reviendrons sur ce problème de non standardisation du formalisme de règle métier plus tard).
Dans le cas du benchmark *"Miss manners"*, comme le montrent la Figure 23 et le Tableau 3 nous pouvons voir que Jess va beaucoup plus vite que Drools, d'ailleurs à 10000 invités et places, ce dernier met plus d'une heure pour se terminer. Cette dominance de Jess s'explique par le fait qu'il soit plus optimisé pour RETE, d'ailleurs Jess est donné pour être l'implémentation de référence de RETE. Aussi bien que Jess, Drools utilise aussi une implémentation de l'algorithme RETE (appelé RETE-OO). Cependant les optimisations de Jess font qu'il va plus vite en cas de gros volume de données. En plus des optimisations "classiques" de RETE que sont le partage de nœuds à une entrée et de nœuds à deux entrées, Jess utilise une structure de données de type *"hashtable"* assez sophistiquée pour représenter les deux mémoires dans chaque nœud de jointure (voir 2.5.2). Jess utilise aussi plusieurs sortes de nœuds qui spécialisent la recherche de motif. Jess utilise également des nœuds spéciaux pour gérer certain éléments de conditions tels que "not" et "test". L'optimisation de la performance en utilisant un système de *"hash"* n'est pas propre à Jess. Comme on peut le voir sur la Figure 24 et le Tableau 4, JRules sur la plateforme windows XP passe de *89,00* en mode sans *"hash"*, à *3,70* en mode avec *"hash"*. Nous pouvons également remarquer sur ce même tableau (et sur les autres d'ailleurs) que OPSJ va plus vite que les autres. Nous savons que OPSJ est à base de RETE aussi mais, du fait de son caractère commercial, nous n'avons pas pu avoir plus d'informations sur la nature des optimisations pour le rendre plus rapide que les autres moteurs.
Dans le cas du benchmark *"Waltz"*, comme le montre la Figure 25 et le Tableau 6 Drools à un moment donné va plus vite que Jess contrairement à *"Miss manners"* mais à la fin se fait rattraper par ce dernier (à la fin Drools devient tellement lent et gourmand en mémoire que la JVM plante). Ceci s'explique par le fait que *"Waltz"* met l'accent sur la complexité des conditions des règles alors que la force de JESS est son optimisation de RETE et est plus efficace quand il y a beaucoup de données dans la WM.
Faire des tests de performances sur plusieurs moteurs de règles est complexe et pourrait manquer de rigueur car, en l'absence d'un format standard pour règles métier, les données des jeux de tests ne sont pas les mêmes et donc il n'y pas pas un référentiel unique et la manière d'écrire les règles impacte beaucoup sur le temps d'exécution des moteurs de règles.

2.8 Classification des règles

Les principaux types de règles que l'on rencontre sont les règles de dérivation, de contraintes d'intégrité, de production et de transformation. Suivant le niveau ou le point de vue où l'on se situe, ces types de règles peuvent devenir d'autres sous-types plus spécifiques. Comme nous le verrons au chapitre sur les modèles, dans l'ingénierie dirigé par les modèles, la couche qui est dédiée aux experts métier est la couche CIM (Computation Independent Model) alors que les couches PIM (Platform Independent Model) et PSM (Platform Specific Model) sont dédiées aux experts technique. Comme nous pouvons le voir sur la Figure 27 [27], les concepts de règles de dérivation, de contraintes d'intégrité et de réaction on plus de sens pour les experts métier et sont sur la couche CIM du MDA alors que les règles de transformation et de production ont plus de sens pour des experts technique ou système et se situent sur les couches PIM/PSM.

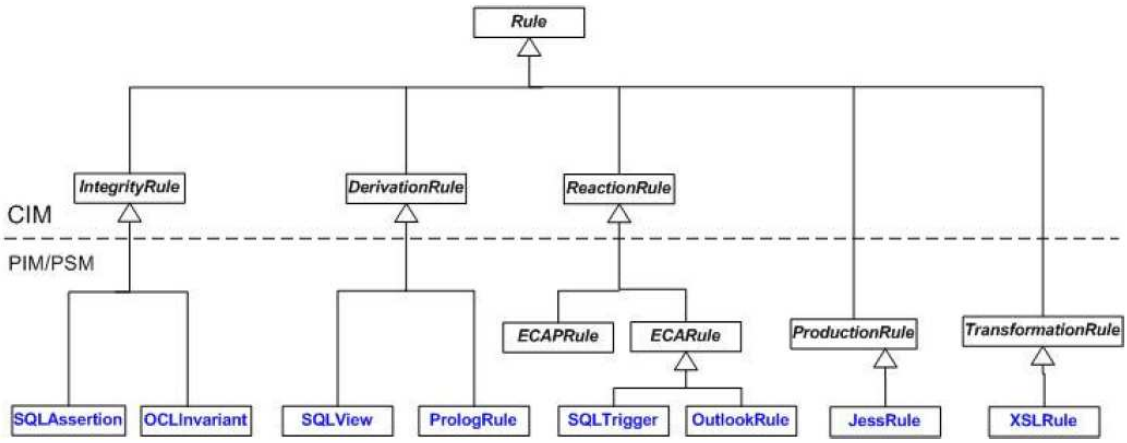

Fig. 27 – Classification des règles

2.8.1 Contraintes d'intégrité ou règles d'intégrité

Les règles d'intégrité ou contraintes d'intégrité sont des formulations logiques (dans un langage logique tel que les logiques de prédicats ou les logiques temporelles). Les contraintes d'intégrité sont des assertions qui assurent la cohérence de l'état d'un système dynamique. Les langages les plus connus pour créer des règles de contraintes sont SQL et OCL. les règles d'intégrité peuvent être mises en place par les règles Even-Condition-Action (ECA) [28], où la partie événement correspond à des changements qui pourraient rendre incohérent l'état du système et où les actions peuvent être des alertes ou des actions de réparation.

2.8.2 Règles de dérivation

Les règles de dérivation sont constituées d'une ou plusieurs conditions et d'une conclusion qui sont toutes les deux des formulations logiques. Pour certains types de règles de dérivation, comme les clauses de Horn bien définies [29], les types des conditions et des actions sont restreints. Voici un exemple de règle de dérivation : *Une voiture est disponible pour une location Si elle n'est pas assignée à une location et qu'elle ne requiert pas d'entretien.*
Les règles de dérivation doivent être sémantiquement distinguées des simples implications. Une implication est une expression d'une formule d'un langage logique, (tel que les logiques de prédicats classiques ou OCL), contenant une valeur booléenne. Alors qu'une règle de dérivation est une méta-expression logique qui ne possède pas de valeur booléenne mais utilise des expressions pour générer une connaissance dérivée.

2.8.3 Règles de réaction

Les règles de réaction sont constituées d'un événement déclencheur, d'une condition, d'une action déclenchée et d'une éventuelle post-condition. Les règles de réaction ne sont utilisables que dans un contexte d'exécution en chaînage avant car leur sémantique est la suivante : observer/vérifier un événement/condition et effectuer une action si et quand un événement/condition est observé/vérifiée.
On distingue deux types de règles de réaction : celles sans post-condition qui sont appelées Event-Condition-Action (ECA) et celles avec post-condition qui sont appelées ECAP.

Règles Event-Condition-Action (ECA)

Les règles ECA ont la forme suivante :

quand un événement
si condition
faire action

Les règles ECA ont été mises en exergue par les recherches dans le domaine des bases de données. Ces règles sont appelées *"triggers"* en SQL. Une utilisation possible des règles ECA peut être la gestion des événements de manière automatique dans une application. Une application pratique des règles ECA est l'utilisation de règles dans Microsoft Outlook qui permet de créer des règles de gestion de mails par rapport à des événements reçus ou envoyés. Il existe également des moteurs de règes ECA comme BizPulse de Savvion.

Règles Event-Condition-Action-Postcondition (ECAP)

Les règles ECAP étendent les règles ECA en y ajoutant une post-condition qui accompagne les déclencheurs actions. Les règles ECAP permettent de spécifier les effets d'une action *"trigger"* sur l'état d'un système de manière déclarative, au lieu de le faire de manière procédurale à la manière de SQL. Voyons maintenant un exemple de règle ECAP :

Sur *la reception d'une facture nécessitant un paiement de* $?x$ *EURO*
Si *la facture est correcte et le solde du compte est de* $?y$ ***Et*** $?y$ *est supérieur à* $?x$
Alors *décaisser un paiement de* $?x$ *EURO*
resultant sur un solde de $?y-?x$

Dans la règle de l'exemple, il y a une différence assez subtile entre la post-condition :

$solde = solde@pre-?x$

et l'opération correspondante :

Mettre à jour compte
Faire solde $:=$ $solde-?x$
Où NumeroDeCompte = ...

Cependant il est facile de voir que la première expression est déclarative alors que la deuxième non.

Dans le cas de règle ECA et ECAP, il peut arriver qu'après l'exécution des actions, que cela déclenche l'exécution d'autres règles. Afin de ne pas avoir ce genre de comportement, on parle des Independent-ECA (IECA). Les règles IECA sont identiques aux règles ECA sauf que dans ce cas les actions réalisées pour une règle n'influent en rien sur les conditions de n'importe quelle autre règle. Cela veut dire que les termes qui sont utilisés dans les conditions d'une règle sont strictement séparés de ceux utilisés dans les actions d'une autre règle. Les IECA sont incluses dans les ECA.

2.8.4 Règles de production

Comme nous l'avons déjà dit dans dans la partie 2.5.2, une règle de production est constituée d'un condition et d'une action. Les règles de production ont commencé à être largement utilisées dans les années 1980 avec les systèmes experts. Les règles de production ne font pas explicitement références aux événements contrairement aux règles ECA. Cependant, ces événements peuvent être simulés en injectant les faits dans la working memory. Dans ce sens, les règles de production peuvent implémenter les règles de réaction. Les règles de production peuvent également implémenter les règles de dérivation. En effet une règle de dérivation peut être simulée par une règle de production de la forme *Si-Condition-Alors-asserter-Action* en utilisant l'action spéciale *asserter* qui change l'état du système

en rajoutant de nouveaux faits à ceux déjà existants.
Comme nous l'avons déjà dit, les plateformes de règles de production sont les plus utilisées dans l'industrie des règles métier.

2.8.5 Règles de transformation

Les règles de transformation sont des règles de réécriture. Les règles de transformation sont constituées de 3 parties : une partie pour les données à transformer, une pour les condition, et enfin une dernière pour le résultat de la transformation qui correspond aux actions. Par exemple on peut avoir une règle de transformation qui transforme un modèle A en un modèle B.

Dans ce document, nous nous concentrons sur les règles de production, car ce sont elles qui implémentent les règles métier.
Dans toute la suite de ce document lorsqu'il sera fait mention de règle, il faudra entendre par là règle de production.

2.9 Conclusion

Dans ce chapitre nous avons rappelé l'approche utilisée traditionnellement pour concevoir et mettre en œuvre une application. Nous avons vu que cette approche est rigide et offre peu de possibilités de manœuvre aux experts fonctionnel qui ne sont pas des informaticiens. Nous avons par la suite présenté l'approche par règle métier et comment elle permet de séparer le "quoi" du "comment" en permettant de faire collaborer les experts métier et système. Nous avons également parlé des Systèmes de Gestion de Règle Métier qui permettent de faciliter la mise en place d'applications orientées règles métier.
Nous avons fini par présenter, étudier et analyser des tests de performance de quelques moteurs de règles et nous avons vu que suivant chaque optimisation propre à chacun d'eux, le résultat n'est pas le même. Cependant ces benchmarks peuvent être faussés par le fait qu'il n'existe pas de formalisme standardisé pour représenter les règles métier. Ainsi il faut réécrire chaque règle dans le formalisme propre au moteur de règle testé. Le référentiel de règles n'étant pas unique, cela fausse l'analyse, bien que l'ordre de performance reste sensiblement pareil.
Dans le prochain chapitre nous allons parler de formalisme de règles métier, des travaux qui sont en cours de réalisation ainsi que de nos travaux sur ce sujet.

Chapitre 3

Formalisme de règle métier

Sommaire

3.1 Introduction

Dans le chapitre précédent, qui portait sur l'approche par règles métier, nous avons exposé son avantage qui était de séparer la logique système de la logique métier en permettant aux experts métier, qui ne sont pas des informaticiens, de pouvoir eux même modifier le comportement de leurs applications, c'est à dire le métier. L'approche par règles métier offre des avantages réels, en mettant les experts métier au cœur du cycle de développement d'une application, car finalement, qui mieux qu'eux connaît les exigences métier de l'application.
L'approche par règles métier est de plus en plus utilisée, de plus en plus de Systèmes de Gestion de Règles Métier existent, ainsi que de plus en plus de moteurs de règles aussi bien commerciaux que gratuits et/ou libres. Ces avancées et la communauté grandissante ajoutent de nouveaux besoins dans le domaine de l'approche par règles métier, le premier étant la flexibilité. En effet le plus gros problème dans la mise en œuvre de l'approche par règles métier est qu'il n'existe pas de formalisme unique et standardisé pour le stockage des règles. Imaginons qu'une entreprise qui suit l'approche utilise Drools et qu'un jour, face à une limitation de ce dernier, doit changer de moteur de règles pour passer à JRules. Imaginons aussi que cette entreprise ait plus de 3000 règles métier. Drools utilise un formalisme nommé DRL pour Drool Rule Language, que nous appellerons r_1, alors que JRules utilise IRL, que nous appellerons r_2, pour Ilog Rule Langage. Pour la migration de Drools vers JRules, nous avons besoin d'un système de transformation T , tel que $r_2 = T(r_1)$, qui transforme un ensemble de règles de Drools vers JRules. Pour cela, nous avons besoin de répondre aux questions suivantes :

- Est-ce qu'une telle transformation T existe en pratique ?
- En combinant d'autres transformations, peut-on obtenir T ? c'est-à-dire existent-ils des transformations T_1 et T_2 tel que : $r_2 = T_1(r'_1)$, $r'_1 = T_2(r_1)$?
- Nous savons qu'aucune transformation, directe ou indirecte, n'existe, pouvons-nous la créer ?

Actuellement il existe des dizaines de moteurs de règles sans aucun système de transformation pour une paire d'entre eux. En plus la majeure partie des moteurs de règles sont commerciaux, donc avec un formalisme de règles propriétaire et non accessible pour créer un système de mapping. Ainsi dans notre exemple de Drools vers JRules, il faudra réécrire toutes les 3000 règles de DRL vers IRL. Imaginons un cas plus complexe encore, où nous devons intégrer plusieurs systèmes avec un total de n moteurs de règles avec chacun d'eux pouvant échanger des règles avec l'autre. Donc nous avons n moteurs de règles avec respectivement r_1, r_2, ..., r_n comme langage de règles. Alors dans ce cas il nous faudrait $n(n-1)$ transformations car chaque formalisme d'un moteur devrait pouvoir être transformation vers un autre. Une telle complexité n'est pas envisageable et une solution est d'avoir un formalisme standardisé qui sera natif dans les moteurs de règles ou au pire fournir une transformation de ce formalisme vers le formalisme propre à chaque moteur. En procédant ainsi, la complexité passe de $n(n-1)$ à $2n$. Il faut noter cependant qu'il existe une bibliothèque (API) en java qui permet d'interagir de manière transparente avec tout moteur de

règles l'implémentant. Nous verrons plus en détail cette bibliothèque (la jsr94) au chapitre 10 dans la partie implémentation.
Depuis quelques années déjà il existe des travaux pour la mise en place d'un formalisme standard de règles métier sans trop de succès. Nous allons voir quelques un de ces travaux en l'occurrence CommonRules de IBM, RuleML, RIF, PRR et SBVR. Nous allons d'abord introduire le modèle de règles et les fondements d'un langage de règle. Nous finirons par présenter nos travaux sur un formalisme de règles métier indépendant de tout moteur de règles.

3.2 Concept d'un méta-modèle pour règle métier

La méta-modélisation consiste à donner les spécifications d'un langage de modélisation grâce à un autre langage de modélisation, appelé méta-langage. Ainsi un méta-langage est un langage qui prend pour objet un autre langage et qui le formalise. Comme tout langage, il est constitué d'une syntaxe et d'une sémantique. La syntaxe spécifie les sujets (concepts, entités) et les verbes (liens entre ces concepts), alors que la sémantique donne une interprétation aux sujets et aux verbes.
Un méta-modèle est constitué de 2 parties :

- Le vocabulaire qui définit les différents concepts du problème et leur relation.
- La définition des règles métier que l'on considère.

3.2.1 Le vocabulaire

Le vocabulaire est la description de l'ensemble des concepts présents dans un problème et de leurs relations. L'élément de base du vocabulaire est le *Type de termes*, sur lequel les différentes relations vont être construites à l'aide de *Types de faits*
Un vocabulaire est constitué de termes et de variables, qui sont des entités atomiques, sur lesquels sont définis les *Types de termes* (élément de base du vocabulaire) et les *Types de faits* (qui permettent la construction des différentes relations).

Type de termes

Les types de termes correspondent aux concepts généraux du modèle (exemple : client, voiture). Ils correspondent en général aux classes dans la représentation objet du vocabulaire.
Les types de termes sont de deux types : métier et commun.
Type de termes métier :
Les types de termes métiers décrivent les concepts généraux qui sont spécifiques au domaine considéré. Par exemple pour un problème de location de voiture on peut avoir comme types de termes métier : Filiale, Client, Contrat, etc.
Type de termes communs :

A l'opposé des types de termes métier, les types de termes communs visent un ensemble plus large de vocabulaires issus de problèmes différents, ils ne sont pas spécifiques au problème considéré. Par exemple comme types de termes communs nous pouvons avoir Date, Temps, Kilométrage, etc.
Nous allons maintenant voir les types de faits qui sont des concepts d'un ordre supérieur, qui décrivent les différentes relations entres les types de termes.

3.2.2 Type de faits

Les types de faits comprennent un nombre variable de rôles sémantiques, chacun joué par un type de termes. Un **rôle sémantique** décrit le rôle syntaxique d'un type de terme dans la phrase.
Exemple :
"Une voiture peut être affectée à un contrat de location".
Le type de terme voiture a le rôle syntaxique "voiture affectée".
Il existe deux types de types de faits :

- Types de faits unaires : ils mettent en jeu un et un seul rôle sémantique donc un seul type de terme. Ils décrivent certaines propriétés que peut avoir un type de termes.
  Exemple :
  "un **client** peut appartenir à un programme de fidélisation".

- Association métier : elle permet d'établir une relation entre plusieurs rôles sémantiques joués par des types de termes.
  Exemple :
  "un **Client** peut passer un **Contrat** de location pour la **Période** donnée".
  Dans cet exemple, le type de termes **Client** joue le rôle sémantique "**Client** passant un **Contrat** de location ", le type de termes **Contrat** joue le rôle sémantique "**Contrat** de location passé".

Un type de fait est ainsi défini par les types de termes qu'il contient et leurs rôles sémantiques respectifs.
Il y a aussi les instances métier qui s'appuient sur les types de termes et les types de faits.
En langage naturel, il peut exister un ensemble de phrases sémantiquement équivalentes qui décrivent le même type de faits.

3.2.3 Instances Métier

Une instance métier est composée de types de termes et de types de faits.
Les instances métier spécialisent un concept général. Il existe deux types d'instances métier : les instances de termes et les instances de faits.
Les instances métier représentent les données d'entrée des problèmes. Elles peuvent aussi permettre d'exprimer la solution. Les instances métier sont donc créées soit par l'utilisateur

soit par le système [1].

Instances de termes

Une instance de termes est l'instanciation d'un type de termes, elle représente ainsi un concept particulier. C'est une variable à laquelle une valeur a été assignée.
Exemple :
"Le **Client** Mr. Anderson"

Instances de faits

Comme l'instance de termes, c'est l'instanciation d'un type de faits donc une phrase. Il faut remplacer tous les types de termes par les instances de termes.
Exemple :
"Le **Client** Mr. Anderson passe le **Contrat** de location 041 pour la **Période** du 13/03/05 au 20/04/04".
C'est une variable à laquelle une valeur a été affectée.

3.3 Règles métier

Les règles métier sont des déclarations structurées de haut niveau. Elles ont pour but de contraindre, contrôler ou influencer certains aspects d'un métier. De manière générale, on ne pense pas en terme de *règles métier*, mais en terme de *Principe Métier*. Un principe métier est un concept de plus haut niveau qui définit les buts à atteindre dans le problème que l'on considère.
Les règles sont des déclarations formelles se conformant à une grammaire précise pour pouvoir être interprétées et être utilisées par un ordinateur.
Les principes métier sont reliés aux règles métier qui les spécifient de manière formelle à travers un ensemble de règles.

3.3.1 Ensemble de règles

Un ensemble de règles permet d'implémenter les principes métier définis dans le problème considéré. Il existe deux types d'ensembles de règles :

- Les ensembles de règles procédurales.
- Les ensembles de règles déclaratives.

Un ensemble de règles est formé à partir des éléments suivants :

- Un ensemble de principes métier : il représente les objectifs ou contraintes qui sont assurés par l'ensemble de règles.

- Un ensemble de règles métier : ce sont les règles qui vont permettre d'implémenter les principes métiers.
- Un ensemble de variables d'entrée : il représente l'ensemble des données nécessaires pour l'application de l'ensemble des règles métiers.
- Un ensemble de variables de sortie : il représente l'ensemble des données résultat obtenues ou attendues après application de l'ensemble des règles métier.

3.3.2 Méta-langage de règles métier

Comme nous l'avons déjà exprimé, une règle métier peut se définir comme étant : "des déclarations qui définissent ou contraignent certains aspects du monde de l'entreprise. Ces déclarations sont atomiques, elles ne peuvent être séparées en plusieurs règles métier [10]". Techniquement, un ensemble de règles métier est un ensemble de ***if***, ***then*** et potentiellement (suivant le moteur de règles) ***else***.
Dans la sous-section suivante nous allons rappeler les structures des règles métier.

Structures des règles métier

Suivant le type de règle (procédurale ou déclarative), la structure et la sémantique de cette structure varient. Dans ce document nous nous focalisons sur les règles de production. Les règles sont de la forme "***if ...then ...else ...***" et sont constituées des éléments suivants :

- Déclarations des variables : permet de définir le contexte d'application de la règle métier.
- Valeur métier (Expression) : c'est une expression dont l'évaluation renvoie un entier correspondant à l'importance de la règle. Elle permet de définir un ordre dans l'exécution des règles. Cette valeur métier peut aussi être appelée salience ou priorité.
- "***If***" ou condition (expression) : cette partie est optionnelle dans une règle. Elle constitue la condition sous laquelle la règle peut être appliquée. Elle est constituée d'une expression qui est évaluée à vrai ou faux. Elle est aussi appelée "body".
- "***Then***" (Expression à effet de bord) : c'est le corps de la règle ou "head". Elle représente l'action à effectuer si la partie condition est évaluée à vrai. Dans le cadre des règles déclaratives cette partie doit être une expression sans effet de bord.
- "***Else***" (optionnelle et Expression à effet de bord). Elle n'est uniquement envisageable que pour les règles procédurales si elles contiennent une partie "If". Elle représente l'action à effectuer si la partie condition est évaluée à faux.

Ci-dessus nous venons de rappeler la structure d'une règle métiers, nous allons maintenant voir quelles sont les **expressions** pouvant être intégrées dans cette structure.

Expressions :
Les expressions sont classées en deux groupes suivant si elles influencent ou non l'état du

système. Il y a des expressions à effet de bord et des expressions sans effet de bord. Une expression est sans effet de bord si et seulement si elle ne modifie pas l'état du système. Certaines expressions modifient la nature du système lors de leur exécution. Elles sont dites à effet de bord et elles peuvent servir pour :

- Démarrer un nouveau processus.
- Exécuter un ensemble de règles métier.
- Activer ou désactiver un processus.
- Activer ou désactiver un ensemble de règles métier.
- Ajouter ou enlever une instance.
- Affecter une valeur à des variables.
- Donner ou retirer à certains objets des droits ou privilèges sur d'autres objets.

3.3.3 Formalisme des règles métier

Le formalisme des règles se doit de respecter un certain nombre de principes :

- Expression explicite : la formulation des règles métier a besoin d'une expression explicite, soit graphiquement, soit avec un langage formel.
- Une représention cohérente : étant donnée que le formalisme des règles métier devient de plus en plus utilisé dans des systèmes, il est nécessaire d'avoir une représentation cohérente et unique pour tous les types de règles métier.
- Extension évolutive.
- Nature déclarative : il faut noter qu'une règle métier est déclarative et non procédurale. Elle décrit un état désiré, possible qui est soit suggéré, soit requis soit prohibé.

3.4 Le modèle conceptuel des règles métier

Pour représenter les concepts et les structures d'un formalisme de règles métier, nous nous sommes basés sur le méta-modèle proposé par le Business Rule Group [1].

3.4.1 Formulation des règles métier

La procédure pour identifier une règle métier est souvent itérative et heuristique. Au début, les règles ne sont que formulations et politiques générales. Comme le dit Barbara Von Halle [7] "ces formulations sont parfois claires, parfois (peut être délibérément) ambiguës, et souvent contiennent plus d'une idée"

- Initialement, la tâche de l'analyste fonctionnel consiste à décomposer les formulations et politiques générales en règles atomiques, où ces dernières sont des formulations des termes, des faits, des dérivations ou des contraintes singuliers sur le métier. L'expert fonctionnel doit aussi évaluer la stabilité d'une règle. Par exemple voir si c'est un aspect fondamental du métier qui est appelé à changer dans un futur proche ou

lointain.

- Ensuite il faut identifier les formulations atomiques comme la définition d'un terme, d'un fait, d'une contrainte ou d'une dérivation. Les termes, les faits, et quelques contraintes peuvent directement être représentés par un modèle graphique. Les contraintes restantes et les dérivations doivent être translatées dans un autre formalisme. Ceci peut être une simple formulation utilisant un langage naturel ou encore un autre moyen utilisant un formalisme plus complexe, comme un langage de logique ou un outil graphique comme ORM (Object Role Modeling) [30, 31]. Quelle que soit la technologie utilisée pour représenter graphiquement le formalisme des règles (vue en table, en arbre, etc.), celle choisie pour l'implémentation (comment les règles sont réellement stockées) sera la même.

3.4.2 Les origines des règles métiers : le modèle

La Figure 28 montre la première partie du modèle conceptuel de règle métier [1] :

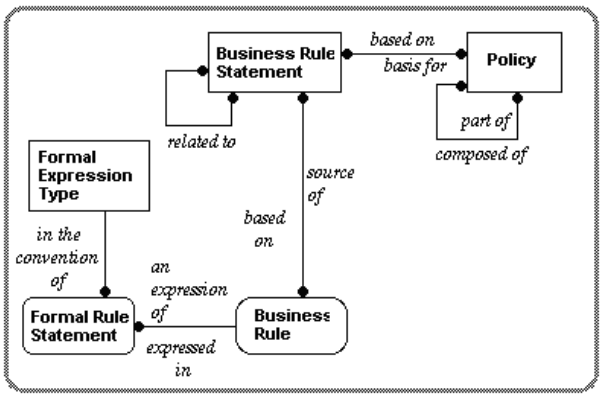

Fig. 28 – L'origine des règles métier

- *"Policy"* : c'est la politique générale d'une entreprise. Chaque *"policy"* est composée de *"policy"* plus détaillées. Un *"policy"* est à la base de une ou plusieurs déclarations de règle métier (*"Business Rule Statement"*).
  Exemple :
  "nous louons nos voitures dans un cadre toujours légal"

- *"Business Rule Statement"* : c'est une formulation sur laquelle se base un métier.
  Exemple :
  "les voitures doivent être vérifiées après chaque location et transfert entre filiales."
  "Si un phare ne marche pas, les ampoules doivent être remplacées. Si des pneus sont usés, ils doivent être remplacés."
  Donc en fin de compte un *"Business Rule Statement"* doit être la source d'une ou plusieurs *"Business Rule"*.

- *"Business Rule"* : comme un *"Business Rule Statement"* , une *"Business Rule"* est une formulation qui définit ou contraint quelques aspects d'un métier mais (contrairement à un *"Business Rule Statement"*), elle ne peut pas être scindée en d'autres *"Business Rule"*. En essayant de les réduire, on perd de l'information. Chaque *"Business Rule"* doit être basée sur une ou plusieurs *"Business Rule Statement"*.
  Exemple :
  "une voiture qui a fait plus de 5000 km depuis sa dernière révision technique doit être programmée pour une révision technique"
  Les règles métiers sont en place dans une entreprise bien avant que personne ne pense à les formaliser ou à les dessiner. Le principe des règles métier est une réalité sous-jacente dans une entreprise. Chaque *"Business Rule"* doit être exprimée à l'aide d'une ou plusieurs *"Formal Rule Statement"*.
- *"Formal Rule Statement" :* c'est l'expression d'une *"Business Rule"* dans un langage formel (Ex : IDEF1X, CASE*Method, ORM, Ross Notation ...)
  Exemple :
  Si voiture.kilometrage_courant > 5000 alors appeler programmation_service_technique(voiture.identifiant)

Le modèle complet de règles métier est présenté à la Figure 29.

3.5 CommonRules de IBM

L'objectif de CommonRules Interlegua [32, 33, 34], qui est un projet de IBM, est de proposer un langage standard de représentation de règles pour permettre leur échange, de manière transparente, sur plusieurs moteurs de règles. CommonRules définit un nouveau langage de règles inter-opérable basé sur XML appelé Business Rule Markup Language (BRML) [35].
Pour établir la sémantique du langage, les chercheurs de IBM ont capturé le noyau commun partagé par la plupart des systèmes de règles commerciaux, incluant SQL, Prolog et autres systèmes de programmation logique [36], les systèmes de règles de production (qui sont descendants de OPS5 [37]) et les systèmes de règles ECA. CommonRules fait abstraction du mode de chaînage (avant ou arrière).
CommonRules fournit des *"translateurs"* entre BRML et quelques langages de moteurs de règles, cependant le développeur peut écrire son propre *"translateur"*.

3.5.1 Vue d'ensemble technique

CommonRules est composé de 3 principaux composants fonctionnels : le langage BRML et ses *"translateurs"*, le transformateur de CommonRules CLP vers OLP et le moteur d'inférence.

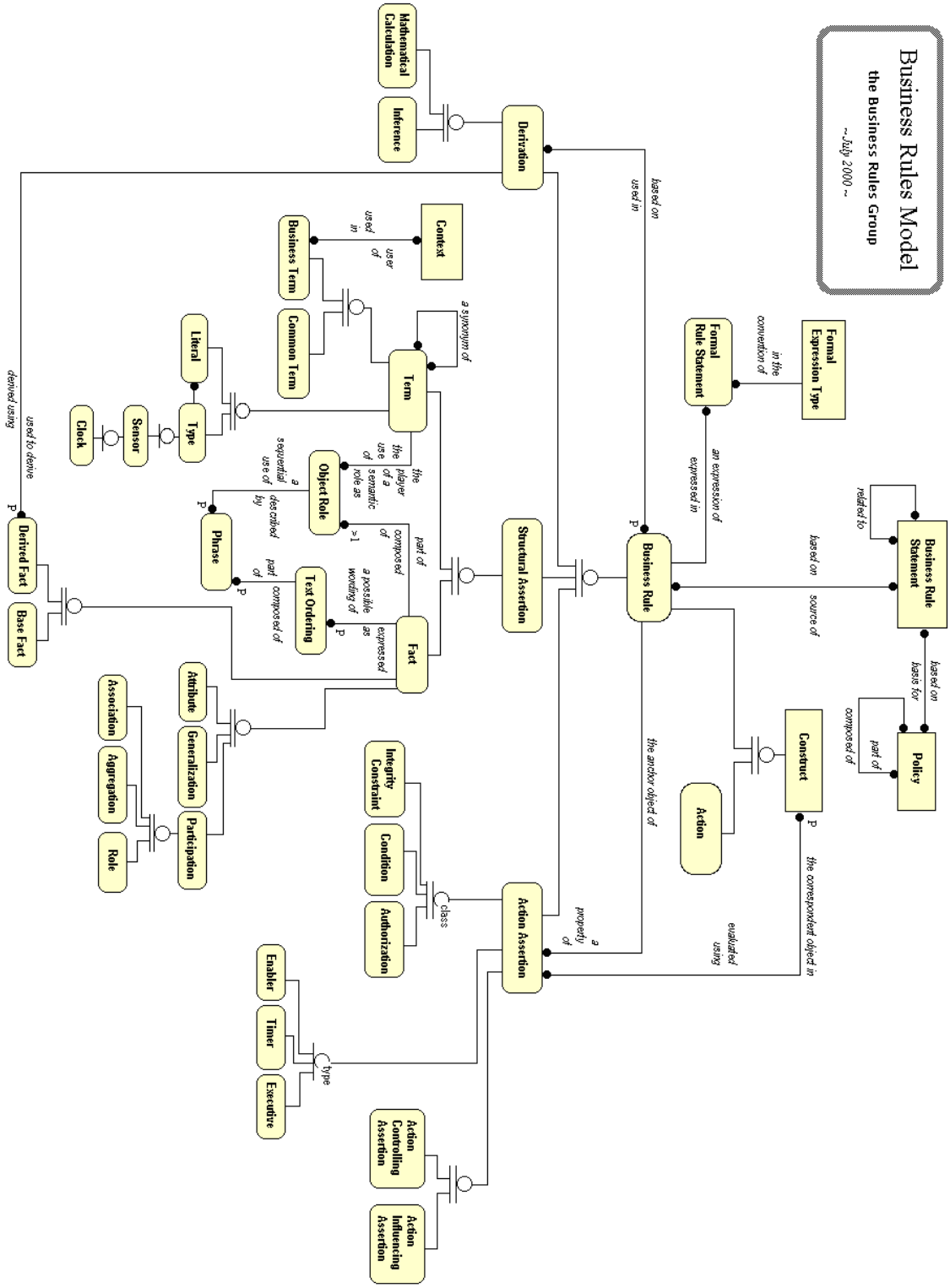

Fig. 29 – Le modèle complet des règles métier [1]

BRML et ses *"translateurs"*

Le but de ce composant est de permettre à plusieurs systèmes de règles de communiquer de manière hétérogène. BRML est un langage pivot de représentation de règles.
CommonRules a adopté une approche par pivot qui consiste à avoir un type d'objet intermédiaire appelé *CLPobj* puis d'avoir des translators qui vont parser ce dernier et générer des langages de règles cibles. Un *CLPobj* est un Courteous Logic Programs (CLP) qui, selon ses auteurs, est une extension plus expressive de la programmation logique ordinaire, [34] embarqué sous forme d'objet java. CommonRules a beaucoup été influencé par les travaux sur le partage de connaissance (Knowledge Sharing Effort) [38].
Les formats de langages de règles supportés sont les suivants :

- le format XML pour CLP.
- le format CLPfile pour CLP (en ASCII).
- le format KIF (Knowledge Information Format) qui est un langage de représentation de la connaissance (Knowledge Representation). KIF peut représenter de la logique classique mais ne dispose pas de moyens pour représenter directement la non-monotonicité (par exemple la negation-as-failure, la gestion des priorité) [39]. Le format KIF est en ASCII.
- le format XSB [40] qui est un système de programmation logique ordinaire (OLP) [41]. XSB fait du backward chaining comme Prolog et son format est en ASCII.
- le format Smodels [42] qui est aussi un système de programmation logique ordinaire (OLP) mais qui fait du forward chaining.

Les *"translateurs"* génèrent d'autres formats à partir d'un objet CLP et inversement comme le montre la Figure 30. Cependant il faut savoir que les XML et les DTD de la version 2.1 ne sont pas compatibles avec ceux définis dans CommonRules 1.1 dû fait que des paramètres du CLP aient changés.

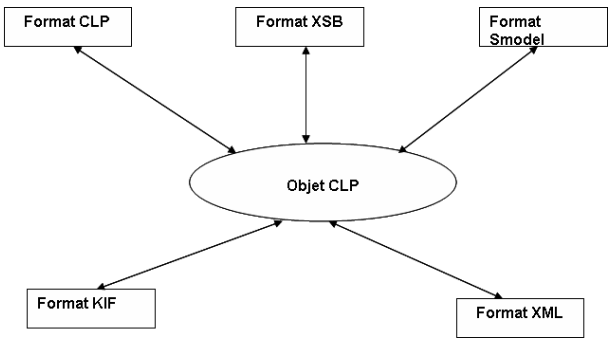

FIG. 30 – Translation de CLP vers d'autres formats

Le composant transformateur de CommonRules CLP vers OLP

Ce compilateur transforme du CLP (Courteous Logic Program) vers du OLP (Ordinary Logic Program).
CLP permet la "prioritisation" par défaut des règles ce qui fournit un niveau de spécification de haut niveau, qui facilite la vision de modularité. En particulier CLP gère les conflits, y compris ceux qui arrivent lors de la mise à jour du ruleset.
Syntaxiquement CLP étend OPL en ajoutant les caractéristiques suivantes :

- Chaque règle a un label optionnel. Ceci sera utilisé pour spécifier les informations pour la "prioritisation".
- Le prédicat "overrides" est utilisé pour formuler une "prioritisation". Par exemple l'instruction "overrides(lab1,lab2)" veut dire que toute règle ayant comme label "lab1" a une priorité supérieure à toute règle ayant comme label "lab2".
- La portée de ce qui est en conflit est spécifiée en utilisant l'élément d'exclusion mutuelle *mutex*.
- Chaque littéral peut supporter la négation.

Le composant moteur d'inférence

Ce composant fournit les mécanismes permettant de charger, d'exécuter les règles mais aussi de récupérer le résultat de l'exécution.
Le moteur d'inférence peut être utilisé aussi bien en ligne de commande qu'en API.

L'architecture de CommonRules

La Figure 31 montre une architecture simplifiée de CommonRules. Nous pouvons voir sur la figure qu'en entrée et en sortie, nous pouvons avoir comme formats du XML, XSB, KIF. Dans la partie bleue nous avons le moteur de règles qui utilise des fonctions externes, des objets java ou une connection à une base de données via JDBC.

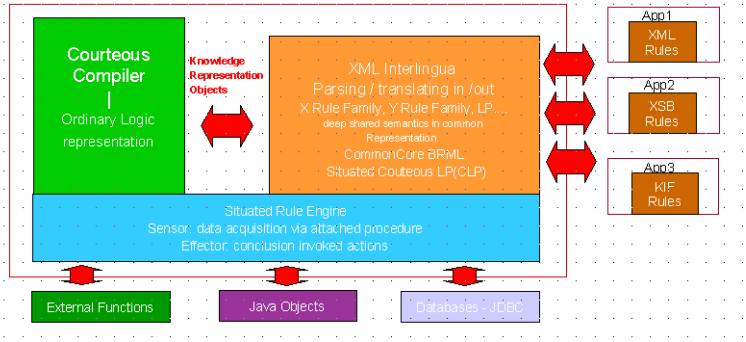

Fig. 31 – Vue simplifiée de l'architecture de CommonRules

3.5.2 Les tests de validation des règles dans CommonRules

Une règle doit être valide avant d'être injectée dans le monteur d'inférence. CommonRules a mis en place plusieurs mécanismes pour vérifier la validité d'une règle :

- Validation syntaxique : il faut que la syntaxe respecte le format de la grammaire du CLP de commonRules.
- Validation sémantique : il faut que les règles respectent les logiques basiques.
  Exemple :
  If a(?X, ?Y) and plusGrandQue(?X, ?Y) and plusPetitQue(?X, ?Z) and egale(?Y, ?Z) then resultat(?X, ?Y) ;
  on ne peut pas avoir $X > Y$ et $X < Z$ si $Y \doteq Z$ pour tout X, Y, Z.
- Validation de liaison ou d'intégrité : toute variable utilisée doit avoir été auparavant déclarée.
- Vérifier qu'il n'y a pas de cycles dans les conditions.
  Exemple :
  <rule1> If managerOf(?Name1, ?Name2) then employeeOf(?Name2, ?Name1) ;
  <rule2> If empolyeeOf(?Name2, ?Name1) then managerOf(?Name1, ?Name2) ;
  Dans cet exemple nous avons la première qui fait appel à la deuxième règle et cette dernière qui fait appel à la première.
- Validation des procédures attachées : il faut qu'une procédure utilisée, existe dans le modèle objet. C'est-à-dire que si on utilise une méthode d'une classe dans une règle, avant de charger la règle dans l'agenda, il faut vérifier que cette méthode existe bien dans la dite classe avec la bonne signature.

CommonRules est l'une des premières tentatives pour la standardisation d'un formalisme de règles métier, cependant les entreprises n'y ont pas beaucoup adhéré. la première version de CommonRules est sortie en juin 1999, cependant ce projet semble être clôturé car depuis 2002 il n'y a pas eu de nouvelle version et le site web n'est plus mise à jour.
CommonRules a été précurseur de beaucoup de nouveautés dans les moteurs de règles (gestions des conflits, utilisation de langage pivot) qui ont été réutilisées dans d'autres tentatives comme l'initiative RuleML qui, de par son architecture ressemble beaucoup à CommonRules.

3.6 L'initiative RuleML

L'initiative RuleML a vu le jour en Août 2000 lors de la conférence international sur l'intelligence artificielle PRICAI2000 [43]. RuleML a pour objectif de développer un formalisme standard de règles métier neutre et ouvert basé sur XML/RDF dans le but de permettre à des systèmes hétérogènes de s'échanger des règles. Le groupe de travail sur RuleML est composé d'académiciens et d'industriels. Les principes de RuleML [27, 44, 45] ont beaucoup servi pour d'autres systèmes de formalismes de règles. RuleML supporte aussi bien le chaînage avant qu'arrière.

3.6.1 Motivation de l'initiative

Le groupe de travail sur RuleML a été mis en place afin de répondre à un besoin urgent d'avoir un langage de règles pour le Web actuel et pour le Web Sémantique [27]. Ses prérogatives étaient aussi de mettre en place un standard qui permettrait aux utilisateurs de systèmes d'inférences et autres systèmes de gestion de connaissances de partager des règles. Le constat de l'époque était que les règles d'inférence étaient de plus en plus utilisées dans le commerce électronique et dans l'architecture du web sémantique.
RuleML offre une syntaxe XML et RDF pour formaliser des règles de représentation de la connaissance, interopérable sur la majeure partie des systèmes de règles commerciaux ou non. Le groupe de travail est composé de constructeurs de moteur de règles, de constructeurs de technologies web, de constructeurs d'outils XML/RDF, d'entreprises clientes, etc.

3.6.2 Les étapes initiales de RuleML

La première des choses que les concepteurs de RuleML ont effectuées a été de structurer la portée du futur langage de règles. Ceci dans le but de cibler les problématiques et d'identifier les tâches afin de proposer des balises et attributs pour le langage.
Voici les différentes étapes dans un ordre chronologique :

1. Recherche web : l'état de l'art du domaine.
2. Comparaison des différentes propositions : laquelle couvre quel type de règles ? Les chevauchements des différentes propositions ?
3. Transformation des DTD : peut-on passer d'une proposition de langage (en DTD) vers une autre en utilisant XSLT ?
4. Harmonisation du langage : si plusieurs balises ou attributs ont été utilisés pour la même partie d'une règle, laquelle garder pour RuleML ? Le langage de départ utilisait les balises suivantes :
   - *<imp>...</imp>* pour les règles de dérivation avec un attribut *direction* qui pouvait avoir comme valeur "*Forward| Backward | bidirectionnel*"
   - *<body>...<body>* pour la partie condition, *<head> ...</head>* pour la partie action et *<var>...</var>* pour les variables de règles.
5. Définir une syntaxe de règles : comment écrire les conjonctions et les disjonctions de prémisses ? Comment les combiner ?
6. Modules de règles : considérer les règles comme un module, qui est un ensemble ou un package de règles peuvent être regroupés par une balise explicite *<rulebase> ...</rulebase>*
7. Application des règles : comment faire appel aux règles ?
8. Expressivité des règles : que doit supporter le langage : la logique classique, la négation (laquelle : forte ou faible ?), la logique de première ordre, la non-monotonicité, la prise en compte des priorités dans les règles, etc ?

9. Sémantiques des règles : doit-on essayer de fixer quelques sémantiques déclaratives des règles ?
10. Règles et RDF.
11. Couplage avec l'ontologie web.
12. Validation des règles : quel outil XML utilisé pour valider une règles ?
13. Compilation des règles : réseau de RETE, arbre de décision, WAM code.
14. XML StyleSheet : comment faire les transformation de RuleML vers un autre langage de règle et vice-versa.
15. Règles sémiformelles : comment utiliser le langage naturel dans RuleML.
16. Documents de règles : au niveau documents XML entiers, comment mixer au mieux, bases de règles et textes ?

3.6.3 Syntaxe et sémantique modulaire de RuleML

RuleML couvre la hiérarchie de règles suivante : règles de réaction, règles de transformation, règles de dérivation, règles de fait, règles de requête et règles de contrainte d'intégrité. Comme le montre la Figure 32, dans la hiérarchie des règles de RuleML nous

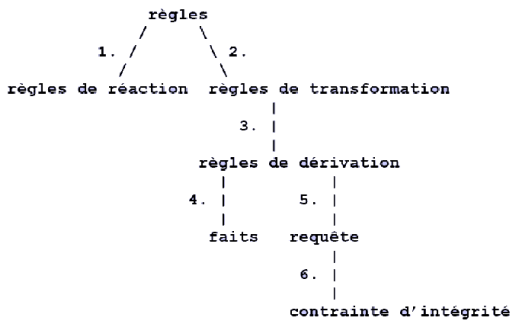

FIG. 32 – Hiérarchie des règles dans RuleML-Vue graphique

avons au niveau le plus haut les règles de réaction et les règles de transformation puis au deuxième niveau nous avons les règles de dérivation, ensuite les règles de fait de requête et de contrainte d'intégrité. La syntaxe des règles d'une catégorie du niveau supérieur est valable pour les catégories du niveau inférieur. Cependant RuleML propose des syntaxes particulières par catégories de règles. Par exemple :

- règle de réaction : <**rule**> <**_body**> <**and**> *prem*1 ... *premN* </**and**> </**_body**> <**_head**> action </**_head**> </**rule**>
- règle de contrainte d'intégrité :<**ic**> <**_body**> <**and**> *prem*1 ... *premN* </**and**> </**_body**> </**ic**> implémentée en <**rule**> <**_body**> <**and**> *prem*1 ... *premN* </**and**> </**_body**> <**_head**> <**signal**> inconsistency </**signal**> </**_head**> </**rule**>

- règle de dérivation :<**imp**> <**_head**> conc </**_head**> <**_body**> <**and**> *prem*1 ... *premN* </**and**> </**_body**> </**imp**> implémentée par <**rule**> <**_body**> <**and**> *prem*1 ... *premN* </**and**> </**_body**> <**_head**> <**assert**> conc </**assert**> </**_head**> </**rule**>
- règles de faits :<**fact**> <**_head**> conc </**_head**> </**fact**> implémentée par <**imp**> <**_head**> conc </**_head**> <**_body**> <**and**> </**and**> </**_body**> </**imp**>

RuleML propose les deux types de négation faible et forte.

3.6.4 Implémentation de RuleML via XSLT

XSLT peut lui-même être vu comme étant un langage de programmation basé sur des règles et qui opère sur des éléments XML. Ces éléments peuvent aussi être des règles exprimées en XML. Pour l'implémentation de ses langages vers des moteurs de règles spécifiques, RuleML utilise XSLT. RuleML propose un certain nombre de transformations vers certain moteurs de règles comme Jess.

Les travaux sur RuleML continuent toujours, cependant, il n'est pas arrivé à s'imposer comme formalisme standard. Nous allons dans la section suivante voir une autre tentative de standardisation chez l'OMG et qui s'intéresse uniquement aux experts métier.

3.7 Semantics of Business Vocabulary and Business Rules (SBVR) de l'OMG

Le standard Semantics of Business Vocabulary and Business Rules [46, 47] de l'OMG permet de fournir un vocabulaire pour décrire le sens des concepts. Une partie de SBVR appelée *"Logical Formulation"* porte principalement sur la structure de ces sens. L'objectif final du standard SBVR est de fournir un méta-modèle qui permet d'établir des interfaces et des échanges de données pour les outils qui créent, organisent, analysent et utilisent les vocabulaires et règles métier [48].

3.7.1 Position du SBVR dans le MDA

SBVR fait entièrement partie de la couche de modèle métier du MDA comme le montre la Figure 33.

Cette position implique deux choses. La première est que SBVR s'adresse aux règles et vocabulaires métier et que les autres aspects des modèles métiers doivent aussi être prises en compte et développés. Ces derniers incluent les processus métier et l'organisation des structures qui sont visés par d'autres standards au niveau de l'OMG. La deuxième chose est que les modèles métiers et les autres modèles supportés par le SBVR ne s'adressent

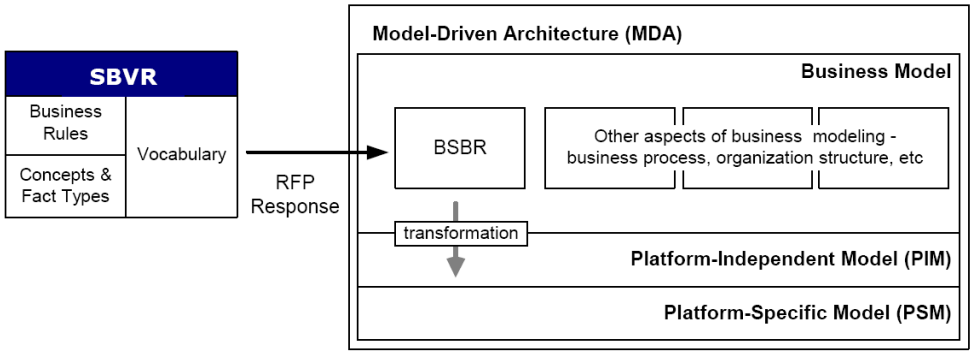

FIG. 33 – Position du SBVR dans le MDA

pas aux experts système (IT) mais aux experts métier, bien que ce standard souligne la nécessité de pouvoir passer du CIM vers le PIM.

3.7.2 Les notions clés du SBVR

Sémantique

La *"Sémantique"* désigne l'étude du langage et des signes linguistiques (mots, expressions, phrases) du point de vue du sens (du grec "semantikos", "qui signifie"). Il s'agit de savoir comment un signe tel que "X" se charge de sens, comment il est utilisé par l'énonciateur, puis perçu et interprété par le co-énonciateur. La sémantique analyse le sens des mots et le processus par lequel ils se chargent de ce sens. L'étude de la sémantique peut être philosophique (sémantique pure) ou linguistique (sémantique descriptive ou théorique). On y ajoute généralement une troisième approche, la sémantique générative. En SBVR ces signes peuvent être de tout type : des mots, des phrases, des codes, des nombres, des icônes, des sons, etc [47].
SBVR dispose de deux types de vocabulaires spécialisés :

- le SBVR "vocabulaire pour décrire des vocabulaires métier" qui s'intéresse à tous les types de termes et de sens (autre que le sens des règles métier).
- le SBVR "vocabulaire pour décrire des règles métier" qui s'intéresse à la spécification du sens des règles métier et se base sur le SBVR "vocabulaire pour décrire des vocabulaires métier".

Ces deux types de vocabulaires sont séparés de sorte qu'ils puissent être utilisés indépendamment, par exemple le SBVR "vocabulaire pour décrire des vocabulaires métier" pourrait être réutilisé dans les processus métier ou les rôles organisationnels.

Le vocabulaire métier

Un vocabulaire métier contient tous les termes et les définitions de concepts spécialisés qu'une organisation ou une communauté donnée utilise dans leur langage en vue d'effectuer leur métier. Le SBVR "vocabulaire pour décrire des vocabulaires métier" se base sur les standards ISO suivants : ISO 1087-1 (2000) qui porte sur la théorie et l'application du vocabulaire, ISO 704 (2000) qui porte sur les principes et méthodes et enfin ISO 860 (1996) qui porte sur l'harmonisation des concepts et des termes. SBVR est le résultat de l'intégration de ces standards ISO, de logiques formelles, de linguistiques et d'expériences pratiques [47].

Règles métier

Pour SBVR, une règle est un élément de guidage qui introduit une obligation ou une nécessité. SBVR subdivise les règles métier en deux catégories :

- les **règles structurelles** (indiquant une nécessité) : ce sont des règles qui portent sur comment est-ce que le métier organise ou structure les éléments qui le composent. Les règles structurelles sont des compléments de définition. Par exemple : Nécessité : Un client a au moins :
  - une réservation de location.
  - une location en cours.
  - une location accomplie dans les 5 dernières années.
- les **règles opératives** (indiquant une obligation) : ce sont des règles qui contrôlent la manière dont les activités du métier sont conduites. Contrairement aux règles structurelles, les règles opératives sont celles qui peuvent être directement violées par les intervenants du métier. Par exemple : un client qui apparaît comme étant intoxiqué ou drogué ne doit pas être en possession d'une voiture de location.

Echange de sémantique

L'objectif du méta-modèle de SBVR est de fournir des interfaces standardisées pour permettre l'échange de données entre des outils qui collectent, organisent, analysent et utilisent les vocabulaires et règles métier. Le méta-modèle SBVR facilitera éventuellement, à différents outils avec des constructeurs différents, la validation, l'analyse, l'alignement, et la fusion de règles métier. Le méta-modèle de SBVR se base sur le méta-formalisme MOF que nous verrons plus en détail à la partie 4.2.5 et utilise la sérialisation XMI 2.1. Le méta-modèle de SBVR ne s'adresse pas aux experts métier mais plutôt aux experts techniques qui développent les outils pour ces experts métier.

3.7.3 Mise en œuvre du SBVR

SBVR n'est pas un formalisme exécutable. Il s'intéresse uniquement à la couche métier (CIM) du MDA (que nous verrons au chapitre 4) et donc s'adresse particulièrement aux experts métier. SBVR utilise le langage que tous les experts métier comprennent : le langage naturel. La mise en œuvre du SBVR se fait en deux étapes : la mise en place du vocabulaire métier et l'écriture des règles métier qui se basera sur les termes et les concepts définis dans le vocabulaire. La mise en place d'un vocabulaire commun est capital dans un projet car il faut que les divers intervenants utilisent le même langage et aient la même définition du sens d'un concept pour qu'il n'y ait pas d'ambiguïté. Nous allons maintenant voir quelques exemples de règles en SBVR :

1. Règle métier structurelle : Il est nécessaire que chaque location *désigne* exactement un type de voiture
   Types de fait utilisés : location *désignant* un type de voiture
2. Règle métier opérative : Il est obligatoire que la durée de chaque location n'excède pas 90 jours.
   Types de fait utilisés : location *possède* une durée.
3. Règle métier opérative : Il est obligatoire que le conducteur d'une location ait un permis d'au moins 2 ans.
   Types de fait utilisés : location *a* un conducteur.

3.8 Standardisation de règles de production par l'Object Management Group (OMG)

En Septembre 2003, l'OMG a sorti une demande de proposition (Request For Proposal) d'un standard pour le formalisme de règles de production [49]. Ce futur standard porte le nom de Production Rules Representation (PRR) dont nous allons faire l'étude dans les sections suivantes. Contrairement aux autres tentatives de standardisation, le PRR est plus ciblé et s'adresse uniquement aux règles de production.

3.8.1 le "Production Rule Representation" de l'OMG

Contexte du problème

Les règles de production permettent, de manière commode, l'implémentation de règles métier. En effet les mécanismes d'inférences ont été pendant longtemps utilisés pour implémenter des systèmes métier sophistiqués (système "expert") qui appliquent une logique complexe de règles métier permettant aux experts métier de prendre des décisions métier critiques et complexes.

D'un autre côté, le nombre d'entreprises utilisant un système de gestion de règles métier augmentent, et pendant ce temps le nombre d'entreprises et d'outils utilisant UML comme langage de modélisation dans des applications métier ne cessent d'augmenter. Ceci a conforté la tendance à combiner modélisation de règle métier et modélisation UML qui peut permettre un meilleur moyen de modéliser des règles de production avec UML. Le manque d'une telle possibilité aboutit souvent aux résultats suivants :

- Omission des règles au niveau du modèle.
- Rajout des règles au modèle en utilisant un moyen pas conforme à la sémantique d'extension de UML.
- Echange de règles impossible entre modèles.
- Manque de standardisation dans les outils permettant la création de règles métier.

Une standardisation du formalisme de règles de production et l'intégration de leur modélisation dans UML seront bénéfiques aux experts métier et système en apportant des solutions aux limitations que nous venons de lister. Cette demande de proposition de l'OMG exige un méta-modèle d'un langage pouvant être utilisé avec des modèles UML, cependant aucun des langages UML ne conviennent pour la modélisation des règles de production.

Différences avec des langages UML existants

Deux mécanismes de UML semblent convenir pour le formalisme de règles de production : Object Constraint Language (OCL) [50] et Action Semantic (AS) [51]. Voyons maintenant pourquoi ni l'un ni l'autre n'est satisfaisant.

Le langage de contrainte d'UML (OCL) :
OCL fournit un langage d'expression formel très riche et simple qui peut être utilisé pour exprimer la partie *conditions* d'une règle de production. Bien que la syntaxe de OCL permette de faire référence aux opérations (méthodes) définies dans le modèle, ces dernières doivent être des opérations de requête. OCL n'offre pas la possibilité de faire appel à des opérations à effet de bord, qui vont changer l'état du système, ce qu'une action d'une règle de production peut très bien faire. Néanmoins il faut savoir que OCL 2.0 permet de faire référence à des opérations qui changent l'état du système. Cependant la sémantique de cette référence est que ces opérations, changeant l'état du système, doivent être invoquées seulement si la véracité de la contrainte est testée. Ce genre d'expressions à effet sont utilisables uniquement dans le cas de post-conditions. Cette sémantique ne satisfait pas les besoins de la partie action d'une règle de production.

Action Semantics (AS) :
Action Semantics peut être utilisée pour faire appel à des opérations avec effet de bord dans la partie action d'une règle de production. En plus Action Semantics permet les blocs de type "SI condition, ALORS action". Cependant les points suivants font que Action Semantics ne soit pas une solution appropriée :

- **les sémantiques d'exécution** : Action Semantics permet deux modes d'exécution

pour les actions : l'exécution parallèle et l'exécution séquentielle qui sont basées sur les flots de contrôles et de données entre les actions. Les règles de production n'ont pas cette notion de séquence car la manière dont elles s'exécutent dépendra du moteur d'inférence utilisé. En plus de cela Action Semantics n'offre que des séquences acycliques alors que lors d'une exécution par un moteur inférence, on peut avoir des règles de production s'exécutant en cycle.

- **expressions à multiples quantificateurs** : Action Semantics ne permet pas d'utiliser plusieurs quantificateurs dans une même expression. De ce fait il ne permet pas d'avoir des expressions composées de plusieurs tuples. Par exemple avec Action Semantics on ne pourrait pas écrire l'expression suivante :
  Pour tout *locataire* l et **pour tout** *Location* loc Si l.pasDeLocation et loc.libre et conforme(l, loc) Alors assignerA(l, loc) ;
  L'utilisation d'ensembles de tuples est essentielle à la gestion de la recherche de motif dans des algorithmes de type Rete [15, 14].

Les exigences du futur standard

Selon l'Object Management Group, les soumissions pour ce standard devront satisfaire les points suivants :

Avoir un méta-modèle :
Il est obligatoire de fournir un méta-modèle de type Meta-Object Facility (MOF) [52] pour règles de production. En termes de MOF, le méta-modèle décrira les éléments en terme d'objets, propriétés et de relations entre eux. Le méta-modèle sera constitué de trois composants majeurs qui devront être spécifiés :

- Une syntaxe abstraite qui est formée des conditions et actions de la règle de production, ainsi que de leurs structures et relations avec les éléments UML existants.
- Une sémantique statique qui est formée de règles bien définies et compréhensibles pour la construction de formules issues de la syntaxe abstraite. Ces règles seront des contraintes sur les méta-classes, méta-associations (relation) et méta-attributs (propriété). Ces contraintes pourront être définies en OCL.
- Une sémantique dynamique : chaque élément du méta-modèle doit être accompagné de sa signification. La sémantique dynamique d'un méta-modèle est en général exprimée en langage naturel, cependant ce langage doit être strict c'est à dire precis et bien défini. Cette sémantique dynamique est l'objectif d'un autre standard de l'OMG, le Semantic Business Vocabulary and business Rule [46, 47], que nous avons présenté dans ce document à la section 3.7.

Fournir un langage de transformation d'une instance du méta-modèle :
Il faudra également, à titre d'exemple, fournir un langage de transformation d'une instance du méta-modèle vers le langage de règle d'un moteur d'inférence spécifique.

3.8.2 Les soumissions reçues pour le langage de règles de production

Pour l'instant (13 décembre 2007), une seule soumission a été reçue par l'OMG concernant le Production Rule Represenation. C'est un document conjointement soumis par Fair Isaac Corporation, ILOG SA et IBM Corporation [53].

La soumission de Fair Isaac, Ilog et IBM

La première chose qu'il faut noter concernant cette soumission, est que le méta-modèle MOF2 qui est fourni ne prend en compte que les règles de production en chaînage avant alors que la demande de proposition exigeait de prendre en compte aussi bien le chaînage avant qu'arrière.
Le méta-modèle qui est fourni est composé :

- d'une structure centrale appelée PRR Core.
- d'une syntaxe abstraite basée sur OCL. Cette syntaxe permettant d'exprimer les expression PRR, est une extension du méta-modèle PRR Core et est appelée PRR OCL.

Aperçu du PRR-Core :
Le PRR-Core est un ensemble de classes qui permet la définition de règles et d'ensembles de règles de production (ruleset). Ici toutes les conditions et les actions sont de simples et "opaques" chaînes de caractères.
Les Figures 34, 35 et 36 sont une partie du modèle montrant les concepts généraux de rulesets et de règles.

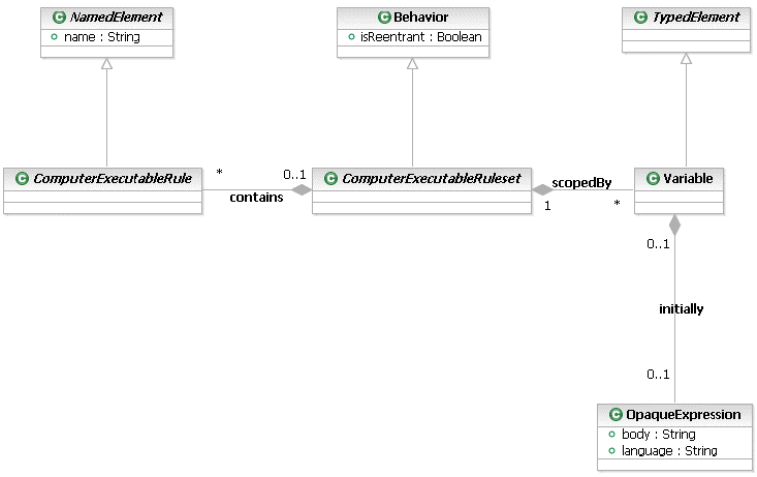

Fig. 34 – Concepts du PRR-Core

Dans les sections précédentes, nous disions que OCL ne peut pas être utilisé pour représenter des règles de production car présente des manquements. Cette soumission propose

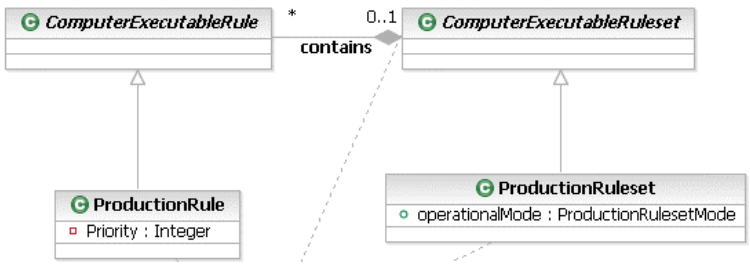

FIG. 35 – Les classe ProductionRuleset du PRR-Core

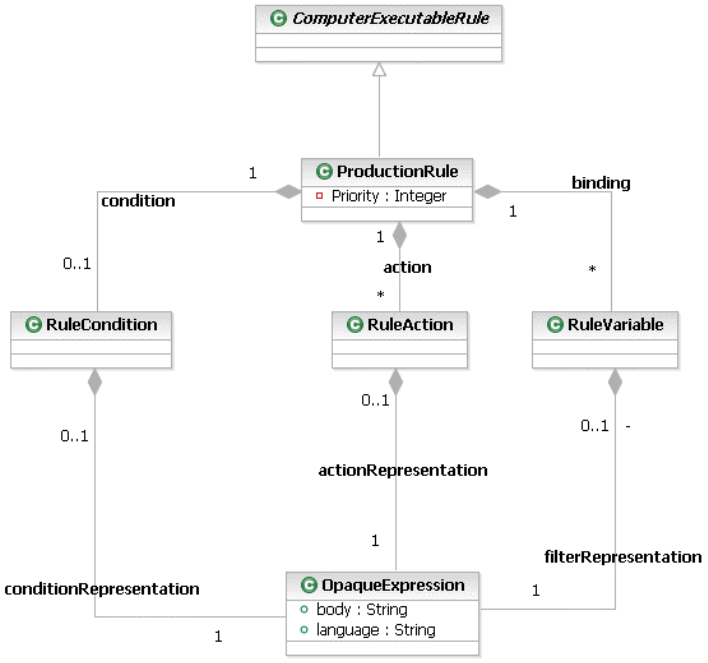

FIG. 36 – Les classe ProductionRule du PRR-Core

PRR-OCL qui est extension de OCL 2.0 qui contourne ces manquements mais ne respectant plus les exigences de la RFP.
Il faut également souligner que les trois (3) auteurs de cette soumission font parti du groupe de travail du W3C pour la mise en place d'un formalisme de règles métier (Rule Interchange Format - RIF). Tout au long de leur proposition, ils ont fait référence au RIF en montrant comment il se chevauche avec le PRR.

3.9 Rule Interchange Format (RIF) du W3C

Le but du RIF (Rule Interchange Format) 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. Vu la variété des types de langages de règles et des moteurs de règles, le groupe de travail du W3C en charge du RIF [54] a fait le choix d'avoir un langage central ou noyau étendu avec des extensions au fur et à mesure selon les besoins des utilisateurs. L'établissement de ce formalisme est prévu pour se dérouler en deux phases. La première phase, qui est prévue pour se terminer en 2008, a pour but de mettre sur place ce langage central et un ensemble réduit d'extensions. La seconde phase aura pour but de créer les extensions suivant les besoins des utilisateurs. Nous allons maintenant parler des travaux qui ont été effectués lors de la première phase en nous basant sur la version Draft du 30 Mars 2007 et disponible en Juin 2007 [55].

3.9.1 Aperçu du RIF

Le RIF a une architecture en couches qui s'organise au tour des langages *dialect* qui permettent d'étendre le langage central (RIF Core). Un *dialect* est un langage de règle qui a une syntaxe et une sémantique bien définies. Un *dialect* peut être une extension (aussi bien d'un point de vue syntaxique que sémantique) d'un autre, mais ils peuvent aussi être incompatibles. Quelqu'en soit le cas, il est cependant indispensable que tous les *dialects* soient des extensions du *dialect* central ou noyau.
D'un point de vue théorique, le *dialect* central du RIF est un langage de *Horn rules* [56, 57] précis avec une notion d'égalité et ayant une sémantique du premier ordre. Les langages *Horn rules*, qui sont à la base de plusieurs applications d'Intelligence Artificielle, sont un sous-ensemble de la logique du premier ordre [58]. Bien qu'étant basé sur les langages de *Horn rules*, le *dialect* central prévoit un certain nombre d'extensions pour supporter les caractéristiques telles ques les objets, les frames, l'utilisation de URIs comme identifiants pour les concepts et l'utilisation des types de données XML Schema. Ces dernières caractéristiques font du RIF un langage de règles pour le Web. Cependant le RIF est conçu pour permettre l'interopérabilité de langages de règles de manière générale et son utilisation n'est pas uniquement destinée au Web. Le RIF prévoit des extensions dans sa deuxième phase pour supporter les logiques du premier ordre pures, la négation faible (NAF), les règles métier (ou production), les règles réactives, etc.

La partie centrale du RIF Core est le ***langage de condition***. Le langage de condition définit la syntaxe et la sémantique pour la partie "body" ou condition et des requêtes.

3.9.2 Le langage de condition du RIF

Le langage de condition du RIF est prévu pour être la base commune des futurs *dialects* du RIF. Ces futurs *dialects* pourront utiliser le langage de condition dans les cas suivants :

- le corps des règles et des requêtes pour les *dialects* de type programmation logique déclarative (LP) ;
- le corps des règles pour les *dialects* de type premier ordre (FO) ;
- les conditions dans le corps des règles pour les *dialects* de type règles de production ;
- les conditions et les événements dans le corps des règles pour les *dialects* de type règles de réaction (RR) ;
- les contraintes d'intégrité (IC).

Le langage de condition du RIF est conçu, pour l'instant, pour être utilisé uniquement dans le corps des règles et les requêtes, et pas dans les parties "head" ou action. La Figure 37 montre une vue du langage de condition.

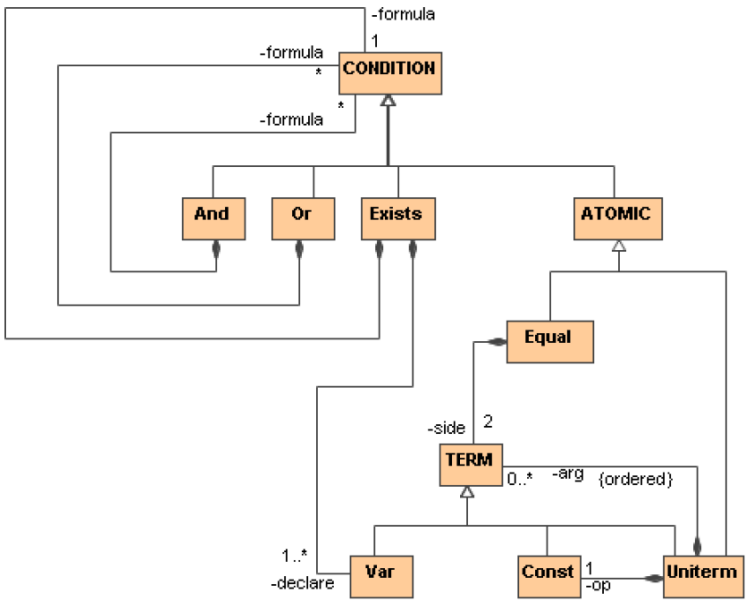

FIG. 37 – Diagramme UML du langage de condition du RIF

3.9.3 Le langage de règles du RIF

Le langage de règle que propose le RIF est une extension de son langage de condition, où les conditions deviennent le corps des règles. La partie action est également une sous

partie du langage de condition par l'utilisation de *ATOMIC* (voir Figure 37)

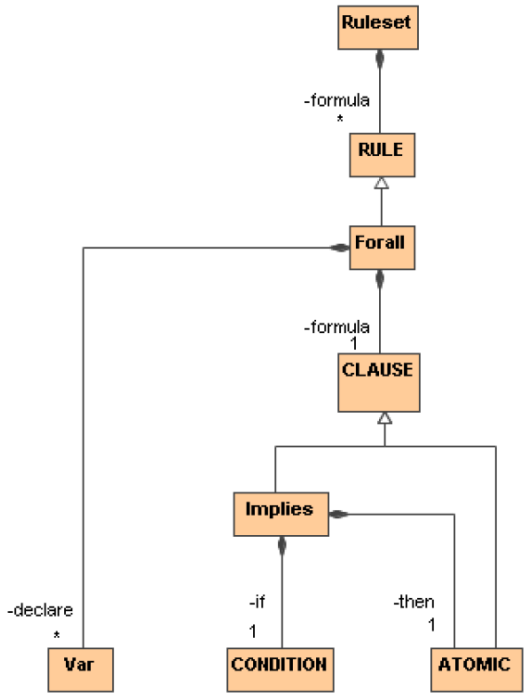

Fig. 38 – Diagramme UML du langage de règle du RIF

3.9.4 Compatibilité du RIF

Bien que l'utilisation du futur standard RIF soit prévue pour être générale, son objectif principal est le Web et devra jouer un rôle clé dans le Web Sémantique. Dans ce sens le RIF prévoit d'être compatible avec OWL et RDF dans ses futures versions.

Après avoir exposé les différentes tentatives de standardisation qui ont existés ou sont en cours, nous allons présenter nos propres travaux dans ce sens, car étant dans un contexte industriel, il fallait avoir assez rapidement un langage de règle fonctionnel indépendant de tout moteur de règle.

3.10 Notre formalisme de règles métier : ERML

Il n'existe pas encore de formalisme standard de règles métier. Dans la suite *e-Citiz*, lors de la spécification de l'intégration de l'approche par règles métier, il était convenu de stocker les règles métier dans un formalisme indépendant de tout moteur de règles,

ce qui a d'ailleurs été à l'origine de la proposition du sujet de cette thèse. En effet, l'un des objectifs dans *e-Citiz* est de permettre aux clients de changer de moteur de règles de manière transparente. Dès le départ nous avions choisi d'avoir une approche par pivot comme le montre la Figure 39. Nous avons commencé par définir une grammaire que nous avons appelée E-Citiz Rule Markup Language (ERML), ensuite nous avons mis en place un moteur de transformations vers les moteurs de règles cibles. Enfin ces moteurs de règles cibles sont embarqués dans le moteur de *e-Citiz*.

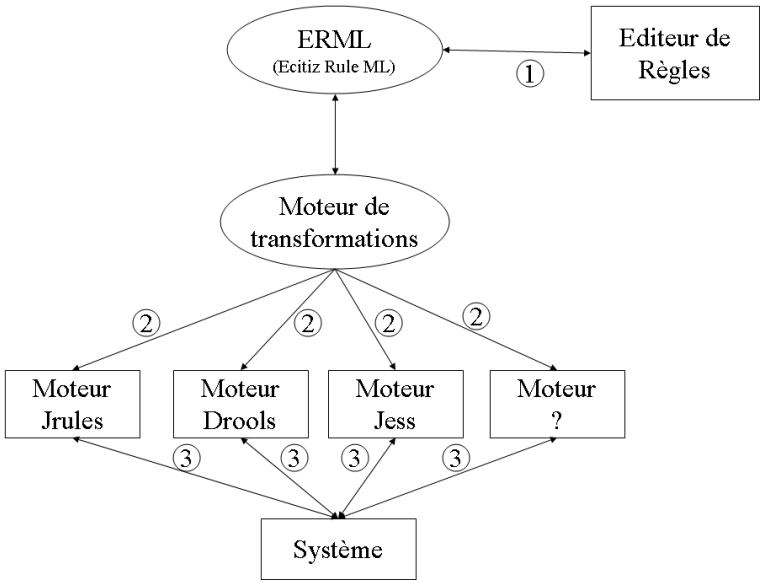

FIG. 39 – Notre formalisme ERML et ses translateurs

Nous allons voir ces deux composants de notre approche : à savoir la grammaire du langage de règles et le moteur de transformations.

3.10.1 Le formalisme ERML

Le contexte industriel du projet imposait certaines contraintes. L'une de ces contraintes était l'utilisation du langage XML pour le stockage des règles car c'est le format utilisé dans le studio *e-Citiz*. Ainsi pour la définition de la grammaire nous avons utilisé XSD. Au départ nous avons commencé par écrire des règles dans plusieurs langages de moteur de règles afin de les comparer et de recenser les constructions communes entre eux. Ensuite nous avons étudié les grammaires proposées par les tentatives de standardisation comme CommonRule et RuleML. Nous allons dans les sous-sections suivantes présenter les éléments clés de notre langage.

Le terme *Ruleset*

ruleset = (rulesetlabel ?, (Fact | Rule))*

Un *Ruleset* est l'élément de plus haut niveau, il représente un ensemble de règles. Un *Ruleset* est composé d'au moins un *rulesetLabel* qui est un ensemble d'informations et de zéro ou plusieurs *Fact* ou *Rule.* Un *Ruleset* peut aussi avoir deux attributs de type *orderRulesMode* qui représentent le mode de priorité et *negationMode* (voir plus loin). La Figure 40 montre comment un ruleset est formalisé en XSD.

```
<xsd:attributeGroup name="Ruleset.attlist">
        <!-- put attributs here -->
        <xsd:attribute name="orderRulesMode" type="orderedRulesModeType" use="optional"/>
        <xsd:attribute name="negationMode" type="negationModeType" use="optional"/>
</xsd:attributeGroup>
<xsd:group name="Ruleset.content">
        <xsd:sequence>
                <xsd:element ref="rulesetlabel" minOccurs="0"/>
                <xsd:element ref="Fact" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="Rule" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
</xsd:group>
<xsd:complexType name="Ruleset.type">
        <xsd:group ref="Ruleset.content" minOccurs="0"/>
        <xsd:attributeGroup ref="Ruleset.attlist"/>
</xsd:complexType>
<xsd:element name="Ruleset" type="Ruleset.type"/>
```

FIG. 40 – Définition en XSD du terme ruleset

Le terme *Fact*

Fact = (rulelabel, (VariableDeclaration), (action | Expression | NegationExpression))*

Les *Fact* sont les faits que l'on connaît du problème à résoudre. Il peut aussi servir uniquement pour la déclaration de variable. Un *Fact* a un *label*, un ensemble de *Variable* qui peut être nul, une *action* , une *Expression* ou une *NegationExpression.* La Figure 41 montre comment le terme *Fact* est modélisé en XSD.

Le terme *Rule*

Rule = (rulelabel, (VariableDeclaration), (conditions, action))*

```
<xsd:attributeGroup name="Fact.attlist"/>
 <xsd:group name="Fact.extend">
        <xsd:choice>
               <xsd:element ref="action"/>
               <xsd:element ref="Expression"/>
               <xsd:element ref="NegationExpression"/>
        </xsd:choice>
 </xsd:group>
 <xsd:group name="Fact.content">
        <xsd:sequence>
               <xsd:element ref="rulelabel"/>
               <xsd:element ref="VariableDeclaration" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:group ref="Fact.extend"/>
        </xsd:sequence>
 </xsd:group>
 <xsd:complexType name="Fact.type">
        <xsd:group ref="Fact.content"/>
        <xsd:attributeGroup ref="Fact.attlist"/>
 </xsd:complexType>
 <xsd:element name="Fact" type="Fact.type"/>
```

FIG. 41 – Définition en XSD du terme Fact

Une règle a un label, utilise un ensemble de déclarations de variable qui peut être vide et possède une partie condition et action. Une règle peut également dire de manière optionnelle si elle a une prémisse qui utilise la négation ou qui est de type implication. La Figure 42 montre comment le terme *Rule* est modélisé en XSD.

```
<xsd:attributeGroup name="Rule.attlist">
        <xsd:attribute name="negatetedPrem" type="xsd:boolean" use="optional"/>
        <xsd:attribute name="implicationalPrem" type="xsd:boolean" use="optional"/>
 </xsd:attributeGroup>
 <xsd:group name="Rule.content">
        <xsd:sequence>
               <xsd:element ref="rulelabel"/>
               <xsd:element ref="VariableDeclaration" minOccurs="0"maxOccurs="unbounded"/>
               <xsd:element ref="conditions"/>
               <xsd:element ref="action"/>
        </xsd:sequence>
 </xsd:group>
 <xsd:complexType name="Rule.type">
        <xsd:group ref="Rule.content"/>
        <xsd:attributeGroup ref="Rule.attlist"/>
 </xsd:complexType>
 <xsd:element name="Rule" type="Rule.type"/>
```

FIG. 42 – Définition en XSD du terme Rule

Le terme *rulesetLabel*

rulesetLabel = (comment ?, name ?, version ?, isWriteable ?)

rulesetLabel est le libellé pour un ruleset, il est optionnel. Ce Libellé, comme le montre la Figure 43, contient un commentaire, un nom, une version et un drapeau pour savoir si le ruleset est disponible en écriture ou pas. Attention le nom du ruleset n'est pas forcément le nom du fichier contenant le ruleset.

```
<xsd:attributeGroup name="rulesetlabel.attlist"/>
  <xsd:group name="rulesetlabel.content">
        <xsd:sequence>
              <xsd:element ref="comment" minOccurs="0"/>
              <xsd:element ref="name" minOccurs="0"/>
              <xsd:element ref="version" minOccurs="0"/>
              <xsd:element name="isWriteable" type="xsd:boolean" minOccurs="0"/>
        </xsd:sequence>
  </xsd:group>
  <xsd:complexType name="rulesetlabel.type">
        <xsd:group ref="rulesetlabel.content"/>
        <xsd:attributeGroup ref="rulesetlabel.attlist"/>
  </xsd:complexType>
  <xsd:element name="rulesetlabel" type="rulesetlabel.type"/>
```

FIG. 43 – Définition en XSD du terme rulesetLabel

Le terme *ruleLabel*

ruleLabel = (comment ?, salience ?, visibile ?, refraction ?, createadTime ?, ?lastModificationDateTime)

Une règle a un libellé. Ce libellé doit avoir un commentaire, une salience (priorité), une visibilité, un drapeau renseignant si la règle est autorisée à boucler ou non lors de l'exécution, une date de création et une date de dernière modification. Tout ceci est optionnel, la seule chose obligatoire pour la propriété d'une règle est son nom. La Figure 44 montre la modélisation en XSD.

Le terme *orderedRulesModeType*

Le terme *orderedRulesModeType* est optionnel et permet d'indiquer quel type d'ordre sera utilisé dans un ruleset. Actuellement il existe 3 types d'ordre :

1. *full-first-order* : les règles seront déclenchées selon l'ordre rencontré dans le fichier de règles.
2. *higher-order* : les règles sont déclenchées selon la priorité (salience) spécifiée. La règle qui a la propriété la plus haute sera exécutée en premier.
3. *qualitative-order* : des prédicats *overrides* sont utilisés pour dire quelle règle aura la primauté sur telle règle. Par exemple nous pouvons avoir *"Règle Vérifier carburant" overrides "Règle Vérifier eau"* voudra dire qu'en cas de conflit, la règle "Règle Vérifier carburant" a la priorité sur la règle "Règle Vérifier eau".

```
<xsd:attributeGroup name="rulelabel.attlist">
        <xsd:attribute name="name" type="xsd:string" use="required"/>
  </xsd:attributeGroup>
  <xsd:group name="rulelabel.content">
        <xsd:sequence>
              <xsd:element ref="comment" minOccurs="0"/>
              <xsd:element ref="salience" minOccurs="0"/>
              <xsd:element name="visible" type="xsd:boolean" minOccurs="0"/>
              <xsd:element name="refraction" type="xsd:boolean" minOccurs="0"/>
              <xsd:element name="lastModificationDateTime" type="xsd:dateTime" minOccurs="0"/>
              <xsd:element name="creationTime" type="xsd:dateTime" minOccurs="0"/>
        </xsd:sequence>
  </xsd:group>
  <xsd:complexType name="rulelabel.type">
        <xsd:group ref="rulelabel.content"/>
        <xsd:attributeGroup ref="rulelabel.attlist"/>
  </xsd:complexType>
  <xsd:element name="rulelabel" type="rulelabel.type"/>
```

FIG. 44 – Définition en XSD du terme ruleLabel

La Figure 45 montre la modélisation en XSD.

```
<xsd:simpleType name="orderedRulesModeType">
        <xsd:restriction base="xsd:string">
              <xsd:enumeration value="full-firts-order"/>
              <xsd:enumeration value="higher-order"/>
              <xsd:enumeration value="qualitative-order"/>
        </xsd:restriction>
  </xsd:simpleType>
```

FIG. 45 – Définition en XSD du terme orderedRulesModeType

Le terme *negationModeType*

Le terme *negationModeType* est optionnel et permet de spécifier quel type de négation utiliser. Il y a deux types de négation :

1. La négation forte qui est utilisée par défaut.
2. La négation faible ou as-failure.

La Figure 46 montre la modélisation en XSD.

```
<xsd:simpleType name="negationModeType">
      <xsd:restriction base="xsd:string">
            <xsd:enumeration value="strong"/>
            <xsd:enumeration value="weak"/>
      </xsd:restriction>
</xsd:simpleType>
```

FIG. 46 – Définition en XSD du terme negationModeType

Le terme *name*

name = (Individual)

Ce terme représente le nom de ruleset ou de la règle. C'est une constante (Individuel) comme le montre la Figure 47.

```
<xsd:attributeGroup name="name.attlist"/>
<xsd:group name="name.content">
        <xsd:sequence>
                <xsd:element ref="Individual"/>
        </xsd:sequence>
</xsd:group>
<xsd:complexType name="name.type">
        <xsd:group ref="name.content"/>
        <xsd:attributeGroup ref="name.attlist"/>
</xsd:complexType>
<xsd:element name="name" type="name.type"/>
```

FIG. 47 – Définition en XSD du terme name

Le terme *salience*

salience = (Individual)

Ce terme représente la priorité d'une règle et il est optionnel. Il n'est pris en compte que si le type d'ordre est par priorité c'est-à-dire si l'attribut *orderedrulesmodetype* est égale à *higher-order*. C'est une constante (Individuel) comme le montre la Figure 48.

```
<xsd:attributeGroup name="salience.attlist"/>
<xsd:group name="salience.content">
        <xsd:sequence>
                <xsd:element ref="Individual"/>
        </xsd:sequence>
</xsd:group>
<xsd:complexType name="salience.type">
        <xsd:group ref="salience.content"/>
        <xsd:attributeGroup ref="salience.attlist"/>
</xsd:complexType>
<xsd:element name="salience" type="salience.type"/>
```

FIG. 48 – Définition en XSD du terme salience

Le terme *conditions*

conditions = (Expression | AndExpression | OrExpression | NegationExpression)+

conditions représente la partie "body" de la règle. Si cette partie est évaluée à vraie, la partie action est déclenchée. Dans une condition on peut avoir soit une expression simple, soit une expression composée soit une expression négative. Une condition peut être vide (ne pas avoir de prémisses), dans ce cas elle est considérée comme étant tout le temps vraie. La Figure 49 montre la modélisation de *conditions* en XSD.

```
<xsd:attributeGroup name="conditions.attlist"/>
 <xsd:group name="conditions.content">
        <xsd:choice>
               <xsd:element ref="Expression" minOccurs="0"/>
               <xsd:element ref="AndExpression" minOccurs="0"/>
               <xsd:element ref="OrExpression" minOccurs="0"/>
               <xsd:element ref="NegationExpression" minOccurs="0"/>
        </xsd:choice>
 </xsd:group>
 <xsd:complexType name="conditions.type">
        <xsd:group ref="conditions.content"/>
        <xsd:attributeGroup ref="conditions.attlist"/>
 </xsd:complexType>
 <xsd:element name="conditions" type="conditions.type"/>
```

FIG. 49 – Définition en XSD du terme conditions

Le terme *action*

action = ((Expression) | NegationExpression+)*

action représente la partie "head" de la règle. On l'appelle aussi partie conséquence ou "then". Dans une action on peut avoir un ensemble d'expressions ou une expression négative. La Figure 50 montre la modélisation de *action* en XSD.

```
<xsd:attributeGroup name="action.attlist"/>
<xsd:group name="action.content" >
        <xsd:choice>
               <xsd:element ref="Expression" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element ref="NegationExpression" minOccurs="0"/>
        </xsd:choice>
</xsd:group>
<xsd:complexType name="action.type">
        <xsd:group ref="action.content"/>
        <xsd:attributeGroup ref="action.attlist"/>
</xsd:complexType>
<xsd:element name="action" type="action.type"/>
```

FIG. 50 – Définition en XSD du terme action

Le terme *Expression*

Expression = (SimpleExpression | ComplexExpression)

Une expression peut être simple ou complexe (voir Figure Figure 51).

```
<xsd:attributeGroup name="Expression.attlist"/>
 <xsd:group name="Expression.extend">
        <xsd:sequence>
               <xsd:choice>
                      <xsd:element ref="SimpleExpression"/>
                      <xsd:element ref="ComplexExpression"/>
               </xsd:choice>
        </xsd:sequence>
 </xsd:group>
 <xsd:group name="Expression.content">
        <xsd:sequence>
               <xsd:group ref="Expression.extend"/>
        </xsd:sequence>
 </xsd:group>
 <xsd:complexType name="Expression.type">
        <xsd:group ref="Expression.content"/>
        <xsd:attributeGroup ref="Expression.attlist"/>
 </xsd:complexType>
 <xsd:element name="Expression" type="Expression.type"/>
```

FIG. 51 – Définition en XSD du terme Expression

Le terme *SimpleExpression*

SimpleExpression = (basicOperator, (Individual | Variable | Invalid), (Individual | Variable | Invalid))

Une expression simple est composée d'un opérateur basique, et de deux opérandes qui peuvent être soit une constante (*Individual*), une variable ou la notion d'invalidité. Elle permet d'imprimer des expressions booléennes simples du genre $x > y$, $z == 1$ etc. Nous avons introduit la notion d'invalidité pour permettre d'exprimer une expression incomplète dans laquelle il manque soit l'opérateur soit une ou deux opérandes. La Figure 52 montre la modélisation en XSD.

Le terme *ComplexExpression*

ComplexExpression = (Operator, (Individual | Variable | SimpleExpression | ComplexExpression | Invalid))*

```
<xsd:attributeGroup name="SimpleExpression.attlist"/>

 <xsd:group name="SimpleExpression.extend">
        <xsd:sequence>
               <xsd:choice maxOccurs="2">
                      <xsd:element ref="Individual"/>
                      <xsd:element ref="Variable"/>
                      <xsd:element ref="Invalid"/>
               </xsd:choice>
        </xsd:sequence>
 </xsd:group>
 <xsd:group name="SimpleExpression.content">
        <xsd:sequence>
               <xsd:element name="basicOperator" type="basicOperator"/>
               <xsd:group ref="SimpleExpression.extend"/>
        </xsd:sequence>
 </xsd:group>
 <xsd:complexType name="SimpleExpression.type">
        <xsd:group ref="SimpleExpression.content"/>
        <xsd:attributeGroup ref="SimpleExpression.attlist"/>
 </xsd:complexType>
 <xsd:element name="SimpleExpression" type="SimpleExpression.type"/>
```

FIG. 52 – Définition en XSD du terme SimpleExpression

Une expression complexe est composée d'un opérateur (basique ou non) et d'un ensemble qui peut être vide de *Individual, Variable, SimpleExpression, ComplexExpression.* Elle permet de représenter des éléments du genre : $client.putID(\text{“}T\text{”})$, $x + (y\%z)$, etc. La Figure 53 montre la modélisation en XSD.

Le terme *Operator*

Operator = (unaryOperator | basicOperator | natifOperator | printOperator | Relation | Invalid)

Il y a plusieurs types d'opérateurs dans la grammaire : unaire, basique, natif, d'impression, les procédures attachées (appels de méthode à la java). Il y a également la notion d'invalidité. Nous verrons plus loin les types d'opérateurs en détail. La Figure 54 montre la modélisation en XSD.

Le terme *basicOperator*

Ce sont les opérateurs basiques arithmétiques enrichis par des opérateurs personnalisés qui sont très utilisés. La Figure 55 montre la modélisation en XSD.

```
<xsd:attributeGroup name="ComplexExpression.attlist"/>
 <xsd:group name="ComplexExpression.extend">
        <xsd:sequence>
              <xsd:choice minOccurs="0" maxOccurs="unbounded">
                    <xsd:element ref="Individual"/>
                    <xsd:element ref="Variable"/>
                    <xsd:element ref="SimpleExpression"/>
                    <xsd:element ref="ComplexExpression"/>
                    <xsd:element ref="Invalid"/>
              </xsd:choice>
        </xsd:sequence>
 </xsd:group>
 <xsd:group name="ComplexExpression.content">
        <xsd:sequence>
              <xsd:element ref="Operator"/>
              <xsd:group ref="ComplexExpression.extend"/>
        </xsd:sequence>
 </xsd:group>
 <xsd:complexType name="ComplexExpression.type">
        <xsd:group ref="ComplexExpression.content"/>
        <xsd:attributeGroup ref="ComplexExpression.attlist"/>
 </xsd:complexType>
 <xsd:element name="ComplexExpression" type="ComplexExpression.type"/>
```

Fig. 53 – Définition en XSD du terme ComplexExpression

```
<xsd:attributeGroup name="Operator.attlist"/>
 <xsd:group name="Operator.content">
        <xsd:sequence>
              <xsd:choice>
                    <xsd:element name="unaryOperator" type="unaryOperator"/>
                    <xsd:element name="basicOperator" type="basicOperator"/>
                    <xsd:element name="natifOperator" type="natifOperator"/>
                    <xsd:element name="printOperator" type="printOperator"/>
                    <xsd:element ref="Relation"/>
                    <xsd:element ref="Invalid"/>
              </xsd:choice>
        </xsd:sequence>
 </xsd:group>
 <xsd:complexType name="Operator.type">
        <xsd:group ref="Operator.content"/>
        <xsd:attributeGroup ref="Operator.attlist"/>
 </xsd:complexType>
 <xsd:element name="Operator" type="Operator.type"/>
```

Fig. 54 – Définition en XSD du terme Operator

```
<xsd:simpleType name="basicOperator">
        <xsd:restriction base="xsd:token">
                <xsd:enumeration value="=="/>
                <xsd:enumeration value="!="/>
                <xsd:enumeration value="greaterThan"/>
                <xsd:enumeration value="greaterThanOrEquals"/>
                <xsd:enumeration value="lessThan"/>
                <xsd:enumeration value="lessThanOrEquals"/>
                <xsd:enumeration value="+"/>
                <xsd:enumeration value="-"/>
                <xsd:enumeration value="*"/>
                <xsd:enumeration value="/"/>
                <xsd:enumeration value="%"/>
                <xsd:enumeration value="equals"/>
                <xsd:enumeration value="isEmpty"/>
                <xsd:enumeration value="isNotEmpty"/>
                <xsd:enumeration value="isNull"/>
                <xsd:enumeration value="isNotNull"/>
                <xsd:enumeration value="isTrue"/>
                <xsd:enumeration value="isFalse"/>
                <xsd:enumeration value="equalsDate"/>
                <xsd:enumeration value="greaterThanDate"/>
                <xsd:enumeration value="greaterThanOrEqualsDate"/>
                <xsd:enumeration value="lessThanDate"/>
                <xsd:enumeration value="lessThanOrEqualsDate"/>
        </xsd:restriction>
 </xsd:simpleType>
```

FIG. 55 – Définition en XSD du terme basicOperator

Le terme *natifOperator*

Cet opérateur permet d'utiliser les opérateurs natifs au moteur de règles : insertion, modification, suppression, affichage, etc. La Figure 56 montre la modélisation en XSD.

```
<xsd:simpleType name="natifOperator">
        <xsd:restriction base="xsd:string">
                <xsd:enumeration value="assert"/>
                <xsd:enumeration value="modify"/>
                <xsd:enumeration value="retract"/>
                <xsd:enumeration value="printErr"/>
                <xsd:enumeration value="printOut"/>
        </xsd:restriction>
 </xsd:simpleType>
```

FIG. 56 – Définition en XSD du terme natifOperator

Le terme *Relation*

Operator = ((Relation) , (argument)*, (Variable)*, (method)*, (Classe)*)*

Une *Relation* permet de représenter des procédures attachées. Une procédure attachée est un appel de méthode. Les règles métier utilisent un modèle métier qui dans notre cas sera

concrétisé par un ensemble de classes. Ces classes possèdent un état (un ensemble d'attributs) et un comportement (un ensemble de méthodes). Une procédure attachée permet d'accéder à l'état et au comportement d'une classe. La Figure 57 montre la modélisation en XSD.

```
<xsd:attributeGroup name="Relation.attlist"/>
 <xsd:group name="Relation.content">
        <xsd:sequence>
               <xsd:element ref="Relation" minOccurs="0"/>
               <xsd:element name="argument" type="xsd:NMTOKEN" minOccurs="0"/>
               <xsd:element ref="Variable" minOccurs="0"/>
               <xsd:element name="method" type="xsd:string" minOccurs="0"/>
               <xsd:element ref="Classe" minOccurs="0"/>
        </xsd:sequence>
 </xsd:group>
 <xsd:complexType name="Relation.type" mixed="true">
        <xsd:group ref="Relation.content"/>
        <xsd:attributeGroup ref="Relation.attlist"/>
 </xsd:complexType>
 <xsd:element name="Relation" type="Relation.type"/>
```

FIG. 57 – Définition en XSD du terme Relation

Le terme *AndExpression*

AndExpression = ((wff | Expression | OrExpression | AndExpression | NegationExpression))*

Une *AndExpression* est une expression complexe pour représenter la conjonction (le "et" booléen). Il se compose d'un ensemble qui peut être vide d'une expression bien formée. La Figure 58 montre la modélisation en XSD.

```
<xsd:attributeGroup name="AndExpression.attlist"/>
 <xsd:group name="AndExpression.content">
        <xsd:choice>
               <xsd:element ref="wff"/>
               <xsd:element ref="Expression"/>
               <xsd:element ref="OrExpression"/>
               <xsd:element ref="AndExpression"/>
               <xsd:element ref="NegationExpression"/>
        </xsd:choice>
 </xsd:group>
 <xsd:complexType name="AndExpression.type">
        <xsd:group ref="AndExpression.content" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:attributeGroup ref="AndExpression.attlist"/>
 </xsd:complexType>
 <xsd:element name="AndExpression" type="AndExpression.type"/>
```

FIG. 58 – Définition en XSD du terme AndExpression

Le terme *OrExpression*

OrExpression = ((wff | Expression | AndExpression | OrExpression | NegationExpression))*

Une *OrExpression* est une expression complexe qui permet de représenter la disjonction (le "ou" booléen). Il se compose d'un ensemble qui peut être vide d'une expression bien formée. La Figure 59 montre la modélisation en XSD.

```
<xsd:attributeGroup name="OrExpression.attlist"/>
 <xsd:group name="OrExpression.content">
        <xsd:choice>
                <xsd:element ref="wff"/>
                <xsd:element ref="Expression"/>
                <xsd:element ref="AndExpression"/>
                <xsd:element ref="OrExpression"/>
                <xsd:element ref="NegationExpression"/>
        </xsd:choice>
 </xsd:group>
 <xsd:complexType name="OrExpression.type">
        <xsd:group ref="OrExpression.content" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:attributeGroup ref="OrExpression.attlist"/>
 </xsd:complexType>
 <xsd:element name="OrExpression" type="OrExpression.type"/>
```

FIG. 59 – Définition en XSD du terme OrExpression

Le terme *wff (well formed formula)*

wff = (Expression | AndExpression | OrExpression | NegationExpression)

Une expression *well formed formula* peut être n'importe quelle expression (Expression, AndExpression, OrExpression, NegationExpression). La Figure 60 montre la modélisation en XSD.

```
<xsd:attributeGroup name="wff.attlist"/>
 <xsd:group name="wff.content">
        <xsd:choice>
                <xsd:element ref="Expression"/>
                <xsd:element ref="AndExpression"/>
                <xsd:element ref="OrExpression"/>
                <xsd:element ref="NegationExpression"/>
        </xsd:choice>
 </xsd:group>
 <xsd:complexType name="wff.type">
        <xsd:group ref="wff.content"/>
        <xsd:attributeGroup ref="wff.attlist"/>
 </xsd:complexType>
 <xsd:element name="wff" type="wff.type"/>
```

FIG. 60 – Définition en XSD du terme wff

Le terme *Individual*

Individual = (type, value)

Un *Individual* est une constante (entier, décimal, chaîne de caractères, ...). La Figure 61 montre la modélisation en XSD.

```
<xsd:attributeGroup name="Individual.attlist">
        <xsd:attribute name="type" type="xsd:NCName" use="required"/>
        <xsd:attribute name="value" type="xsd:string" use="required"/>
</xsd:attributeGroup>
<xsd:group name="Individual.content">
        <xsd:sequence/>
</xsd:group>
<xsd:complexType name="Individual.type" mixed="true">
        <xsd:group ref="Individual.content"/>
        <xsd:attributeGroup ref="Individual.attlist"/>
</xsd:complexType>
<xsd:element name="Individual" type="Individual.type"/>
```

FIG. 61 – Définition en XSD du terme Individual

Le terme *Variable*

Variable = (argument | Classe, field)

Un *Variable* permet de représenter une variable simple ou de classe du genre *objet.variable* ou *Objet.variableStatic*. La Figure 62 montre la modélisation en XSD.

```
<xsd:attributeGroup name="Variable.attlist"/>
 <xsd:group name="Variable.content">
        <xsd:sequence>
                <xsd:choice>
                        <xsd:element name="argument" type="xsd:NMTOKEN"/>
                        <xsd:element ref="Classe"/>
                </xsd:choice>
                <xsd:element name="field" type="xsd:string" minOccurs="0"/>
        </xsd:sequence>
 </xsd:group>
 <xsd:complexType name="Variable.type" mixed="true">
        <xsd:group ref="Variable.content"/>
        <xsd:attributeGroup ref="Variable.attlist"/>
 </xsd:complexType>
 <xsd:element name="Variable" type="Variable.type"/>
```

FIG. 62 – Définition en XSD du terme Variable

Le terme *VariableDeclaration*

VariableDeclaration = (ClasseName)

Un *VariableDeclaration* permet de faire une déclaration de variables qui seront utilisées dans les expressions. Pour déclarer une variable il faut lui donner un nom et un type qui sont représentés par *ClasseName*. La Figure 63 montre la modélisation en XSD.

```
<xsd:attributeGroup name="VariableDeclaration.attlist">
        <xsd:attribute name="VariableName" type="xsd:string" use="required"/>
 </xsd:attributeGroup>
 <xsd:group name="VariableDeclaration.content">
        <xsd:sequence>
                <xsd:element ref="ClasseName"/>
        </xsd:sequence>
 </xsd:group>
 <xsd:complexType name="VariableDeclaration.type">
        <xsd:group ref="VariableDeclaration.content"/>
        <xsd:attributeGroup ref="VariableDeclaration.attlist"/>
 </xsd:complexType>
 <xsd:element name="VariableDeclaration" type="VariableDeclaration.type"/>
```

FIG. 63 – Définition en XSD du terme VariableDeclaration

Le terme *NegationExpression*

NegationExpression = (strong | Expression)

Le terme *NegationExpression* permet de représenter une négation. La Figure 64 montre la modélisation en XSD.

```
<xsd:attributeGroup name="NegationExpression.attlist"/>
 <xsd:group name="NegationExpression.content">
        <xsd:sequence>
                <xsd:choice>
                        <xsd:element ref="strong"/>
                        <xsd:element ref="Expression"/>
                </xsd:choice>
        </xsd:sequence>
 </xsd:group>
 <xsd:complexType name="NegationExpression.type">
        <xsd:group ref="NegationExpression.content"/>
        <xsd:attributeGroup ref="NegationExpression.attlist"/>
 </xsd:complexType>
 <xsd:element name="NegationExpression" type="NegationExpression.type"/>
```

FIG. 64 – Définition en XSD du terme NegationExpression

Nous pouvons noter que la grammaire est complexe, ce qui est normale car la complexité d'une grammaire est croissante suivant son degré d'expressivité. Les fonctionnalités

qu'offrent la grammaire ne sont pas toutes nécessaires et utilisées dans *e-Citiz* pour l'instant, car au départ nous voulions la faire en nous soustrayant de toute contrainte et en pensant d'une manière générale à la philosophie de l'approche par règles métier.
Après la mise au point de la grammaire l'étape suivante était de mettre en place le moteur de transformations vers les moteurs de règles sélectionnés (voir Figure 39).

3.10.2 Le moteur de transformations de ERML vers des langages de règles spécifiques

Le principe est, depuis la grammaire ci-dessus, de créer des mécanismes qui permettent d'avoir les règles dans le format natif à un moteur de règles. Il s'agit de transformer le modèle de notre formalisme vers le modèle des moteurs de règles cible. Il existe plusieurs types de transformations comme nous l'exposons au chapitre 4. Pour faire ces translations deux solutions étaient possibles. La première solution était de le faire au niveau objet en utilisant un modèle de la grammaire et un modèle du moteur cible afin de faire une transformation de modèle à modèles. La deuxième solution était de le faire au niveau XML en utilisant XSLT afin de traiter un document XML et de générer un autre document XML, ce qui correspond au modèle de transformation par template. Ne disposant pas d'un modèle approprié pour chaque moteur de règles, notre choix s'est vite porté sur l'utilisation de la transformation par template par XSLT. Le principe de la transformation de modèles par template est de faire une recherche de motif et de remplacer les morceaux manquants par le resultat de la recherche.

Transformation ERML vers Drools

Dans le cadre d'une application concrète de nos recherches pour la suite *e-Citiz*, les moteurs de règles sur lesquels nous nous sommes focalisés en premier étaient Drools et JRules. Sans pour autant rentrer dans les détails de Drools, il faut savoir qu'il utilise un format XML avec plusieurs grammaires suivant le langage utilisé. *e-Citiz* tournant sur un plateforme Java, nous avons choisi le formalisme Java de Drools.

Transformation des opérateurs natifs (natifOperator)

Drools a ses propres opérateurs natifs : assertObject, modifyObject, retractObject, println. Comme le montre la Figure 65 *assertObject* de ERML sera transformé en *drools.assertObject*, *modifyObject* de ERML sera transformé en *drools.modifyObject* et *retractObject* de ERML sera transformé en *drools.retractObject*

```
<xsl:variable name="assertObject">
        <xsl:text>drools.assertObject</xsl:text>
</xsl:variable>
<xsl:variable name="modifyObject">
        <xsl:text>drools.modifyObject</xsl:text>
</xsl:variable>
<xsl:variable name="retractObject">
        <xsl:text>drools.retractObject</xsl:text>
</xsl:variable>
<xsl:variable name="printErrMethod">
        <xsl:text>java.lang.System.err.println</xsl:text>
</xsl:variable>
<xsl:variable name="printOutMethod">
        <xsl:text>java.lang.System.out.println</xsl:text>
</xsl:variable>
```

FIG. 65 – Transformation des opérateurs natifs vers Drools

Transformation du terme Ruleset

Comme le montre la Figure 66, si on rencontre un élément *Ruleset* de ERML on écrit d'abord les commentaires (s'ils existent) sous forme de commentaires XML car Drools n'a pas de balise dans ce sens. Après nous créons l'élément de haut niveau *rule-set* de Drools en prenant le soin de donner les namespaces nécessaires. Puis nous lisons le nom du ruleset dans ERML (*rulesetlabel/name/Individual/@value*) que nous donnons au ruleset pour Drools sous forme d'un attribut qui se nomme *name* pour l'élément *rule-set*. Il faut savoir que la grammaire de Drools est moins riche que ERML, ce qui entraîne une perte d'information qui cependant, n'est pas indispensable au bon fonctionnellement (il s'agit de la date de création, de modification, etc.). Après ceci il ne reste plus qu'à traiter les éléments qui constituent le ruleset.

Transformation du terme Rule

A la Figure 67, dans le cas où au niveau du ERML un commentaire existe, nous le transformons en commentaire XML (comme pour le ruleset, Drools n'a pas de balise pour les commentaires). Dans Drools l'élément de haut niveau est le *rule-set* qui est un ensemble de *rule*. La transformation d'un *Rule* en ERML donne un *rule* en Drools. L'attribut *name* de *rulelabel* est aussi un attribut *name* de *rule*. La propriété est aussi transformée en attribut *salience* pour *rule*. Une fois le traitement de *ruleLabel* terminé, nous passons à la transformation des déclarations de variables, des conditions et enfin des actions (*Variable-Declaration*, *conditions* et *action*").

```
<xsl:template match="/Ruleset">
    <xsl:if test="rulesetlabel/comment">
        <xsl:value-of select="$newline"/>
        <xsl:comment>&#x20;----------&#x20;</xsl:comment>
        <xsl:value-of select="$newline"/>
        <xsl:comment>
            <xsl:value-of select="rulesetlabel/comment"/>
        </xsl:comment>
        <xsl:comment>&#x20;----------&#x20;</xsl:comment>
        <xsl:value-of select="$newline"/>
        <xsl:value-of select="$newline"/>
    </xsl:if>
    <rule-set xmlns="http://drools.org/rules ">
        <xsl:attribute name="name">
            <xsl:value-of select="rulesetlabel/name/Individual/@value"/>
        </xsl:attribute>
        <xsl:apply-templates select="Rule | Fact"/>
    </rule-set>
</xsl:template>
```

Fig. 66 – Transformation du terme Ruleset vers Drools

```
<xsl:template match="Rule | Fact">
    <!-- Rule comment -->
    <xsl:if test="rulelabel/comment">
        <xsl:value-of select="$newline"/>
        <xsl:comment>&#x20;----------&#x20;</xsl:comment>
        <xsl:value-of select="$newline"/>
        <xsl:comment>
            <xsl:value-of select="rulelabel/comment"/>
        </xsl:comment>
        <xsl:comment>&#x20;----------&#x20;</xsl:comment>
        <xsl:value-of select="$newline"/>
        <xsl:value-of select="$newline"/>
    </xsl:if>
    <rule>
        <xsl:attribute name="name">
            <xsl:value-of select="rulelabel/@name"/>
        </xsl:attribute>
        <xsl:if test="rulelabel/salience">
            <xsl:attribute name="salience">
                <xsl:apply-templates select="rulelabel/salience/Individual"/>
            </xsl:attribute>
        </xsl:if>
        <xsl:apply-templates select="VariableDeclaration | conditions | action"/>
    </rule>
</xsl:template>
```

Fig. 67 – Transformation du terme Rule vers Drools

Transformation du terme VariableDeclaration

Un langage de règles est à type de grammaire à contexte lié (contextuel ou sensitif), cela implique que toutes les variables qui sont utilisées dans les règles doivent être déclarées. *VariableDeclaration* de ERML permet de déclarer une variable.

Dans Drools les variables sont déclarées en utilisant l'élément *parameter* qui a un attribut *identifier* qui contient le nom de la variable. Cet élément *parameter* a aussi un sous-élément *class* qui contient le type de la variable comme le montre la Figure 68.

```
<!-- VariableDeclaration matching, apply template conditions-->
  <xsl:template match="VariableDeclaration">
        <xsl:element name="parameter">
              <xsl:attribute name="identifier">
                    <xsl:value-of select="@VariableName"/>
              </xsl:attribute>
              <xsl:element name="class">
                    <xsl:value-of select="ClasseName"/>
              </xsl:element>
              <xsl:apply-templates select="conditions"/>
        </xsl:element>
  </xsl:template>
```

FIG. 68 – Transformation du terme VariableDeclaration vers Drools

Transformation du terme conditions

L'élément *conditions* représente la partie condition de la règle en ERML. Dans Drools une condition est spécifiée avec l'élément *java :condition* (sémantique Java de Drools). Suivant le type d'expression ERML rencontré, un traitement différent sera effectué comme le montre la Figure 69.

Transformation du terme action

La partie action est plus simple que la partie condition dans ERML car elle n'est composée que d'expressions.
Dans Drools l'élément *java :consequence* permet de spécifier la partie conséquence ou action d'une règle. Ici le principe est de boucler sur les différentes expressions qui composent la partie action du formalisme ERML. Après chaque traitement d'une expression nous ajoutons un *";"* (qui est obligatoire dans la sémantique Java de Drools) et si ce n'est pas la dernière expression on va à la ligne (voir Figure 70).

Transformation du terme NegationExpression

Pour l'instant uniquement la négation forte est prise en compte.
Dans la version *beta 18* qui était disponible au moment de l'intégration, Drools n'avait pas

```
<xsl:template match="conditions">
    <xsl:value-of select="$newline"/>
    <xsl:choose>
        <xsl:when test="OrExpression">
            <xsl:apply-templates select="OrExpression"/>
        </xsl:when>
        <xsl:when test="AndExpression">
            <xsl:apply-templates select="AndExpression"/>
        </xsl:when>
        <xsl:when test="Expression">
            <xsl:element name="java:condition" namespace="{$uri}">
                <xsl:apply-templates select="Expression"/>
            </xsl:element>
        </xsl:when>
        <xsl:when test="NegationExpression">
            <xsl:element name="java:condition" namespace="{$uri}">
                <xsl:apply-templates select="NegationExpression"/>
            </xsl:element>
        </xsl:when>
    </xsl:choose>
</xsl:template>
```

Fig. 69 – Transformation du terme conditions vers Drools

```
<!-- Action matching, apply template Expression-->
<xsl:template match="action">
    <xsl:element name="java:consequence" namespace="{$uri}">
        <xsl:value-of select="$newline"/>
        <xsl:for-each select="Expression">
            <xsl:apply-templates select="."/>
            <xsl:text>;</xsl:text>
            <xsl:if test="not(position() = last())">
                <xsl:value-of select="$newline"/>
            </xsl:if>
        </xsl:for-each>
    </xsl:element>
</xsl:template>
```

Fig. 70 – Transformation du terme action vers Drools

un opérateur interne natif qui gère la négation. Cependant il est possible de gérer la négation par "truchement" en utilisant la nature Java de drools (utilisation du " !" qui exprime la négation en Java). Donc traiter une *NegationExpression* revient à faire le traitement de l'expression contenue dans *strong* puis de mettre la négation de Java devant (voir Figure 71). Il faut savoir que depuis la première version release de Drools (la version 2.1) la négation est supportée en natif.

```
<xsl:template match="NegationExpression">
        <xsl:value-of select="$newline"/>
        <xsl:text disable-output-escaping="no">&lt;![CDATA[!</xsl:text>
        <xsl:apply-templates select="strong/Expression"/>
        <xsl:text disable-output-escaping="no">]]&gt;</xsl:text>
</xsl:template>
```

FIG. 71 – Transformation du terme NegationExpression vers Drools

Transformation du terme OrExpression

Comme pour la négation, Drools n'a pas d'opérateur natif qui gère la disjonction (l'opérateur OU booléen). Pour gérer la disjonction nous utilisons toujours le caractère "Java" de Drools avec l'opérateur *"||"*. Comme le montre la Figure 72, le principe est de traiter les différents types d'expression et de les séparer par la disjonction de Java.

Transformation du terme AndExpression

Dans Drools il n'existe pas un opérateur explicite pour spécifier une conjonction. Par défaut dans la condition les expressions qui se suivent sont considérées comme étant dans la même conjonction. Comme le montre la Figure 73, pour les expressions simples (*expression* et *NegationExpression*) on en crée autant d'éléments *java :condition*.

Transformation du terme Expression

Dans Drools, lorsqu'on a une *Expression*, un traitement différent sera effectué suivant un *SimpleExpression* et *ComplexExpression* comme le montre la Figure 74.

Transformation du terme SimpleExpression

SimpleExpression est l'expression la plus simple avec un opérateur et deux opérandes. Comme le montre la Figure 75, dans Drools nous utilisons la notation infixée de Java pour les types de bases.

```
<!-- OrExpression matching, apply template Expression. it must look like <![CDATA[Expression1 ||
  Expression2 ...]]>-->
  <xsl:template match="OrExpression">
        <xsl:element name="java:condition" namespace="{$uri}">
              <xsl:value-of select="$newline"/>
              <xsl:text disable-output-escaping="no">&lt;![CDATA[</xsl:text>
              <xsl:if test="Expression">
                    <xsl:for-each select="Expression">
                          <xsl:apply-templates select="."/>
                          <xsl:if test="not(position() = last())">
                                <xsl:text>||</xsl:text>
                          </xsl:if>
                    </xsl:for-each>
                    <xsl:if test="NegationExpression">
                          <xsl:text>||</xsl:text>
                    </xsl:if>
              </xsl:if>
              <xsl:if test="NegationExpression">
                    <xsl:for-each select="NegationExpression">
                          <xsl:text>!</xsl:text>
                          <xsl:apply-templates select="strong/Expression"/>
                          <xsl:if test="not(position() = last())">
                                <xsl:text>||</xsl:text>
                          </xsl:if>
                    </xsl:for-each>
              </xsl:if>
              <xsl:text disable-output-escaping="no">]]&gt;</xsl:text>
        </xsl:element>
  </xsl:template>
```

Fig. 72 – Transformation du terme OrExpression vers Drools

```
<!-- AndExpression matching, apply template Expression and OrExpression. -->
  <xsl:template match="AndExpression">
        <xsl:for-each select="Expression | NegationExpression">
              <xsl:element name="java:condition" namespace="{$uri}">
                    <xsl:apply-templates select="."/>
              </xsl:element>
        </xsl:for-each>
        <xsl:apply-templates select="OrExpression"/>
        <xsl:apply-templates select="AndExpression"/>
  </xsl:template>
```

Fig. 73 – Transformation du terme AndExpression vers Drools

```
<!-- Expression matching -->
  <xsl:template match="Expression">
        <xsl:apply-templates select="SimpleExpression | ComplexExpression"/>
  </xsl:template>
```

Fig. 74 – Transformation du terme Expression vers Drools

```
<!-- SimpleExpression matching.-->
 <xsl:template match="SimpleExpression">
        <xsl:text>(</xsl:text>
        <xsl:apply-templates select="child::*[2]"/>
        <xsl:apply-templates select="basicOperator"/>
        <xsl:apply-templates select="child::*[3]"/>
        <xsl:text>)</xsl:text>
 </xsl:template>
```

FIG. 75 – Transformation du terme SimpleExpression vers Drools

Transformation du terme ComplexExpression

Comme le cas de *SimpleExpression* on traite les opérateurs suivant le type puis les opérandes qui peuvent être une expression de n'importe quel type (voir Figure 76).

```
<!-- ComplexExpression matching.-->
<xsl:template match="ComplexExpression">
    <xsl:if test="Operator/basicOperator">
    ...
    </xsl:if>
    <xsl:if test="Operator/natifOperator | Operator/printOperator">
        <xsl:apply-templates select="Operator/natifOperator | Operator/printOperator"/>
        <xsl:text>(</xsl:text>
        <xsl:apply-templates select="Individual | Variable | ComplexExpression | SimpleExpression"/>
        <xsl:text>)</xsl:text>
    </xsl:if>
    <xsl:if test="Operator/unaryOperator">
        <xsl:apply-templates select="Operator/unaryOperator"/>
        <xsl:text> </xsl:text>
        <xsl:apply-templates select="Individual | Variable | ComplexExpression | SimpleExpression"/>
    </xsl:if>
    <xsl:if test="Operator/Relation">
        <xsl:apply-templates select="Operator/Relation"/>
        <xsl:text>(</xsl:text>
        <xsl:for-each select="Individual | Variable | ComplexExpression | SimpleExpression">
            <xsl:apply-templates select="."/>
            <xsl:if test="not(position() = last())">
                <xsl:text>, </xsl:text>
            </xsl:if>
        </xsl:for-each>
        <xsl:text>)</xsl:text>
    </xsl:if>
</xsl:template>
```

FIG. 76 – Transformation du terme ComplexExpression vers Drools

Transformation du terme basicOperator

ERML dispose d'opérateurs natifs basiques qui sont listés à la Figure 77. Au moment de la génération, ces opérateurs seront transformés en leur équivalent Java. Par exemple l'opérateur *lessThan* sera transformé en *"<"* et *isNotNull* sera transformé en *" !=null"* et ainsi de suite.

L'étape suivante après la mise en place de notre formalisme de règles et du moteur de

```
<!-- basicOperator matching-->
  <xsl:template match="basicOperator">
        <xsl:choose>
              <xsl:when test="(../basicOperator = 'lessThan')">
                    <xsl:text disable-output-escaping="no">&lt;</xsl:text>
              </xsl:when>
              <xsl:when test="(../basicOperator = 'lessThanOrEquals')">
                    <xsl:text disable-output-escaping="no">&lt;=</xsl:text>
              </xsl:when>
              <xsl:when test="(../basicOperator = 'greaterThan')">
                    <xsl:text disable-output-escaping="no">&gt;</xsl:text>
              </xsl:when>
              <xsl:when test="(../basicOperator = 'greaterThanOrEquals')">
                    <xsl:text disable-output-escaping="no">&gt;=</xsl:text>
              </xsl:when>
              <xsl:when test="(../basicOperator = 'equals')">
                    <xsl:text disable-output-escaping="no">.equals(</xsl:text>
              </xsl:when>
              <xsl:when test="(../basicOperator = 'isEmpty')">
                    <xsl:text disable-output-escaping="no">.equals("")</xsl:text>
              </xsl:when>
              <xsl:when test="(../basicOperator = 'isNotEmpty')">
                    <xsl:text disable-output-escaping="no">.equals("")</xsl:text>
              </xsl:when>
              <xsl:when test="(../basicOperator = 'isNull')">
                    <xsl:text disable-output-escaping="no"> == null</xsl:text>
              </xsl:when>
              <xsl:when test="(../basicOperator = 'isNotNull')">
                    <xsl:text disable-output-escaping="no"> != null</xsl:text>
              </xsl:when>
              <xsl:when test="(../basicOperator = 'isTrue')">
                    <xsl:text disable-output-escaping="no"> == true</xsl:text>
              </xsl:when>
              <xsl:when test="(../basicOperator = 'isFalse')">
                    <xsl:text disable-output-escaping="no"> == false</xsl:text>
              </xsl:when>
              <xsl:when test="(../basicOperator = 'equalsDate' or ../basicOperator =
'greaterThanDate' or ../basicOperator = 'greaterThanOrEqualsDate' or ../basicOperator =
'lessThanDate' or ../basicOperator = 'lessThanOrEqualsDate')">
                    <xsl:text disable-output-escaping="no">.compareTo(</xsl:text>
              </xsl:when>
              <xsl:otherwise>
                    <xsl:value-of select="../basicOperator"/>
              </xsl:otherwise>
        </xsl:choose>
  </xsl:template>
```

Fig. 77 – Transformation du terme basicOperator vers Drools

transformations était de les intégrer dans l'architecture d'*e-Citiz*. Dans la partie implémentation au chapitre 8 nous expliquons comment cette intégration a été effectuée.

3.11 Conclusion

Il existe de plus en plus de moteurs de règles, l'approche par règles métier est de plus en plus utilisée et les règles sont cruciales dans l'architecture du Web Sémantique. Le besoin d'un formalisme pour représenter et échanger des règles n'est pas nouveau et existe depuis les premières applications de l'Intelligence Artificielle. Depuis longtemps des tentatives de standardisation de langage de règles existent mais n'ont pas eu le succès attendu, car bien souvent un seul langage tente d'adresser tous les types de règles et de moteurs qui existent. Pour le cas de RuleML, non seulement son langage veut adresser tous les types de règles mais également prendre en compte aussi bien le chaînage avant qu'arrière. Comme nous l'avons dit et montré, CommonRules et RuleML ont beaucoup influencé les autres tentatives. Depuis quelque temps l'OMG et le W3C s'intéressent au formalisme de règles. Nous avons vu que l'OMG était en stade de finalisation dans le processus de standardisation du *Semantics of Business Vocabulary and Business Rules* (SBVR) qui ne s'intéresse qu'aux experts métier et à la couche CIM du MDA. Le principe du SBVR est intéressant mais comme tout standard, il faudrait qu'il y ait des outils pour faciliter sa mise en œuvre et ainsi son adoption. Il faudrait que le SBVR passe rapidement du mode contemplentif au mode productif, c'est-à-dire il faudrait que depuis les spécifications que l'on arrive à générer des règles métier dans la couche PIM puis PSM afin qu'elles soient exécutables par des moteurs de règles. En plus l'étape de définition du vocabulaire, qui sera utilisé dans la partie règles métier, doit fortement se basée sur les travaux dans le domaine des ontologies car en fin de compte, ces vocabulaires ne sont ni plus ni moins que des ontologies (taxonomie).
Au niveau de l'OMG nous avons également vu le *Production Rule Representation* (PRR) qui est une standardisation d'un formalisme de règles plus bas que le SBVR et uniquement ciblé règles de production, qui sont les plus utilisées dans l'industrie. Nous pensons, bien que n'étant qu'au début du processus de standardisation, le PRR peut être facilement mis en place car plus ciblé.
Comme tentative de standardisation, nous avons également vu le *Rule Interchange Format* (RIF) du W3C qui bénéficie de l'expérience de RuleML car beaucoup d'acteurs du groupe de travail étaient également dans celui de RuleML. Le RIF est prévu pour se dérouler sur 2 phases et la première devrait se terminer en 2008 avec la proposition d'un langage central qui sera enrichi d'extensions dans la deuxième phase. Les concepteurs de RIF ont adopté une bonne approche qui est d'avoir un langage central et des sous-langages par types de règles.
Nous avons fini par présenter notre langage de règle *E-Citiz Rule Markup Language* (ERML). Lors de l'état de l'art, très rapidement nous nous sommes rendus compte que RuleML était bien trop compliqué pour nos besoins, car notre contexte était celui de règles de production en chaînage avant. Cependant pour la mise en place de ERML nous nous

sommes fortement basés sur les travaux de RuleML et CommonRule. ERML répond à tous les besoins de *e-Citiz* et plus (par exemple la prise en compte de plusieurs types de gestion des conflits, les deux négations, etc). Lors de l'établissement de ERML nous avons voulu le faire de manière générale de sorte qu'on puisse l'utiliser pour tout système de règles de production en chaînage avant.
Le langage ERML est couplé à un moteur de transformations basé sur XSLT pour transformer une instance du langage vers un autre langage de moteur de règles.
Dans la partie implémentation nous verrons le modèle objet du langage et comment il est utilisé en production dans *e-Citiz*.
Nous allons maintenant voir, dans la partie suivante, l'approche par ingénierie dirigée par des modèles que nous avons suivie pour faire nos modèles et appliquer des services dessus.

Deuxième partie

MDA, MDD, MDE ou Ingénierie Dirigée par les Modèles (IDM)

Chapitre 4

L'Ingénierie Dirigée par les Modèles

Sommaire

Ce chapitre présente l'approche de l'Ingénierie Dirigée par les Modèles (IDM) ou Model Driven Architecture (MDA) de l'Object Management Group (OMG) [59].
Une nouvelle approche consiste à penser que l'ingénierie logiciel guidée par les modèles est l'avenir des applications [60]. Il est temps d'élaborer des applications à partir de modèles et, surtout, de faire en sorte que ces modèles soient au centre du cycle de vie de ces applications, autrement dit qu'ils soient productifs. L'approche MDA, qui a été introduite par l'OMG en 2001, préconise l'utilisation massive des modèles.
Nous tenons à préciser que les termes *Model Driven Engineering (MDE)* et *Model Driven Development (MDE)* désignent la même chose que MDA ou IDM et que c'est purement à titre commercial qu'ils existent car le terme *Model Driven Architecture* est déposé et protégé par l'OMG (cependant dans ce document nous continuerons à l'utiliser).

4.1 Introduction

En dépit de l'assurance que toute application jamais construite ait pour finalité d'être intégrée et mise à jour, bien encore des développeurs de logiciels l'ignorent et font la conception uniquement en ayant comme seul objectif le cahier des charges devant eux. Beaucoup d'applications sont construites ignorant la réalité des changements constants.
Le but du "design" n'est pas seulement d'aboutir à un système plus facile à développer, à intégrer et à maintenir, mais aussi nous devons être en mesure d'automatiser ne serait-ce qu'une partie de la construction. Cela permet de prendre un modèle, défini dans un standard comme UML [61], MOF [52] ou CWM [62], le mettre dans un ordinateur et automatiser la construction du stockage de données et d'autres fondations de l'application. Et même mieux, quand on aura besoin de connecter ces "bâtiments" à d'autres, il sera possible d'automatiser la génération de briques et des translateurs basés sur le modèle défini ; et lorsqu'un nouveau type de matériau plus résistant sera connu, nous pourrons le re-générer pour la nouvelle infrastructure. Ceci est la promesse du Model Driven Architecture [59] : permettre la définition d'applications et de modèles de données, compréhensibles par les machines, flexibles à long terme avec les avantages suivants :

- implémentation : de nouvelles infrastructures pourront être intégrées ;
- intégration : maintenant en plus de l'implémentation, le modèle (design) existe, une partie de l'intégration pourra alors être automatisée et s'adapter aux nouvelles infrastructures ;
- maintenance : l'existence du modèle dans une forme machine-readable, permet aux développeurs d'avoir un accès direct à la spécification du système, facilitant ainsi la maintenance ;
- test et simulation : du moment que les modèles développés peuvent être utilisés pour générer du code, on peut également les valider depuis des besoins et faire des tests sur différentes infrastructures. On peut même s'en servir pour faire des simulations sur le comportement du futur système.

La modélisation de systèmes changera irrévocablement la manière de créer des logiciels [60]. En réalité tous les logiciels sont modélisés actuellement. Malheureusement, la plupart des modèles sont fugaces, créés juste avant le design des données ou logiciels qui l'implémentent. Ce modèle n'existe que dans la tête du programmeur et sera perdu à jamais. Ceci malgré le besoin d'intégrer ce que l'on est entrain de construire, de ce qui a déjà été construit, et de ce qui le sera.
En réalité le MDA n'est rien d'autre qu'une autre étape dans le développement d'un logiciel. L'automatisation de la création d'un logiciel depuis un modèle n'est qu'un autre niveau de compilation. Les modèles offrent plusieurs avantages et le plus important est qu'ils procurent différents niveaux d'abstraction, facilitant la gestion de la complexité inhérente aux applications dans le but d'une meilleure compréhension [60]. Leurs représentations graphiques facilitent la collaboration entre les différentes entités qui travaillent ensemble sur un même projet informatique.

4.2 Le Model Driven Architecture

4.2.1 Aperçu

Le Model-Driven Architecture ou l'Ingénierie Dirigée par les Modèles se base sur une idée bien connue et très vielle : "la séparation de la spécification d'une opération du système, des détails sur la manière de la réaliser et la manière dont ce système utilise les capacités de sa plateforme". Le MDA fournit une approche et des outils pour :

- spécifier un système indépendamment de la plateforme qui le supporte ;
- spécifier des plateformes ;
- choisir une plateforme particulière pour le système ;
- transformer la spécification du système pour une plateforme particulière.

Les 3 principaux objectifs du MDA sont la portabilité, l'interopérabilité et la réutilisabilité.

4.2.2 Les concepts basiques

Dans cette sous-section, nous allons rapidement passer en revue les concepts de base qui sont utilisés dans le cadre du MDA.

Système

Les concepts du MDA sont présentés en terme de système existant ou planifié. Ce système peut tout inclure : un programme, un système d'ordinateur simple, une combinaison de parties de divers systèmes, une fédération de systèmes, etc.

Modèle

Le modèle d'un système est une description ou spécification de ce système et de son environnement pour un certain but. Un modèle est souvent présenté sous la forme d'une combinaison de dessins et de textes. Le texte peut être écrit avec un langage de modélisation ou avec un langage naturel.

L'orienté modèle

MDA est une approche qui augmente le pouvoir du modèle dans un développement de système. Il est orienté modèle parce qu'il fournit un moyen d'utiliser les modèles afin de diriger le cours de la compréhension, de la construction, du déploiement, des opérations, de la maintenance et de la modification. Le principe clé de MDA est l'utilisation de modèles aux différentes phases du cycle de développement d'une application. Plus précisément, MDA préconise l'élaboration de modèles d'exigences (Computation Independent Model - CIM), d'analyse et de conception (Platform Independent Model - PIM) et de code (Platform Specific Model - PSM) [60]. Comme le montre la Figure 78, MDA possède une architecture à 4 couches : le niveau $M0$ contenant les entités à modéliser (ici les applications informatiques), le niveau $M1$ contenant les différents modèles de l'application informatique, le niveau $M2$ contenant les différents métamodèles qui ont été utilisés pour définir les modèles, et le niveau $M3$, contenant le métamétamodèle qui a permis de définir uniformément les métamodèles.

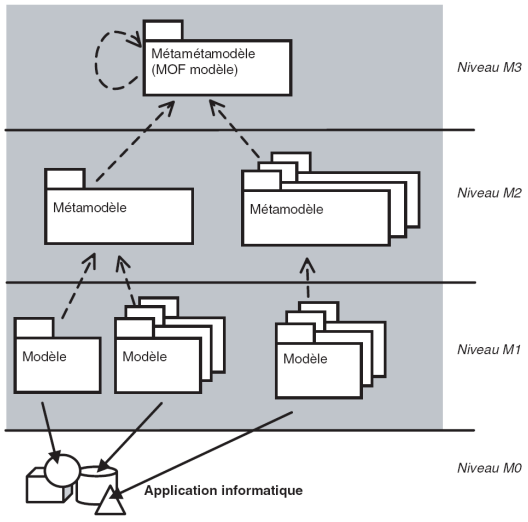

FIG. 78 – Architecture à quatre couches de MDA

The Computation Independent Model ou modèle d'exigence (CIM)

La première chose à faire lors de la construction d'une nouvelle application est bien entendu de spécifier les exigences du client. Bien que très en amont, cette étape doit fortement bénéficier des modèles.
Un CIM ou modèle indépendant de la programmation ou encore modèle d'exigences est une vue du système depuis un point de vue indépendant du système. Un CIM ne montre pas les détails de la structure du système. Un CIM est parfois appelé modèle et vocabulaire du domaine qui est familier aux experts de ce domaine.
Nous supposons que l'utilisateur premier, l'expert métier, ne connaît rien en modèles et artefacts utilisés pour réaliser les fonctionnalités du CIM. Le CIM joue un rôle très important dans le rapprochement entre ceux qui sont experts du domaine et des besoins d'un coté et ceux qui sont experts en design et constructions de l'autre.
Il est important de noter qu'un modèle d'exigences ne contient pas d'information sur la réalisation de l'application ni sur les traitements.
Avec UML, un modèle d'exigences peut se résumer en un diagramme de cas d'utilisation. Ces exigences contiennent en effet les fonctionnalités fournies par l'application (cas d'utilisation) ainsi que les différentes entités qui interagissent avec elle (acteurs) sans apporter d'information sur le fonctionnement de l'application [60].
Dans une optique plus large, un modèle d'exigences est considéré comme une entité complexe, constituée entre autres d'un glossaire (ontologie), de définitions de processus métier, des exigences et des cas d'utilisation ainsi que d'une vue systématique de l'application [60].
Si MDA n'émet aucune préconisation quant à l'élaboration des modèles d'exigences, des travaux sont en cours pour ajouter à UML les concepts nécessaires pour couvrir cette phase en amont.

Le modèle indépendant de la plateforme ou modèle d'analyse et de conception abstraite (PIM)

Une fois le modèle d'exigences réalisé, le travail d'analyse et de conception peut commencer. Dans l'approche MDA, cette phase utilise elle aussi un modèle. Un PIM ou modèle d'analyse et de conception abstraite exhibe un degré spécifique d'indépendance vis-à-vis de la plateforme de telle sorte qu'il soit approprié à un nombre différent ou similaire de plateformes.
Une technique très commune pour atteindre l'indépendance vis-à-vis de la plateforme est de définir un modèle système pour une machine virtuelle neutre. Une machine virtuelle est définie comme étant un ensemble de parties et de services (communications, planification, nommage, etc.) qui est définie indépendamment de toute plateforme spécifique et qui est ensuite réalisée de manière spécifique à une plateforme sur différentes plateformes.
MDA considère que les modèles d'analyse et de conception doivent être indépendants de toute plateforme d'implémentation, qu'elle soit J2EE, .NET, PHP, etc. En n'intégrant les détails d'implémentation que très tard dans le cycle de développement, il est possible de

maximiser la séparation des préoccupations entre la logique de l'application et les techniques d'implémentation [60].
Quelque soit le ou les langages utilisés, le rôle des modèles d'analyse et de conception est d'être pérennes et de faire le lien entre le modèle d'exigences et le code de l'application. Ce modèle doit être productif puisqu'il constitue le socle de tout le processus de génération de code défini par MDA. La productivité des PIM signifie qu'ils doivent être riches de suffisamment d'information pour qu'une génération automatique de code soit envisagée [60].

Le modèle spécifique à la plateforme ou modèle de code ou de conception concrète (PSM)

Une fois les modèles d'analyse et de conception abstraite réalisés, le travail de génération de code peut commencer. Cette phase, la plus délicate du MDA, doit elle aussi utiliser les modèles. Elle inclut l'application des patrons de conception technique. MDA considère que le code d'une application peut être facilement obtenu à partir de modèles de code. La différence principale entre un modèle de code et un modèle d'analyse et de conception (PIM) réside dans le fait que le modèle de code est lié à une plateforme d'exécution. Dans le jargon MDA, ces modèles de code sont appelés des PSM (Platform Specific Model).
Un PSM ou modèle de code ou de conception concrète combine les spécifications d'un PIM aux détails (langage d'implémentation) dans le cadre de l'utilisation d'une plateforme spécifique par un système [63].
Pour résumer, les modèles de code servent essentiellement à faciliter la génération de code à partir d'un modèle d'analyse et de conception abstraite (PIM). Ils contiennent toutes les informations nécessaires à l'exploitation d'une plateforme d'exécution, comme les informations permettant de manipuler les systèmes de fichiers ou les systèmes d'authentification.
Pour élaborer un des modèles de code, MDA propose, entre autres, l'utilisation de profils UML. Un profil UML est une adaptation de UML à un domaine particulier (profil UML EJB etc.) [60].

Le modèle de plateforme

Un modèle de plateforme fournit un ensemble de concepts techniques, représentant les différents types de parties qui forment une plateforme ainsi que les services fournis par cette plateforme.

La transformation de modèles

C'est le processus de la conversion d'un modèle en un autre du même système (voir Figure 79). Les transformations de modèles préconisées par MDA sont essentiellement les transformations CIM vers PIM et PIM vers PSM (inter-couches) ou PIM vers PIM et PSM vers PSM (intra-couches). La génération de code à partir des PSM n'est quant à elle pas

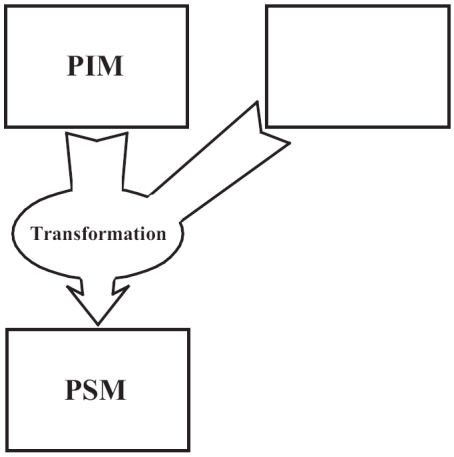

Fig. 79 – La transformation de modèle

considérée comme une transformation de modèle à part entière. MDA envisage aussi des transformations inverses : code vers PSM, PSM vers PIM et PIM vers CIM [60]. Cette transformation inverse est appelée *"reverse ingineering"* [64].
Après avoir transformé les PIM vers des PSM, on effectue manuellement différentes opérations de raffinement sur les PSM afin qu'ils puissent servir à la génération de code. En effet les transformations PIM vers PSM ne permettent pas d'avoir un modèle directement exploitable, il est nécessaire de raffiner encore ce modèle afin d'effectuer la dernière opération, qui est la génération de code.

Implémentation

Une implémentation est une spécification, qui fournit toute l'information nécessaire à la construction d'un système et de sa mise en fonction.

4.2.3 Comment mettre en œuvre MDA ?

Pour mettre en œuvre MDA, l'approche à suivre consiste à mettre en place un CIM puis un PIM. Et afin de pouvoir produire un PSM, il pourrait être nécessaire de procéder à un marquage du modèle PIM. A partir du PSM on peut produire un modèle de code et enfin du code. Comme le montre la Figure 80 la transformation dans le sens inverse, appelée reverse engineering, est aussi envisageable.

Mise en œuvre CIM

Les besoins du système sont modélisés dans un CIM, qui décrit la situation dans laquelle le système sera utilisé. Un tel modèle est appelé modèle domaine ou modèle métier. Un CIM

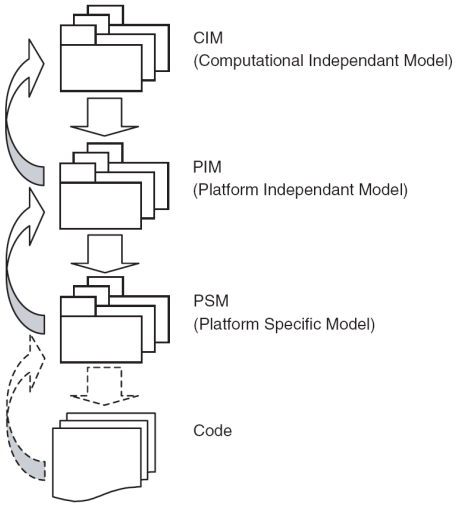

FIG. 80 – Les étapes du Model Driven Architecture

est un modèle du système qui montre l'environnement dans lequel le système doit opérer. Dans une approche IDM d'un système, les besoins du CIM doivent pouvoir provenir du PIM et du PSM qui les implémentent, et vice versa.

Mise en œuvre PIM

Le PIM décrit le système mais ne montre pas les détails d'utilisation d'une quelconque plateforme. Théoriquement, le PIM devrait pouvoir être, en partie, automatiquement généré depuis le CIM.

Modèle de plateforme

C'est l'architecte qui choisit la plateforme (il peut y en avoir plusieurs) qui permettra d'implémenter le système. L'architecte doit avoir à portée de main le modèle de la plateforme. Souvent ce modèle est en premier lieu sous forme d'un logiciel ou d'un manuel ou même dans la tête de l'architecte. les modèles MDA se baseront sur des modèles de plateformes détaillés par exemple modélisés avec du UML, OCL et stockés sous fourme de répertoire MOF (que nous verrons plus loin dans ce chapitre).

Transformation (Mapping)

Un mapping MDA fournit des spécifications permettant, principalement, la transformation d'un PIM dans un PSM d'une plateforme particulière, d'un PIM vers un autre PIM et d'un PSM vers un autre PSM. Le modèle de la plateforme déterminera la nature de la transformation. Une transformation peut être effectuée manuellement ou avec l'aide d'un

ordinateur (automatiquement).
La transformation d'un modèle est le processus de conversion d'un modèle en un autre modèle du même système [65]. Dans le cadre d'une transformation PIM vers PSM, la donnée de départ de la transformation est le PIM et le résultat est le PSM et l'enregistrement de la transformation.

Langage de transformation

Une transformation est spécifiée en utilisant un langage pour décrire une transformation d'un modèle vers un autre [66]. Cette description peut être un langage naturel, un algorithme ou encore un langage de transformation de modèle (MOF) [67]. La qualité supérieure d'un langage de transformation est la portabilité [59].

4.2.4 Les approches de la transformation de modèles

Cette sous-section présente les approches qui sont utilisées pour faire des transformations de modèles (de PIM vers du PSM, de PIM vers du PIM, de PSM ver du PSM, etc).

Marquage de modèle

Dans la transformation par marquage de modèle, l'architecte marque des éléments du PIM pour indiquer les mappings qui seront utilisés lors de la transformation de ce PIM en PSM. La Figure 81 étend le pattern MDA pour montrer, avec plus de détails, une des

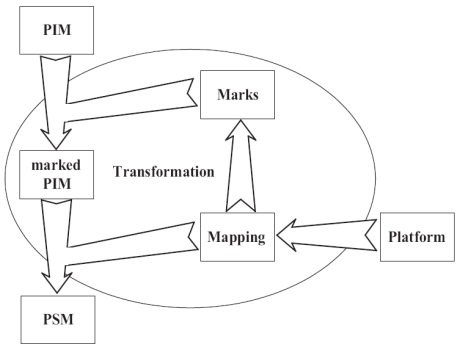

Fig. 81 – Transformation par marquage de modèle

manières dont une transformation est effectuée.
Comme le montre la figure un ensemble de marques est utilisé pour marquer le PIM, ce qui donne un PIM "enrichi". Ce PIM "enrichi" ou marqué est utilisé pour être transformé vers le modèle de sortie.

Transformation par métamodèles

Le principe ici est de se baser sur le métamodèle du PIM et de celui du PSM pour spécifier la transformation de modèle du PIM vers le PSM (voir Figure 82). Cette approche consiste à appliquer les concepts de l'ingénierie des modèles aux transformations des modèles elles-mêmes [68]. L'objectif est de modéliser les transformations de modèles et de rendre les modèles de transformation pérennes et productives et d'exprimer leur indépendance vis-à-vis des plateformes d'exécution. Cette approche est actuellement au stade de la recherche. Le standard MOF2.0 QVT (Query View Transformation) [69] en cours de définition à l'OMG a pour objectif de définir le métamodèle permettant l'élaboration des modèles de transformation de modèles. Actuellement, seuls quelques prototypes supportent cette approche. Nous verrons plus en détail QVT dans la section 4.2.5.

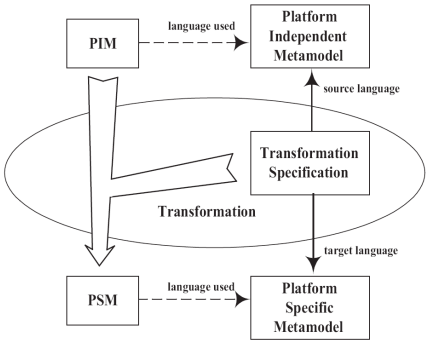

FIG. 82 – Transformation par métamodèle

Transformation par modèles ou par programmation

La transformation par modèles est basée sur le même principe que la transformation par mètamodèles sauf qu'ici, on se base directement sur les modèles pour spécifier la transformation de modèle du PIM vers le PSM (voir Figure 83). De manière concrète, cette approche consiste à utiliser les langages de programmation orientée objet. L'idée est de programmer une transformation de modèles de la même manière que l'on programme n'importe quelle application informatique. Ces transformations sont donc des applications informatiques qui ont la particularité de manipuler des modèles. Cette approche est la plus utilisée car elle est très puissante et fortement outillée [60].

Transformation par template ou pattern

Ce type de transformation étend les transformation par mètamodèles et modèles en les enrichissant de templates (voir Figure 84). Le principe est de définir les canevas des modèles cibles souhaités en y déclarant des paramètres. Ces paramètres seront substitués par les

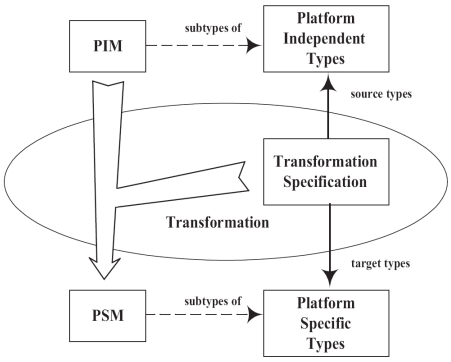

Fig. 83 – Transformation par modèle ou par programmation

informations contenues dans les modèles sources. On appelle ces canevas des “modèles cibles” ou des “modèles templates”. L’exécution d’une transformation consiste à prendre un modèle template et à remplacer ses paramètres par les valeurs d’un modèle source. Cette approche nécessite un langage particulier permettant la définition de modèles templates. De tels langages sont en cours d’élaboration et n’ont pas encore la maturité des langages de programmation orientée objet utilisés dans l’approche par modèles ou par programmation.

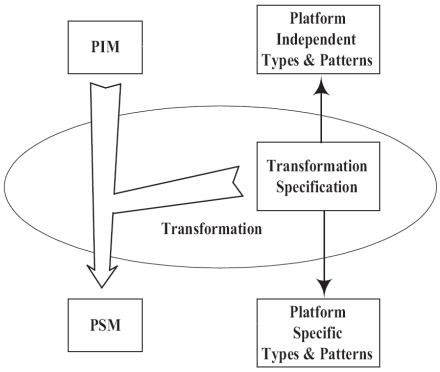

Fig. 84 – Transformation par template ou pattern

Transformation par fusion de modèles

La transformation par fusion de modèles est une autre forme de transformation par template ou pattern. Le principe est de combiner deux modèles pour avoir un troisième modèle (voir figure 87).

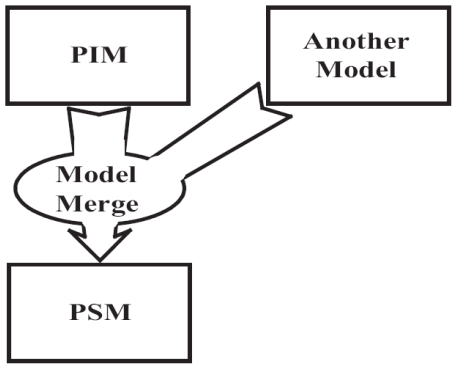

FIG. 85 – Transformation par fusion de modèles

Transformation par informations additionnelles

Le principe de la transformation par informations additionnelles est de rajouter des informations au PIM pour guider la transformation. Ce type de transformation peut servir à étendre la transformation par pattern ou template en fournissant des informations utiles pour le marquage du PIM.

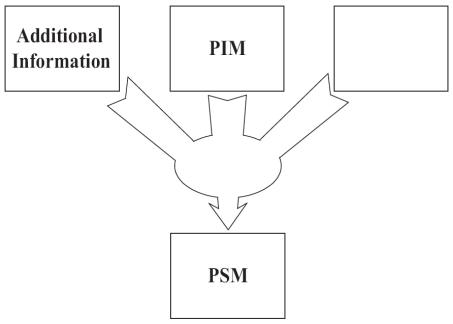

FIG. 86 – Transformation par informations additionnelles

4.2.5 Technologies de modélisation

Nous avons vu que MDA préconisait l'élaboration de différents modèles : CIM, PIM et PSM. En réalité, MDA est beaucoup plus général et préconise de modéliser n'importe quelle information nécessaire au cycle de développement des applications [70]. Nous pouvons donc trouver des modèles de test, de déploiement, de plateforme, etc. Afin de structurer cet ensemble de modèles, MDA définit la notion de formalisme de modélisation [60].

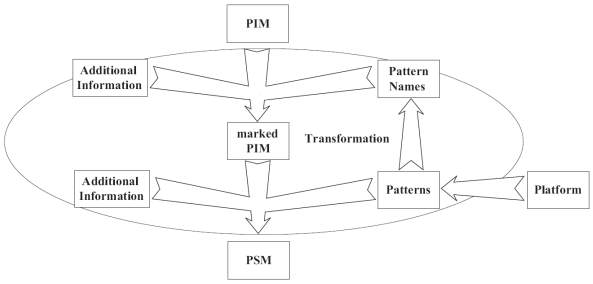

Fig. 87 – Transformation par informations additionnelles

Le formalisme de modélisation MOF (Meta Object Facility)

Un formalisme de modélisation est un langage qui permet d'exprimer des modèles. Chaque modèle a donc un formalisme de modélisation [70]. Les modèles d'exigences (CIM) ont leur propre formalisme, qui est différent du formalisme permettant d'exprimer les modèles de conception et d'analyse abstraite (PIM). Le formalisme préconisé par l'OMG pour le PIM est UML [59].
Un formalisme définit les concepts ainsi que les relations entre eux nécessaires à l'expression de modèles.
Les notions de modèles et de formalismes de modélisation ne sont pas suffisantes pour mettre en œuvre MDA. Il est aussi très important de pouvoir exprimer des liens de traçabilité ainsi que des transformations entre modèles. Pour pouvoir faire cela il est nécessaire de pouvoir travailler non seulement au niveau des modèles mais aussi au niveau des formalismes de modélisation. Il faut exprimer des liens entre les concepts des différents formalismes. Par exemple, il faut pouvoir exprimer que le concept de classe UML doit être transformé dans le concept de classe Java.
Pour cela, MDA préconise de modéliser les formalismes de modélisation eux-mêmes. L'objectif est de disposer d'un formalisme permettant l'expression de modèles de formalismes de modélisation. Dans le jargon de MDA, un tel formalisme est appelé métaformalisme, et les modèles qu'il permet d'exprimer sont appelés des métamodèles. Nous pouvons donc faire une analogie entre métamodèles et formalismes de modélisation (voir Figure 88).
La question qui se pose est de savoir s'il est possible de construire un métamétaformalisme permettant d'exprimer des métaformalismes, voire un métamétaméta...formalisme, et ainsi de suite. MDA réponds que seul trois niveaux sont nécessaires : le modèle, le formalisme de modélisation, aussi appelé métamodèle, et le métaformalisme appelé métamétamodèle [59]. Pour enrayer la montée dans les niveaux méta, MDA fait en sorte que le métaformalisme soit à lui-même son propre formalisme. Un métamétaformalisme n'est dès lors plus nécessaire [60].
MOF pour Meta Object Facility qui est un métamétamodèle permet de spécifier des formalismes de modélisation. Un métamodèle MDA est un diagramme de classe élaboré selon

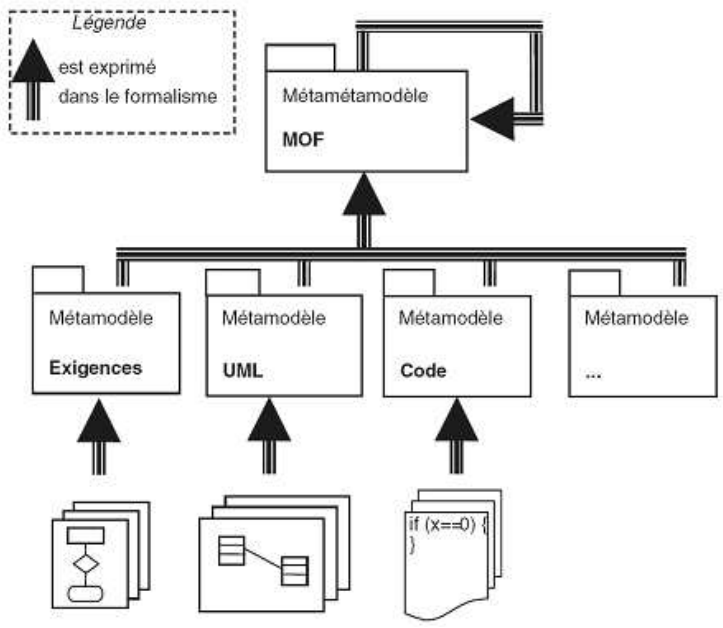

FIG. 88 – Modèle, métamodèle et métamétamodèle

les règles du standard MOF. Le grand avantage apporté par MOF est que tous les métamodèles sont structurés de la même manière. Une structuration commune des métamodèles permet de définir des mécanismes génériques. Parmi ces mécanismes génériques il y a le mécanisme XMI (XML Metadata Interchange) [71] qui permet de construire des grammaires XML à partir de n'importe quel métamodèle afin de stocker les modèles instances au format XML [60].
La Figure 89 illustre une partie de ce que pourrait être un diagramme de classes représentant MOF 1.4.

Modélisation de la transformation de modèles avec QVT (Query View Transformation)

Nous avons déjà mentionné que les transformations de modèles étaient au cœur de MDA et qu'il était important de les modéliser. Pour ce faire, l'OMG a élaboré le standard MOF2.0 QVT (Query View Transformation). Le principe de QVT est d'utiliser l'idée même du MDA, à savoir la modélisation, pour faire la transformation de modèles [68]. Ce standard définit le métamodèle permettant l'élaboration de modèles de transformation. La transformation d'UML vers la plateforme Java (voir Figure 90), par exemple, est élaborée sous la forme d'un modèle de transformation conforme au métamodèle MOF2.0 QVT. QVT permet de modéliser des transformations bi-directionnelles, d'avoir une traçabilité des transformations, d'avoir des transformations réagissant comme des transactions avec des *"commit et rollback"*.

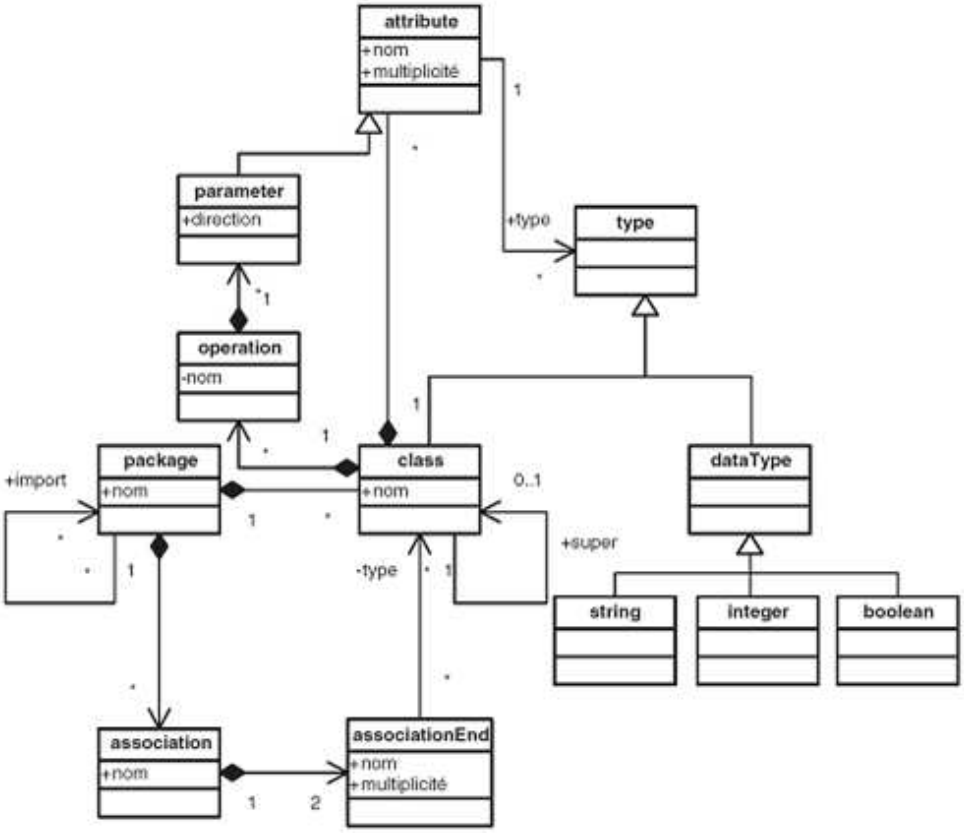

Fig. 89 – Représentation de MOF 1.4 sous forme de diagramme de classe

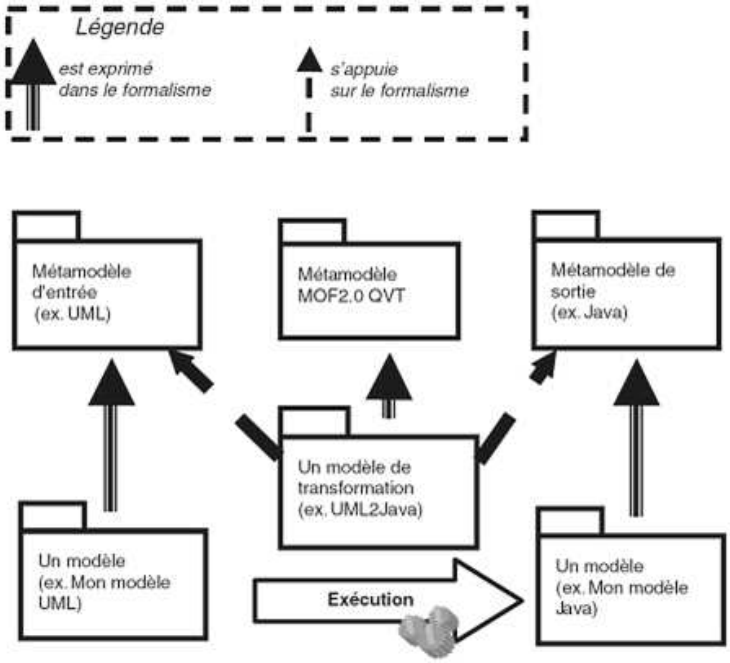

FIG. 90 – Transformation de modèles UML vers du Java en utilisant QVT

La spécification QVT [69] dépend des spécifications de MOF2.0 [69] et OCL2.0 [50]. QVT est composé de 3 métamodèles et langages de transformation qui sont liés : *Relations*, *Operational Mappings* et *Core*. QVT a une nature hybride déclarative et impérative. QVT déclarative a une architecture à 2 niveaux comme le montre la Figure 91. Le métamodèle *Relations* est déclaratif et permet la création de recherche de motifs complexes et de canevas (templates) objets. Les traces entre les éléments de modèle intervenant dans une transformation sont créées implicitement. La sémantique des relations sont définies en utilisant une combinaison de langage naturel (Anglais) et des prédicats de premier ordre. Nous avons également le métamodèle *Core* qui a été défini en utilisant un minimum d'extensions de MOF et OCL. Le package *Core* définit la sémantique des concepts déclaratifs.

En faisant une comparaison par analogie à la machine virtuelle dans l'architecture Java, on peut dire que le langage *Core* est le Byte Code Java, les sémantiques définies par *Core* joue le rôle de la spécification de la machine virtuelle Java, le langage *Relations* le rôle du langage Java et enfin la transformation standard de *Relations* vers *Core* est comme la spécification du compilateur Java qui produit le Byte Code.
En plus des langages déclaratifs *Relations* et *Core*, QVT fournit 2 mécanismes pour une utilisation impérative des transformations avec *Operational Mappings* et *Black-Box MOF Operation*. Le principe du métamodèle *Black-Box MOF Operation* est de permettre aux algorithmes complexes d'être codés dans n'importe quel langage de programmation modélisable avec MOF, permettant ainsi l'utilisation des DSL (Domain Specific Labrairies).
Comme le montre la Figure 92 la spécification définit 3 packages principaux, un pour chaque

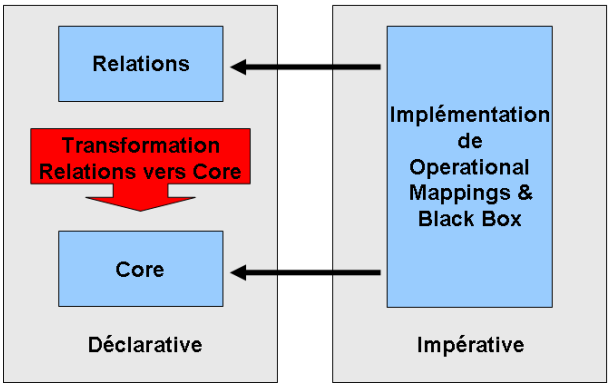

Fig. 91 – Relations entre les métamodèles de QVT

langage défini : *QVTCore*, *QVTRelation* et *QVTOperational*. Le package *QVTBase* définit les structures communes pour les transformations. *QVTRelation* utilise en plus les expressions de canevas définies dans *QVTTemplateExp*. *QVTOperational* hérite de *QVTRelation* vu qu'il utilise le même framework de traces et il utilise aussi les expressions impératives définies dans le package *ImperativeOCL*. Tout QVT dépends du package *EssentialOCL* de OCL2.0 et les les packages de langage de EMOF.
Il faut savoir qu'au moment où nous écrivions ces lignes (Juillet 2007), la spécification de QVT n'était pas encore terminée chez l'OMG.

4.3 Conclusion

L'approche par ingénierie dirigée par des modèles préconise l'utilisation massive de modèles à toutes les étapes du cycle de développement d'une application. MDA fournit les outils pour faciliter l'implémentation, l'intégration, la maintenance, le test et la simulation d'un système.
L'approche MDA, avec son architecture à 4 couches, en permettant la génération automatique de code, accélère la mise en place d'un système. L'approche MDA existe depuis 2001, ce qui fait d'elle une jeune approche. Bien qu'il y ait de plus en plus d'outils pour mettre en œuvre cette approche, il en manque toujours. La plus part des outils qui existent s'intéressent uniquement aux couches PIM et PSM. Il n'y a encore rien concernant la couche CIM. Une autre limitation de l'approche est qu'elle ne dit absolument rien sur la sémantique et s'intéresse uniquement au contenu.
L'approche par ingénierie dirigée par des modèles nous a permis de faire des modèles qui sont indépendants de toute plateforme. Elle nous a également permis de mettre en place des opérations de transformation et de validation de modèles de manière automatique.
L'importance de la sémantique dans les règles métier nous a amené à nous intéresser au Web Sémantique, que nous allons voir au chapitre prochain. L'approche par règles métier

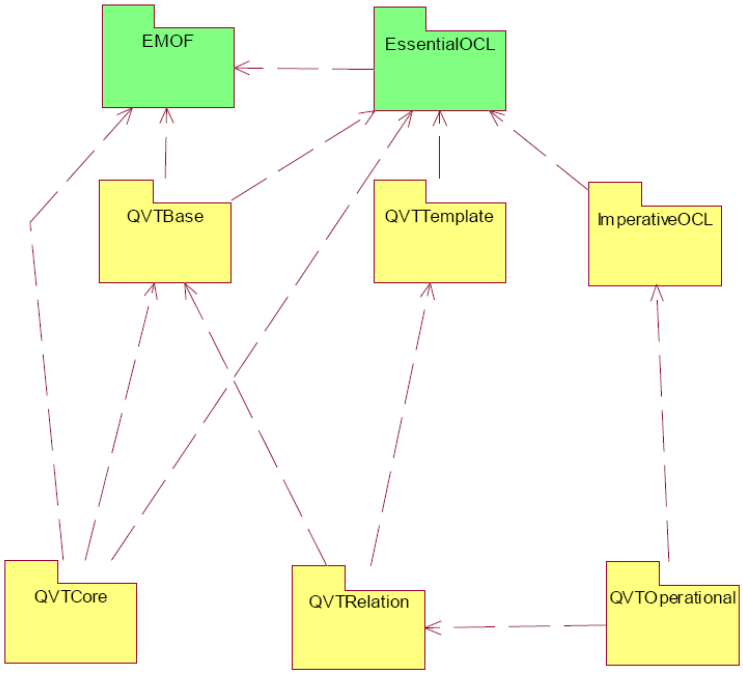

FIG. 92 – Dépendances des packages dans la spécification de QVT

et l'approche MDA sont complémentaires car la dernière permet de s'affranchir des détails techniques alors que la première vise la séparation de la logique métier de la logique système (technique).

Troisième partie

Web Sémantique et Raisonnement

Chapitre 5

Le Web Sémantique

Sommaire

5.1 Introduction

La simplicité d'utilisation du Web à l'état actuel a un prix : il est très simple de s'y perdre ou de tomber sur un document qui est loin de faire parti de ses critères de recherche. L'objectif du Web Sémantique [72] est de mettre en place des standards et technologies de telle sorte que les machines (agent logiciel) puissent "comprendre" plus d'information sur le Web. Ainsi les résultats de recherche seront plus exacts et il y aura une meilleure facilité d'intégration de données et de navigation etc.
Le Web à l'état actuel, n'est pleinement exploitable que par les humains. En effet la majeure partie du contenu du Web a été conçue jusqu'à aujourd'hui pour une exploitation par l'homme et non pour une manipulation sensée par des programmes informatiques. Certes les ordinateurs peuvent traiter convenablement le contenu des pages Web, mais en général, ils ne disposent pas de moyens pour en saisir le sens (ex : est-ce que le document est une facture, un bouquin, etc). D'où la nécessité d'un nouveau paradigme pour faire un Web pour les machines (programmes ou des agents logiciels).
Avec le Web sémantique on pourra sémantiquement ajouter de riches descriptions à toute ressource. Par exemple nous pourrons directement demander un document ayant pour méta donnée : "Mouhamed Diouf comme auteur ".
Dans ce chapitre nous allons voir ce qu'est le Web Sémantique et ses principes de base.

5.2 Qu'est-ce que c'est que le Web Sémantique ?

Le Web Sémantique n'est pas un autre Web à part, mais seulement une extension de l'actuel, dans lequel les informations sont données ainsi que leur sens bien défini, permettant ainsi mieux aux hommes et aux machines de pouvoir travailler en coopération [73, 72].
Le mot *sémantique* implique sens. Pour le Web Sémantique, *sémantique* veut dire que le sens des données dans le Web peut être connu, non seulement par les humains, mais aussi par les machines. Actuellement, la majeure partie de la compréhension (sens) dans le Web est inférée par l'Homme qui lit les pages web et les étiquette des liens, ou qui écrit des logiciels spécialisés. Le Web Sémantique est une vision dans la quelle aussi bien les machines que les hommes peuvent chercher, lire, comprendre et utiliser des données à travers le Web pour accomplir des tâches profitables pour les utilisateurs. Nous avons l'habitude d'utiliser des logiciels pour accomplir des interactions sur le Web. En effet les gens parcourent le Web pour faire des achats sur des sites, utiliser des moteurs de recherche, lire des libellés de liens web avant de choisir lequel suivre. Il serait plus efficace et moins long pour une personne, si elle pouvait lancer une procédure qui pourrait, suivant des critères définies, faire tout ce que nous avons cité précédemment (lancer une recherche, comprendre les étiquettes des liens, suivre un lien). Le rôle du Web Sémantique est de permettre de telles procédures.
Ceci permettra d'avoir des résultats plus exacts lors d'une recherche, de savoir quand intégrer des informations de différentes sources et quelles informations comparer. Pour atteindre son but, le Web Sémantique propose des techniques pour associer des informations

descriptives, sémantiquement riches, à toute ressource (on parle de méta-données). Le Web Sémantique est plus une vision qu'une technologie.

5.3 Les principes du Web Sémantique

En ajoutant des méta-données sur des documents, nous pouvons, par exemple, faire la recherche de documents ayant dans les méta-données "Tim B. Lee" comme auteur. Nous pouvons également faire des requêtes du genre : "afficher tous les documents ayant comme catégorie *document de recherche*". Le Web Sémantique ne fournit pas des URIs uniquement aux documents, comme le fait le Web actuel, mais il le fait aussi pour les concepts et les relations qui définissent et mettent en relation les ressources du Web. Dans l'exemple ci-dessus en donnant un identifiant unique aux personnes et en définissant les concepts "auteur" et "document de recherche", on fournit de manière claire, une manière d'identifier la personne et quelle est la relation entre elle et un document particulier. Ces différentes informations descriptives peuvent être combinées et intégrées dans le but d'avoir une connaissance plus globale d'un domaine donné.
Le Web Sémantique a pour objectif d'ajouter une information spécifique au Web pour aider à l'automatisation des services, de découvrir, corréler et inférer des connaissances qui sont plausibles. Nous allons maintenant voir les principes qui régissent le Web Sémantique.

5.3.1 Principe 1 : Identification par URI

Les personnes, les places et les choses du monde réel peuvent être référencées dans le Web Sémantique en utilisant un ensemble d'identifiants. Dans le Web Sémantique les identifiants sont des URIs (Uniforme Resource Identifier) comme le montre l'exemple ci-dessous.
Exemple : La *personne* dont l'email est : *mailto :mdiouf@genigraph.fr.*
La *ville* dont le site officiel est *http ://www.toulouse.fr*

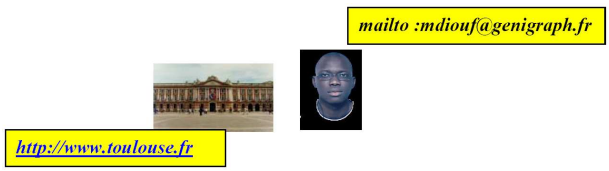

Fig. 93 – Identification par URI

5.3.2 Principe 2 : Typage des ressources et liens Web

Le Web actuel est composé de ressources et de liens (voir Figure 94). Les ressources sont des documents web conçus pour la conception humaine et ne contiennent pas de descriptions expliquant en quoi elles sont utilisées et quelles relations maintiennent t-elles avec d'autres documents. Quand une personne peut à coup d'œil savoir si un tel document est une facture, un roman ou encore un article de recherche, une machine ignore souvent ce genre d'information. De la même manière, une personne peut connaître le lien entre deux ressources en lisant le libellé du lien hyper-texte qu'il y a entre elles. Alors que pour une machine il est impossible de procéder de la sorte. L'un des objectifs du Web Sémantique est de typer les ressources et les liens du web (voir Figure 94). Il faut aussi savoir que le

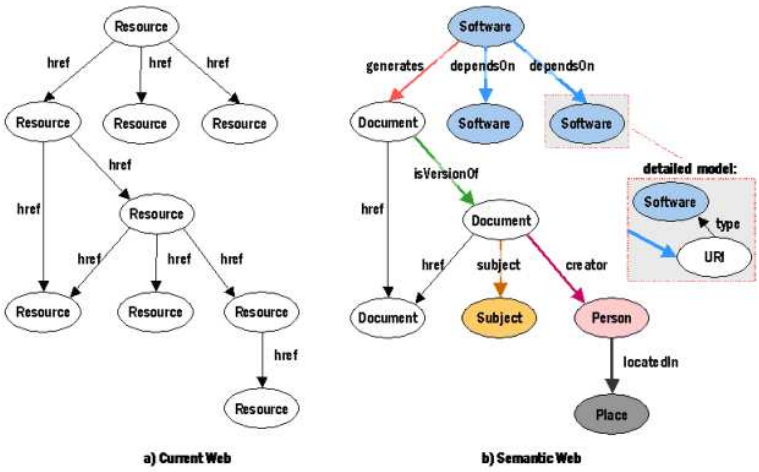

FIG. 94 – Typage des ressources et liens du Web

typage de ressources et de liens était déjà présent dans la proposition originale du Web faite par Tim Berners Lee [74] comme le montre la Figure 95.

5.3.3 Principe 3 : Information partielle tolérée

Lorsqu'une ressource référencée n'est plus disponible, le fonctionnement doit continuer. Le web actuel est scalable car une information incomplète est déjà tolérée comme le montre la Figure 96. Le Web Sémantique continuera de tolérer une information partielle.

5.3.4 Principe 4 : Une vérité absolue n'est pas indispensable

Un des problèmes du Web est que n'importe qui peut dire n'importe quoi sur n'importe quoi posant ainsi la question : dans quelle mesure peut-on croire à quelque chose ? Dans ce sens, les signatures numériques apportent un élément de réponse en permettant de savoir

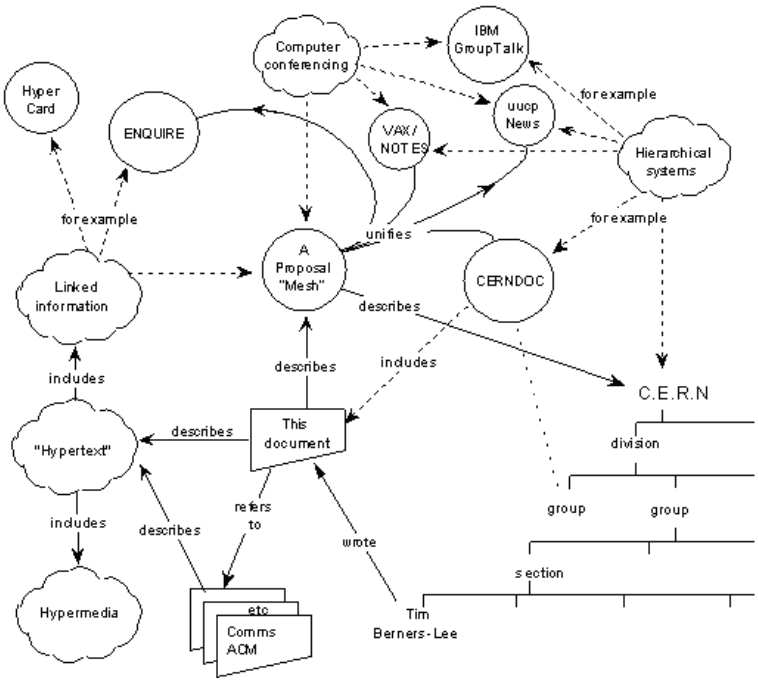

Fig. 95 – La vision du web de Tim B. Lee en 1989

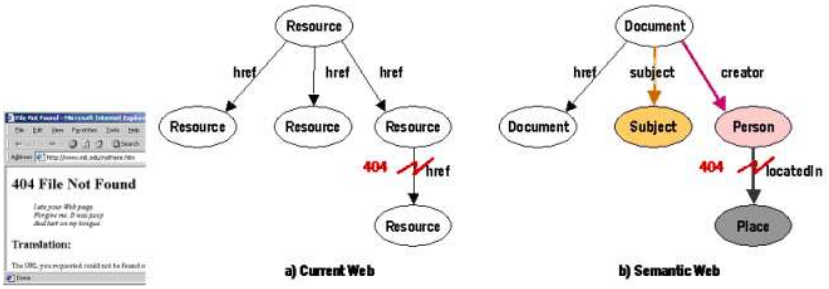

Fig. 96 – Information partielle tolérée

qui à dit quoi, quand et en quel honneur.
La confiance dans le Web est la même que chez chez l'homme : on croit plus à nos amis, un peu moins à leurs amis et moins aux amis de ces derniers (voir Figure 97).
Tout dans le web n'est pas vrai et le Web Sémantique ne pourra pas y changer quelque chose. La véracité ou fiabilité doit être validée par l'application qui utilise l'information issue du Web. Ces applications doivent décider en quoi croire en se basant sur le contexte des déclarations : qui a dit quoi, quand et en qualité de quoi.

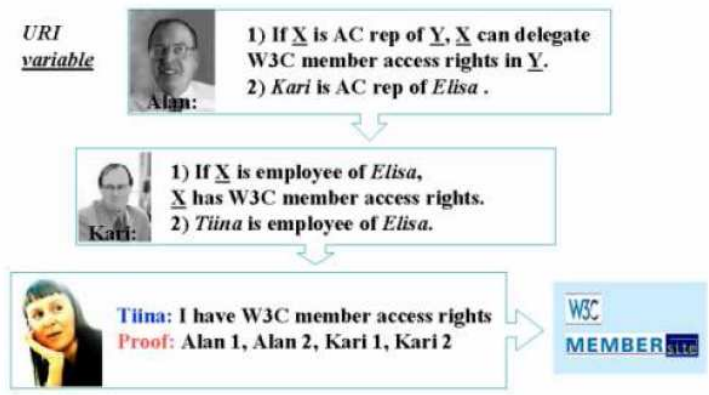

FIG. 97 – La confiance dans le Web [2]

5.3.5 Principe 5 : Supporter l'évolution

Il est commun de voir des concepts similaires définis par différents groupes de personnes à des endroits différents ou même par le même groupe à des moments différents. Et souvent il est intéressent de combiner des données du Web utilisant ces concepts. On doit pouvoir combiner une nouvelle information et une ancienne sans modifier cette dernière comme le montre la Figure 98. Le Web Sémantique pour cela utilise des conventions descriptives qui peuvent être étendues comme c'est le cas de la compréhension humaine. En plus ces conventions facilitent la combinaison effective de différents travaux de diverses communautés, même si elles utilisent des vocabulaires différents. Le Web Sémantique fournit des outils communautaires qui peuvent être utilisés pour lever des ambiguïtés et clarifier des inconsistances.

5.3.6 Principe 6 : Architecture minimaliste

Le Web Sémantique se veut de ne pas être compliqué pour une adaptation simple et rapide. Il rend les choses simples ... simples et celles compliquées possibles. Le but est de ne pas standardiser plus que nécessaire.

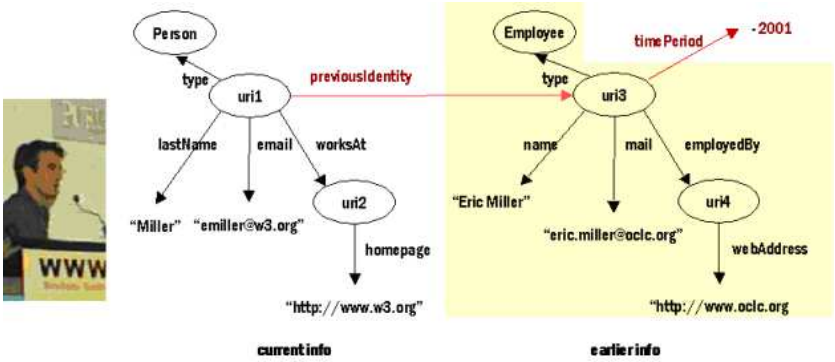

Fig. 98 – Support de l'évolution [2]

5.4 Technologies utilisées pour mettre en œuvre le Web Sémantique

5.4.1 Les couches du Web Sémantique

Le W3C (World Wide Web Consortium) étant leader dans la mise en place de technologies pour le Web, est en charge de mettre en place et de promouvoir le Web Sémantique. Les principes du Web Sémantique sont implémentés en utilisant les couches des technologies web et des standards.

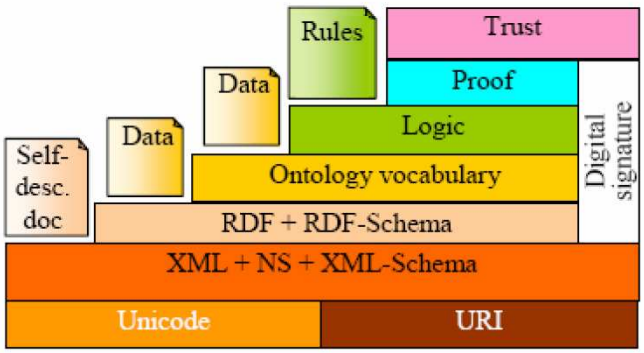

Fig. 99 – Les couches du Web Sémantique

- La couche Unicode et URI permet l'utilisation de caractères internationaux et permet d'en donner un sens pour identifier un objet dans le Web Sémantique.
- XML pour Extensible Markup Language est, depuis 1998, utilisé pour s'échanger des données sur le Web et est devenu incontournable. Ainsi, la couche XML + namespace + xmlschema permet de pouvoir intégrer dans le Web Sémantique, des documents basés sur les standards XML.

- Avec la couche RDF [75] pour Resource Description Framework et RDF Schema il est possible décrire des objets avec les URI et de définir des vocabulaires qui puissent être référencés par les URI. RDF est un langage flexible qui est capable de décrire toutes sortes d'information et de méta-données. Nous parlerons plus amplement de RDF et RDFS au chapitre suivant.
- La couche Ontology permet de supporter une évolution des vocabulaires en définissant des relations entre plusieurs concepts. En effet sur cette couche nous avons des langages qui peuvent être utilisés pour définir des vocabulaires et les relations entre des termes dans un contexte spécifique. Nous reviendrons également sur les ontologies au chapitre suivant.
- La couche Digital Signature permet de détecter une altération des documents. Cette couche permet de vérifier de manière non ambiguë qu'une certaine personne est bien l'auteur d'un certain document. Cette technologie basée sur les notions de clé publique et privée est bien connue et largement utilisée à l'heure actuelle.
- La couche logique permet de faire des raisonnements logiques permettant d'établir la consistance et l'exactitude des données mais aussi d'inférer des conclusions qui ne sont pas définies explicitement.
- La couche proof exécute les règles et évalue en accord avec la couche Trust quand est ce que la preuve est tangible ou pas. Cette couche permet d'avoir une traçabilité d'un raisonnement logique.

Chaque couche de l'architecture du Web Sémantique se base sur la couche d'en-dessous. Chaque couche devient progressivement plus spécialisée et plus complexe que celle du bas. Les couches ne sont pas interdépendantes, ce qui permet de les développer et les rendre opérationnelles de manière séparée et indépendante.
Cependant restreindre le Web Sémantique à cette infrastructure serait trop limitatif. Ce sont les applications développées sur celle-ci qui font et feront vivre cette vision et qui seront, d'une certaine manière, la preuve du concept. Bien sûr, de manière duale, le développement des outils, intégrant les standards du Web Sémantique, doit permettre de réaliser plus facilement et à moindre coût des applications ou des services développés aujourd'hui de manière souvent ad-hoc. Nous allons voir certains de ces outils et autres projets du Web Sémantique.

5.5 Quelques outils pour le Web Sémantique

5.5.1 Le projet Annotea

Le projet Annotea [76, 77, 78] est un projet du W3C pour faire de l'annotation web. Les annotations sont des commentaires, des notes, des explications ou d'autres types de remarques externes qui peuvent être associés à tout document web ou une partie sans que cela ne soit nécessaire de modifier le document.
Une fois qu'on a un document on peut charger les annotations le concernant depuis un

serveur afin de les exploiter.
On utilise un schema basé sur RDF pour décrire les annotations comme des méta-données. Xpointer est utilisé pour localiser des annotations dans un document annoté.
Les annotations sont stockées dans des serveurs qui peuvent être interrogés.[76]. Il existe des clients d'annotation (editor/browser) comme Amaya. La Figure 100 montre un aperçu de l'éditeur Annotea.

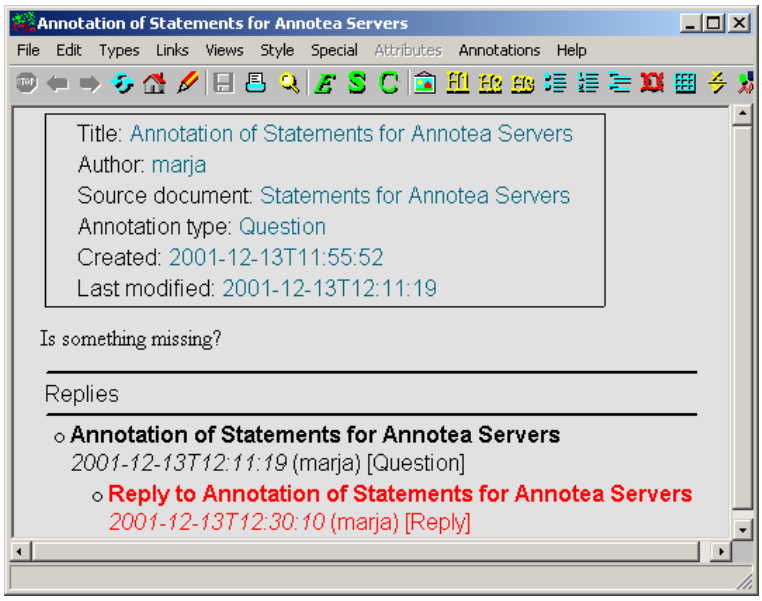

Fig. 100 – l'éditeur d'annotation Annotea

5.5.2 Amaya Editor/Browser

Amaya est avant tout un navigateur et éditeur de pages web. Amaya [79, 80] est le fruit d'une collaboration entre W3C et l'INRIA. Amaya est un client Annotea pour lire des annotations se basant sur RDF, XLink [81] et XPointer [82]. Comme nous l'avons déjà dit, les annotations peuvent être localement stockées dans des serveurs. Lorsqu'un document est chargé, Amaya interroge les serveurs d'annotation pour avoir les méta-données liées à ce document. Amaya utilise XPointer pour décrire où les annotations doivent être attachées sur un document. Avec cette technique, il devient possible d'annoter tout document Web de manière indépendante, sans qu'il ne soit nécessaire de l'éditer. Finalement Amaya utilise une icône, pour signaler une annotation, sur laquelle il faut cliquer pour voir l'annotation concernée (comme le montre la Figure 102). Ces icônes sont positionnées en utilisant XLink.

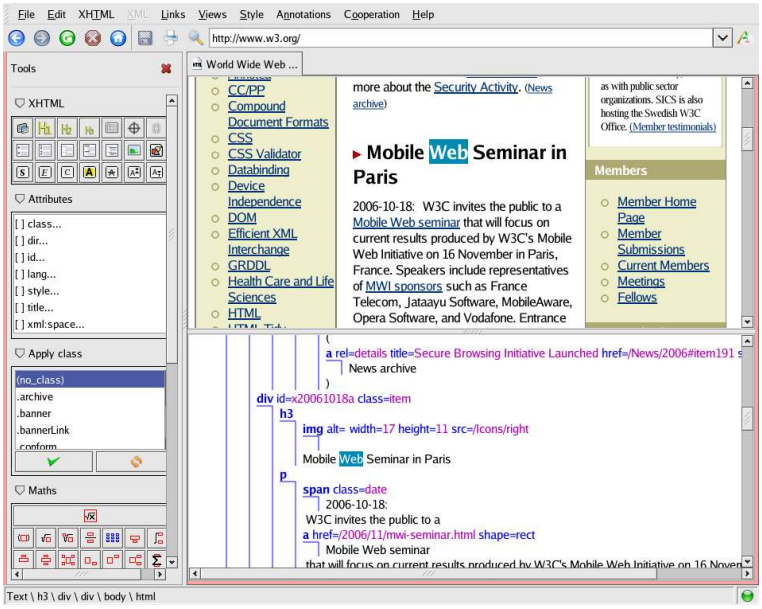

FIG. 101 – Amaya Editor/Browser

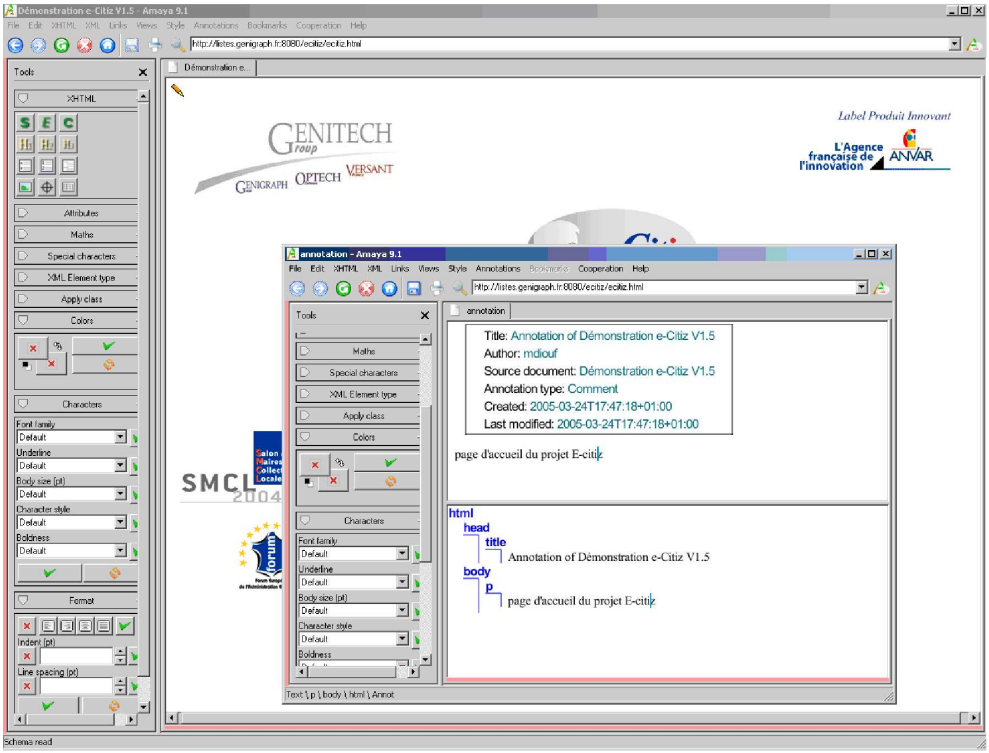

FIG. 102 – Amaya et annotation

5.5.3 Music Brainz

Music Brainz est une méta-base de données de musique qui offre des informations concernant un CD ou une piste mp3. Un lecteur de CD peut utiliser Music Brainz pour identifer un CD. On peut aussi l'utiliser pour "tagger" des mp3. La Figure 103 montre le schéma RDF de Music Brainz. Music Brainz qui s'appelait avant CDIndex, est l'un des premiers services Web Sémantique [83]. Le principe n'est pas nouveau et depuis longtemps, a été proposé par d'autre système comme Internet Compact Disc DataBase (CDDB). La nouveauté qu'apporte Musin Brainz est que c'est un projet ouvert (open source) qui est alimenté par les utilisateurs et se base entièrement sur RDF. Music Brainz fournit également une bibliothèque (API) qui peut être utilisée par d'autres programmes.

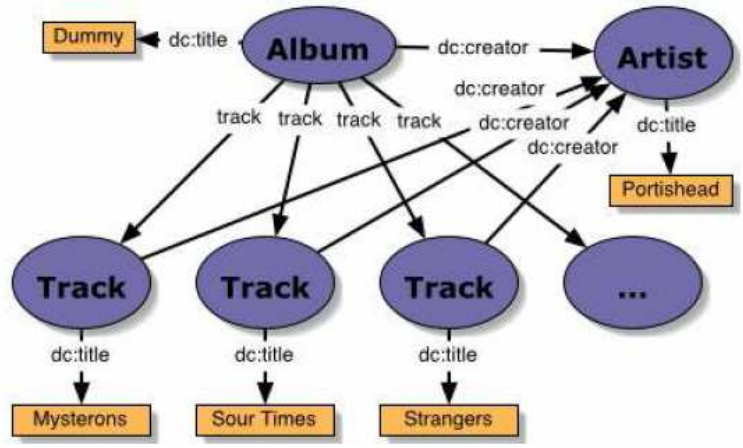

Fig. 103 – Schéma RDF de Music Brainz

5.5.4 JENA

Jena [84, 85] est un framework en Java qui permet de construire des applications Web Sémantique. Il fournit un environnement de programmation pour RDF, RDFS et OWL (voir la Figure 104). Il inclut également un moteur de règles d'inférence.
Le framework contient :

- Un api RDF.
- Un api pour la lecture et l'écriture de RDF en RDF/XML, N3, et N-Triples.
- Un api OWL.
- Un système de persistance.
- RDQL, un langage de requête pour RDF.

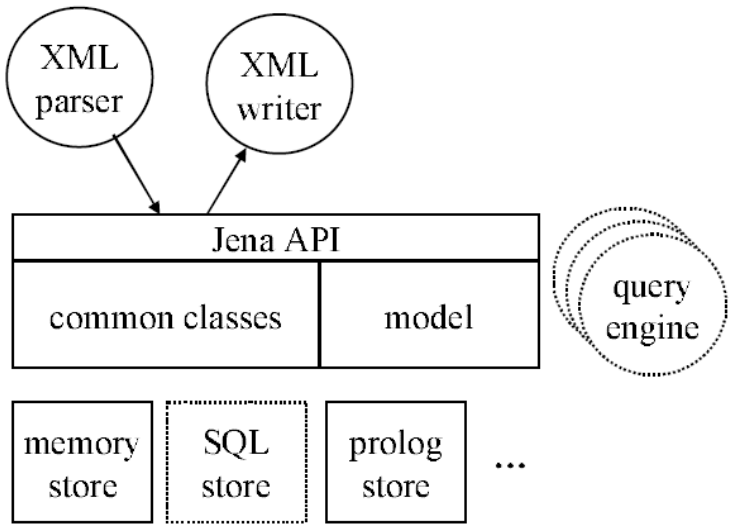

FIG. 104 – Architecture de Jena

5.5.5 Joseki

Joseki [86] est un API Web pour interroger et mettre à jour des modèles RDF distants à travers le Web. Les modèles ont des URL et peuvent être utilisés dans une requête HTTP. Joseki fait parti de l'outillage de Jena pour RDF. Le système comprend :

- Une API cliente pour manipuler des modèles RDF distants.
- Un serveur RDF qui peut être embarqué dans un application isolée ou dans une application web qui sera déployée dans un serveur d'application.

5.5.6 CWM

CWM (Closed World Machine) [87] est un processeur de données pour le Web Sémantique, au même titre que sed, awk pour les fichiers textes ou encore XSLT pour XML. C'est un "raisonneur" en *chaînage avant* qui peut être utilisé pour faire des requêtes, des vérifications, des transformations et des filtrages d'information. Il a été écrit en python.

5.5.7 Piggy Bank

Piggy Bank [88] est un projet rentrant toujours dans le sens de proposer des applications qui mettent en œuvre le Web Sémantique dans le but de prouver que son infrastructure est "implémentable". Piggy Bank est une extension pour navigateur Web qui permet à l'utilisateur de manipuler le contenu du Web Sémantique dans le Web actuel. En effet il permet à un navigateur de pouvoir charger les données sémantique, locales ou distantes, liées à une page lors de son chargement dans un navigateur classique. Dans le cas ou il n'existerait

pas de données sémantique pour une page, Piggy Bank peut les créer en restructurant le contenu de la page Web en format Web Sémantique. Ainsi Piggy Bank propose une manière évolutive et graduelle aux utilisateurs du Web actuel de pouvoir évoluer vers la vision Web Sémantique. Piggy Bank propose à ses utilisateurs qui le souhaitent de pouvoir partager les données sémantique qu'ils ont pu récolter en les chargeant sur un serveur Web Sémantique (Semantic Bank) ainsi, les utilisateurs pourront partager des données. Ceci ressemble beaucoup aux principes d'annotations mises en œuvre par Annotea et Amaya.

5.6 Conclusion

Le Web Sémantique n'est pas pour le futur, il est déjà là. Son objectif est de rendre le Web actuel aussi bien exploitable par les humains que par les machines, qui deviennent des agents intelligents. Son principe est de mettre en place une structure permettant de donner un sens au contenu du Web, qui est certes sensé (pour les humains), en typant non seulement les ressources mais aussi les liens entre elles dans le Web. Cette infrastructure a une architecture en couche qui, plus on monte vers le haut est de moins en moins aboutie. Voir le Web Sémantique comme étant une simple infrastructure est plutôt limitateur car, le Web Sémantique est plus une vision qu'une technologie. Cependant pour donner de la crédibilité à cette vision il faut des applications implémentées, en se basant sur son infrastructure, pour faciliter son adoption de manière fonctionnelle par les utilisateurs. Le Web Sémantique n'a pas pour objectif de remplacer le Web actuel mais plutôt de l'étendre. Pour une adaptation rapide il faudrait que les standards proposés pour le Web Sémantique soient simples et faciles d'utilisation, car le succès du Web actuel est surtout dû au fait que le langage HTML est simple et diversement utilisé. En ajoutant de la sémantique dans le Web, le Web Sémantique laisse présager d'applications autonomes évoluées avec des mécanismes inférentiels puissants pour faciliter et alléger le travail de l'homme en utilisant le raisonnement.
L'étude du Web Sémantique nous as permis de voir comment utiliser ses principes pour rajouter du "sens" dans nos modèles. Maintenant, ce qui restait à voir c'était, une fois sémantiquement riches, comment raisonner de manière automatique sur nos modèles.
Au chapitre suivant, nous allons parler d'ontologies et de raisonnement dans le cadre du Web Sémantique.

Chapitre 6

Ontologies et Raisonnement

Sommaire

6.1 Introduction

Le Web Sémantique est la prochaine étape de World Wide Web selon l'inventeur du Web actuel [73]. Au chapitre précédent nous avons vu les principes de base du Web Sémantique en décrivant notamment son architecture à plusieurs couches. Comme nous l'avons déjà dit, le Web Sémantique laisse présager d'applications autonomes évoluées avec des mécanismes inférentiels puissants. Dans ce présent chapitre nous allons nous intéresser plus précisément aux couches d'ontologies et de logique de l'architecture du Web Sémantique ainsi qu'aux moteurs de raisonnement.

6.2 La logique dans le Web Sémantique

Dans le domaine du Web Sémantique, les machines auront besoin d'appliquer des raisonnements logiques sur tout type de fait. Ces faits pouvant être sous des formes variées, qui seront distribués à travers tout le Web. La logique, qui ici veut dire logique formelle, va jouer différents rôles dans le Web Sémantique [89].

6.2.1 Application et évaluation des règles

L'utilisation de règles pour contrôler le comportement d'une application n'est pas du tout nouveau. Le principe étant d'évaluer la véracité des conditions afin d'exécuter les actions. La Logique intervient dans l'évaluation des conditions et l'application de la règle (pattern-matching, élection, etc). En plus des autres applications basées sur des règles, le Web Sémantique a quelques besoins additionnels concernant ces règles :

- Un formalisme de règles compatible Web : actuellement il n'existe pas de standards acceptés bien que certains soient en cours de finalisation comme le Semantic Web Rule Language [90] et le Web Rule Language (WRL). Ces langages se basent sur RDF.
- La possibilité de spécifier les types de règles, leurs relations et leurs contraintes.
- Une solution pour gérer des règles incompatibles : le Web Sémantique sera l'un des plus gros systèmes ouverts, donc les règles proviendront de diverses sources et peuvent être en conflit. Il faudra un système permettant de résoudre ces conflits.

6.2.2 Faire des inférences

La logique interviendra pour faire des inférences sur des faits non évidents (par exemple la mère de Mr Smith est Anna alors Anna est une femme). Le principe de l'inférence est, à partir de données connues et de règles apprises, de pouvoir générer de nouvelles connaissances implicites qui ne sont pas explicites.
Il y a une importante distinction à faire entre les systèmes d'inférence classiques qui sont

issus de l'intelligence artificiel et l'inférence dans le Web Sémantique. En effet ces deux types de systèmes se différencient de par la manière dont ils gèrent une connaissance incomplète. Il y a une distinction importante entre le *monde ouvert* et le le *monde fermé*. Dans un *monde fermé* toutes les données sont supposées être connues, alors toute donnée absente est considérée comme étant fausse. Alors que dans un *monde ouvert*, une donnée absente est considérée comme étant inconnue. Le Web Sémantique est un système ouvert. Dans un large système ouvert, il y a beaucoup de risques qu'il contienne des informations contradictoires et incorrectes qui posera de sérieux problèmes d'intégration.

6.2.3 Explication

La logique dans le Web Sémantique servira également à expliquer pourquoi une certaine condition a été prise, par exemple un système doit être capable de dire pourquoi un chèque de Mr Smith a été déclaré solvable. Le fait de pouvoir expliquer pourquoi une certaine décision a été prise, c'est à dire d'avoir une traçabilité d'un raisonnement, sera l'un des plus grands apports du Web Sémantique. Dans le Web Sémantique beaucoup de choses se feront automatiquement par des agents intelligents (des machines), qui prendront des décisions importantes à la place de l'homme, alors il est primordiale d'avoir une trace de ces décisions prises, pour pouvoir les prouver mais aussi les annuler.

6.2.4 Contradictions et interprétations

Le Web Sémantique sera un système ouvert immense contenant des informations contradictoires. En cas de contradiction les conséquences peuvent être des plus surprenantes. Comme dans le monde réel, dans le Web Sémantique il peut exister des problèmes d'interprétation mais aussi de pollution de (bases de) connaissances.
Par exemple si on a la déclaration suivante : *Bob est debout devant sa maison.* En se basant uniquement sur cette affirmation on pense systématiquement que Bob est un homme. Par contre si on ajoute la déclaration suivante : *Bob est un chien*, alors de suite notre interprétation change.

6.2.5 Spécification d'ontologies et représentation de connaissance

L'ontologie est un domaine de grande importance pour le Web Sémantique. De manière générale, une ontologie définit les concepts sur lesquelles un système peut se baser pour faire un raisonnement. La logique est également utilisée pour représenter une connaissance, par exemple en disant le genre de concepts qui peuvent être dites sur un sujet et comment les comprendre. L'ontologie fournit les concepts et les termes alors que la logique fournit les moyens de faire des assertions qui les définissent et les utilisent.

6.2.6 Combiner des informations

Comme nous le disions dans la section précédente, le Web Sémantique sera le plus grand système ouvert qui existe, ainsi, la combinaison de diverses sources de données aboutira aux problèmes suivants :

- Différentes sources de données peuvent utiliser des ontologies différentes décrivant les mêmes choses ;
- Différentes sources peuvent avoir des sémantiques différentes pour les mêmes choses. Par exemple le concept *Personne* peut dans une base de données signifier toute personne travaillant pour l'entreprise, alors que dans un autre domaine cela voudra dire tout être vivant ;
- Différentes sources peuvent avoir des informations contradictoires ;
- Différentes sources peuvent avoir des niveaux d'intégrité différents.

L'intégration automatique de données dans le Web Sémantique soulève beaucoup de questions sans réponses. L'une de ces questions les plus préoccupantes est comment empêcher la pollution de bases de connaissance lors de la fusion de plusieurs autres bases de connaissance. Par exemple imaginons un peu que nous ayons deux ensembles de règles avec des parties qui se contredisent, en faisant une fusion automatique de ces deux ensembles, quel est le choix à faire en cas de règles qui se contredisent ? Que garde t-on ? Que supprime t-on ?

6.3 Resources Description Framework (RDF)

RDF [75], qui est normalisé par le W3C, est un modèle conceptuel permettant de décrire des concepts, simplement et sans ambiguïté. Ses applications visent initialement le Web Sémantique mais elles peuvent s'étendre plus largement à l'ingénierie des connaissances. RDF [91] permet de décrire tout concept selon un mécanisme simple : il s'agit d'expressions minimales, composées d'un sujet, d'un verbe ou propriété et d'un complément ou objet ; on parle de déclaration ou triplet RDF. Par exemple : *Clément a pour mère Annabelle* est une déclaration RDF possible. D'autres déclarations où "Clément" est sujet peuvent conduire, grâce à ce simple mécanisme, à une connaissance poussée de ce sujet. On remarquera la proximité de RDF avec le langage naturel, cependant RDF est un modèle formel.

6.3.1 Le modèle RDF

RDF est une infrastructure qui permet l'encodage, l'échange et la réutilisation de méta-donnés structurées. RDF est également un moyen d'exploiter les méta-données. Il permet l'interopérabilité entre applications qui s'échangent des informations compréhensibles par des machines sur le Web.
RDF est une application de XML qui impose des contraintes structurelles nécessaires pour permettre une manière unique d'exprimer une sémantique.

RDF est formé d'un triplet :

- ***Ressources*** : Elles représentent tout ce qui est décrit par une expression RDF. Une ressource peut par exemple être une page web entière, une partie de la page web ou encore une collection de pages web.
- ***Propriétés*** : C'est un aspect, une caractéristique, un attribut, ou une relation spécifique utilisée pour décrire une ressource. Chaque propriété a un sens spécifique. Pour une propriété, il faut définir ses valeurs acceptées, les types de ressources qu'elle peut décrire et ses relations d'avec les autres propriétés.
- ***Objet*** : un objet est la valeur d'une ressource pour une propriété donnée.

La Figure 105 montre un exemple de document RDF qui définit une filiale d'une entreprise ayant comme nom "Génigraph"

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:group="http://xmlns.com/wordnet/1.6/"
        xmlns:filiale="http://www.genitech.fr/rdf/filiale/"
                xmlns:employes="http://www.genitech.fr/rdf/employes/">

<group:filiale rdf:about="http://www.genitech.fr/filiale/genigraph" people:name="Genigraph">
 <filiale:employes>
  <rdf:Seq rdf:about="http://www.genitech.fr/filiale/genigraph/Empoyes">
   <rdf:li>
    <empleye:Person rdf:about=http://www.genitech.fr/rdf/employes/Mouhamed
employe:name="Mouhamed"/>
   </rdf:li>
   <rdf:li>
    <empleye:Person rdf:about="http://www.genitech.fr/rdf/employes/Mickael"
employe:name="Mickael"/>
   </rdf:li>
   <rdf:li>
    <empleye:Person rdf:about="http://www.genitech.fr/rdf/employes/Olivier"
employe:name="Olivier"/>
   </rdf:li>
  </rdf:Seq>
 </filiale:employes>
</group:filiale>

</rdf:RDF>
```

FIG. 105 – Exemple d'un document RDF

6.3.2 XML vs RDF

XML est limité pour deux raisons. La première est qu'en XML il peut exister plusieurs manières de dire la même chose car il peut y avoir plusieurs structures valides pour la même donnée comme le montre la Figure 106. Et la deuxième est qu'il n'impose pas une interprétation commune pour une donnée. On lui reproche souvent aussi d'être verbeux. XML a été pensé pour des documents et non pour des données :

- Il y a plusieurs manières de dire la même chose.
- Structure d'arbre hybride : confusion et pas standard.
- Rend les opérations basiques plus complexes (exemple : la fusion de documents XML)

RDF est orienté données :

- Structure simple : triplet.
- Le nombre de changements que l'on peut faire pour un triplet est assez réduit.
- Fusionner deux documents est très simple, il suffit de fusionner les deux en un.

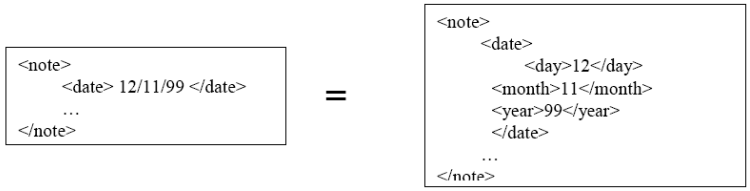

Fig. 106 – Plusieurs formalismes différents pour la même donnée

6.4 Ontologie

Une ontologie est une spécification explicite formelle d'une conceptualisation partagée. Une conceptualisation est un modèle abstrait de la manière dont les gens pensent et voient les choses dans le monde, en général restreint à un ensemble de sujets particuliers.
Une ontologie, au sens web sémantique, peut être définie comme étant un ensemble structuré de savoirs. Une ontologie en général définit des classes ou catégories, des termes et des relations. Elle va aussi définir quelle classe sera utilisée avec les relations. Elle définit également les types de données ainsi que les contraintes sur l'utilisation des classes et des propriétés.
Suivant le domaine de prédilection, une ontologie peut aller de la taxonomie (connaissance avec un minimum d'hiérarchie ou structure Parent/Enfant) au thésaurus ou lexique (mots et synonymes) ou modèles conceptuels (avec une connaissance plus complexe), à la théorie de logique (avec une connaissance très riche, complexe, consistante et sensée) [92].
Les ontologies ne sont pas des systèmes de logique mais fournissent plutôt les objets qui sont utilisés dans une logique de raisonnement. Les ontologies peuvent spécifier des règles pour gérer certaines logiques d'inférence. L'un des grands apports du Web Sémantique est la standardisation d'un langage d'ontologies pour le Web. Ce standards existe depuis début 2004 mais long fut le processus pour y arriver. Nous allons voir maintenant ce standard et les langages qui y ont mené.

6.4.1 Les langages d'ontologie pour le Web

Les classes standards de RDF permettent de définir des caractéristiques basiques de classes pour une ontologie. Cependant des langages plus évolués comme OWL offrent plus de puissance dans la conception d'ontologies [93]. Avant la standardisation d'un langage

d'ontologie pour le web, plusieurs langages ont été utilisés dont XOL (XML-based Ontology Exchange Language) [94], SHOE (Simple HTML Ontology Extension), OML (Ontology Markup Language) [95, 96], RDF(S) (Resource Definition Framework (Schema)), OIL (Ontology Interchange Language), DAML+OIL (DARPA Agent Markup Language + OIL)[97]. Nous allons voir les cas de SHOE et de DAML+OIL. SHOE a mis l'accent sur le fait que les ontologies pouvaient fortement se lier et être l'objet de changements. SHOE a influencé la mise en place de OWL.

SHOE

L'une des premières tentatives pour définir un langage d'ontologie pour le Web fut SHOE [98, 99]. SHOE est un langage basé sur les frames avec une syntaxe en XML qui s'intègre facilement au HTML. SHOE utilise les références URI pour les noms, ce qui était une importante innovation qui fut adoptée par DAML-ONT and DAML+OIL. La Figure 107 montre un exemple simple du langage SHOE.

```
<INSTANCE KEY="http://www.cs.umd.edu/users/george/">

        <USE-ONTOLOGY
                  ID="cs-dept-ontology"
                  URL="http://www.cs.umd.edu/projects/plus/SHOE/onts/cs.html"
                  VERSION="1.0"
                  PREFIX="cs">

        <CATEGORY NAME="cs.GraduateStudent">
        <CATEGORY NAME="cs.ResearchAssistant">

        <RELATION NAME="cs.name">
                <ARG POS=TO VALUE="George Stephanopolous">
        </RELATION>
        <RELATION NAME="cs.age">
                <ARG POS=TO VALUE="52">
        </RELATION>
        <RELATION NAME="cs.advisor">
                <ARG POS=TO VALUE="http://www.cs.umd.edu/users/smith">
        </RELATION>

        <INSTANCE KEY="http://www.cs.umd.edu/users/george/#BRUNHILDA">

                <CATEGORY NAME="cs.Lecturer">

                <RELATION NAME= "cs.name">
                        <ARG POS=TO VALUE="Brun Hilda">
                </RELATION>
                <RELATION NAME="cs.age">
                        <ARG POS=TO VALUE="23">
                </RELATION>

        </INSTANCE>
</INSTANCE>
```

FIG. 107 – Exemple d'un document SHOE

DAML-ONT

En 1999 le programme DARPA Agent Markup Language (DAML) fut initié dans le but de fournir les fondations de la prochaine génération du Web "sémantique". Dès le début du projet, il fut décidé que l'adoption d'un langage d'ontologies commun faciliterait l'interopérabilité sémantique entre les projets variés du programme. RDFS, qui était déjà proposé comme standard W3C, état vu comme étant un bon point de départ mais insuffisamment expressif pour satisfaire les besoins de DAML. Un nouvel langage fut alors créé, qui étendait RDF avec des constructions prises des langages de représentation de connaissance orientés objet et basés sur les frames.
DAML+ONT [100] fut fortement couplé à RDFS pour garder une compatibilité entre les deux langages. Ce lien fort d'avec RDF a posé de sérieux problèmes pour l'utilisation de DAML+ONT.

OIL

Au même moment où DAML-ONT était en cours de développement, un groupe de chercheur (largement composé d'européens) avait conçu un autre langage d'ontologie pour le Web appelé OIL (Ontology Inference Layer) [101]. OIL était le premier langage à combiner des éléments issus des logiques de description, des langages de frames et aussi de standards web comme XML et RDF. OIL a beaucoup mis l'accent sur la rigueur formelle et le langage était explicitement conçu de telle sorte que sa sémantique puisse être spécifiée via une transformation logique de description SHIQ [102]. La structure du langage était basée sur les frames utilisant un style de définition de classe composée. OIL a une syntaxe XML et RDF.

DAML+OIL

DAML+OIL [103] est un langage sémantique pour le Web. DAML+OIL est issue de la fusion des groupes de travail de DAML-ONT et OIL afin de mieux atteindre leur objectif qui était d'avoir un langage d'ontologies pour le Web hautement flexible [104]. Influencé par DAML+ONT, DAML+OIL est plus fortement intégré à RDF. Cependant, DAML+OIL fournit seulement la signification des éléments de RDF qui étaient consistants avec sa propre syntaxe et langage de modèle de logique de description. Une ontologie DAML+OIL est composée de readers, de classes, d'éléments, de propriétés et d'instances. La Figure 108 montre un exemple du langage DAML.

Web Ontologie Language (OWL)

OWL pour Web Ontology Language est une recommandation initiée par le W3C [105] dans le but d'offrir un moyen plus évolué que RDFS. Comme le montre la Figure 109, OWL qui est intéressé par la sémantique, se situe au dessus de RDF/RDF schema, qui à leur

```
<daml:Class rdf:ID="Man">
  <rdfs:subClassOf rdf:resource="#Person"/>
  <rdfs:subClassOf rdf:resource="#Male"/>
</daml:Class>

<daml:Class rdf:ID="Woman">
  <rdfs:subClassOf rdf:resource="#Person"/>
  <rdfs:subClassOf rdf:resource="#Female"/>
</daml:Class>
```

FIG. 108 – Exemple d'un document DAML

tour se situent au dessus de XML/XML Schema dans l'architure du web Sémantique [105]. OWL est issue de DAML+OIL [106]. OWL est composé de 3 sous langages : *OWL Lite*, *OWL DL* et *OWL Full*. Ces 3 sous langages se différencient selon leur degré d'expressivité. Comme le montre la figure 110, OWL DL est une extension de OWL Lite et OWL Full est une extension de OWL DL.

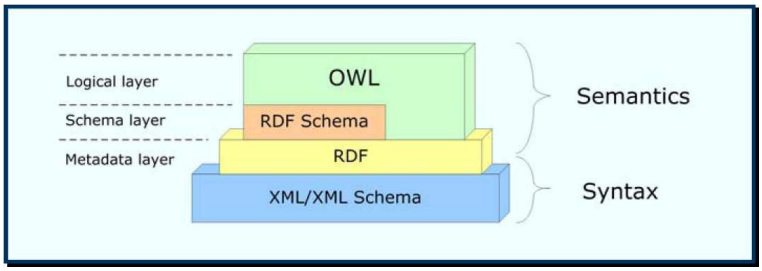

FIG. 109 – OWL dans l'architecture du Web Sémantique

OWL Lite :
Syntaxiquement parlant c'est le sous langage le plus simple. Il a pour but d'être utilisé dans le cas où on a une simple hiérarchie de classes et des contraintes simples. OWL Lite ne permet pas de faire une logique complexe, son utilisation tient juste à des formulations simples : création d'une taxonomie ou d'un thésaurus.

OWL DL :
Il est plus expressif que OWL Lite et est basé sur les logiques de description (DL). Les logiques de descriptions sont une partie décidable de la logique de premier ordre (FOL) et donc permet de faire un raisonnement logique automatique. OWL DL garantit une complétude des raisonnements (toutes les inférences sont calculables) et leur décidabilité (que leur calcul se fait en une durée finie).

OWL Full :
C'est le sous langage le plus expressif mais aussi le plus complexe. Il est utilisé dans les cas où ce qui importe c'est d'avoir un grand degré d'expressivité et non une décidabilité ou un raisonnement automatique. OWL Full n'est pas sujette au raisonnement automatique car n'est pas décidable.

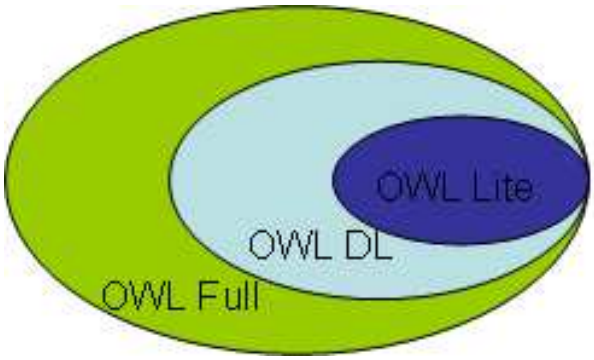

Fig. 110 – Dépendance hiérarchique entre les sous langages OWL

Cette section ne se veut pas d'être un descriptif complet de OWL mais se contente juste de mettre en évidence les éléments les plus importants de ce langage.

Les composants d'une ontologie OWL

Une ontologie OWL est composée de propriétés, de classes et d'individus (instances de classes) qu'on appelle triplet. En fait de compte une connaissance que représente OWL/RDF n'est rien d'autre qu'un ensemble de ces triplets.

Les classes :
Les classes sont interprétées comme étant des ensembles d'individus. Elles peuvent être exprimées en terme de relation super-classe/sous-classes ou père/enfants appelée taxonomie. Une des caractéristiques clés de OWL DL c'est que les relations super-classe/sous-classes peuvent être calculées automatiquement par un reasoner (moteur de raisonnement), on parle alors d'hiérarchie inférée. Les classes sont formées de descriptions qui spécifient les conditions qui doivent être satisfaites par un individu pour appartenir à cette classe.

Les propriétés :
Les propriétés sont des relations binaires sur les individus, par exemple la propriété *est-PéreDe* peut lier Peter et Mathew. Une propriété peut être limitée, transitive et/ou symétrique. Les propriétés correspondent aux rôles en Description Logic et aux relations en

UML et autres méta-modèles orientés objet. OWL propose deux catégories principales de propriété :

- Les "Object properties" qui lient des individus à des individus.
- "Datatype properties" qui lient des individus à des valeurs de données, c'est à dire des types primitifs (entier, chaînes de caractères, etc).

Une propriété de type objet est définie comme étant une instance de la classe *owl :ObjectProperty*. Une propriété de type donnée est définie comme étant une instance de la classe *owl :DataProperty*. Aussi bien *owl :ObjectProperty* et *owl :DataProperty* sont toutes les deux des sous-classes de la classe RDF *rdf :Property*.

Un axiome de propriété définit les caractéristiques d'une propriété. Dans sa forme la plus simple, un axiome d'une propriété définit l'existence d'une propriété. Par exemple *<owl :ObjectProperty rdf :ID="estParentDe">* définit une propriété *estParentDe* avec la restriction que ses valeurs doivent être des individus.

Souvent les axiomes de propriété définissent des caractéristiques additionnelles de propriétés. OWL supporte les constructions suivantes pour les axiomes de propriété :

- Schéma de construction RDF : *rdfs :subPropertyOf, rdfs :domain, rdfs :range.*
- Relations avec d'autres propriétés : *owl :equivalentProperty et owl :inverseOf.*
- Contraintes globales de cardinalité : *owl :FunctionalProperty et owl :InverseFunctionalProperty.*
- Caractéristique logiques de propriété : *owl :SymmetricProperty, pwl :TransitiveProperty*

Nous verrons plus loin dans ce document les détails de ces axiomes de propriété.

Les individus ou instances de classe :

La définition d'un individu consiste à énoncer un "fait", encore appelé "axiome d'individu". On peut distinguer deux types de faits :

- *les faits concernant l'appartenance à une classe.*
  La plupart des faits concerne la déclaration de l'appartenance d'un individu à une classe et les valeurs de propriété de cet individu. Un fait s'exprime de la manière suivante :
  Exemple 6.1

```
<Humain rdf:ID="Pierre">
  <aPourPere rdf:resource="#Jacques"/>
  <aPourFrere rdf:resource="#Paul"/>
</Humain>
```

  Le fait déclaré dans cet exemple exprime l'existence d'un Humain nommé *"Pierre"* dont le père s'appelle *"Jacques "*, et qu'il a un frère nommé *"Paul"*.
- *les faits concernant l'identité des individus.*
  Une difficulté qui peut éventuellement apparaître dans le nommage des individus concerne la non-unicité éventuelle des noms attribués aux individus. Par exemple, un même individu pourrait être désigné de plusieurs façons différentes. C'est la raison

pour laquelle OWL propose un mécanisme permettant de lever cette ambiguité, à l'aide des propriétés *owl :sameAs*, *owl :differentFrom* et *owl :allDifferent*. L'exemple suivant permet de déclarer que les noms *"David"* et *"Daoud"* désignent la même personne :
Exemple 6.2

```
<rdf:Description rdf:about="#David">
  <owl:sameAs rdf:resource="#Daoud" />
</rdf:Description>
```

Une fois que l'on sait écrire une classe en OWL, la création d'une ontologie se fait par l'écriture d'instances de ses objets, et la description des relations qui lient ces instances.

Nous venons de voir les langages d'ontologies pour le web, nous allons maintenant voir comment les utiliser pour un raisonnement dans le Web Sémantique.

6.5 Techniques de raisonnement dans le Web Sémantique

L'une des caractéristiques principales des ontologies décrites en OWL DL est qu'elles peuvent être traitées par un moteur de raisonnement. L'un des services principaux qu'offrent un moteur de raisonnement est de tester si oui ou non une classe est une sous-classe d'une autre classe (subsumption). En procédant de la sorte sur toutes les classes d'une ontologie, il est possible pour un moteur de raisonnement d'inférer sur toute l'hiérarchie de classes dans une ontologie.
Un autre service standard qu'offre un moteur de raisonnement est de tester la consistance d'une ontologie. En se basant sur les conditions d'une classe, le moteur de raisonnement peut vérifier s'il est possible pour une classe d'avoir une instance. Lors de la création d'ontologies assez consistantes (plus de 1000 classes), l'utilisation de moteur de raisonnement pour calculer les relations classe/sous classe devient vitale. Ceci permet une classification simple, on parle alors de classification *assertée* et de classification *inférée*.
Comme moteur de raisonnement on peut avoir des systèmes raisonnement basés sur les logiques de description ou sur les règles.

6.5.1 Systèmes de raisonnement basés sur les logiques de description

OWL bénéficie des résultats de plus de 15 ans de recherche dans les Logiques de Description (DL) [107, 44]. En effet, pour OWL, une sémantique est définie de telle sorte que les gros fragments (expressions) du langage peuvent directement être exprimés en Description Logics [108]. Les logiques de descriptions sont une famille qui permet la représentation de

connaissance et sont des descendants de la théories des réseaux sémantique et de KL-ONE [109]. Les logiques de description décrivent les domaines en terme de concepts (classes), roles (propriétés et relations), appelés T-box, et individuels (instances), appelés A-Box. Dans la terminologie des logiques de description, un tuple de T-box et de A-box définit une base de connaissance. Les algorithmes implémentés et les systèmes de raisonnement basés sur les logiques de descriptions existent déjà et peuvent être utilisés pour faciliter la "découverte" de connaissances dans le Web Sémantique. Les logiques de descriptions forment un sous-ensemble de la logique du premier ordre (First Order Logic - FOL). OWL Lite et OWL DL ne sont rien d'autre que des logiques de descriptions utilisant une syntaxe RDF [110]. Ainsi la sémantique de OWL, aussi bien que la décidabilité et la complexité de ses problémes basiques d'inférence peuvent être traités par certains travaux sur les logiques de descriptions. Ce qui n'est pas le cas de OWL Full qui est indécidable. Cependant, bien que OWL DL et les langages de descriptions aient beaucoup de points communs, ils diffèrent sur certain points, notamment les espaces de nommage (namespace) et la possibilité d'importer d'autres ontologies de OWL DL. Avec quelques restrictions, on peut voir que les bases logiques de OWL DL peuvent être caractérisées par les logiques de descriptions de type $SHIQ(D_n)^-$ et OWL lite caractérisées par $SHIF(D)$ dans un temps polynomial [111]. Cela veut dire qu'avec quelques restrictions, les documents OWL peuvent être automatiquement transformés en T-box $SHIQ(D_n)^-$. La partie RDF des documents OWL peuvent être transformée en A-box $SHIQ(D_n)^-$ [112, 113].
La logique $SHIQ(D_n)^-$ est intéressante pour des applications pratiques parce qu'il existe des systèmes d'inférence avec un haut niveau d'optimisation comme Racer, FaCT qui peuvent être utilisés pour fournir des services de raisonnement [114]. Dans de tels systèmes, en se basant sur les T-box nous pouvons faire les raisonnements suivants :

1. Consistance de concepts : est-ce que l'ensemble des objets décrits par un concept est vide ?
2. Concept subsumé : y'a t-il une relation de sous-ensemble entre les objets définis par deux concepts ?
3. Trouver toutes les incohérences entre les concepts définis dans un T-box. Les concepts incohérents sont souvent dus à des erreurs de modélisation.
4. Déterminer (calculer) les parents et les fils d'un concept : les parents d'un concept sont les concepts les plus spécifiques dans un T-box qui subsument ce concept (parents immédiats). De la même manière, les enfants d'un concept sont les concepts immédiats que subsume ce concept.

Avec la terminologies A-box nous pouvons faire les raisonnements suivants :

1. Verifier la consistance d'un A-box : est ce que les contraintes fournies dans un A-box se contredisent ?
2. Test d'instances : est qu'un individu est une instance d'un concept ?
3. Recherche d'instances : trouver tous les individus d'un A-box suivant une condition bien définie.

4. Calculer les types directs d'un individuel : trouver le concept d'un T-box le plus spécifique pour un individu.

6.5.2 Systèmes de raisonnement basés sur les règles

Une autre alternative à l'utilisation de systèmes de raisonnement des logiques de descriptions pour le Web Sémantique est d'utiliser des systèmes de raisonnement orientés règles. De tels systèmes incluent DAMLJessKB [115] et OWLLisaKB. Le premier utilise le moteur de règle Jess pour faire des inférences sur les ontologies DAML, tandis que le second utilise le système de règles Lisa pour faire des inférences sur des ontologies OWL. Le problème avec cette deuxième alternative est que les systèmes de règles nécessitent une composition manuelle des règles qui reflètent la sémantique des éléments des ontologies OWL, ce qui est tout à fait faisable mais laborieux. Ceci expliquerait pourquoi de tels systèmes soient uniquement utilisés dans le cas de OWL Lite [114]. Ce genre de services d'inférence basés sur des règles est inclus dans le framework Jena [116] qui est développé par Hewlett-Packard.

Les systèmes de raisonnement des logiques de description sont les plus adaptés pour les ontologies DAML+OIL ou OWL. Dans la sous-section suivante nous allons en décrire certains.

6.5.3 Evaluation de systèmes de raisonnement basés sur les logiques de description

Nous allons évaluer les systèmes de raisonnement basés sur les logiques de description suivant : Cerebra, FaCT, FaCT++ et Racer. Nous nous sommes basés sur les critères de type de license, de leur expressivité, de leur prise en charge de OWL, de leur capacité à raisonner sur les A-Box et leur capacité d'interconnection.

Cerebra

Cerebra [117] de Cerebra Inc est un système commercial qui fournit des services de raisonnement et de gestion d'ontologies. L'une des caractéristiques les plus intéressantes de Cerebra est sa possibilité à rendre persistant une base de connaissance. Cerebra peut charger des documents OWL en local ou par le WEB. Cerebra fournit une connectivité pour les applications clientes basées sur Java et .Net. Les services Web peuvent utiliser son interface SOAP ou RMI. Cerebra supporte l'inférence sur la partie taxonomie d'une ontologie, c'est-à-dire les TBox, mais elle ne supporte pas les inférences sur les instances (ABox). Le langage de logique de Cerebra est de type $SHIQ$.

FaCT

FaCT est un système de raisonnement gratuit et qui est développé à l'université de Manchester [118, 119, 120]. Au départ, FaCT se basait sur la logique de type $SHFDL$ puis a évolué pour inclure $SHIF$ et finalement $SHIQ$. FaCT permet une connectivité pour des clients par CORBA. Les clients peuvent également utiliser le standard DIG/1.0 qui fournit un protocole de communication simple par l'échange de requête en XML et les réponses par HTTP. FaCT implémente des algorithmes optimisés pour résoudre le problème de subsumption dans les logiques de descriptions, ceci en fait un pionnier dans ce domaine. Cependant le manque de gestion des inférences sur ABox fait que FaCT ne puisse pas être utilisé pour OWL.

FaCT++

La majeure partie des limitations de FaCT a été corrigée dans sa version suivante qui est FaCT++ et qui fait partie du projet Wonderweb. FaCT++ se différencie de FaCT sur plusieurs aspects. FaCT++ est la ré-implémentation de FaCT en C++ en améliorant son expressivité de sorte qu'il puisse être utilisé pour OWL DL. Son langage est de type $SHIQ(D)$. FaCT++ est accessible via le standard DIG/1.1.

Racer

Racer [112, 121] est un moteur de raisonnement pour les logiques de description très expressives. Racer est le premier système du genre à supporter le raisonnement sur aussi bien les TBox que les ABox, et ceci est son principal avantage. Racer est développé à l'université à Hamburg, il est commercial mais il existe une licence gratuite pour les chercheurs. Racer peut communiquer avec des clients par les protocoles TCP et DIG/1.1 avec l'utilisation des langages C++ et Java. Depuis sa version 1.7.7 Racer peut traiter en natif un format à la Lisp, XML, RDF, RDFS, DAML+OIL et OWL. La logique utilisée par Racer est de type $SHIQ$ incluant les instances (ABox).
Le tableau 8 donne le récapitulatif de la comparaison.

6.6 Conclusion

Dans ce chapitre nous avons vu le langage d'ontologie web OWL ainsi que les travaux qui ont influencés son élaboration, notamment RDF, SHOE, OIL, DAML et DAML+OIL. Depuis 2004 OWL est une recommandation du W3C pour faire des ontologies pour le Web. Nous nous sommes également intéressés à la couche logique du Web Sémantique. Nous avons montré quel était le lien entre OWL DL et les logiques de description, en montrant que OWL DL est une forme de DL très expressive. L'objectif de la corrélation faite entre OWL DL et les logiques de description était de montrer que les recherches de

	Type de Licence	**Connectivité**	**Type de Raisonnement**	**Support natif de OWL**	**Raisonnement sur les instances (ABox)**
Cerebra	Commercial	RMI, SOAP	*SHIQ*	Oui	Non
FaCT	Gratuit	CORBA, DIG/1.0	*SHIQ*	Non	Non
FaCT++	Gratuit	DIG/1.1	*SHIQ*(D)	Non	Non
Racer	Gratuit/com	TCP, DIG/1.0	*SHIQ*(D)	Oui	Oui

Tab. 8 – Récapitulatif de la comparaison de systèmes de raisonnement basés sur les logiques de description

ce domaine depuis plus de 15 ans peuvent être utilisées pour faire un raisonnement sur les ontologies Web en utilisant des systèmes de raisonnement existants. Nous avons fini en faisant une étude comparative de Cerebra, FaCT, FaCT++ et Racer qui sont des moteurs de raisonnement.
Après avoir introduit les concepts d'ingénierie dirigée par les modèles et de Web Sémantique, nous allons maintenant montrer dans le prochain chapitre, comment les utiliser pour faire de la génération automatique de règles métier par enrichissement sémantique de modèles.

Quatrième partie

Enrichissement sémantique des modèles MDA pour la génération de règles métier simples et génériques

Chapitre 7

Enrichissement sémantique des modèles MDA pour la génération de règles métier simples et génériques

Sommaire

7.1 Introduction

Durant la première partie de la thèse nous avons passé beaucoup de temps à travailler sur l'état de l'art. Cet état de l'art portait non seulement sur les règles métier (utilisation et formalisme) mais aussi sur les modèles conceptuels au sens MDA et le Web Sémantique, notamment tout ce qui est raisonnement d'ontologies et moteur de raisonnement. Le formalisme de règles métier suscite beaucoup d'intérêts actuellement (SBVR [47] et PRR [49] de l'OMG, RIF du W3C [54]) ainsi que les travaux sur les services (transformation, validation, tissage, translation, simulation etc.) sur les modèles.
Ces différents domaines, qui peuvent sembler être éloignés les uns des autres à première vue, sont contiguës et se chevauchent même. Arrivé à ce point, il a fallu faire un choix entre :

- soit se focaliser sur le formalisme de règles métier ;
- soit combiner les domaines déjà étudiés et voir comment apporter une nouvelle approche.

Après réflexion, nous sommes arrivés au constat suivant : les règles métiers travaillent sur des modèles, les modèles MDA n'ont aucune notion de sémantique et le Web Sémantique porte uniquement sur la sémantique. En plus de cela nous avons remarqué que dans *e-Citiz* nous avions beaucoup de modèles, à la fois natifs et créés par les experts fonctionnels. Alors nous nous sommes posés la question suivante : pourquoi ne pas essayer d'utiliser les principes du Web Sémantique ou une autre méthode pour rajouter de la sémantique dans les modèles conceptuels afin de pouvoir générer des règles génériques simples et plus (par exemple fusion de bases de connaissance) ?
Le Model Driven Architecture ne dit absolument rien sur la sémantique dans les modèles, dans les prochaines sections nous allons voir quels sont les mécanismes possibles pour rajouter de la sémantique dans les modèles. Dans un premier temps nous allons recenser ces mécanismes puis dans un deuxième temps en faire un comparatif pour faire un choix.

7.2 Les mécanismes potentiels permettant d'ajouter de la sémantique aux modèles MDA

Comme nous l'avons déjà dit, le MDA ne dit absolument rien sur la sémantique. Cependant MDA préconise l'utilisation de UML pour ses modèles [60], et ce langage de modélisation offre des éléments qui pourraient nous permettre d'arriver à ajouter de la sémantique dans des modèles. Nous allons rappeler les principes des profils UML, du langage de contrainte OCL et Action Semantics. Nous allons voir aussi une nouvelle norme en cours de finalisation à l'OMG ayant pour nom Ontology Definition Metamodel.

7.2.1 Les profils UML

UML permet de modéliser des applications orientées objet. Ses concepts sont suffisamment génériques pour être utilisés dans d'autres domaines [122]. Le problème est qu'il devient dès lors de plus en plus difficile en regardant un modèle de savoir s'il modélise une application objet, un méta-modèle, une base de données ou tout autre concept [60].
Afin de spécialiser un modèle UML à un domaine d'application, l'OMG a standardisé le concept de profil UML. Un profil UML est un ensemble de techniques et de mécanismes permettant d'adapter UML à un domaine particulier. Cette adaptation est dynamique donc ne modifie en rien la structure du méta-modèle UML et peut se faire sur n'importe quel modèle UML.
Un profil UML se définit aux travers du concept de stéréotype. Un stéréotype est une sorte d'étiquette nommée que l'on peut coller sur n'importe quel élément d'un modèle UML. Lorsqu'un stéréotype est collé sur un élément d'un modèle, le nom du stéréotype définit la nouvelle signification de l'élément. Par exemple, coller un stéréotype nommé *"Sécurisé"* sur une classe UML signifie que la classe en question n'est plus une simple classe UML mais qu'elle est une classe sécurisée. Utiliser un profil consiste donc à coller sur un modèle UML un ensemble de stéréotypes.
D'un point de vue graphique, un stéréotype se représente sous la forme d'une chaîne de caractères contenant le nom du stéréotype encadré de guillemets. Si le stéréotype est collé sur une classe, cette chaîne de caractères doit apparaître au-dessus du nom de la classe comme le montre l'exemple de la figure 111.

Fig. 111 – Exemple d'utilisation d'un stéréotype sur une classe

Il est important de noter que pour qu'un profil UML soit efficace, il doit être bien utilisé et bien compris par tous les intervenants sur les mêmes modèles. Par exemple il faudrait que le stéréotype "Sécurisé" puisse être accepté par tous les intervenants sur le modèle comme définissant un type de classe particulier.
Afin d'éviter une floraison de profils, l'OMG a défini des profils standards. Ces profils définissent un ensemble de stéréotypes et expliquent leur signification en langage naturel.
L'intérêt principal des profils est qu'il est possible de leur associer des traitements spécifiques au domaine couvert par le profil. Ces traitements permettent de rendre les modèles UML profilés beaucoup plus productifs que les modèles UML simples, car disposant d'informations supplémentaires grâce aux stéréotypes.
UML2.0 Superstructure [61] facilite la création de nouveaux profils en simplifiant la définition des concepts de profils et de stéréotypes dans le méta-modèle. Ces concepts de profils sont représentés par des méta-classes. Un modèle instance de ces méta-classes correspond à

la définition d'un nouveau profil et non à l'application d'un profil existant dans un modèle UML.

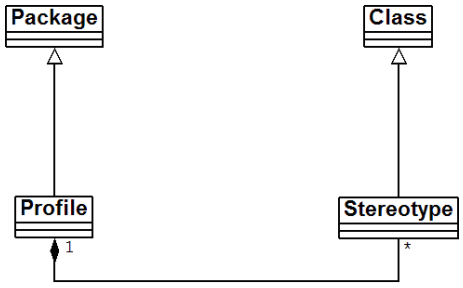

FIG. 112 – Les profils dans UML2.0

La figure 112 illustre la partie du métamodèle UML2.0 qui contient les métaclasses relatives aux profils. Dans ce métamodèle on peut voir que la métaclasse *Profil* hérite de la métaclasse *Package*. Cela signifie que les définitions de nouveaux profils sont maintenant considérées comme étant des modèles UML (comme étant des packages). La métaclasse *Stéréotype* hérite de son côté de la métaclasse *Class*. Cela veut dire que les définitions de stéréotypes sont considérées comme étant des classes UML avec le nom de la classe correspondant au nom du stéréotype.

7.2.2 Le standard OCL

Dans UML il n'était pas possible d'exprimer précisément ce que fait une opération, c'est-à-dire son corps. En effet dans le métamodèle UML, une opération est définie uniquement par son nom, ses paramètres et les exceptions qu'elle émet. Le langage OCL [50], Object Constraint Language, a été défini par l'OMG pour combler cette lacune et permettre la modélisation du corps des opérations dans un modèle UML.
OCL peut exprimer des contraintes sur tous les éléments d'un modèle UML. Il est utilisé pour exprimer des pré et post-conditions sur les opérations. Par exemple, il est possible de dire, en utilisant OCL, que l'opération *accepterUneLocationDeVoiture* a comme précondition *verifierClientMajeur*, ce qui veut dire qu'avant de faire appel à l'opération qui accepte la location d'une voiture il faut d'abord vérifier que le client est majeur.
Les expressions OCL ne génèrent aucun effet de bord. L'évaluation d'une expression OCL n'entraîne aucun changement d'état dans le modèle auquel elle est rattachée. Une contrainte OCL est une expression dont l'évaluation doit retourner vrai ou faux. L'évaluation d'une contrainte OCL permet de la sorte de savoir si la contrainte est respectée ou non.
OCL dispose d'un formalisme textuel, on parle d'expression OCL. Une expression OCL porte sur un élément du modèle UML. Pour être évaluée, une expression OCL doit être rattachée à un contexte, qui doit être directement relié à un modèle UML. L'exemple suivant permet de spécifier, pour un compte bancaire, que la valeur de retour de l'opération

getSolde doit être positive avant toute invocation de l'opération *debit* :
context *CompteBancaire : :getSolde() :debit() :Integer*
pre *: solde > 0*
Le métamodèle OCL permet de représenter n'importe quelle expression OCL sous forme de modèle. Cette représentation permet de rendre les expressions OCL pérennes et productives. De plus, le lien fort qui unit les modèles UML et les expressions OCL représentées sous forme de modèles, permet d'exploiter pleinement les modèles UML des PIM de MDA [60]. Pour des raisons évidentes de simplicité d'utilisation, OCL reste avant tout un langage textuel. Cependant, un effort important a été fourni pour définir la façon de passer automatiquement de la forme textuelle à la forme modèle bien qu'elle ne soit pas triviale.

7.2.3 Action Semantics

Jusqu'à sa version 1.4, UML était très critiqué parce qu'il ne permettait pas de spécifier des créations, des suppressions ou des modifications d'éléments de modèles. Ces actions ne pouvant pas non plus être spécifiées à l'aide du langage OCL, puisque celui-ci est sans effet de bord, il était nécessaire de standardiser un nouveau langage. C'est ce qui a donné naissance au langage AS (Action Semantics) [60].
L'objectif de Action Semantics [51] est de permettre de définir des actions. Une action au sens AS est une opération sur un modèle qui fait changer l'état du modèle. Grâce à ces actions, il est possible de modifier les valeurs des attributs, de créer ou de supprimer des objets, de créer de nouveaux liens entre les objets, etc. Ainsi le concept d'Action Semantics permet de spécifier pleinement le corps des opérations UML.
Au départ AS était un langage développé à part entière hors de UML puis inclus dans la version 1.5. Dans UML 2.0, il est enfin totalement intégré au métamodèle.
AS n'est standardisé que sous forme de métamodèle, et aucune syntaxe concrète n'est définie, contrairement à OCL. Ceci offre l'avantage d'utiliser la syntaxe que l'on veut.
Le package *IntermediateActions* de UML2.0 Superstructure [61] définit des métaclasses représentant des actions exécutables sur des éléments de modèle. Il existe, par exemple, des actions pour ajouter des liens entre les objets, émettre des messages, naviguer parmi les instances d'une classe, etc. Le package contient en tout une cinquantaine de métaclasses.
Comme nous venons de le dire, AS n'est défini que sous forme de métamodèle et ne propose pas de format concret (textuel). Il n'est donc pas possible de spécifier des activités sous un format concret. Le standard précise cependant que plusieurs formats concrets peuvent être utilisés et qu'il est du ressort du concepteur d'outil de modélisation de fournir un format concret et d'expliquer comment ce dernier peut se traduire en un format de modèles. Ce travail étant encore particulièrement difficile à réaliser, aucune proposition de format concret n'a encore réellement séduit les utilisateurs. C'est la raison pour laquelle AS n'est pas encore pleinement exploité, contrairement à OCL [60]. Ce manque de format concret (textuel), rend l'utilisation de AS complexe.
Afin de mieux comprendre l'importance d'un format concret, nous allons monter un exemple simple d'activité. Cette activité contient trois actions : la création d'un compte

bancaire, l'affectation d'un montant au solde et l'appel à une opération de débit. Nous proposons d'exprimer cette activité en utilisant une syntaxe proche du langage Java. Cette activité se présenterait de la façon suivante :
CompteBancaire cd = new CompteBancaire() ;
cd.solde = 100 ;
cd.debit(10) ;
La Figure 113 donne une représentation de cette activité sous forme de modèle. Dans cette illustration les actions sont sous forme de cercle et les flux de données sous forme de flèche. Pour que l'exemple soit complet, il faudrait faire apparaître l'activité qui définirait la séquence entre les actions. Cet exemple montre combien il est difficile de passer d'un format concret à un format de modèle avec Action Semantic.

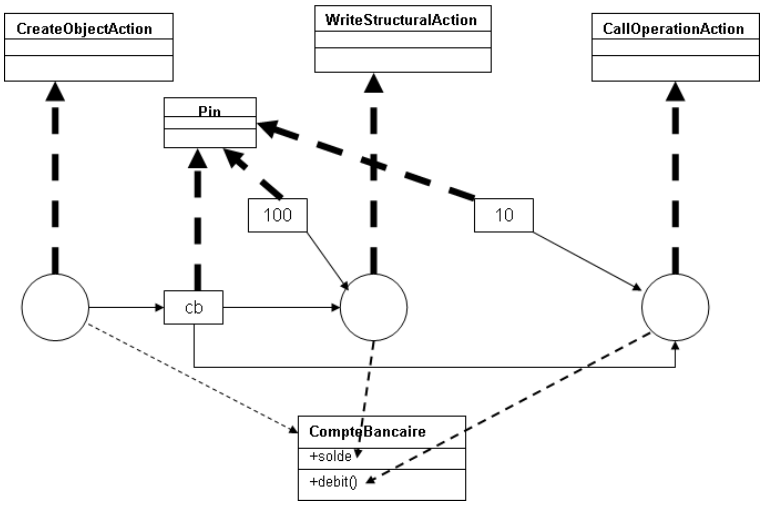

FIG. 113 – Les actions sous forme de modèle dans Action Semantics

7.2.4 Ontology Definition Metamodel (ODM)

L'ontology Definition Metamodel (ODM) [123] a pour objectif de permettre la modélisation d'ontologies en utilisant une instance de MOF. ODM est une spécification de l'OMG qui, à l'heure actuelle (Septembre 2007) est encore en phase de finalisation. L'ingénierie d'ontologies a atteint un certain degré de maturité et un franc succès dans certains domaines. Cependant, il y a toujours un écart entre l'ingénierie d'ontologies et l'ingénierie traditionnelle de logiciels. Depuis quelques décennies, l'ingénierie de logiciels s'est définie sur différents langages de modélisation tel que UML. L'objectif premier des travaux sur ODM est de réduire l'écart entre deux différents, mais complémentaires, domaines d'ingénierie (surtout celui du génie logiciel) [124].

Besoin

Une ontologie définit les termes et concepts (sens) utilisés pour décrire et représenter une connaissance. Une ontologie peut aller de la taxonomie (connaissance avec un minimum d'hiérarchie ou structure Parent/Enfant) au thésaurus ou lexique (mots et synonymes) ou modèles conceptuel (avec une connaissance plus complexe), à la théorie de logique (avec une connaissance très riche, complexe, consistante et sensée) [92]. Une ontologie doit être exprimée dans une syntaxe bien définie permettant une automatisation du traitement par les machines.
Les ontologies sont souvent capturées dans des langages de représentation de la connaissance (KR : Knowledge Representation) qui proviennent des théories de l'intelligence artificielle. Ces langages sont pour la plupart structurés suivant des formalismes de logique tels que la logique de prédicats. Cette structure a rendu la syntaxe de ces langages inhabituelle et peu commode à ceux qui ont l'habitude d'utiliser d'autres langages de modélisation, ce qui est le cas de la plupart des développeurs de logiciel ; ceci explique le manque d'utilisation large de ces langages et donc le développement d'ontologies, retardant ainsi l'adoption des technologies du Web Sémantique.
D'un autre côté UML [61] est un langage de modélisation populaire et largement répandu en matière de modélisation conceptuelle. Beaucoup de communautés utilisent UML avec une armada d'outils très évolués aussi bien en open source qu'en propriétaire.
La familiarité des utilisateurs avec UML, la disponibilité des outils UML, l'existence de plusieurs modèles de domaines et la similarité de ces modèles aux ontologies, suggèrent qu'UML puisse être utilisé pour faire des développements d'ontologies. Ceci permettra aussi de créer un lien entre la communauté UML et celle du Web Sémantique [125] avec comme bénéfices :

- La génération de descriptions d'ontologies standards depuis des modèles UML existants ;
- La génération de modèles UML depuis une ontologie ;
- L'intégration d'ontologies standards dans les modèles UML [123].

Depuis quelques années il y a de plus en plus d'intérêt à utiliser des ontologies pour communiquer de la connaissance dans les logiciels systèmes [124]. Cet intérêt s'explique par le fait de vouloir avoir un haut niveau formel de définition des concepts et des relations entre eux, permettant ainsi l'automatisation de tâches par les machines.

Principe

UML dispose de beaucoup d'outils permettant de faire de la conceptualisation de modèles. Le Model Driven Architecture et son architecture à quatre couches fournit une base solide pour définir le métamodèle de n'importe quel langage de modélisation. Il offre donc une fondation pour rapprocher l'ingénierie de logiciels et des méthodologies comme UML et les technologies du Web Semantique basées sur RDF et OWL. L'objectif de l'ODM est de permettre la modélisation d'ontologies en UML permettant ainsi un héritage de tout

l'outillage de ce dernier. En effet une fois qu'un langage de modélisation du Web Sémantique comme OWL est modélisé en MOF, ses utilisateurs peuvent utiliser les capacités de MOF pour la création, la gestion de modèles, la génération de code et l'interopérabilité avec des modèles MOF. Comme le montre la Figure 114, la RFP (Request For Procedure) d'ODM sollicitait les spécifications normatives suivantes :

- Un métamodèle compatible MOF2 pour la définition d'ontologies.
- Un profil UML2 pour permettre l'utilisation des notations UML pour la définition d'ontologies.
- Un langage de transformation du métamodèle vers le profil et de ODM vers les langages d'ontologies web tels que Description Logic, OWL DL.

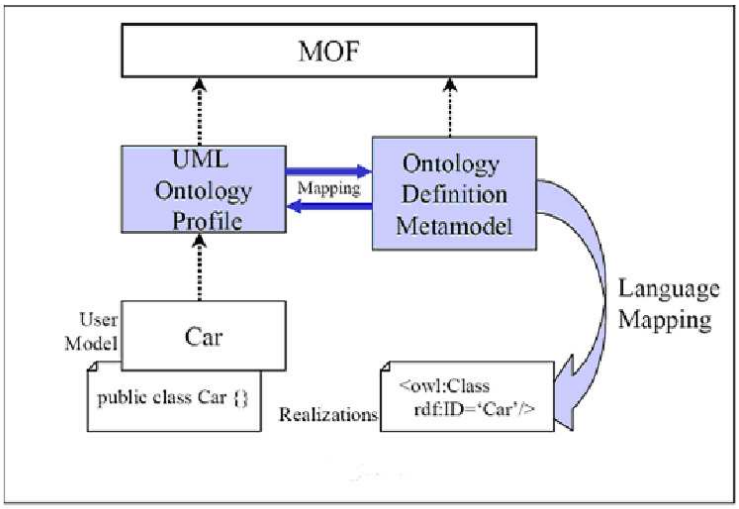

FIG. 114 – Principe de ODM

ODM est le fruit de recherches dans le domaine du MDA et des ontologies [126, 127, 128, 129, 130, 131, 132].
La soumission qui est acceptée et qui est en phase de finalisation dans le cycle de standardisation de l'OMG est celle de IBM et de Sandpiper Software [123]. Cette soumission offre beaucoup d'avantages :

- Permettre d'exprimer avec un niveau riche d'expressivité et de complexité, des modèles conceptuels dans les domaines que sont les méthodologies UML et ER (Entité Relation), ainsi que des ontologies formelles représentées en logique de description ou logique du premier ordre ;
- Permettre l'utilisation de moteurs de raisonnements pour comprendre, valider et appliquer les ontologies développées en utilisant ODM ;
- Fournir des profils et des mappings suffisants non seulement pour échanger des modèles développés en utilisant des formalismes différents mais aussi pour pouvoir les vérifier et les valider ;
- Fournir une famille de spécifications qui marient MDA et les technologies du Web

Sémantique pour permettre les Semantics Web services, la communication et l'interopérabilité d'ontologies.

Ainsi les ontologies à base de ODM peuvent être utilisées :

- Pour l'échange de connaissances entre des systèmes hétérogènes ;
- Pour représenter une connaissance en ontologies ;
- Pour la spécification d'expressions qui sont en entrée ou en sortie de moteurs d'inférence.

ODM n'a pas pour but :

- De spécifier une théorie de preuve ou de règles d'inférence ;
- De spécifier une translation ou des transformations entre les notations utilisées par des systèmes hétérogènes.

Il faut savoir qu'au départ il y avait plusieurs approches pour faire de la modélisation d'ontologies en UML. Ces approches pouvaient être classées dans deux approches [133] :

- Etendre UML avec de nouveaux concepts pour permettre les concepts spécifiques à l'ontologie (par exemple les propriétés) ;
- Utiliser le standard UML et définir un profil UML pour l'ontologie.

Le souci avec la première approche est qu'en modifiant UML et en standardisant les changements, les éditeurs UML actuels seraient obsolètes, ce qui ne sera pas le cas avec la deuxième approche car aucune modification ne sera apportée à UML. Cependant, comme ce fut le cas pour Action Semantics, il n'est pas exclu que les métamodèles de ODM soient incorporés un jour dans le métamodèle UML.

Vue d'ensemble de l'ODM

ODM sera composé de six métamodèles dont 4 normatifs et 2 informatifs. Les 4 métamodèles normatifs sont :

- Un métamodèle qui représente une description abstraite de RDF.
- Un métamodèle pour OWL ;
- Un métamodèle pour Topic Maps ;
- Un métamodèle pour Common Logic.

Dans la présente version de ce standard (Septembre 2007) il n'y pas encore de métamodèle pour Common Logic. La Figure 115, montre la structure des packages des métamodèles ODM.

Le document inclut aussi un métamodèle informatif pour les langages de type Description Logics.

Le document définit également 3 profils UML à utiliser dans ODM pour RDF, OWL et Topic Maps ainsi que des transformations exprimées en MOF QVT. Dans l'architecture à 4 couches du MDA, ODM se situe au même niveau que UML, c'est-à-dire au niveau 2 (Figure 116) [134].

Le principe de ODM est de définir les concepts individuels en MOF, créant ainsi un métamodèle ODM. Afin de permettre l'utilisation d'éditeur graphique UML pour manipuler ces concepts individuels qui constituent le métamodèle ODM, ODM définit aussi un profil

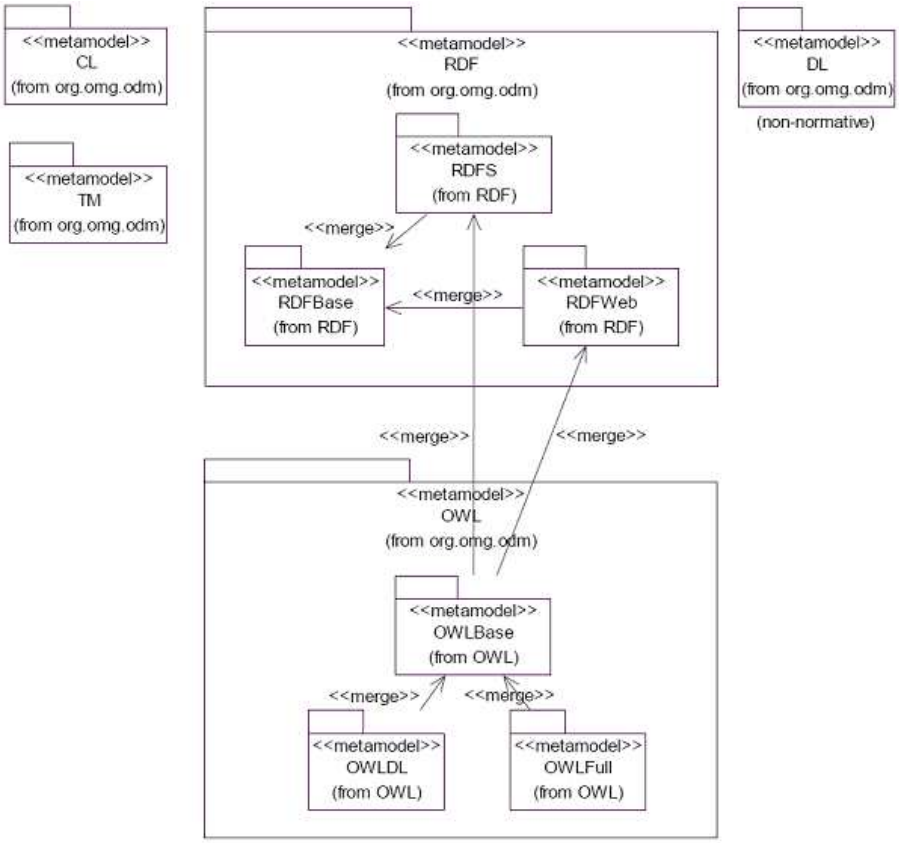

FIG. 115 – Structures des packages des métamodèles de ODM

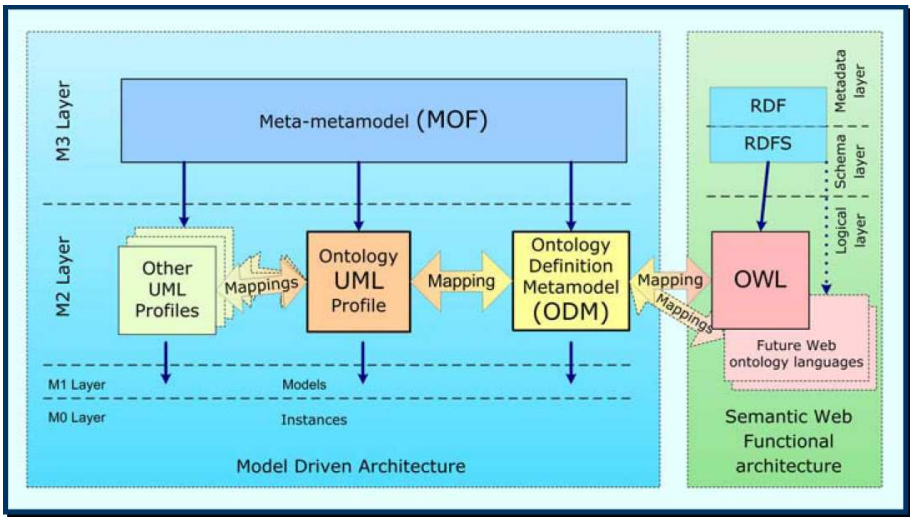

FIG. 116 – Modélisation d'ontologie dans le contexte du MDA et du Web Sémantique

UML pour les ontologies. Ceci permet aux éditeurs UML, très matures, d'être utilisés pour créer graphiquement des ontologies.

Pourquoi ne pas utiliser UML pour modéliser directement une ontologie ?

Finalement on peut se poser la question à savoir pourquoi ne pas utiliser UML pour modéliser des ontologies ? En effet à première vue UML semble suffire car un modèle d'ontologie ressemble beaucoup à un diagramme de classe. Cependant les raisons suivantes font que UML n'est pas assez riche pour modéliser complètement une ontologie [123] :

- UML ne permet aucune forme de traitement automatique de la sémantique ;
- En UML pour utiliser la notion de disjonction, il faut que les classes qui interviennent aient un super-type commun, ce qui n'est pas le cas en ontologie ;
- En UML une instance ne peut exister sans la classe qui définit sa structure alors qu'en ontologie, ceci est envisageable ;
- En UML une propriété, qui est un attribut ou une association, ne peut pas être un élément autonome au même titre qu'une classe (du fait de l'encapsulation) alors qu'en modélisation d'ontologies une propriété existe en dehors de toute structure. En UML les propriétés sont de seconde classe alors que dans les ontologies elles sont de première classe [130, 135].
- Monotonicité : les langages d'ontologies sont monotoniques alors que UML, un langage orienté objet est non monotonique. Un système est monotonique, si y ajouter de nouveaux faits, n'infère pas sur les précédents faits [129].
- Méta-niveaux : Les langages d'ontologies n'ont pas une séparation rigide entre les méta-niveaux. Par exemple, en OWL Full, une instance d'une classe peut être une autre classe. Ce qui n'est pas le cas de UML [126, 127].
- Spécialisation/généralisation : avec UML une relation de spécialisation/généralisation a pour but de permettre la réutilisation d'un comportement. Alors que dans les langages d'ontologies l'idée derrière une relation de spécialisation/généralisation est un ensemble théorique.
- Modularité : les langages d'ontologies n'ont pas de profils, de packages ou aucun autre mécanismes de modularité supportés par UML et les autre langages orientés objet [131].
- Conteneurs et listes : les packages UML ne sont pas au même méta-niveau que les conteneurs RDF (Bag, Seq et Alt). La manière de représenter les packages font qu'ils ne soient pas adaptés aux conteneurs RDF [128].
- Les langages d'ontologies peuvent construire des classes en utilisant des opérations booléennes (unions, intersection et complément) et des quantificateurs. En UML il n'y a pas de correspondant.
- Contraintes de cardinalité : cette incompatibilité vient de la nature première classe des propriétés d'ontologies. Dans les langages d'ontologies il est possible de spécifier une contrainte de cardinalité globale à un domaine d'une propriété, alors qu'avec UML la contrainte de cardinalité doit être spécifiée séparément pour chaque association vers

une propriété.
- Sous-propriété : les langages d'ontologies permettent à une propriété d'être une sous-propriété d'une propriété. UML offre la possibilité de dire qu'une association est une spécialisation d'une autre, bien que cette construction soit rarement utilisée.
- Espace de nommage (namespace) : les langages d'ontologies pour le Web Sémantique utilisent les URIs pour référencer leurs constructions. UML dispose de plusieurs manières de se connecter à ses constructions (par exemple chaque classe est un namespace de ses attributs, qui peuvent avoir une portée privée, protégée ou publique).

7.2.5 Avantages et inconvénients des solutions présentées

Maintenant que nous avons fait une présentation des éventuelles solutions permettant d'ajouter de la sémantique aux modèles conceptuels, nous allons voir leurs avantages et leurs inconvénients.

Le cas des profils UML

Nous avons vu que les profils UML permettent de spécialiser un modèle UML. Les profils UML peuvent être utilisés dans tous modèles UML et ne modifient pas la structure. Nous avons aussi vu que finalement un profil UML n'est rien d'autre qu'un ensemble d'étiquettes que l'on colle sur des modèles. Dans ce sens, pour traiter la sémantique dans des modèles on peut faire un raisonnement sur ces étiquettes. Par exemple on peut mettre en œuvre le raisonnement suivant : *Si l'étiquette est "Sécurisée" alors seules les classes ayant pour étiquette "Directeur" peuvent l'utiliser.* Un tel raisonnement est tout à fait possible, mais ce n'est pas un raisonnement automatique.
Un raisonnement automatique n'est possible que si la sémantique est formellement définie, ce qui n'est pas le cas pour les profils UML et donc pas exploitable pour une machine.

Le cas de Object Constraint Language (OCL)

L'Object Contraint Language permet d'exprimer n'importe quel type de contraintes sur des modèles UML. Par exemple nous pouvons exprimer la contrainte : *Avant de louer une voiture il faut vous assurer que la voiture est en état.* OCL semble donc être une bonne solution pour notre problématique qui est d'ajouter de la sémantique dans les modèles MDA, cependant ce n'est pas le cas. Le premier souci avec OCL c'est qu'il ne permet pas d'exprimer des opérations d'effets de bord, bien qu'ils existent de plus en plus de travaux pour y remédier. L'autre souci c'est que, comme les profils UML, la sémantique n'est pas formellement définie en OCL et donc n'offre pas de possibilité de traitement automatique par des machines.

Le cas de Action Semantics (AS)

Nous avons vu que Action Semantics a été standardisé pour résoudre les carences d'OCL, c'est-à-dire permettre d'exprimer aussi bien des opérations sans effet de bord que des opérations de création, de suppression et de modification d'éléments de modèles. Ainsi AS permet de spécifier pleinement le corps même des opérations dans un modèle UML. Nous avons vu aussi qu'Action Semantics n'existe que sous forme de métamodèle et qu'il n'existe pas de format concret (textuel) comme OCL, ce qui rend son utilisation complexe. Outre cette complexite d'AS, nous avons toujours le même souci que les profils UML et OCL, c'est-à-dire la sémantique n'est pas formalisée et donc n'offre aucune forme d'inférence automatique.

Le cas de Ontology Definition Metamodel (ODM)

Les mécanismes de UML ne sont pas des solutions adaptées, d'ailleurs ceci n'est pas une surprise car le métamétamodèle qui les définit ne s'intéresse pas du tout à la définition d'une sémantique formelle. D'un autre côté nous avons le domaine du Web Sémantique qui ne s'intéresse qu'au traitement automatique de la sémantique. Alors quoi de plus naturel que d'opter pour l'utilisation des techniques du Web sémantique pour rajouter de la sémantique aux modèles conceptuels.
En 2004, le W3C avait fini de mettre sur place OWL qui est une recommandation pour la définition d'ontologies [105]. Après l'étude de OWL nous nous sommes rendus compte qu'il était parfait pour nos besoins, à savoir définir une sémantique et en faire un traitement automatique. Cependant nous étions confrontés à un problème, qui était, comment concilier Web Sémantique et modèles MDA ? C'est là qu'intervient l'Ontologie Definition Metamodel qui, comme nous l'avons expliqué, a pour objectif de permettre la modélisation d'une ontologie en utilisant une instance du métamétamodèle Meta Object Facility (MOF). Dans notre approche, que nous allons expliquer dans la section suivante, le but est de permettre d'ajouter de la sémantique dans un modèle puis de faire un traitement automatique pouvant aboutir à la manipulation de règles métier. Pour l'instant, chaque domaine a ses propres outils, dans notre cas, avec l'utilisation de ODM, il est possible d'utiliser un même éditeur pour faire et des modèles UML et des ontologies. ODM nous permettra de passer du MOF au OWL et donc de pouvoir utiliser les mécanismes de traitement automatique.

7.3 Notre Approche

Nous avons expliqué que les règles métier utilisent des modèles, les modèles MDA ne disent absolument rien sur la sémantique. Le Web Sémantique, avec le concept de traitement automatique du contexte par des machines, laisse penser à une nouvelle vague d'applications très excitantes. Notre approche est de lier modèles MDA et Web Sémantique, en ajoutant de la sémantique aux modèles, dans le but de pouvoir générer des règles métier

ou faire d'autres traitements (par exemple fusion de bases de connaissance).

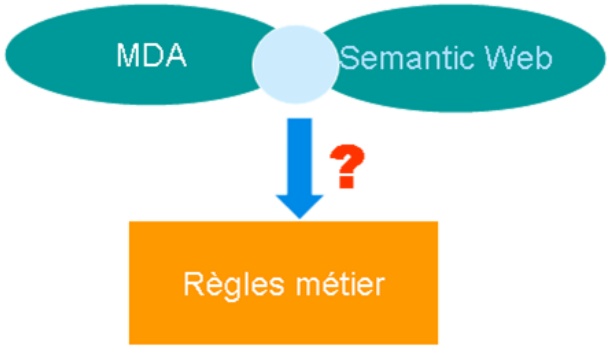

FIG. 117 – Combinaison de modèles MDA et Sémantique Web pour un traitement sur les règles métier

7.3.1 Génération de règles métier

Notre objectif est en rajoutant de la sémantique aux modèles, pouvoir générer des règles métier. Par exemple, à la Figure 118, nous avons un modèle qui conceptualise que chaque *Humain* a un père et une mère, qu'un père ne peut être que de type *Homme* et une mère de type *Femme*. En se basant sur ces assertions, nous aimerions pouvoir générer automatiquement les règles métier suivantes :

- Tout *Humain* doit avoir une mère et un père.
- Si *Humain1* est la mère de *Humain2* Alors *Humain1* est une *Femme*.
- Si *Humain1* est le père de *Humain2* Alors *Humain1* est un *Homme*.

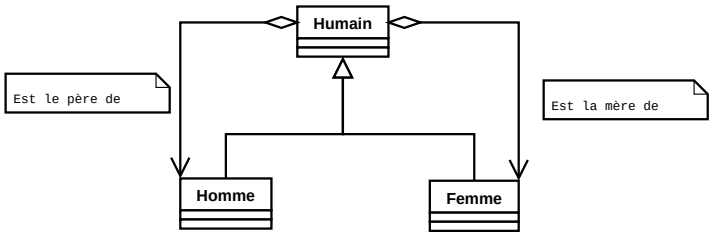

FIG. 118 – Exemple d'un petit modèle

Pour pouvoir faire une telle génération il est clair qu'il faille rajouter des informations sur ces modèles. Une telle opération est appelé enrichissement sémantique [136, 137, 138, 139, 140, 141, 142].
De manière plus concrète, comme le montre la Figure 119, se basant sur notre petit modèle de la la Figure 118, en y rajoutant une sémantique (ontologie simple dans cet exemple), alors pouvoir générer la règle *Si la mère du Christ s'appelle Marie alors Marie est une Femme*.

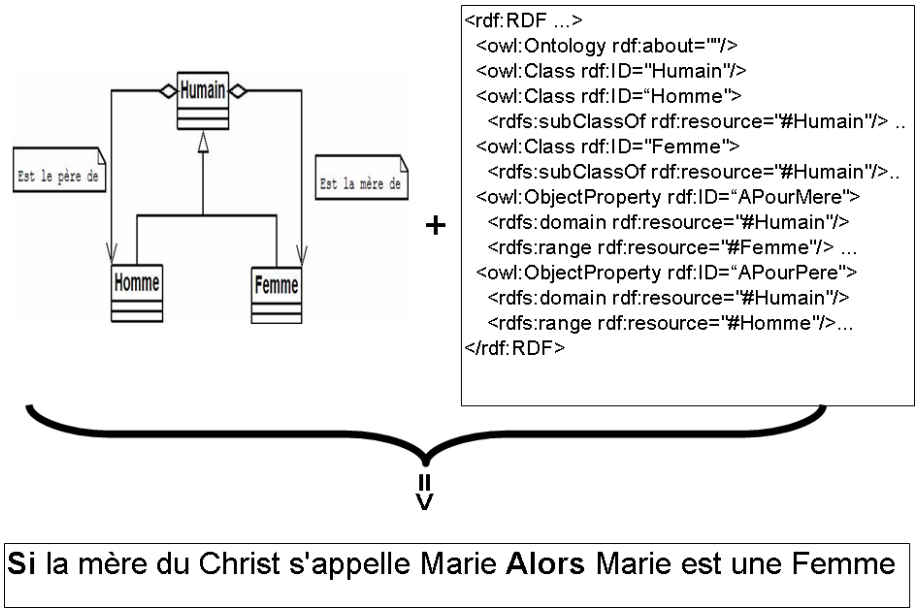

FIG. 119 – Injecter de la sémantique dans les modèles pour la génération de règles métier simples et génériques

7.3.2 L'architecture de notre approche

De manière générale, pour la génération de règles métier on se base sur les raisonnements de type *Tboxes* (concernant la hiérarchisation des classes et la définition des propriétés : transitive, fonctionnelle, symétrique, etc.) et *Aboxes* (concernant les assertions sur les individus).
De manière générale notre objectif consiste à combiner une sémantique (en OWL/RDF par l'intermédiaire de ODM) et un modèle MDA dans le but d'obtenir un modèle sémantiquement riche. A partir de ce modèle sémantiquement riche et avec l'utilisation de Java Métadata Interface [143] (en se basant sur le XMI [71] généré) ou d'un moteur de raisonnement, on pourra générer des règles métier simples. Notre architecture se divise en deux variantes.

Première variante : Utilisation massive de ODM

La Figure 120 montre l'architecture de la première variante de notre approche.
Notre procédé consiste : à partir d'un modèle en entrée, faire de l'enrichissement sémantique en utilisant n'importe quel outil UML acceptant l'import de profils UML (comme MagicDraw par exemple) pour OWL de ODM afin de rajouter de la sémantique formelle ou informelle. On obtient ainsi un modèle sémantiquement riche. A partir de ces informations on peut envisager 2 possibilités : utiliser JMI pour exploiter le XMI, mais là on perd de l'information car le XMI étant moins expressif que OWL. La seconde possibilité est de

transformer le modèle sémantiquement riche en modèle OWL/RDF en utilisant encore la spécification ODM. Enfin utiliser les caractéristiques de représentation de la connaissance de OWL/RDF et un moteur de raisonnement (comme racer [113, 121, 112]) pour générer des règles métier en se basant sur les propriétés et les instances.

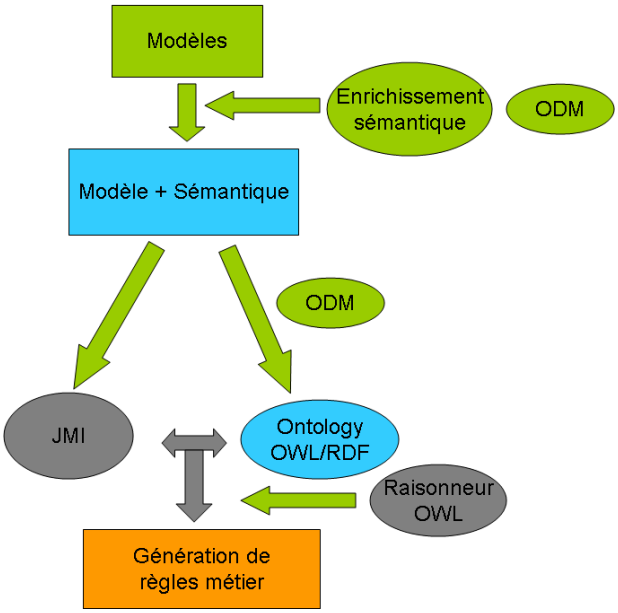

FIG. 120 – Archtecture de notre approche (Première variante)

Seconde variante : Utilisation d'éditeurs d'ontologies externes

La seconde variante consiste (voir la Figure 121) : à partir d'un modèle en entrée, d'utiliser ODM directement pour générer un modèle d'ontologie OWL. Ce modèle d'ontologie OWL sera enrichi en utilisant n'importe quel éditeur graphique d'ontologies (comme protégé [144] ou Swoop [145]) afin de rajouter de la sémantique formelle ou informelle. Nous obtenons à ce stade un modèle sémantiquement riche sous forme de modèle OWL/RDF. A partir de ce stade l'enchaînement devient la même chose que dans la première variante à savoir repasser en XMI ou utiliser un moteur de raisonnement pour générer des règles.

La génération de ces règles métier se fera en utilisant l'approche MDA. Nous allons, comme le montre la figure 122, générer les règles métier au format SBVR [47] au niveau de la couche CIM puis descendre vers les couches PIM en utilisant comme formalisme notre langage ERML, RIF du W3C, PRR de OMG ou RuleML [54, 49, 45]. Depuis la couche PIM, nous faisons une dernière génération vers la couche PSM afin d'avoir un format de règles exécutable. Cette dernière partie pour la génération d'un format exécutable a déjà été effectuée dans nos premiers travaux portant sur le formalisme Ecitiz Rule Markup Language (ERML) présentés à la section 3.10.

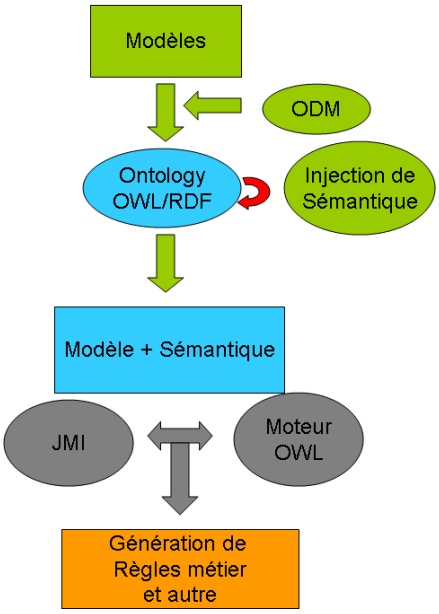

Fig. 121 – Archtecture de notre approche (Seconde variante)

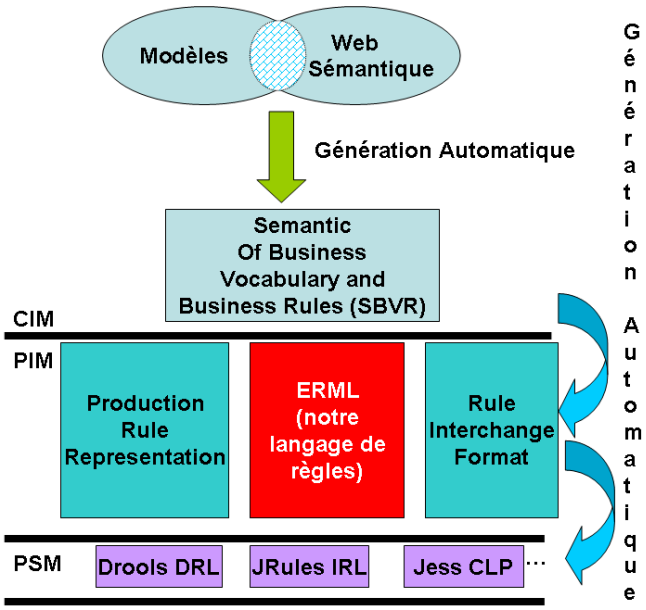

Fig. 122 – Archtecture de notre approche à travers les couches du MDA

Comment générer réellement des règles métier

Comme nous l'avons dit au paragraphe précédent, notre approche consiste à enrichir sémantiquement un modèle UML en vue d'utiliser les techniques du Web Sémantique pour la génération de règles métier. Une fois qu'on a enrichi le modèle UML en le transformant par ODM, on obtient à la fin un modèle OWL. Depuis ce modèles OWL nous utilisons la syntaxe et la sémantique des éléments OWL (T-box et A-box). Pour une propriété on peut définir un ou plusieurs axiomes qui fournissent des caractéristiques additionnelles. OWL supporte les constructions suivantes pour les axiomes sur les propriétés :

- Les éléments RDF schéma : *rdfs :subPropertyOf*, *rdfs :domain* et *rdfs :range*.
- Les relations avec les autres propriétés : *owl :equivalentProperty* et *owl :inverseOf*.
- Les contraintes globales de cardinalité : *owl :FunctionalProperty* et *owl :InverseFunctionalProperty*.
- Les caractéristiques de propriété logiques : *owl :SymmetricProperty* and *owl :TransitiveProperty*
- La déclaration des individus (instances).

Dans les paragraphes suivants, nous allons détailler comment à partir de chaque axiome sur les propriétés et les individus, en utilisant aussi bien leur sémantique que leur syntaxe, nous pouvons générer des règles métier.

Sous-propriété d'une propriété : Un axiome *rdfs :subPropertyOf* définit qu'une propriété est une sous-propriété d'une autre propriété. Sémantiquement si nous avons $P2$ *rdfs :subPropertyOf* $P1$ cela veut dire que l'ensemble des paires $\{sujet, objet\}$ de $P2$ est inclus ou égal à l'ensemble des paires $\{sujet, objet\}$ de $P1$. Ainsi nous pourrons générer des règles ressemblant à :

SI
*Predicat*2 *rdfs* :*subPropertyOf* *Predicat*1
ALORS
*Le domaine de Predicat*2 *est* $\subseteq$ *le domaine de Predicat*1
ET
*Le rang de Predicat*2 *est* $\subseteq$ *le rang de Predicat*1

Domaine et rang d'une propriété : Pour une propriété, on peut définir un ou plusieurs domaines et\ou rangs en utilisant respectivement *rdfs :domain* et *rdfs :range*. D'un point de vue syntaxe, *rdfs :domain* et *rdfs :range* sont des déclarations qui lient une propriété à une classe. Les axiomes de type *rdfs :domain* affirment que les sujets de la propriété concernée doivent appartenir à la classe indiquée par l'axiome. Les axiomes de types *rdfs :range* affirment que les objets de la propriété doivent appartenir à la classe indiquée par l'axiome. Ainsi nous pourrons générer que :
si nous avons :

$$\text{Predicat} : \text{Domaine1} \longmapsto \text{Range1}$$

Alors générer la règle :

SI	Objet1	Predicat	Objet2
ALORS	Objet1	est de type	Domaine1
ET	Objet2	est de type	Range1

Par exemple si nous avons :

$$\text{APourMère} : \text{Humain} \longmapsto \text{Femme}$$

Alors générer la règle :

SI	Objet1	APourMère	Objet2
ALORS	Objet1	est de type	Humain
ET	Objet2	est de type	Femme

Axiome d'équivalence : La construction *owl :equivalentProperty* peut être utilisée pour spécifier que deux propriétés peuvent avoir le même ensemble de paires {sujet, objet}, c'est-à-dire que les propriétés ont les mêmes domaines et les mêmes rangs. Cela ne veut nullement dire que les deux propriétés sont les mêmes, pour cela il existe la construction *owl :sameAs*. Avec cette construction nous pourrons générer les règles suivantes :
Si nous avons :

$$Predicat1 : Domaine1 \longmapsto Range1 \; et \; Predicat2 \; owl :sameAs \; Predicat1$$

Générer alors que :

SI
*Object*1 *Predicat*2 *Object*2
ALORS
*Object*1 *est de type Domaine*1
ET
*Object*2 *est de type Range*1

Ici nous pouvons remarquer que nous n'avons pas d'information sur *Predicat2*, mais que sa sémantique nous permet d'inférer la génération de règles. Comme illustration prenons l'exemple suivant :
aPourMère : Humain $\longmapsto$ Femme et aPourMaman *owl :equivalentProperty* aPourMère
Générer :

SI	Objet1	aPourMaman	Objet2
ALORS	Objet1	est de type	Humain
ET	Objet2	est de type	Femme

Axiome d'inverse : Les propriétés ont un sens, c'est-à-dire qu'une propriété va de domaine vers rang. Dans la pratique il est bien de pouvoir définir une relation dans les deux sens par exemple : *une personne possède une voiture*, *une voiture appartient à une*

personne. Pour définir un axiome d'inverse, on utilise *owl :inverseOf*. Syntaxiquement cela veut dire que si une proprété $P2$ est inverse à une propriété $P1$ alors pour l'ensemble de paires $\{sujet, objet\}$ de $P1$ correspond l'ensemble des paires $\{objet, sujet\}$ de $P2$ et vice versa. Avec cette construction, on peut alors générer des règles du genre :

$$Predicat1 : Domaine1 \longmapsto Range1 \text{ } et \text{ } Pridicat2 \text{ } owl :inverseOf \text{ } Predicat1$$

Générer alors que :

SI
*Object*1 *Predicat*2 *Object*2
ALORS
*Object*1 *est de type Range*1
ET
*Object*2 *est de type Domaine*1

Comme illustration prenons l'exemple suivant :
PossedeVoiture : Humain $\longmapsto$ Voiture et VoitureAppartient *owl :inverseOf* PossedeVoiture
Générer :

SI	Objet1	VoitureAppartient	Objet2
ALORS	Objet1	est de type	Voiture
ET	Objet2	est de type	Humain

Axiome d'unicité : Une propriété fonctionnelle est une propriété qui ne peut avoir qu'une seule valeur pour chaque instance ; cela veut dire qu'il ne peut pas y avoir de $y1$ et $y2$, avec $y1$ différent de $y2$ telqu'il existe les paires $(x, y1)$ et $(x, y2)$ pour la propriété fonctionnelle. Une propriété est déclarée fonctionnelle en utilisant la construction *owl :functionalProperty*. Avec un axiome d'unicité on peut générer des règles du genre :

$$Predicat1 \text{ } owl :functionalProperty$$

Générer alors que :

SI
*Object*1 *Predicat*1 *Object*2
ET
*Object*1 *Predicat*1 *Object*3
ALORS
*Object*2 *et Object*3 *sont les mêmes*

Comme illustration prenons l'exemple suivant :
aPourMère : Humain $\longmapsto$ Femme et aPourMère *owl :functionalProperty*
Générer :

SI	Objet1	aPourMère	Objet2
ET	Objet1	aPourMère	Objet3
ALORS	Objet2 et Objet3	sont les	mêmes

Axiome d'unicité inverse : Une propriété peut être déclarée inversement fonctionnelle, alors l'objet (valeur) permet de déterminer de manière unique le sujet de la propriété ; cela veut dire qu'il ne peut pas y avoir de $x1$ et $x2$, avec $x1$ différent de $x2$ telqu'il existe les paires $(x1, y)$ et $(x2, y)$ pour la propriété inversement fonctionnelle. Une propriété est déclarée inversement fonctionnelle en utilisant la construction *owl :inverseFunctionalProperty*. Avec un axiome d'unicité inverse on peut générer des règles du genre :

$$Predicat1\ owl : inverseFunctionalProperty$$

Générer alors que :

SI
*Object*1 *Predicat*1 *Object*
ET
*Object*2 *Predicat*1 *Object*
ALORS
*Object*1 *et Object*2 *sont les mêmes*

Axiome de transivité : Lorsqu'on définit une propriété P comme étant transitive, cela veut dire que s'il existe une paire (x, y) de P et un autre paire (y, z) de P, alors on peut en déduire qu'il existe une paire (x, z) de P. On définit une propriété comme étant transitive en utilisant la construction *owl :TransitiveProperty*. Avec une axiome de transivité, on peut générer les règles suivantes :

$$Predicat1\ owl : TransitiveProperty$$

Générer alors que :

SI
*Object*1 *Predicat*1 *Object*2
ET
*Object*2 *Predicat*1 *Object*3
ALORS
*Object*1 *Predicat*1 *Object*3

Axiome de symétrie : Une propriété P est symétrique si pour toute paires (x, y) instance de P alors (y, x) est aussi instance de P. La construction qui permet de rendre une propriété symétrique est *owl :SymmetricProperty*. Pour une propriété symétrique, le domaine et le rang sont les mêmes. Avec une axiome de symétrie, on pourra générer des règles du genre :

$$Predicat1\ owl : SymmetricProperty$$

Générer alors que :

SI
*Object*1 *Predicat*1 *Object*2
ALORS
*Object*2 *Predicat*1 *Object*1

Notre objectif n'est pas d'inventer ou de créer une connaissance mais plutôt de rendre explicite une connaissance implicite.
Dans la partie implémentation, au chapitre 9, nous montrerons comment nous avons concrètement essayé d'implémenter notre approche. La figure 123 donne une idée de ce que nous avons dans le cadre d'un prototype simple.
Notre approche d'enrichissement sémantique de modèles peut également servir à d'autres finalités que la génération de règle métier, par exemple à fusionner et à aligner des règles métier (bases de connaissance) comme nous l'avons montré dans l'une de nos publications [141].

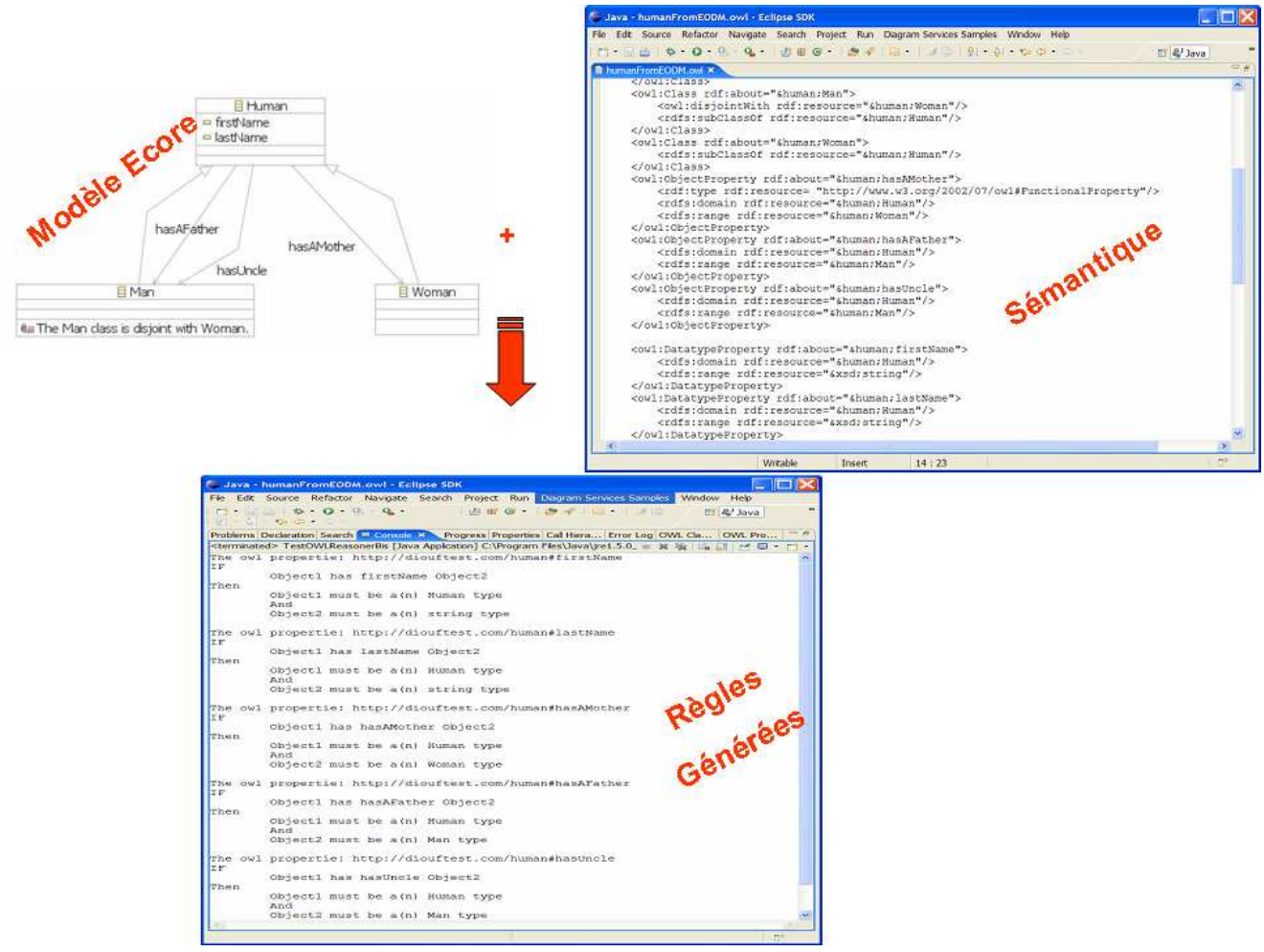

FIG. 123 – Prototypage de notre approche

7.3.3 Fusion de règles métier

Le partage d'information est la clé principale des systèmes d'information coopératifs. L'un des avantages particuliers au partage d'information entre plusieurs systèmes est qu'il est possible de déduire de la connaissance additionnelle qui n'est pas disponible en local à aucun système, mais disponible de manière collective sur tous les systèmes.
Le Web Sémantique sera l'un des systèmes ouverts les plus grands au monde. Les informations et autres connaissances vont provenir d'un peu partout.
Les bases de connaissances qui interviennent dans la couche logique seront amenées à être fusionnées ou alignées. Ce travail doit être réalisé, que l'objectif soit d'avoir une seule base de connaissance cohérente qui intègre toutes les autres sources (**fusionner**), soit de rendre une source consistante et cohérente avec une autre source mais en les gardant séparées (**aligner**) [146]. Lors de la fusion ou l'alignement de bases de connaissances, plusieurs problèmes peuvent survenir amenant à une pollution de la connaissance. La modularité dans les logiques de programmation est, depuis les années 90, activement investiguée [147].
Une autre application de notre approche pourrait être un système d'aide à la fusion ou l'alignement des règles métier dans une base de connaissances.
Comme le montre la figure 124, supposons que nous ayons un modèle M_A avec une base de donnée BD_A dans laquelle nous avons les champs $\{Nom, Ville\ de\ naissance\}$ et un ensemble de règles métier BC_A avec les règles R_{A_1} et R_{A_2} :
R_{A_1} : *Si* Nom est vide *Alors* afficher une erreur.
R_{A_2} : *Si* Ville de naissance est vide *Alors* afficher une erreur.
Supposons aussi avoir un autre modèle M_B et une base de données BD_B avec les champs $\{Nom, Ville\ de\ résidence\}$ et une base de connaissances BC_B avec les règles R_{B_1} et R_{B_2} :
R_{B_1} : *Si* Nom est vide *Alors* revenir à l'étape précédente.
R_{B_2} : *Si* Ville de résidence est vide *Alors* revenir à l'étape précédente.
Imaginons que l'on doive écrire une application qui va se baser sur les modèles M_A et M_B ainsi que sur les bases de données BD_A et BD_B, appelons les M_{A+B} et BD_{A+B}. En fusionnant les modèles et les bases de données, il devient évident de vouloir en faire autant pour les bases de connaissances (ici, les règles métier).
La fusion de bases de connaissances pose un sérieux problème de pollution de connaissance parce que des règles peuvent être contradictoires. Par exemple en voulant fusionner BC_A et BC_B, nous pouvons voir que R_{A_1} et R_{B_1} ont la même condition mais pas la même action ; dans ce cas que fait-on ?
Actuellement ce travail d'alignement et de fusion des règles métier se fait à la main, il n'y pas d'outils pour automatiser partiellement ou globalement cette procédure. Cette opération manuelle est particulièrement laborieuse et coûteuse en temps. Il est facile de voir que dans cette procédure de fusion et d'alignement, beaucoup de mécanismes pourraient être automatisés totalement ou partiellement.
Déjà arriver à détecter ce genre de problèmes serait une bonne avancée. Nous pouvons aussi voir que R_{A_2} et R_{B_2} sont disjointes donc on pourra les réutiliser dans BD_{A+B}, sauf qu'il faudra par exemple réécrire R_{A_2} du modèle M_A vers le modèle M_{A+B} et R_{B_2} du modèle M_B vers le modèle M_{A+B}.

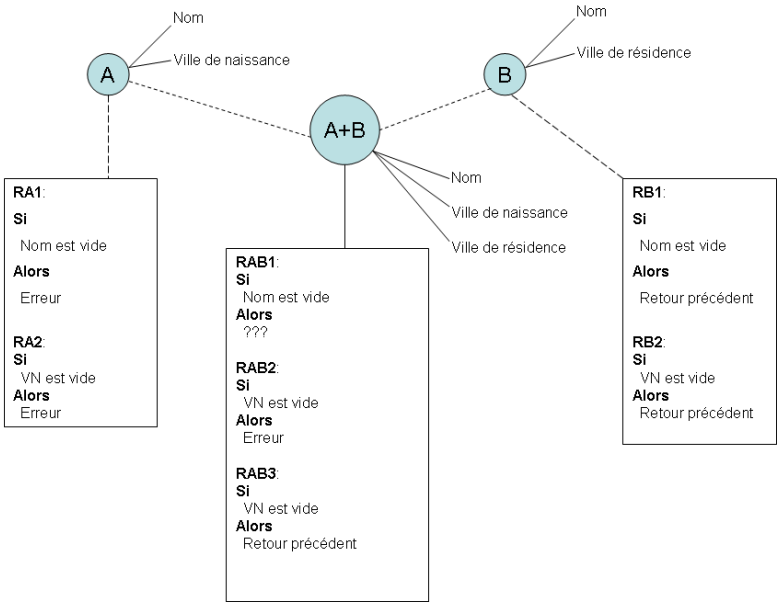

FIG. 124 – Fusion de règles métier

Nous pouvons voir depuis notre exemple que les principaux problèmes que pose la fusion de connaissances sont, premièrement la détection de contradiction ou de similarité et deuxièmement la réécriture de règles.
En utilisant l'approche par injection de connaissances dans les modèles conceptuels, nous pourrions proposons une manière sémi-automatique basée sur la sémantique et non sur la syntaxe pour fusionner et aligner des bases de connaissance. Le principe est de ramener ce problème, à un problème connu et déjà résolu, à savoir la fusion d'ontologies. Comme pour le cas de la génération de règles métier, les modèles concernés seront transformés en modèles d'ontologies simples avec ODM. Ensuite il faudra enrichir sémantiquement les ontologies, puis utiliser l'algorithme de PROMPT pour fusionner et aligner ces ontologies [146]. Le principe de PROMPT est de prendre deux ontologies en entrée et guider l'utilisateur à les fusionner en une seule ontologie. Tous les termes et concepts constituant les règles métier vont constituer les ontologies, ainsi fusionner les règles reviendra à fusionner les ontologies, ce que nous savons faire.

7.4 Conclusion

L'intitulé de cette thèse est *"Spécification et mise en œuvre d'un langage de modélisation de règles métier"* et l'objectif initial était d'avoir un formalisme de règles métier indépendant de toute plateforme puis de pouvoir générer les règles dans le formalisme d'un moteur de règles bien ciblé (JRules, Drools, Blaze, etc) [11].
Dès le départ, notre approche suivait l'ingénierie dirigée par les modèles (Model-Driven

Architecture), car dans l'architecture multi-couches, notre langage de formalisme de règles (Ecitiz Rule Markup Language) se trouve au niveau M2 ou PIM. A partir de la nous avons un moteur de transformation de modèle ERML vers du modèle Drools, JRules et Jess. Ce moteur de transformations de modèles est écrit en XSLT [11].
En 2005 l'Object Management Group (OMG) et le World Wide Web Consortium (W3C) se sont intéressés de très prés à la standardisation d'un formalisme de règles métier [54][49]. Ces processus de standardisation et nos travaux portant sur le Web Sémantique nous ont amené à reconsidérer nos plans de recherche. Dans la première partie de cette thèse nous avons vu que la phase d'élicitation et d'extraction des règles prend du temps ainsi que leur écriture. Alors nous nous sommes intéressés à générer automatiquement une partie de ces règles en se basant sur l'existant. En effet, en général, les entreprises capitalisent leur connaissances sous forme de modèles depuis longtemps. Notre objectif est, en se basant sur ces modèles et en les enrichissant sémantiquement avec les technologies du Web Sémantique, de pouvoir faire de l'inférence sur ce modèle sémantiquement riche pour faire de la génération de règles métier. Ces modèles hérités sont souvent en UML, alors nous nous sommes posés la question à savoir comment faire pour y rajouter de la connaissance. Nous avons montré que UML ne possède pas de mécanisme permettant de traiter la sémantique car UML, du fait de son appartenance au domaine du génie logiciel, est orienté contenu et non contexte donc n'offre pas de possibilité de traitement automatique de la sémantique. Nos travaux sur le Web Sémantique nous ont montré qu'en utilisant RDF/OWL une solution était possible. En RDF et OWL, une connaissance est simplement une collection d'assertions, chacune d'elles étant composées d'un sujet, d'un verbe et d'un objet et de rien d'autre. Ces triplets peuvent être directement utilisés dans des applications en les injectant dans la mémoire de travail d'un moteur d'inférence afin de faire un raisonnement. Dans notre cas la finalité de ce raisonnement est de générer des règles.
La standardisation de ODM est arrivée au bon moment et nous permet, depuis un modèle UML, d'avoir son équivalent en modèle d'ontologie OWL. Sur ce modèle, nous faisons de l'enrichissement sémantique et l'utilisation d'un moteur de raisonnement comme Racer nous permet de générer des règles métier dans un langage non exécutable (proche de SBVR) au niveau de la couche CIM dans un premier temps. Ensuite nous utilisons notre langage ERML ainsi que le moteur de transformations de modèles pour descendre vers les couches PIM et PSM. Le jour où les travaux de l'OMG(MDA, PRR) et du W3C(RIF) deviendront disponibles, il sera possible de les utiliser directement à la place de ERML ou de faire une transformation de ce dernier vers le modèle standardisé en utilisant QVT.
Nous avons également montré dans ce chapitre comment notre approche pourrait être utilisée pour proposer un système sémi-automatique d'aide à la fusion ou l'alignement de bases de règles métier en utilisant l'algorithme PROMPT.
Nous savons que vouloir générer toutes les règles métier est utopique (surtout lorsque le métier est complexe), mais la génération d'une partie facilitera beaucoup le travail d'extraction qui est celui des experts métier. Pour l'instant nous arrivons uniquement à générer des règles de contrainte et des règles de dérivation. Ceci est dû au fait que pour l'instant, pour la génération des règles, nous nous basons uniquement sur les TBoxs et les ABoxs. Pour pouvoir générer d'autres types de règles il faudra se baser sur d'autres fondements

que les structures RDF/OWL pour raisonner.
Notre approche a fait l'objet de plusieurs publications, en majorité internationales [136, 137, 138, 139, 140, 141, 142]. Nous avons aussi implémenté un prototype pour valider notre approche que nous présentons dans la partie suivante sur l'implémentation au chapitre 9. Dans cette partie, nous allons également voir comment nous avons procédé pour intégrer nos travaux sur le formalisme de règles métier dans la plateforme d'e-services *e-Citiz*.

Cinquième partie

Implémentation

Chapitre 8

L'intégration de l'approche par règles métier et du langage ERML dans *e-Citiz*

Sommaire

8.1 Introduction

Au départ de nos travaux, nous travaillions de manière complètement séparée du contexte *e-Citiz* car, d'un côté nous ne voulions pas être uniquement influencés par les besoins d'*e-Citiz* mais plutôt avoir une vision globale et générale du problème considéré et dans un deuxième temps spécialiser la solution dans le cadre d'*e-Citiz*. D'un autre côté, intégrer dès le départ la plateforme *e-Citiz*, pouvait être lourd pour nous et nous contraindre. Une fois le langage de règles ERML établi, il fallait l'intégrer dans *e-Citiz*, héritant d'avantages (API déjà existant) et de contraintes (contraintes sur certains choix technologiques déja établis).
L'intégration dans *e-Citiz* s'est effectuée en 3 phases suivant le studio, le générateur et le moteur. Nous allons voir dans les sections suivantes une description technique d'*e-Citiz*, comment s'est passée cette intégration et les choix que nous avons eu à faire.
Nous rappelons que pour une meilleure compréhension de ce chapitre il faudrait se référer à la section 1.2 sur le contexte de la thèse.

8.2 Description technique de *e-Citiz*

8.2.1 Architecture

Les e-services développés avec *e-Citiz* implémentent une architecture applicative multi-couche (N-tiers) J2EE. La Figure 125 montre l'architecture de *e-Citiz*. La couche Présen-

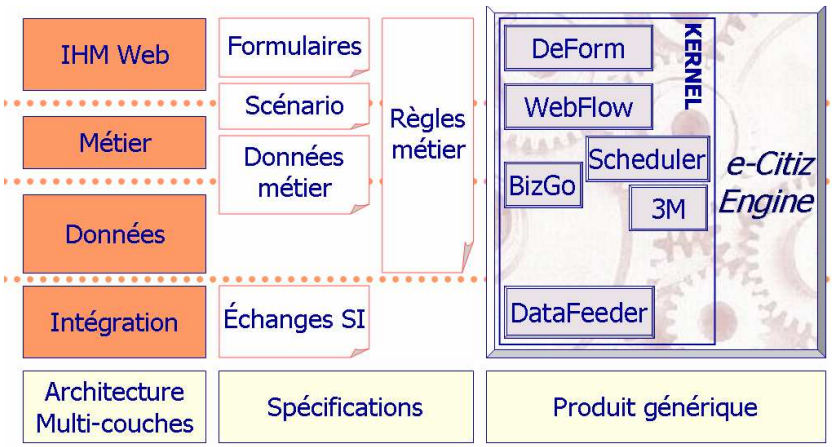

FIG. 125 – Architecture J2EE d'*e-Citiz*

tation est implémentée par des pages JSP et les spécifications techniques des formulaires sont implémentées en XML. Cette couche s'appuie également sur les standards suivants : XML Schema, Xpath, XHML, PDF, CSS, XSLT, HTTPS.

La couche Coordination (ou Contrôle) est implémentée par un moteur de workflow interactif (WebFlow) de type MVC2 et ses spécifications techniques utilisent XML. Cette couche s'appuie également sur les standards Servlet et HTTPS.
Les couches Services et Domaine sont implémentées par des classes Java. Les classes du Domaine métier, appelé BizGo, sont persistantes en base de données par JDO (Java Data Object). Cette couche de Persistance garantit une portabilité en terme de base de données. Le schéma conceptuel de données de la base est déduit de la modélisation métier réalisée avec le Studio (approche top-down). Ceci apporte une flexibilité de spécification et surtout de l'évolution par rapport à une approche classique de réalisation de la base puis du modèle métier (bottom-up). C'est l'un des avantages de *e-Citiz* et de sa démarche MDA (Model-Driven Architecture) standardisée par l'OMG. Le schéma de la base de données permet l'écriture rapide de scripts SQL par un Administrateur de Base de Données (DBA). La Figure 126 montre l'architecture par composants d'*e-Citiz*.

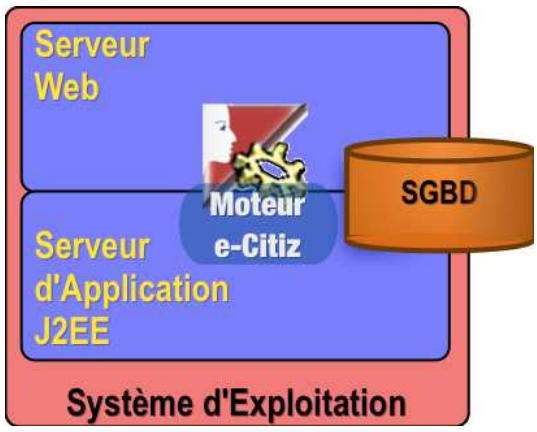

Fig. 126 – Architecture par composants d'*e-Citiz*

8.2.2 Infrastructure logicielle

e-Citiz s'appuie sur des standards (J2EE, XML, ...) et permet un déploiement des e-services produits sur la plupart des infrastructures logicielles disponibles (telles que JBoss, Weblogic, WebSphere, PostgreSQL, Oracle, mySQL, ...). Par exemple la plateforme cible suivante, utilisant exclusivement des logiciels libres, est proposée :

- serveur Web : Apache Tomcat ;
- serveur d'application J2EE : JBoss qui peut être supervisé par tout outil JMX ;
- système d'exploitation du serveur Web et du serveur de données : Linux ;
- système de gestion de base de données : la base de données relationnelle PostgreSQL.

8.3 Modèle objet de règles

Une fois les travaux sur le langage de règle terminés, il restait à l'intégrer dans le Studio *e-Citiz*. La grammaire a été modélisée en XSD et donc les règles sont stockées en XML. Pour manipuler les règles dans *e-Citiz* deux choix étaient possibles. Le premier était de manipuler directement le XML en utilisant des interfaces comme DOM ou SAX. Ce premier choix fût vite rejeté car était lourd et difficile à maintenir. Le deuxième choix était d'utiliser une solution d'association de données ("Data Binding" en anglais) qui permet de compiler XSD en une ou plusieurs classes. Rapidement notre choix s'est porté sur le deuxième car plus simple et était déjà utilisé dans *e-Citiz*. Il existe plusieurs API pour faire de l'association de données, notre choix se porta sur une solution de Sun qui s'appelle JAXB (Java XML Binding) [148] car également déjà utilisée dans *e-Citiz*. Ainsi un modèle de règles a été généré en utilisant JAXB. Cependant un problème bien connu avec les solutions d'association de données comme JAXB est celui du "cast" qui pousse souvent à faire beaucoup de "instanceof" pour savoir le type de l'objet manipulé. Il était plus judicieux d'avoir un autre modèle plus lisible et de faire un mapping du modèle propre vers le modèle généré pas JAXB. Nous allons maintenant voir ce modèle objet de règles ERML qui est calqué sur le langage ERML.

8.3.1 Le package *Repository*

La classe de plus haut niveau dans le modèle est le *coreModel*. Dans un système d'édition de règles métier deux entités sont prises en compte : le modèle de règle et le modèle d'objet métier qui est l'ensemble des éléments sur lesquels on se base pour écrire les règles (Business Object Model : BOM). Le *coreModel* est composé d'un *repository*. Comme le montre la Figure 127 , un *Repository* est composé d'un *Ruleset* et d'un *BusinessModel*. Un *Ruleset* est composé d'une liste de variables globales (*GlobalVariableIterator*) pouvant être utilisée dans les règles, d'une liste d'imports (*ClassImportIterator*), d'une liste de règles (*RuleIterator*) et d'une propriété (*RulesetProperties*).

8.3.2 Les packages *Rule* and *RuleTemplate*

Comme le montre la Figure 128, un *Rule* est composé de propriétés (*RuleProperties*), d'une liste de variables (*VariableIterator*) qui sera utilisée dans la règle, d'une liste de prémisses (*PremiseIterator*) et d'une liste d'actions (*ActionIterator*). Une *RuleTemplate* sert à créer des modèles, au sens exemple, de règles qui serviront à accélérer l'écriture de règles. En effet il arrive souvent que des règles aient beaucoup de similitudes et qu'il y ait juste une condition qui les différencient. Dans ce cas il serait plus judicieux d'avoir un éditeur te type tableur. Une règle peut avoir un nom, une priorité (ou salience), un booléen pour dire s'il est visible ou pas, un booléen pour dire si la règle est autorisée à boucler ou pas, une date de dernière modification et une date de création.

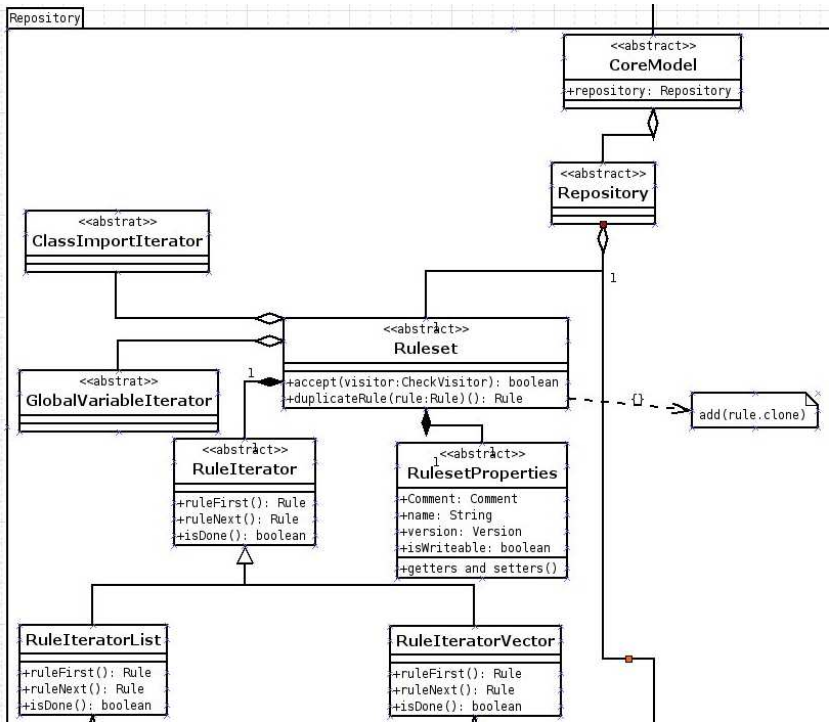

Fig. 127 – Package *Repository*

Prémisse ou condition de règle

Une règle peut avoir une ou plusieurs conditions. Une condition est appelée prémisse. Il existe plusieurs types de prémisses (Figure 129) :

- prémisse simple (*PremiseSimple*) qui est composée d'une seule expression ;
- prémisse négative (*PremiseNot*) qui est une expression avec une négation ;
- *PremiseAnd* et *PremiseOr* pour formuler des conjonctions et des disjonctions.

Modèle d'expression

Il existe deux types d'expressions : les expressions simples et les expressions complexes (Figure 129). Une expression simple (*SimpleExpressionCore*) est une expression qui a un opérateur basique et deux opérandes. Une expression complexe (*ComplexExpressionCore*) peut avoir n'importe quel opérateur et n'importe quel nombre d'opérandes.

Modèle d'opérateur

Il existe plusieurs types d'opérateurs :

- *OperatorBasicCore* qui sont les opérateurs par défaut du système. Ces opérateurs sont ceux arithmétiques et d'autres sur des dates et des chaînes de caractères pour

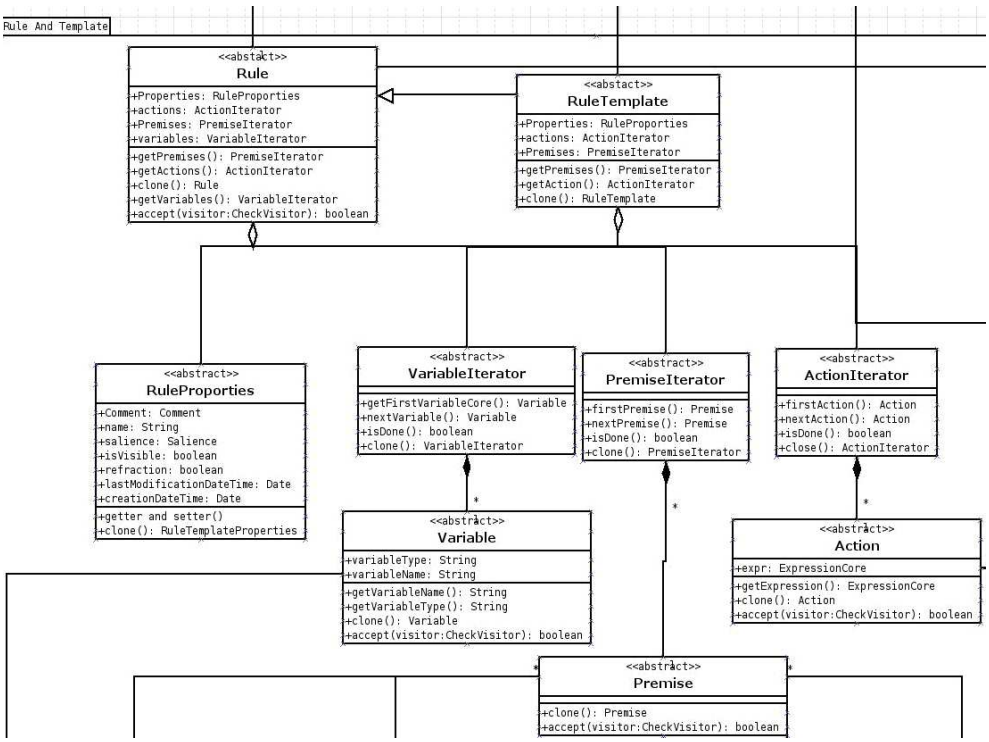

FIG. 128 – Package des règles

l'instant.

- *OperatorNatifCore* qui sont les opérateurs natifs aux moteurs de règles. Il s'agit des opérateurs sur l'engine à savoir l'assertion, la modification et la suppression de faits dans l'agenda.
- *OperatorPrintCore* qui sont les opérateurs d'écriture sur les sorties standard.
- *OperatorUnaryCore* qui permet de faire des construction de la forme *new Instance()* ;
- *RelationCore* qui permet de faire des appels de procédures attachées sous la forme *objet.Method*. Une *RelationCore* peut avoir une variable, une méthode sous forme de chaîne de caractères mais aussi une autre *RelationCore* pour faire des constructions du genre *(object.Method()).Method2()*.

8.3.3 Le package *Ruleset and Rule Tools*

Comme nous l'avons dit au départ de cette section, ce modèle de règle a été créé dans le but d'avoir un modèle plus intuitif que le modèle généré par JAXB. Afin de pouvoir sérialiser/désérialiser les objets en XML, il est nécessaire d'avoir une transformation de modèle entre notre modèle objet de règle et le modèle objet généré. Comme le montre la Figure 130, ces opérateurs de mapping sont effectués par *ExportRuleset* et *ImportRuleset*. Le modèle propose également un moyen de faire des vérifications avant les sauvegardes.

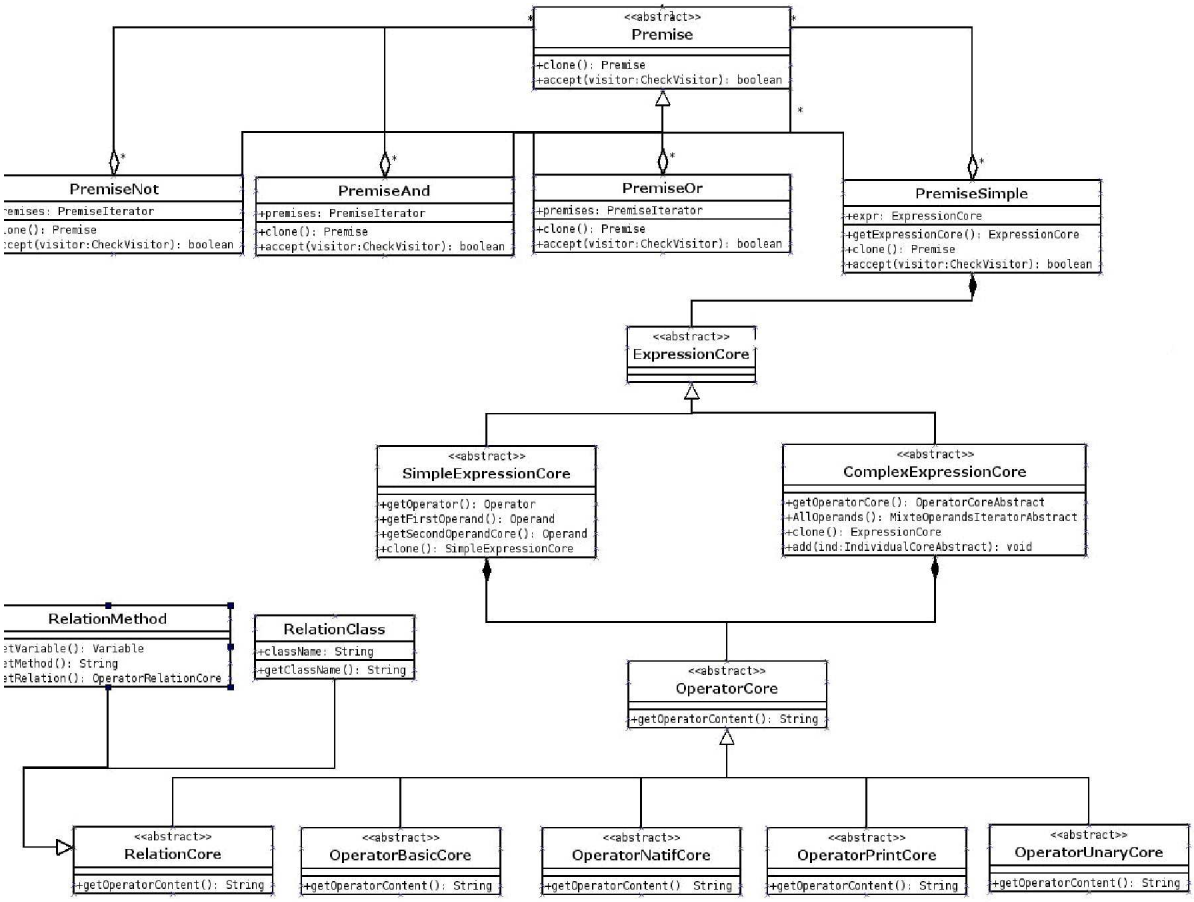

Fig. 129 – Package *Premise*

Ces vérifications se font sous forme de visiteur pour les éléments *Ruleset,Rule* et *Variable.*

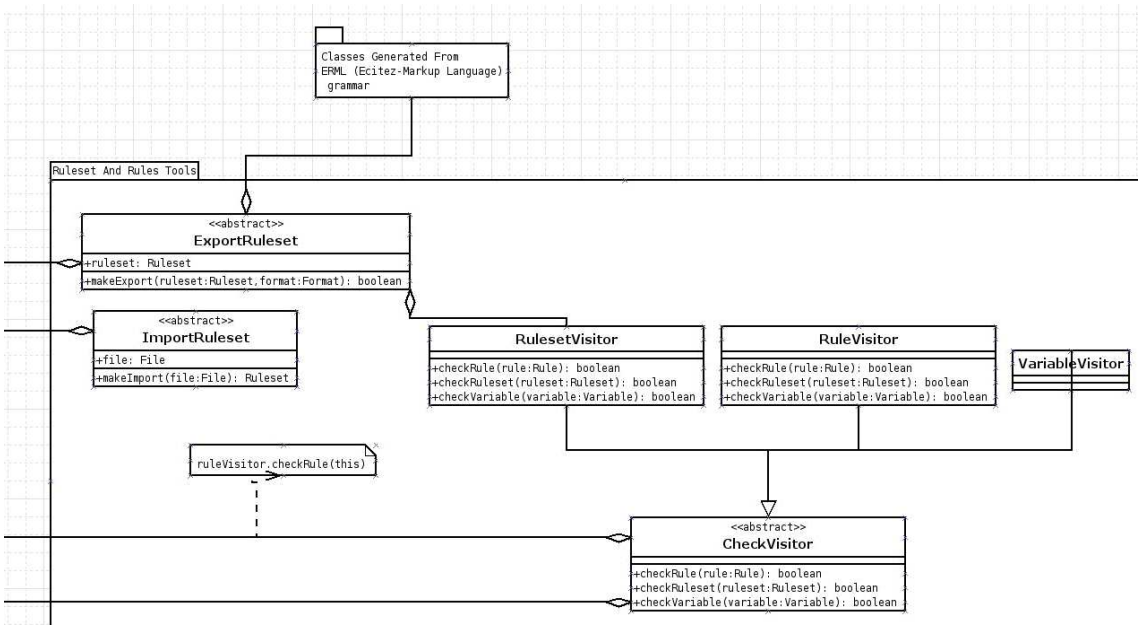

FIG. 130 – Package *Ruleset And Rules Tools*

8.3.4 Le package *Business Model*

Nous venons de présenter le modèle de règle pour stoker les règles métier, cependant pour les écrire, nous avons besoin d'un modèle objet métier (BOM). Ce dernier a la même structure qu'un modèle de classe. Certes, l'expert métier utilise un éditeur en langage naturel, mais derrière du code métier est généré.
Comme le montre la Figure 131, nous avons mis en place un mécanisme rendant possible le chargement du modèle métier depuis une classe Java (*BusinessModelExecutionLoaderFromJAVA*), depuis un XSD (*BusinessModelExecutionLoaderFromXSD*), depuis un précédent modèle métier (*BusinessModelExecutionLoaderFromEML*) et enfin depuis une archive java (BusinessModelExecutionLoaderFromJAR). Une fois que le modèle métier objet est chargé, il est possible de le sauvegarder en même temps que les règles. Le modèle métier objet est composé d'une liste de classes métier (*ClassListIterator*). Chaque classe métier *ClassBusinessModel* est constituée d'une liste de méthodes *MethodBusinessModel* et de variables métier *VariableBusinessModel.* Chaque élément du modèle métier possède une propriété qui contient une *businessNotation* et un booléen *isVisible* qui permet de définir si l'élément concerné peut être utilisé dans les règles ou non.

Nous allons maintenant voir comment nous avons procédé pour intégrer nos travaux dans la plateforme *e-Citiz.*

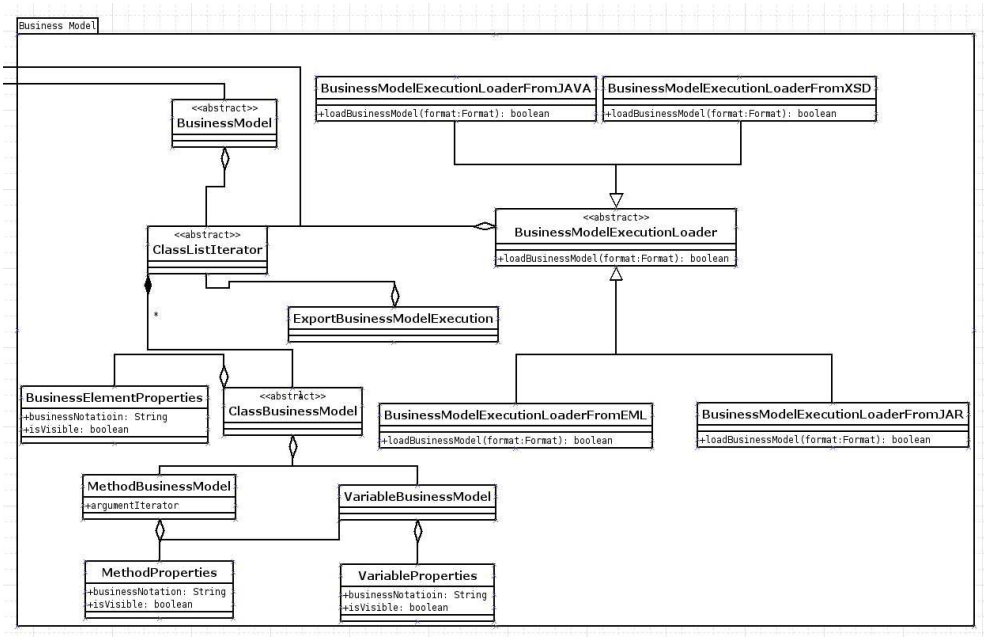

Fig. 131 – Package *Business Model*

8.4 Intégration dans *e-Citiz*

Arrivés à ce stade nous avions le module de stockage des règles, le module de transformation du ERML vers les formalismes de Drools et de Jrules et enfin le module du modèle objet. Comme nous l'avons dit, *e-Citiz* est subdivisé en 3 principaux composants, le studio, le générateur et le moteur (voir Figure 132).
L'intégration de nos travaux dans *e-Citiz* s'est faite en 3 temps :

1. intégrer le modèle de règles dans celui du studio d'abord, puis concevoir et réaliser un éditeur en langage naturel bien intégré dans l'ergonomie du Studio, qui permettrait d'éditer des règles. Il faudrait stocker les règles en utilisant le modèle de règles et enfin les charger en même temps que les autres éléments constituant l'e-service. Et en dernier, sérialiser/désérialiser une instance du modèle de règles dans le langage de règle ;

2. Au niveau du générateur, mettre en place le moteur de transformations basé sur XSLT ;

3. Au niveau du moteur, intégrer les moteurs de règles, proposer une API pour dialoguer avec ces derniers et enfin rendre visible le résultat de l'exécution des moteurs de règles.

Nous allons voir dans chaque étape le travail réalisé lors de l'intégration de nos travaux concernant le formalisme de règles métier.

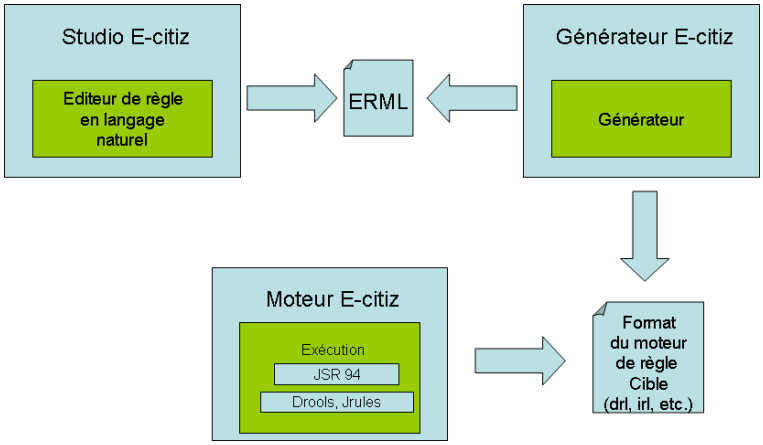

FIG. 132 – Architecture d'*e-Citiz* avec les règles métier

8.4.1 Intégration côté Studio

Le but recherché dans cette première partie de l'utilisation de l'approche par règles métier dans le studio, était de pouvoir faire la validation métier des étapes. Avant, cette validation se faisait uniquement par des classes Java, ce que des experts métier ne peuvent pas directement faire.
Le plus difficile au début de l'intégration dans le studio était d'avoir un point d'entrée dans le modèle car ce dernier est conséquent.
La première version de l'éditeur de règles métier n'était pas complètement intégrée au studio, ce qui, quelques temps après a fait l'occasion de sa refonte.
Les règles métier se base sur un modèle métier et nous avons exposé à la section précédente le package permettant de le mettre en place. Dans le cadre d'*e-Citiz*, le modèle métier qui est composé de 3 sous-modèles, est déjà en place et défini par l'expert métier. Ce qui restait à faire était de le mettre sous une forme exploitable par les règles métier. Le modèle *e-Citiz* qui nous a intéressé en premier était le modèle de représentation, appellé *BizGo*. Il représente les éléments constituant l'e-service tels que les étapes et les attributs les constituants. Donc à partir du *BizGo* nous générons le modèle métier de règles que l'on nomme *BusinessModel*.
La première version de l'éditeur n'était pas complètement intégrée dans la vue d'Eclipse et se détachait du Studio depuis un point de lancement comme le montre la Figure 133. Dans le cas de la validation métier, l'utilisateur peut choisir de ne pas utiliser la validation métier, comme le montre la Figure 134. Il peut aussi choisir d'utiliser une classe java comme le montre la Figure 135. Et enfin il peut choisir d'utiliser les règles métier, dans ce cas l'éditeur de règles s'affiche (voir Figure 136). Nous avons modifié le modèle du *BizGo* pour qu'une activité puisse avoir un attribut qui représente l'ID du ruleset qui lui est associée. La Figure 137 montre une vision globale de la première version de l'éditeur de règles.

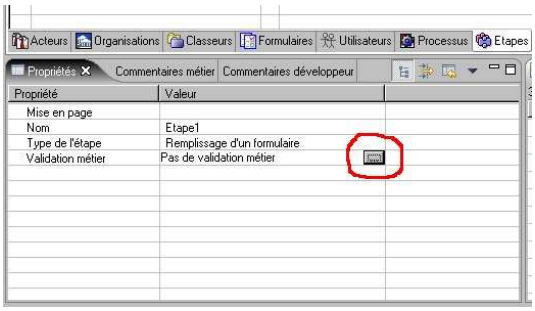

Fig. 133 – Point de lancement de l'éditeur dans le studio

Fig. 134 – Pas de règles pour la validation métier

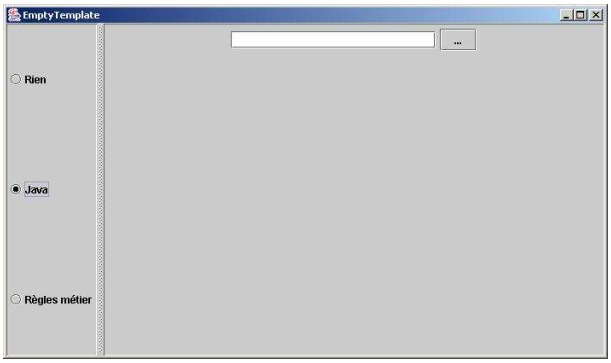

Fig. 135 – Utilisation de classe Java pour la validation métier

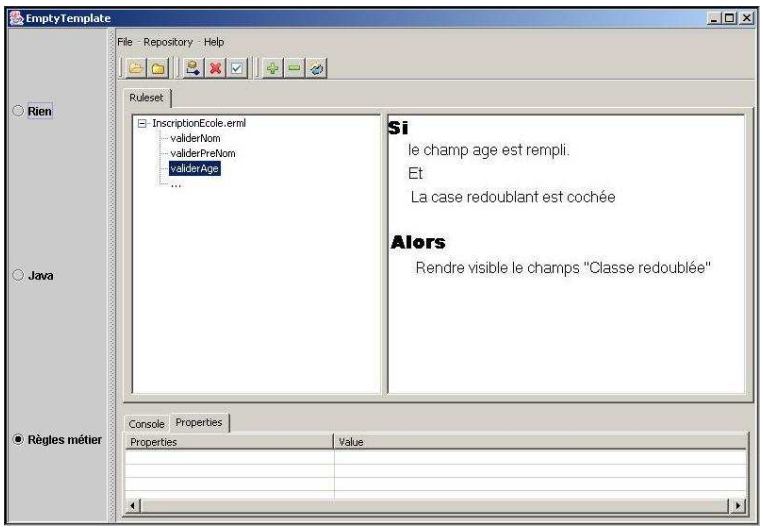

FIG. 136 – Règles métier pour la validation métier

Génération du modèle métier (*BusinessModel*)

Cette génération se fait par le chargeur *BizgoToRulesBusinessModel*. Le plus difficile n'est pas de générer le modèle mais plutôt de garder la cohérence entre les différents modèles. Pour cela nous avons mis en place un système de notification qui permet en cas de changement dans le modèle *BizGo* (par exemple un expert métier qui supprime un élément d'un étape), que le modèle métier (*BusinessModel*) en soit notifié comme le montre la Figure 138.
Nous avons hérité de deux interfaces dans le Studio : *Referencable*, qui permet à un élément de l'instance d'un modèle de pouvoir être "référencé", et *Referencer* qui permet à un élément de pouvoir "référencé" un autre élément qui va le notifier de tous ses événements de suppression et de mise à jour.
Lors du chargement (de la creation) du modèle métier depuis le modèle de représentation du studio, le *BizgoToRulesBusinessModel* référence chaque élément du modèle de représentation qui a une entrée dans le modèle métier. Ainsi dès qu'un événement de suppression ou de mise à jour intervient, le chargeur en sera notifié.
Nous avons également mis en place un autre système de notification entre le modèle métier et le modèle de règle. Par exemple, si on supprime un attribut dans le modèle de représentation depuis le studio, il faudrait que les règles l'utilisant en tiennent compte et signalent l'erreur afin d'invalider l'élément concerné en mémoire comme le montre la Figure 139.

Pour invalider une feuille, il faudrait que le modèle de règles puisse accepter une règle non valide, ce qui n'était pas le cas au départ. Nous avons introduit la notion d'invalidité qui peut être utilisée partout dans la règle. Ainsi, pour la recherche d'erreur avec l'analyseur nous faisons juste une recherche de ces objets invalides. Nous avons dû modifier la grammaire pour accepter cette notion d'entité invalide. Cependant nous ne permettons

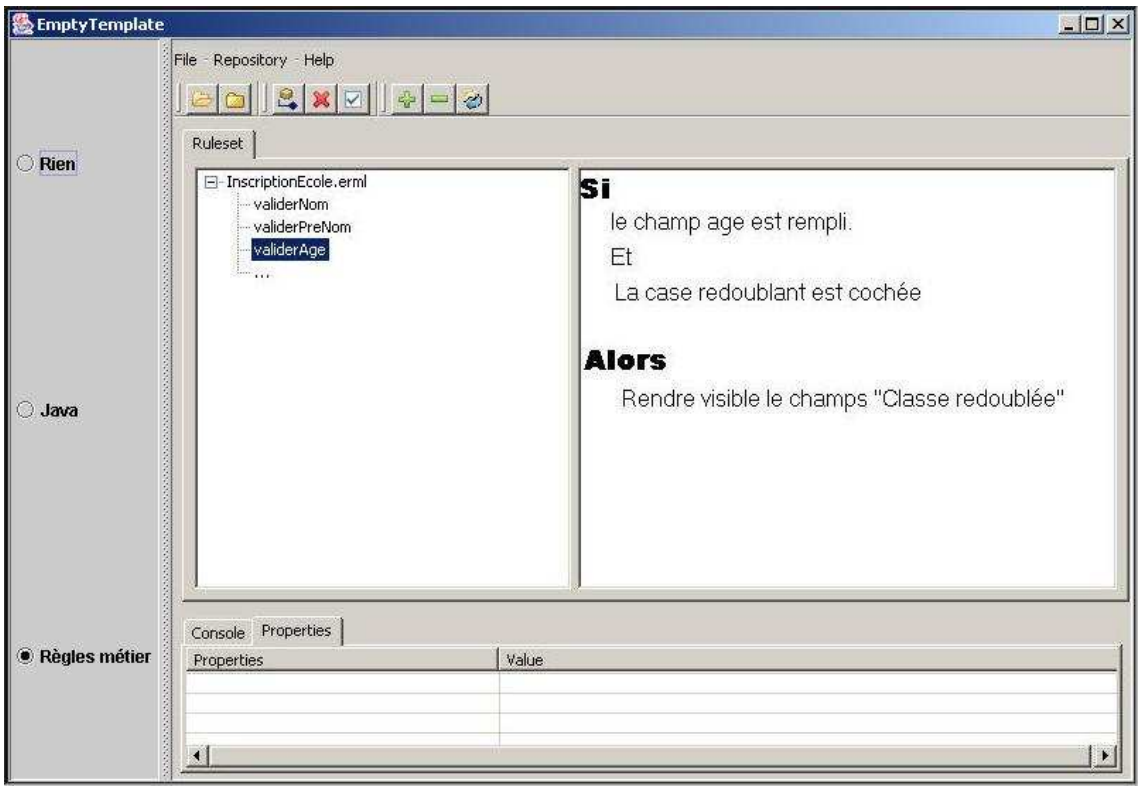

Fig. 137 – Vision globale de l'éditeur de règles métier

de faire des exports que si toutes les règles sont valides. Pour cela, dans le studio, nous utilisons un analyseur pour signaler les erreurs.

L'éditeur d'expressions

Une règle est constituée d'expressions. Lors de l'implémentation nous avions décidé, par soucis de ré-utilisation, d'intégrer un éditeur d'expression qui existait déjà dans le Studio *e-Citiz* dans le cadre de l'éditeur de requêtes. Cependant il fallait adapter cet éditeur à nos besoins. L'une des grosses différences entre les requêtes et les règles métier c'est que dans

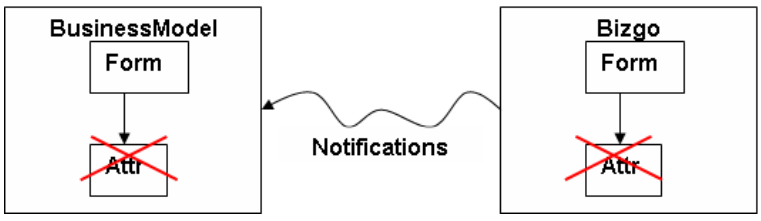

Fig. 138 – Système de notification entre le Bizgo et le BusinessModel

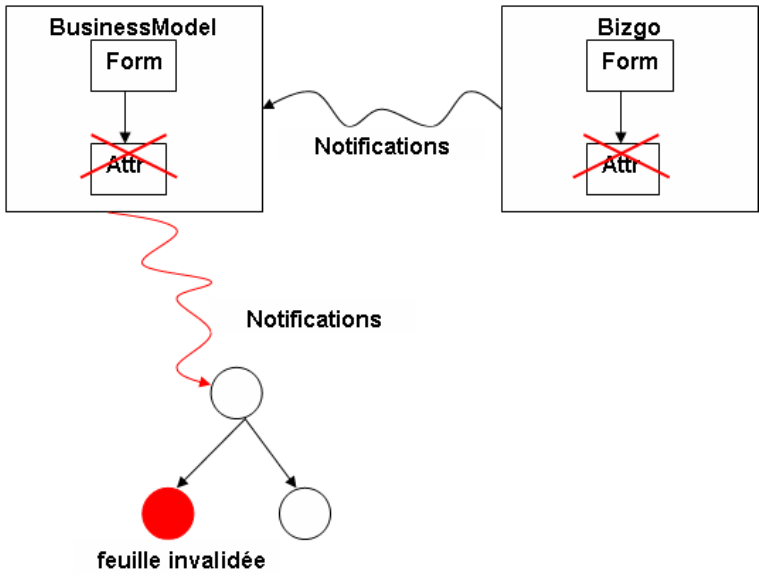

FIG. 139 – Système de notification entre le modèle de règles, *BusinessModel* et le *BizGo*

le premier, on a une seule expression alors que dans le second (cas des règles métier), on peut avoir plusieurs expressions car chaque prémisse ou action en est une.
Pour rendre l'éditeur plus agréable à l'expert métier, l'éditeur d'expression lui propose un assistant qui se contextualise en fonction de l'endroit ou l'on se trouve dans l'expression (voir Figure 140).

La plateforme *e-Citiz* suit l'approche de l'ingénierie dirigée par les modèles et dans ce sens au niveau du Studio il n'est pas encore question, dans le modèle de représentation, de classe, de méthodes java, etc, car tout se fait en référençant les éléments par leur ID. Cet ID ne sera transformé en terme de classes, méthodes et attributs qu'au moment de la génération. Alors que la première version de l'éditeur de règles, depuis le studio, générait les règles directement au format ERML qui, comme nous l'avons vu au à la section 3.10, utilise les notions de classes, méthodes et attributs. D'une certaine manière, l'éditeur de règles cassait l'éthique du studio, en plus de cela, son externalisation n'était pas simple à gérer et à intégrer dans l'ergonomie du reste du studio. Ainsi nous avons dû procéder à une refonte de la première version.

Le nouvel éditeur

Le fonctionnement du nouvel éditeur est le même, ce qui a changé c'est juste la manière de stocker les règles. Maintenant nous ne stockons plus les règles directement dans le format ERML. Nous avons mis en place une version allégée du langage ERML qui utilise des ID pour désigner des éléments et ce nouveau langage est directement intégré dans le modèle de représentation du studio à savoir le *Bizgo*. L'utilisation de ce nouveau langage allégé,

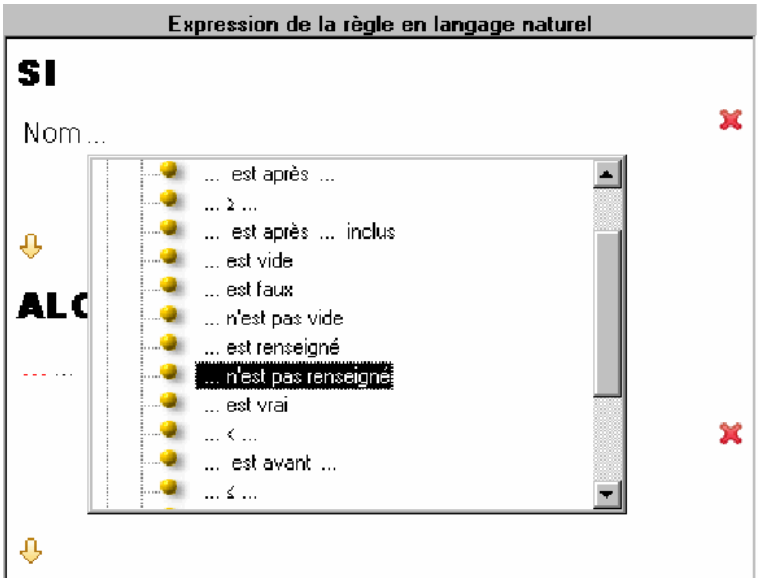

Fig. 140 – L'assistant de l'éditeur d'expressions

dont nous ne ferons pas l'étude dans ce document car très proche de ERML qui a été largement vu à la section 3.10, rajoute cependant un autre niveau de compilation. En effet il faut passer lors de la génération, que nous verrons à la section suivante, du ERML light vers ERML et terminer par une compilation du ERML vers le langage du moteur de règles cible.
La Figure 141 montre un aperçu de la nouvelle version de l'éditeur intégrée comme une vue dans le Studio *e-Citiz*.

Les 3 types d'utilisation des règles qui sont, pour l'instant, implémentés dans le studio ont le même éditeur pour écrire les conditions, ce qui change c'est la partie action. Nous avons étudié et proposé un éditeur personnalisé pour chaque type de règles afin que les experts métier puissent écrire les règles plus rapidement et de manière plus intuitive. Ainsi, pour les règles de validation, si une condition est vérifiée on ajoute des messages dans la liste des erreurs ou des avertissements comme le montre la Figure 142.

Pour les règles de navigation l'action consiste à choisir un point de sortie parmi tous ceux qui existent comme le montre la Figure 143. Ainsi, de manière dynamique, en fonction des conditions, un point de sortie (qui est représenté par un bouton sur la page du navigateur) sera dynamiquement choisi.

Pour les règles de visibilité et d'accessibilité appelées règles d'interaction, l'action consiste à choisir des éléments graphiques, une propriété (visible ou éditable) et une valeur booléenne.

L'éditeur de règles est également doté d'un analyseur d'erreurs qui parcourt les règles

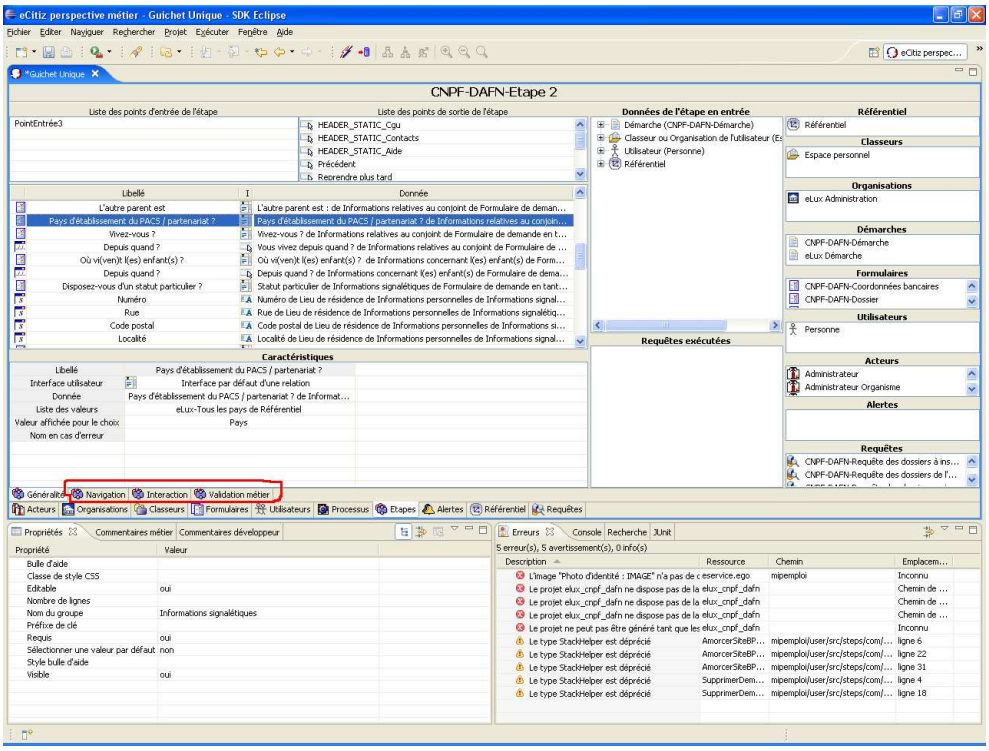

FIG. 141 – La nouvelle version de l'éditeur comme vue dans le Studio

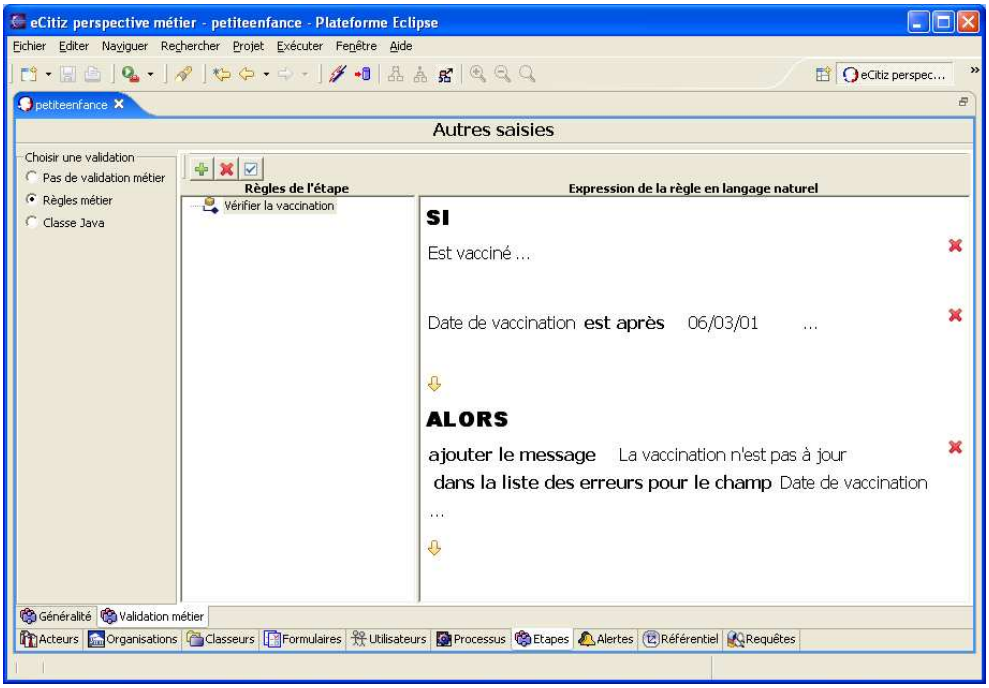

FIG. 142 – Règles de validation dans la nouvelle version de l'éditeur

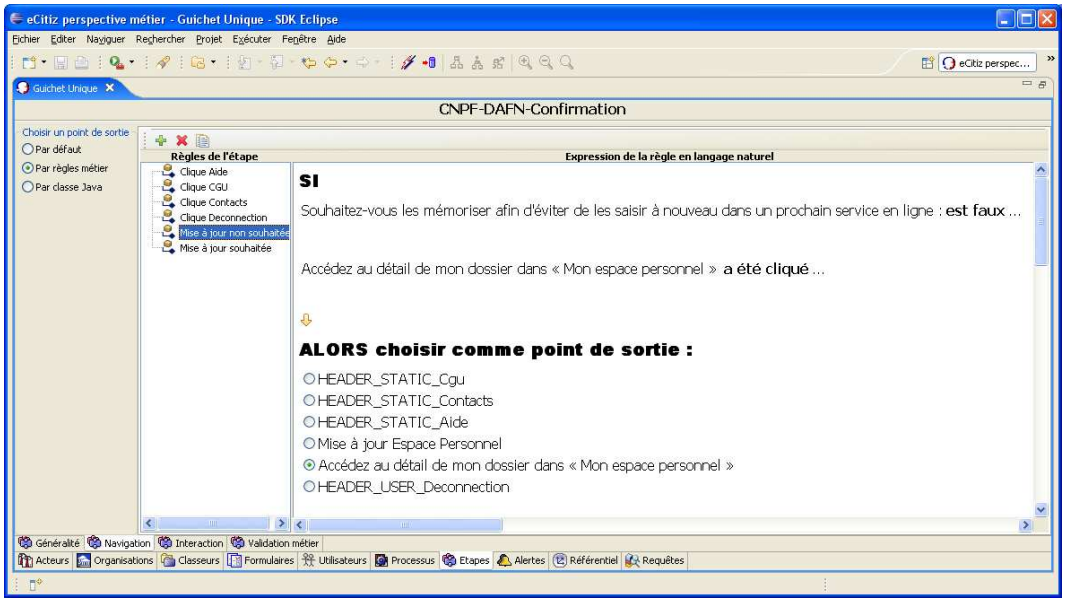

Fig. 143 – Règles de navigation dans la nouvelle version de l'éditeur

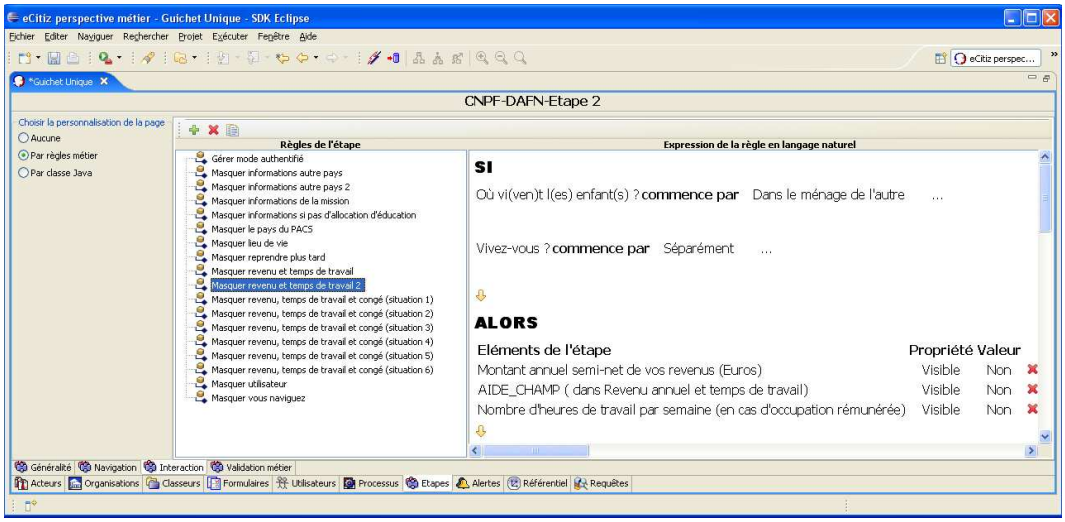

Fig. 144 – Règles d'interaction dans la nouvelle version de l'éditeur

pour vérifier si elles sont bien formées et les signale dans une vue de telle sorte qu'en double-cliquant sur l'erreur, la règle concernée s'affiche dans l'éditeur pour faciliter la correction.
Nous allons maintenant voir ce qui a été fait pour intégrer les règles au niveau du générateur.

8.4.2 Intégration côté Générateur

Une fois que la spécification de l'e-service est terminée dans le Studio *e-Citiz*, on procède à la phase de génération qui correspond au passage du PIM vers le PSM dans l'approche par MDA. Le générateur transforme les éléments de la spécification en instances de modèles Java : par exemple les étapes sont maintenant des classes et ses éléments sont des variables qui sont exploitables par des méthodes d'accès. Nous n'entrerons pas dans les détails de la génération de l'application entière dans ce document.
Dans le cas des règles métier, dans un premier temps nous faisons une génération du ERML light, intégré au *BizGo*, vers notre langage de règles ERML. Dans un second temps nous utilisons notre moteur de transformations pour générer les règles vers Drools ou JRules.

Génération de ERML light vers ERML

Comme nous l'avons déjà dit, la seule différence entre ERML light et ERML est que dans le premier, les éléments du modèle utilisé pour écrire les règles sont référencés par leur ID, alors que dans le second les éléments sont référencés par des notions de classes, de méthodes et d'attribut. Comme type de transformation de modèle, nous avons utilisé une transformation par programmation car plus simple dans ce cas vu que le modèle du ERML light est disponible sous forme d'objets Java.
A partir de cette génération nous obtenons des instances de ERML.
Nous allons maintenant voir la dernière phase de compilation, qui correspond à la génération des règles dans le format d'un moteur de règles cible.

Génération de ERML vers le format d'un moteur de règles cible

Pour cette génération finale, comme nous l'avons déjà dit à la section 3.10.2, nous utilisons une transformation de modèle par template en utilisant XSLT. En se basant sur le format du moteur cible, nous chargeons le bon XSLT, en lui donnant le fichier contenant les règles au format ERML.
L'exécution génére un fichier contenant les règles au format du moteur de règles (ici Drools ou Jrules).
Nous allons maintenant voir la dernière intégration concernant le moteur *e-Citiz*.

8.4.3 Intégration côté Moteur

Le moteur *e-Citiz* s'occupe de l'exécution des e-services. Dans le cas des règles métier, ce qu'il faudrait c'est la possibilité de lancer une instance d'un moteur de règles, créer une session d'exécution, ensuite injecter dans la mémoire de travail du moteur de règles les faits sur lesquels les règles travaillent, charger les règles dans l'agenda du moteur de règles. Et ensuite déclencher l'exécution des règles et recueillir le résultat de l'exécution afin d'en faire un traitement qui peut être un simple affichage dans le cas des règles de validation, et une modification de modèles dans le cas des règles de navigation et d'interaction.
Nous avons embarqué les moteurs de règles qui nous intéressent dans le moteur *e-Citiz*. Et concernant l'interaction avec ces derniers, nous utilisons La JSR 94, que nous présentons en détail au chapitre 10, qui permet d'interagir de manière homogène avec tous les moteurs de règles l'implémentant.
Ces mécanismes de la JSR94 sont utilisés dans le moteur *e-Citiz* pour faire une validation dynamique, choisir un point de sortie lorsque l'utilisateur aura fini de remplir une page ou encore faire apparaître/disparaître ou rendre accessible ou non des éléments graphiques de la page.

8.5 Perspectives et Conclusion

L'intégration au niveau du studio a été la plus laborieuse car ses modèles sont complexes et il n'était pas trivial de savoir ou se brancher pour avoir l'information nécessaire. Au départ nous avions un éditeur de règles qui n'était pas complètement intégré dans le studio. Nous utilisions aussi, au départ, ERML pour stocker directement les règles mises en place depuis l'éditeur. En procédant de la sorte nous cassions l'éthique du studio qui était de référencer tous les éléments des modèles d'un e-service par un ID et ce dernier n'était réellement remplacé par sa vraie valeur qu'au moment de la génération. Tout cela nous a conduit à proposer une version allégée de ERML et refondre l'éditeur externe de règles et de le faire comme étant une vue du studio. Cette dérivation du langage principal était au fond inévitable car au départ de sa conception nous l'avions fait de manière très générale pour qu'il puisse être utilisé dans d'autres circonstances.
Dans l'éditeur de règles, suivant le type de règles que l'on est entrain décrire, un assistant est proposé ainsi qu'une spécialisation de la partie action de sorte que les experts métier puissent de manière intuitive et rapide créer des règles. Dans le moteur nous utilisons la JSR94 pour avoir une certaine flexibilité au niveau des interactions avec le moteur de règles. Cette intégration peut être améliorée sur plusieurs points. Le premier étant au niveau de l'éditeur de règles, en effet actuellement, bien que le langage de règles soit expressif, l'éditeur est assez limité. Actuellement il n'est pas possible depuis l'éditeur de faire des disjonctions ou des imbrications de disjonctions et de conjonctions. Pour faire une disjonction on utilise la disjonction car on a $(A \vee B \longrightarrow C) = (A \longrightarrow C \wedge B \longrightarrow C)$. Le modèle de règles ERML supporte très bien ce genre de construction (l'utilisation de la disjonction) mais

c'est uniquement au niveau graphique que nous n'avons pas eu le temps de le rendre utilisable.
Une amélioration intéressante au niveau du moteur de règles est d'externaliser les fichiers de règles du moteur *e-Citiz* ou de pouvoir, à l'exécution, y accéder. A l'heure actuelle ils sont embarqués dans l'archive de l'application Web. Le souci avec ce dernier est que si pour des raisons quelconques les experts métier modifient les règles, il faudra alors refaire l'archive et le re-déployer. L'un des principaux avantages de l'approche par règles métier est de pouvoir, à souhait, modifier les règles qui sont automatiquement prises en comptes dans la nouvelle politique et ceci à chaud par le système d'information.

Chapitre 9

Prototypage de notre approche sur la génération de règles métier par enrichissement sémantique de modèle

Sommaire

9.1 Introduction

Au chapitre 7 nous avons exposé notre approche concernant la génération de règles métier par enrichissement sémantique de modèles en utilisant des technologies du Web Sémantique. Nous y avions également présenté ODM qui est un standard développé par l'OMG qui permet de transformer un modèle UML en modèle OWL et inversement. Notre objectif n'est pas d'inventer une connaissance mais de rendre une connaissance implicite explicite, pour une pleine utilisation en raisonnement par des machines. Ici nous allons voir comment nous sommes entrain de procéder actuellement pour mettre en œuvre notre approche.
Dans un premier temps nous allons présenter l'implémentation (l'unique que nous connaissons pour l'instant) de ODM que nous utilisons qui s'intitule EODM pour Eclipse Ontology Definition Metamodel, qui comme son nom l'indique, est basée sur la plateforme Eclipse. Nous présenterons par la suite la méthodologie que nous avons adoptée, ainsi que l'API que nous avons mis en place pour le chargement des modèles et l'extension des capacités d'inférence du moteur de raisonnement.

9.2 Implémentation du standard ODM

IBM qui est l'un des auteurs de la soumission acceptée pour ODM, propose une implémentation de la spécification ODM. Cette implémentation s'appelle IODT pour Integrated Ontology Development Toolkit. IODT est un paquetage pour faire un développement orienté ontologie qui est composé d'un méta-modèle EMF basé sur ODM appelé EODM, d'un système de gestion d'une base d'ontologies appelé Minerva et d'autres extensions.
EODM est aussi un projet open source qui fait parti du paquetage EMFT d'eclipse [149]. EODM comporte un parser et un serializer RDF/OWL, un reasoner et des translateurs depuis et vers d'autres langages de modélisation (EMF). EODM qui est intégré à la plateforme Eclipse permet la construction d'ontologies, leur gestion et leur visualisation.
La Figure 145 montre l'architecture de EODM. Comparé à d'autres outils comme

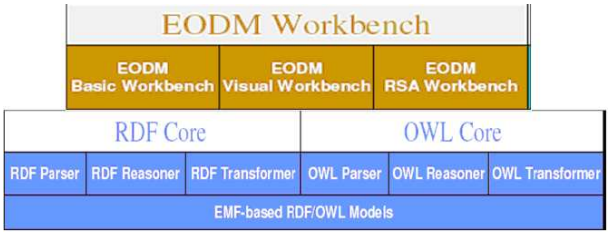

FIG. 145 – Architecture de EODM

JENA [84], la nouveauté apportée par EODM est sa capacité à faire une transformation bi-directionnelle entre un modèle RDF/OWL et un modèle ECore. L'interopérabilité entre

un modèle RDF/OWL et un modèle EMF permet aux développeurs d'applications Web Sémantique de développer des ontologies en utilisant les outils de modélisation compatibles MOF.
Dans son implémentation actuelle, EMF n'offre pas de définition formelle, d'inférence et de modélisation de la sémantique (connaissance). Le but de EODM est d'ajouter ces fonctionnalités à EMF.
Le principe de EODM est le suivant : en utilisant EMF et ODM, un modèle en mémoire, c'est à dire un ensemble de classes, est généré. Ce modèle en mémoire est enrichie de classes et méthodes helper qui permettent de créer, éditer et parcourir tout modèle OWL. EODM comporte également un parser OWL, qui permet de charger des fichiers OWL en EMF et inversement, de générer des fichiers OWL depuis un modèle EMF. La transformation bidirectionnelle entre un modèle OWL et un modèle EMF permet d'utiliser ce dernier comme modèle pivot vers les autres modèles (Java, UML, XML, XSD). De la sorte, les experts techniques peuvent développer leurs modèles en utilisant leurs outils de modélisation favoris, l'exporter en ECore (MOF), transformer ce dernier en ontologie OWL, l'enrichir de sémantique afin de faire de l'inférence. La Figure 146 montre les opérations que permet EODM.

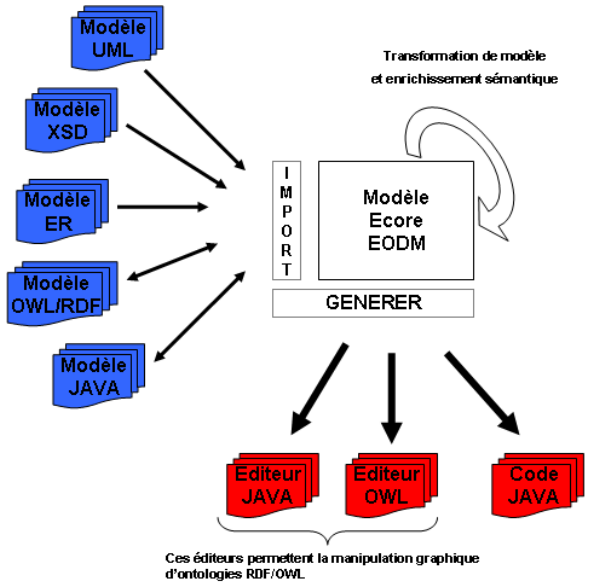

Fig. 146 – Architecture des systèmes d'ontologies basés sur EMF

Pour résumer, dans le processus de transformation de modèles, en utilisant les systèmes d'ingénierie d'ontologies basés sur EMF comme EODM, on commence par capturer la sémantique formelle et informelle des modèles. Le moteur de transformation de modèles transforme le modèle de la sémantique formelle en modèles OWL en utilisant le mapping pré-défini entre OWL et le modèle en entrée (ECore), on obtient alors une ontologie OWL. La sémantique informelle est capturée comme axiomes additionnels et ajoutée au modèle OWL en utilisant un éditeur OWL. Ce processus facilite la capture automatique de la

connaissance (sémantique) et augmente la productivité des experts fonctionnels. Tout ce processus de capture et representation de la sémantique formelle et informelle d'un modèle en modèle OWL est appelé *semantics enrichment* [124].

9.3 Prototype

L'approche que nous avons présentée au chapitre 7 consiste à faire de l'enrichissement sémantique sur des modèles UML afin de pouvoir générer des règles métier simples pour aider à la phase de recensement des règles métier par les experts fonctionnel. En effet, depuis quelques temps maintenant, il est question de rendre les modèles UML productifs par l'approche de l'ingénierie dirigée par les modèles. Imaginons un peu les capacités qu'offriraient ces modèles UML sémantiquement riches. Nous avons également vu qu'UML n'offrait pas de solutions appropriées pour l'ajout et le traitement sémantique, ce que le Web Sémantique offre. Dans notre approche nous proposons d'utiliser Ontology Definition Metamodel (ODM) de l'OMG [123] pour enrichir sémantiquement les modèles UML et d'utiliser un moteur de raisonnement pour générer des règles métier. Nous allons dans les sous-sections suivantes présenter un prototype que nous sommes entrain de mettre en place pour valider et éprouver notre approche.

9.3.1 Outillage technique

Pour réaliser ce prototype nous nous basons sur la plateforme Eclipse en utilisant le langage Java. Nous utilisons l'implémentation EODM de ODM que nous avons présentée à la section précédente. Le moteur de raisonnement que nous utilisons est également embarqué dans EODM. Nous avons choisi le moteur de raisonnement de EODM car, dans notre cas plus simple à étendre car entièrement écrit en Java. Cependant nous envisageons d'utiliser le moteur Racer [112] par l'intermédiaire du protocol HTTP afin de pouvoir faire des requêtes depuis la plateforme Eclipse. Comme éditeur d'ontologies nous utilisons aussi bien un format textuel, par le biais de l'api de EODM, qu'un format graphique avec l'éditeur Protégé [144]. En ce qui concerne le format de sortie des règles générées, nous sommes toujours à la recherche d'une implémentation de SBVR.

9.3.2 Méthodologie

Première étape : génération du modèle OWL par EODM

Nous allons maintenant décrire notre procédé. En premier lieu nous utilisons l'API de EODM pour charger notre modèle ECore et le transformer en modèle OWL. La Figure 147 représente le modèle ECore que nous allons utiliser pour illustrer notre approche. La

Figure 148 montre le modèle sous un point de vue graphique.

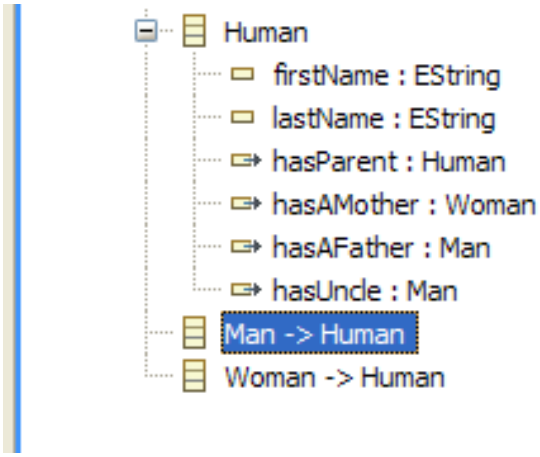

Fig. 147 – Modèle ECore simple

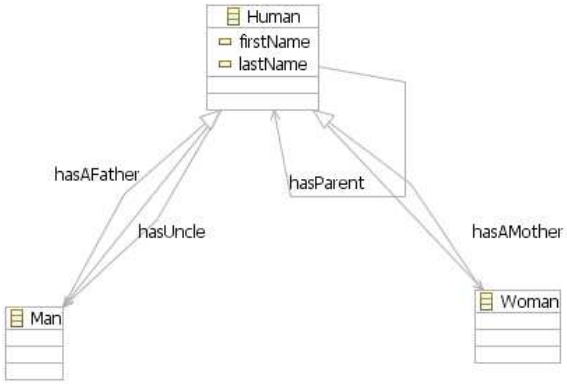

Fig. 148 – Vue graphique du modèle ECore simple

Le modèle de test est simple : nous avons les classes “Homme” et “Femme” qui étendent la classe “Humain” qui a un nom, un prénom, une propriété “hasParent” de type humain, une propriété “hasMother” de type femme, une propriété “hasFather” de type homme et une propriété “hasUncle” de type homme.
L'extrait de code suivant montre comment nous procédons, en utilisant EODM, pour générer du OWL depuis ECore. A la ligne 6 nous initialisons un transformateur que nous utilisons à la ligne 9 pour convertir le modèle ECore de la Figure 147.

Extrait de code pour la génération du modèle OWL depuis un modèle ECore

```
1  ...
2
3  try
```

```
{
        // Creation d' un transformer ecore vers OWL
        ECore2OWLTransformer ecoreToOWL = new ECore2OWLTransformer();
        // utilisation du transformer cree pour convertir le model ecore
        // en modele OWL
        ecoreToOWL.convertECore2OWL("./humanFromEODM.ecore",
                                    "./humanFromEODML_Generated.owl");
}
catch (OWLTransformationException e)
{
        e.printStackTrace();
}

...
```

Le modèle OWL généré n'est à ce stade qu'une simple taxonomie, c'est à dire un ensemble de relations classe/sous-classe.

Extrait du modèle OWL généré depuis le modèle ECore

```
<rdf:RDF xmlns:human="&human;" xmlns:xsd="&xsd;" xmlns:rdf="&rdf;" xmlns:rdfs="&rdfs;" xmlns:owl="&owl;">
    <owl:Ontology rdf:about="">
        </owl:Ontology>
        <owl:Class rdf:about="&human;Human">
        </owl:Class>
        <owl:Class rdf:about="&human;Man">
                <owl:disjointWith rdf:resource="&human;Woman"/>
                <rdfs:subClassOf rdf:resource="&human;Human"/>
        </owl:Class>
        <owl:Class rdf:about="&human;Woman">
                <rdfs:subClassOf rdf:resource="&human;Human"/>
        </owl:Class>
        <owl:DatatypeProperty rdf:about="&human;firstName">
                <rdfs:domain rdf:resource="&human;Human"/>
                <rdfs:range rdf:resource="&xsd;string"/>
        </owl:DatatypeProperty>
        <owl:DatatypeProperty rdf:about="&human;lastName">
                <rdfs:domain rdf:resource="&human;Human"/>
                <rdfs:range rdf:resource="&xsd;string"/>
        </owl:DatatypeProperty>
        <owl:ObjectProperty rdf:about="&human;hasParent">
                <rdfs:domain rdf:resource="&human;Human"/>
                <rdfs:range rdf:resource="&human;Human"/>
        </owl:ObjectProperty>
        <owl:ObjectProperty rdf:about="&human;hasAMother">
                <rdfs:domain rdf:resource="&human;Human"/>
                <rdfs:range rdf:resource="&human;Woman"/>
        </owl:ObjectProperty>
        <owl:ObjectProperty rdf:about="&human;hasAFather">
                <rdfs:domain rdf:resource="&human;Human"/>
                <rdfs:range rdf:resource="&human;Man"/>
        </owl:ObjectProperty>
        <owl:ObjectProperty rdf:about="&human;hasUncle">
                <rdfs:domain rdf:resource="&human;Human"/>
                <rdfs:range rdf:resource="&human;Man"/>
        </owl:ObjectProperty>
</rdf:RDF>
```

Deuxième étape : enrichissement sémantique

La seconde étape consiste à enrichir la taxonomie générée. Pour cela, nous utilisons l'éditeur d'ontologies Protogé, bien que rien n'empêche de le faire manuellement. Comme le montre l'extrait de code suivant, nous avons enrichi le modèle OWL en plusieurs points. A la ligne 22, pour la propriété "hasParent" nous avons rajouté une caractéristique de transitivité. Aux lignes 27 et 33, pour les propriétés "hasMother" et "hasFather" nous avons rajouté une caractéristique de fonctionnalité. Nous avons également rajouté des instances du modèle OWL que nous aurions bien pu mettre dans un fichier différent. Par exemple à la ligne 69 nous avons la déclaration de l'instance de "human" nommée *Patrick* qui a comme mère *Agnes* et comme père *David* et *Daouda*.

Extrait du modèle OWL enrichi

```
1  <rdf:RDF xmlns:human="&human;" xmlns:xsd="&xsd;" xmlns:rdf="&rdf;" xmlns:rdfs="&rdfs;" xmlns:owl="&owl;">
2      <owl:Ontology rdf:about="">
3          </owl:Ontology>
4          <owl:Class rdf:about="&human;Human">
5          </owl:Class>
6          <owl:Class rdf:about="&human;Man">
7                  <owl:disjointWith rdf:resource="&human;Woman"/>
8                  <rdfs:subClassOf rdf:resource="&human;Human"/>
9          </owl:Class>
10         <owl:Class rdf:about="&human;Woman">
11                 <rdfs:subClassOf rdf:resource="&human;Human"/>
12         </owl:Class>
13         <owl:DatatypeProperty rdf:about="&human;firstName">
14                 <rdfs:domain rdf:resource="&human;Human"/>
15                 <rdfs:range rdf:resource="&xsd;string"/>
16         </owl:DatatypeProperty>
17         <owl:DatatypeProperty rdf:about="&human;lastName">
18                 <rdfs:domain rdf:resource="&human;Human"/>
19                 <rdfs:range rdf:resource="&xsd;string"/>
20         </owl:DatatypeProperty>
21         <owl:ObjectProperty rdf:about="&human;hasParent">
22                 <rdf:type rdf:resource= "http://www.w3.org/2002/07/owl#TransitiveProperty"/>
23                 <rdfs:domain rdf:resource="&human;Human"/>
24                 <rdfs:range rdf:resource="&human;Human"/>
25         </owl:ObjectProperty>
26         <owl:ObjectProperty rdf:about="&human;hasAMother">
27                 <rdf:type rdf:resource= "http://www.w3.org/2002/07/owl#FunctionalProperty"/>
28                 <rdfs:subPropertyOf rdf:resource="&human;hasParent"/>
29                 <rdfs:domain rdf:resource="&human;Human"/>
30                 <rdfs:range rdf:resource="&human;Woman"/>
31         </owl:ObjectProperty>
32         <owl:ObjectProperty rdf:about="&human;hasAFather">
33                 <rdf:type rdf:resource= "http://www.w3.org/2002/07/owl#FunctionalProperty"/>
34                 <rdfs:subPropertyOf rdf:resource="&human;hasParent"/>
35                 <rdfs:domain rdf:resource="&human;Human"/>
36                 <rdfs:range rdf:resource="&human;Man"/>
37         </owl:ObjectProperty>
38         <owl:ObjectProperty rdf:about="&human;hasUncle">
39                 <rdfs:domain rdf:resource="&human;Human"/>
40                 <rdfs:range rdf:resource="&human;Man"/>
41         </owl:ObjectProperty>
42         <human:Man rdf:about="&human;Adam">
43                 <human:lastName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Adam</human:lastName>
44                 <human:firstName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Adam</human:firstName>
45         </human:Man>
```

```
46          <human:Woman rdf:about="&human;Eve">
47                  <human:hasAFather rdf:resource="&human;Adam"/>
48                  <human:lastName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Eve</human:lastName>
49                  <human:firstName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Eve</human:firstName>
50          </human:Woman>
51          <human:Man rdf:about="&human;David">
52                  <human:hasAMother rdf:resource="&human;Eve"/>
53                  <human:hasAFather rdf:resource="&human;Adam"/>
54                  <human:lastName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Hamer</human:lastName>
55                  <human:firstName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">David</human:firstName>
56          </human:Man>
57          <human:Man rdf:about="&human;Daouda">
58                  <human:hasAMother rdf:resource="&human;Eve"/>
59                  <human:hasAFather rdf:resource="&human;Adam"/>
60                  <human:lastName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Hamer</human:lastName>
61                  <human:firstName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Daouda</human:firstName>
62          </human:Man>
63          <human:Woman rdf:about="&human;Agnes">
64                  <human:hasAMother rdf:resource="&human;Eve"/>
65                  <human:hasAFather rdf:resource="&human;Adam"/>
66                  <human:lastName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Palmer</human:lastName>
67                  <human:firstName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Agnes</human:firstName>
68          </human:Woman>
69          <human:Man rdf:about="&human;Patrick">
70                  <human:hasAMother rdf:resource="&human;Agnes"/>
71                  <human:hasAFather rdf:resource="&human;David"/>
72                  <human:hasAFather rdf:resource="&human;Daouda"/>
73                  <human:lastName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Hamer</human:lastName>
74                  <human:firstName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Patrick</human:firstName>
75          </human:Man>
76  </rdf:RDF>
```

Troisième étape : utilisation d'un moteur de raisonnement

Maintenant que nous avons un modèle sémantiquement riche, nous utilisons un moteur de raisonnement basé sur la logique de description. Pour des soucis de simplicité, ici nous utilisons un moteur de raisonnement intégré dans EODM que nous étendons comme le montre l'extrait de code suivant. Actuellement ce moteur de raisonnement ne peut faire que des raisonnements d'ordre structurel c'est à dire utilisant les classes et les propriétés, alors que dans notre approche nous proposons aussi de nous baser sur les instances ou individuels des classes. Pour atteindre notre objectif nous avons dû étendre les capacités du moteur et aussi afin de pouvoir formatter la sortie suivant un format proche du SBVR. A la ligne 6 nous déclarons le moteur de raisonnement et à la ligne 8 nous l'initialisons avec notre ontologie que nous chargeons depuis un fichier. A partir de la ligne 15 nous avons une méthode qui permet de vérifier si pour une propriété, un domaine et une portée sont définis, afin de pouvoir les rendre explicite dans la règle générée. A la ligne 52 nous avons une méthode qui permet de traiter pour une instance, toutes ses propriétés et de voir si ces propriétés sont fonctionnelles et si c'est le cas, vérifier s'il existe deux valeurs différentes pour cette même propriété et dans ce cas, ces dernières sont égales.

Extrait du code pour la génération de règles métier

```
1   ...
2
```

```
3   public void testStructuralReasoner()
4   {
5
6           OWLTaxonomyReasoner reasoner = StructuralReasonerFactory.instance()
7                           .createOWLTaxonmyReasoner();
8           reasoner.initialize(ontology);
9           runTest(reasoner);
10
11  }
12
13  ...
14
15  private void generateRuleFromPropertyDomainAndRange(Property property)
16  {
17          EList domain = property.getRDFSDomain();
18          EList range = property.getRDFSRange();
19          if ((domain == null || domain.isEmpty()) && (range == null || range.isEmpty()))
20          {
21                  return;
22          }
23          else
24          {
25                  String ruleSBVR = "IF\n Object1 " + property.getLocalName()
26                                  + " Object2\nThen\n";
27                  if (domain != null)
28                  {
29                          Iterator domainIterator = domain.iterator();
30                          while (domainIterator.hasNext())
31                          {
32                                  RDFSClass oneDomain = (RDFSClass) domainIterator.next();
33                                  ruleSBVR += "   Object1 must be a(n) "
34                                                  + oneDomain.getLocalName() + " type\n";
35                          }
36                  }
37                  if (range != null)
38                  {
39                          Iterator rangeIterator = range.iterator();
40                          while (rangeIterator.hasNext())
41                          {
42                                  RDFSClass oneRange = (RDFSClass) rangeIterator.next();
43                                  ruleSBVR += "   And\n   Object2 must be a(n) "
44                                                  + oneRange.getLocalName() + " type\n";
45                          }
46                  }
47                  System.out.println(ruleSBVR);
48          }
49  }
50  ...
51
52  private void treatIndividual(OWLTaxonomyReasoner anReasoner, Individual individual, List allProperties)
53  {
54          Iterator iterator = allProperties.iterator();
55          while (iterator.hasNext())
56          {
57                  Property aProperty = (Property) iterator.next();
58                  List propertyValues = getPropertyValues(aProperty, individual);
59                  if (propertyValues == null)
60                  {
61                          break;
62                  }
63                  Iterator iterator2 = propertyValues.iterator();
64                  String all = "";
65                  Boolean functional = aProperty.getFunctional();
66                  if (functional != null && functional.booleanValue())
67                  {
```

```
68                          while (iterator2.hasNext())
69                          {
70                                  Object object = iterator2.next();
71                                  if (object instanceof ObjectSlot)
72                                  {
73                                          EObjectResolvingEList list = (EObjectResolvingEList) ((ObjectSlot) object)
74                                                  .getContent();
75                                          Individual ind = (Individual) list.get(0);
76                                          String localName = ind.getLocalName();
77                                          all += localName + " ";
78                                          treatIndividualRangeForAProperty(individual, aProperty, localName);
79                                  }
80                                  else if (object instanceof DatatypeSlot)
81                                  {
82                                          all += ((DatatypeSlot) object).toString() + " ";
83                                  }
84                          }
85                          if (propertyValues.size() > 1)
86                          {
87                                  all += " are the same thing";
88                                  System.out.println(all);
89                          }
90                  }
91          }
92  }
93
94  ...
```

Le listing ci-dessous montre le résultat de l'exécution du moteur de raisonnement. Nous pouvons voir que de la ligne 2 à la ligne 54 les règles sont générées en se basant aussi bien sur la syntaxe que sur la sémantique de l'ontologie. A partir de la ligne 56 nous avons le résultat de l'application des règles sur les instances que nous avons créées. Par exemple à la ligne 72 nous avons l'affirmation que "David" et "Daouda" sont les mêmes choses car dans la déclaration des instances dans l'ontologie enrichie que nous avons présentée ci-dessus, nous avions les *Abox* disant que "Patrick" a pour père "David" et que également qu'il a pour père "Daouda". Par ailleurs nous avions une *Tbox* qui définissait la propriété "hasFather" comme étant fonctionnelle, de ce fait, en se basant sur ces *Abox* et *Tbox* nous avons pu dire de manière explicite que "David" et "Daouda" sont les mêmes choses.

Résultat de l'exécution du moteur de raisonnement

```
1   Test ...
2   IF
3           Object1 firstName Object2
4   Then
5           Object1 must be a(n) Human type
6           And
7           Object2 must be a(n) string type
8
9   IF
10          Object1 lastName Object2
11  Then
12          Object1 must be a(n) Human type
13          And
14          Object2 must be a(n) string type
15
16  IF
17          Object1 hasParent Object2
18  Then
19          Object1 must be a(n) Human type
```

```
20          And
21          Object2 must be a(n) Human type
22
23  IF
24          hasAMother is a sub property of hasParent
25  Then
26          The domain of hasAMother must be include is the domain of hasParent
27          And
28           The range of hasAMother must be include is the range of hasParent
29  IF
30          hasAFather is a sub property of hasParent
31  Then
32          The domain of hasAFather must be include is the domain of hasParent
33          And
34           The range of hasAFather must be include is the range of hasParent
35  IF
36          Object1 hasAMother Object2
37  Then
38          Object1 must be a(n) Human type
39          And
40          Object2 must be a(n) Woman type
41
42  IF
43          Object1 hasAFather Object2
44  Then
45          Object1 must be a(n) Human type
46          And
47          Object2 must be a(n) Man type
48
49  IF
50          Object1 hasUncle Object2
51  Then
52          Object1 must be a(n) Human type
53          And
54          Object2 must be a(n) Man type
55
56  The owl individual: http://diouftest.com/human#Adam subclass of Man
57  The owl individual: http://diouftest.com/human#Eve subclass of Woman
58  Eve hasAFather Adam ---> Adam must be a Man
59  The owl individual: http://diouftest.com/human#David subclass of Man
60  David hasAMother Eve ---> Eve must be a Woman
61  David hasAFather Adam ---> Adam must be a Man
62  The owl individual: http://diouftest.com/human#Daouda subclass of Man
63  Daouda hasAMother Eve ---> Eve must be a Woman
64  Daouda hasAFather Adam ---> Adam must be a Man
65  The owl individual: http://diouftest.com/human#Agnes subclass of Woman
66  Agnes hasAMother Eve ---> Eve must be a Woman
67  Agnes hasAFather Adam ---> Adam must be a Man
68  The owl individual: http://diouftest.com/human#Patrick subclass of Man
69  Patrick hasAMother Agnes ---> Agnes must be a Woman
70  Patrick hasAFather David ---> David must be a Man
71  Patrick hasAFather Daouda ---> Daouda must be a Man
72  David Daouda  are the same thing
```

9.4 Conclusion

Le prototype n'en est qu'à ses débuts car, des techniques de raisonnement que nous avons listées au chapitre 7, nous n'en utilisons que les raisonnements se basant sur le domaine et la portée. Les règles qui sont générées pour l'instant sont dans un format

naturel proche du SBVR mais qui n'est pas du SBVR strict. La raison de cela est que pour l'instant, (Septembre 2007), SBVR est toujours en cours de finalisation et que pour l'instant nous n'avons pas vu d'implémentation utilisable, mais cela se saurait tarder.
Nous n'avons aucunement la prétention de vouloir générer tout type de règles métier, ce qui est utopiste car en cas de métier complexe les experts métier devront écrire eux-mêmes les règles complexes. Notre objectif est de leur faire gagner du temps en générant les règles les plus simples. Pour l'instant les règles que nous générons sont simples et sans effets de bord, et il faudrait voir comment faire pour générer des règles avec effets de bord. Pour cela il faudra voir comment gérer les effets de bord avec OWL ou reconsidérer l'utilisation de OCL dans le futur car des travaux sont en cours pour l'entendre afin de pouvoir modéliser des effets de bord avec.
Avec notre approche pour la génération de règles métier, il faudrait faire de l'enrichissement sémantique en utilisant OWL. Maintenant ce qu'il faudrait étudier, c'est de voir entre écrire du OWL et écrire les règles métier que nous générons, c'est quoi le plus simple. Avec des outils d'édition d'ontologies comme Protégé, la première solution peut être très simple.

Chapitre 10

La JSR 94

Sommaire

Un moteur de règles évalue et exécute des règles qui sont des paires de *if-else*. La force des règles métier réside dans le fait qu'elles permettent de séparer la connaissance et sa logique d'implémentation et aussi de permettre de changer cette dernière sans changer le code source. Au chapitre 2 sur l'approche par règles métier nous avons longuement détaillé les règles métier. Nous avons également vu au chapitre 3 sur le formalisme de règles qu'il n'existait pas encore un formalisme standard permettant d'échanger des règles. Cependant il existe une bibliothèque qui permet de manipuler les moteurs de règles qui l'implémentent de manière homogène dans une plateforme Java.
La JSR 94 [150] est une spécification qui fournit un API Java permettant d'accéder depuis un contexte Java à des moteurs de règles de manière homogène.

10.1 Introduction

La plupart des moteurs de règles ont des API propriétaires les rendant difficiles à intégrer dans une application. Si on décide pour une raison quelconque de changer de moteur de règles, c'est presque la totalité de l'application qu'il faudra réécrire. La JSR 94 tente de standardiser l'implémentation des moteurs de règles pour les technologies Java. Drools, Fair Isaac Blaze Advisor, ILOG Jrules et Jess supportent la JSR 94.
La JSR 94 fournit des indications sur l'administration des règles et leur exécution mais ne dit rien, absolument rien sur la manière de les représenter.

10.2 Les moteurs de règles

Le but principal d'un moteur de règles est d'externaliser le métier d'une application. On peut le voir comme un interpréteur sophistiqué de *if-then*, ces derniers étant les règles. Les données en entrée pour un moteur de règles sont :

- Un ensemble de règles appelé ruleset.
- Des data objects.

Les données en sortie dépendent des données en entrée et cela peut être, soit des modifications d'objets, soit la création d'objets soit des effets de bord.

10.2.1 La JSR 94

La JSR 94 fournit un API simple pour accéder à un moteur de règles depuis un environnement Java SE ou Java EE. Elle permet de :

- Enregistrer et désenregistrer des règles.
- Analyser (parser) une règle.
- Inspecter les méta-données d'une règle.
- Exécuter une règle.
- Retrouver le résultat.

- Filtrer le résultat.

Il faut savoir que la JSR 94 ne standardise pas du tout :

- Le moteur de règles lui-même.
- Le flot d'exécution des règles.
- Le langage utilisé pour décrire les règles.
- Le mécanisme de déploiement.

10.3 L'architecture de la JSR 94

Les API sont définis en deux grands ensembles :

- **L'API d'administration de règles** : c'est le package *javax.rules.admin*, il fournit des classes pour le chargement des règles et associe des actions comme *l'execution set*. Une *execution set* est une collection de règles. Les règles peuvent être chargées depuis des sources externes comme des URI, un InputStream, un XML Element, un arbre abstraite binaire ou un Reader. Il fournit également des méthodes pour inscrire et désinscrire une *execution set* et permet aussi de définir des autorisations d'accès sur ces derniers.
- **L'API client runtime :** c'est le package *javax.rules* qui fournit les outils nécessaires au client pour exécuter les règles et obtenir les résultats. Seules les règles qui ont été enregistrées en utilisant l'API d'administration sont accessibles.

10.3.1 L'API d'administration de règles

Cet API utilise un **RuleServiceProvider** pour avoir une instance de l'interface **RuleAdministrator**, qui fournit des méthodes pour inscrire et désinscrire des *execution sets*. L'API d'administration a le fonctionnement suivant :

- Récupérer une instance de l'interface **RuleAdministrator** en utilisant **RuleServiceProvider**.

- Créer une **RuleExecutionSet** depuis une ressource sérialisée ou non qui peut être :
  1. org.w3c.dom.Element pour un élément XML.
  2. java.io.InputStream pour lire depuis un flux binaire.
  3. java.io.Reader pour lire depuis un flux de caractères.
  4. java.lang.String pour lire depuis une URI.
- Enregistrer une **RuleExecutionSet** sur une URI pour les utilisations au niveau du runtime.
- Désenregistrer un **RuleExecutionSet** pour que l'utilisation ne soit plus possible au niveau du runtime.
- Retrouver les structures des méta-données des *execution sets* en retrouvant des objets **Rule** à partir d'un **RuleExecutionSet**.

- Accèder et modifier les propriétés spécifiques à l'application ou au constructeur sur les *execution set*.

10.3.2 L'API client runtime

Cet API permet d'accéder à l'API du moteur de règles de manière similaire que JDBC pour les bases de données. Les vendeurs (constructeurs de moteur de règles) permettent aux clients d'accéder à l'implémentation de leur moteur de règles à travers un **RuleServiceProvider**. Cette classe permet un accès mais aussi l'administration et l'exécution des règles. Chaque vendeur fournit l'URL d'un *rule service provider* unique. Et ce *rule service provider* devra être enregistré avec un **RuleServiceProviderManager** dans le but d'être utilisable par les clients.
L'épine dorsale de cette API est l'interface **RuleRuntime**, qui fournit aux clients des méthodes pour créer une **RuleSession** utilisée pour exécuter les règles. Une **RuleSession** est une connexion runtime entre un client et un moteur de règles ; elle est associée à une seule *execution set* et consomme des ressources du moteur de règles. Cependant elle doit être explicitement détruite si elle n'est plus utilisée par le client. Donc la **RuleSession** fait deux choses :

- Elle fournit un mécanisme pour accéder à toutes les *executions sets* qui sont enregistrées au niveau du **RuleServiceProvider**.
- Elle définit le type de session que le client veut ouvrir : stateful ou stateless :
  - Une **StatelessRuleSession** fonctionne sur la base d'une requête par client.
  - Une **StateFullRuleSession** est une session durant laquelle, les objets demeurent aussi longtemps que la connexion client-moteur de règles.

Les fonctionnalités de haut niveau du moteur de règles sont :

- Acquérir une instance du *rule service provider* du moteur spécifique depuis la classe **RuleServiceManager**.
- Acquérir une **RuleRuntime** depuis la **RuleServiceProvider**.
- Créer une **RuleSession** depuis la **RuleRuntime**.
- Obtenir une **java.util.List** des URIs enregistrés.
- Interagir avec une **RuleSession**.
- Retrouver des méta-données d'une **RuleSession** depuis l'interface **RuleExecutionSetMetadata**.
- Fournir une interface **ObjectFilter** pour filtrer les résultats obtenus à l'exécution d'un **RuleExecutionSet**.
- Utiliser des intances de **Handle** pour accéder aux objets ajoutés lors d'une **statefulRuleSession**.

L'extrait de code suivant montre comment utiliser l'API de la JSR94.

Exemple d'utilisation de la JSR94

```
1  // Loading this class will automatically register this provider
2  with the
```

```
3  // provider manager.
4  Class.forName("org.drools.jsr94.rules.RuleServiceProviderImpl");
5
6  // Get the rule service provider from the provider manager.
7  RuleServiceProvider serviceProvider =
8  RuleServiceProviderManager.getRuleServiceProvider("http://drools.org/RuleServiceProvider");
9  // Get the rule administrator.
10 RuleAdministrator ruleAdministrator = serviceProvider.getRuleAdministrator();
11
12 // Get an input stream to a test XML ruleset.
13 // This rule execution set is part of the TCK.
14 InputStream inStream =
15 org.jcp.jsr94.tck.model.Customer.class.getResourceAsStream(
16 "/org/jcp/jsr94/tck/tck_res_1.xml");
17
18 // Parse the ruleset from the XML document.
19 RuleExecutionSet res1 =
20 ruleAdministrator.getLocalRuleExecutionSetProvider(
21 null).createRuleExecutionSet( inStream, null );
22 inStream.close();
23
24 // Register the rule execution set.
25 String uri = res1.getName();
26 ruleAdministrator.registerRuleExecutionSet(uri, res1, null);
27
28 System.out.println("\nRuntime API\n");
29 RuleRuntime ruleRuntime = serviceProvider.getRuleRuntime();
30
31 // Create a statelessRuleSession.
32 StatelessRuleSession statelessRuleSession =
33 (StatelessRuleSession) ruleRuntime.createRuleSession(uri,
34 new HashMap(), RuleRuntime.STATELESS_SESSION_TYPE);
```

10.4 Conclusion

La JSR 94 offre une interface homogène pour interagir avec divers moteurs de règles. Elle est d'un grand apport en matière de standardisation dans ce domaine qui manque cruellement de standards.
Cependant il faut savoir que la JSR 94 ne dit absolument rien sur la manière de représenter une règle métier. L'objectif de départ de la spécification est amplement rempli, cependant le groupe de réflexion devrait s'impliquer beaucoup plus pour proposer un langage standard de formalisation des règles métier. En effet le plus dur n'est pas, en cas de changement de moteur de règles, de modifier l'utilisation d'une implémentation spécifique, mais plutôt de réécrire toutes les règles dans un autre formalise.

Chapitre 11

Conclusion et Perspectives

Conclusion

Dans ce document nous avons abordé 3 thèmes : l'approche par règles métier, l'ingénierie dirigée par les modèles et le Web Sémantique.
Nous avons introduit l'approche par règles métier ou Business Rule Approach en montrant comment elle se différencie de l'approche classique des cycles de développement. Cette différenciation se fait en mettant au cœur du cycle de développement les experts métier, qui ne sont pas des informaticiens, et qui connaissent mieux que quiconque le métier du futur système. Nous avons montré également comment par l'utilisation d'éditeurs de règles métier en langage naturel, l'expert métier peut mettre en œuvre le métier, libérant ainsi les experts technique qui ne s'occupent que de l'aspect système qui est déjà très ardu. Nous avons abordé une problématique de l'approche par règle métier, qui est le manque d'un formalisme standard de règles. Cette manque rend difficile tout changement d'un moteur de règles dans une application l'utilisant. Nous avons exposé certaines initiatives qui sont déjà en cours pour solutionner ce problème. Nous avons aussi exposé nos travaux dans ce sens avec ERML qui est notre langage de règles métier.
Concernant la méthodologie adoptée, nous avons abordé l'ingénierie dirigée par les modèles ou Model Driven Architecture. L'IDM nous a apporté de bonnes pratiques et un outillage pour faire de la transformation de modèle et de la génération de code.
Le dernier thème que nous avons abordé est celui du Web Sémantique qui, plus qu'une technologie, est un concept qui associe à chaque ressource et lien entre elles du Web actuel, des structures de données et des mécanismes afin de le rendre plus exploitable aussi bien par les hommes que par les agents artificiels.
La finalité de ces trois thèmes abordés, et de cette thèse, est la génération de règles métier de la couche d'exigences (CIM) du MDA à la couche de modèles concrets (PSM). Notre approche a été, dans un premier temps de nous focaliser sur la génération la plus opérationnelle et la plus simple à réaliser, à savoir celle entre la couche de modèles abstraits (PIM) et celle de modèles concrets. Cette génération se base sur l'établissement d'un formalisme

de règles métier indépendant de tout moteur et ensuite la mise en place d'un moteur de transformation pour avoir en sortie le formalisme spécifique à un moteur de règles donné. Dans la deuxième partie de notre travail, nous sommes remontés à la couche M3 (exigences) de l'IDM pour montrer comment, depuis des modèles UML existant, en les enrichissant sémantiquement nous pouvons générer des règles métier. Cet enrichissement sémantique se fait en utilisant des technologies du Web Sémantique.
Concrètement, les résultats de nos travaux ont été implémentés. La première partie, qui concerne ERML, a été intégrée dans la plateforme d'*e-Citiz* et est utilisée en production. La deuxième partie, qui concerne la génération des règles métier sur la couche d'exigences, a commencé à être implémentée dans le cadre d'un prototype.

Perspectives

Comme perspectives de nos travaux nous citons les sept suivants :

1. **Gérer la négation par échec.** Notre formalisme de règles métier prend en compte uniquement la négation forte (strong negation). Il faudrait mettre en place la négation par échec (negation as failure) pour d'autres règles qui ne sont pas de type production. En effet, dans le contexte des moteurs de règles actuels, plus orientés industrie, le problème ne se pose pas car étant dans un contexte de monde fermé (close world assumption). Contrairement à un contexte de monde ouvert (open world assumption) qui dit qu'une information peut être vraie, fausse ou inconnue, dans un contexte fermé, on a juste vrai ou faux, tout ce qui est inconnu est considéré comme étant faux. Etant dans un contexte industriel, nous nous sommes uniquement intéressé au contexte de monde fermé car plus réalisable.
2. **Formaliser la sortie du générateur de règles en langage naturel.** Nous avons vu que l'OMG a proposé le standard Semantic of Business Vocabulary and Business Rule pour représenter les règles en langage naturel. Cependant aucune implémentation n'est encore disponible. Les règles que nous générons avec le prototype sont en langage naturel mais ne sont pas formalisées. Il serait judicieux d'utiliser une implémentation de SBVR le jour où elle sera disponible.
3. **Générer des règles métier du CIM vers le PIM.** Nous avons vu que notre objectif est de générer des règles du CIM vers le PSM. Nous avons montré que nous savons générer des règles sur le CIM et également du PIM vers le PSM. Cependant il nous manque la génération du CIM vers le PIM. Ce manquement est une limitation héritée de l'ingénierie dirigée par les modèles car il y a peu de travaux, à l'heure actuelle, pour des transformations du CIM vers le PIM. Cependant, en arrivant à bien formaliser le résultat de la génération sur le CIM et en réutilisant les principes de l'IDM, une solution est possible.
4. **Utiliser d'autres types de modèles UML que ceux de classes**. Dans notre approche actuelle, nous utilisons les modèles de classes (diagramme de classes) pour faire de la génération de règles métier. Nous pensons qu'il est possible de se baser

sur des diagrammes d'activités pour générer , non pas uniquement des règles métier, mais aussi des processus métier.

5. **Améliorer l'éditeur de règles *e-Citiz*.** Cet éditeur de règles que nous avons réalisé pourrait être amélioré pour être plus ergonomique et éditer des règles plus compliquées avec imbrication de conjonctions et de disjonctions. Car, bien que le formalisme ERML supporte cette imbrication, l'éditeur ne la propose pas.
6. **Prendre en compte les types collections et tables dans l'éditeur.** *e-Citiz* possède des types complexes dont les tables et les collections. Pour l'instant l'éditeur de règles ne les prend pas en compte bien que cela soit fort utile.
7. **Externaliser du e-service les fichiers de règles et une partie de leur gestion.** Actuellement les fichiers contenant les règles sont inclus dans l'archive du e-service, ce qui fait que si quelqu'un modifie les règles, alors il faut refaire l'archive et la re-déployer. Ceci n'est pas une bonne solution et les fichiers de règles métier et certains mécanismes les gérant doivent être externaliser de l'archive afin de permettre des modifications à chaud des règles.

Bibliographie

[1] David Hay and Keri Anderson Healy. *Defining Business rules What Are They Really ?* Technical Report 1.3, The Business Rules Group, July 2000.

[2] The Semantic Web Activity. Semantic web. website, 2001.

[3] Ronald G. Ross. *The Business Rule Book.* Business Rule Solutions, LLC, Houston, USA, 1997.

[4] Hannu Jaakkola and Bernhard Thalheim. *Software Quality and Life Cycles.* In *ADBIS Research Communications*, 2005.

[5] Ronald G. Ross. *Principles of the Business Rule Approach.* Addison-Wesley, Boston, USA, 2003.

[6] Mouhamed Diouf, Joseph Xiong, Christelle Farenc, and Marco Winckler. *AUTOMATING GUIDELINES INSPECTION From Web site Specification to Deployment. CADUI*, 2006.

[7] Barbara von Halle. *Business Rules Applied.* John Wiley & Sons, New York, USA, 2002.

[8] Ernest Friedman-Hill. *JESS in Action.* Manning Publications Co, Greenwich, UK, 2003.

[9] D. Hay and K. A Healy. *GUIDE Business Rule Project Final Report.* Technical report, October 1997.

[10] Business Rule Group Community BRG. *http ://www.brcommunity.com.* Web page.

[11] Mouhamed Diouf, Kaninda Musumbu, and Sofian Maabout. *Standard Business Rules Language : why and how ? The 2006 International Conference on Artificial Intelligence*, June 2006.

[12] Mark Stefik. *Introduction to knowledge systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[13] A. Gupta, C. Forgy, and A. Newell. *High-speed implementations of rule-based systems. ACM Trans. Comput. Syst.*, 7(2) :119–146, 1989.

[14] Charles Forgy. *Rete : A Fast Algorithm for the Many Patterns/Many Objects Match Problem. Artif. Intell.*, 19(1) :17–37, 1982.

[15] Horst Bunke, Thomas Glauser, and T.-H. Tran. *An Efficient Implementation of Graph Grammars Based on the RETE Matching Algorithm.* In *Graph-Grammars and Their Application to Computer Science*, pages 174–189, 1990.

[16] Minsu Jang and Joo-Chan Sohn. *Bossam : An Extended Rule Engine for OWL Inferencing.* In *Rules and Rule Markup Languages for the Semantic Web*, volume 3323/2004, pages 128–138. Springer Berlin / Heidelberg, October 2004.

[17] Cláudia Antunes and Arlindo L. Oliveira. *Inference of Sequential Association Rules Guided by Context-Free Grammars.* In *ICGI*, pages 1–13, 2002.

[18] Jeong A. Kang and Albert Mo Kim Cheng. *Reducing Matching Time for OPS5 Production Systems.* In *COMPSAC*, pages 429–, 2001.

[19] Nancy Martin. *Programming Expert Systems in OPS5 - An Introduction to Rule-Based Programming(1).* In *Int. CMG Conference*, pages 829–832, 1985.

[20] Ilog Jrules. *Ilog Jrules, http ://www.ilog.com.*

[21] Drools. *Drools rule engine, http ://www.drools.org.*

[22] Fair Isaac. *Fair Isaac, www.fairisaac.com/rules.*

[23] Neil Madden. *Optimising RETE for Low-Memory Multiagent Systems.* In *GAME-ON*, pages 77–, 2003.

[24] Paul D. Vincent. *Fair Isaac Blaze Advisor Structured Rules Language - a commercial rules representation.* In *Rule Languages for Interoperability*, 2005.

[25] *Haley, www.haley.com.*

[26] David A. Brant, Timothy Grose, Bernie Lofaso, and Daniel P. Miranker. *Effects of Database Size on Rule System Performance : Five Case Studies.* In *VLDB*, pages 287–296, 1991.

[27] Harold Boley, Said Tabet, and Gerd Wagner. *Design Rationale for RuleML : A Markup Language for Semantic Web Rules.* In *SWWS*, pages 381–401, 2001.

[28] C. W. Tan and Angela Goh. *Implementing ECA rules in an active database. Knowl.-Based Syst.*, 12(4) :137–144, 1999.

[29] Stephen G. Pimentel and John L. Cuadrado. *A Horn Clause Theory of Inheritance and Temporal Reasoning.* In *EPIA*, pages 63–72, 1989.

[30] Dave Cuyler and Terry A. Halpin. *Two Meta-Models for Object-Role Modeling.* In *Information Modeling Methods and Methodologies*, pages 17–42. 2005.

[31] Frank Shou-Cheng Tseng and Teng-Kai Fan. *Extending the Concepts of Object Role Modeling to Capture Natural Language Semantics for Database Access.* In *Databases and Applications*, pages 234–239, 2005.

[32] IBM T.J. Watson Research Center. *CommonRules project.* Intelligent Agents project (1994-97), 1997.

[33] Benjamin N. Grosof. *Representing e-commerce rules via situated courteous logic programs in RuleML. Electronic Commerce Research and Applications*, 3(1) :2–20, 2004.

[34] Benjamin N. Grosof, Yannis Labrou, and Hoi Y. Chan. *A Declarative Approach to Business Rules in Contracts : Courteous Logic Programs in XML.* In ACM Press, editor, *ACM Conference on Electronic Commerce (EC99)*, New York, 1999.

[35] Benjamin N. Grosof and Yannis Labrou. *An Approach to Using XML and a Rule-Based Content Language with an Agent Communication Language.* In *Issues in Agent Communication*, pages 96–117, 2000.

[36] Chitta Baral and Michael Gelfond. *Logic Programming and Knowledge Representation. J. Log. Program.*, 19/20 :73–148, 1994.

[37] Thomas Cooper and Nancy Wogrin. *Rule-based programming with OPS5.* Morgan Kaufmann Publishers, San Francisco, CA, 1988.

[38] Ramesh Patil, Richard Fikes, Peter F. Patel-Schneider, Don McKay, Timothy W. Finin, Thomas R. Gruber, and Robert Neches. *The DARPA Knowledge Sharing Effort : A Progress Report.* In *KR*, pages 777–788, 1992.

[39] Michael R. Genesereth. *Knowledge Interchange Format.* In *KR*, pages 599–600, 1991.

[40] Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. *The XSB Programming System.* In *Workshop on Programming with Logic Databases (Informal Proceedings), ILPS*, page 164, 1993.

[41] Benjamin N. Grosof. *DIPLOMAT : Compiling Prioritized Default Rules into Ordinary Logic Programs, for E-Commerce Applications.* In *AAAI/IAAI*, pages 912–913, 1999.

[42] Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. *Smodels : A System for Answer Set Programming. CoRR*, cs.AI/0003033, 2000.

[43] Riichiro Mizoguchi and John K. Slaney, editors. *PRICAI 2000, Topics in Artificial Intelligence, 6th Pacific Rim International Conference on Artificial Intelligence, Melbourne, Australia, August 28 - September 1, 2000, Proceedings*, volume 1886 of *Lecture Notes in Computer Science*. Springer, 2000.

[44] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. *SWRL : A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission*, May 2004.

[45] RuleML. *The RuleML initiative.*

[46] OMG. *Business Semantics of Business Rules RFP.* br/2003-06-03, 2003.

[47] The Object Management Group OMG. *Semantics of Business Vocabulary and Business Rules (SBVR).* OMG Specification, March 2006.

[48] Donald Chapin. *Semantics of Business Vocabulary & Business Rules (SBVR).* In *Rule Languages for Interoperability*, 2005.

[49] The Object Management Group OMG. *Production Rule Representation (PRR) RFP.* OMG Request For Proposal (br/2003-09-03), 2003.

[50] The Object Management Group OMG. *UML 2.0 OCL Specification.* OMG Specification, October 2003.

[51] The Action Semantics Consortium. *Action semantics for the uml.* OMG Specification (ad/2001-03-01), March 2001.

[52] The Object Management Group OMG. *Meta Objet Facility (MOF) Specification Version 1.4.* OMG Specification (formal/02-04-03), April 2002.

[53] Fair Isaac-Corporation, Ilog SA, and IBM Corporation. *Production Rule Representation (PRR) Submission.* OMG Request For Proposal (bmi/2007-03-05), 2007.

[54] W3C. *Rule Interchange Format (RIF).* W3C Workgroup, 2005.

[55] Harold Boley and Michæl Kifer. *RIF Core Design, W3C Working 30 March 2007.* Technical report, W3C, 2007.

[56] Radim Belohlávek and Vilém Vychodil. *Fuzzy Horn logic I. Arch. Math. Log.*, 45(1) :3–51, 2006.

[57] Alexej P. Pynko. *A relative interpolation theorem for infinitary universal Horn logic and its applications. Arch. Math. Log.*, 45(3) :267–305, 2006.

[58] Alon Y. Levy and Marie-Christine Rousset. *CARIN : A Representation Language Combining Horn Rules and Description Logics.* In *European Conference on Artificial Intelligence*, pages 323–327, 1996.

[59] The Object Management Group OMG. *Model Driven Archtecture Guide Version 1.0.1.* OMG Specification, June 2003.

[60] Xavier Blanc. *MDA en action.* Eyrolles, France, 2005.

[61] The Object Management Group. *Unified Modeling Language : Superstructure.* OMG Specification, February 2004.

[62] The Object Management Group OMG. *Common Warehouse Metamodel (CWM) specification.* OMG Specification (ad/2001-02-01), February 2001.

[63] Miguel Ángel Sánchez Vidales, Ana Fermoso García, and Luis Joyanes Aguilar. *From the platform independent model (PIM) to the final code model (FCM) according to the model driven architecture (MDA).* In *IADIS AC*, pages 417–420, 2005.

[64] Yann-Gaël Guéhéneuc. *A reverse engineering tool for precise class diagrams.* In *CASCON*, pages 28–41, 2004.

[65] Pontus Boström, Mats Neovius, Ian Oliver, and Marina A. Waldén. *Formal Transformation of Platform Independent Models into Platform Specific Models.* In *B*, pages 186–200, 2007.

[66] Jean-Marc Jezequel. *A MDA Approach to Model & Implement Transformations.* In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[67] M. B. Kuznetsov. *UML model transformation and its application to MDA technology.* volume 33, pages 44–53, 2007.

[68] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. *Reflective Model Driven Engineering.* In *UML*, pages 175–189, 2003.

[69] The Object Management Group OMG. *MOF QVT Final Adopted Specification.* OMG Specification (ptc/05-11-01), April 2002.

[70] Xavier Blanc, Franklin Ramalho, and Jacques Robin. *Metamodel Reuse with MOF.* In *MoDELS*, pages 661–675, 2005.

[71] The Object Management Group OMG. *MOF 2.0/XMI Mapping Specification, v2.1.* OMG Specification (formal/05-09-01), 2005.

[72] T. Berners-Lee, J. Hendler, and O. Lassila. *The Semantic Web.* Scientiific American, May 2001.

[73] Tim Berners-Lee. *Weaving the Web.* Orien Business Books, London, UK, 1999.

[74] Tim Berners Lee. *Information Management : A Proposal.* CERN, March 1989.

[75] W3C workgroup on RDF(G. Glyne and J. Carroll). *Resource Description Framework (RDF) : Concepts and Abstract Syntax.* W3C Recommendation, 2005.

[76] José Kahan, Marja-Riitta Koivunen, Eric Prud'hommeaux, and Ralph R. Swick. *Annotea : an open RDF infrastructure for shared Web annotations. Computer Networks*, 39(5) :589–608, 2002.

[77] Marja-Riitta Koivunen. *Semantic Authoring By Tagging with Annotea Social Bookmarks and Topics.* In *Proc. of the 1st Semantic Authoring and Annotation Workshop (SAAW2006)*, 2006.

[78] *Annotea project.*

[79] Vincent Quint and Iréne Vatton. *Towards active web clients.* In *DocEng '05 : Proceedings of the 2005 ACM symposium on Document engineering*, pages 168–176, New York, NY, USA, 2005. ACM Press.

[80] Vincent Quint and Irène Vatton. *An introduction to Amaya. World Wide Web J.*, 2(2) :39–46, 1997.

[81] Brent Halsey and Kenneth M. Anderson. *XLink and open hypermedia systems : a preliminary investigation.* In *HYPERTEXT '00 : Proceedings of the eleventh ACM on Hypertext and hypermedia*, pages 212–213, New York, NY, USA, 2000. ACM Press.

[82] Paul Grosso, Eve Maler, Jonathan Marsh, and Norman Walsh. *XPointer Framework.* Technical report, W3C, March 2003.

[83] Aaron Swartz. *MusicBrainz : A Semantic Web Service. IEEE Intelligent Systems*, 17(1) :76–77, 2002.

[84] Hewlett-Packard Development Company. *Jena : A Semantic Web Framework for Java.* Framework, 2000.

[85] Brian McBride. *Jena : Implementing the RDF Model and Syntax Specification.* In *SemWeb*, 2001.

[86] HP Labs. *Semantic Web Tools : Jena Toolkit, Joseki, BrownSauce.*, 2003.

[87] Ossi Nykänen. *On implementing fuzzy semantic web agents with the cwm rule system.* In *ICWI*, pages 1075–1078, 2004.

[88] David Huynh, Stefano Mazzocchi, and David R. Karger. *Piggy Bank : Experience the Semantic Web Inside Your Web Browser.* In *International Semantic Web Conference*, pages 413–430, 2005.

[89] Thomas B. Passin. *Explorer's guide to the Semantic Web.* Manning Publications Co, Greenwich, UK, 2004.

[90] Martin J. O'Connor, Holger Knublauch, Samson W. Tu, Benjamin N. Grosof, Mike Dean, William E. Grosso, and Mark A. Musen. *Supporting Rule System Interoperability on the Semantic Web with SWRL.* In *International Semantic Web Conference*, pages 974–986, 2005.

[91] Brian McBride. *The Resource Description Framework (RDF) and its Vocabulary Description Language RDFS.* In *Handbook on Ontologies*, pages 51–66, 2004.

[92] Stephen Cranefield and Jin Pan. *Bridging the Gap Between the Model-Driven Architecture and Ontology Engineering. Proc. of AOSE 2004 Workshop*, 2004.

[93] Lee W. Lacy. *OWL : Representing Information Using the Web Ontology Language.* Trafford, Victoria, CA, 2005.

[94] Peter D. Karp, Vinay K. Chaudhri, and Jerome F. Thomere. *XOL : An XML-Based Ontology Exchange Language.* Technical Report 559, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Jul 1999.

[95] Robert Kent. *Conceptual Knowledge Markup Language : An introduction. Netnomics*, 2(2) :139–169, March 2000. available at http ://ideas.repec.org/a/kap/netnom/v2y2000i2p139-169.html.

[96] Robin McEntire, Peter D. Karp, Neil F. Abernethy, David Benton, Gregg Helt, Matt DeJongh, Robert Kent, Anthony Kosky, Suzanna Lewis, Dan Hodnett, Eric P. Neumann, Frank Olken, Dhiraj K. Pathak, Peter Tarczy-Hornoch, Luca Toldo, and Thodoros Topaloglou. *An Evaluation of Ontology Exchange Languages for Bioinformatics.* In *ISMB*, pages 239–250, 2000.

[97] Dan Connolly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. *DAML+OIL (March 2001) reference description.* Technical report, W3C note, 18 December 2001.

[98] Jeff Heflin and James A. Hendler. *Dynamic Ontologies on the Web.* In *AAAI/IAAI*, pages 443–449, 2000.

[99] Jeff Heflin, James A. Hendler, and Sean Luke. *SHOE : A Blueprint for the Semantic Web.* In *Spinning the Semantic Web*, pages 29–63, 2003.

[100] Deborah L. McGuinness, Richard Fikes, Lynn Andrea Stein, and James A. Hendler. *DAML-ONT : An Ontology Language for the Semantic Web.* In *Spinning the Semantic Web*, pages 65–93, 2003.

[101] Dieter Fensel, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, and Peter F. Patel-Schneider. *OIL : An Ontology Infrastructure for the Semantic Web. IEEE Intelligent Systems*, 16(2) :38–45, 2001.

[102] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. *Practical Reasoning for Expressive Description Logics. CoRR*, cs.LO/0005014, 2000.

[103] Ian Horrocks. *DAML+OIL : A Reason-able Web Ontology Language.* In *EDBT*, pages 2–13, 2002.

[104] I. Horrocks, P. Patel-Schneider, and F. van Harmelen. *From SHIQ and RDF to OWL : The making of a web ontology language. Journal of Web Semantics*, 1(1) :7–26, 2003rdf.

[105] W3C OWL M. K. Smith, C. Welty, and D. L. McGuinness. *OWL Web Ontology Language Reference.* W3C Standard, February 2004.

[106] Deborah L. McGuinness, Richard Fikes, James Hendler, and Lynn Andrea Stein. *IEEE Intelligent Systems : DAML + OIL : An Ontology Language for the Semantic Web.* 3(11), 2002.

[107] F. Baader, I. Horrocks, and U. Sattler. *Description logics as ontology languages for the semantic web*, 2003.

[108] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook : Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[109] Ronald J. Brachman and James G. Schmolze. *An Overview of the KL-ONE Knowledge Representation System. Cognitive Science*, 9(2) :171–216, 1985.

[110] Ian Horrocks and Peter F. Patel-Schneider. *Reducing OWL Entailment to Description Logic Satisfiability.* In *International Semantic Web Conference*, pages 17–29, 2003.

[111] Franz Baader, Ian Horrocks, and Ulrike Sattler. *Description Logics as Ontology Languages for the Semantic Web.* In *Mechanizing Mathematical Reasoning*, pages 228–248, 2005.

[112] V. Haarslev and R. Moller. *Racer : An owl reasoning agent for the semantic web*, 2003.

[113] Jing Mei and Elena Paslaru Bontas. *Reasoning Paradigms for OWL Ontologies.* Technical report, Freie Universität Berlin, November 2004.

[114] Dimitrios A. Koutsomitropoulos, Dimitrios P. Meidanis, Anastasia N. Kandili, and Theodore S. Papatheodorou. *OWL-Based Knowledge Discovery Using Description Logics Reasoners.* In *ICEIS (4)*, pages 43–50, 2006.

[115] Joseph Kopena and William C. Regli. *DAMLJessKB : A Tool for Reasoning with the Semantic Web. IEEE Intelligent Systems*, 18(3) :74–77, 2003.

[116] Brian McBride. *Jena : A Semantic Web Toolkit. IEEE Internet Computing*, 6(6) :55–59, 2002.

[117] Christian Fillies, Gary Ng, and Annika Thunell. *Cerebra Construct : Inferences for End Users.* In *WWW (Posters)*, 2003.

[118] Ian Horrocks. *FaCT.* In *Description Logics*, 1998.

[119] Ian Horrocks. *The FaCT System.* In *TABLEAUX*, pages 307–312, 1998.

[120] Ian Horrocks and Ulrike Sattler. *Optimised Reasoning for SHIQ.* In *ECAI*, pages 277–281, 2002.

[121] V. Haarslev and R. Möller. *Description of the RACER System and its Applications.* In *Proceedings International Workshop on Description Logics (DL-2001), Stanford, USA, 1.-3. August*, pages 131–141, 2001.

[122] Ana Gabriela Garis, Daniel Riesco, German Montejano, and Narayan C. Debnath. *UML Profiles for Design Patterns.* In *Computers and Their Applications*, pages 435–440, 2005.

[123] The Object Management Group OMG, IBM, and Sandpiper Software. *Ontology Definition Metamodel.* OMG Specification, June 2006.

[124] Yue Pan, Guotong Xie, Li Ma, Yang Yang, ZhaoMing Qiu, and Juhnyoung Lee. *IBM Research Report : An MDA-based system for Ontology Engineering.* Technical Report RC23795, IBM Research Division, Yorktown Heights, NY US, November 2005.

[125] The Object Management Group OMG. *Request For Proposal for Ontology Definition Metamodel.* OMG Request For Proposal, March 2003.

[126] Kenneth Baclawski, Mieczyslaw K. Kokar, Paul A. Kogut, Lewis Hart, Jeffrey Smith, William S. Holmes III, Jerzy Letkowski, and Michael L. Aronson. *Extending UML to Support Ontology Engineering for the Semantic Web. Lecture Notes in Computer Science*, 2185 :342+, 2001.

[127] Kenneth Baclawski, Mieczyslaw M. Kokar, Jeffrey E. Smith, Evan Wallace, Jerzy Letkowski, Manfred R. Koethe, , and Paul Kogut. *UOL : Unified Ontology Language. Assorted paper discussed at the DC Ontology SIG Meeting*, November 2002.

[128] Saartje Brockmans, Raphael Volz, Andreas Eberhart, and Peter Löffler. *Visual Modeling of OWL DL Ontologies Using UML.* In *International Semantic Web Conference*, pages 198–213, 2004.

[129] Stephen Cranefield. *Networked Knowledge representation and exchange using UML and RDF. Journal of digital information*, 1(8), 2001.

[130] Dragan Djuric, Dragan Gasevic, and Vladan Devedzic. *Ontology Modeling and MDA. Journal of Object Technology*, 4(1) :109–128, 2005.

[131] Kateryna Falkovych, Marta Sabou, and Heiner Stuckenschmidt. *UML for the Semantic Web : Transformation-Based Approaches.* In *Knowledge Transformation for the Semantic Web*, pages 92–106. 2003.

[132] Elisa F. Kendall, Mark E. Dutra, and Deborah L. McGuinness. *Towards A Commercial Ontology Development Environment.* In *Proceedings of the 1st International Semantic Web Consference (Posters and Demos)*, 2002.

[133] Dragan Durić. *MDA-based Ontology Infrastructure. Proc. of ComSIS 2004, Vol 1, No 1*, February 2004.

[134] Dragan Gaěvič, Dragan Djurié, and Vladan Devedžić. *Model Driven Architecture and Ontology Development.* Springer-Verlag, Berlin, DE, 2006.

[135] Dragan Djuric, Dragan Gasevic, Vladan Devedzic, and Violeta Damjanovic. *A UML Profile for OWL Ontologies.* In *MDAFA*, pages 204–219, 2004.

[136] Mouhamed Diouf, Maabout Sofian, and Musumbu Kaninda. *Génération automatique de règles métier par enrichissement sémantique de modèles.* In *Actes de la conférence INFORSID*, Perros Guirec, France, Mai 2007.

[137] Mouhamed Diouf, Kaninda Musumbu, and Sofian Maabout. *Adding Semantics on Models for Automatic Business Rules Generation.* In Thomas Andre Artiba Abdelhakim Xu Zongwei Yang Shanlin, Chen Guoqing, editor, *International Conference on Industrial Engineering and Systems Management.* Tsinghua University Press, May 2007.

[138] Mouhamed Diouf, Kaninda Musumbu, and Sofian Maabout. *Génération et fusion de règles métier par tissage du MDA et le Web Sémantique.* In RJCIA2007, editor, *Proceedings of RJCIA2007*, Juillet 2007.

[139] Mouhamed Diouf, Sofian Maabout, and Kaninda Musumbu. *Merging Model Driven Architecture and Semantic Web for business rules generation.* In *The First International Conference on Web Reasoning and Rule Systems (RR07)*, LNCS, Innsbruck (Austria), June 2007. Springer.

[140] Mouhamed Diouf, Sofian Maabout, and Kaninda Musumbu. *Semantics enrichment in Model Driven Architecture : automatic business rules generation.* In *Proceedings of the International Conference on E-Business, Enterprise Information Systems and E-Government*, Las Vegas (USA), June 2007. CSREA Press.

[141] Mouhamed Diouf, Sofian Maabout, and Kaninda Musumbu. *Merging and aligning business rules sets using Semantic Web technics.* In *Proceedings of the International Conference on Semantic Web and Web Services (SWWS)*, Las Vegas (USA), June 2007. CSREA Press.

[142] Mouhamed Diouf, Sofian Maabout, and Kaninda Musumbu. *GENERATING AND MERGING BUSINESS RULES by weaving MDA and Semantic Web.* In Luis Ferreira Pires and Slimane Hammoudi, editors, *Proceedings of the 3rd International Workshop on Model-Driven Enterprise Information Systems (MDEIS)*, number 19-28, Funchal (Madaira), June 2007. CSREA Press.

[143] Java Community Process(JCP). *Java Metadata Interface (JMI).* Sun Java Specification Request (JSR 40), 2002.

[144] H. Knublauch. *Ontology-Driven Software Development in the Context of the Semantic Web : An Example Scenario with Protege/OWL. 1st International Workshop on the Model-Driven Semantic Web (MDSW2004)*, 2004.

[145] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, Bernardo C. Grau, and James Hendler. *Swoop : A Web Ontology Editing Browser.* In *Semantic Grid –The Convergence of Technologies*, volume 4, pages 144–153, June 2006.

[146] Natalya Fridman Noy and Mark A. Musen. *PROMPT : Algorithm and Tool for Automated Ontology Merging and Alignment.* In *AAAI/IAAI*, pages 450–455, 2000.

[147] Michele Bugliesi, Evelina Lamma, and Paola Mello. *Modularity in Logic Programming. J. Log. Program.*, 19/20 :443–502, 1994.

[148] Sun JAXB. Jaxb, http ://java.sun.com/webservices/jaxb/.

[149] Eclipse project. *Eclipse Ontology Definition Metamodel project.* Eclipse project, 2006.

[150] Java Community Process(JCP). *JSR 94 API for business rules.* Sun Java Specification Request (JSR 94), 2000.

Index

D

E

F

V

W

Spécification et mise en œuvre d'un formalisme de règles métier

Résumé : Cette thèse CIFRE se positionne dans le cadre du génie logiciel en ayant pour objectif la facilitation de l'intégration de l'approche par règles métier en proposant un formalisme indépendant de tout moteur de règles et en générant une partie de ces règles.

Dans un premier temps, nous introduisons l'approche par règles métier comme étant une alternative à l'approche classique des cycles de développement. Nous introduisons également le formalisme de règles, les systèmes de gestion de règles métier et présentons un banc d'essai des moteurs de règles les plus utilisés. Dans une seconde partie nous parlons de l'ingénierie dirigée par les modèles et comment nous nous en servons. Dans une troisième partie nous parlons du Web Sémantique et de ses technologies que nous utilisons. Dans la quatrième partie nous montrons comment nous générons des règles en langage naturel en nous basant sur la syntaxe et la sémantique du langage d'ontologie Web. Nous finirons par la cinquième partie dans laquelle nous exposons les implémentations que nous avons réalisées.

Mots-clés : règles métier, ingénierie dirigée par les modèles, web sémantique, formalisme de règles, raisonnement, enrichissement sémantique, e-service.

Specification and implementation of a business rule's formalism

Abstract : This thesis is about software engineering and focuses on how to facilitate the integration of the business rules approach. We propose a rule formalism independently of any rule engine. We also propose an approach for generating a part of these rules based on UML models.

First we start by introducing the business rules approach like an alternative to the classical approach of software development cycles. We introduce also rules formalism, business rules management systems and present a workbench for the most used rules engines. In the second part we talk about model driven architecture and how we use that. In the third part we introduce the Semantic Web and how we use that. In the fourth part we explain our approach for generating business rules in natural language based on the syntaxe and semantics of the web ontology language. Will finish by the fifth part in which we expose implementations we have realized.

Keywords : business rules, model driven architecture, semantic web, rule formalisme, reasoning, semantics enrichment, e-service.

Thèse en informatique, préparée, dans le cadre d'une convention CIFRE, au *LaBRI* (Université Bordeaux I, 351 cours de la Libération, 33405 Talence) et chez *Génigraph* (Tertial II 216, Route de St Simon 31100 Toulouse).