Hes·so VALAIS WALLIS
Haute Ecole Spécialisée de Suisse occidentale

Travail de diplôme 2007

Filière Informatique de gestion

Web 3.0 : Interrogation intelligente

Etudiant : Favre Frédéric

Professeur : Anne Le Calvé

www.hevs.ch

Préface

Actuellement les données disponibles sur le Web n'ont de sens que pour l'utilisateur. Des solutions bien plus puissantes seront envisageables lorsque les machines comprendront et pourront traiter ces données. L'un des grands enjeux futurs est de permettre la mise en relation de l'ensemble des données et de les valoriser par un système de déductions performant. Le web sémantique est la technologie centrale dans le développement de ce Web 3.0. Le projet Memoria-Mea se prépare déjà à ce changement majeur en incluant un moteur sémantique pour le traitement de ses données.

Nowadays, data available on the Web make sense only to humans. Much powerful solutions will be possible when those data will make sense to the machines too. The challenge at stake is to link all the data and enrich the knowledge with powerful reasoning systems. The semantic Web is the core technology in the development of this Web 3.0. The Memoria-Mea project already takes a step toward this major change by including a semantic knowledge base engine to handle its data.

Favre Frédéric
HES-SO Valais, décembre 2007

1. TABLE DES MATIÈRES

2. PRÉSENTATION DU TRAVAIL

2.1 Description

Memoria-Mea est un projet de gestion de données personnelles composé de plusieurs modules distincts. Ces modules fournissent toutes sortes d'informations liées à un utilisateur (ses données personnelles, ses relations, ses fichiers multimédias, ses données géographiques, …) au moteur sémantique OntoMea. Le web sémantique permet de mettre à disposition des données non plus pour l'utilisateur mais également pour les machines. Ces données sont appelées métadonnées et sont structurées selon des normes établies par le W3C.

Installé en local sur la machine de l'utilisateur, OntoMea est un prototype de moteur sémantique permettant de gérer une base de connaissances et d'effectuer des déductions de nouvelles informations grâce à un moteur d'inférences intégré qui utilise des ontologies définies. Développé en Java, il offre aux différents modules plusieurs services accessibles grâce à un serveur Web embarqué. Ces services permettent notamment de gérer la base de connaissances et d'effectuer des requêtes.

Le but de ce travail de diplôme est d'améliorer la solution existante notamment en lui intégrant un moteur d'inférences capables d'effectuer un large éventail de déductions sur de grandes quantités de données.

Dans un premier temps, il est nécessaire d'implémenter un système de stockage et de mise à jour performant des données afin d'optimiser la gestion des informations et des inférences. Ensuite, après un travail d'analyse des solutions existantes, un moteur de déductions robuste et performant doit être intégré au prototype.

2.2 Déroulement du projet

Afin de mener à bien le travail à réaliser, la planification comporte cinq phases. Pour plus de détail sur l'emploi du temps, consulter l'annexe 2 : Emploi du temps.

- Une phase de recherche et de compréhension des technologies. Cette partie a pour but de se familiariser avec le Web sémantique et avec le prototype d'OntoMea. Le cahier des charges est rédigé à la fin de cette phase. Durée : 10 jours.
- Une phase d'analyse des outils et de tests comparatifs. Les différentes solutions permettant de réaliser le travail sont implémentées et testées. Durée 13 jours.
- Une phase de développement durant laquelle les différentes fonctionnalités à apporter sont implémenter dans la solution initiale du moteur OntoMea. Puis une application de démonstration sur les points d'intérêts est réalisée. Durée 24 jours.
- Une phase de tests et de débogage. Durée 2 jours.
- Une phase de documentation comprenant la rédaction de ce document ainsi que de ses annexes. Durée 10 jours.

2.3 Travail à réaliser

Moteur OntoMea

Le moteur OntoMea est installé sur les postes de chaque utilisateur du Memoria-Mea. Il est basé sur la librairie Jena qui lit les ontologies et les données. Chaque utilisateur dispose de ses données personnelles, le serveur doit pouvoir les utiliser ainsi que d'autres données réparties sur le web.

De nombreux modules peuvent être installés sur la même machine afin d'accéder à la base de connaissances. Pour effectuer des requêtes spécifiques, le serveur dispose de divers services effectuant des requêtes SPARQL sur les données et retournant le résultat au format XML.

Inférence :

- Analyse, tests et choix d'un moteur d'inférences.
- Mise en place d'un moteur d'inférences.
- Définir le format des règles de déductions selon les standards et la faisabilité technologique.
- Créer des règles de déductions.

Stockage et gestion des données :

- Gérer le stockage des données (fichiers, MYSQL, HSQLDB, …) en tenant compte de la faisabilité technologique et des performances.
- Gérer la mise à jour des données en tenant compte des contraintes du web sémantique.
- Attribuer de façon efficace des URI aux différentes ressources.

Nice To Have :

- Analyser la couche de communication entre le serveur OntoMea et les différents modules puis implémenter au besoin la meilleure solution (Web Services, Semantic Web Services, …)
- Inclure la possibilité d'ajouter des données provenant du Web (Points of Interests)
- Effectuer des tests de montée en charge avec des banques de données conséquentes (GeoNames, …)

Modèle de données

Les données sont structurées selon les règles établies pour le web sémantique. Le modèle utilisé par le serveur OntoMea se base sur des ontologies déjà existantes (foaf, geoNames) afin de permettre une intégration facile de diverses données dans le système.

Must Have :

- Rechercher les ontologies existantes et les réutiliser si possible (POI).
- Définir le type des données nécessaires (Profil, Localité, Document multimédia, POI) et les inclure dans le modèle.

Intégration d'applications externes

Afin de démontrer la flexibilité des ontologies grâce au web sémantique, des données externes sont intégrées dans le système. Les données concernant les POI d'un utilisateur sont stockées dans une base de données SQLSERVER, un service est nécessaire afin de les transformer en données XML utilisables par le système. Ces données peuvent, par la suite, être utilisées par les différents modules de Mémoria-Mea.

Must Have

- Créer un modèle ajoutant les POI au modèle de base.
- Développer un service de conversion de données SQLSERVER → RDF
- Utiliser ses nouvelles données sur un client mobile.

2.4 Structure du document

Maintenant que les bases du travail ont été posées, il paraît important d'expliquer en quelques mots la structure de ce rapport. A noter que les chapitres suivent la planification du travail définie dans le déroulement du projet.

Dans un premier temps, les notions propres au Web sémantique sont expliquées dans un chapitre théorique intitulé Le Web Sémantique. Ces notions apparaissant dans tous les chapitres de ce document, il est essentiel de les dégrossir à ce stade. A noter que la plupart des termes importants sont définis succinctement dans le Glossaire. Ensuite, la création d'une ontologie avec le logiciel Protégé est détaillée dans le chapitre Ontologies OWL avec Protégé. L'accent est mis particulièrement sur les notions de restriction, d'inférence et de raisonneur qui seront souvent utilisées par la suite. Ces deux chapitres correspondent à la première étape de la planification du travail.

A ce stade, la plupart des notions du Web sémantiques ont été définies. Les deux chapitres suivants correspondent à la phase d'analyse. Le premier, intitulé Analyse des outils, est une étude des différentes solutions disponibles. Les différentes fonctionnalités de chacune d'entre elles sont décrites et accompagnées d'exemples d'implémentations. Puis, dans le chapitre Comparatif des outils, on retrouve un récapitulatif comprenant différents comparatifs accompagnés de graphiques et des résultats des différents tests effectués.

A cette étape, la meilleure solution a été définie. Les chapitres suivants traitent du développement des différentes applications. La partie Le moteur OntoMea est véritablement le chapitre central de ce rapport. Les différentes

améliorations apportées au moteur sémantique y sont décrites en détails. Le chapitre L'interface des points d'intérêts permet de décrire un cas d'utilisation concret d'OntoMea. Des exemples d'utilisation comme la gestion et le transfert des données, les déductions du moteur d'inférences ou la mise à disposition de données sémantiques sur le Web sont abordés.

Finalement, dans la Conclusion, un bilan du travail réalisé est dressé. Les possibilités et les limites du web sémantique dans le cadre de ce projet y sont analysées. Pour finir, les différentes possibilités d'évolutions futures sont décrites et brièvement commentées.

3. LE WEB SÉMANTIQUE

3.1 Introduction

« Le Web sémantique désigne un ensemble de technologies visant à rendre le contenu des ressources du World Wide Web accessible et utilisable par les programmes et agents logiciels, grâce à un système de métadonnées formelles, utilisant notamment la famille de langages développés par le W3C. »[1]

L'avantage du Web sémantique est qu'il utilise les mêmes fonctionnalités que le Web classique. Il permet de publier et de distribuer des documents. Contrairement à la plupart des pages existantes à l'heure actuelle, un site utilisant les technologies du Web sémantique fournit, en plus d'informations compréhensibles par l'utilisateur, des informations appelées métadonnées compréhensibles par les logiciels. La notion de métadonnées avait déjà été évoquée en 1994 par Tim Berners-Lee, l'inventeur du Web.

Afin d'établir une norme permettant aux différentes applications utilisant le Web sémantique de se comprendre, le W3C a publié plusieurs langages permettant de standardiser la structure des ressources. Parmi ceux-ci, on distingue deux types de langages :

- Les langages permettant de décrire des données comme RDF (Resource Description Framework).
- Les langages permettant de structurer les données comme OWL (Web Ontology Language).

Un parallèle peut être fait avec la programmation orientée objet. Le premier type de langage représente les instances et le second, les classes.

Alors que le Web 2.0 a surtout révolutionné la couche visible du Web avec des technologies telles que AJAX, le Web 3.0 révolutionnera les couches profondes, notamment, la structure des données à travers le Web sémantique.

[1] Source Wikipedia Web sémantique

3.2 Les métadonnées

Définition

Les métadonnées sont des données qui décrivent des données. Dans le Web sémantique il existe plusieurs types de métadonnées. Il est possible de décrire des instances, des concepts ou des propriétés.

- Les instances représentent une information précise. Par exemple, lorsque l'on gère des utilisateurs, Pierre et Paul sont des instances d'utilisateur.
- En reprenant l'exemple ci-dessus, Pierre et Paul font partie du concept d'Utilisateur. Dans le Web sémantique, ces concepts sont décrits dans des ontologies ou schémas.
- Les propriétés permettent de mettre en relations des instances avec d'autres instances ou avec des littéraux. Pierre connaît Paul ou Pierre a 30 ans sont des propriétés.

A noter que chaque instance possède une URI définie permettant de distinguer la ressource. Par exemple, dans le cadre de Memoria-Mea, les utilisateurs sont définis selon l'URI suivante :

```
http://www.memoria-mea.ch/user/nom
```

Les triplets

Le Web sémantique structure les données sous forme de triplets. Ces triplets sont composés des éléments suivants :

- Un sujet qui est en général une instance. Il s'agit de l'élément décrit par le triplet.
- Un prédicat qui représente un type de propriété applicable au sujet.
- Un objet qui, comme cité ci-dessus, peut être une autre instance ou un littéral.

Cette structure permet de lier les différentes instances entre elles. Le résultat obtenu est appelé graphe, un réseau de données semblable à une toile. Une telle structure permet de mettre en relation toutes sortes d'informations provenant de n'importe quels domaines du Web pour peu qu'ils respectent la même structure de données. Certaines sources utilisent le mot « modèle » à la place de « graphe ». Dans ce rapport, les deux mots ont la même signification.

Ontologies

Une ontologie permet de structurer un ensemble de concepts afin de leur donner un sens. Elle définit notamment l'ensemble de rapport possible entre ces concepts et permettent de créer des restrictions et des conditions sur ces liens. Une ontologie peut être utilisée pour déduire de nouvelles informations à partir des informations disponibles. On parle alors d'inférences.

Une ontologie est souvent appelée schéma ou schéma d'ontologie. Dans ce rapport, ces termes ont la même signification.

3.3 Les langages

RDF (Resource Description Framework)

Le langage RDF est à la base du Web sémantique. La syntaxe principale utilisée par RDF est RDF/XML. En prenant l'exemple utilisé ci-dessus, il est possible de l'intégrer dans une page Web grâce code suivant :

```
<rdf:RDF>
  <Utilisateur rdf:about="Paul" />
  <Utilisateur rdf:about="Pierre">
     <connait rdf:resource="Paul"/>
     <aAge>30</aAge >
  </Utilisateur>
</rdf:RDF>
```

RDF sert de base aux langages plus complexes du Web sémantiques notamment, les langages d'ontologie tels que RDFS et OWL.

RDFS (RDF Schema)

RDFS est une extension du langage RDF. Il permet de décrire des ontologies simples en ajoutant des notions de classe, de sous-classe, de sous-propriété, de domaine et de portée. RDFS est inclus dans le langage d'ontologie OWL.

OWL (Web Ontology Language)

OWL permet de définir des ontologies structurées. En plus des concepts de RDF et RDFS, OWL ajoute les concepts de classes équivalentes, de propriété équivalente, d'égalité de deux ressources, de leurs différences, du contraire, de symétrie et de cardinalité.[2]

[2] Source Wikipedia OWL

OWL définit trois sous-langages différents :

- OWL/Lite est de la version la moins expressive de OWL.
- OWL/DL est une version plus complète. Il s'agit, en général, de la version utilisée par les raisonneurs tels que Racer ou Pellet.
- OWL/Full est la version la plus complète. Elle est destinée aux utilisateurs désirant une description très précise des concepts. Cependant, aucun raisonneur ne supporte à l'heure actuelle cette version.

Le chapitre « Ontologies OWL avec Protégé » reprend en détail les différentes possibilités du langage OWL notamment, les différentes propriétés et la notion d'inférence.

SPARQL

Le langage SPARQL définit la syntaxe des requêtes effectuées sur un graphe de données RDF. Inspiré du langage SQL, il se base sur les triplets contenus dans le graphe. La structure d'une requête SPARQL est la suivante :

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT DISTINCT ?nom ?image ?description
WHERE {
      ?personne rdf:type foaf:Person .
      ?personne foaf:name ?nom .
      ?image rdf:type foaf:Image .
      ?personne foaf:img ?image .
      ?image dc:description ?description
}
```

Elle est composée de trois parties distinctes :

- Les PREFIX qui sont en fait les préfixes des URI des ressources à utiliser.
- La clause SELECT, semblable à celle du langage SQL, permet de définir les différentes ressources retournées par la requête.
- La clause WHERE, composée d'un ensemble des triplets, permet de définir les conditions dans la sélection.

3.4 Stockage des données

Fichiers

A la base, les données sémantiques sont stockées dans des fichiers RDF ou OWL ou, directement dans des pages Web HTML, ASPX ou PHP. Les applications utilisant ces données sont capables de les reconnaître grâce à la structure RDF/XML. Ce type de stockage privilégie la facilité d'accès aux données au travers du Web et au détriment de la performance.

Triples Store

Les données sémantiques étant structurées en triplets, il est possible de les stocker dans des architectures de base de données directement sous forme de graphes. Cette solution, appelée Triple Store, permet également de classer les données dans plusieurs graphes différents appelés Named graphs. Le principal inconvénient des Triples Store est de devoir centraliser l'ensemble des données dans une base de données. Les informations ne sont donc plus utilisées directement depuis le Web. Cependant, les performances en lecture sont très largement en dessus du stockage fichier.

3.5 La notion d'inférence

Introduction

En se basant sur les ontologies, il est possible de déduire de nouvelles informations à partir des données de base. A la base, l'inférence est un concept philosophique qui décrit une opération mentale permettant de tirer des conclusions à partir de règles données. Dans le cadre du Web sémantique, l'opération est effectuée par des algorithmes de raisonnements complexes et les règles sont structurées dans les langages tels qu'OWL et RDFS.

Raisonneurs

Un raisonneur est un logiciel permettant d'effectuer des inférences. La plupart utilisent le sous-langage OWL/DL. Ces raisonneurs sont pourvus d'une interface DIG (développée par DL Implementation Group) permettant d'échanger des données logiques dans un standard XML. Parmi les raisonneurs disponibles, citons ceux utilisés durant ce projet :

- Pellet
- Racer

4. ONTOLOGIES OWL AVEC PROTÉGÉ

4.1 Introduction

Ce chapitre a pour but d'expliquer les principes de base d'une ontologie à travers le logiciel Protégé. Cela permet, grâce à l'interface graphique, de présenter et d'expliquer plus concrètement les notions de schéma, de classe, de propriété ou d'inférence expliquées dans le chapitre Le Web Sémantique.

Les notions présentées ont été sélectionnées par rapport à leur importance pour la suite du rapport. Il ne s'agit donc pas d'un tutorial complet de l'outil Protégé. Un tel tutorial existe à l'adresse :

- http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf

4.2 Présentation de Protégé

Protégé est un éditeur d'ontologies Open Source. La plateforme Protégé permet la création et l'édition d'ontologies grâce à deux outils distincts :

- Protégé-Frame permet de créer facilement une interface graphique afin de gérer une ontologie. Cet outil ne demande aucune notion de programmation. Il génère automatiquement les formulaires nécessaires en se basant sur le schéma d'ontologie créé. Il offre également la possibilité de personnaliser l'interface selon les besoins de l'utilisateur
- Protégé-OWL est une extension de Protégé qui supporte le langage OWL. Il permet de décrire plus précisément les classes, les propriétés et les instances grâce aux nombreuses propriétés offertes par OWL. Il est également possible d'interroger un raisonneur via une interface DIG afin de contrôler l'intégrité du modèle et de créer un modèle d'inférences.

Pour la suite de ce descriptif, l'extension Protégé-OWL a été utilisée notamment grâce à la possibilité de créer des données inférées. L'exemple utilisé pour la démonstration est basé sur le tutorial de Protégé-OWL (voir Références). Il s'agit d'une ontologie décrivant pizzas.

4.3 Création d'une ontologie OWL

Classes

La première étape, lors de la création d'une ontologie OWL, est d'organiser les différents éléments nécessaires en classes.

La hiérarchie des classes est extrêmement importante car, à l'instar de la programmation orientée objet, les classes filles hériteront des propriétés et caractéristiques des classes parentes. Il est possible de définir deux sortes de classes :

- Les classes primaires sont les classes de base. Chaque instance doit appartenir au moins à une de ces classes.
- Les classes définies ne possèdent pas d'instances dans le modèle de base. Elles possèdent uniquement des restrictions (voir Restrictions). Il est nécessaire d'utiliser un raisonneur qui déduira quelle instance appartient à la classe définie dans le modèle d'inférences.

Pour décrire cette hiérarchie, les propriétés RDFS (intégrées dans OWL) *rdfs:subClassOf* et *rdf:type* sont utilisées. Il suffira donc d'un raisonneur RDFS pour effectuer des déductions sur ces propriétés.

RDFS permet également de définir certaines informations de descriptions de la classe :

- rdfs:label indique le nom de la classe
- rdfs:comment permet d'ajouter certains commentaires notamment la description
- owl:versionInfo détermine la version
- …

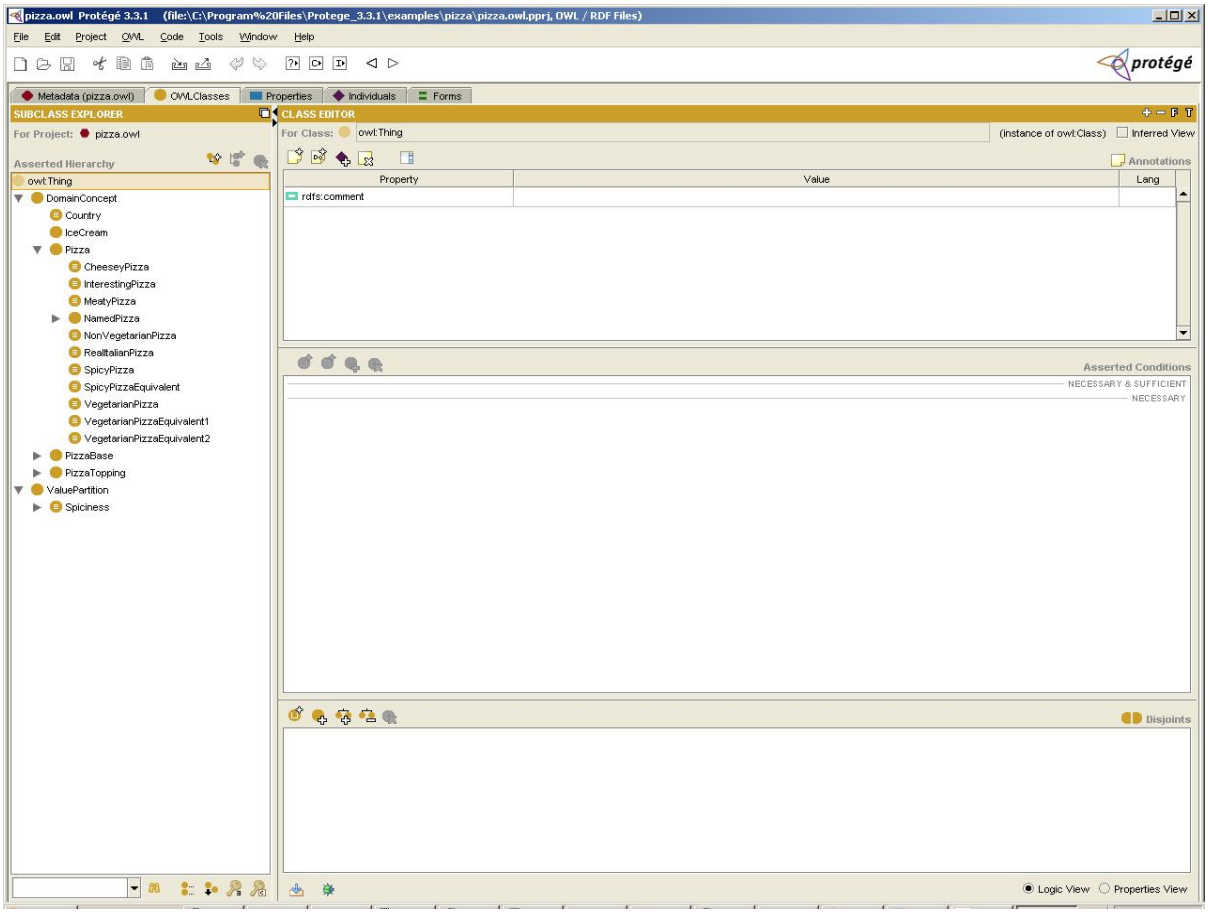


Figure 1 : Interface de gestion des classes

Classes disjointes

La méthode *owl:disjointWith* définit qu'une instance ne peut pas appartenir à deux classes disjointes.

Il s'agit ici d'une propriété OWL. Pour pouvoir créer des inférences à partir d'un modèle utilisant ces propriétés, il faut utiliser un raisonneur capable d'interpréter le langage OWL.

Dans Protégé-OWL, la création de classes disjointes est automatisée. Il est possible de définir une classe fille comme disjointe de toutes les autres classes filles d'une classe parente donnée.

Cette propriété permet de mettre en évidence une autre fonctionnalité intéressante du raisonneur (voir Utilisation d'un raisonneur). En effet, si par exemple, une classe définie a comme restriction d'appartenir à deux classes disjointes simultanément, le raisonneur permet de ressortir cette incohérence et définit le schéma comme non-valide.

Figure 2 : Classes disjointes

Propriétés OWL

L'interface de gestion des propriétés OWL est similaire à celle de la gestion des classes. Il est également possible d'utiliser l'héritage en créant une hiérarchie grâce la propriété *rdfs:subPropertyOf*.

En plus de *owl:ObjectProprety*, chaque propriété peut appartenir au classes suivantes :

- *owl:FunctionalPropriety* définit que l'ensemble des objets liés à un sujet donné par une propriété de ce type sont égaux.
- *owl:InverseFunctionalPropriety* définit que l'ensemble des sujets liés à un objet par une propriété de ce type sont égaux.
- owl:SymmetricProprety définit que si un sujet A est lié à un objet B par la propriété P alors B est lié à A par la propriété P.
- owl:TransitivePropriety définit que si A -> B -> C alors A -> C

De plus, la propriété *owl:inverseOf* permet de définir la propriété inverse à une propriété donnée. Evidemment, la création d'inférences basées sur ces propriétés et ces classes nécessite l'utilisation d'un raisonneur OWL.

Dans Protégé-OWL, toutes ces fonctionnalités sont accessibles directement depuis l'interface dédiée aux propriétés.

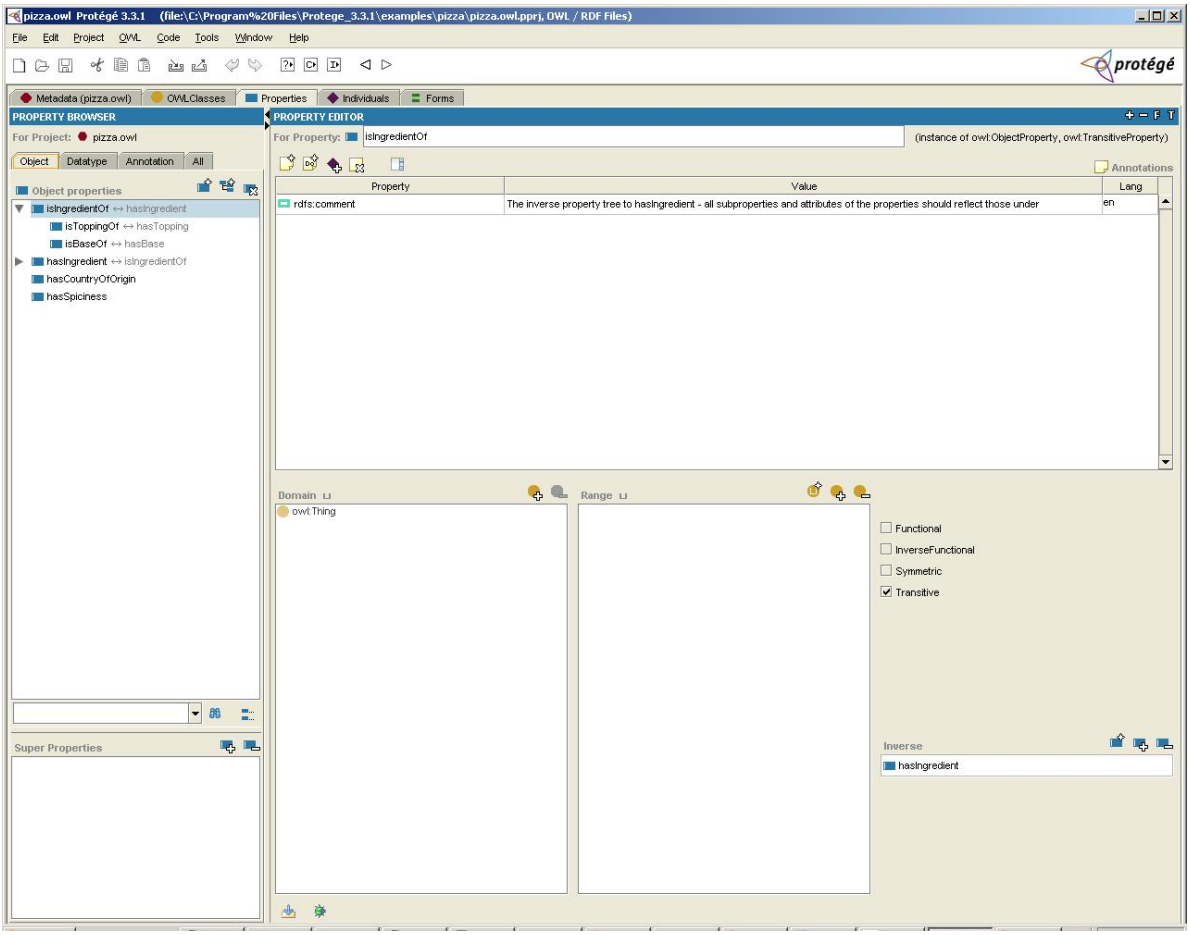


Figure 3 : Interface de gestion des propriétés

Domain et range

Une autre notion importante pour la gestion des propriétés est la définition pour chacune d'elles de leur domaine et de leur portée.

Le domaine permet de déterminer à quelles classes de sujets une propriété peut être ajoutée. La portée détermine à quelles classes l'objet doit appartenir.

Dans l'exemple suivant, il s'agit du domaine et de la portée de la propriété *isTopping*. Elle s'applique à un sujet du type *PizzaTopping* et, a comme valeur, un objet du type *Pizza*. Si ces propriétés ne sont pas respectées dans le modèle, le raisonneur indique des incohérences.

Figure 4 : Définition du domaine et de la portée d'une propriété

4.4 Restrictions

Description

Dans le langage OWL, les propriétés sont utilisées afin de créer des restrictions. Comme son nom l'indique, les restrictions servent à définir si une instance appartient à une classe donnée. En OWL, il existe trois types de restrictions :

- Les restrictions de quantificateurs
- Les restrictions de cardinalité
- Les restrictions de valeur

Les restrictions de quantificateurs s'appliquent à une propriété OWL d'une classe. Les deux propriétés du langage OWL suivantes sont utilisées :

- *owl:someValuesFrom* qui représente le quantificateur existentiel qui signifie «il existe au moins un ». Par exemple, pour dire qu'une *Pizza* doit posséder une *PizzaBase*, on utilise la restriction « *hasBase owl:someValuesFrom PizzaBase* »
- owl:allValuesFrom est le quantificateur universel. Sa signification est « pour tout ». La restriction « *hasBase owl:allValuesFrom ThinAndCrispyBase* » signifie que toutes les valeurs de *hasBase* sont du type *ThinAndCrispyBase*.

Dans Protégé-OWL, ces restrictions sont représentées par les symboles de quantification.

Les restrictions de cardinalité *owl:minCardinality*, *owl:maxCardinality* et *owl:cardinality* permettent de définir la cardinalité d'une propriété. Protégé-OWL utilise respectivement les symboles mathématiques <, > et = pour représenter ces propriétés.

Les restrictions de valeur précisent la valeur que doit avoir une propriété de la classe. En OWL, elles sont représentées par la propriété *owl:hasValue*.

Catégories de restrictions

Les restrictions peuvent êtres classées dans trois catégories distinctes :

- Les restrictions héritées sont, comme leur nom l'indique, héritées des classes parentes.

- Les restrictions nécessaires déterminent que tous les objets appartenant à une classe donnée possèdent les propriétés définies.
- Les restrictions nécessaires et suffisantes permettent d'affirmer qu'un objet correspondant aux restrictions définies appartient forcément à la classe.

Interfaces

L'interface de Progété-OWL répertorie les différentes restrictions par catégories et par types. Elle dispose également d'un éditeur de restrictions offrant toutes les options citées ci-dessus.

Figure 5 : Restrictions d'une classe

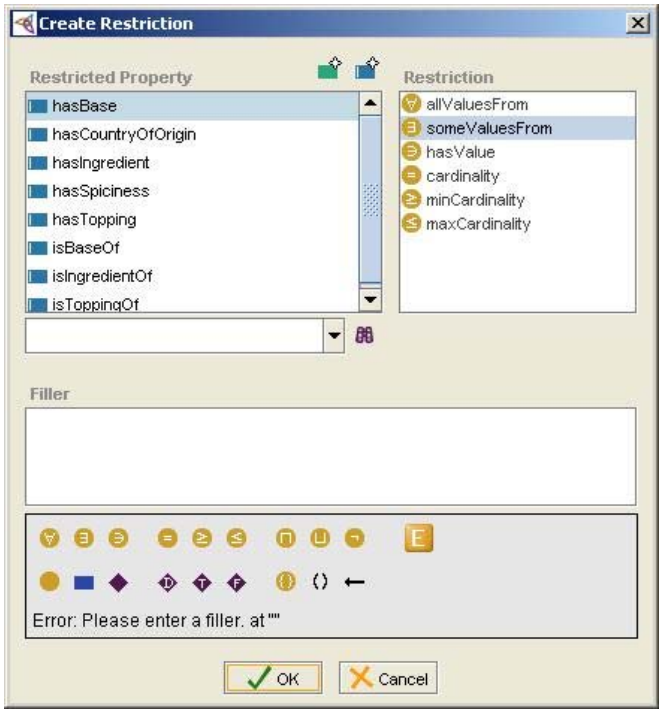


Figure 6 : Editeur de restriction de Protégé-OWL

4.5 Utilisation d'un raisonneur

Introduction

Un raisonneur est un algorithme qui parcourt un schéma d'ontologies et analyse sont contenu. Il est utilisé dans deux cas précis :

- Pour vérifier la validité du schéma
- Pour créer des inférences

La seule condition lors du choix du raisonneur est qu'il comprenne le langage OWL et puisse créer les inférences grâce à ces règles.

Portégé-OWL permet d'utiliser une interface DIG. Les raisonneurs Racer et Pellet sont supportés. Dans l'exemple suivant, les tests ont été effectués avec Pellet.

Validation du schéma

Le test de validation d'un schéma permet de vérifier s'il n'existe pas d'informations inconsistantes.

Une information inconsistante peut, par exemple, être une restriction qui stipule qu'une classe doit hériter de deux classes disjointes. Cette information est

impossible à réaliser en respectant les règles établies par le schéma d'ontologies. Le raisonneur indiquera donc une incohérence (Voir Classes disjointes).

Avec Protégé-OWL, la première chose à faire, une fois le raisonneur installé, est de préciser l'adresse et le port de connexion à l'interface DIG. Pour Pellet, il s'agit par défaut de localhost:8081 (8080 pour Racer) lors d'une installation en local. Ensuite, il faut sélectionner l'option *check consistency* dans le menu OWL.

Le raisonneur retourne le résultat en indiquant quelles sont les classes inconsistantes.

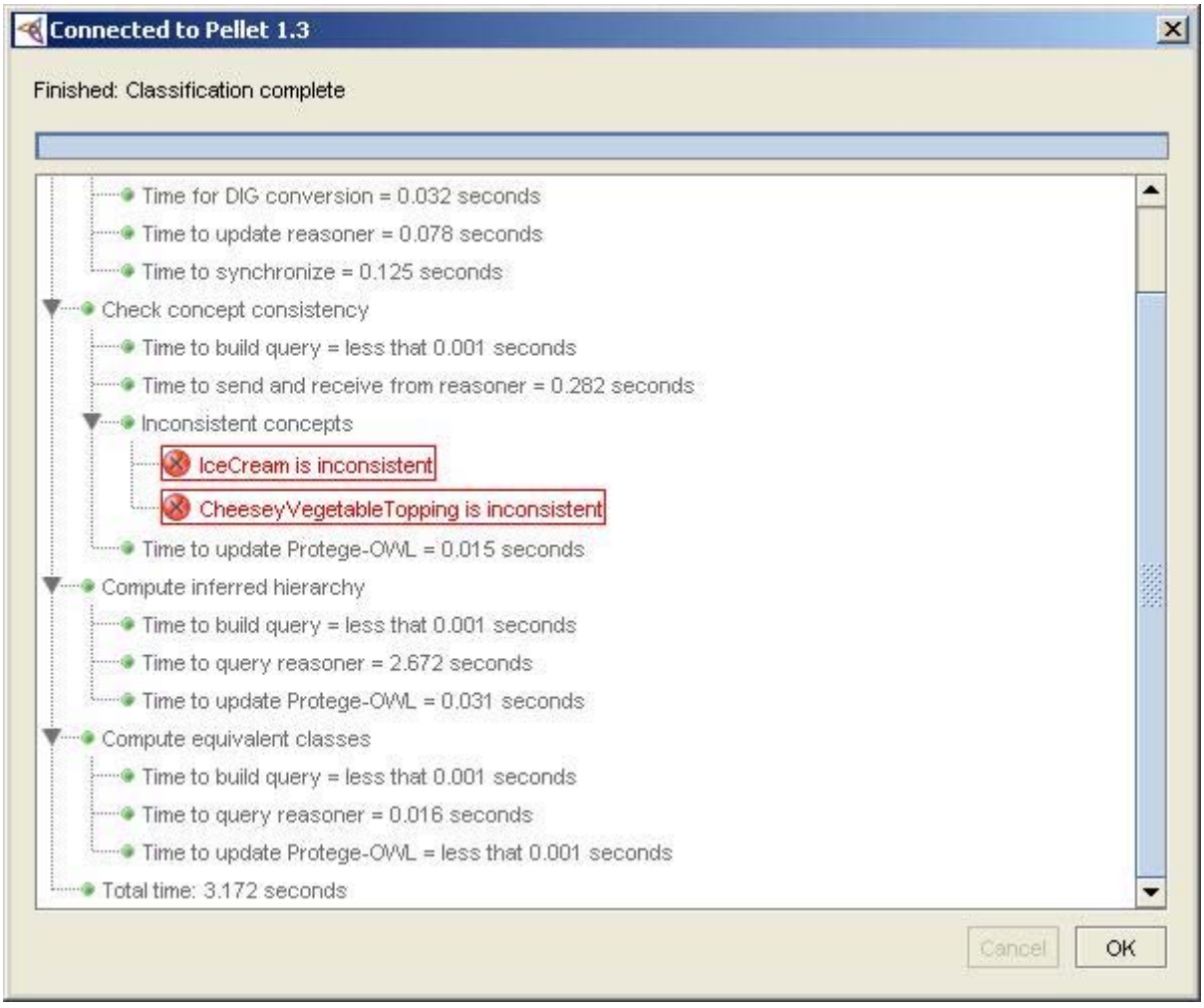


Figure 7 : Analyse par un raisonneur (ici Pellet)

Inférences

La seconde fonctionnalité du raisonneur est de créer un modèle de données inférées en se basant sur le modèle de base et des règles et restrictions définies par l'ontologie.

En utilisant l'option *classify taxonomy dans le menu OWL,* Protégé-OWL affiche alors le modèle inféré. Les nouvelles propriétés déduites par le raisonneur sont affichées en bleu. Ce procédé permet d'enrichir le schéma d'ontologies de base. On remarque, sur l'exemple ci-dessous, que de nombreuses nouvelles classes ont été ajoutées à la classe *SpicyPizza* en se basant sur ses restrictions.

A noter qu'il est possible d'inférer un modèle inconsistant. Cependant, lors d'une requête, par exemple en SPARQL avec l'API Jena, une inconsistance risque de retourner des erreurs.

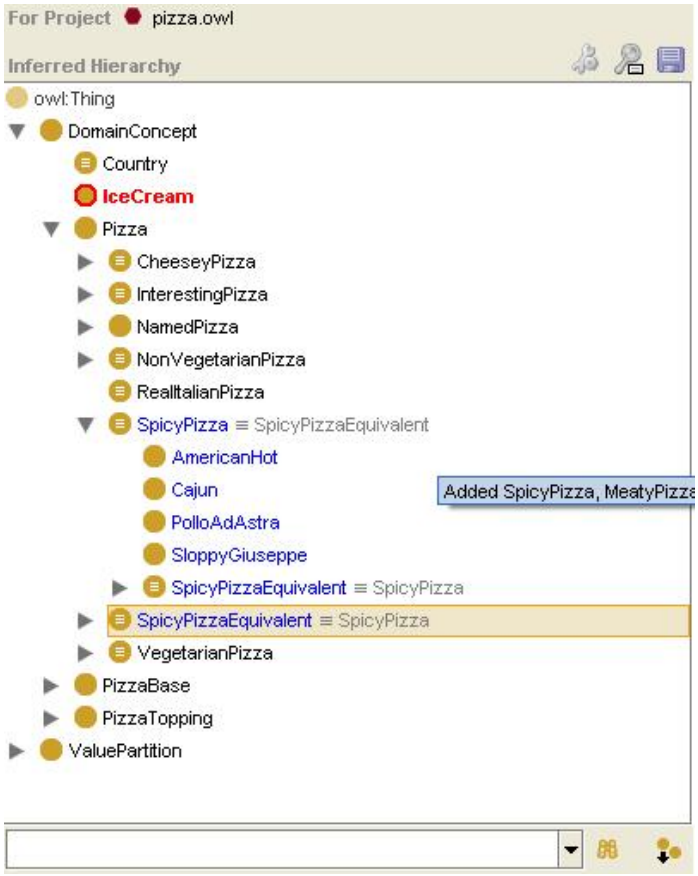


Figure 8 : Explorateur de classes du modèle inféré

4.6 Importation d'ontologies

Un des grands principes du Web sémantique est de permettre de rassembler différentes données provenant du Web. Pour cela, il existe certaines ontologies à importer.

Dans l'exemple suivant, on importe dans *notre* schéma l'ontologie foaf qui représente les concepts liés aux personnes. Protégé-OWL possède une interface permettant de gérer les différentes ontologies du modèle. En précisant l'Url de foaf, il est possible d'ajouter toutes les classes, les propriétés et les restrictions provenant de cette ontologie.

L'avantage d'effectuer une telle importation est de pouvoir, par la suite, créer des liens entre nos données et d'autres données du Web sémantique utilisant également l'ontologie foaf.

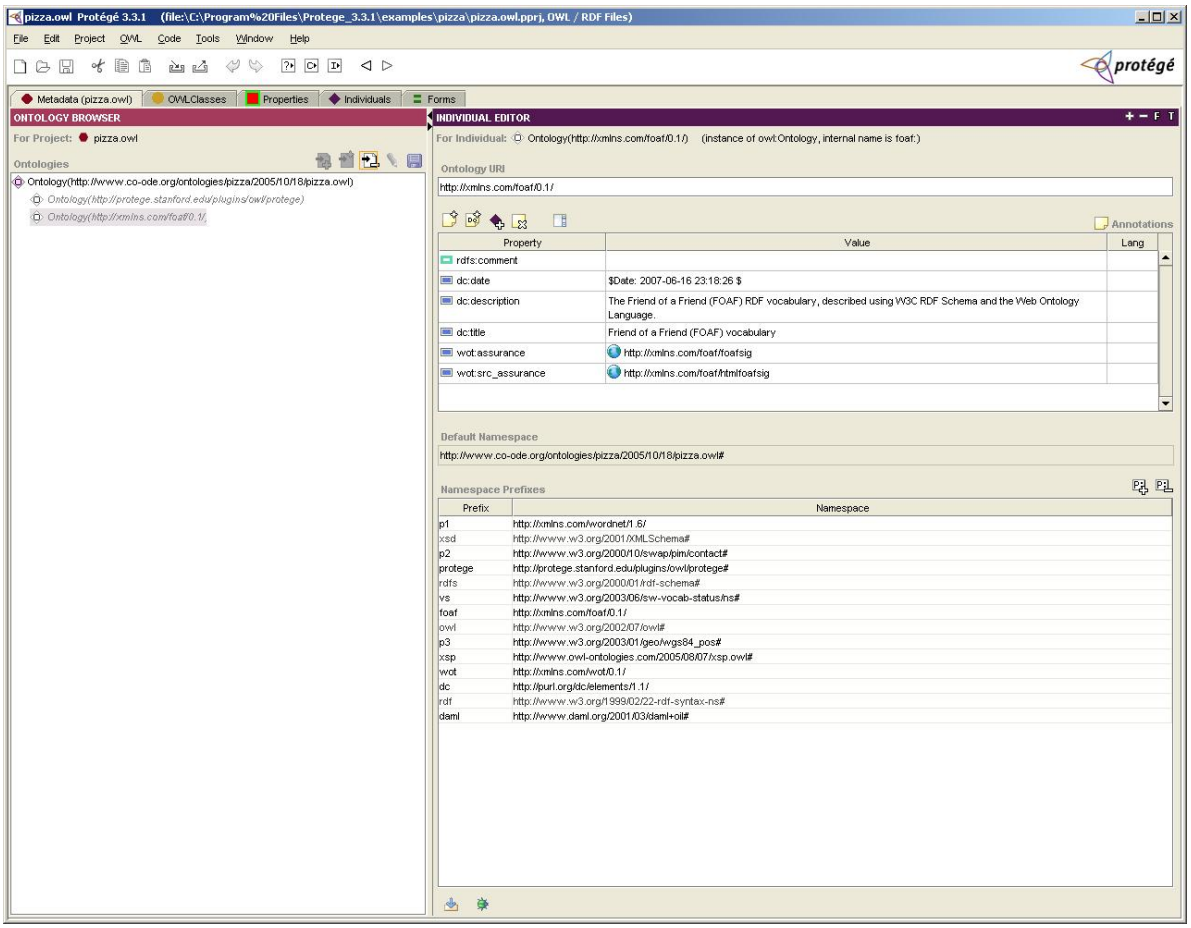


Figure 9 : Interface de gestion des ontologies

Une fois l'intégration de la nouvelle ontologie effectuée, l'interface de gestion des classes contient à présent la liste des classes importées. Il est possible, de la même façon que pour l'ontologie de base, d'ajouter des classes, de modifier la hiérarchie et de créer des restrictions en utilisant les nouveaux éléments fournis.

Ici, par exemple, il est possible d'ajouter à la classe *Person* de foaf une propriété *likePizza* ayant pour portée un objet de type *Pizza* de notre ontologie. On pourra alors effectuer des recherches sur les deux ontologies comme par exemple, savoir où habitent les personnes qui aiment les *SpicyPizza*.

Le Web sémantique permet alors, grâce à la structure qu'il propose, de lier toutes les données du Web entre elles.

Figure 10 : Classes importées de l'ontologie foaf

4.7 Références

- Site officiel de Protégé
- Site du raisonneur Racer
- Site du raisonneur Pellet
- Ontologie Foaf
- Tutorial Protégé-OWL

5. ANALYSE DES OUTILS

5.1 Introduction

Ce chapitre correspond à la phase d'analyse des outils disponibles pour la gestion des données sémantiques. Toutes les solutions détaillées appartiennent à des catégories différentes. Cependant, chacune d'entre elles apporte des fonctionnalités utilisables pour la suite de ce travail. Les outils présentés dans ce chapitre sont les suivants :

- Jena. Un framework sémantique Java possédant un riche panel de fonctionnalités et une importante communauté sur le Web.
- Mulgara. Un système de stockage de données sémantiques dans un Triples Store.
- HSQLDB. Un système de bases de données embarquées.
- SDB. Un composant pour Jena permettant de gérer efficacement les Triples Store et les Named graphs.
- JRDF. Un framework compatible avec Mulgara.

Les différentes analyses sont accompagnées d'exemples de code pour chaque solution. Tous ces outils ont été implémentés lors de cette phase d'analyse.

5.2 Jena

Présentation

Jena est un Framework Java destiné au développement d'applications du Web sémantique. Il offre des outils de programmation pour la gestion des standards RDF, RDFS, OWL et SPARQL notamment et inclut un moteur d'inférences basé sur des règles de déductions. Développé par le «HP Labs Semantic Web Programme », Jena est un projet Open Source. Les principales fonctionnalités offertes par Jena sont :

- Une API pour le standard RDF
- Une API pour le standard OWL
- La lecture et l'écriture de données RDF dans les formats RDF/XML, N3 et N-Triples
- Le stockage des ontologies en mémoire ou persistant

- Un moteur de requête SPARQL
- Un moteur d'inférences

Installation

Jena est une librairie Java. Pour l'utiliser dans un projet, il est nécessaire de rajouter la librairie jena.jar dans le fichier .classpath du projet. Les logiciels de développement permettent d'effectuer l'ajout automatiquement. Sous Eclipse, il suffit d'aller dans les propriétés du projet sous Java Build Path et d'ajouter le fichier JAR.

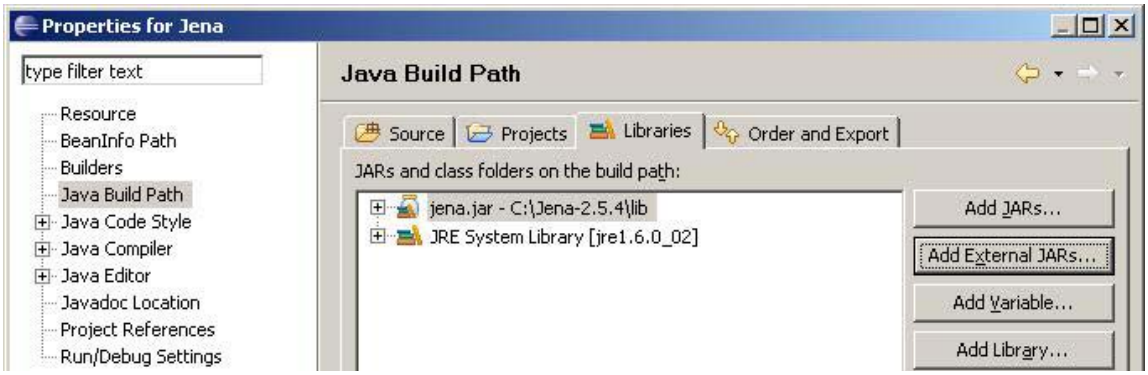


Figure 11 : Ajout de la libraire Jena dans un projet Java sous Eclipse

Utilisation

Création de graphes

Jena possède des classes pour la gestion des graphes, des ressources, des propriétés et des littéraux. Pour les graphes, la classe Jena correspondante s'appelle *Model*. Pour les autres objets, les classes sont respectivement *Resource*, *Proprety* et *Litteral.* L'exemple suivant représente la création du graphe et l'ajout d'un triplet.

La première étape consiste à créer un nouveau graphe. Il s'agit ici d'un graphe stocké en mémoire. La création de graphes persistants sera abordée plus loin.

```
Model model = ModelFactory.createDefaultModel();
```

Puis il faut instancier une nouvelle ressource qui sera référencée par une URI distincte.

```
Resource johnSmith = model.createResource("http://JohnSmith");
```

Enfin la propriété est ajoutée à la ressource en spécifiant le littéral.

```
Proprety fullNameProprety  = model.createProprety("http://mypropreties#fullname");
johnSmith.addProperty(fullNameProprety, "John Smith");
```

Pour gérer les données du graphe, Jena propose une classe *Statement* qui correspond à un triplet. Chaque triplet est constitué d'un sujet, d'un prédicat et d'un objet. Pour parcourir le graphe, il faut implémenter un objet *StmtIterator* qui contient tous les triplets.

```
StmtIterator iter = model.listStatements();
while (iter.hasNext())
{
  System.out.println(iter.nextStatement().toString());
}
```

Le programme affiche le résultat suivant :

[http://somewhere/JohnSmith, http://mypropreties#fullname, "John Smith"]

Ecriture de fichiers RDF

Pour sauvegarder le graphe sous forme de fichier RDF, la classe *Model* de Jena propose une méthode *write*. Le graphe est automatiquement transformé selon les normes W3C du standard RDF.

Avant de sauvegarder, il est préférable de définir un préfixe pour les nouvelles propriétés qui ont été créées. Dans l'exemple précédent, la propriété http://mypropreties#fullname ne possède pas de préfixe défini. Grâce au code suivant, il est possible d'inclure au graphe les préfixes désirés.

```
model.setNsPrefix( "prop", "http://mypropreties# ");
```

Le graphe est maintenant prêt à être généré. Jena propose deux standards d'écriture de fichiers : RDF/XML et N-TRIPLE. Pour préciser le standard à utiliser, il suffit d'ajouter à la méthode *write* le type du standard.

```
Model.write(new FileWriter(new File("data.rdf")), "RDF/XML-ABBREV") ;
```

Le fichier ainsi obtenu est conforme au standard RDF et est ainsi utilisable par n'importe quelle application du web sémantique.

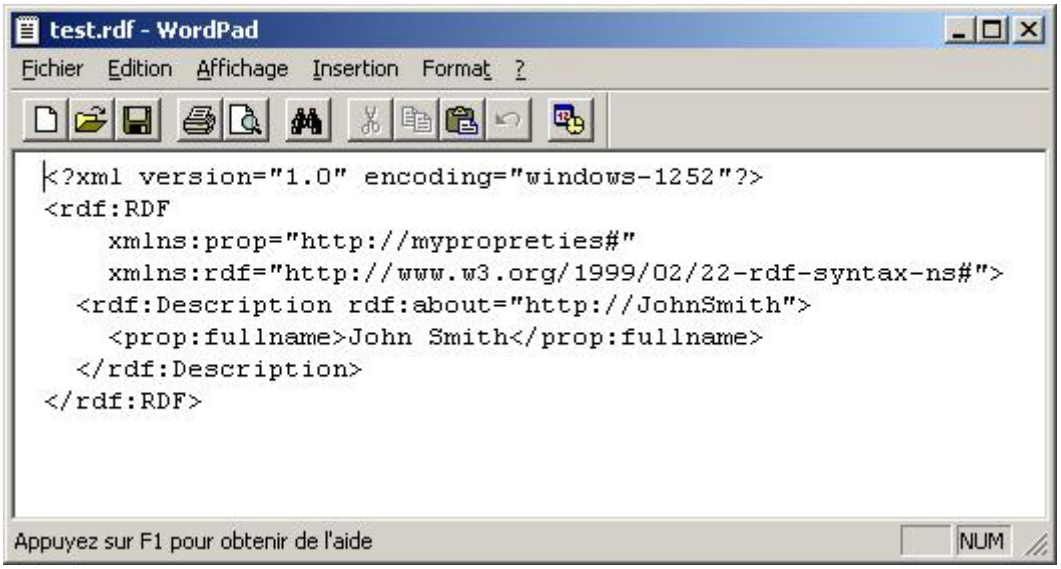


Figure 12 : Fichier RDF généré par Jena

Lecture de données RDF

Pour lire un fichier RDF, il faut dans un premier temps créer le flux d'entrée. Si le fichier est stocké en local, Jena offre un objet *FileManager* qui permet de créer le flux de lecture du fichier grâce à la méthode suivante :

```
InputStream in = FileManager.get().open( "file.rdf" );
```

Si le fichier se trouve sur le web, la solution est d'utiliser l'objet *URL* de Java pour ouvrir le flux de la façon suivante.

```
InputStream in = new URL("http://domain.com/file.rdf").openStream() ;
```

Une fois le flux instancié, il faut utiliser la méthode *read* de l'objet *Model*. Le premier argument de la méthode est le flux d'entrée, le second correspond à l'URI qui va être utilisée dans le cas où le fichier contient des URI relatives. Jena retourne un avertissement si les données RDF contiennent des données relatives. Il est nécessaire d'utiliser des URI absolues pour le stockage des données dans une DB ou sur un fichier.

Syntaxe de la méthode de lecture :

```
model.read(in,"http://uri");
```

Une autre solution que propose Jena est l'utilisation de l'objet *FileManager* et de sa méthode *readModel*. Il est possible de passer comme argument le chemin du fichier ainsi que les protocoles à utiliser comme file et http. Cependant, le flux est généré automatiquement ce qui rend cette méthode moins personnalisable.

```
FileManager.get().readmodel(model, "file://c:/file.rdf" ) ;
```

Opérations sur les graphes

Jena propose trois types d'opérations sur les graphes :

- Union
- Intersection
- Différence

Ces opérations permettent de gérer les informations de plusieurs graphes simultanément. Elles sont extrêmement utiles lors de l'utilisation des Named Graphs avec SDB par exemple.

La syntaxe est la suivante :

```
model1.union(model2) ;
```

Requêtes SPARQL

Jena utilise le composant ARQ qui interprète les requêtes SPARQL. ARQ possède une API intégrée à Jena qui permet d'effectuer les requêtes et de gérer les résultats. Les différents objets fournis par l'API de ARQ sont les suivants :

- *Query* est une classe qui représente la requête à effectuer. C'est en fait un container qui contient tous les détails pour effectuer une requête SPARQL. Les objets *Query* sont habituellement créés à partir de la *QueryFactory* qui contient les différents outils de parsing.
- *QueryExecution* est l'objet qui exécute la requête.
- *QueryExecutionFactory* crée les instances de *QueryExecution*. Cet objet peut avoir comme argument un objet *Query* ou un *String* représentant la requête.
- Pour les requêtes de sélection :
  - *QuerySolution* représente un élément du résultat.
  - *ResultSet* est composé de l'ensemble des *QuerySolution*

- *ResultSetFormatter* transforme un *ResultSet* en différents formats. Les formats disponibles sont le format texte, XML et RDF (un objet *Model* pour Jena)

Il existe plusieurs modes pour effectuer une requête, parcourir le résultat obtenu et le formater. Voilà un exemple simple qui sélectionne un élément et le formate en texte.

```
String queryString = "SELECT $s $p $o WHERE {<http://JohnSmith> $p $o}" ;
Query query = QueryFactory.create(queryString) ;
QueryExecution qexec = QueryExecutionFactory.create(query, model) ;
ResultSet results = qexec.execSelect() ;
ResultSetFormatter.out(System.out, results, query) ;
```

Les résultats peuvent être parcourus avec la méthode *nextSolution* de l'objet *ResultSet* ou formatés grâce aux différentes méthodes fournies par l'objet *ResultSetFormatter.* L'exemple suivant montre comment accéder à la variable p de la requête.

```
QuerySolution  qsol = results.nextSolution();
RDFNode x = qsol.get("p");
if(x.isResource())
  System.out.println(((Resource)x).getURI());
```

```
Query :

SELECT $s $p $o WHERE {<http://JohnSmith> $p $o}


Resultat de RDFNode :

http://mypropreties#fullname


Resultat du ResultSetFormatter :

-----------------------------------------------------
| s | p                                 | o           |
=====================================================
|   | <http://mypropreties#fullname> | "John Smith" |
-----------------------------------------------------
```

Figure 13 : Résultat d'un requête SPARQL avec l'API Jena

Utilisation de bases de données

Jena permet de créer des graphes permanents en utilisant une base de données. Les bases de données supportées sont SQLServer, Oracle, Mysql, HSQL, PostgreSQL et Derby.

L'API permet de se connecter à la base de données et de créer un objet *Model* persistant. Jena utilise son propre layout destiné à optimiser les temps d'insertion et de lecture des données. Le code suivant permet de se connecter à une base de données HSQLDB.

```
String className =          "org.hsqldb.jdbcDriver";
Class.forName (className);
String DB_URL =            "jdbc:hsqldb:file:filename";
String DB_USER =           "sa";
String DB_PASSWD =         "";
String DB =                "HSQL";

IDBConnection conn = new DBConnection (DB_URL, DB_USER, DB_PASSWD,DB);
ModelMaker maker = ModelFactory.createModelRDBMaker(conn) ;

Model model = maker.createDefaultModel();

conn.close();
```

Graphe d'ontologies

En parallèle au modèle classique de Jena, il est possible de créer un graphe d'ontologies. Ces modèles comportent, en plus des données RDF, des schémas d'ontologies qui permettent de structurer les données en classes et de définir les relations entre elles. Pour cela, il existe différents standards de description des ontologies. Jena supporte les standards suivants :

- RDFS
- OWL
- DAML+OIL

Pour créer un graphe d'ontologies persistant, il faut tout d'abord créer un graphe de base persistant. Pour cela, un objet *ModelMaker* est instancié. Cet objet vas servir à créer le *OntModel* correspondant au *Model.*

```
Model base = maker.createModel( "ontology.owl", false );
OntModel m = ModelFactory.createOntologyModel( getModelSpec( maker ), base );
m.read("ontology.owl");
```

Inférences

Introduction

Jena offre une API complète capable d'effectuer des inférences sur les différents graphes de données. Pour effectuer ces inférences, il est nécessaire d'utiliser les deux objets suivants :

- *Reasoner* permet de paramétrer le type de données à inférer, de déterminer les schémas à utiliser et de créer le modèle d'inférence *InfModel*.
- *InfModel* est le modèle inféré. Pour sa création, il est nécessaire d'avoir un *Reasoner, Model* de base et facultativement un ou plusieurs *Model* d'ontologies.

Raisonneur

Le raisonneur a pour but de déduire de nouvelles données, appelées données inférées, en se basant sur différents formats de règles. Il existe un type de raisonneur pour chaque format héritant de l'interface *Reasoner*. Ces raisonneurs sont les suivants :

- *OWLReasoner*
- *RDFSreasoner*
- *GenericRuleReasoner*
- *DIGReasoner*

Jena propose deux modes de construction des *Model* d'inférences. La première solution se base sur l'objet *Reasoner*. Une fois celui-ci instancié selon le type de raisonneur nécessaire, il faut lui ajouter les schémas d'ontologies sur lesquels il se base pour effectuer les déductions. La méthode *bindSchema* permet d'ajouter l'objet *Model* contenant les ontologies au raisonneur. Puis, grâce à l'objet *ModelFactory*, il est possible de générer une instance d'*InfModel* contenant toutes les données inférées.

```
Reasoner reasoner = ReasonerRegistry.getOWLReasoner();
reasoner = reasoner.bindSchema(schema);
InfModel infmodel = ModelFactory.createInfModel(reasoner, data);
```

La seconde solution est d'utiliser les graphes d'ontologies présentés au chapitre précédent. En utilisant cette méthode, il est possible de se passer de l'objet *Reasoner* en passant directement le type d'inférences que l'on désire dans les spécifications lors de la génération d'un *OntModel*.

Dans l'exemple suivant, un graphe d'inférences est créé en utilisant le raisonneur OWL/Micro de Jena. Une fois l'*OntModel* instancié, l'ajout de nouvelles données entraînera une régénération automatique des inférences.

```
OntModel om =
ModelFactory.createOntologyModel(OntModelSpec.OWL_MEM_MICRO_RULE_INF, model);
```

Graphe d'inférences

Les graphes d'inférences, que ce soit des *InfModel* ou des *OntModel*[3], générés avec l'un des processus décrits ci-dessus, contiennent l'ensemble des données de base auxquelles s'ajoutent les nouvelles données inférées. Il est possible de récupérer les données inférées en utilisant la méthode suivante :

```
Model deductionModel = infModel.getDeductionsModel() ;
```

S'il est nécessaire d'inférer à nouveau l'un de ces graphes, sans pour autant devoir redéfinir toutes les caractéristiques du raisonneur, l'objet *InfModel* possède une méthode *rebind* effectuant cette opération.

Références

- Home page du projet Jena
- PDF d'introduction à Jena
- Tutorial
- Requête SPARQL avec Jena

5.3 SDB

Présentation

SDB est un composant Java pour Jena destiné au stockage d'informations RDF. Il comporte un interpréteur de requête SPARQL basé sur le moteur ARQ de Jena et permet de transformer ces requêtes en SQL Standard.

SDB est compatible avec un grand nombre de bases de données. Les développeurs garantissent le fonctionnement avec les bases de données Oracle, MSSQL, PostgreSQL, HSQLDB, MySQL et Apache Derby. Il est malgré tout

[3] OntModel hérite de InfModel

possible de configurer SDB pour n'importe quelle autre base de données, toutefois sans garantie de compatibilité.

Le principal avantage, par rapport au stockage de données de l'API Jena, est un gros gain de performances grâce à des algorithmes gérant les triplets de manière optimisée.

Installation

Une fois la base de données choisie et installée correctement, il est nécessaire de créer un fichier de configuration TTL contenant les informations de connexion à la base de données. Puis, avec la commande suivante, le serveur peut être démarré (avec le nom du fichier TTL correspondant).

```
bin/sdbconfig --sdb=sdb.ttl –create
```

Dans le cas d'une utilisation embarquée dans une application Java, il faut ajouter la libraire sdb.jar dans le classpath. Cette librairie se basant sur les classes de Jena, il est possible d'intégrer SDB à une solution Jena existante.

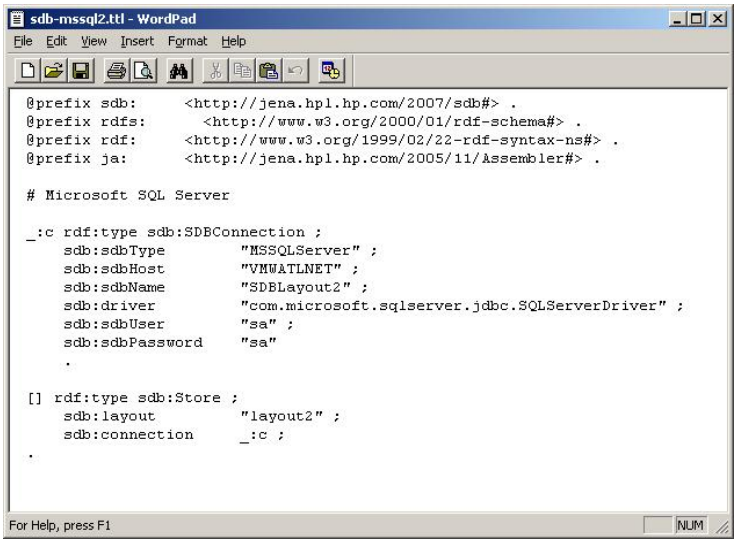


Figure 14 : Fichier de configuration TTL pour MSSQL

Utilisation

Schémas

SDB propose deux schémas de stockage différents.

Le layout1 est le schéma de base de Jena. Il comporte deux tables. Une table contient l'ensemble des triplets (table Triples) et l'autre, les préfixes utilisés dans les données RDF intégrées (table Prefixes). La table Triples contient les triplets « sujet-prédicat-objet » sous forme de chaîne de caractères.

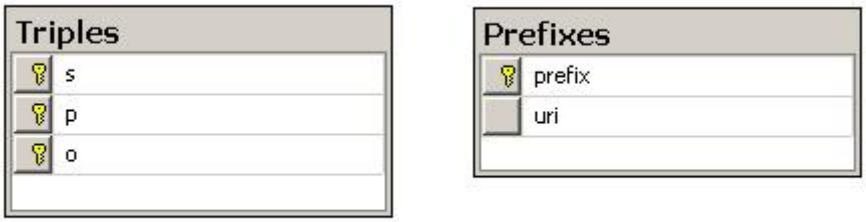


Figure 15 : Diagramme du layout1

Ce schéma est le moins performant mais il est compatible avec toutes les bases de données créées par Jena. Il est donc plus flexible.

Le layout2 est propre à SDB. Il est utilisé par les algorithmes pour accroître les performances. Cependant, il ne peut pas être utilisé avec des données déjà existantes. En plus des tables Triples et Prefixes, il est composé de deux tables supplémentaires :

- La table Quads qui correspond à la table Triples du layout1 avec un champ supplémentaire pour la gestion des Named Graphs.
- La table Nodes qui décrit en détails chaque élément.

Les tables Triples et Quads contiennent ici les références sur la table Nodes. Le moteur travaille ainsi en grande partie avec des références ce qui implique un gain de performances important.

Il existe deux modes pour ce schéma :

- Le mode hash où les références se font sur le champ hash (Int64) de la table Nodes.
- Le mode index où SDB ajoute un champ id dans la table Nodes pour référencer les éléments.

A noter qu'aucun des deux schémas n'utilise de clefs étrangères. Les références sont gérées directement par le composant SDB.

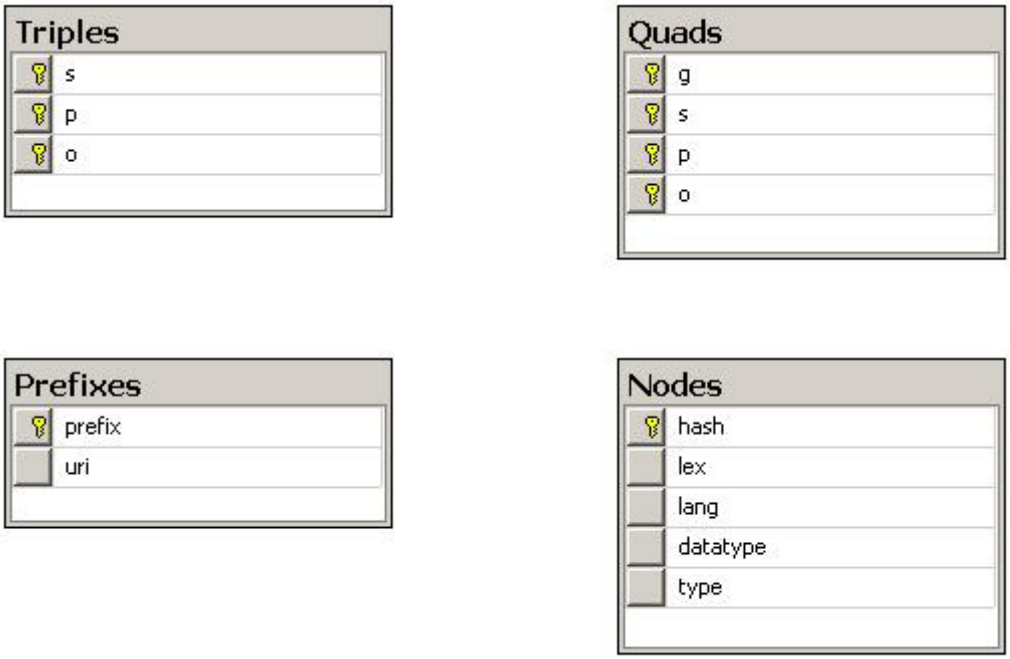


Figure 16 : Diagramme du layout2 en mode hash

Connexion

Comme il a été précisé auparavant, le composant SDB se base sur les classes de Jena afin de permettre une intégration facile.

La première étape est de créer un objet *Store* qui contient toutes les informations afin de permettre à l'application de se connecter à la base de données, de charger et de stocker des informations RDF. Il existe deux façons de créer un Store

- En lisant un fichier de configuration TTL grâce à l'objet *SDBFactory*.

```
Store store = SDBFactory.connectStore("sdb.ttl") ;
```

- Directement dans le code Java.

```
StoreDesc storeDesc = new StoreDesc(LayoutType.LayoutTripleNodesHash,
                                    DatabaseType.Derby) ;
JDBC.loadDriverDerby() ;
String jdbcURL = "jdbc:derby:DB/SDB2";
SDBConnection conn = new SDBConnection(jdbcURL, null, null) ;
```

```
Store store = SDBFactory.connectStore(conn, storeDesc) ;
```

Il est préférable d'utiliser les fichiers de configuration par soucis de flexibilité.

Query

SDB utilise le moteur ARC. L'interrogation de la base de données via des requêtes SPARQL est similaire à l'interrogation avec ARC. SDB transforme la requête SPARQL en requête SQL. Le code suivant décrit la façon d'interroger en Java une base de données grâce au composant SDB. L'objet *Store* créé précédemment contient toutes les informations nécessaires.

```
Dataset ds = DatasetStore.create(store) ;
QueryExecution qe = QueryExecutionFactory.create(query, ds) ;
try {
   ResultSet rs = qe.execSelect() ;
   ResultSetFormatter.out(rs) ;
} finally { qe.close() ; }
```

Utilisation du Model Jena

Il est possible de créer un *Model* Jena depuis un objet *Store* de SDB. La base de données contient un modèle par défaut dans la table Triples. Ce graphe est accessible avec le code suivant :

```
Store store = StoreFactory.create("sdb.ttl") ;
Model model = SDBFactory.connectDefaultModel(store) ;

StmtIterator sIter = model.listStatements() ;
while(sIter.hasNext())
{
   Statement stmt = sIter.nextStatement() ;
   System.out.println(stmt) ;
}
sIter.close() ;
store.close() ;
```

Pour accéder aux Named graphs, il suffit de remplacer la méthode *connectDefaultMode(Store store)* par la méthode *connectNamedModel(Store store , String iri)* où *iri* représente le nom du graphe.

Références

- HomePage
- Lien Wikipedia
- Article d'un développeur de SDB

5.4 HSQLDB

Présentation

HSQLDB est un moteur de base de données écrit en Java. Il possède un driver JDBC (qui permet la connexion d'un programme java à la base de données) et supporte un ensemble complet de requêtes SQL.

Fonctionnement

HSQLDB comporte un moteur de base de données léger (moins de 100k) et rapide qui permet de gérer des données autant en mémoire que sur un disque. De plus il est possible de l'utiliser comme librairie dans une application Java sans avoir à installer et à démarrer un serveur. Cette solution embarquée permet une flexibilité importante de l'application.

Il est également possible de faire marcher HSQLDB en temps que serveur distant. Il existe différents outils permettant la gestion d'un serveur.

De nombreux projet Open Source utilise HSQLDB pour sa légèreté, sa flexibilité et sa vitesse. La libraire Jena permet l'utilisation d'une base de données HSQLDB pour le stockage des données sémantiques.

Avantages

- Sa légèreté est idéale pour des solutions embarquées
- Compatibilité avec Jena
- Capable de gérer autant des données en mémoires que sur un disque
- Compatible avec le standard SQL
- Performant

Performances

Le graphique suivant est un comparatif de performance entre certaines des autres solutions existantes lors d'un test de lecture sur de grandes quantités de données. On peut remarquer que même avec une couche de persistance comme Hibernate, les performances sont intéressantes.

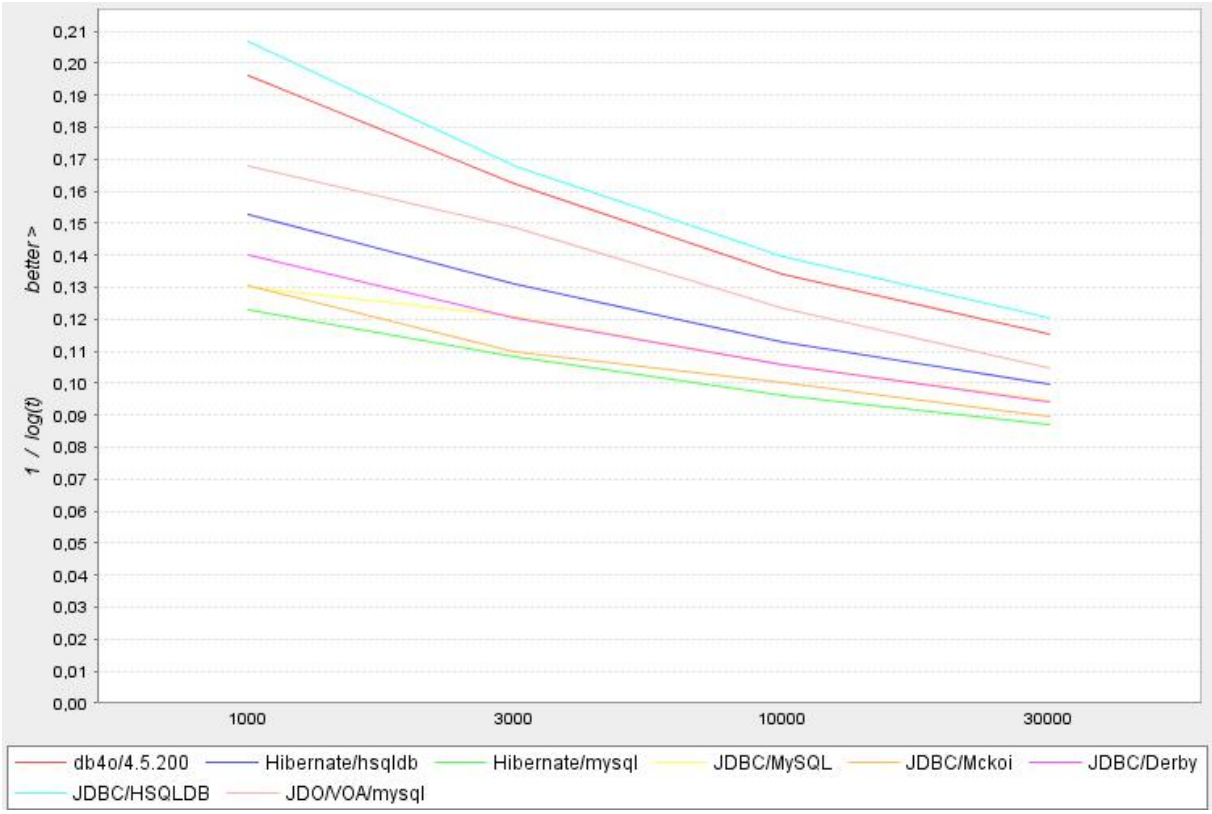


Figure 17 : Graphisme de comparaison des performances

5.5 Mulgara Semantic Store

Présentation

Mulgara est une solution java de stockage de données. Elle utilise le moteur de stockage XA TripleStore qui est un Triples Store transactionnel scalable. Mulgara stocke les métadonnées sous forme de triplets (subject-predicate-object) conformément au standard RDF. Le langage de requête ITQL (Interactive Tucana Query Language) basé sur SQL est utilisé pour interroger le Triples Store.

Cette solution offre de nombreux avantages :

- Utilisation d'un Triples.
- La scalabilité du système permet de gérer efficacement toutes quantités de données.
- Capacité à stocker de nombreux graphes sur le même serveur grâce à l'utilisation de Named Graphs.
- Permet l'utilisation de différents protocoles de communication (RMI, SOAP,…)
- Interface possible avec diverses API (Jena, JRDF)

Cependant, dans la version actuelle de Mulgara, le langage de requête SPARQL utilisé généralement dans le mode du web sémantique n'est pas supporté. Dans la dernière version de Mulgara (1.1.0), l'interfaçage avec Jena est obsolète à cause de problèmes de compatibilité.

Installation

Mulgara nécessite l'installation préalable de Java sur la machine (JDK ou JRE). Les systèmes d'exploitation Linux et Windows sont supportés. Il suffit ensuite de lancer le serveur avec la ligne de commande suivante :

```
> java –jar mulgara-1.1.0.jar
```

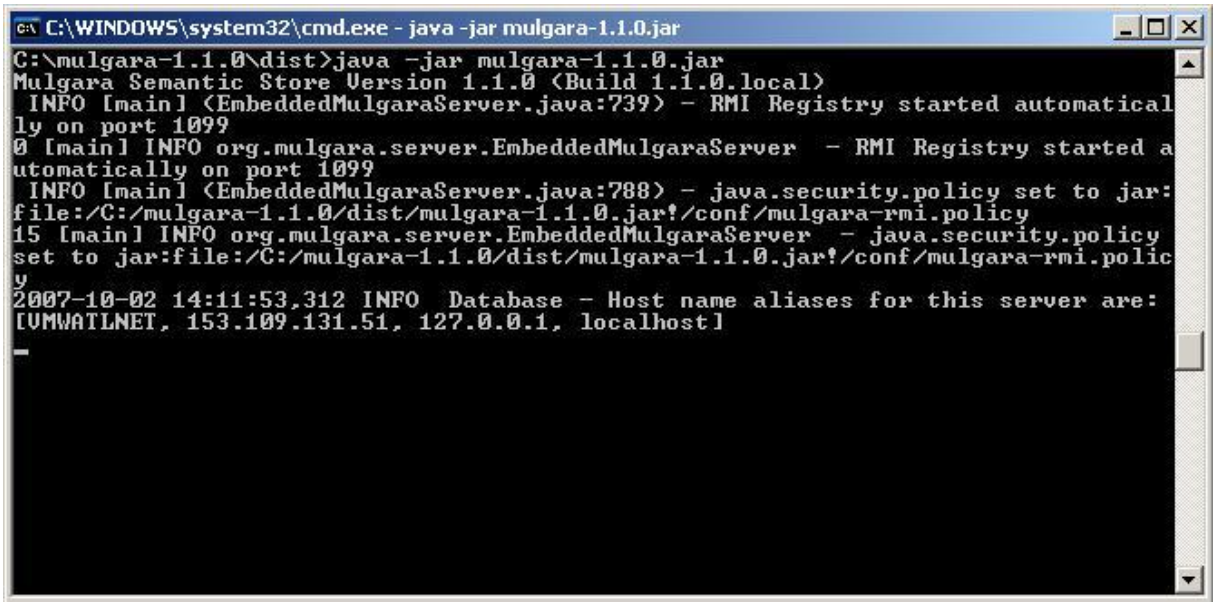


Figure 18 : Console de lancement du serveur Mulgara

Au démarrage, il est possible de spécifier plusieurs paramètres de configuration :

- -n, --normi désactive le démarrage automatique de RMI.
- -x, --shutdown arrête le serveur local.
- -c, --serverconfig utilise des configurations externes.
- -k, --serverhost définit le hostname du serveur (par défaut localhost).
- -o, --httphost définit le hostname du serveur web (par défaut localhost).
- -p, --port définit le port des requêtes HTTP (par défaut 8080).
- -r, --rmiport définit le port des requêtes RMI (par défaut 1099).
- -s, --servername définit le nom du serveur RMI (par défaut server1).
- -a, --path définit le chemin où seront stockées les données.

Utilisation

iTQL Shell

iTQL Shell est une interface console qui permet d'effectuer des requêtes iTQL sur le serveur Mulgara. Il suffit d'exécuter la commande suivante pour lancer l'application. Il existe également une interface web accessible lorsque le serveur est démarré à l'adresse http://server:8080/webui/ (selon l'adresse ip du serveur).

```
> java -jar itql-1.1.0.jar
```

Figure 19 : Console de commande iTQL

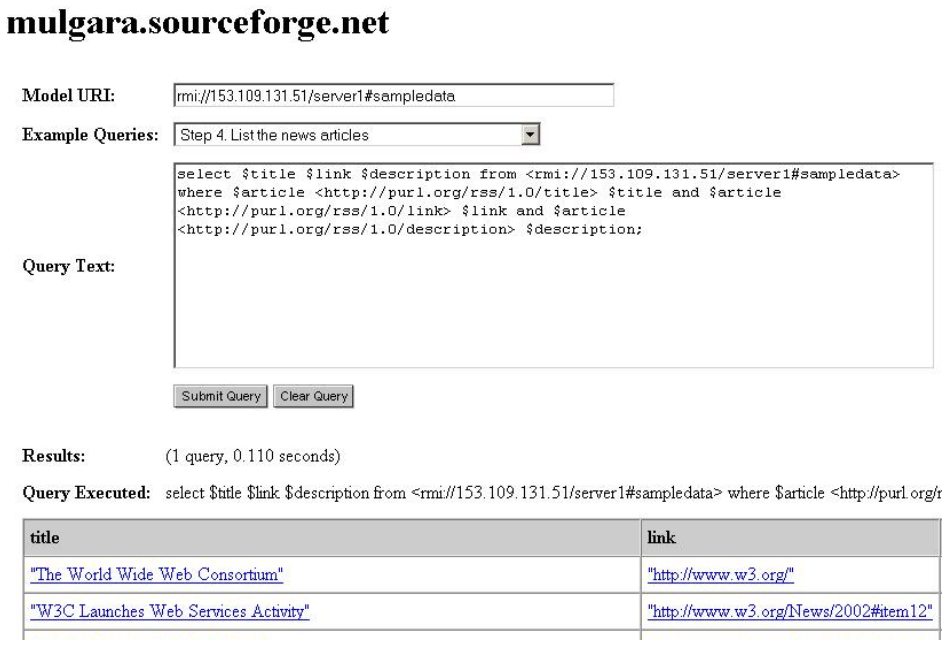


Figure 20 : WebViewer de Mulgara

iTQL Bean

Il est également possible d'accéder au Triples Store directement depuis une application Java en utilisant les librairies fournies par le driver de Mulgara. La requête est en format iTQL. L'adresse du serveur est contenue dans la requête elle-même.

```
ItqlInterpreterBean interpreter = new ItqlInterpreterBean();
String query = "select $s $p $o from <rmi://153.109.131.51/server1#sampledata> where $s $p $o ;";
Answer answer = interpreter.executeQuery(query);
```

SOAP

Mulgara dispose d'un accès WebService SOAP. Bien que moins rapide que les autres méthodes, le protocole SOAP offre une très grande flexibilité au niveau de l'architecture et des langages de programmation utilisés. De plus, se basant sur le protocole HTTP, la communication traverse sans problèmes les sécurités réseaux. Voici un exemple simple d'interfaçage entre une application C# .NET et le WebService SOAP de Mulgara. L'application permet l'envoi d'une requête iTQL et affiche le résultat. Elle utilise le fichier de définition WDSL suivant :

```
http://server:8080/webservices/services/ItqlBeanService?wsdl.
```

Cette application utilise le code suivant pour l'accès au Web Service :

```
WebReference.ItqlInterpreterBeanService service = new
WindowsApplication1.WebReference.ItqlInterpreterBeanService();
 richTextBox1.Text = service.executeQueryToString(textBox1.Text);
```

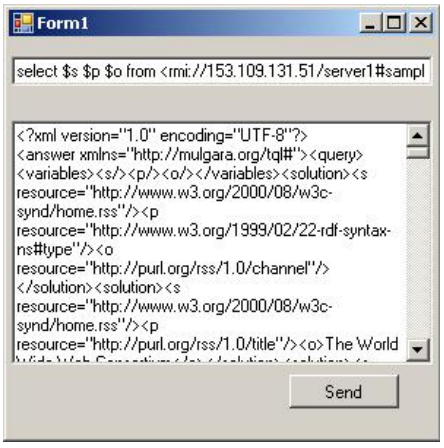


Figure 21 : Application C# utilisant le WebService SOAP de Mulgara

Inférences

L'inférence est un procédé qui permet de déduire de nouvelles déclarations RDF à partir de celles existantes. Dans Mulgara, il est possible d'ajouter des graphes d'ontologies à un graphe de base. Les nouvelles déclarations déduites seront automatiquement ajoutées dans un graphe d'inférences.

Par défaut dans Mulgara il existe trois types de graphes différents :

- Les modèles de base
- Les modèles de schémas
- Les modèles d'inférences

Graphe de base

Il s'agit du graphe utilisé par Mulgara pour stocker les instances RDF.

Schémas

Un modèle de schéma est basé sur RDFS ou sur OWL. Ces schémas définissent un set de règles et indiquent la façon et les moments où celles-ci doivent être appliquées aux déclarations de bases.

Il est possible de déterminer, en configurant Mulgara, quelles sont les données de base à inférer et de sélectionner les ontologies à utiliser. Il est également possible de déterminer à quel moment la création des inférences doit avoir lieu. Si les graphes d'inférences sont automatiquement mis à jour, on parle de *forward chaining*. Si les règles sont appliquées uniquement au moment où une requête est effectuée sur le modèle, on parle de *backward chaining*.

Graphe d'inférence

Le graphe d'inférences continent le résultat de l'exécution des règles définies dans les schémas sur les données contenues dans un ou plusieurs graphes de base. Normalement, le graphe d'inférences n'est pas interrogé directement par l'utilisateur mais est utilisé par Mulgara lors de requête sur le modèle de base.

L'intérêt de séparer les données de base des données inférées permet de modifier le modèle d'inférences à tout moment. Si par exemple, un graphe de base ou une ontologie change, il suffit de remettre à jour le modèle d'inférences sans avoir à modifier les autres données.

Interrogation

ITQL

Il est possible d'interroger les données inférées directement depuis une requête ITQL. Pour cela, il est nécessaire de préciser au serveur Mulgara les préfixes des langages OWL et RDFS avec la commande suivante :

```
alias <http://www.w3.org/2002/07/owl#> as owl;
alias <http://www.w3.org/2000/01/rdf-schema#> as rdfs;
```

Les propriétés de ces langages sont alors accessibles au travers de requête ITQL. Par exemple, pour utiliser la propriété *owl:sameAs* qui permet de définir deux ressources comme égales, il suffit de l'insérer dans une requête de la façon suivante :

```
select $x <owl:sameAs> $y
from <rmi://mysite.com/server1#camera>
where $x <owl:sameAs> $y;
```

Cette requête retournera toutes les ressources équivalentes dans le modèle *camera* pour peu que cette propriété est été définie dans les schémas OWL.

SOFA

SOFA (Simple Ontology Framework API) est une API Java qui permet de gérer un modèle d'inférences. Elle fonctionne indépendamment du système de stockage et peut donc facilement être interfacée avec Mulgara. Voici quelques fonctionnalités que propose SOFA :

- Permet d'effectuer des inférences.
- Supporte les langages RDF, RDFS, OWL et DAML/OIL.
- Permet de créer et d'interroger des schémas d'ontologies.

Références

- Site officiel du projet Mulgara

5.6 JRDF

Présentation

JRDF est une API Java destinée à l'implémentation d'applications utilisant des données RDF. Elle est basée sur plusieurs solutions existantes :

- Jena
- Sesame
- Aquamarine
- Sergey Melnik's RDF API

JRDF est une solution encore récente et est encore assez peu utilisée. Il existe peu de documentations et de tutoriels d'utilisation avec les autres solutions hormis Mulgara. Malgré tout, JRDF offre déjà de nombreuses fonctionnalités pour la création de solution RDF :

- La gestion de graphes de données.
- La création et la manipulation des objets (Statments, Ressources, Nodes, …)
- La possibilité de stocker ces données en mémoire ou sur le disque.
- Gestion du type de données RDF.
- La gestion de données locales et globales.
- Un convertisseur de données RDF et objets Java.
- Un interpréteur de requête SPARQL.

Certaines fonctionnalités essentielles sont encore en cours de développement et seront disponibles dans les versions futures :

- Les transactions.
- La gestion des évènements.
- La gestion de la sécurité.
- L'utilisation de moteurs d'inférences.

Installation

Pour utiliser l'API JRDF, il est nécessaire d'ajouter la librairie jrdf-0.5.0.jar au classpath du projet Java. Toutes les classes seront alors disponibles.

Références

- Homepage
- Documentation technique
- Intégration à Mulgara

6. COMPARATIF DES OUTILS

6.1 Performance

Matériel

Le test de performance a été réalisé avec le matériel suivant :

Hardware

- Intel DualCore 6300 1.86Ghz
- 1Gb de RAM

Software

- Windows 2003 Server
- MSSQL Server 2003
- Java 1.6

La machine de test correspond au type de machine pour lesquelles est destinée le moteur OntoMea.

Conditions

Pour effectuer un test de performances, il est nécessaire d'utiliser une grande quantité de données. Le fichier http://www.snee.com/rdf/sneeair.rdf qui contient 81327 triplets a été utilisé.

Le test a été effectué en deux étapes :

- une étape d'écriture qui consiste à lire le fichier copié préalablement en local et de stocker toutes ses informations dans le système de stockage de la solution.
- Une étape de lecture qui consiste à remonter toutes les informations stockées en mémoire.

Résultats

Le tableau ci-dessous correspond aux résultats moyens obtenus en secondes de chacune des solutions durant les phases de lecture et d'écriture. A noter que les solutions utilisant Jena retournent un graphe de données alors que les solutions

utilisant Mulgara retournent un résultat ITQL (XML). Le graphe de données est une structure bien plus complexe et efficace lors du traitement de ces données. La solution Mulgara SOAP n'a pas réussi à retourner l'ensemble des données et a indiqué une erreur dans le protocole SOAP.

A la vue de ces résultats, les solutions Mulgara RMI et Jena/SDB/HSQLDB/Layout2 sont clairement les plus performantes. Dans un développement futur, si un serveur central serait nécessaire, la solution Jena/SDB/SqlServer/Layout2 est également très performante.

Solution	Lecture (en s)	Ecriture (en s)
Jena/SDB/SqlServer/Layout1	6.5	370
Jena/SDB/SqlServer/Layout2	6.5	25
Jena/SDB/HSQLDB/Layout1	27	66
Jena/SDB/HSQLDB/Layout2	2.2	14.8
Jena/HSQLDB/Layout1	9.6	94.3
Mulgara RMI	1	15.2
Mulgara SOAP		15.4

Figure 22 : Performances des diverses solutions

Graphiques

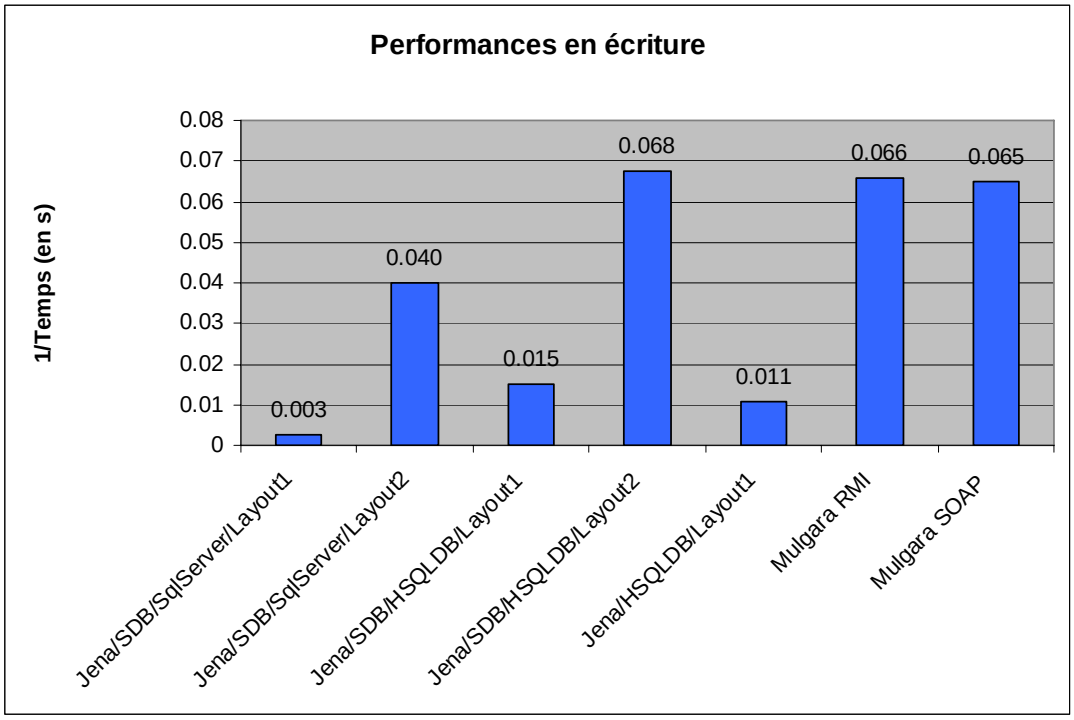


Figure 23 : graphisme de performances en écriture

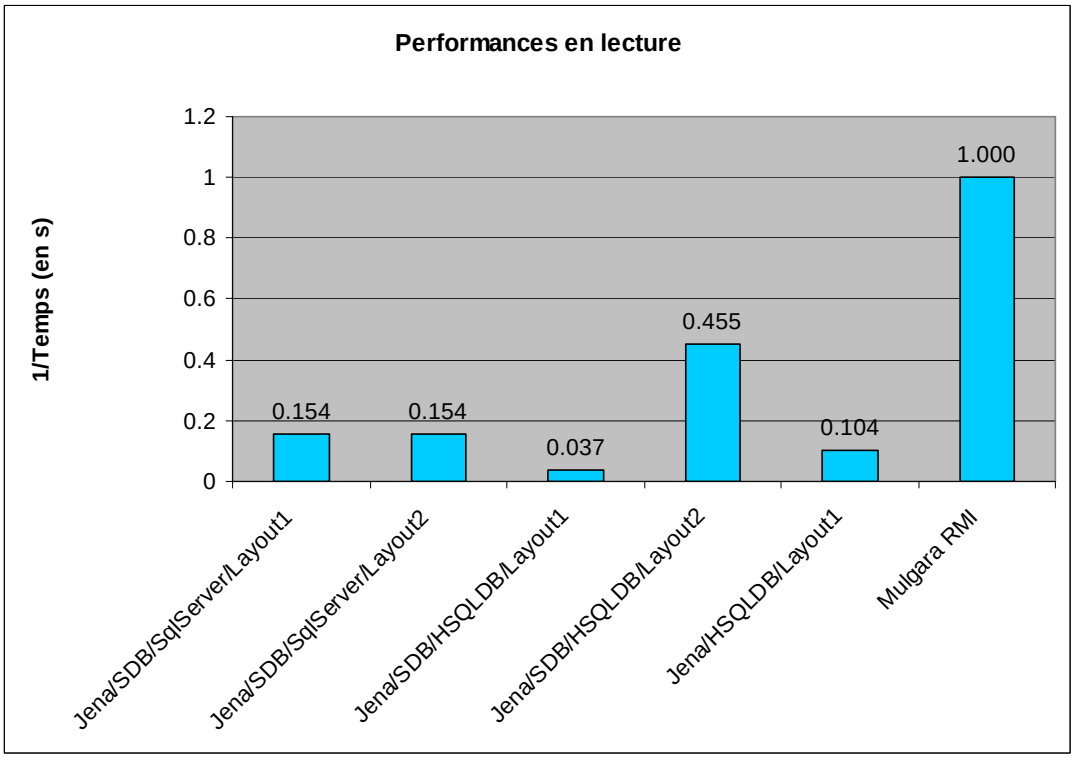


Figure 24: graphisme de performances en lecture

6.2 Fonctionnalités

Tous les outils testés appartiennent à des catégories différentes. Cependant, chacun d'entre eux comporte des fonctionnalités utiles pour la phase de développement. Le tableau ci-dessous résume les avantages de chaque solution dans chacune des catégories suivantes :

- Stockage. Capacité de stocker ou de gérer une base de données.
- Inférences. Possibilités de déduire de nouvelles informations selon le langage utilisé.
- Gestion des données. Modification et manipulation des graphes.
- Requêtes. Type de langage de requêtes utilisé pour interroger les données sémantiques.

	Jena	Mulgara	SDB	JRDF	HSQLDB
Stockage	✔	✔	✔	✔	✔
Serveur DB		✔			✔
DB embarquée					✔
Gestion Triples Store	✔	✔	✔		
Named graphs		✔	✔		
Mémoire	✔			✔	
Inferences	✔	✔			
RDFS	✔	✔			
OWL	✔	✔			
Règles générique	✔				
Interface DIG	✔				
Gestion des données	✔			✔	
Création de graphes	✔			✔	
Modification de ressources	✔			✔	
Requêtes	✔	✔	✔		
SPARQL	✔		✔		
iTQL		✔			

Figure 25 : Fonctionnalités des diverses solutions

6.3 Conclusion

En prenant en compte les résultats des différents comparatifs et la structure du prototype d'OntoMea, la solution Jena/SDB/HSQLDB a été choisie. Les avantages d'un tel choix sont les suivants :

- L'ensemble des composants est compatible en se référant à leur documentation respective.
- Cette solution obtient le deuxième meilleur résultat en performance. En plus, le résultat obtenu est sous forme de graphes.
- Les composants sont complémentaires. En regroupant toutes leurs fonctionnalités, on obtient tous les outils nécessaires pour effectuer le travail de développement.
- La solution initiale du moteur OntoMea utilise déjà une partie de Jena pour la gestion des graphes. Les modifications de la structure sont moins importantes.
- La possibilité, grâce à HSQLDB, de gérer une base de données embarquée.

Ce choix met un terme à la phase d'analyse. Le chapitre suivant détaille l'implémentation des différents éléments dans le moteur OntoMea. Il se base sur les résultats obtenus et sur les tests d'implémentations décrits tout au long de l'analyse des outils.

7. LE MOTEUR ONTOMEA

7.1 Introduction

Dans le projet Memoria-Mea, le moteur sémantique OntoMea est utilisé afin de traiter les données provenant de différents modules. Ces données décrivent différentes informations multimédias numériques concernant un utilisateur. Cela peut être, par exemple, des images, des mails, des coordonnées géographiques ou des points d'intérêts. Son rôle dans le projet global est de traiter les données sémantiques et de fournir un système de déductions performant basé sur les ontologies fournies.

Les différents modules fournissent des données sous différentes formes (XML, RDF, …) qui sont alors formatées et intégrées dans la base de connaissances du moteur OntoMea. Ces informations sont alors disponibles pour les modules de Memroria-Mea grâce aux différents services fournis par un serveur Web embarqué.

Au niveau technique, le moteur OntoMea est développé en Java en se basant sur le framework sémantique Jena. Il est destiné à un usage local et gère ainsi les données propres à un utilisateur. L'installation du moteur OntoMea nécessite uniquement une plateforme Windows ainsi qu'une version de Java supérieure ou égale à 1.5.

Dans un premier temps, afin de bien comprendre le cadre du travail, il est nécessaire de présenter la solution initiale et son fonctionnement. Par la suite, les différents ajouts effectués seront détaillés. A noter que les termes « graphes » et « modèles » ont ici la même signification, l'un étant le terme commun et l'autre celui employer par Jena.

7.2 Solution initiale

Description

La solution de base pour ce travail de diplôme comporte un prototype du moteur OntoMea possédant une interface console classique. Les données sont stockées dans des fichiers RDF et les différents schémas dans des fichiers OWL. Toutes ces informations sont chargées lors du lancement du moteur. En ce qui concerne la structure des données chargées, la solution comporte trois graphes distincts :

- Un graphe de données (asserted model) contenant les différentes instances.
- Un graphe de schémas (schema model) contenant les ontologies.
- Un graphe d'inférences (inferred model) déduit à partir du graphe de données et des schémas d'ontologies.

Structure

La solution OntoMea comporte un répertoire KBOntoMea contenant toutes les informations nécessaires à la gestion des données sémantiques. Ce répertoire est structuré de la façon suivante :

- Un répertoire Cache destiné à la sauvegarde de données téléchargées sur le Web notamment, les informations concernant les données géographiques (geonames).
- Un répertoire Data contenant l'ensemble des données chargées lors du lancement du moteur. Ce répertoire est composé de sous-répertoires pour chaque type de données (user, files, …)
- Un répertoire MCEL contenant les fichiers XLS de transformations des données sous le format MCEL en données sémantiques RDF.
- Un répertoire Schémas contenant les différents schémas d'ontologies téléchargés (voir Configuration).

Fonctionnement

Configuration

La configuration du moteur OntoMea est basée sur le fichier OntoMeaConfig.xml disponible à la racine de la solution. Il comporte les informations concernant l'adresse et le port sur lesquels les services sont atteignables.

Il est également possible de spécifier les schémas d'ontologies que l'on désire charger dans la base de connaissances lors du démarrage. Les informations nécessaires sur les schémas sont les suivantes :

- Le nom de l'ontologie.
- L'Url du fichier décrivant l'ontologie.
- Le lien en local où sauver le schéma après son téléchargement.

Par défaut, le moteur contrôle si le fichier existe déjà dans le répertoire des schémas. Dans le cas contraire, il le télécharge depuis l'Url définit dans le fichier de configuration.

Le fichier est analysé par la classe *OntoMeaXMLConfiguration*. Cette classe permet, grâce à sa méthode *readConfiguration*, de parser les informations et de les transmettre aux différents éléments de l'application. La structure XML est la suivante :

```
<ontoMeaConfig>
<server>
<host>localhost</host>
<port>2020</port>
</server>
<schemas>
  <schema>
        <name>Ontologie's Name</name>
        <url>http://www.exemple.ch/ontologie.owl</url>
        <localCopy>ontologie.owl</localCopy>
  </schema>
</schemas>
</ontoMeaConfig>
```

Lecture des données

Une fois le fichier de configuration lu, le moteur charge en mémoire les données provenant des divers fichiers. Dans un premier temps il se base sur les données de configuration pour sélectionner les schémas d'ontologies à inclure dans le graphe des schémas.

Puis, les fichiers de données stockés dans le répertoire « Data » sont ajoutés dans le graphe de données. Il est nécessaire de spécifier en code les sous-répertoires qu'il faut parcourir lors du chargement. A noter qu'il n'y a pas de gestion des Named graphs.

Pour la lecture dans un fichier, la méthode *read* de la classe *Model* est utilisée avec comme argument un *InputStream* sur le fichier.

Inférences

Le prototype du moteur OntoMea possède une gestion basique des inférences. Il s'agit d'un raisonneur OWL Full[4]. Les inférences sont effectuées automatiquement après la fin du chargement des données. Le nouveau graphe contenant les déductions est stocké dans le graphe d'inférences du moteur. Les requêtes SPARQL provenant des services peuvent alors interroger directement le graphe inféré.

[4] A noter que la définition de OWL Full pour Jena ne correspond pas au langage OWL/Full

Services

Le moteur OntoMea propose trois services accessibles par les différents modules de Memoria-Mea. Ces services sont basés sur l'architecture de Joseki et sont interrogeables directement à travers le réseau via l'adresse et le port spécifié dans le fichier de configuration. Les différents services disponibles sont les suivants :

- KBService qui propose plusieurs fonctionnalités permettant d'interroger et de gérer la base de connaissances. Par exemple, il est possible de récupérer les données propres à un utilisateur ou de les mettre à jour.
- LocService permet d'interroger les Web Service de GeoName afin de récupérer des données géographiques sous forme XML.
- SparqlService permet d'effectuer directement une requête SPARQL sur le graphe inféré.

Serveur Web

Le serveur Web embarqué permet d'accéder aux différents services. Il peut être appelé directement depuis n'importe quelle application (.NET, Java, PHP, …) en spécifiant les coordonnées indiquées dans le fichier de configuration. Le résultat est renvoyé en format XML. Par exemple, pour interroger le service LocService afin de connaître via GeoName l'ensemble des pays d'Europe, il faut utilise l'Url suivantes :

```
http://localhost:2020/loc?query=countries_list&continentCode=eu&lang=fr
```

Cette requête est composée des éléments suivants :

- « localhost » correspond à l'adresse du serveur. Le moteur OntoMea étant destiné à une utilisation locale, cette valeur vaut par défaut localhost.
- 2020 correspond au numéro de port par défaut. A modifier si l'on souhaite traverser un firewall en spécifiant le port 80 (HTTP) ou 443 (HTTPS).
- « loc » correspond au service des locations. Cette argument peut-être remplacé par les valeurs « kb » ou « sparql » selon le service à interroger.
- « countries_list » détermine la méthode du service que l'on désire interroger.

- « &continentCode=eu&lang=fr » correspond aux arguments nécessaires à la méthode.

Améliorations

En partant de la base présentée ci-dessus, plusieurs nouvelles fonctionnalités sont nécessaires afin d'améliorer le moteur OntoMea.

Stockage des données

Dans un premier temps, il semble nécessaire de pouvoir stocker les différents graphes de façon persistante dans une base de données afin d'éviter les temps de chargement sur des grandes quantités de données lors du démarrage du serveur. Cependant, il faut également pouvoir à tout moment mettre à jour les données selon leur provenance. Il est alors nécessaire d'utiliser les Named Graphs.

Inférences

Il existe plusieurs langages permettant d'effectuer des déductions sur un graphe de données sémantiques. Le prototype OntoMea propose uniquement un raisonneur OWL Full. Afin d'optimiser les performances, il est nécessaire de pouvoir sélectionner le type d'inférences à effectuer selon la complexité et la quantité de données.

Interface

Une fois les données stockées de manière persistante, il est nécessaire de pouvoir modifier la configuration de notre moteur en cours d'exécution grâce à une interface graphique.

7.3 Schémas de fonctionnement

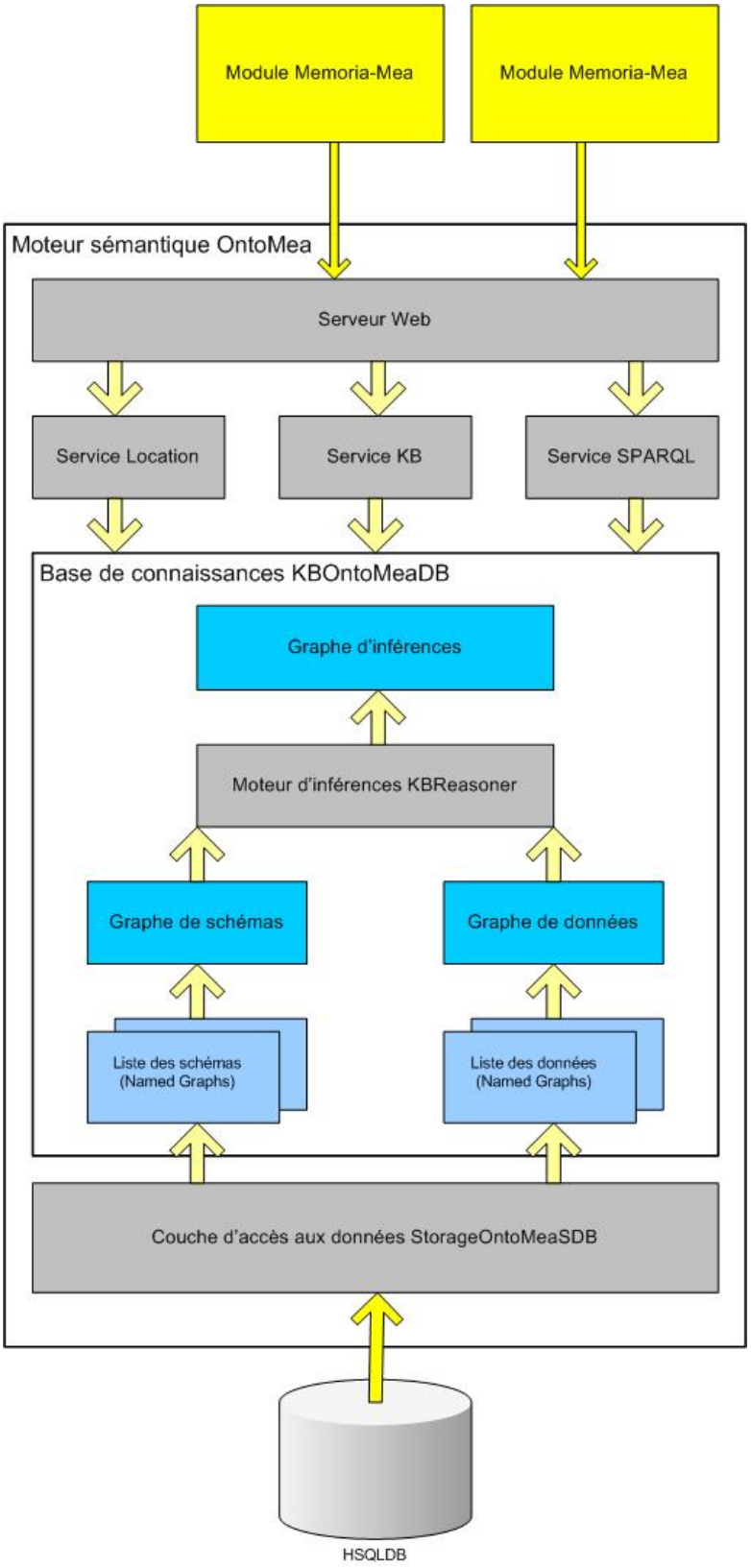


Figure 26 : Schéma de fonctionnement d'OntoMea

7.4 Persistance des données

Introduction

Après l'analyse des différents outils disponibles pour le stockage des données sémantiques et la réalisation de plusieurs tests de performances et de montée en charge, le choix s'est porté sur la solution Jena/SDB/HSQLDB.

Cette décision se base sur un compromis entre performances et fonctionnalités offert par cette solution. En effet, vu que le moteur OntoMea est destiné à un usage local, il est nécessaire d'utiliser une base de données embarquée telle que HSQLDB tout en gardant la possibilité de changer de système de base de données facilement. Le composant SDB permet, grâce aux fichiers de configuration TTL, de préciser de manière efficace les paramètres de connexion à la base de données (voir SDB).

De plus afin de permettre une gestion efficace de la mise à jour des données, il est possible grâce à la structure layout2 de SDB d'intégrer facilement la gestion des Named Graphs à une solution utilisant le framework Jena.

L'implémentation d'un système de stockage persistant pour le moteur OntoMea doit tenir compte des évolutions futures de ce projet notamment, la possibilité de se procurer les données directement depuis un serveur central. Il est donc nécessaire de créer une interface d'accès aux données indépendantes de la base de connaissances afin de pouvoir modifier le système de stockage sans conséquences pour le moteur. Cette interface, appelée *StorageOntoMea*, est décrite dans le chapitre suivant.

Structure

La solution du moteur OntoMea en mode base de données reprend la structure de la solution initiale (voir Solution initiale/Structure). Cependant, dans le répertoire KBOntoMea, un nouveau répertoire « DB » est ajouté dans le cas de l'utilisation d'une base de données embarquée.

Afin de préserver le contenu de la base de connaissances, un répertoire « import » a été ajouté à la racine de la solution permettant aux différents modules d'ajouter ou de mettre à jour des Named Graphs. Les ajouts ne se font donc plus directement dans le répertoire KBOntoMea.

Classe d'accès aux données

Description

L'interface *StorageOntoMea* fournit les méthodes utilisées par la base de connaissances du moteur OntoMea. Les différentes classes d'accès aux données doivent impérativement implémenter les méthodes suivantes :

- *getPersistantOntModel* renvoie un *OntModel* persistant pour un *Model*[5] classique passé en argument. Cette méthode n'est pas utilisée par le moteur actuel mais est tout de même fournie par l'interface en prévision des évolutions futures.
- *getPersistantModel* : en passant en argument le nom du graphe, cette méthode permet de récupérer un Named Graph de la base de données.
- *saveModel* permet de sauvegarder un graphe persistant ou non dans la base de données en spécifiant le nom que l'on désire lui attribuer.
- *close* déconnecte l'application de la base de données
- *getGraphNames* effectue une requête sur la base de données, lui demandant de retourner l'ensemble de Named Graphs qu'elle contient.

Dans le cadre de ce travail de diplôme, une classe implémentant l'interface *StorageOntoMea* utilisant le composant SDB a été développée. La base de connaissances (*KBOntoMeaDB*) possède une variable d'instance de type *StorageOntoMea*. Pour sélectionner le mode de connexion à la base de données, il suffit d'instancier cette variable selon l'objet choisi.

```
private StorageOntoMea som = new StorageOntoMeaSDB();
```

Configuration

Le composant SDB utilise un fichier de configuration pour se connecter à la base de données (voir SDB/Installation). Lors du lancement de l'application, il est possible de passer en paramètre l'adresse du fichier TTL grâce à l'argument --dbconfig via la commande suivante :

[5] La classe *Model* de Jena correspond à un graphe de données

```
start javaw -cp %CP% meaKBSrc/ServerOntoMea –dbconfig config.ttl
```

Par défaut, le fichier OntoMeaConfig.xml situé à la racine de l'application est utilisé.

Fonctionnement

Le composant SDB utilise un objet *Store* qui contient toutes les informations propres à la connexion. La classe *StorageOntoMeaSDB* possède une méthode *createStore* permettant d'instancier cet objet. Cette méthode est appelée automatiquement lorsque l'application effectue une connexion à la base de données et que la valeur de l'objet *Store* est à null.

La méthode *createStore* permet non seulement de retourner l'objet instancié mais gère la création de la base de données dans le cas où celle-ci n'existerait pas encore.

Une fois l'objet *Store* créé, il est possible de charger les différents graphes dans la base de connaissances grâce aux méthodes héritées de l'interface. Les détails quant à la façon dont l'application récupère les graphes persistants sont expliqués dans le chapitre SDB/Utilisation/Utilisation du Model Jena.

Base de connaissances

Description

La base de connaissances est l'élément central du moteur OntoMea. Elle contient toutes les informations des différents graphes ainsi que toutes les méthodes d'interrogation, d'inférences et de gestion de ces graphes. Il existe deux modes de fonctionnement de la base de connaissances :

- Un mode fichier qui charge les données depuis des fichiers à l'instar de la solution initiale.
- Un mode base de données qui utilise la couche d'accès aux données décrite au chapitre précédent. Ce mode possède également une interface graphique de gestion.

La structure de la base de connaissance est la suivant :

- *KBOntoMea* est une classe abstraite qui implémente une partie des méthodes communes aux deux modes de fonctionnement. Le reste des méthodes sont abstraites est doivent être redéfinies dans les classes filles.

- *KBOntoMeaDB* hérite de la classe abstraite *KBOntoMea*, elle contient une instance de *StorageOntoMea*.
- *KBOntoMeaFile* hérite de la classe abstraite *KBOntoMea.*

Une instance de type *KBOntoMea* est créée au démarrage de l'application selon le type de base de connaissances nécessaire. Par défaut, il s'agit du mode base de données. Cependant, il est parfaitement possible de lancer l'application en mode fichier grâce à la commande suivante

```
start javaw -cp %CP% meaKBSrc/ServerOntoMea – filemode
```

A noter également qu'il existe deux fichiers BAT (OntoMeaDB.bat et OntoMeaFile.bat) à la racine de la solution permettant de lancer le moteur dans les deux modes différents.

Le mode fichier

Le mode fichier correspond au fonctionnement initial du moteur OntoMea. Il permet l'exécution en console de l'application. Le fonctionnement de ce mode est décrit plus en détails dans le chapitre Solution initiale/Fonctionnement.

Le mode base de données

Le mode base de données est l'une des principales améliorations réalisées durant ce travail de diplôme. Basé sur la couche d'accès aux données, il modifie en profondeur la gestion des données sémantiques. Son fonctionnement est décrit en détail au chapitre suivant.

Gestion de données persistantes

Schéma

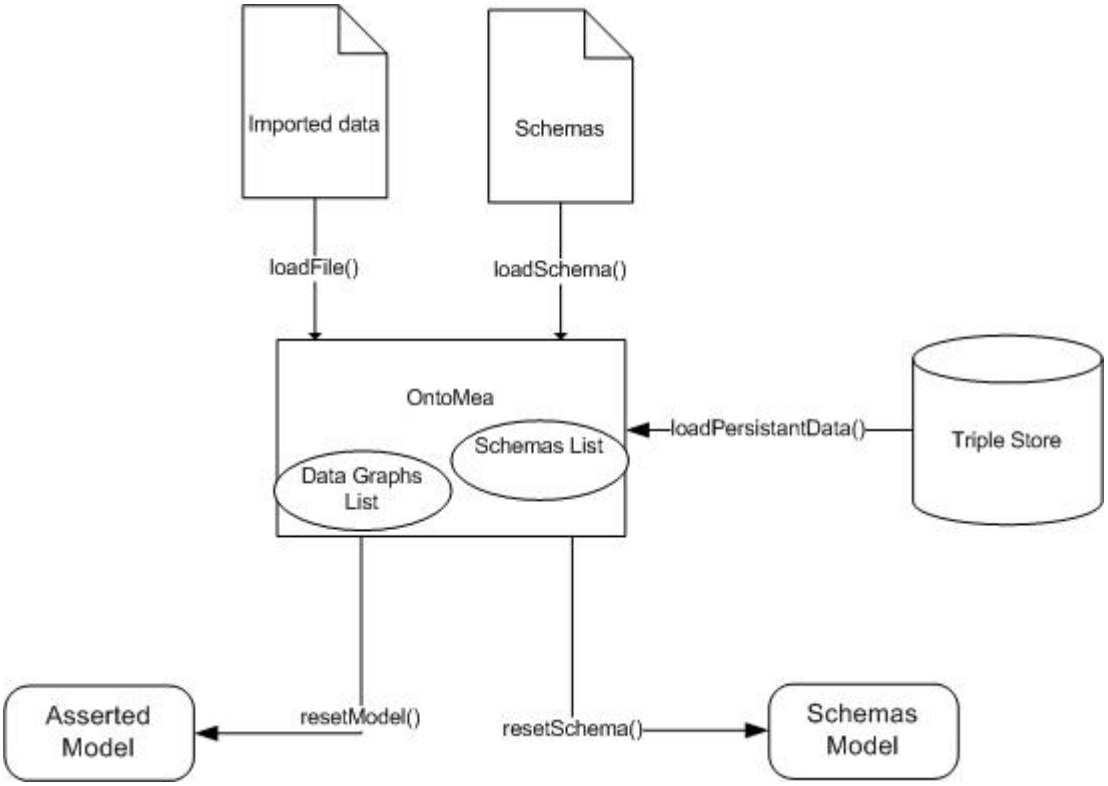


Figure 27 : Fonctionnement de KBOntoMeaDB

Chargement des données

La première étape lors du lancement de l'application est de récupérer les données persistantes grâce à la méthode *loadPersistantData* qui est évidemment une méthode propre à *KBOntoMeaDB*. Une fois les noms des graphes récupérés grâce à la couche de persistance, les différents *Model* sont créés et insérés dans deux listes distinctes selon leur type (données ou schémas). Ces deux listes permettent ensuite de générer le graphe de données et celui de schémas grâce aux méthodes *resetModel* et *resetSchema*.

Le nom des graphes est défini selon leur type et leur provenance. Par exemple un schéma provenant du fichier foaf.owl aura comme nom :

- http://www.memoria-mea.ch/schema/foaf.owl

Toutes les informations concernant les noms de graphes sont stockés dans des constantes de la classe *KBOntoMea.*

Pour des raisons de performances, seuls les Named Graphs sont stockés automatiquement dans la DB. En effets les inférences effectuées sur un *Model*

persistant sont beaucoup plus gourmandes en ressources. Pour cette raison, les graphes de base et de schémas sont uniquement en mémoire. Il est cependant possible d'effectuer une copie d'un graphe dans la base de données grâce à la méthode *saveModel* de la classe de persistance. C'est le cas, notamment, du graphe de base qui est sauvegardé en cas de modification. Le graphe de schémas est généré lors du lancement de l'application selon la configuration et n'est jamais stocké dans la base de données.

Ajout de données

Une fois les données persistantes récupérées, le moteur OntoMea vérifie si le répertoire « import » et ses sous-répertoires contiennent des nouvelles données à importer dans la base de connaissances. Le processus est entièrement automatisé et se déroule en sept étapes distinctes :

1. Lister les différents fichiers existants dans le répertoire « import ».
2. Générer le nom du graphe.
3. Contrôler si le graphe existe déjà dans la liste des graphes de données.
4. Ajouter le graphe ou le mettre à jour selon les résultats de l'étape précédente.
5. Supprimer le fichier du répertoire « import » et en faire une sauvegarde dans le répertoire KBOntoMea/Data/DB/Cache.
6. Mettre à jour le graphe de base grâce à la méthode *resetModel*.
7. Sauvegarder le graphe de base dans la DB.

Le processus décrit ci-dessus est le fonctionnement automatique de mise à jour de la base de connaissances lors du lancement de l'application. Il est également possible d'intégrer directement un fichier en cours d'exécution, grâce au service *KBService* ou directement depuis l'interface graphique. Dans ce cas, le processus reprend les étapes 2-3-4-6-7 du mode automatique. L'étape 5 est réalisée uniquement si le nouveau fichier appartient au répertoire « import ».

Mises à jour de données

La mise à jour des données suit exactement les mêmes processus que l'ajout. L'application se base toujours sur le nom du graphe pour déterminer s'il s'agit d'une mise à jour ou d'une insertion. Le processus étant automatisé, les différents modules n'ont donc pas à se soucier de la gestion des Named Graphs.

Ajout de schémas

La configuration des schémas est basée sur le fichier de configuration OntoMeaConfig.xml. Tous les schémas utilisés par la base de connaissances sont décrits sous le format suivant :

```
<schema>
        <name>Ontologie's Name</name>
        <url>http://www.exemple.ch/ontologie.owl</url>
        <localCopy>ontologie.owl</localCopy>
</schema>
```

Lors de l'exécution de la méthode *loadSchemas,* le moteur, en se basant sur les données de configuration, va effectuer le processus ci-dessous. A noter qu'il est également possible d'effectuer une mise à jour des schémas grâce à l'argument *reload* de la méthode *loadSchemas*. A la fin du processus, le graphe de schémas est généré à partir des différents Named Graphs correspondant aux données du fichier de configuration.

Processus d'ajout de schémas

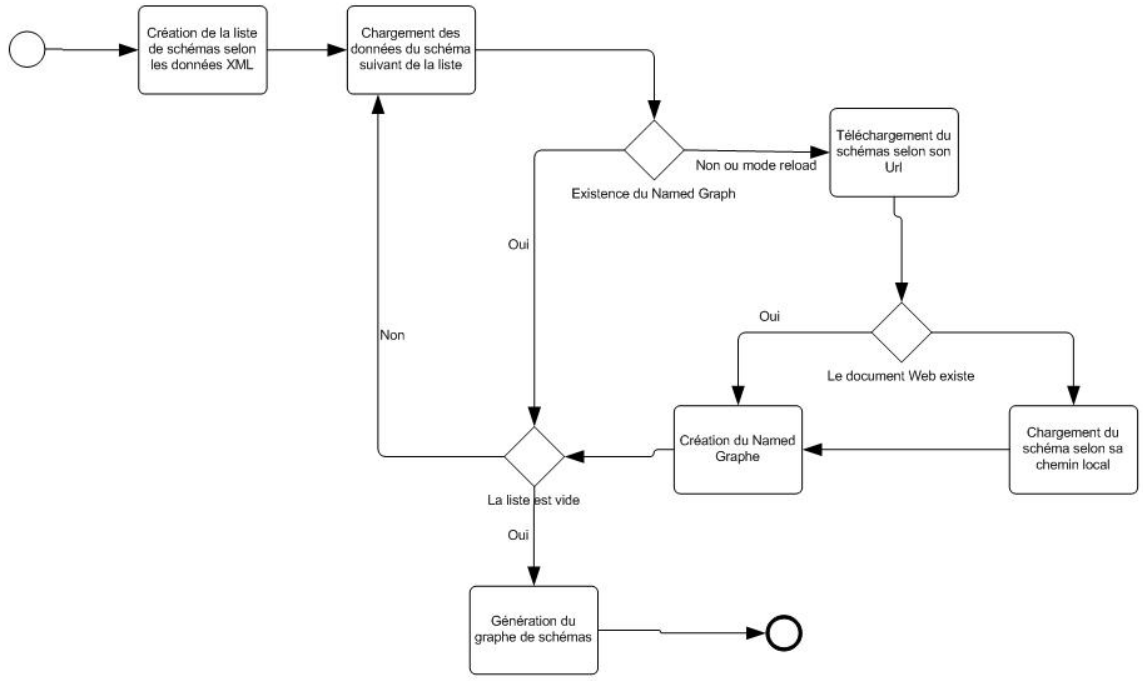


Figure 28 : Processus d'ajout de schémas

Sauvegarde

Les différents *Model* peuvent être sauvegardés de deux manières différentes :

- En créant directement un graphe persistant depuis la couche d'accès aux données grâce à la méthode *getPersistantModel*. De cette façon, toutes les modifications effectuées sur le *Model* sont instantanément sauvegardées dans la base de données.
- En utilisant la méthode *saveModel* de la couche d'accès aux données. Cette méthode peut être appliquée à tous les objets *Model,* qu'ils soient persistants ou non. Elle effectue une copie dans la base de données du *Model* passé en argument.

Dans la solution OntoMea, l'ensemble des Named Graphs sont sauvegardés en utilisant la première solution. Par contre, pour le graphe de base, la seconde solution est utilisée pour une raison de performances, comme précisé au chapitre Chargement des données.

Pour plus d'informations sur l'utilisation des graphes persistants avec Jena, se référer à l'annexe 5 : Jena Database interface.

7.5 Moteur d'inférence

Introduction

Le Web sémantique offre plusieurs langages capables de décrire des règles d'inférences sur une ontologie. Jena offre, pour chaque langage, un raisonneur particulier. La complexité et la puissance de déduction dépendent du type de raisonneurs choisi. Dans ce contexte, il est nécessaire à l'utilisateur de pouvoir configurer à sa guise le moteur d'inférence afin d'avoir le meilleur rapport performance/déduction possible.

Les quatre raisonneurs disponibles sont les suivants :

- RDFS
- OWL
- Générique (utilise des règles prédéfinies)
- DIG

A noter que Jena offre également un raisonneur DAML + OIL qui a été volontairement omis dans la solution. En effet OWL, plus récent, a été créé à partir de DAML + OIL et offre de nombreuses fonctionnalités supplémentaires. OWL est donc en train de remplacer DAML + OIL.

Une nouvelle classe appelée *KBReasoner* est ajoutée à la solution. Elle comporte toutes les informations de configuration du raisonneur ainsi que les différentes méthodes utilisées par la base de connaissances.

Configuration

Le moteur d'inférences se base sur le fichier de configuration d'Ontomea. Il est possible de préciser l'état et les caractéristiques des quatre modes d'inférence. La structure XML de configuration des raisonneurs est la suivante :

```
<reasoners>
  <reasoner>
        <name>RDFS</name>
        <status>ON</status>
  </reasoner>
  <reasoner>
        <name>OWL</name>
        <status>ON</status>
        <type>FULL</type>
  </reasoner>
  <reasoner>
        <name>RULE</name>
        <status>OFF</status>
        <files>
              <file>
                    <uri>test.rule</uri>
              </file>
        </files>
  </reasoner>
  <reasoner>
        <name>DIG</name>
        <status>OFF</status>
        <uri>http://localhost:8081</uri>
  </reasoner>
</reasoners>
```

De la même façon que pour les schémas, la classe *OntoMeaXMLConfiguration* permet de récupérer ces données et de les transmettre au moteur d'inférences. Il est parfaitement possible d'activer plusieurs raisonneurs en même temps. La classe *KBReasoner* gère automatiquement la meilleure configuration selon les types de raisonneurs sélectionnés. Par exemple, si les raisonneurs OWL et RDFS sont activés en même temps, le moteur d'inférences utilisera uniquement le raisonneur OWL car les règles RDFS sont incluses dans le standard RDFS.

Fonctionnement des inférences

La classe abstraite *KBOntoMea* possède en variable d'instance un objet de type *KBReasoner*. Le moteur d'inférences fonctionne exactement de la même manière dans le cas d'une exécution de l'application en mode base de données ou en mode fichiers.

La classe *KBReasoner* contient deux méthodes publiques accessibles depuis la base de connaissances :

- *inferModel* qui permet de créer un graphe d'inférences *InfModel* à partir de la configuration du moteur OntoMea.
- *checkValidity* permet d'analyser un graphe d'inférences et de déterminer s'il contient des incohérences. Cette méthode, étant très gourmande en ressources, n'est pas utilisée par défaut par la base de connaissances dans un souci de performance.

Dans le moteur OntoMea, la gestion des inférences est automatisée. A chaque requête sur la base de connaissance, une vérification est faite sur le statut du graphe d'inférences et, dans le cas où une mise à jour est nécessaire, un appel à la méthode *inferModel* est exécuté. L'utilisateur n'a donc à aucun moment besoin de se soucier de l'état du modèle d'inférences.

La création des différents raisonneurs se base sur l'analyse faite du framework Jena (voir Jena/Inférences). Le schéma ci-dessous permet de rappeler la base du fonctionnement des inférences avec Jena. Dans le cadre de l'application OntoMea, l'objet *Reasoner* est utilisé pour DIG et le raisonneur générique et *OntModel* pour OWL et RDFS.

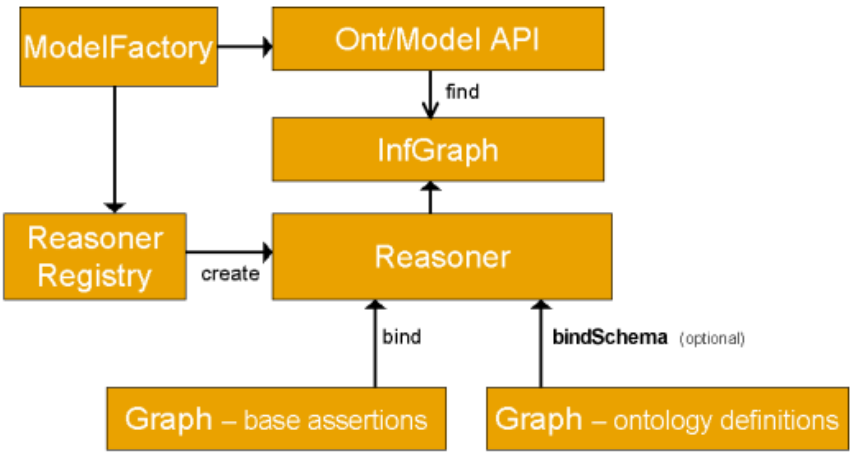


Figure 29 : Création de Model avec Jena (source http://jena.sourceforge.net/inference/)

La suite de ce chapitre sur le moteur d'inférences détaille le fonctionnement des différents raisonneurs.

Les différents raisonneurs

Raisonneur RDFS

Le raisonneur RDFS est la solution la plus légère mais également la plus performante. Elle est idéale pour les ontologies simples n'utilisant que les règles de déductions suivantes :

- *subClassOf*
- *subPropertyOf*
- *type*
- *domain*
- *range*

Prenons un exemple précis d'une déduction effectuée par le raisonneur RDFS. Il est possible grâce à la propriété *subClassOf* de gérer l'héritage avec des données RDF. Dans ce cas, l'application a chargé une ontologie simple utilisant cette propriété :

```
<rdf:Description rdf:about=" #Mother">
   <rdfs:subClassOf rdf:resource=" #Person"/>
</rdf:Description>
<rdf:Description rdf:about=" #Father">
   <rdfs:subClassOf rdf:resource=" #Person"/>
</rdf:Description>
```

Cette ontologie précise que les classes *Mother* et *Father* sont des sous-classes de *Person*. Maintenant le fichier d'instances suivant est ajouté à la base de connaissance :

```
<rdf:Description rdf:about=" #Marie">
   <rdf:type rdf:resource=" #Mother"/>
</rdf:Description>
```

Si on désire récupérer toutes les ressources du type *Person* il est nécessaire d'activer le raisonneur RDFS (et/ou OWL, DIG) sans quoi, la requête retournera un résultat vide.

En fait, le raisonneur ajoute au modèle de base les lignes suivantes :

```
<rdf:Description rdf:about=" #Marie">
   <rdf:type rdf:resource=" #Person"/>
</rdf:Description>
```

Ce fonctionnement est commun à tous les raisonneurs. Le nombre et la complexité des nouvelles données déduites dépendent du type de raisonneur.

Raisonneur OWL

Le raisonneur OWL de Jena est une implémentation du standard OWL/Lite. S'il est nécessaire d'utiliser un raisonneur OWL/DL, il est préférable d'utiliser l'interface DIG d'OntoMea.

Jena offre trois niveaux de complexité pour son raisonneur OWL :

- Micro
- Mini
- Full[6]

Le niveau de complexité augmente selon le type de raisonneur sélectionné et, en contre partie, les performances diminuent. Malgré tout, OWL/Micro permet d'effectuer la plupart des déductions OWL et reste le meilleur compromis pour la plupart des ontologies.

Dans le fichier de configuration, en plus de son statut, il est possible de préciser le type de raisonneur OWL dans la balise XML <type></type>. Les choix possibles correspondent aux différents niveaux cités ci-dessus (MICRO, MINI, FULL). Par défaut, le niveau MICRO est activé.

Le raisonneur OWL permet d'effectuer des déductions plus complexes que RDFS. Il est possible de créer des classes définies avec comme condition des restrictions OWL. Par exemple, si la classe *Car* a une restriction *owl :hasValue* 4 sur la propriété *hasWheelNumber*, le raisonneur déduira automatiquement que l'instance suivante appartient à la classe *Car* :

```
<Vehicule rdf:ID=" #MyVehicule">
   <has:WheelNumber>4</ has:WheelNumber >
</Vehicule >
```

Raisonneur Générique

Description

Le raisonneur générique de Jena permet d'exécuter des inférences selon un set de règles définies. Ces règles peuvent provenir de sets déjà définis comme

[6] A noté que la définition d'OWL Full pour Jena ne correspond pas au langage OWL/Full

RDFS ou OWL[7]. Il est également possible de créer ses propres règles en se basant sur la syntaxe fournie par Jena.

Dans le moteur OntoMea, le raisonneur générique fonctionne en parallèle avec les autres raisonneurs. Il n'est donc pas nécessaire d'ajouter des sets de règles RDFS ou OWL lors de la création de règles personnalisées.

Il est possible dans le fichier de configuration de préciser quels sont les fichiers contenant les règles personnalisées. Les formats admis pour la balise <uri></uri> sont les suivants :

- l'URL du fichier de règles.
- Le chemin absolu du fichier sur la machine locale.
- Le nom du fichier à condition qu'il se trouve dans le répertoire KBOntoMea/Data/Rules.

Le moteur d'inférences d'OntoMea automatise la mise en commune des différentes règles prédéfinies dans un set utilisable par le raisonneur générique.

Structure

Le raisonneur générique est en fait constitué de deux moteurs de règles distincts :

- *Forward engine* permet de déduire des inférences pendant la création du graphe inféré. Ce graphe contient alors les données de base ainsi que les nouvelles déductions générées par le raisonneur générique.
- *Backward engine* est un moteur d'inférence qui s'exécute lors d'une requête sur le graphe d'inférences. Aucunes nouvelles données ne sont ajoutées aux données de bases. Lorsque des inférences sont déduites lors de l'exécution d'une requête, les résultats obtenus sont stockés en mémoire.

Le mode de fonctionnement du raisonneur générique peut être défini grâce à la propriété *PROPruleMode* d'un objet *GenericRuleReasoner*. Les valeurs attribuables sont les suivantes :

- forward
- forward RETE[8]
- backward

[7] Pour ces sets de règles, il est préférable d'utiliser le raisonneur prédéfini.
[8] L'algorithme de Rete est un algorithme performant de filtrage par motif (« pattern matching ») intervenant dans l'implémentation de systèmes de règles de production (source Wikipedia)

- hybrid

Le moteur OntoMea utilise par défaut la configuration hybride. Cette structure permet d'effectuer des inférences en deux temps. Les règles attribuées au *Backward engine* peuvent se baser sur les résultats obtenus par le *Forward engine*.

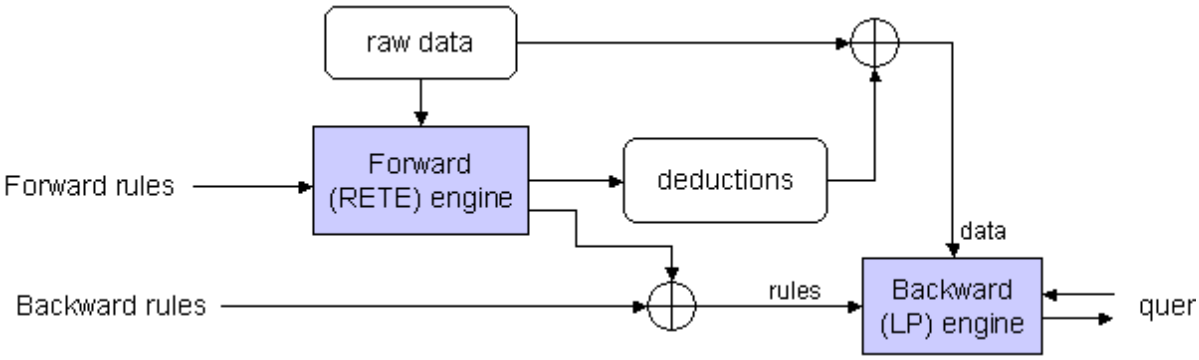


Figure 30 : Fonctionnement du raisonneur générique (http://jena.sourceforge.net/inference/)

Format de règles

Le format utilisé par les règles personnalisées de Jena est composé de deux parties distinctes :

- Une liste de conditions (premises).
- Une liste de conclusions.

Le moteur d'inférences basé sur les règles analyse les conditions et, s'il trouve des ressources qui les vérifient, ajoute les déclarations définies dans les conclusions de la règle. Les conditions peuvent prendre deux formes différentes :

- Un triplet. Par exemple, *(?s rdfs :subClassOf ?p)* signifie que la ressource ?s doit être une sous-classe de ?p.
- Une fonction. Par exemple *greaterThan(?x, ?y).*

Les conclusions ne sont possibles que sous la forme de triplets.

Il existe deux modes d'exécution des règles. Comme défini dans le chapitre précédent, les règles peuvent être exécutées avant ou après la création du graphe d'inférences. Pour spécifier le mode d'exécution d'une règle personnalisée, il est nécessaire d'utiliser les formats suivants :

- conditions -> conclusions pour le mode *forward*.
- conclusions <- conditions pour le mode *backward*

Il est évidement nécessaire que le raisonneur soit configuré dans le mode approprié ou dans le mode hybride. Dans le cas contraire, la règle sera exécutée dans le mode défini par le raisonneur.

Pour plus d'information quant au format des règles avec Jena, se référer à l'annexe Annexe 3 : Jena Inferences support.

.

Interface DIG

Le dernier raisonneur disponible dans l'application OntoMea est un raisonneur interfacé avec un moteur externe offrant une interface DIG. Ce genre de moteur utilise le langage OWL/DL et peut être une alternative au raisonneur OWL de Jena qui est une implémentation de OWL/Lite.

Afin de configurer OntoMea pour une utilisation avec un moteur externe, il est nécessaire de préciser dans le fichier de configuration l'URL de l'interface DIG.

La création d'un raisonneur utilisant une interface DIG est un peu plus complexe que pour les autres modes d'inférences. Dans un premier temps, il faut créer une ressource dans un *Model* définissant l'adresse de l'interface DIG.

```
Model cModel = ModelFactory.createDefaultModel();
Resource conf = cModel.createResource();
conf.addProperty( ReasonerVocabulary.EXT_REASONER_URL, cModel.createResource(
DIGURI ) );
```

Une fois la ressource créée, l'Objet *DIGReasonerFactory* permet de générer un *DIGReasoner*.

```
DIGReasonerFactory drf = (DIGReasonerFactory) ReasonerRegistry.theRegistry().getFactory(
DIGReasonerFactory.URI );
DIGReasoner r = (DIGReasoner) drf.create( conf );
```

Enfin, à partir du raisonneur ainsi créé, il est possible grâce à l'objet *ModelFactory* de générer le graphe d'inférences selon les spécificités nécessaires.

```
r = (DIGReasoner)r.bindSchema(KBOntoMea.kb.schemaModel);
OntModelSpec spec = new OntModelSpec( OntModelSpec.OWL_DL_MEM );
spec.setReasoner( r );
OntModel t = ModelFactory.createOntologyModel( spec,m);
```

Le graphe d'inférences généré par ce processus ne contient pas les nouvelles déductions par défaut. Lors d'une requête, le moteur d'inférences externe sera interrogé.

A noter qu'OWL/DL, bien que plus complète, nécessite une plus grande précision dans les schémas d'ontologies. De nombreux schémas fonctionnant

parfaitement avec OWL/Lite peuvent engendrer des erreurs lors de leur utilisation via l'interface DIG.

7.6 Interface graphique

Description

Dans la solution initiale, le moteur OntoMea s'exécutait uniquement dans une interface console. Pour modifier certains paramètres, il était nécessaire de modifier le fichier de configuration et de relancer le serveur. De plus, il était difficile d'analyser le fonctionnement du moteur sans interface de logs.

Afin de permettre de configurer le serveur en cours d'exécution et d'ajouter les nouvelles fonctionnalités décrites lors des chapitres précédents, une interface graphique est désormais disponible lors du lancement en mode base de données.

Par défaut, lors du lancement de d'OntoMea, l'application est minimisée dans la zone de notifications de Windows. En règle générale, l'utilisateur n'a pas à se soucier de la configuration de son moteur. Par contre, il est possible en cliquant sur l'icône de faire apparaître la fenêtre de gestion. Le bouton de minimisation permet de la cacher à nouveau une fois le moteur configuré.

Fonctionnalités

L'interface a été séparée en cinq onglets distincts :

- « Main » est l'interface principale du moteur. On y retrouve les principales fonctionnalités de gestion ainsi que les logs de fonctionnement du serveur.
- « Inferences » permet de configurer en cours d'exécution le moteur d'inférences d'OntoMea.
- « Models » permet de visionner et d'exporter les Named Graphs contenus dans la base de connaissances.
- « Query » affiche l'ensemble des requêtes effectuées sur la base de connaissances et, le cas échéant, les erreurs survenues.
- « Points of interests » permet d'importer les données des points d'intérêt d'un utilisateur. (voir L'interface des points d'intérêts).

Les différentes fonctionnalités de ces onglets sont détaillées dans les chapitres suivants.

Le menu « Main »

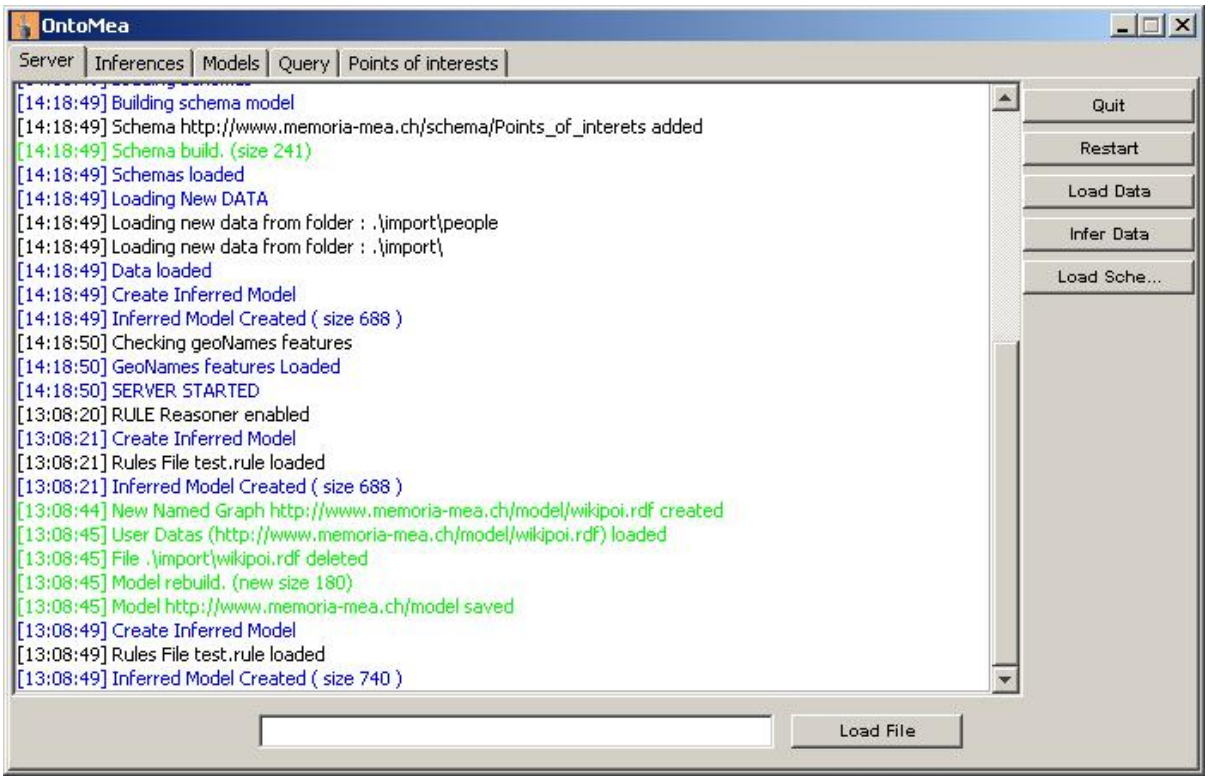


Figure 31 : Interface principale

Cette interface apparaît par défaut lors du lancement de l'application. Elle comporte l'ensemble des logs de fonctionnement du moteur. La couleur indique le type de logs affichés. Les couleurs utilisées par l'interface sont les suivantes :

- Noir pour les informations de base (modification de la configuration, chargement des Named Graphs, …).
- Bleu pour définir les étapes lors du fonctionnement du moteur (chargement des données, création du graphe d'inférences).
- Vert pour les opérations modifiant la base de connaissances (ajout de données, mise à jour des graphes de base ou de schémas, exportation d'un graphe,…)
- Rouge pour les erreurs de fonctionnement du serveur.

En plus de la journalisation des évènements, il est possible d'effectuer diverses opérations sur le serveur en cours d'exécution :

- « Quit » permet de quitter l'application. Contrairement au mode console, le mode base de données doit pouvoir fermer correctement les différents éléments, notamment la connexion à la base de

données. L'évènement de fermeture a été surchargé afin de permettre un arrêt propre du serveur depuis le bouton de fermeture de la fenêtre Windows, ainsi que depuis le menu contextuel lié à l'icône dans la zone de notifications.

- « Restart » permet de redémarrer le serveur. Cette fonctionnalité est utile si l'on désire prendre en compte des modifications effectuées dans le fichier de configuration.
- « Load Data » vérifie s'il existe des données importées dans les répertoires correspondants et les intègre dans la base de connaissances. Cette fonctionnalité ne nécessite plus de redémarrage du serveur.
- « Load Schemas » permet de recharger les schémas définis dans le fichier de configuration. Cette fonctionnalité permet d'aller chercher les dernières versions disponibles sur le Web ou en local. A noter qu'en cas d'ajout de schémas dans le fichier de configuration, un redémarrage du serveur est nécessaire afin de prendre en compte les nouvelles informations.
- « Infer Data » force le moteur d'inférences à régénérer le graphe inféré. Cette fonctionnalité est utile si l'on désire tester le nombre de déductions obtenues par les divers raisonneurs. Cependant la régénération est effectuée automatiquement en cas de modification de la configuration si le moteur reçoit une requête.
- « Load File » permet de charger un fichier RDF ou OWL dans la base de connaissances. Il est possible d'y insérer un chemin local ou une URL. Si le fichier est valide, un nouveau Named Graphs persistant est généré.

Le menu « Inferences »

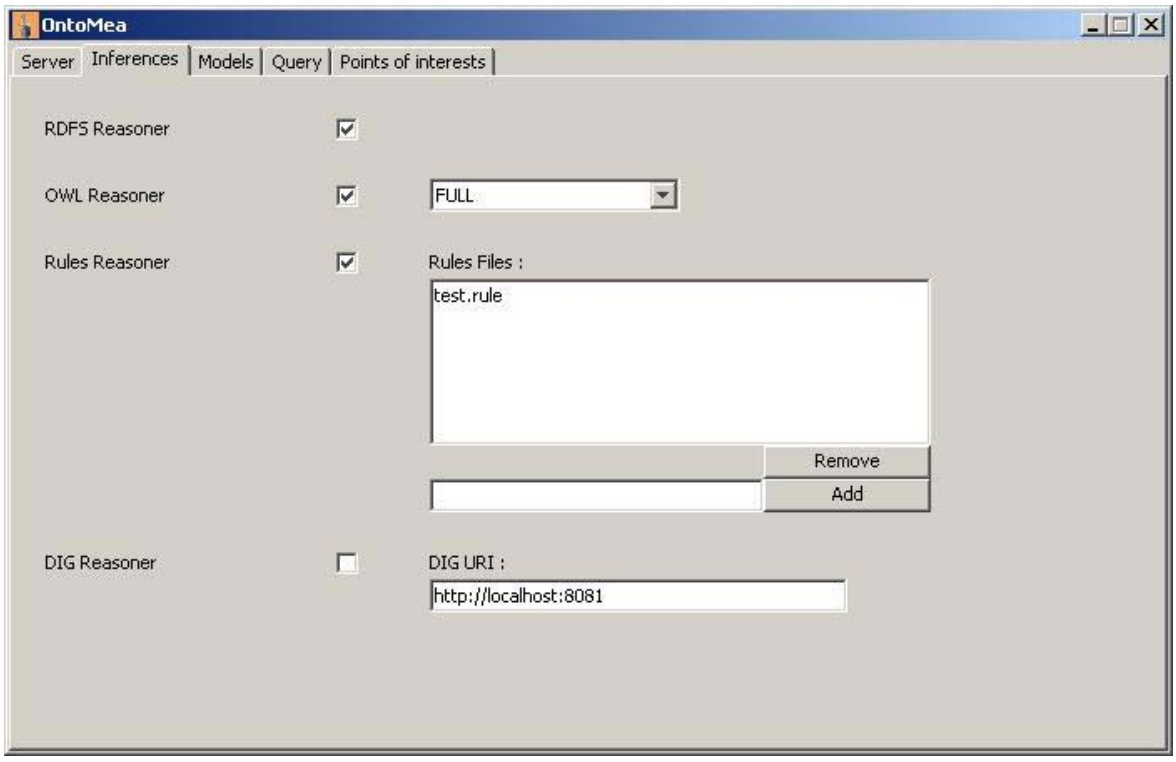


Figure 32 : Interface de gestion du moteur d'inférences

Au lancement de l'application, l'interface de gestion des inférences reprend les paramètres du fichier de configuration. Il est possible de les modifier en cours d'exécution afin de déterminer quelle est la configuration optimale du moteur d'inférences selon les données à inférer. Chaque modification depuis cette interface entraînera une régénération du graphe d'inférences lors d'une requête sur la base de connaissances. Cependant, les modifications effectuées ne modifient pas le fichier de configuration. Les paramètres de configuration des différents raisonneurs sont expliqués en détail dans le chapitre Les différents raisonneurs.

A noter que certains raisonneurs sont inclus dans d'autres. Le tableau suivant détaille ces inclusions.

include	RDFS	OWL	Generic	DIG
RDFS	X			
OWL	X	X		
Generic	X*	X*	X	
DIG	X	X		X

* : need the include a predefined set of rules

Figure 33 : Tableau des raisonneurs

Le menu « Models »

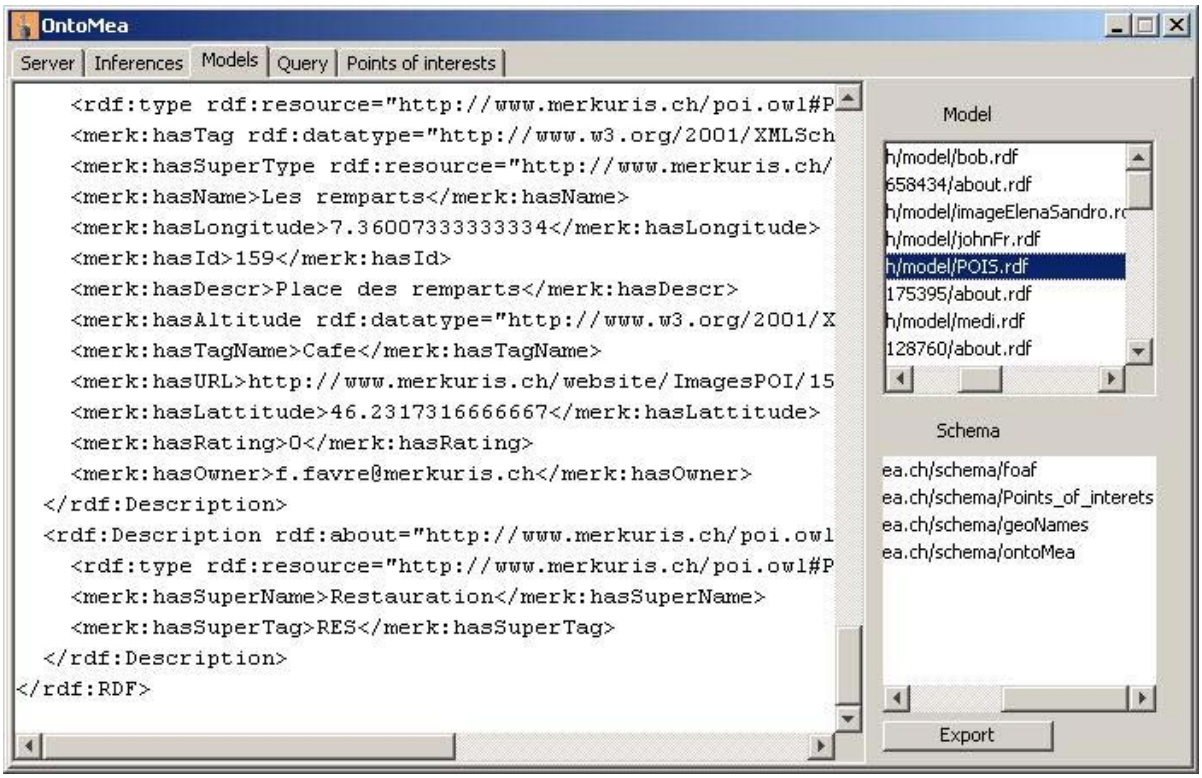


Figure 34 : Interface de visualisation des graphes

Cette interface permet de visualiser l'ensemble des Named Graphs contenus dans la base de connaissances. Elle est composée de deux listes, l'une pour les graphes de données et l'autre pour les schémas. En sélectionnant un graphe, il est possible d'afficher l'ensemble des triplets qu'il contient sous format RDF.

Il est également possible d'exporter un graphe dans un fichier RDF. Les fichiers exportés sont placés automatiquement dans le répertoire « Export » à la racine de la solution.

Le menu « Query »

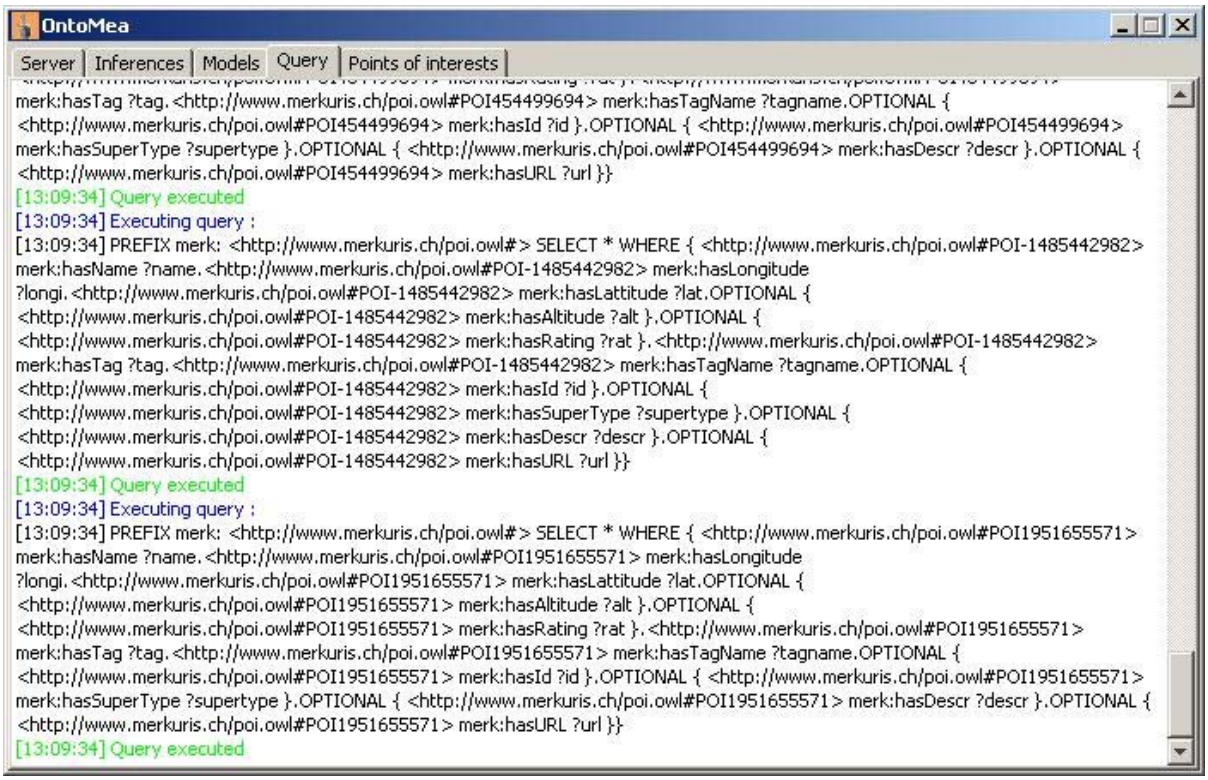


Figure 35 : Interface de journalisation des requêtes

En cliquant sur l'onglet « Query », il est possible de visualiser l'ensemble des requêtes effectuées sur la base de connaissances. Les couleurs utilisées sont les suivantes :

- Noir pour les requêtes.
- Bleu lors de la réception d'une requête.
- Vert lorsque la requête a été exécutée avec succès.
- Rouge lorsqu'une erreur s'est produite.

Cet outil a été développé pour permettre d'identifier, en cas d'erreur, les requêtes impliquées.

Le menu « Points of interests

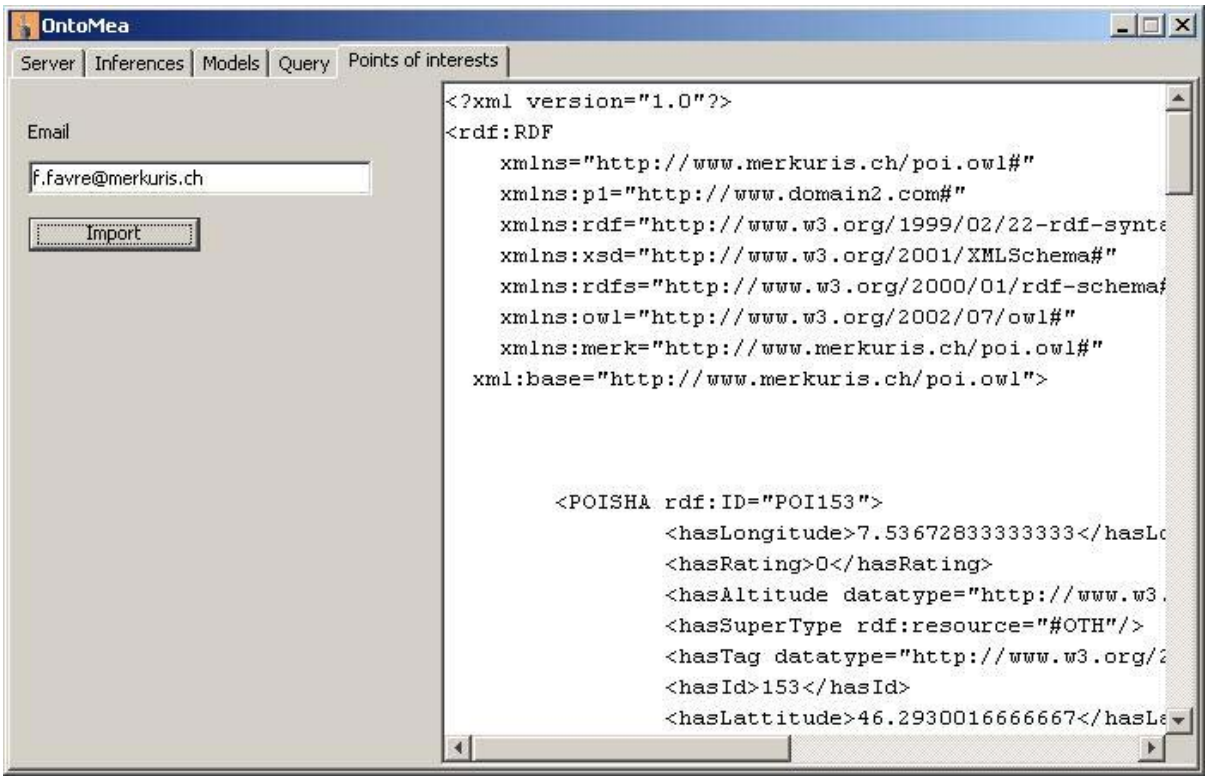


Figure 36 : Interface d'importation des points d'intérêts

Dans le chapitre suivant L'interface des points d'intérêts, un cas d'utilisation du moteur OntoMea est développé. Il s'agit ici d'intégrer des données provenant d'un serveur externe et de les réutiliser dans des modules utilisant le moteur sémantique. Cette interface est destinée à importer ces données dans la base de connaissances. Pour accéder aux informations, il est nécessaire de s'identifier avec une variable passée en POST. La fonctionnalité de base d'importation de données ne peut donc pas être utilisée dans ce cas, les variables ne pouvant être passées qu'en GET.

Cette interface permet d'appeler une fonctionnalité du serveur des points d'intérêts qui génère un fichier RDF à partir de données provenant d'un serveur MSSQL. Un fois le fichier récupéré, il est intégré dans la base de connaissances selon la méthode définie dans le chapitre Ajout de données.

7.7 Problèmes rencontrés

Performance des graphes persistant

Les opérations effectuées sur les graphes persistants entraînent une mise à jour automatique dans la base de données. Les opérations complexes, comme la création d'inférences, ralentissent considérablement l'application si elles sont effectuées sur un graphe persistant. Dans un premier temps, les graphes utilisés par le moteur d'inférences (données et schémas) étaient stockés de manière persistante.

Afin de contourner ce problème, ces graphes ne sont plus persistants. Le graphe de données est copié lors du lancement de l'application et est sauvé dans la base de données en cas de modification. Le graphe de schémas quant à lui n'est plus du tout stocké mais est régénéré à l'exécution.

Bien que, en utilisant cette manière de procéder, le lancement du moteur OntoMea est légèrement plus lent (environ une seconde), les temps d'attente en cours d'exécution ont été considérablement diminués. Sur les tests d'inférences avec plus des 6000 triplets, le temps d'attente est passé d'environ 15 secondes, selon les raisonneurs activés, à moins d'une seconde.

Compatibilité des schémas

Les schémas importés dans la base de connaissances ne sont pas forcément compatibles avec les raisonneurs sélectionneurs. Par défaut, pour OWL, Jena fournit un raisonneur OWL en version full. Un serveur DIG lui fournit un raisonneur OWL/DL. Ces deux méthodes d'inférences nécessitent des schémas compatibles, sans quoi la génération d'inférences déclenche des erreurs.

Afin de permettre au moteur d'effectuer des inférences OWL sans se soucier des problèmes de compatibilité des schémas, le moteur OntoMea permet à l'utilisateur de sélectionner le niveau du raisonneur OWL (Micro, Mini, Full). Au niveau Micro, le moteur d'inférences est moins sensible aux soucis de compatibilités.

En modifiant le niveau du raisonneur OWL, il est possible de sélectionner le meilleur rapport puissance/performance possible. Bien que le niveau Micro soit, en théorie, le moins performant, il a été capable d'exécuter tous les tests d'inférences effectués durant le projet.

Fonctionnement du raisonneur OWL

Lors de la correction du problème précédent, un autre problème est survenu. Le fonctionnement des niveaux du raisonneur OWL ne semble pas correspondre à la documentation de Jena (voir annexe 3). En effet, lors des tests d'inférences, les niveaux Mini et Full ont systématiquement fournis les mêmes résultats. De plus le niveau Micro a permis d'effectuer des inférences uniquement disponibles avec la version Mini ou Full. Il s'avère, en fait, que le niveau Micro fonctionne au niveau Mini et que le niveau Mini utilise le niveau Full.

Aucune solution n'a été trouvée pour ce problème. Cependant, cela n'entrave en rien le bon fonctionnement du moteur OntoMea. Il a été décidé de laisser, malgré tout, les trois niveaux en prévision d'une future correction de Jena.

7.8 Améliorations futures

Partage de données

A l'heure actuelle, le moteur OntoMea est destiné à un fonctionnement local. Cependant, il peut être utile de permettre à plusieurs utilisateurs de partager des informations dans le cadre du projet Memoria-Mea.

Deux solutions peuvent être envisagées :

- Un réseau Peer-to-Peer de moteurs OntoMea communiquant directement entre eux. Cette solution peut être envisagée dans le cas d'une utilisation dans un réseau privé. Chaque moteur est à la fois client et serveur, ce qui peut engendrer des problèmes de communications lors d'une utilisation à plus grandes échelles (derrière un firewall par exemple).
- Un serveur central permettant de distribuer les Named Graphs nécessaires aux clients OntoMea. Cette solution s'applique bien à une utilisation par le Web.

Dans l'éventualité où une telle solution serait implémentée, la structure du moteur permet déjà de modifier facilement la couche d'accès aux données. La seule impérative est que la nouvelle couche implémente l'interface *StorageOntoMea* et redéfinisse ses méthodes. Cette modification serait alors parfaitement transparente pour la base de connaissances. Pour plus d'information sur la classe d'accès aux données, se référer au chapitre Classe d'accès aux données.

8. L'INTERFACE DES POINTS D'INTÉRÊTS

8.1 Introduction

Ce chapitre présente un exemple d'utilisation du moteur sémantique OntoMea au travers d'un module .NET permettant de visionner les points d'intérêt (POI) d'un utilisateur MerKuris. A noter que cette application a un rôle de démonstrateur et n'est en aucun cas un module fonctionnel de Memoria-Mea.

MerKuris est une application mobile avec GPS permettant d'afficher en temps réel des utilisateurs et des points d'intérêts sur une carte fournie par le Web Service Mappoint de Microsoft. Ce service offre également un serveur Web permettant d'accéder à des données stockées dans une base de données MSSQL Server.

Les fonctionnalités développées dans cette application permettent de mettre en évidence les différents principes du Web sémantique au travers du moteur OntoMea. Une première étape consiste à présenter les différentes méthodes de transfert de données entre les différentes applications de l'architecture :

- Le transfert de données entre OntoMea et le module des POI sous forme de résultats SPARQL.
- Le transfert de données depuis le serveur Web MerKuris vers la base de connaissances du moteur OntoMea par génération de fichiers RDF.
- La sauvegarde de données importées depuis Wikipedia par le module des POI dans la base de connaissances.

Le module a été développé afin de démontrer les capacités de déduction du moteur d'inférences d'OntoMea. Un schéma d'ontologies compatible avec OWL/Lite et OWL/DL est disponible sur le serveur MerKuris.

8.2 Schémas

Le schéma suivant représente l'architecture qui est utilisée par l'interface des POI. Les différentes phases de transfert de données, représentées par des

flèches, utilisent chacune une méthode particulière décrite dans les chapitres suivants. Ce cas d'utilisation permet de présenter plusieurs moyens de transfert de données en utilisant le moteur sémantique OntoMea.

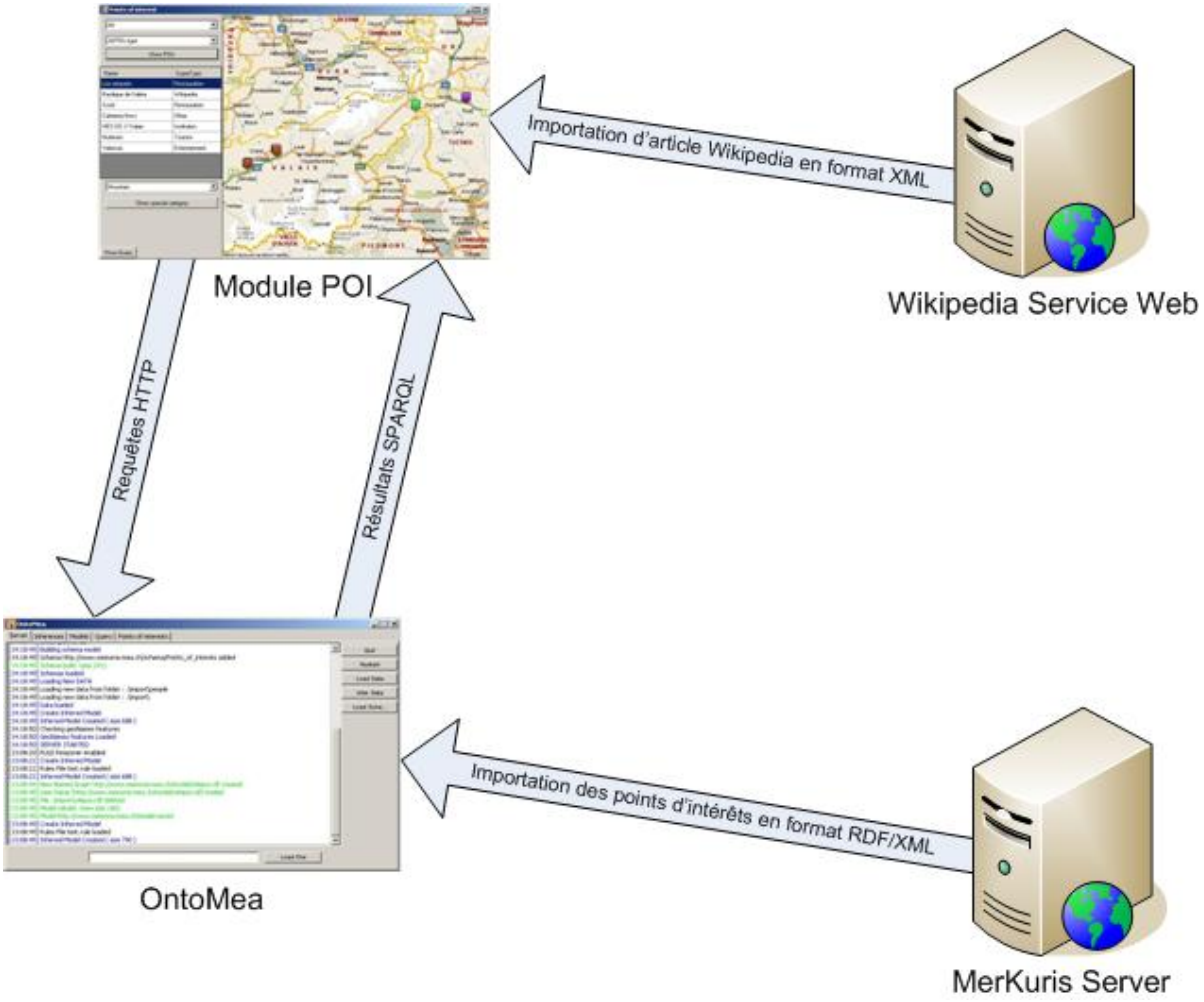


Figure 37: Schéma de fonctionnement de l'interface des POI

8.3 Transfert de données

De MerKuris à OntoMea

Le serveur de MerKuris possède une interface Web permettant d'accéder à des données personnelles. Afin de permettre au moteur OntoMea de récupérer des données RDF valides, un moteur de conversion, basé sur des templates, a été développé. Il est possible d'accéder à ces informations grâce à l'URL suivante :

http://www.merkuris.ch:8080/pois.rdf

Pour plus de sécurité, toutes les informations transmises au serveur sont passées en POST. Il est donc nécessaire de construire un objet *HttpURLConnection* afin de pouvoir transmettre l'email de l'utilisateur pour récupérer ses POI. Le code suivant est utilisé :

```
HttpURLConnection  conn = (HttpURLConnection)url.openConnection();
conn.setDoInput(true);
conn.setDoOutput(true);
conn.setRequestMethod("POST");
conn.setRequestProperty("Content-type", "application/x-www-form-urlencoded");
conn.connect();

OutputStreamWriter  writer = new OutputStreamWriter(conn.getOutputStream());
writer.write(data,0,data.length());
writer.flush();
```

Le serveur retourne dans le flux les données RDF générées. Le moteur OntoMea peut ensuite les intégrer dans sa base de connaissances.

A noter que pour la création d'inférences sur ces données, il est nécessaire d'ajouter au fichier de configuration le nouveau schéma d'ontologies fourni par MerKuris à l'adresse suivante :

```
http://www.merkuris.ch/merk.owl
```

Données OntoMea

Une fois les données intégrées dans la base de connaissances, le module de visionnage des POI peut y accéder au travers des services disponibles du moteur sémantique. Afin de charger les POI selon les critères définis par l'utilisateur, une requête SPARQL est envoyée au *SPARQLService* qui retourne les résultats au format XML. Le module possède une classe d'accès aux services appelée *OntoMeaServices* ainsi qu'un convertisseur des résultats XML appelé *XMLResult*. Le code suivant permet d'accéder directement aux services (ici le service SPARQL) et de récupérer les résultats dans une liste de *XMLResult*.

```
OntoMeaServices  oms = OntoMeaServices.Query("sparql?query=" +
      System.Web.HttpUtility.UrlEncode(query), null);
List<XMLResult> l = oms.getResultSet(true);
```

Il suffit alors de parcourir la liste de résultats et d'en extraire les données. Pour ce module, une classe *POI* ayant pour constructeur *POI(XMLResult xml)* a été créée. Cela permet de transformer directement un résultat XLM en instance de *POI*.

Données Wikipedia

Wikipedia propose un service Web permettant de récupérer les articles possédant des coordonnées géographiques dans un rayon défini autour d'une position latitude-longitude. Voici un exemple de requêtes sur ce service :

http://ws.geonames.org/findNearbyWikipedia?lat=47&lng=9

Le module permet de visionner les différents articles dans un rayon de dix kilomètres autour d'un *POI*. Pour cela, il convertit le fichier XML reçu du Web Service en plusieurs instances de *POI*.

Les Templates

Afin de permettre la transformation de données objet en fichiers RDF valides, un template a été généré à partir du logiciel Protégé. Ce fichier est utilisé sur le serveur MerKuris ainsi que sur le module des *POI* afin de générer des données compatibles avec le schéma d'ontologies en remplaçant les différentes balises. Potentiellement, ce fichier pourrait être utilisé par toutes applications permettant de générer des points d'intérêts.

```
<?xml version="1.0"?>
<rdf:RDF
   xmlns="http://www.merkuris.ch/poi.owl#"
   xmlns:p1="http://www.domain2.com#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:merk="http://www.merkuris.ch/poi.owl#"
  xml:base="http://www.merkuris.ch/poi.owl">


  [LOOPPOI]
  <POI[TAGCATEGORY] rdf:ID="POI[ID]">
            <hasLongitude>[LONGITUDE]</hasLongitude>
            <hasRating>[RATING]</hasRating>
            <hasAltitude datatype="http://www.w3.org/2001/XMLSchema#float">[ALTITUDE]
             </hasAltitude>
            <hasSuperType rdf:resource="#[SUPERTYPETAG]"/>
            <hasTag datatype="http://www.w3.org/2001/XMLSchema#string">[TYPETAG]
             </hasTag>
            <hasId>[ID]</hasId>
            <hasLattitude>[LATITUDE]</hasLattitude>
            <hasName>[NAME]</hasName>
            <hasTagName>[TYPENAME]</hasTagName>
            <hasDescr>[DESCRIPTION]</hasDescr>
            <hasOwner>[EMAILOWNER]</hasOwner>
            <hasURL>[URLPICTURE]</hasURL>
  </POI[TAGCATEGORY]>
  [/LOOPPOI]
```

```
[LOOPSUPERTYPE]
<POISC[SUPERTYPENAME] rdf:ID="[SUPERTYPETAG]">
        <hasSuperName>[SUPERTYPENAME]</hasSuperName>
        <hasSuperTag>[SUPERTYPETAG]</hasSuperTag>
</POISC[SUPERTYPENAME]>
[/LOOPSUPERTYPE]

<POISCWikipedia rdf:ID="WIKI">
        <hasSuperName>Wikipedia</hasSuperName>
        <hasSuperTag>WIKI</hasSuperTag>
</POISCWikipedia>
</rdf:RDF>
```

8.4 Sauvegarde de données

Afin de démontrer les possibilités de mise à jour de la base de connaissances depuis un module, il est possible d'y sauvegarder des *POI* Wikipedia. Pour cela, le template de conversion est utilisé pour transformer les données en fichiers RDF. Ensuite, le service *KBService* du moteur OntoMea, destiné à la gestion de la base de connaissances, est appelé grâce à la méthode suivante :

```
oms.executeQuery("kb?cmd=update_data&fileURL=file:///"+f.FullName);
```

Le fichier est alors chargé par le moteur sémantique et les données sont insérées dans le Named graphs approprié.

8.5 Web Service Mappoint

Microsoft fournit un Web Service SOAP permettant à une application d'accéder à une base de données de cartes géographiques. En connaissant les positions latitude-longitude des points d'intérêts, il est possible de charger la carte appropriée.

Une librairie DLL d'accès au service Mappoint a été ajoutée à la solution. Elle fournit un objet *MapPointManager* qui permet d'exécuter les méthodes suivantes :

- *getBestMap* fournit la meilleure carte possible avec les différents POI sélectionnés.
- *getImageByHeightWidth* permet de récupérer une image d'une carte centrée sur un POI.

Ces méthodes retournent un tableau de byte contenant l'image. Afin de l'afficher dans notre application, il est nécessaire d'ouvrir un flux sur la mémoire :

```
Bitmap.FromStream(new System.IO.MemoryStream(image);
```

8.6 Fonctionnalités

Sélection des POI

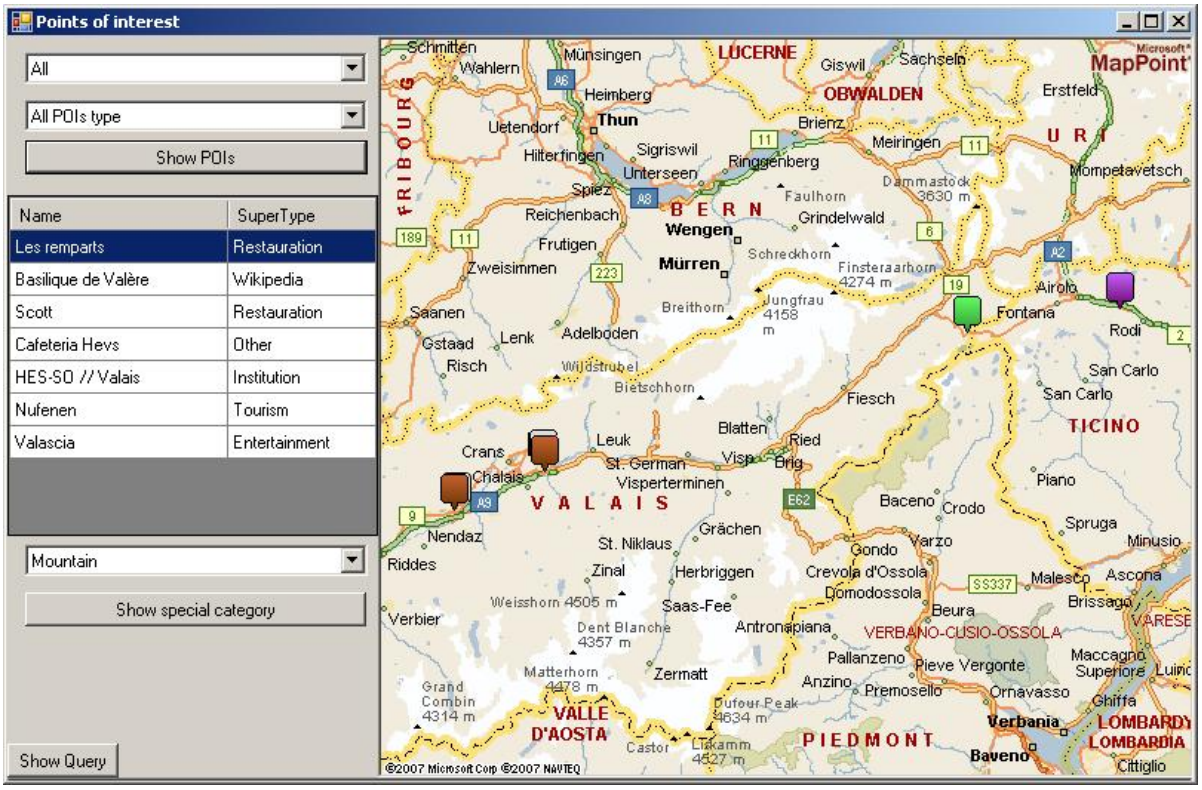


Figure 38 : Interface de visionnage des POI

L'interface de visionnage des points d'intérêts permet de sélectionner les données contenues dans la base de connaissances selon différents critères. Les fonctionnalités ont été spécialement pensées afin de démontrer les différentes possibilités du moteur sémantique. Les différents modes de sélection suivants permettent de mettre en évidence les capacités du moteur d'inférences :

- Sélection d'une catégorie de POI (publique, privés, partagé).
- Sélection de tous les POI. Nécessite un raisonneur RDFS.
- Sélection d'un super type de POI. Nécessite un raisonneur OWL.

- Sélection d'une catégorie spéciale. Nécessite un raisonneur OWL et des règles génériques.

En sélectionnant un POI sur la carte, il est possible d'afficher les caractéristiques suivantes :

- Son nom,
- Sa description,
- Son évaluation,
- L'image qui lui est associée,
- La liste des articles Wikipedia dans un rayon de dix kilomètres.

Figure 39 : Détail d'un POI

Les articles Wikipedia

En cliquant sur le bouton Wikipedia lié au point d'intérêt sélectionné, l'application interroge le service Web Wikipedia et affiche les nouveaux POI générés. En sélectionnant l'un des articles, il est possible d'afficher son nom et sa définition. De plus l'article complet est disponible en cliquant sur l'image associée.

Les informations chargées sont alors disponibles pour être insérées dans la base de connaissances. L'utilisateur peut sélectionner les articles qu'il désire sauvegarder et cliquant sur le bouton « Save ». Un fichier RDF sera généré selon la méthode décrite au chapitre Données Wikipedia.

Lors des prochaines recherches de POI dans la base de connaissances, il sera possible de sélectionner les données Wikipedia ainsi ajoutées.

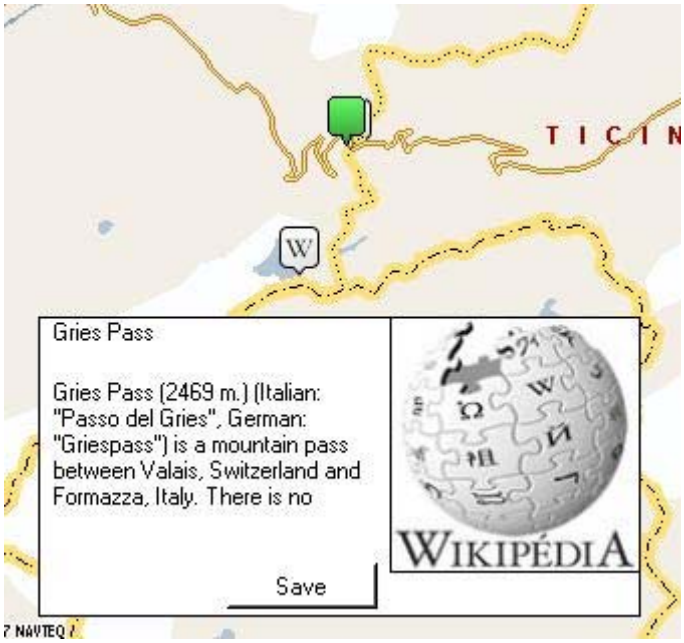


Figure 40 : Détail d'un POI Wikipedia

8.7 Exemples d'utilisation

Introduction

Ce chapitre décrit plusieurs exemples d'utilisation du moteur d'inférences au travers du module des points d'intérêts. Un schéma d'ontologies OWL a été créé avec Protégé. Afin de bien comprendre les implications des règles de déductions sur les différents modes de sélections, il semble primordial de décrire globalement la structure des informations sémantiques.

Il existe deux classes principales :

- *POI* qui représente un point d'intérêts. Les POI n'appartiennent pas directement à la classe *POI* mais aux classes filles suivantes selon leur catégorie:
  - *POIPRI* pour la catégorie privée
  - *POIPUB* pour la catégorie publique
  - *POISHA* pour la catégorie partagée
  - *POIWIKI* pour les articles Wikipedia

- *POISuperType* est une classe représentant le type de POI (Restauration, Transport, …). Chaque instance de POI possède un *POISuperType*. Pour chaque super type, une classe fille est créée.

En plus de ces classes de base, il existe de nombreuses classes définies contenant des restrictions OWL. Pour avoir une vision globale de la structure, l'image suivante représente la hiérarchie telle qu'elle apparaît dans Protégé.

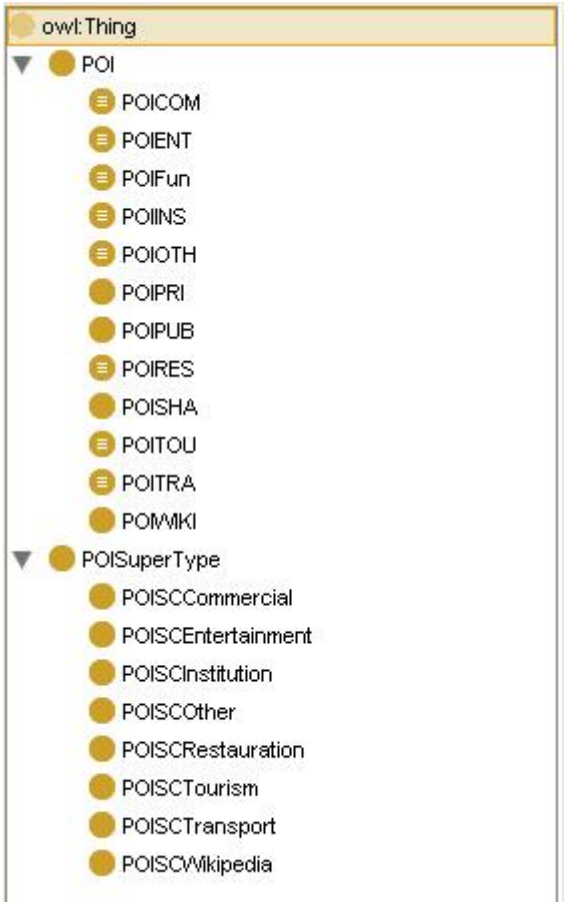


Figure 41 : Hiérarchie des classes de POI dans l'ontologie

Schéma

Le schéma suivant représente l'ensemble des types de POI fournis par le serveur MerKuris. Les différentes catégories sont regroupées dans une super catégorie représentée par un objet *POISuperType* dans notre ontologie.

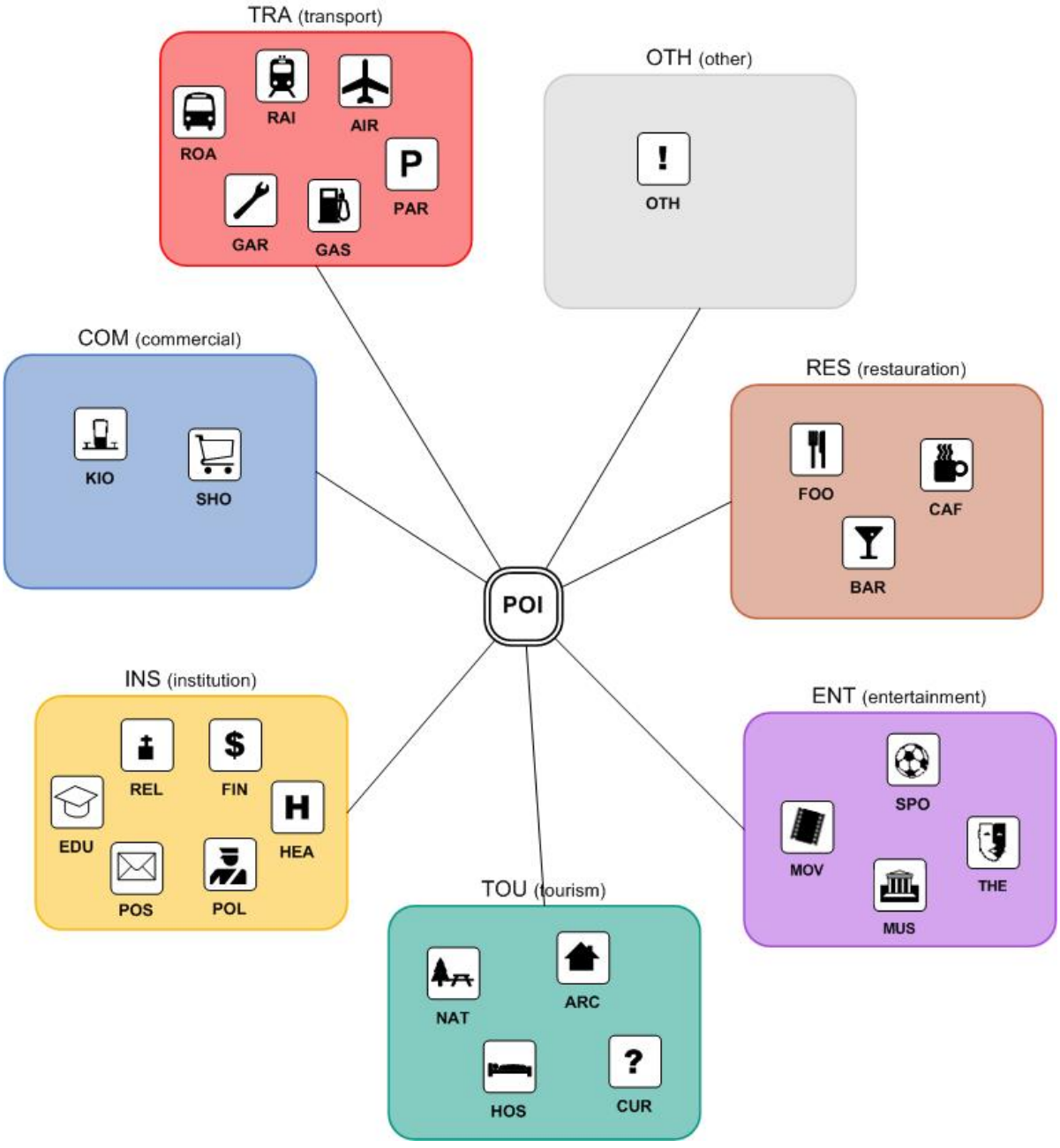


Figure 42: Organigramme des points d'intérêts

Cas d'utilisations

Cas 1

L'exemple le plus basique est une sélection de POI selon une catégorie définie. En effet, les instances de POI publics appartiennent forcément à la classe *POIPUB*. Pour ce type de sélection, aucun raisonneur n'est nécessaire. L'utilisateur peu effectuer cette requête sans activer le moteur d'inférences d'OntoMea.

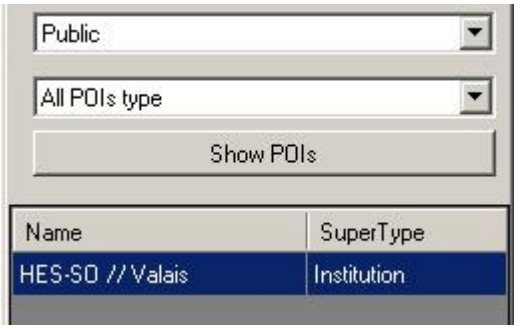


Figure 43 : Résultat cas 1

Cas 2

L'utilisateur décide d'afficher tous les POI contenus dans la base de connaissances. Cependant, en gardant la même configuration du moteur d'inférences que lors du cas 1, il ne reçoit aucune information. La requête qui a été envoyée sur le serveur est la suivante :

```
PREFIX merk: <http://www.merkuris.ch/poi.owl#>  SELECT * WHERE { ?poiUri a merk:POI .
?poiUri merk:hasLattitude ?lat } ORDER BY ?lat
```

La classe *POI* est la classe parente de toutes les classes de base des POI. Le moteur d'inférences a ainsi besoin d'utiliser la propriété RDFS *subClassOf*. En activant le raisonneur RDFS, l'utilisateur pourra alors sélectionner tous les données.

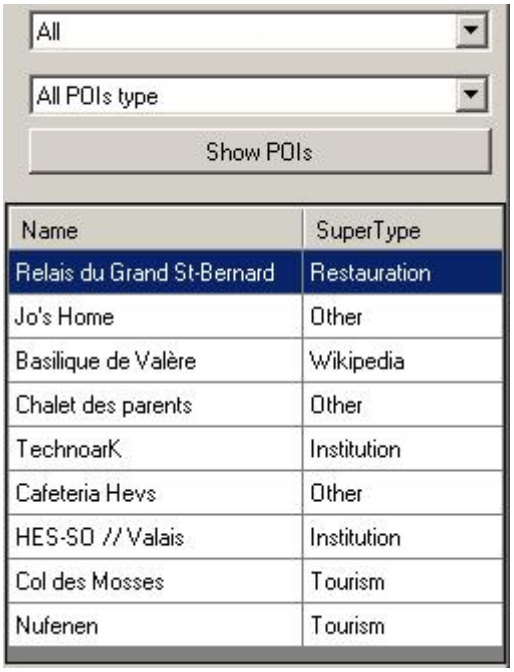


Figure 44 : Résultat cas 2

Cas 3

Le troisième cas de figure est la sélection d'une classe liée à un *POISuperType*. En effet, la classe définie *POIRES*, par exemple, possède une restriction nécessaire et suffisante indiquant qu'un *POI* est de type *POIINS* s'il a comme super type une instance de *POISCInstitution*. Ce type de restriction utilise la propriété OWL *someValueOf*. Il sera donc nécessaire d'activer le raisonneur OWL ou DIG.

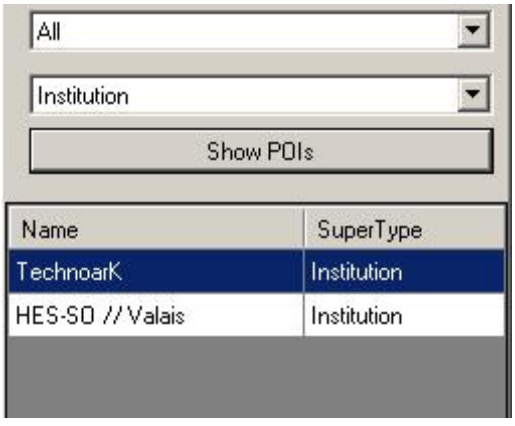


Figure 45 : Résultat cas 3

Cas 4

Le dernier cas d'utilisation présenté ici est l'utilisation de règles prédéfinies. Il est donc nécessaire dans ce cas d'activer le raisonneur de règles génériques. La règle suivante à été définie :

```
@prefix merk: <http://www.merkuris.ch/poi.owl#>.
[merk1: (?P rdf:type merk:POI) (?P merk:hasAltitude ?A) ge(?A,1500)  -> (?P rdf:type
merk:POIMountain) ]
```

Cette règle définit que toutes instances de type *POI* ayant une altitude supérieur à 1500 mètres appartiennent à la classe *POIMountain*. A noter que cette classe n'est même pas définie dans l'ontologie. La requête suivante permet de récupérer tous les POI de montagne.

```
PREFIX merk: <http://www.merkuris.ch/poi.owl#>  SELECT * WHERE { ?poiUri a
merk:POIMountain . ?poiUri merk:hasLattitude ?lat } ORDER BY ?lat
```

Pour cette règle particulière, il est nécessaire de combiner les raisonneurs RDFS et générique. En effet, la règle fait référence à la classe *POI* et donc à la propriété *subClassOf* (Voir Cas 1).

Figure 46 : Résultat cas 4

9. CONCLUSION

Les améliorations apportées au moteur OntoMea durant ce travail permettent au projet Memoria-Mea de disposer des nombreuses spécificités technologiques du Web sémantiques. Le moteur d'inférences est maintenant totalement personnalisable et est capable de s'adapter aux différents formats d'ontologies. De plus, une attention particulière a été apportée aux performances du moteur. L'utilisation de technologie du Web sémantique telles que les Named Graphs et les Triples Stores permet d'offrir une grande flexibilité dans la gestion des données sémantiques.

Cependant, la durée de ce projet a nécessité une focalisation sur des points particuliers, comme les inférences et le stockage, mettant de côté d'autres notions importantes dans cette catégorie de service. Lors de l'utilisation de données personnelles, il est capital de donner une importance particulière aux notions de partage et de sécurité. Les évolutions futures du moteur sémantique OntoMea devront concerner ces points particuliers en implémentant une architecture basée sur un serveur central par exemple. Dans cette optique, le développement réalisé durant ce travail a été effectué en tenant compte des possibles améliorations à venir.

D'un point de vue personnel, le fait de travailler avec de nouvelles technologies m'a permis de découvrir de nombreuses solutions et architectures différentes et originales.

Le Web sémantique étant une technologie récente, il n'existe pas encore une réelle méthodologie pour le traitement de telles informations. La phase d'analyse a donc été extrêmement importante afin de dégrossir les différentes solutions émergeantes. Outre le Web sémantique lui-même, certaines étapes de ce travail ont concernées de nombreux domaines de l'informatique tels que les bases de données, le réseau, le développement ou l'algorithmie. Bien qu'il ait été parfois difficile de trouver de la documentation de qualité, les nombreuses recherches effectuées ont été enrichissantes et m'ont permis de compléter mes connaissances informatiques.

Pour finir, je souhaite signaler les excellentes conditions de travail et je tiens à remercier mon professeur chargé du suivi Anne Le Calvé ainsi que mon responsable technique Fabian Cretton pour leur disponibilité et leurs conseils tout au long de ce travail.

10. BIBLIOGRAPHIE

- MEMORIA MEA. *Memoria Mea* [En ligne] http://www.memoria-mea.ch/
- WIKIPEDIA. *Web sémantique* [En ligne] http://fr.wikipedia.org/wiki/Web_s%C3%A9mantique
- TRANSNETS. *Web 3.0 : définition* [En ligne] http://pisani.blog.lemonde.fr/2007/10/07/web-30-definitions/
- W3C. *Resource Description Framework (RDF)* [En ligne] http://www.w3.org/RDF/
- W3C. *RDF Vocabulary Description Language 1.0: RDF Schema* [En ligne] http://www.w3.org/TR/rdf-schema/
- W3C. *Web Ontology Language (OWL)* [En ligne] http://www.w3.org/2004/OWL/
- W3C. *SPARQL Query Language for RDF* [En ligne] http://www.w3.org/TR/rdf-sparql-query/
- WIKIPEDIA. *Inférences* [En ligne] http://fr.wikipedia.org/wiki/Inf%C3%A9rence
- PELLET REASONER. *Pellet : the Open Source OWL DL Reasoner* [En ligne] http://pellet.owldl.com/
- RACER REASONER. *Racer* [en ligne] http://www.sts.tu-harburg.de/~r.f.moeller/racer/
- PROTEGE. *The Protégé Ontology Editor* [en ligne] http://protege.stanford.edu/
- FOAF ONTOLOGY. *FOAF Vocabulary Specification* [en ligne] http://xmlns.com/foaf/spec/
- TUTORIAL PROTEGE-OWL. *A Practical Guide To Building OWL Ontologies Using The Protégé-OWL Plugin and CO-ODE Tools* [en ligne] http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf
- JENA. *Jena – A Semantic Web Framework for Java* [en ligne] http://jena.sourceforge.net/
- SDB. *SDB - A SPARQL Database for Jena* [en ligne] http://jena.sourceforge.net/SDB/

- HSQLDB. *hsqldb - 100% Java Database* [en ligne] http://hsqldb.org/
- MULGARA. *Mulgara semantic store* [en ligne] http://docs.mulgara.org/
- JRDF. *JRDF (Java RDF)* [en ligne] http://jrdf.sourceforge.net/
- JENA. *Jena 2 Inference support* [en ligne] http://jena.sourceforge.net/inference/
- JENA. *HOWTO use Jena with an external DIG reasoner* [en ligne] http://jena.sourceforge.net/how-to/dig-reasoner.html
- JENA. *HowTo Use HSQLDB with Jena2* [en ligne] http://jena.sourceforge.net/DB/hsql-howto.html
- MERKURIS. *MerKuris – Discovering Your Friends* [en ligne] http://www.merkuris.ch/website/

11. TABLE DES ILLUSTRATIONS

12. GLOSSAIRE

DAML + OIL

Langage de description des ontologies. Ancêtre de OWL.

DIG (Interface)

L'interface DIG (développée par DL Implementation Group) permet d'échanger des données logiques dans un standard XML.

Graphe

Un graphe est une représentation symbolique d'un réseau.

Inférence

« L'inférence est une opération mentale qui consiste à tirer une conclusion (d'une série de propositions reconnues pour vraies). Ces conclusions sont tirées à partir de règles de base. »

(Source Wikipedia)

Modèle

Voir Graphe

Moteur d'inférences

« Un moteur d'inférence est un logiciel correspondant à un algorithme de simulation des raisonnements déductifs. »

(Source Wikipedia)

Named Graph

Un Triple Store permet de stocker les différents graphes séparément en leur donnant un nom. Les graphes ainsi nommés s'appellent Named Graph.

Ontologie

« En informatique et en science de l'information, une ontologie est un ensemble structuré de concepts permettant de donner un sens aux informations. Elle est aussi un modèle de données qui représente un ensemble de concepts dans un domaine et les rapports entre ces concepts. Elle est employée pour raisonner au sujet des objets dans ce domaine. »

(Source Wikipedia)

OWL

« Web Ontology Language — dit OWL — est un dialecte XML basé sur une syntaxe RDF. Il fournit les moyens pour définir des ontologies Web structurées. »

(Source Wikipedia)

RDF

« Resource Description Framework (RDF) est un modèle de graphe destiné à décrire de façon formelle les ressources Web et leurs métadonnées, de façon à permettre le traitement automatique de telles descriptions. Développé par le W3C, RDF est le langage de base du Web sémantique. Une des syntaxes (sérialisation) de ce langage est RDF/XML. »

(Source Wikipedia)

RDFS

« RDF Schema ou RDFS est un langage extensible de représentation des connaissances. Il appartient à la famille des langages du Web sémantique publiés par le W3C. RDFS fournit des éléments de base pour la définition d'ontologies ou vocabulaires destinés à structurer des ressources RDF. »

(Source Wikipedia)

Schéma

Voir Ontologie

Triplet

En sémantique, un triplet est l'union des trois ressources sujet, prédicat et objet. Un ensemble de triplets forme un graphe.

Triples Store

Un Triple Store est un système de stockage des données ayant pour structure un graphe de triplets.

Web sémantique

« Le Web sémantique désigne un ensemble de technologies visant à rendre le contenu des ressources du World Wide Web accessible et utilisable par les programmes et agents logiciels, grâce à un système de métadonnées formelles, utilisant notamment la famille de langages développés par le W3C »

(Source Wikipedia)

13. DÉCLARATION SUR L'HONNEUR

Je déclare, par ce document, que j'ai effectué le travail de diplôme ci-annexé seul, sans autre aide que celles dûment signalées dans les références, et que je n'ai utilisé que les sources expressément mentionnées. Je ne donnerai aucune copie de ce rapport à un tiers sans l'autorisation conjointe du RF et du professeur chargé du suivi du travail de diplôme, y compris au partenaire de recherche appliquée avec lequel j'ai collaboré, à l'exception des personnes qui m'ont fourni les principales informations nécessaires à la rédaction de ce travail et que je cite ci-après :

- Fabian Cretton, Assistant, Institut Informatique de gestion, HES-SO Valais Wallis

Signature :

14. LISTE DES ANNEXES

Annexe 1 : Cahier des charges

Annexe 2 : Emploi du temps

Annexe 3 : Jena Inferences support

Annexe 4 : Jena Database interface

Annexe 1

Cahier des charges

Favre Frédéric
HES-SO Valais, décembre 2007

Travail de diplôme

Web 3.0 : l'interrogation intelligente

Cahier des charges

Favre Frédéric

1. TABLE DES MATIÈRE

2. SUJET DU TRAVAIL

Le but de ce travail de diplôme est de mettre en place un moteur d'inférence capable d'effectuer des déductions sur une grande quantité de données et de l'intégrer dans le moteur OntoMea. Ce moteur gère des ontologies créées selon les normes du web sémantique. Il est nécessaire de prendre en compte la faisabilité technologique et les conséquences sur les performances d'un système de déductions sur de grandes quantités de données.

3. CADRES ET CONTRAINTES

3.1 Web sémantique

Le Web sémantique désigne un ensemble de technologies visant à rendre le contenu des ressources du World Wide Web accessible et utilisable par les programmes et agents logiciels, grâce à un système de métadonnées formelles, utilisant notamment la famille de langages développés par le W3C. (source wikipedia)

3.2 Memoria Mea

Memoria Mea est un projet de gestion d'informations personnelles. Il a pour but de permettre à l'utilisateur de gérer tout type d'information multimédia numérique.

Le module OntoMea est un sous-module de Memoria-Mea. Il a apporte au projet la possibilité d'effectuer des recherches sémantiques sur les données personnelles selon un modèle d'ontologie prédéfinies.

3.3 OntoMea

Actuellement la solution existante se compose d'un moteur java (OntoMea) basé sur la libraire Jena ainsi que différents clients de test écrits en .NET.

Le moteur est capable de traiter des ontologies stockées en local dans des fichiers .rdf et comporte un accès Web permettant à diverses applications d'appeler des services effectuant des requêtes sparql.

Le langage SPARQL définit la syntaxe et la sémantique nécessaire à l'expression de requêtes sur une base de données de type RDF et la forme possible des résultats. (source wikipedia)

3.4 Inférences

Un moteur d'inférence permet d'effectuer des déductions selon certaines règles préétablies. Ces règles peuvent être exprimées dans différents langages et sont interprétées par un raisonneur.

La libraire Jena comporte un moteur d'inférences paramétrable et offre la possibilité d'inclure un raisonneur externe via une interface DIG.

3.5 Problématique

L'intégration d'un moteur d'inférences pose plusieurs difficultés techniques :

- Comment stocker les données ?
    - Plusieurs solutions existent (DB, Fichiers, Triple Store, …).
    - Quelle solution est la plus appropriée ?
- Comment gérer la mise à jour des données ?
    - Les données proviennent de plusieurs sources différentes.
- Quels sont les moteurs d'inférences efficaces à l'heure actuelle ?
- Comment intégrer de tels moteurs dans le moteur OntoMea ?

4. Travail à réaliser

4.1 Déroulement du travail

Le projet est découpé en trois étapes distinctes :

- Une phase d'analyse qui consiste à étudier les différentes solutions disponibles et définir celles qui semble les plus appropriées dans le cadres du projet. L'analyse porte essentiellement sur les systèmes de stockage des données et les moteurs d'inférences.
- Une phase de développement qui consiste à intégrer les différents composants dans la solution actuelle.

- Une phase de test dans laquelle des données extérieur (Points Of Interests) seront exporté d'une base de données relationnelles et intégré dans le moteur OntoMea.

4.2 Moteur OntoMea

Le serveur java est installé sur les postes de chaque utilisateur du système. Il est basé sur la librairie Jena qui lit les schémas des ontologies et les données.

Chaque utilisateur dispose de ses données personnelles, le serveur doit pouvoir les utiliser ainsi que d'autres données réparties sur le web.

De nombreux modules peuvent être installés en même temps que le serveur java. Pour effectuer des requêtes spécifiques le serveur dispose de divers services effectuant des requêtes sparql sur les données et retournant le résultat sparql (en format Xml).

Inférence :

- Analyse, tests et choix d'un moteur d'inférence.
- Mise en place d'un moteur d'inférences.
- Définir le format des règles de déductions selon les standards et la faisabilité technologique.
- Créer des règles de déductions.

Stockage et gestion des données :

- Gérer le stockage des données (fichiers, MySQL, HSQLDB, …) en tenant compte de la faisabilité technologique et des performances.
- Gérer la mise à jour des données en tenant compte des contraintes du web sémantique.
- Attribuer de façon efficace des URI aux différentes ressources.

Nice To Have :

- Analyser la couche de communication entre le serveur OntoMea et les différents modules et implémenter au besoin la meilleure solution (Web Services, Semantic Web Services, …)
- Inclure la possibilité d'ajouter des données provenant du Web (Points of Interests)
- Affecter des tests de montée en charge avec des banques de données conséquentes (GeoNames, …)

4.3 Modèle de données

Les données sont structurées selon les règles établies pour le web sémantique.

Le modèle utilisé par le serveur OntoMea se base sur des ontologies déjà existantes (foaf, geoNames) afin de permettre une intégration facile de diverses données dans le système.

Must Have :

- Rechercher les ontologies existantes et les réutiliser si possible (POI).
- Définir le type des données nécessaires (Profil, Localité, Document multimédia, POI) et les inclure dans le modèle.

4.4 Intégration d'applications externes

Afin de démontrer la flexibilité des ontologies grâce au web sémantique, des données externes sont intégrées dans le système.

Ces données concernant les POI d'un utilisateur sont stockées dans une base de données SQLSERVER, un service est nécessaire afin de les transformer en données au format XML valides et utilisables par le système.

Ces données peuvent par la suite être utilisées par les différents modules de Mémoria-Mea.

Must Have

- Créer un modèle ajoutant les POI au modèle de base.
- Développer un service de conversion de données SqlServer-> Rdf
- Utiliser ses nouvelles données sur un client mobile.

Annexe 2

Emploi du temps

Favre Frédéric
HES-SO Valais, décembre 2007

DATE	ETAPE	DETAIL
17.09.2007	Analyse	Prise de connaissance du TD
18.09.2007		Recherche sur le Web de documentation sur les diverses technologies : Web Sémantique, OWL, RDF, Inférences, …
19.09.2007		Recherche sur le Web de documentation sur les diverses technologies : Web Sémantique, OWL, RDF, Inférences, …
20.09.2007		Familiarisation avec le Web sémantique avec le logiciel Protégé
21.09.2007		Familiarisation avec le Web sémantique avec le logiciel Protégé
22.09.2007		
23.09.2007		
24.09.2007		Test de la solution actuelle de OntoMea
25.09.2007		Ebauche du cahier des charges en se basant sur la solution actuelle
26.09.2007	Rédaction du cahier des charges	Analyse des besoins
27.09.2007		Rédaction
28.09.2007		Corrections
29.09.2007		
30.09.2007		
01.10.2007	Test des solutions existantes	Recherche des solutions existantes
02.10.2007		Recherche des solutions existantes
03.10.2007		Tests d'implémentation Jena HSQLDB
04.10.2007		Documentation HSQLDB
05.10.2007		Tests d'implémentation de Mulgara
06.10.2007		
07.10.2007		
08.10.2007		Tests d'implémentation de Mulgara
09.10.2007		Documentation Mulgara
10.10.2007		Tests d'implémentation de Jena SDB HSQLDB et SQLSERVER
11.10.2007		Tests d'implémentation de Jena SDB HSQLDB et SQLSERVER
12.10.2007		Documentation comparatif des outils, ajout de SDB et Jena
13.10.2007		
14.10.2007		
15.10.2007		Tests de performances des diverses solutions + tableau comparatif

16.10.2007		Analyse des moteurs d'inférences de Mulgara et Jena
17.10.2007		Documentation comparatif des outils, ajout des inférences
18.10.2007	Implémentation des données persistantes	Modification de la solution OntoMea pour permettre l'implémentation d'un solution persistant, Héritage, Polymorphisme sur KBOntoMea (Classe KBOntoMeaDB)
19.10.2007		Implémentation de la solutions Jena+SDB+HSQLDB, création de la classe StorageOntoMea
20.10.2007		
21.10.2007		
22.10.2007		Ajout de la gestion des Named Graphs, Modification des méthode abstraites héritées de KBOntoMea
23.10.2007		Gestion de la mise à jour des données via les Named Graphes. Regénération automatique des modèles de base et de schémas
24.10.2007		Ajout de l'interface graphique, Frame de base, log
25.10.2007		Ajout de l'interface graphique, visionnage des modèles
26.10.2007		Amélioration de l'algorythme de sauvegarde et de régéneration, ajout et modification divers
27.10.2007		
28.10.2007		
29.10.2007		Correction des divers bugs et améloration du code
30.10.2007	Implémentation du moteur d'inférences	Modification de fichiers de configuration, ajout des données XML pour les moteurs d'inferences, création de la classe KBReasoner
31.10.2007		Implémentation des moteurs d'inférences OWL, RDFS, Générique et DIG + Test
01.11.2007		
02.11.2007		Implémentation des moteurs d'inférences OWL, RDFS, Générique et DIG + Test
03.11.2007		
04.11.2007		
05.11.2007		Implémentation des moteurs d'inférences OWL, RDFS, Générique et DIG + Test

06.11.2007		Ajout des options d'inférences dans l'interface graphique
07.11.2007		Correction des divers bugs et améloration du code
08.11.2007		Documentation sur la création d'ontologies et sur les inférences via Protégé, Démonstration du moteur OntoMea
09.11.2007		Correction de divers fonctionnalité du moteur OntoMea
10.11.2007		
11.11.2007		
12.11.2007	Implémentation de l'interface des POIs	Création de la solution C# .NET, création d'un schéma de POIs
13.11.2007		Implémentation des communication entre OntoMea et le WebService Merkuris, Transofmationdes données SQLSERVER en données RDF
14.11.2007		Chargement des données RDF des POIs sur l'application .NET. Affichage de la carte MapPoint et des Pois
15.11.2007		Ajoute d'options de sélection des POIs utilisant des règles d'inférences RDFS et OWL
16.11.2007		Affichage du détails des POIs, chargement des données Wikipedia
17.11.2007		
18.11.2007		
19.11.2007		Affichage des données WikiPedia
20.11.2007		Ajout de règles génériques et utilisation de ces règles dans l'inferface
21.11.2007		
22.11.2007	Tests	Débuggage Serveur OntoMea
23.11.2007		Débuggage .NET
24.11.2007		
25.11.2007		
26.11.2007	Documentation	Documentation technique OntoMea
27.11.2007		Documentation technique OntoMea
28.11.2007		Documentation UI Points of interest
29.11.2007		Documentation UI Points of interest
30.11.2007		Intruduction

01.12.2007		
02.12.2007		
03.12.2007		Documentation théorie sur Web sémantique
04.12.2007		Ajout du comparatif des performances + finition documentation d'analyse
05.12.2007		Correction et relecture
06.12.2007		
07.12.2007		
08.12.2007		
09.12.2007		
10.12.2007	Remise du travail de diplôme	

Annexe 3

Jena Inferences support

Favre Frédéric
HES-SO Valais, décembre 2007

Jena 2 Inference support

This section of the documentation describes the current support for inference available within Jena2. It includes an outline of the general inference API, together with details of the specific rule engines and configurations for RDFS and OWL inference supplied with Jena2.

Not all of the fine details of the API are covered here: refer to the Jena2 Javadoc to get the full details of the capabilities of the API.

Note that this is a preliminary version of this document, some errors or inconsistencies are possible, and feedback to the author (via the jena-dev support list) is welcomed.

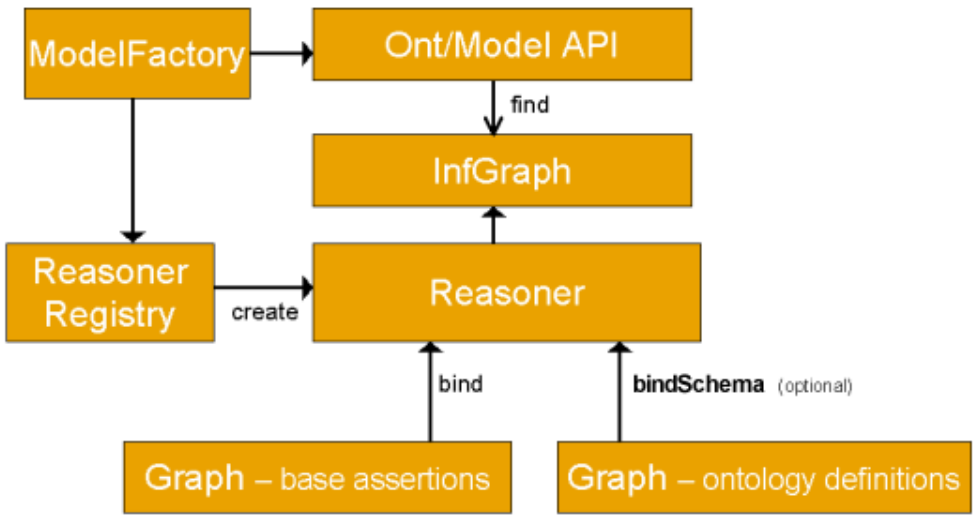
Index

Overview of inference support

The Jena2 inference subsystem is designed to allow a range of inference engines or reasoners to be plugged into Jena. Such engines are used to derive additional RDF assertions which are entailed from some base RDF together with any optional ontology information and the axioms and rules associated with the reasoner. The primary use of this mechanism is to support the use of languages such as RDFS and OWL which allow additional facts to be inferred from instance data and class descriptions. However, the machinery is designed to be quite general and, in particular, it includes a generic rule engine that can be used for many RDF processing or transformation tasks.

We will try to use the term *inference* to refer to the abstract process of deriving additional information and the term *reasoner* to refer to a specific code object that performs this task. Such usage is arbitrary and if we slip into using equivalent terms like *reasoning* and *inference engine* please forgive us.

The overall structure of the inference machinery is illustrated below.



Applications normally access the inference machinery by using the `ModelFactory` to associate a data set with some reasoner to create a new Model. Queries to the created model will return not only those statements that were present in the original data but also additional statements than can be derived from the data using the rules or other inference mechanisms implemented by the reasoner.

As illustrated the inference machinery is actually implemented at the level of the Graph SPI, so that any of the different Model interfaces can be constructed around an inference Graph. In particular, the Ontology API provides convenient ways to link appropriate reasoners into the OntModels that it constructs. As part of the general RDF API we also provide an InfModel, this is an extension to the normal Model interface that provides additional control and access to an underlying inference graph.

The reasoner API supports the notion of specializing a reasoner by binding it to a set of schema or ontology data using the bindSchema call. The specialized reasoner can then be attached to different sets of instance data using bind calls. In situations where the same schema information is to be used multiple times with different sets of instance data then this technique allows for some reuse of inferences across the different uses of the schema. In RDF there is no strong separation between schema (aka Ontology AKA tbox) data and instance (AKA abox) data and so any data, whether class or instance related, can be included in either the bind or bindSchema calls - the names are suggestive rather than restrictive.

To keep the design as open ended as possible Jena2 also includes a ReasonerRegistry. This is a static class though which the set of reasoners currently available can be examined. It is possible to register new reasoner types and to dynamically search for reasoners of a given type. The ReasonerRegistry also provides convenient access to prebuilt instances of the main supplied reasoners.

Available reasoners

Included in the Jena distribution are a number of predefined reasoners:

Transitive reasoner
: Provides support for storing and traversing class and property lattices. This implements just the *transitive* and *symmetric* properties of rdfs:subPropertyOf and rdfs:subClassOf.

RDFS rule reasoner
: Implements a configurable subset of the RDFS entailments.

OWL, OWL Mini, OWL Micro Reasoners
: A set of useful but incomplete implementation of the OWL/Lite subset of the OWL/Full language.

DAML micro reasoner
: Used internally to enable the legacy DAML API to provide minimal (RDFS scale) inferencing.

Generic rule reasoner
: A rule based reasoner that supports user defined rules. Forward chaining, tabled backward chaining and hybrid execution strategies are supported.

[index]

The Inference API

Generic reasoner API

Finding a reasoner

For each type of reasoner there is a factory class (which conforms to the interface ReasonerFactory) an instance of which can be used to create instances of the associated Reasoner. The factory instances can be located by going directly to a known factory class and using the static theInstance() method or by retrieval from a global ReasonerRegistry which stores factory instances indexed by URI assigned to the reasoner.

In addition, there are convenience methods on the ReasonerRegistry for locating a prebuilt instance of each of the main reasoners (getTransitiveReasoner, getRDFSReasoner, getRDFSSimpleReasoner,

`getOWLReasoner`, `getOWLMiniReasoner`, `getOWLMicroReasoner`).

Note that the factory objects for constructing reasoners are just there to simplify the design and extension of the registry service. Once you have a reasoner instance, the same instance can reused multiple times by binding it to different datasets, without risk of interference - there is no need to create a new reasoner instance each time.

If working with the Ontology API it is not always necessary to explicitly locate a reasoner. The prebuilt instances of OntModelSpec provide easy access to the appropriate reasoners to use for different Ontology configurations.

Similarly, if all you want is a plain RDF Model with RDFS inference included then the convenience methods `ModelFactory.createRDFSModel` can be used.

Configuring a reasoner

The behaviour of many of the reasoners can be configured. To allow arbitrary configuration information to be passed to reasoners we use RDF to encode the configuration details. The `ReasonerFactory.create` method can be passed a Jena `Resource` object, the properties of that object will be used to configure the created reasoner.

To simplify the code required for simple cases we also provide a direct Java method to set a single configuration parameter, `Reasoner.setParameter`. The parameter being set is identified by the corresponding configuration property.

For the built in reasoners the available configuration parameters are described below and are predefined in the `ReasonerVocabulary` class.

The parameter value can normally be a String or a structured value. For example, to set a boolean value one can use the strings "true" or "false", or in Java use a Boolean object or in RDF use an instance of xsd:Boolean

Applying a reasoner to data

Once you have an instance of a reasoner it can then be attached to a set of RDF data to create an inference model. This can either be done by putting all the RDF data into one Model or by separating into two components - schema and instance data. For some external reasoners a hard separation may be required. For all of the built in reasoners the separation is arbitrary. The prime value of this separation is to allow some deductions from one set of data (typically some schema definitions) to be efficiently applied to several subsidiary sets of data (typically sets of instance data).

If you want to specialize the reasoner this way, by partially-applying it to a set schema data, use the `Reasoner.bindSchema` method which returns a new, specialized, reasoner.

To bind the reasoner to the final data set to create an inference model see the `ModelFactory` methods, particularly `ModelFactory.createInfModel`.

Accessing inferences

Finally, having created a inference model then any API operations which access RDF statements will be able to access additional statements which are entailed from the bound data by means of the reasoner. Depending on the reasoner these additional *virtual* statements may all be precomputed the first time the model is touched, may be dynamically recomputed each time or may be computed on-demand but cached.

Reasoner description

The reasoners can be described using RDF metadata which can be searched to locate reasoners with appropriate properties. The calls `Reasoner.getCapabilities` and `Reasoner.supportsProperty` are used to access this descriptive metadata.

[API index] [main index]

Some small examples

These initial examples are not designed to illustrate the power of the reasoners but to illustrate the

code required to set one up.

Let us first create a Jena model containing the statements that some property "p" is a subproperty of another property "q" and that we have a resource "a" with value "foo" for "p". This could be done by writing an RDF/XML or N3 file and reading that in but we have chosen to use the RDF API:

```
String NS = "urn:x-hp-jena:eg/";

// Build a trivial example data set
Model rdfsExample = ModelFactory.createDefaultModel();
Property p = rdfsExample.createProperty(NS, "p");
Property q = rdfsExample.createProperty(NS, "q");
rdfsExample.add(p, RDFS.subPropertyOf, q);
rdfsExample.createResource(NS+"a").addProperty(p, "foo");
```

Now we can create an inference model which performs RDFS inference over this data by using:

```
InfModel inf = ModelFactory.createRDFSModel(rdfsExample);  // [1]
```

We can then check that resulting model shows that "a" also has property "q" of value "foo" by virtue of the subPropertyOf entailment:

```
Resource a = inf.getResource(NS+"a");
System.out.println("Statement: " + a.getProperty(q));
```

Which prints the output:

```
Statement: [urn:x-hp-jena:eg/a, urn:x-hp-jena:eg/q, Literal]
```

Alternatively we could have created an empty inference model and then added in the statements directly to that model.

If we wanted to use a different reasoner which is not available as a convenience method or wanted to configure one we would change line [1]. For example, to create the same set up manually we could replace [1] by:

```
Reasoner reasoner = ReasonerRegistry.getRDFSReasoner();

InfModel inf = ModelFactory.createInfModel(reasoner, rdfsExample);
```

or even more manually by

```
Reasoner reasoner = RDFSRuleReasonerFactory.theInstance().create(null);
InfModel inf = ModelFactory.createInfModel(reasoner, rdfsExample);
```

The purpose of creating a new reasoner instance like this variant would be to enable configuration parameters to be set. For example, if we were to listStatements on inf Model we would see that it also "includes" all the RDFS axioms, of which there are quite a lot. It is sometimes useful to suppress these and only see the "interesting" entailments. This can be done by setting the processing level parameter by creating a description of a new reasoner configuration and passing that to the factory method:

```
Resource config = ModelFactory.createDefaultModel()
                  .createResource()
                  .addProperty(ReasonerVocabulary.PROPsetRDFSLevel, "simple");
Reasoner reasoner = RDFSRuleReasonerFactory.theInstance().create(config);
InfModel inf = ModelFactory.createInfModel(reasoner, rdfsExample);
```

This is a rather long winded way of setting a single parameter, though it can be useful in the cases where you want to store this sort of configuration information in a separate (RDF) configuration file. For hardwired cases the following alternative is often simpler:

```
Reasoner reasoner = RDFSRuleReasonerFactory.theInstance()Create(null);
reasoner.setParameter(ReasonerVocabulary.PROPsetRDFSLevel,
                      ReasonerVocabulary.RDFS_SIMPLE);
InfModel inf = ModelFactory.createInfModel(reasoner, rdfsExample);
```

Finally, supposing you have a more complex set of schema information, defined in a Model called *schema,* and you want to apply this schema to several sets of instance data without redoing too many of the same intermediate deductions. This can be done by using the SPI level methods:

```
Reasoner boundReasoner = reasoner.bindSchema(schema);
InfModel inf = ModelFactory.createInfModel(boundReasoner, data);
```

This creates a new reasoner, independent from the original, which contains the schema data. Any queries to an InfModel created using the boundReasoner will see the schema statements, the data statements and any statements entailed from the combination of the two. Any updates to the InfModel will be reflected in updates to the underlying data model - the schema model will not be affected.

[API index] [main index]

Operations on inference models

For many applications one simply creates a model incorporating some inference step, using the `ModelFactory` methods, and then just works within the standard Jena Model API to access the entailed statements. However, sometimes it is necessary to gain more control over the processing or to access additional reasoner features not available as *virtual* triples.

Validation

The most common reasoner operation which can't be exposed through additional triples in the inference model is that of validation. Typically the ontology languages used with the semantic web allow constraints to be expressed, the validation interface is used to detect when such constraints are violated by some data set.

A simple but typical example is that of datatype ranges in RDFS. RDFS allows us to specify the range of a property as lying within the value space of some datatype. If an RDF statement asserts an object value for that property which lies outside the given value space there is an inconsistency.

To test for inconsistencies with a data set using a reasoner we use the `InfModel.validate()` interface. This performs a global check across the schema and instance data looking for inconsistencies. The result is a `ValidityReport` object which comprises a simple pass/fail flag (`ValidityReport.isValid()`) together with a list of specific reports (instances of the `ValidityReport.Report` interface) which detail any detected inconsistencies. At a minimum the individual reports should be printable descriptions of the problem but they can also contain an arbitrary reasoner-specific object which can be used to pass additional information which can be used for programmatic handling of the violations.

For example, to check a data set and list any problems one could do something like:

```
Model data = FileManager.get().loadModel(fname);
InfModel infmodel = ModelFactory.createRDFSModel(data);
ValidityReport validity = infmodel.validate();
if (validity.isValid()) {
    System.out.println("OK");
} else {
    System.out.println("Conflicts");
    for (Iterator i = validity.getReports(); i.hasNext(); ) {
        System.out.println(" - " + i.next());
    }
}
```

The file `testing/reasoners/rdfs/dttest2.nt` declares a property bar with range `xsd:integer` and attaches a bar value to some resource with the value `"25.5"^^xsd:decimal`. If we run the above sample code on this file we see:

> *Conflicts*
> *- Error (dtRange): Property http://www.hpl.hp.com/semweb/2003/eg#bar has a typed range Datatype[http://www.w3.org/2001/XMLSchema#integer -> class java.math.BigInteger]that is not compatible with 25.5:http://www.w3.org/2001/XMLSchema#decimal*

Whereas the file `testing/reasoners/rdfs/dttest3.nt` uses the value `"25"^^xsd:decimal` instead, which is a valid integer and so passes.

Note that the individual validation records can include warnings as well as errors. A warning does not affect the overall `isValid()` status but may indicate some issue the application may wish to be aware

of. For example, it would be possible to develop a modification to the RDFS reasoner which warned about use of a property on a resource that is not explicitly declared to have the type of the domain of the property.

A particular case of this arises in the case of OWL. In the Description Logic community a class which cannot have an instance is regarded as "inconsistent". That term is used because it generally arises from an error in the ontology. However, it is not a logical inconsistency - i.e. something giving rise to a contradiction. Having an instance of such a class is, clearly a logical error. In the Jena 2.2 release we clarified the semantics of `isValid()`. An ontology which is logically consistent but contains empty classes is regarded as valid (that is `isValid()` is false only if there is a logical inconsistency). Class expressions which cannot be instantiated are treated as warnings rather than errors. To make it easier to test for this case there is an additional method `Report.isClean()` which returns true if the ontology is both valid (logically consistent) and generated no warnings (such as inconsistent classes).

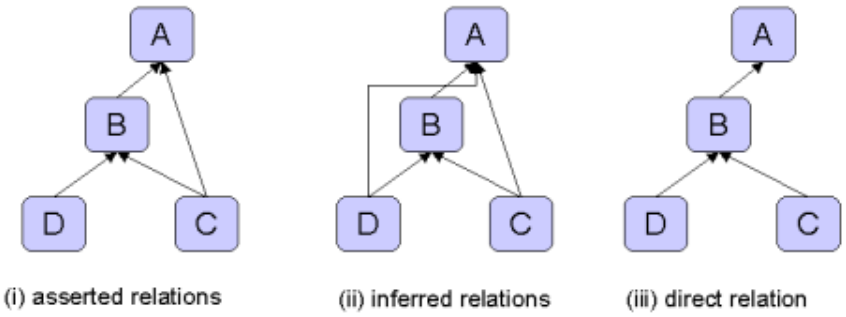
Extended list statements

The default API supports accessing all entailed information at the level of individual triples. This is surprisingly flexible but there are queries which cannot be easily supported this way. The first such is when the query needs to make reference to an expression which is not already present in the data. For example, in description logic systems it is often possible to ask if there are any instances of some class expression. Whereas using the triple-based approach we can only ask if there are any instances of some class already defined (though it could be defined by a bNode rather than be explicitly named).

To overcome this limitation the `InfModel` API supports a notion of "posit", that is a set of assertions which can be used to temporarily declare new information such as the definition of some class expression. These temporary assertions can then be referenced by the other arguments to the `listStatements` command. With the current reasoners this is an expensive operation, involving the temporary creation of an entire new model with the additional posits added and all inference has to start again from scratch. Thus it is worth considering preloading your data with expressions you might need to query over. However, for some external reasoners, especially description logic reasoners, we anticipate restricted uses of this form of listStatement will be important.

Direct and indirect relationships

The second type of operation that is not obviously convenient at the triple level involves distinguishing between direct and indirect relationships. If a relation is transitive, for example rdfs:subClassOf, then we can define the notion of the *minimal* or *direct* form of the relationship from which all other values of the relation can be derived by transitive closure.



Normally, when an InfGraph is queried for a transitive relation the results returned show the inferred relations, i.e. the full transitive closure (all the links (ii) in the illustration). However, in some cases, such when as building a hierarchical UI widget to represent the graph, it is more convenient to only see the direct relations (iii). This is achieved by defining special *direct* aliases for those relations which can be queried this way. For the built in reasoners this functionality is available for `rdfs:subClassOf` and `rdfs:subPropertyOf` and the direct aliases for these are defined in `ReasonerVocabulary`.

Typically the easiest way to work with such indirect and direct relations is to use the Ontology API which hides the grubby details of these property aliases.

Derivations

It is sometimes useful to be able to trace where an inferred statement was generated from. This is

achieved using the `InfModel.getDerivation(Statement)` method. This returns a iterator over a set `Derviation` objects through which a brief description of the source of the derivation can be obtained. Typically understanding this involves tracing the sources for other statements which were used in this derivation and the `Derivation.PrintTrace` method is used to do this recursively.

The general form of the Derivation objects is quite abstract but in the case of the rule-based reasoners they have a more detailed internal structure that can be accessed - see `RuleDerivation`.

Derviation information is rather expensive to compute and store. For this reason, it is not recorded by default and `InfModel.serDerivationLogging(true)` must be used to enable derivations to be recorded. This should be called before any queries are made to the inference model.

As an illustration suppose that we have a raw data model which asserts three triples:

```
eg:A eg:p eg:B .
eg:B eg:p eg:C .
eg:C eg:p eg:D .
```

and suppose that we have a trivial rule set which computes the transitive closure over relation eg:p

```
String rules = "[rule1: (?a eg:p ?b) (?b eg:p ?c) -> (?a eg:p ?c)]";
Reasoner reasoner = new GenericRuleReasoner(Rule.parseRules(rules));
reasoner.setDerivationLogging(true);
InfModel inf = ModelFactory.createInfModel(reasoner, rawData);
```

Then we can query whether eg:A is related through eg:p to eg:D and list the derivation route using the following code fragment:

```
PrintWriter out = new PrintWriter(System.out);
for (StmtIterator i = inf.listStatements(A, p, D); i.hasNext(); ) {
    Statement s = i.nextStatement();
    System.out.println("Statement is " + s);
    for (Iterator id = inf.getDerivation(s); id.hasNext(); ) {
        Derivation deriv = (Derivation) id.next();
        deriv.printTrace(out, true);
    }
}
out.flush();
```

Which generates the output:

```
Statement is [urn:x-hp:eg/A, urn:x-hp:eg/p, urn:x-hp:eg/D]
Rule rule1 concluded (eg:A eg:p eg:D) <-
    Fact (eg:A eg:p eg:B)
    Rule rule1 concluded (eg:B eg:p eg:D) <-
        Fact (eg:B eg:p eg:C)
        Fact (eg:C eg:p eg:D)
```

Accessing raw data and deductions

From an `InfModel` it is easy to retrieve the original, unchanged, data over which the model has been computed using the `getRawModel()` call. This returns a model equivalent to the one used in the initial `bind` call. It might not be the same Java object but it uses the same Java object to hold the underlying data graph.

Some reasoners, notably the forward chaining rule engine, store the deduced statements in a concrete form and this set of deductions can be obtained separately by using the `getDeductionsModel()` call.

Processing control

Having bound a `Model` into an `InfModel` by using a `Reasoner` its content can still be changed by the normal `add` and `remove` calls to the `InfModel`. Any such change the model will usually cause all current deductions and temporary rules to be discarded and inference will start again from scratch at the next query. Some reasoners, such as the RETE-based forward rule engine, can work incrementally.

In the non-incremental case then the processing will not be started until a query is made. In that way a sequence of add and removes can be undertaken without redundant work being performed at each change. In some applications it can be convenient to trigger the initial processing ahead of time to

reduce the latency of the first query. This can be achieved using the `InfModel.prepare()` call. This call is not necessary in other cases, any query will automatically trigger an internal prepare phase if one is required.

There are times when the data in a model bound into an InfModel can is changed "behind the scenes" instead of through calls to the InfModel. If this occurs the result of future queries to the InfModel are unpredictable. To overcome this and force the InfModel to reconsult the raw data use the `InfModel.rebind()` call.

Finally, some reasoners can store both intermediate and final query results between calls. This can substantially reduce the cost of working with the inference services but at the expense of memory usage. It is possible to force an InfModel to discard all such cached state by using the `InfModel.reset()` call. It there are any outstanding queries (i.e. StmtIterators which have not been read to the end yet) then those will be aborted (the next hasNext() call will return false).

Tracing

When developing new reasoner configurations, especially new rule sets for the rule engines, it is sometimes useful to be able to trace the operations of the associated inference engine. Though, often this generates too much information to be of use and selective use of the `print` builtin can be more effective.

Tracing is not supported by a convenience API call but, for those reasoners that support it, it can be enabled using:

```
reasoner.setParameter(ReasonerVocabulary.PROPtraceOn, Boolean.TRUE);
```

Dynamic tracing control is sometimes possible on the InfModel itself by retrieving its underlying InfGraph and calling `setTraceOn()` call. If you need to make use of this see the full javadoc for the relevant InfGraph implementation.

[API index] [main index]

The RDFS reasoner

RDFS reasoner - intro and coverage

Jena2 includes an RDFS reasoner (`RDFSRuleReasoner`) which supports almost all of the RDFS entailments described by the RDF Core working group [RDF Semantics]. The only omissions are deliberate and are described below.

This reasoner is accessed using `ModelFactory.createRDFSModel` or manually via `ReasonerRegistery.getRDFSReasoner()`.

During the preview phases of Jena2 experimental RDFS reasoners were released, some of which are still included in the code base for now but applications should not rely on their stability or continued existence.

When configured in *full* mode (see below for configuration information) then the RDFS reasoner implements all RDFS entailments except for the bNode closure rules. These closure rules imply, for example, that for all triples of the form:

```
eg:a eg:p nnn^^datatype .
```

we should introduce the corresponding blank nodes:

```
eg:a eg:p _:anon1 .
_:anon1 rdf:type datatype .
```

Whilst such rules are both correct and necessary to reduce RDF datatype entailment down to simple entailment they are not useful in implementation terms. In Jena simple entailment can be

implemented by translating a graph containing bNodes to an equivalent query containing variables in place of the bNodes. Such a query is can directly match the literal node and the RDF API can be used to extract the datatype of the literal. The value to applications of directly seeing the additional bNode triples, even in *virtual* triple form, is negligible and so this has been deliberately omitted from the reasoner.

[RDFS index] [main index]

RDFS configuration

The RDFSRuleReasoner can be configured to work at three different compliance levels:

Full
: This implements all of the RDFS axioms and closure rules with the exception of bNode entailments and datatypes (rdfD 1). See above for comments on these. This is an expensive mode because all statements in the data graph need to be checked for possible use of container membership properties. It also generates type assertions for all resources and properties mentioned in the data (rdf1, rdfs4a, rdfs4b).

Default
: This omits the expensive checks for container membership properties and the "everything is a resource" and "everything used as a property is one" rules (rdf1, rdfs4a, rdfs4b). The latter information is available through the Jena API and creating virtual triples to this effect has little practical value.
This mode does include all the axiomatic rules. Thus, for example, even querying an "empty" RDFS InfModel will return triples such as `[rdf:type rdfs:range rdfs:Class]`.

Simple
: This implements just the transitive closure of subPropertyOf and subClassOf relations, the domain and range entailments and the implications of subPropertyOf and subClassOf. It omits all of the axioms. This is probably the most useful mode but is not the default because it is a less complete implementation of the standard.

The level can be set using the `setParameter` call, e.g.

```
reasoner.setParameter(ReasonerVocabulary.PROPsetRDFSLevel,
                      ReasonerVocabulary.RDFS_SIMPLE);
```

or by constructing an RDF configuration description and passing that to the RDFSRuleReasonerFactory e.g.

```
Resource config = ModelFactory.createDefaultModel()
                  .createResource()
                  .addProperty(ReasonerVocabulary.PROPsetRDFSLevel, "simple");
Reasoner reasoner = RDFSRuleReasonerFactory.theInstance()Create(config);
```

Summary of parameters

Parameter	Values	Description
PROPsetRDFSLevel	"full", "default", "simple"	Sets the RDFS processing level as described above.
PROPenableCMPScan	Boolean	If true forces a preprocessing pass which finds all usages of rdf:_n properties and declares them as ContainerMembershipProperties. This is implied by setting the level parameter to "full" and is not normally used directly.
PROPtraceOn	Boolean	If true switches on exhaustive tracing of rule executions to the log4j *info* appender.
PROPderivationLogging	Boolean	If true causes derivation routes to be recorded internally so that future getDerivation calls can return useful information.

[RDFS index] [main index]

RDFS Example

As a complete worked example let us create a simple RDFS schema, some instance data and use an instance of the RDFS reasoner to query the two.

We shall use a trivial schema:

```
<rdf:Description rdf:about="&eg;mum">
  <rdfs:subPropertyOf rdf:resource="&eg;parent"/>
</rdf:Description>

<rdf:Description rdf:about="&eg;parent">
  <rdfs:range  rdf:resource="&eg;Person"/>
  <rdfs:domain rdf:resource="&eg;Person"/>
</rdf:Description>

<rdf:Description rdf:about="&eg;age">
  <rdfs:range rdf:resource="&xsd;integer" />
</rdf:Description>
```

This defines a property `parent` from `Person` to `Person`, a sub-property `mum` of `parent` and an integer-valued property `age`.

We shall also use the even simpler instance file:

```
<Teenager rdf:about="&eg;colin">
    <mum rdf:resource="&eg;rosy" />
    <age>13</age>
</Teenager>
```

Which defines a `Teenager` called `colin` who has a `mum` `rosy` and an `age` of 13.

Then the following code fragment can be used to read files containing these definitions, create an inference model and query it for information on the `rdf:type` of `colin` and the `rdf:type` of `Person`:

```
Model schema = FileManager.get().loadModel("file:data/rdfsDemoSchema.rdf");
Model data = FileManager.get().loadModel("file:data/rdfsDemoData.rdf");
InfModel infmodel = ModelFactory.createRDFSModel(schema, data);

Resource colin = infmodel.getResource("urn:x-hp:eg/colin");
System.out.println("colin has types:");
printStatements(infmodel, colin, RDF.type, null);

Resource Person = infmodel.getResource("urn:x-hp:eg/Person");
System.out.println("\nPerson has types:");
printStatements(infmodel, Person, RDF.type, null);
```

This produces the output:

```
colin has types:
 - (eg:colin rdf:type eg:Teenager)
 - (eg:colin rdf:type rdfs:Resource)
 - (eg:colin rdf:type eg:Person)

Person has types:
 - (eg:Person rdf:type rdfs:Class)
 - (eg:Person rdf:type rdfs:Resource)
```

This says that `colin` is both a `Teenager` (by direct definition), a `Person` (because he has a `mum` which means he has a `parent` and the domain of `parent` is `Person`) and an `rdfs:Resource`. It also says that `Person` is an `rdfs:Class`, even though that wasn't explicitly in the schema, because it is used as object of range and domain statements.

If we add the additional code:

```
ValidityReport validity = infmodel.validate();
if (validity.isValid()) {
    System.out.println("\nOK");
} else {
```

```
        System.out.println("\nConflicts");
        for (Iterator i = validity.getReports(); i.hasNext(); ) {
            ValidityReport.Report report = (ValidityReport.Report)i.next();
            System.out.println(" - " + report);
        }
    }
```

Then we get the additional output:

```
Conflicts
 - Error (dtRange): Property urn:x-hp:eg/age has a typed range
Datatype[http://www.w3.org/2001/XMLSchema#integer -> class java.math.BigInteger]
that is not compatible with 13
```

because the age was given using an RDF plain litera where as the schema requires it to be a datatyped literal which is compatible with xsd:integer.

[RDFS index] [main index]

RDFS implementation and performance notes

The RDFSRuleReasoner is a hybrid implementation. The subproperty and subclass lattices are eagerly computed and stored in a compact in-memory form using the TransitiveReasoner (see below). The identification of which container membership properties (properties like rdf:_1) are present is implemented using a preprocessing hook. The rest of the RDFS operations are implemented by explicit rule sets executed by the general hybrid rule reasoner. The three different processing levels correspond to different rule sets. These rule sets are located by looking for files "`etc/*.rules`" on the classpath and so could, in principle, be overridden by applications wishing to modify the rules.

Performance for in-memory queries appears to be good. Using a synthetic dataset we obtain the following times to determine the extension of a class from a class hierarchy:

Set	#concepts	total instances	#instances of concept	JenaRDFS	XSB*
1	155	1550	310	0.07	0.16
2	780	7800	1560	0.25	0.47
3	3905	39050	7810	1.16	2.11

The times are in seconds, normalized to a 1.1GHz Pentium processor. The XSB* figures are taken from a pre-published paper and may not be directly comparable (for example they do not include any rule compilation time) - they are just offered to illustrate that the RDFSRuleReasoner has broadly similar scaling and performance to other rule-based implementations.

The Jena RDFS implementation has not been tested and evaluated over database models. The Jena architecture makes it easy to construct such models but in the absence of caching we would expect the performance to be poor. Future work on adapting the rule engines to exploit the capabilities of the more sophisticated database backends will be considered.

[RDFS index] [main index]

The OWL reasoner

The second major set of reasoners supplied with Jena2 is a rule-based implementation of the OWL/lite subset of OWL/full.

The current release includes a default OWL reasoner and two small/faster configurations. Each of the configurations is intended to be a sound implementation of a subset of OWL/full semantics but none of

them is complete (in the technical sense). For complete OWL DL reasoning use an external DL reasoner such as Pellet, Racer or FaCT. The jena DIG interface makes it easy to connect to any reasoner that supports the DIG standard. Performance (especially memory use) of the fuller reasoner configuration still leaves something to be desired and will the subject of future work - time permitting.

See also subsection 5 for notes on more specific limitations of the current implementation.

OWL coverage

The Jena OWL reasoners could be described as instance-based reasoners. That is, they work by using rules to propagate the if- and only-if- implications of the OWL constructs on instance data. Reasoning about classes is done indirectly - for each declared class a prototypical instance is created and elaborated. If the prototype for a class A can be deduced as being a member of class B then we conclude that A is a subClassOf B. This approach is in contrast to more sophisticated Description Logic reasoners which work with class expressions and can be less efficient when handling instance data but more efficient with complex class expressions and able to provide complete reasoning.

We thus anticipate that the OWL rule reasoner will be most suited to applications involving primarily instance reasoning with relatively simple, regular ontologies and least suited to applications involving large rich ontologies. A better characterisation of the tradeoffs involved would be useful and will be sought.

We intend that the OWL reasoners should be smooth extensions of the RDFS reasoner described above. That is all RDFS entailments found by the RDFS reasoner will also be found by the OWL reasoners and scaling on RDFS schemas should be similar (though there are some costs, see later). The instance-based implementation technique is in keeping with this "RDFS plus a bit" approach.

Another reason for choosing this inference approach is that it makes it possible to experiment with support for different constructs, including constructs that go beyond OWL, by modification of the rule set. In particular, some applications of interest to ourselves involve ontology transformation which very often implies the need to support property composition. This is something straightforward to express in rule-based form and harder to express in standard Description Logics.

Since RDFS is not a subset of the OWL/Lite or OWL/DL languages the Jena implementation is an incomplete implementation of OWL/full. We provide three implementations a default ("full" one), a slightly cut down "mini" and a rather smaller/faster "micro". The default OWL rule reasoner (`ReasonerRegistry.getOWLReasoner()`) supports the constructs as listed below. The OWLMini reasoner is nearly the same but omits the forward entailments from minCardinality/someValuesFrom restrictions - that is it avoids introducing bNodes which avoids some infinite expansions and enables it to meet the Jena API contract more precisely. The OWLMicro reasoner just supports RDFS plus the various property axioms, intersectionOf, unionOf (partial) and hasValue. It omits the cardinality restrictions and equality axioms, which enables it to achieve much higher performance.

Constructs	Supported by	Notes
rdfs:subClassOf, rdfs:subPropertyOf, rdf:type	all	Normal RDFS semantics supported including meta use (e.g. taking the subPropertyOf subClassOf).
rdfs:domain, rdfs:range	all	Stronger if-and-only-if semantics supported
owl:intersectionOf	all	
owl:unionOf	all	Partial support. If C=unionOf(A,B) then will infer that A,B are subclasses of C, and thus that instances of A or B are instances of C. Does not handle the reverse (that an instance of C must be either an instance of A or an instance of B).
owl:equivalentClass	all	
owl:disjointWith	full, mini	

owl:sameAs, owl:differentFrom, owl:distinctMembers	full, mini	owl:distinctMembers is currently translated into a quadratic set of owl:differentFrom assertions.
Owl:Thing	all	
owl:equivalentProperty, owl:inverseOf	all	
owl:FunctionalProperty, owl:InverseFunctionalProperty	all	
owl:SymmeticProperty, owl:TransitiveProperty	all	
owl:someValuesFrom	full, (mini)	Full supports both directions (existence of a value implies membership of someValuesFrom restriction, membership of someValuesFrom implies the existence of a bNode representing the value). Mini omits the latter "bNode introduction" which avoids some infinite closures.
owl:allValuesFrom	full, mini	Partial support, forward direction only (member of a allValuesFrom(p, C) implies that all p values are of type C). Does handle cases where the reverse direction is trivially true (e.g. by virtue of a global rdfs:range axiom).
owl:minCardinality, owl:maxCardinality, owl:cardinality	full, (mini)	Restricted to cardinalities of 0 or 1, though higher cardinalities are partially supported in validation for the case of literal-valued properties. Mini omits the bNodes introduction in the minCardinality(1) case, see someValuesFrom above.
owl:hasValue	all	

The critical constructs which go beyond OWL/lite and are not supported in the Jena OWL reasoner are complementOf and oneOf. As noted above the support for unionOf is partial (due to limitations of the rule based approach) but is useful for traversing class hierarchies.

Even within these constructs rule based implementations are limited in the extent to which they can handle equality reasoning - propositions provable by reasoning over concrete and introduced instances are covered but reasoning by cases is not supported.

Nevertheless, the full reasoner passes the normative OWL working group positive and negative entailment tests for the supported constructs, though some tests need modification for the comprehension axioms (see below).

The OWL rule set does include incomplete support for validation of datasets using the above constructs. Specifically, it tests for:

- Illegal existence of a property restricted by a maxCardinality(0) restriction.
- Two individuals both sameAs and differentFrom each other.
- Two classes declared as disjoint but where one subsumes the other (currently reported as a violation concerning the class prototypes, error message to be improved).
- Range or a allValuesFrom violations for DatatypeProperties.
- Too many literal-values for a DatatypeProperty restricted by a maxCardinality(N) restriction.

[OWL index] [main index]

OWL Configuration

This reasoner is accessed using `ModelFactory.createOntologyModel` with the prebuilt `OntModelSpec` `OWL_MEM_RULE_INF` or manually via `ReasonerRegistery.getOWLReasoner()`.

There are no OWL-specific configuration parameters though the reasoner supports the standard control parameters:

Parameter	Values	Description
PROPtraceOn	boolean	If true switches on exhaustive tracing of rule executions to the log4j *info* appender.
PROPderivationLogging	Boolean	If true causes derivation routes to be recorded internally so that future getDerivation calls can return useful information.

As we gain experience with the ways in which OWL is used and the capabilities of the rule-based approach we imagine useful subsets of functionality emerging - like that that supported by the RDFS reasoner in the form of the level settings.

[OWL index] [main index]

OWL Example

As an example of using the OWL inference support, consider the sample schema and data file in the data directory - owlDemoSchema.xml and owlDemoData.xml.

The schema file shows a simple, artificial ontology concerning computers which defines a GamingComputer as a Computer which includes at least one bundle of type GameBundle and a component with the value gamingGraphics.

The data file shows information on serveral hypothetical computer configurations including two different descriptions of the configurations "whiteBoxZX" and "bigName42".

We can create an instance of the OWL reasoner, specialized to the demo schema and then apply that to the demo data to obtain an inference model, as follows:

```
Model schema = FileManager.get().loadModel("file:data/owlDemoSchema.owl");
Model data = FileManager.get().loadModel("file:data/owlDemoData.rdf");
Reasoner reasoner = ReasonerRegistry.getOWLReasoner();
reasoner = reasoner.bindSchema(schema);
InfModel infmodel = ModelFactory.createInfModel(reasoner, data);
```

A typical example operation on such a model would be to find out all we know about a specific instance, for example the nForce mother board. This can be done using:

```
Resource nForce = infmodel.getResource("urn:x-hp:eg/nForce");
System.out.println("nForce *:");
printStatements(infmodel, nForce, null, null);
```

where `printStatements` is defined by:

```
public void printStatements(Model m, Resource s, Property p, Resource o) {
    for (StmtIterator i = m.listStatements(s,p,o); i.hasNext(); ) {
        Statement stmt = i.nextStatement();
        System.out.println(" - " + PrintUtil.print(stmt));
    }
}
```

This produces the output:

```
nForce *:
 - (eg:nForce rdf:type owl:Thing)
 - (eg:nForce owl:sameAs eg:unknownMB)
 - (eg:nForce owl:sameAs eg:nForce)
 - (eg:nForce rdf:type eg:MotherBoard)
 - (eg:nForce rdf:type rdfs:Resource)
```

```
- (eg:nForce rdf:type a3b24:f7822755ad:-7ffd)
- (eg:nForce eg:hasGraphics eg:gamingGraphics)
- (eg:nForce eg:hasComponent eg:gamingGraphics)
```

Note that this includes inferences based on subClass inheritance (being an `eg:MotherBoard` implies it is an `owl:Thing` and an `rdfs:Resource`), property inheritance (`eg:hasComponent eg:gameGraphics` derives from `hasGraphics` being a subProperty of `hasComponent`) and cardinality reasoning (it is the sameAs `eg:unknownMB` because computers are defined to have only one motherboard and the two diffferent descriptions of `whileBoxZX` use these two different terms for the mother board). The anonymous rdf:type statement referencesthe `"hasValue(eg:hasComponent, eg:gamingGraphics)"` restriction mentioned in the definition of GamingComputer.

A second, typical operation is instance recognition. Testing if an individual is an instance of a class expression. In this case the `whileBoxZX` is identifiable as a `GamingComputer` because it is a `Computer`, is explicitly declared as having an appropriate bundle and can be inferred to have a `gamingGraphics` component from the combination of the `nForce` inferences we've already seen and the transitivity of `hasComponent`. We can test this using:

```
Resource gamingComputer = infmodel.getResource("urn:x-hp:eg/GamingComputer");
Resource whiteBox = infmodel.getResource("urn:x-hp:eg/whiteBoxZX");
if (infmodel.contains(whiteBox, RDF.type, gamingComputer)) {
    System.out.println("White box recognized as gaming computer");
} else {
    System.out.println("Failed to recognize white box correctly");
}
```

Which gnerates the output:

```
White box recognized as gaming computer
```

Finally, we can check for inconsistencies within the data by using the validation interface:

```
ValidityReport validity = infmodel.validate();
if (validity.isValid()) {
    System.out.println("OK");
} else {
    System.out.println("Conflicts");
    for (Iterator i = validity.getReports(); i.hasNext(); ) {
        ValidityReport.Report report = (ValidityReport.Report)i.next();
        System.out.println(" - " + report);
    }
}
```

Which generates the output:

```
Conflicts
 - Error (conflict): Two individuals both same and different, may be
   due to disjoint classes or functional properties
Culprit = eg:nForce2
Implicated node: eg:bigNameSpecialMB

... + 3 other similar reports
```

This is due to the two records for the `bigName42` configuration referencing two motherboards which are explicitly defined to be different resources and thus violate the FunctionProperty nature of `hasMotherBoard`.

[OWL index] [main index]

OWL notes and limitations

Comprehension axioms

A critical implication of our variant of the instance-based approach is that the reasoner does not directly answer queries relating to dynamically introduced class expressions.

For example, given a model containing the RDF assertions corresponding to the two OWL axioms:

```
class A = intersectionOf (minCardinality(P, 1), maxCardinality(P,1))
class B = cardinality(P,1)
```

Then the reasoner can demonstrate that classes A and B are equivalent, in particular that any instance of A is an instance of B and vice versa. However, given a model just containing the first set of assertions you cannot directly query the inference model for the individual triples that make up *cardinality(P,1)*. If the relevant class expressions are not already present in your model then you need to use the list-with-posits mechanism described above, though be warned that such posits start inference afresh each time and can be expensive.

Actually, it would be possible to introduce comprehension axioms for simple cases like this example. We have, so far, chosen not to do so. First, since the OWL/full closure is generally infiite, some limitation on comprehension inferences seems to be useful. Secondly, the typical queries that Jena applications expect to be able to issue would suddenly jump in size and cost - causing a support nightmare. For example, queries such as (a, rdf:type, *) would become near-unusable.

Approximately, 10 of the OWL working group tests for the supported OWL subset currently rely on such comprehension inferences. The shipping version of the Jena rule reasoner passes these tests only after they have been rewritten to avoid the comprehension requirements.

Prototypes

As noted above the current OWL rule set introduces prototypical instances for each defined class. These prototypical instances used to be visible to queries. From release 2.1 they are used internally but should not longer be visible.

Direct / indirect

We noted above that the Jena reasoners support a separation of direct and indirect relations for transitive properties such as subClassOf. The current implementation of the full and mini OWL reasoner fails to do this and the direct forms of the queries will fail. The OWL Micro reasoner, which is but a small extension of RDFS, does suppor the direct queries.

This does not affect querying though the Ontology API, which works around this limitation. It only affects direct RDF accesses to the inference model.

Performance

The OWL reasoners use the rule engines for all inference. The full and mini configurations omit some of the performance tricks employed by the RDFS reasoner (notably the use of the custom transitive reasoner) making those OWL reasoner configurations slower than the RDFS reasoner on pure RDFS data (typically around x3-4 slow down). The OWL Micro reasoner is intended to be as close to RDFS performance while also supporting the core OWL constructs as described earlier.

Once the owl constructs are used then substantial reasoning can be required. The most expensive aspect of the supported constructs is the equality reasoning implied by use of cardinality restrictions and FunctionalProperties. The current rule set implements equality reasoning by identifying all sameAs deductions during the initial forward "prepare" phase. This may require the entire instance dataset to be touched several times searching for occurrences of FunctionalProperties.

Beyond this the rules implementing the OWL constructs can interact in complex ways leading to serious performance overheads for complex ontologies. Characterising the sorts of ontologies and inference problems that are well tackled by this sort of implementation and those best handled by plugging a Description Logic engine, or a saturation theorem prover, into Jena is a topic for future work.

One random hint: explicitly importing the owl.owl definitions causes much duplication of rule use and a substantial slow down - the OWL axioms that the reasoner can handle are already built in and don't need to be redeclared.

Incompleteness

The rule based approach cannot offer a complete solution for OWL/Lite, let alone the OWL/Full fragment corresponding to the OWL/Lite constructs. In addition the current implementation is still under development and may well have ommisions and oversights. We intend that the reasoner should be sound (all inferred triples should be valid) but not complete.

[OWL index] [main index]

DAML support

This is minimal legacy support. The DAMLMicroReasoner is essentially the RDFS reasoner augmented by axioms declaring the equivalence between the DAML constructs and their RDFS aliases. It is invoked using the prebuilt `OntModelSpec` `DAML_MEM_RULE_INF`.

There are **no** plans to go beyond this and offer more complete DAML inference support.

[index]

The transitive reasoner

The TransitiveReasoner provides support for storing and traversing class and property lattices. This implements just the *transitive* and *symmetric* properties of `rdfs:subPropertyOf` and `rdfs:subClassOf`. It is not all that exciting on its own but is one of the building blocks used for the more complex reasoners. It is a hardwired Java implementation that stores the class and property lattices as graph structures. It is slightly higher performance, and somewhat more space efficient, than the alternative of using the pure rule engines to performance transitive closure but its main advantage is that it implements the direct/minimal version of those relations as well as the transitively closed version.

The `GenericRuleReasoner` (see below) can optionally use an instance of the transitive reasoner for handling these two properties. This is the approach used in the default RDFS reasoner.

It has no configuration options.

[Index]

The general purpose rule engine

Overview of the rule engine(s)

Jena2 includes a general purpose rule-based reasoner which is used to implement both the RDFS and OWL reasoners but is also available for general use. This reasoner supports rule-based inference over RDF graphs and provides forward chaining, backward chaining and a hybrid execution model. To be more exact, there are two internal rule engines one forward chaining RETE engine and one tabled datalog engine - they can be run separately or the forward engine can be used to prime the backward engine which in turn will be used to answer queries.

The various engine configurations are all accessible through a single parameterized reasoner `GenericRuleReasoner`. At a minimum a `GenericRuleReasoner` requires a ruleset to define its behaviour. A `GenericRuleReasoner` instance with a ruleset can be used like any of the other reasoners described above - that is it can be bound to a data model and used to answer queries to the resulting inference model.

The rule reasoner can also be extended by registering new procedural primitives. The current release includes a starting set of primitives which are sufficient for the RDFS and OWL implementations but is easily extensible.

[rule index] [main index]

Rule syntax and structure

A rule for the rule-based reasoner is defined by a Java `Rule` object with a list of body terms (premises), a list of head terms (conclusions) and an optional name and optional direction. Each term or `ClauseEntry` is either a triple pattern, an extended triple pattern or a call to a builtin primitive. A rule set is simply a List of Rules.

For convenience a rather simple parser is included with Rule which allows rules to be specified in reasonably compact form in text source files. However, it would be perfectly possible to define alternative parsers which handle rules encoded using, say, XML or RDF and generate Rule objects as output. It would also be possible to build a real parser for the current text file syntax which offered better error recovery and diagnostics.

An informal description of the simplified text rule syntax is:

```
Rule      :=   bare-rule .
           or   [ bare-rule ]
               or   [ ruleName : bare-rule ]

bare-rule :=   term, ... term -> hterm, ... hterm    // forward rule
           or   term, ... term <- term, ... term    // backward rule

hterm     :=   term
           or   [ bare-rule ]

term      :=   (node, node, node)           // triple pattern
           or   (node, node, functor)        // extended triple pattern
           or   builtin(node, ... node)      // invoke procedural primitive

functor   :=   functorName(node, ... node)  // structured literal

node      :=   uri-ref                          // e.g. http://foo.com/eg
           or   prefix:localname                 // e.g. rdf:type
           or   <uri-ref>                        // e.g. <myscheme:myuri>
           or   ?varname                       // variable
           or   'a literal'                    // a plain string literal
           or   'lex'^^typeURI                 // a typed literal, xsd:* type names supp
           or   number                         // e.g. 42 or 25.5
```

The "," separators are optional.

The difference between the forward and backward rule syntax is only relevant for the hybrid execution strategy, see below.

The *functor* in an extended triple pattern is used to create and access structured literal values. The functorName can be any simple identifier and is not related to the execution of builtin procedural primitives, it is just a datastructure. It is useful when a single semantic structure is defined across multiple triples and allows a rule to collect those triples together in one place.

To keep rules readable qname syntax is supported for URI refs. The set of known prefixes is those registered with the `PrintUtil` object. This initially knows about rdf, rdfs, owl, daml, xsd and a test namespace eg, but more mappings can be registered in java code. In addition it is possible to define additional prefix mappings in the rule file, see below.

Here are some example rules which illustrate most of these constructs:

```
[allID: (?C rdf:type owl:Restriction), (?C owl:onProperty ?P),
     (?C owl:allValuesFrom ?D)  -> (?C owl:equivalentClass all(?P, ?D)) ]

[all2: (?C rdfs:subClassOf all(?P, ?D)) -> print('Rule for ', ?C)
                [all1b: (?Y rdf:type ?D) <- (?X ?P ?Y), (?X rdf:type ?C) ] ]

[max1: (?A rdf:type max(?P, 1)), (?A ?P ?B), (?A ?P ?C)
      -> (?B owl:sameAs ?C) ]
```

Rule `allID` illustrates the functor use for collecting the components of an OWL restriction into a single datastructure which can then fire further rules. Rule `all2` illustrates a forward rule which creates a new backward rule and also calls the `print` procedural primitive. Rule `max1` illustrates use of numeric

literals.

Rule files may be loaded and parsed using:

```
    List rules = Rule.rulesFromURL("file:myfile.rules");
```

or

```
    BufferedReader br = /* open reader */ ;
    List rules = Rule.parseRules( Rule.rulesParserFromReader(br) );
```

or

```
    String ruleSrc = /* list of rules in line */
    List rules = Rule.parseRules( rulesSrc );
```

In the first two cases (reading from a URL or a BufferedReader) the rule file is preprocessed by a simple processor which strips comments and supports some additional macro commands:

`# ...`
: A comment line.

`// ...`
: A comment line.

`@prefix pre: <http://domain/url#>.`
: Defines a prefix `pre` which can be used in the rules. The prefix is local to the rule file.

`@include <urlToRuleFile>.`
: Includes the rules defined in the given file in this file. The included rules will appear before the user defined rules, irrespective of where in the file the @include directive appears. A set of special cases is supported to allow a rule file to include the predefined rules for RDFS and OWL - in place of a real URL for a rule file use one of the keywords `RDFS` `OWL` `OWLMicro` `OWLMini` (case insensitive).

So an example complete rule file which includes the RDFS rules and defines a single extra rule is:

```
    # Example rule file
    @prefix pre: <http://jena.hpl.hp.com/prefix#>.
    @include <RDFS>.

    [rule1: (?f pre:father ?a) (?u pre:brother ?f) -> (?u pre:uncle ?a)]
```

[Rule index] [main index]

Forward chaining engine

If the rule reasoner is configured to run in forward mode then only the forward chaining engine will be used. The first time the inference Model is queried (or when an explicit `prepare()` call is made, see above) then all of the relevant data in the model will be submitted to the rule engine. Any rules which fire that create additional triples do so in an internal *deductions* graph and can in turn trigger additional rules. There is a *remove* primitive that can be used to remove triples and such removals can also trigger rules to fire in removal mode. This cascade of rule firings continues until no more rules can fire. It is perfectly possible, though not a good idea, to write rules that will loop infinitely at this point.

Once the preparation phase is complete the inference graph will act as if it were the union of all the statements in the original model together with all the statements in the internal deductions graph generated by the rule firings. All queries will see all of these statements and will be of similar speed to normal model accesses. It is possible to separately access the original raw data and the set of deduced statements if required, see above.

If the inference model is changed by adding or removing statements through the normal API then this will trigger further rule firings. The forward rules work incrementally and only the consequences of the added or removed triples will be explored. The default rule engine is based on the standard RETE algorithm (C.L Forgy, *RETE: A fast algorithm for the many pattern/many object pattern match problem,* Artificial Intelligence 1982) which is optimized for such incremental changes.

When run in forward mode all rules are treated as forward even if they were written in backward ("<-") syntax. This allows the same rule set to be used in different modes to explore the performance tradeoffs.

There is no guarantee of the order in which matching rules will fire or the order in which body terms will be tested, however once a rule fires its head-terms will be executed in left-to-right sequence.

In forward mode then head-terms which assert backward rules (such as `all1b` above) are ignored.

There are in fact two forward engines included within the Jena2 code base, an earlier non-RETE implementation is retained for now because it can be more efficient in some circumstances but has identical external semantics. This alternative engine is likely to be eliminated in a future release once more tuning has been done to the default RETE engine.

[Rule index] [main index]

Backward chaining engine

If the rule reasoner is run in backward chaining mode it uses a logic programming (LP) engine with a similar execution strategy to Prolog engines. When the inference Model is queried then the query is translated into a goal and the engine attempts to satisfy that goal by matching to any stored triples and by goal resolution against the backward chaining rules.

Except as noted below rules will be executed in top-to-bottom, left-to-right order with backtracking, as in SLD resolution. In fact, the rule language is essentially datalog rather than full prolog, whilst the functor syntax within rules does allow some creation of nested data structures they are flat (not recursive) and so can be regarded a syntactic sugar for datalog.

As a datalog language the rule syntax is a little surprising because it restricts all properties to be binary (as in RDF) and allows variables in any position including the property position. In effect, rules of the form:

```
(s, p, o), (s1, p1, o1) ... <- (sb1, pb1, ob1), ....
```

Can be thought of as being translated to datalog rules of the form:

```
triple(s, p, o)    :- triple(sb1, pb1, ob1), ...
triple(s1, p1, o1) :- triple(sb1, pb1, ob1), ...
...
```

where "triple/3" is a hidden implicit predicate. Internally, this transformation is not actually used, instead the rules are implemented directly.

In addition, all the data in the raw model supplied to the engine is treated as ifit were a set of `triple(s,p,o)` facts which are prepended to the front of the rule set. Again, the implementation does not actually work that way but consults the source graph, with all its storage and indexing capabilities, directly.

Because the order of triples in a Model is not defined then this is one violation to strict top-to-bottom execution. Essentially all ground facts are consulted before all rule clauses but the ordering of ground facts is arbitrary.

Tabling

The LP engine supports tabling. When a goal is tabled then all previously computed matches to that goal are recorded (memoized) and used when satisfying future similar goals. When such a tabled goal is called and all known answers have been consumed then the goal will suspend until some other execution branch has generated new results and then be resumed. This allows one to successfully run recursive rules such as transitive closure which would be infinite loops in normal SLD prolog. This execution strategy, SLG, is essentially the same as that used in the well known XSB system.

In the Jena rule engine the goals to be tabled are identified by the property field of the triple. One can request that all goals be tabled by calling the `tableAll()` primitive or that all goals involving a given property `P` be tabled by calling `table(P)`. Note that if any property is tabled then goals such as `(A, ?P, ?X)` will all be tabled because the property variable might match one of the tabled properties.

Thus the rule set:

```
-> table(rdfs:subClassOf).
[r1: (?A rdfs:subClassOf ?C) <- (?A rdfs:subClassOf ?B) (?B rdfs:subClassOf ?C)]
```

will successfully compute the transitive closure of the subClassOf relation. Any query of the form (*, rdfs:subClassOf, *) will be satisfied by a mixture of ground facts and resolution of rule r1. Without the first line this rule would be an infinite loop.

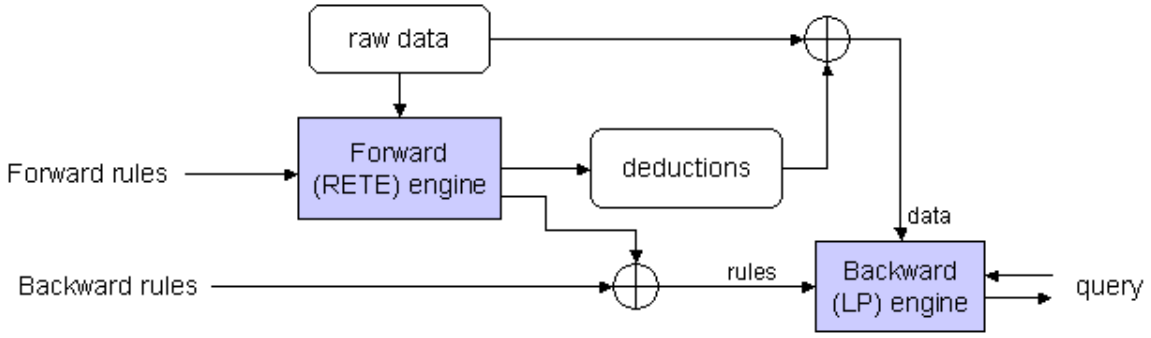
The tabled results of each query are kept indefinitely. This means that queries can exploit all of the results of the subgoals involved in previous queries. In essence we build up a closure of the data set in response to successive queries. The `reset()` operation on the inference model will force these tabled results to be discarded, thus saving memory and the expense of response time for future queries.

When the inference Model is updated by adding or removing statements all tabled results are discarded by an internal `reset()` and the next query will rebuild the tabled results from scratch.

[Rule index] [main index]

Hybrid rule engine

The rule reasoner has the option of employing both of the individual rule engines in conjunction. When run in this *hybrid* mode the data flows look something like this:



The forward engine runs, as described above, and maintains a set of inferred statements in the *deductions* store. Any forward rules which assert new backward rules will instantiate those rules according to the forward variable bindings and pass the instantiated rules on to the backward engine.

Queries are answered by using the backward chaining LP engine, employing the merge of the supplied and generated rules applied to the merge of the raw and deduced data.

This split allows the ruleset developer to achieve greater performance by only including backward rules which are relevant to the dataset at hand. In particular, we can use the forward rules to compile a set of backward rules from the ontology information in the dataset. As a simple example consider trying to implement the RDFS subPropertyOf entailments using a rule engine. A simple approach would involve rules like:

```
(?a ?q ?b) <- (?p rdfs:subPropertyOf ?q), (?a ?p ?b) .
```

Such a rule would work but every goal would match the head of this rule and so every query would invoke a dynamic test for whether there was a subProperty of the property being queried for. Instead the hybrid rule:

```
(?p rdfs:subPropertyOf ?q), notEqual(?p,?q) -> [ (?a ?q ?b) <- (?a ?p ?b) ] .
```

would precompile all the declared subPropertyOf relationships into simple chain rules which would only fire if the query goal references a property which actually has a sub property. If there are no subPropertyOf relationships then there will be no overhead at query time for such a rule.

Note that there are no loops in the above data flows. The backward rules are not employed when searching for matches to forward rule terms. This two-phase execution is simple to understand and keeps the semantics of the rule engines straightforward. However, it does mean that care needs to be take when formulating rules. If in the above example there were ways that the subPropertyOf relation could be derived from some other relations then that derivation would have to be accessible to the forward rules for the above to be complete.

Updates to an inference Model working in hybrid mode will discard all the tabled LP results, as they do in the pure backward case. However, the forward rules still work incrementally, including incrementally

asserting or removing backward rules in response to the data changes.

[Rule index] [main index]

GenericRuleReasoner configuration

As with the other reasoners there are a set of parameters, identified by RDF properties, to control behaviour of the `GenericRuleReasoner`. These parameters can be set using the `Reasoner.setParameter` call or passed into the Reasoner factory in an RDF Model.

The primary parameter required to instantiate a useful `GenericRuleReasoner` is a rule set which can be passed into the constructor, for example:

```
String ruleSrc = "[rule1: (?a eg:p ?b) (?b eg:p ?c) -> (?a eg:p ?c)]";
List rules = Rule.parseRules(ruleSrc);
...
Reasoner reasoner = new GenericRuleReasoner(rules);
```

A short cut, useful when the rules are defined in local text files using the syntax described earlier, is the `ruleSet` parameter which gives a file name which should be loadable from either the classpath or relative to the current working directory.

Summary of parameters

Parameter	Values	Description
PROPruleMode	"forward", "forwardRETE", "backward", "hybrid"	Sets the rule direction mode as discussed above. Default is "hybrid".
PROPruleSet	filename-string	The name of a rule text file which can be found on the classpath or from the current directory.
PROPenableTGCCaching	Boolean	If true, causes an instance of the TransitiveReasoner to be inserted in the forward dataflow to cache the transitive closure of the subProperty and subClass lattices.
PROPenableFunctorFiltering	Boolean	If set to true, this causes the structured literals (functors) generated by rules to be filtered out of any final queries. This allows them to be used for storing intermediate results hidden from the view of the InfModel's clients.
PROPenableOWLTranslation	Boolean	If set to true this causes a procedural preprocessing step to be inserted in the dataflow which supports the OWL reasoner (it translates intersectionOf clauses into groups of backward rules in a way that is clumsy to express in pure rule form).
PROPtraceOn	Boolean	If true, switches on exhaustive tracing of rule executions to the log4j *info* appender.
PROPderivationLogging	Boolean	If true, causes derivation routes to be recorded internally so that future getDerivation calls can return useful information.

[Rule index] [main index]

Builtin primitives

The procedural primitives which can be called by the rules are each implemented by a Java object stored in a registry. Additional primitives can be created and registered - see below for more details.

Each primitive can optionally be used in either the rule body, the rule head or both. If used in the rule body then as well as binding variables (and any procedural side-effects like printing) the primitive can act as a test - if it returns false the rule will not match. Primitives using in the rule head are only used for their side effects.

The set of builtin primitives available at the time writing are:

Builtin	Operations
isLiteral(?x) notLiteral(?x) isFunctor(?x) notFunctor(?x) isBNode(?x) notBNode(?x)	Test whether the single argument is or is not a literal, a functor-valued literal or a blank-node, respectively.
bound(?x...) unbound(?x..)	Test if all of the arguments are bound (not bound) variables
equal(?x,?y) notEqual(?x,?y)	Test if x=y (or x != y). The equality test is semantic equality so that, for example, the xsd:int 1 and the xsd:decimal 1 would test equal.
lessThan(?x, ?y), greaterThan(?x, ?y) le(?x, ?y), ge(?x, ?y)	Test if x is <, >, <= or >= y. Only passes if both x and y are numbers or time instants (can be integer or floating point or XSDDateTime).
sum(?a, ?b, ?c) addOne(?a, ?c) difference(?a, ?b, ?c) min(?a, ?b, ?c) max(?a, ?b, ?c) product(?a, ?b, ?c) quotient(?a, ?b, ?c)	Sets c to be (a+b), (a+1) (a-b), min(a,b), max(a,b), (a*b), (a/b). Note that these do not run backwards, if in sum a and c are bound and b is unbound then the test will fail rather than bind b to (c-a). This could be fixed.
strConcat(?a1, .. ?an, ?t) uriConcat(?a1, .. ?an, ?t)	Concatenates the lexical form of all the arguments except the last, then binds the last argument to a plain literal (strConcat) or a URI node (uriConcat) with that lexical form. In both cases if an argument node is a URI node the URI will be used as the lexical form.
regex(?t, ?p) regex(?t, ?p, ?m1, .. ?mn)	Matches the lexical form of a literal (?t) against a regular expression pattern given by another literal (?p). If the match succeeds, and if there are any additional arguments then it will bind the first n capture groups to the arguments ?m1 to ?mn. The regular expression pattern syntax is that provided by java.util.regex. Note that the capture groups are numbered from 1 and the first capture group will be bound to ?m1, we ignore the implicit capature group 0 which corresponds to the entire matched string. So for example `regexp('foo bar', '(.*) (.*)', ?m1, ?m2)` will bind `m1` to `"foo"` and `m2` to `"bar"`.
now(?x)	Binds ?x to an xsd:dateTime value corresponding to the current time.
makeTemp(?x)	Binds ?x to a newly created blank node.
makeInstance(?x, ?p, ?v) makeInstance(?x, ?p, ?t, ?v)	Binds ?v to be a blank node which is asserted as the value of the ?p property on resource ?x and optionally has type ?t. Multiple calls with the same arguments will return the same blank node each time - thus allowing this call to be used in backward

	rules.
noValue(?x, ?p) noValue(?x ?p ?v)	True if there is no known triple (x, p, *) or (x, p, v) in the model or the explicit forward deductions so far.
remove(n, ...) drop(n, ...)	Remove the statement (triple) which caused the n'th body term of this (forward-only) rule to match. Remove will propagate the change to other consequent rules including the firing rule (which must thus be guarded by some other clauses). Drop will silently remove the triple(s) from the graph but not fire any rules as a consequence. These are clearly non-monotonic operations and, in particular, the behaviour of a rule set in which different rules both drop and create the same triple(s) is undefined.
isDType(?l, ?t) notDType(?l, ?t)	Tests if literal ?l is (or is not) an instance of the datatype defined by resource ?t.
print(?x, ...)	Print (to standard out) a representation of each argument. This is useful for debugging rather than serious IO work.
listContains(?l, ?x) listNotContains(?l, ?x)	Passes if ?l is a list which contains (does not contain) the element ?x, both arguments must be ground, can not be used as a generator.
listEntry(?list, ?index, ?val)	Binds ?val to the ?index'th entry in the RDF list ?list. If there is no such entry the variable will be unbound and the call will fail. Only useable in rule bodies.
listLength(?l, ?len)	Binds ?len to the length of the list ?l.
listEqual(?la, ?lb) listNotEqual(?la, ?lb)	listEqual tests if the two arguments are both lists and contain the same elements. The equality test is semantic equality on literals (sameValueAs) but will not take into account owl:sameAs aliases. listNotEqual is the negation of this (passes if listEqual fails).
listMapAsObject(?s, ?p ?l) listMapAsSubject(?l, ?p, ?o)	These can only be used as actions in the head of a rule. They deduce a set of triples derived from the list argument ?l : listMapAsObject asserts triples (?s ?p ?x) for each ?x in the list ?l, listMapAsSubject asserts triples (?x ?p ?o).
table(?p) tableAll()	Declare that all goals involving property ?p (or all goals) should be tabled by the backward engine.

[Rule index] [main index]

Example

As a simple illustration suppose we wish to create a simple ontology language in which we can declare one property as being the concatenation of two others and to build a rule reasoner to implement this.

As a simple design we define two properties eg:concatFirst, eg:concatSecond which declare the first and second properties in a concatenation. Thus the triples:

```
eg:r eg:concatFirst  eg:p .
eg:r eg:concatSecond eg:q .
```

mean that the property r = p o q.

Suppose we have a Jena Model rawModel which contains the above assertions together with the additional facts:

```
eg:A eg:p eg:B .
eg:B eg:q eg:C .
```

Then we want to be able to conclude that A is related to C through the composite relation r. The following code fragment constructs and runs a rule reasoner instance to implement this:

```
String rules =
    "[r1: (?c eg:concatFirst ?p), (?c eg:concatSecond ?q) -> " +
    "     [r1b: (?x ?c ?y) <- (?x ?p ?z) (?z ?q ?y)] ]";
Reasoner reasoner = new GenericRuleReasoner(Rule.parseRules(rules));
InfModel inf = ModelFactory.createInfModel(reasoner, rawData);
System.out.println("A * * =>");
Iterator list = inf.listStatements(A, null, (RDFNode)null);
while (list.hasNext()) {
    System.out.println(" - " + list.next());
}
```

When run on a rawData model contain the above four triples this generates the (correct) output:

```
A * * =>
 - [urn:x-hp:eg/A, urn:x-hp:eg/p, urn:x-hp:eg/B]
 - [urn:x-hp:eg/A, urn:x-hp:eg/r, urn:x-hp:eg/C]
```

Example 2

As a second example, we'll look at ways to define a property as being both symmetric and transitive. Of course, this can be done directly in OWL but there are times when one might wish to do this outside of the full OWL rule set and, in any case, it makes for a compact illustration.

This time we'll put the rules in a separate file to simplify editing them and we'll use the machinery for configuring a reasoner using an RDF specification. The code then looks something like this:

```
// Register a namespace for use in the demo
String demoURI = "http://jena.hpl.hp.com/demo#";
PrintUtil.registerPrefix("demo", demoURI);

// Create an (RDF) specification of a hybrid reasoner which
// loads its data from an external file.
Model m = ModelFactory.createDefaultModel();
Resource configuration =  m.createResource();
configuration.addProperty(ReasonerVocabulary.PROPruleMode, "hybrid");
configuration.addProperty(ReasonerVocabulary.PROPruleSet,  "data/demo.rules");

// Create an instance of such a reasoner
Reasoner reasoner = GenericRuleReasonerFactory.theInstance().create(configuration);

// Load test data
Model data = FileManager.get().loadModel("file:data/demoData.rdf");
InfModel infmodel = ModelFactory.createInfModel(reasoner, data);

// Query for all things related to "a" by "p"
Property p = data.getProperty(demoURI, "p");
Resource a = data.getResource(demoURI + "a");
StmtIterator i = infmodel.listStatements(a, p, (RDFNode)null);
while (i.hasNext()) {
    System.out.println(" - " + PrintUtil.print(i.nextStatement()));
}
```

Here is file `data/demo.rules` which defines property `demo:p` as being both symmetric and transitive using pure forward rules:

```
[transitiveRule: (?A demo:p ?B), (?B demo:p ?C) -> (?A > demo:p ?C) ]
[symmetricRule: (?Y demo:p ?X) -> (?X demo:p ?Y) ]
```

Running this on data/demoData.rdf gives the correct output:

```
- (demo:a demo:p demo:c)
- (demo:a demo:p demo:a)
- (demo:a demo:p demo:d)
- (demo:a demo:p demo:b)
```

However, those example rules are overly specialized. It would be better to define a new class of property to indicate symmetric-transitive properties and and make demo:p a member of that class. We can generalize the rules to support this:

```
[transitiveRule: (?P rdf:type demo:TransProp)(?A ?P ?B), (?B ?P ?C)
                      -> (?A ?P ?C) ]
[symmetricRule: (?P rdf:type demo:TransProp)(?Y ?P ?X)
                      -> (?X ?P ?Y) ]
```

These rules work but they compute the complete symmetric-transitive closure of p when the graph is first prepared. Suppose we have a lot of p values but only want to query some of them it would be better to compute the closure on demand using backward rules. We could do this using the same rules run in pure backward mode but then the rules would fire lots of times as they checked every property at query time to see if it has been declared as a demo:TransProp. The hybrid rule system allows us to get round this by using forward rules to recognize any demo:TransProp declarations once and to generate the appropriate backward rules:

```
-> tableAll().

[rule1: (?P rdf:type demo:TransProp) ->
                         [ (?X ?P ?Y) <- (?Y ?P ?X) ]
                         [ (?A ?P ?C) <- (?A ?P ?B), (?B ?P ?C) ]
]
```

[rule index] [main index]

Combining RDFS/OWL with custom rules

Sometimes one wishes to write generic inference rules but combine them with some RDFS or OWL infernece. With the current Jena architecture limited forms of this is possible but you need to be aware of the limitations.

There are two ways of achieving this sort of configuration within Jena (not counting using an external engine that already supports such a combination).

Firstly, it is possible to cascade reasoners, i.e. to construct one InfModel using another InfModel as the base data. The strength of this appoach is that the two inference processes are separate and so can be of different sorts. For example one could create a GenericRuleReasoner whose base model is an external OWL reasoner. The chief weakness of the approach is that it is "layered" - the outer InfModel can see the results of the inner InfModel but not vice versa. For some applications that layering is fine and it is clear which way the inference should be layered, for some it is not. A second possible weakness is performance. A query to an InfModel is generally expensive and involves lots of queries to the data. The outer InfModel in our layered case will typically issue a lot of queries to the inner model, each of which may trigger more inferenece. If the inner model caches all of its inferences (e.g. a forward rule engine) then there may not be very much redundancy there but if not then performance can suffer dramatically.

Secondly, one can create a single GenericRuleReasoner whose rules combine rules for RDFS or OWL and custom rules. At first glance this looks like it gets round the layering limitation. However, the default Jena RDFS and OWL rulesets use the Hybrid rule engine. The hybrid engine is itself layered, forward rules do not see the results of any backward rules. Thus layering is still present though you have finer grain control - all your inferences you want the RDFS/OWL rules to see should be forward, all the inferences which need all of the results of the RDFS/OWL rules should be backward. Note that the RDFS and OWL rulesets assume certain settings for the GenericRuleReasoner so a typical configuration is:

```
Model data = FileManager.get().loadModel("file:data.n3");

List rules = Rule.rulesFromURL("myrules.rules");

GenericRuleReasoner reasoner = new GenericRuleReasoner(rules);
reasoner.setOWLTranslation(true);               // not needed in RDFS case
reasoner.setTransitiveClosureCaching(true);

InfModel inf = ModelFactory.createInfModel(reasoner, data);
```

Where the myrules.rules file will use @include to include one of the RDFS or OWL rule sets.

One useful variant on this option, at least in simple cases, is to manually include a pure (non-hybrid) ruleset for the RDFS/OWL fragment you want so that there is no layering problem. [The reason the default rulesets use the hybrid mode is a performance tradeoff - trying to balance the better performance of forward reasoning with the cost of computing all possible answers when an application might only want a few.]

A simple example of this is that the *interesting* bits of RDFS can be captured by enabled TranstiveClosureCaching and including just the four core rules:

```
[rdfs2:  (?x ?p ?y), (?p rdfs:domain ?c) -> (?x rdf:type ?c)]
[rdfs3:  (?x ?p ?y), (?p rdfs:range ?c) -> (?y rdf:type ?c)]
[rdfs6:  (?a ?p ?b), (?p rdfs:subPropertyOf ?q) -> (?a ?q ?b)]
[rdfs9:  (?x rdfs:subClassOf ?y), (?a rdf:type ?x) -> (?a rdf:type ?y)]
```

[rule index] [main index]

Notes

One final aspect of the general rule engine to mention is that of validation rules. We described earlier how reasoners can implement a `validate` call which returns a set of error reports and warnings about inconsistencies in a dataset. Some reasoners (e.g. the RDFS reasoner) implement this feature through procedural code. Others (e.g. the OWL reasoner) does so using yet more rules.

Validation rules take the general form:

```
(?v rb:validation on()) ...  ->
    [ (?X rb:violation error('summary', 'description', args)) <- ...) ] .
```

First the validate call with "switch on" validation by insert an additional triple into the graph of the form:

```
_:anon rb:validation on() .
```

This makes it possible to build rules, such as the template above, which are ignored unless validation has been switched on - thus avoiding potential overhead in normal operation. This is optional and the "validation on()" guard can be omitted.

Then the validate call queries the inference graph for all triples of the form:

```
?x rb:violation f(summary, description, args) .
```

The subject resource is the "prime suspect" implicated in the inconsistency, the relation `rb:violation` is a reserved property used to communicate validation reports from the rules to the reasoner, the object is a structured (functor-valued) literal. The name of the functor indicates the type of violation and is normally `error` or `warning`, the first argument is a short form summary of the type of problem, the second is a descriptive text and the remaining arguments are other resources involved in the inconsistency.

Future extensions will improve the formatting capabilities and flexibility of this mechanism.

[Rule index] [main index]

Extensions

There are several places at which the rule system can be extended by application code.

Rule syntax

First, as mentioned earlier, the rule engines themselves only see rules in terms of the Rule Java object. Thus applications are free to define an alternative rule syntax so long as it can be compiled into Rule objects.

Builtins

Second, the set of procedural builtins can be extended. A builtin should implement the `Builtin` interface. The easiest way to achieve this is by subclassing `BaseBuiltin` and defining a name (`getName`), the number of arguments expected (`getArgLength`) and one or both of `bodyCall` and

headAction. The bodyCall method is used when the builtin is invoked in the body of a rule clause and should return true or false according to whether the test passes. In both cases the arguments may be variables or bound values and the supplied RuleContext object can be used to dereference bound variables and to bind new variables.

Once the Builtin has been defined then an instance of it needs to be registered with BuiltinRegistry for it to be seen by the rule parser and interpreters.

The easiest way to experiment with this is to look at the examples in the builtins directory.

Preprocessing hooks

The rule reasoner can optionally run a sequence of procedural preprocessing hooks over the data at the time the inference graph is *prepared*. These procedural hooks can be used to perform tests or translations which are slow or inconvenient to express in rule form. See GenericRuleReasoner.addPreprocessingHook and the RulePreprocessHook class for more details.

[Index]

Extending the inference support

Apart from the extension points in the rule reasoner discussed above, the intention is that it should be possible to plug external inference engines into Jena. The core interfaces of InfGraph and Reasoner are kept as simple and generic as we can to make this possible and the ReasonerRegistry provides a means for mapping from reasoner ids (URIs) to reasoner instances at run time.

In a future Jena release we plan to provide at least one adapter to an example, freely available, reasoner to both validate the machinery and to provide an example of how this extension can be done.

[Index]

Futures

Whilst we can make no firm commitments, at the time of writing we intend to continue work on the Jena reasoning support. The key activities on the drawing board are:

- Develop a custom equality reasoner which can handle the "owl:sameAs" and related processing more efficiently that the plain rules engine.
- Tune the RETE engine to perform better with highly non-ground patterns.
- Tune the LP engine to further reduce memory usage (in particular explore subsumption tabling rather than the current variant tabling).
- Investigate routes to better integrating the rule reasoner with underlying database engines. This is a rather larger and longer term task than the others above and is the least likely to happen in the near future.

[Index]

Author: Dave Reynolds
Last modification: $Id: index.html,v 1.37 2007/09/19 07:53:13 der Exp $

Annexe 4

Jena Database interface

Favre Frédéric
HES-SO Valais, décembre 2007

Jena2 Database Interface - How To Create Persistent Models

The Jena2 persistent storage subsystem implements an extension of the Model class that provides transparent persistence for models through the use of a database engine. Three database engines are currently supported, MySQL, Oracle, PostgreSQL and Microsoft SQL server, on both Linux and WindowsXP.

This document provides a brief overview of creating and accessing Jena2 persistent models. Users are now encouraged to use the factory mechanisms described in `model-factory`. Previous mechanisms based on directly calling `ModelRDB` methods are now deprecated.

The various options for configuring and accessing persistent models are described in `Options`. For details on installing and configuring the various database engines for use with Jena2, see the database engine-specific "how to" documents (HSQLDB Howto, MySQL HowTo, Derby HowTo, PostgreSQL HowTo, Oracle HowTo, Microsoft SQL Server HowTo).

Creating and Accessing Persistent Models

In Jena2, all databases are multi-model and each model is, by default, stored in separate tables. Models may share database tables using the `StoreWithModel` option. Currently, model names may be any string however users are encouraged to use URIs as model names for compatibility with planned future Jena features. Note that the model name "DEFAULT" is reserved for use by Jena (as the name of the default model) and attempts to create such a named model will cause an exception.

As mentioned above, there are two mechanisms for creating persistent models, one using factory methods and another using constructors for the ModelRDB class. However, the factory methods do not return a ModelRDB instance. Consequently, certain methods defined on ModelRDB are not available for factory-created models, e.g., remove, setDoDuplicateCheck. If an application needs these methods, the ModelRDB constructors should be used. This is an interim measure until the factory-created models support all the ModelRDB capabilities. Below we review creating and opening models for each mechanism.

Factory Methods

Creating or opening a model is a three-step process. First, the driver class must be loaded and a connection established to the database (note that in Jena2, the database type is specified as part of the database connection). Second, a model maker class is constructed. The model maker creates persistent instances of the Model class. Third, the model maker class is invoked to create new models or to open existing models. The following examples show how this is done.

```
// database URL
String M_DB_URL         = "jdbc:mysql://localhost/test";
// User name
String M_DB_USER        = "test";
// Password
String M_DB_PASSWD      = "";
// Database engine name
String M_DB = "MySQL";
// JDBC driver
String M_DBDRIVER_CLASS = "com.mysql.jdbc.Driver";
// load the the driver class
Class.forName(M_DBDRIVER_CLASS);

// create a database connection
IDBConnection conn = new DBConnection(M_DB_URL, M_DB_USER, M_DB_PASSWD, M_DB);

// create a model maker with the given connection parameters
ModelMaker maker = ModelFactory.createModelRDBMaker(conn);
// create a default model
Model defModel = maker.createDefaultModel();
...
// Open existing default model
Model defModel = maker.openModel();

// or create a named model
Model nmModel = maker.createModel("MyNamedModel");
...
// or open a previously created named model
```

```
Model prvModel = maker.openModel("AnExistingModel");
```

ModelRDB Methods

If using datbase-specific, low-level options (see "Options for Initialization and Access"), then the application may need to more directly use the ModelRDB interface.

Database-backed RDF models are instances of the class `jena.db.ModelRDB` which supports the full `jena.model.Model` interface and also provides static methods to create, extend and reopen database instances.

ModelRDB supports several options (see "Options for Initialization and Access"). Some options alter the underlying database table structure and must be specified before the database is formatted. These methods are invoked on the underlying database driver instance for the connection. Ideally, the driver class need not be exposed to Jena applications. Consequently, use of the driver class to set these options should be considered an interim measure until the option setting can be integrated into the factory methods. See Enabling URI Compression for an example of setting these options. Other options apply to models. See Disable Duplicate Checking for an example of setting these types of options.

Creating an instance of ModelRDB is a two-step process. As with the factory methods, the first step is to load the driver class and establish a database connection. Second, the static methods on ModelRDB are used to create new ModelRDB instances or to open existing ones. The following examples show how this is done.

```
// As before ...
String M_DB_URL          = "jdbc:mysql://localhost/test";
String M_DB_USER         = "test";
String M_DB_PASSWD       = "";
String M_DB              = "MySQL";
String M_DBDRIVER_CLASS  = "com.mysql.jdbc.Driver";
// load the the driver class
Class.forName(M_DBDRIVER_CLASS);

// create a database connection
IDBConnection conn = new DBConnection(M_DB_URL, M_DB_USER, M_DB_PASSWD, M_DB);

// ---- Directly use ModelRDB

// create a default model
ModelRDB defModel = ModelRDB.createModel(conn);
...
// Open an existing model.
ModelRDB defModel2 = ModelRDB.openModel(conn);
...
// create a named model
ModelRDB nmModel = ModelRDB.createModel(conn,"MyModelName");
...
// open a named model
ModelRDB nmModel2 = ModelRDB.openModel(conn,"ExistingModelName");
...
```

Enable URI prefix compression

As an example of setting a database configuration option, the following shows how URI prefix compression is enabled. By default, URIs are stored fully expanded in the statement tables. Also shown is doubling the size of the prefix cache.

```
Class.forName(M_DBDRIVER_CLASS);
DBConnection dbcon = new DBConnection(M_DB_URL, M_DB_USER, M_DB_PASSWD, M_DB);
dbcon.getDriver().setDoCompressURI(true);
ModelRDB model = ModelRDB.createModel(dbcon, "myModelName");
// double the size of the prefix cache
int cacheSize = dbcon.getDriver().getCompressCacheSize();
dbcon.getDriver().setCompressCacheSize(cacheSize*2);
```

Detect multiple models per database

In Jena2, all databases are multi-model and each model is, by default, stored in separate tables. Here is an example that checks to see if a specific named model exists and then creates or reopens it as necessary:

```
Class.forName(M_DBDRIVER_CLASS);
DBConnection dbcon = new DBConnection(M_DB_URL, M_DB_USER, M_DB_PASSWD, M_DB);
ModelRDB model;
if( !dbcon.containsModel("myModelName")
        model = ModelRDB.createModel(dbcon, "myModelName");
else
        model = ModelRDB.open(dbcon, "myModelName");
```

Share tables among models

In this example, a configuration option is used to specify that a new model should share the tables of an existing model. Unlike the previous option, this option can be set after the database is formatted. It affects only subsequently created models.

```
Class.forName(M_DBDRIVER_CLASS);
DBConnection dbcon = new DBConnection(M_DB_URL, M_DB_USER, M_DB_PASSWD, M_DB);
ModelRDB model1, model2;
model1 = ModelRDB.createModel(dbcon, "myModel1");  // create a new model
dbcon.getDriver().setStoreWithModel("myModel1");
model2 = ModelRDB.createModel(dbcon, "myModel2"); // model2 is stored with model1
```

Disable duplicate checking

Some options apply to ModelRDB, e.g., query processing options. This example show how checking for duplicate statements can be disabled, as might be desirable when loading a large number of statements that the user is certain are duplicate-free.

```
Class.forName(M_DBDRIVER_CLASS);
DBConnection dbcon = new DBConnection(M_DB_URL, M_DB_USER, M_DB_PASSWD, M_DB);
ModelRDB model = ModelRDB.createModel(dbcon, "myModelName");
model.setDoDuplicateCheck(false);  // disable duplicate checking
```