PROYECTO DE GRADO

Editando y Razonando con Redes de Ontologías

Abril 2015

Ignacio Vidal Gomez

Universidad de la República - Montevideo, Uruguay

Tutor: Dr.-Ing. Regina Motz

Resumen

Este proyecto surge de la necesidad de integrar distintas ontologías dentro de redes ontológicas, con el objetivo de lograr un modelo de conocimiento más amplio. Estas ontologías en general, son mantenidas de forma independiente e incluso por diferentes grupos, por lo cual, además de lograr su integración es necesario mantener dicha independencia.
Existen casos donde la integración no es tan sencilla y presenta problemas, en particular, no es posible integrar dos ontologías que modelan un mismo objeto de la realidad de forma diferente (como un concepto en una y como un individuo en la otra). Para resolver esto se define un nuevo tipo de axioma llamado metamodelling que funciona relacionando los dos objetos de las distintas ontologías de forma que uno de ellos es un meta-modelo del otro. Este nuevo axioma implica además la creación de un nuevo tipo de lógica llamada ALCQM, para la definición de ontologías así como nuevas reglas para razonar en dicha lógica.

Por lo tanto el proyecto consiste en estudiar e implementar una solución para crear, editar y razonar con redes de ontologías utilizando lógica descriptiva ALCQM. Estas redes de ontologías están formadas por distintas ontologías vinculadas entre sí por cuatro tipos de relaciones, incluido el nuevo axioma de metamodelling. La lógica ALCQM es una extensión de ALCQ agregando la relación de metamodelling que vincula objetos de ontologías diferentes dentro de una red.

Para lograr esto, se implementó una solución en tres niveles distintos. En primer lugar se realizó una extensión del lenguaje OWL implementado en la API Java del mismo, agregando la relación de metamodelling, dado que dicha API es la base de todas las aplicaciones que trabajan con ontologías. Por otro lado, se desarrolló sobre el editor de ontologías Protégé, dos plugins, el primero enfocado en la redes ontológicas, permite editar y trabajar con ellas, y el segundo, apuntado al manejo de relaciones de metamodelling en ontologías individuales. Finalmente se modificó el razonador Pellet encargado de los chequeos de consistencia y la inferencia de nuevo conocimiento en las ontologías. Se agregaron las tres nuevas reglas y la condición definidas para ALCQM que permiten trabajar con redes de ontologías

La herramienta lograda fue verificada en diferentes condiciones de prueba, dando resultados satisfactorios teniendo en cuenta que se trata de una versión inicial. La misma permite crear y gestionar redes de ontologías e implementa en la práctica los conceptos definidos de redes, metamodelling y lógica ALCQM que hasta el momento eran inexistentes.

Palabras clave: Ontologías, OWL, Metamodelling, ALCQ, ALCQM, Protégé, Pellet, OWLAPI, Web semántica.

Contenido

1. Introducción

1.1. Problema

Una ontología es una representación formal y consensuada de conceptos que proveen un conocimiento compartido y común, procesable por máquinas e interoperable a través de agentes (software, individuos y organizaciones).[19]
La Figura 1 muestra un ejemplo de un ontología que modela una familia. En ella se describen las relaciones de ser padre/madre, hijo o hermano de una persona. Los objetos marcados con rombos violetas simbolizan individuos y los círculos amarillos representan clases o conceptos. Las líneas naranjas que unen individuos indican relaciones entre ellos, como ser hermano o ser padre, las líneas violetas de la parte superior representan la relación de *subclase* (por ejemplo *Padre* es un *Persona*) y las líneas azules indican la clase a la que pertenece un individuo (por ejemplo *Silvia* es *Madre* e *Hijo*).

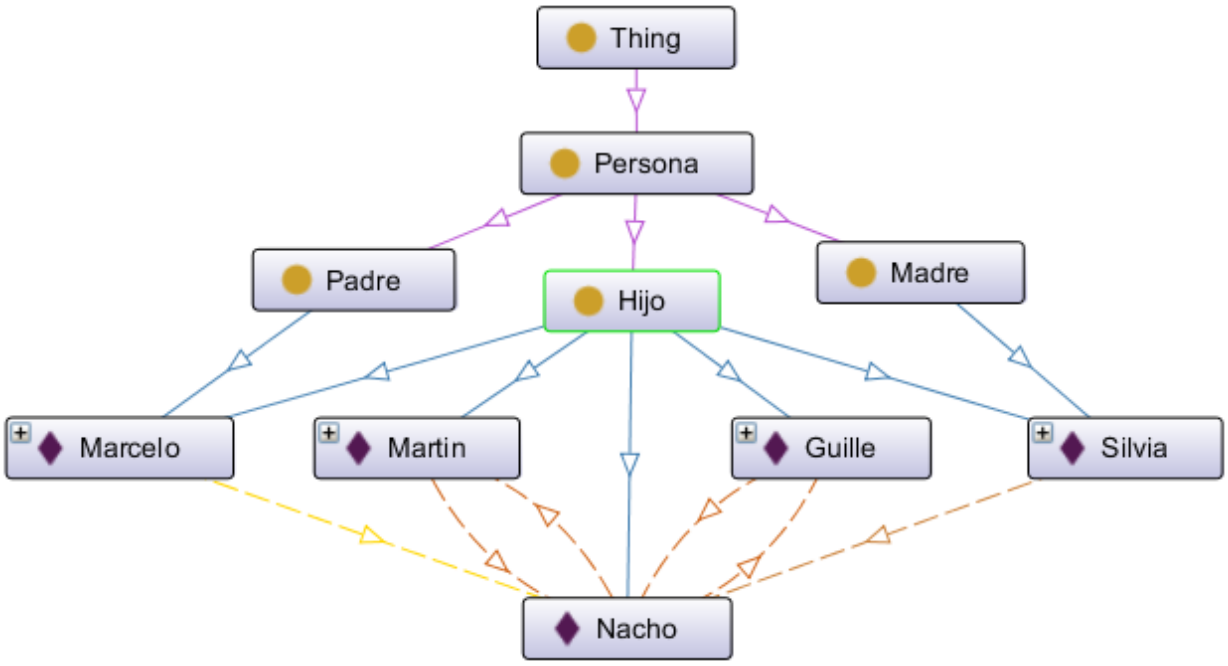


Figura 1. Ejemplo de Ontología

Las ontologías utilizan familias de lenguajes formales, llamados Lógicas Descriptivas, para describir de forma precisa los conceptos de un determinado dominio y las relaciones existentes entre estos. El uso de estos lenguajes permite implementar aplicaciones llamadas razonadores que pueden detectar inconsistencias e inferir nuevo conocimiento. Este poder de razonamiento automatizado se vuelve muy importante principalmente cuando las ontologías crecen y llegan a contener miles de conceptos, ya que de otra forma es imposible realizarlo.

A medida que el desarrollo de ontologías fue aumentando y tomando importancia surgió la necesidad de integrar los conceptos definidos en las mismas y en muchos casos la posibilidad de relacionar y extender conceptos provenientes de varias ontologías. Como resultado de dicha necesidad surgen las redes de ontologías.

Una red de ontologías, como se describe en el siguiente capítulo, es “una colección de ontologías vinculadas entre sí, a través de una variedad de relaciones” [2], con 3 objetivos principales, definir conceptualmente cada relación, visualizar el modelo conceptual desde un nivel de abstracción más elevado y por último facilitar el análisis de impacto cuando una de las ontologías de la red evoluciona.

Inicialmente la creación de las redes de ontologías se realiza editando y generando relaciones entre los conceptos de las ontologías de forma manual. Esta forma de trabajo no resulta cómoda ni amigable, ni tampoco provee una forma de verificar la consistencia de la red. Se hace necesario entonces contar con una herramienta que permita crear y editar dichas redes de ontologías, así como también que sea capaz de procesar las nuevas relaciones para chequear consistencia y generar inferencias.

1.2. Motivación

La integración de bases de conocimiento u ontologías no siempre resulta sencillo. Un problema particular que se presenta al integrar ontologías dentro de una misma red ontológica es el hecho de tener un mismo objeto modelado de formas diferentes. Mientras que en una ontología un objeto se representa como un individuo, en otra se modela mediante un concepto, por lo tanto la integración no es del todo posible.
Observemos por ejemplo un caso del mundo real de una aplicación sobre objetos geográficos que requiere integrar ontologías existentes y vincularlas dentro de una red de ontologías. Para describir este caso se muestra en la *Figura 2* un escenario simplificado de la aplicación que ilustra el problema presente. En la misma se muestra dos ontologías separadas por una línea las cuales conceptualizan las mismas entidades a distintos niveles de granularidad. En la ontología sobre la línea los ríos y los lagos son formalizados como individuos, mientras que en la ontología de abajo los mismos son conceptos. Si quisiéramos integrar estas ontologías dentro de una sola (o como parte de una red) sería necesario interpretar el individuo río y el concepto Río como el mismo objeto de la realidad. Lo mismo ocurre con lago y Lago. En este sentido la solución planteada en “*Reasoning for ALCQ extended with a flexible metamodelling hierarchy*”[1] consiste en igualar el individuo ***río*** al concepto ***Río*** y el individuo ***lago*** con el concepto ***Lago***. Estas equivalencias son llamadas axiomas de **metamodelling** y en ese caso, decimos que las ontologías están relacionadas a través de una relación de metamodelling.
En la Figura 2, los axiomas de metamodelling se encuentran representados por las líneas discontinuas. Luego de agregar dichos axiomas el concepto *ObjetoHidrográfico* es un meta-concepto, dado que es un concepto que contiene un individuo que también es un concepto.
De esta manera se logran integrar ontologías que poseen un mismo objeto (por ejemplo *rio*) en distintos niveles (individuo y concepto), dentro de una misma red ontológica.
Más adelante en el capítulo 2, veremos cómo en base a la nueva relación de metamodelling se define una nueva *Lógica Descriptiva* llamada *ALCQM.*

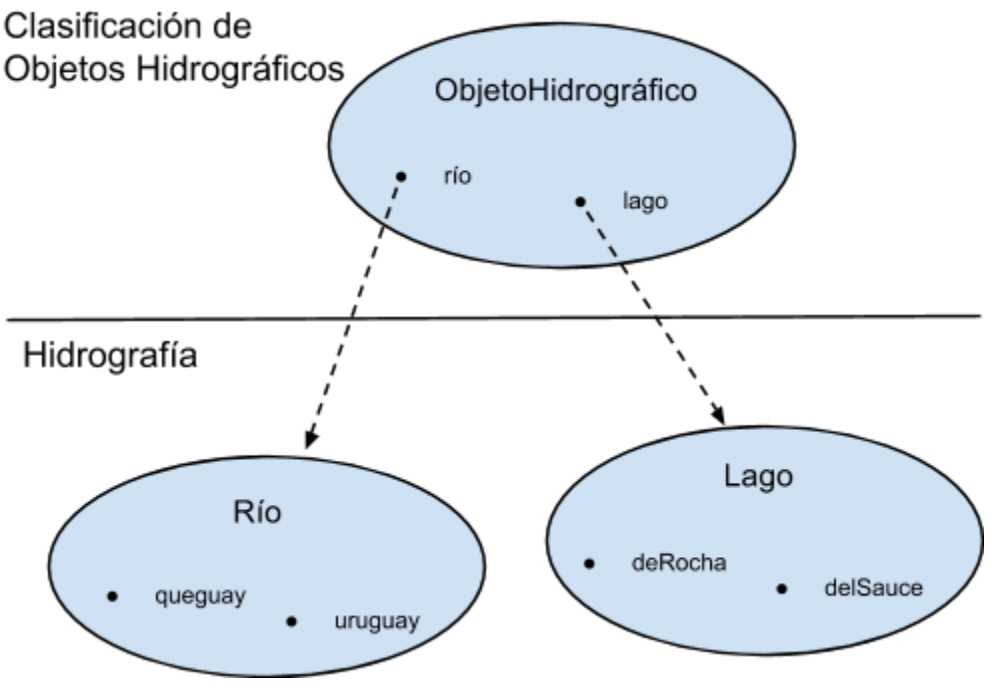


Figura 2. Dos ontología sobre hidrografía

1.3. Resultados obtenidos

El conjunto de herramientas desarrollado en el presente proyecto fue realizado en 3 niveles.
Como resultado más visible y cercano al usuario se implementó una herramienta de edición, un plugin llamado OntoRed que brinda una interfaz a través de Protégé[18] con la cual se pueden crear y mantener las redes de ontologías. En el mismo se puede importar nuevas ontologías, visualizar las relaciones y eliminar las mismas.
En un segundo nivel, se generó una versión modificada de la API de OWL[17], sobre la cual está implementado Protégé y sus servicios (así como otras muchas herramientas que trabajan con ontologías). Dicha API fue extendida con la implementación de la relación de metamodelling y herramientas para el manejo de la misma.
En el tercer nivel se obtuvo el resultado más importante del proyecto, el razonador *Pellet*[16] extendido para lógica ALCQM y relaciones de metamodelling. Partiendo de la versión oficial de Pellet se implementaron tres nuevas reglas y una condición de chequeo de ciclos que dota al razonador con la capacidad de chequear e inferir nuevo conocimiento de ontologías y redes de ontologías con relaciones de metamodelling. Se generaron dos versiones de Pellet, un plugin para ejecutar desde Protégé y una versión standalone.
Adicionalmente como producto extra se desarrolló otro plugin, llamado Metamodelling View, el cual provee una interfaz dentro de Protégé que permite crear relaciones de metamodelling en ontologías individuales, a diferencia del plugin OntoRed que esta diseñado para redes ontológicas.

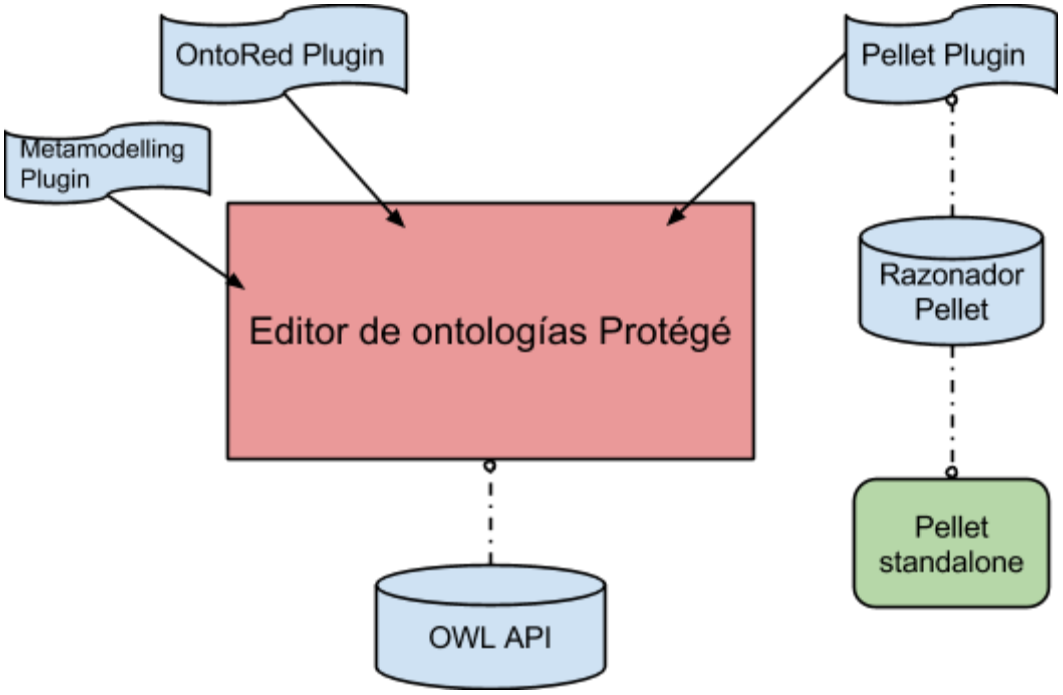


Figura 3.Arquitectura de la solución.

En la Figura 3 se puede ver un esquema de la arquitectura de la solución obtenida, los plugins OntoRed y Metamodelling View para la gestión de redes y relaciones de metamodelling, la OWLAPI modificada, y las dos versiones generadas del razonador Pellet, por un lado como plugin de Protégé y por otro su versión standalone.

El resto del documento se organiza de la siguiente manera: en el capítulo 2 se desarrollan los fundamentos principales sobre los que se desarrolla el trabajo. En el capítulo 3 se realiza la descripción del análisis, diseño e implementación de la herramienta en sus distintos niveles y componentes. En el 4 se describe el producto logrado, así como su instalación y modo de uso, detallando además el proceso de verificación. Para finalizar en el capítulo 5 se presentan las conclusiones, limitaciones de la solución y trabajos a futuro propuestos.
Junto al informe se incluye la sección de anexos, el anexo I y II se encuentra la lista de clases modificadas y creadas en la implementación de la herramienta, en el III el manual de uso de la herramienta y en el anexo IV la lista de casos de prueba definidos para el razonador.

2. Fundamentos Básicos

2.1. Redes de ontologías

En este punto se presentan los fundamentos y definiciones relativos a las redes de ontologías tomados del trabajo *"Reasoning for ALCQ extended with a flexible meta-modelling hierarchy."*[1], sobre el cual se basa el presente proyecto.

Llamamos **Red de ontologías** a: "una colección de ontologías vinculadas mediante distintas relaciones". Formalmente definimos una red como:

Def. 1: Red de ontologías :
Una red es un par ($\mathbb{O}$, $\mathbb{R}$) tal que $\mathbb{O}$ = { O_1 ,…, O_n } es un conjunto de ontologías y $\mathbb{R}$ = {$\Re_1$,…, $\Re_m$} es un conjunto de relaciones entre ontologías. La ontología asociada a una *red de ontologías* se denota por Ont($\mathbb{O}$, $\mathbb{R}$) y es definida como $\bigcup_{i=1}^{n} O_1 \cup \bigcup_{i=1}^{m} \Re A_i$ donde $\Re A_i$ es el conjunto de axiomas asociados a la relación $\Re_i$ para todo 1 ≤ i ≤ m.

A su vez, una relación entre redes está definida como:

Def. 2: Relación entre ontologías:
$\Re$ es una relación entre las ontologías O_1 y O_2 si existe un conjunto $\Re A$ de axiomas que expresan $\Re$.

Dichas relaciones que vinculan a las ontologías en una red pueden ser de cuatro tipos:

Def. 3: Extension:
$\Re$ es una relación ***extension*** entre O_1 y O_3 si la signatura y los axiomas de O_1 están incluidos en los de O_2 . El conjunto $\Re A$ de axiomas es $O_2 \setminus O_1$.

Def. 4: Mapping:
$\Re$ es una relación ***mapping*** entre O_1 y O_2 si existe un conjunto $\Re A$ de axiomas que tiene una de las siguiente formas:

C ⊑ D C ≡ D C ⊓ D ⊑ ⊥ D(a) a ≡ b

donde C, a ∈ O_1 y D, b ∈ O_2 .

Def. 5: Link:
$\Re$ es una relación ***link*** entre O_1 y O_2 si:
1. existe un conjunto $\Re L$ de nuevos roles llamados *linking role*. Dado $\Re L$, definimos el lenguaje C_1 generado a partir de $\Re L$ por inducción. Las reglas para C_1 son las siguientes:
(a) Todo concepto básico de O_1 esta en C_1,
(b) C_1 es cerrado bajo ⊔, ¬,

(c) Si C ∈ C_1 y R ∈ O_1 entonces ∃R.C, y ≥nR.C pertenece a C_1,
(d) Si C ∈ C_2 y L ∈ $\Re_L$ entonces ∃L.C, y ≥nL.C pertenece a C_1,
2. existe un conjunto $\Re_A$ de axiomas que tiene una de las siguientes formas:
$C_1 \sqsubseteq D_1$ $C_2 \sqsubseteq D_2$ $C_1(a_1)$ $C_2(a_2)$ $\langle a_1, a_2 \rangle : L$
donde todo C_1, D_1, a_1 pertenece a C_1, a_2 pertenece a O_2 y L es un *linking role* perteneciente a $\Re_L$.

Def. 6: Metamodelling:
Decimos que $\Re$ es una relación de ***metamodelling*** entre O_1 y O_2 si existe un conjunto $\Re_A$ de axiomas de la forma $a =_m A$ para a ∈ O_1 y A ∈ O_2.
Estos axiomas deben satisfacer la restricción: para cada $a =_m A \in \Re_A$, existe tal que A(b) ∈ O_2.

2.2. Lógica ALCQM

Las **Lógicas Descriptivas** (DL por *Description Logics*) son una familia de lenguajes de representación del conocimiento que pueden ser usados para representar conocimiento terminológico de un dominio de aplicación de una forma estructurada y formalmente bien comprendida. Se diseñaron como una extensión de marcos (frames) y redes semánticas, pero a diferencia de estos, las DLs están dotadas con una semántica formal basada en lógica y poseen características importantes como:

- Un *formalismo descriptivo*: conceptos, roles, individuos y constructores.
- Un *formalismo terminológico*: axiomas terminológicos que introducen descripciones complejas y propiedades de la *terminología descriptiva*.
- Un *formalismo asertivo*: que introduce propiedades de individuos.
- Son *capaces de inferir nuevo conocimiento* a partir de conocimiento dado; tienen por tanto, algoritmos de razonamiento que son decidibles.

Los bloques de construcción sintácticos básicos son los conceptos atómicos (predicados unarios), los roles atómicos (predicados binarios) e individuos (constantes).
El poder expresivo del lenguaje está restringido a un pequeño conjunto de constructores para crear conceptos complejos y roles.
El conocimiento implícito sobre conceptos e individuos puede inferirse automáticamente con la ayuda de procedimientos de inferencia implementados en los llamados razonadores de DL.

La base de conocimiento de las DLs se compone por dos tipos diferentes de sentencias, las cuales se agrupan en ABox (aserciones) y TBox (términos). En general la TBox contiene las sentencias que describen conceptos jerárquicos, por ejemplo *"toda persona es un empleado"*, mientras que la ABox contiene las sentencias que indican a donde pertenecen los individuos en la jerarquía, por ejemplo *"Juan es un empleado"*.

Esta distinción de tipos de sentencia resulta útil para describir y formular procedimientos de decisión. Desde el punto de vista del modelado de la base de conocimiento (KB) tiene sentido poder distinguir entre conceptos y su jerarquía (TBox), y la pertenencia de los individuos al mismo (ABox). Por otro lado, dado que

la complejidad de la TBox puede afectar el rendimiento de un algoritmo de decisión resulta conveniente dicha separación en dos conjuntos.

Una familia de *DLs* muy popular es la conocida como ***AL*** (por *Attributive Language*), dentro de esta familia encontramos la lógica ***ALCQ***. La sintaxis e ALCQ soporta la descripción de conceptos, roles (relaciones) e individuos. Además los roles y los conceptos se pueden combinar mediante una variedad de operadores para formar expresiones más complejas. Entre los operadores de ALCQ se encuentran las conectivas lógicas estándares, los cuantificadores universales y existenciales y las restricciones de cardinalidad.

La DL sobre la cual se desarrolla el presente trabajo es la lógica ***ALCQM*** presentada en [1].
ALCQM surge con el fin de expresar *meta-modelos* de conceptos, pudiendo integrar varias ontologías de distintas fuentes donde un mismo objeto se define como concepto y como individuo. La sintaxis de ALCQM se obtiene de ALCQ añadiendo a esta la relación que permite igualar individuos con concepto, la relación de *metamodelling*.

Un axioma de ***metamodelling*** es un nuevo tipo de sentencia de la forma $a =_m A$, donde a es un *individuo* y A es un *concepto atómico.*

Debido a que se igualan objetos de distintos niveles, es clave en ALCQM definir correctamente la semántica para detectar las inconsistencias que pueda generar un *metamodelling*. En una semántica adecuada la interpretación de un individuo y de un concepto igualados mediante este tipo de relación debe ser la misma.

Como se vió antes las sentencias en ALCQ están divididas en dos grupos, *TBox* y *ABox*. Para ALCQM se define un nuevo conjunto llamado ***MBox***:

MBox es el conjunto M que contiene los axiomas de tipo *metamodelling.*

De esta forma denotamos una ontología o una base de conocimiento en ALCQM por sus tres conjuntos de axiomas como:

$O = (T, A, M)$ con T TBox, A ABox y M MBox.

Con estas definiciones la red de ontologías presentada en el capítulo de introducción se describe como se ve en la Figura 4 dividiendo los distintos axiomas en los tres conjuntos diferentes: TBox, ABox y MBox.

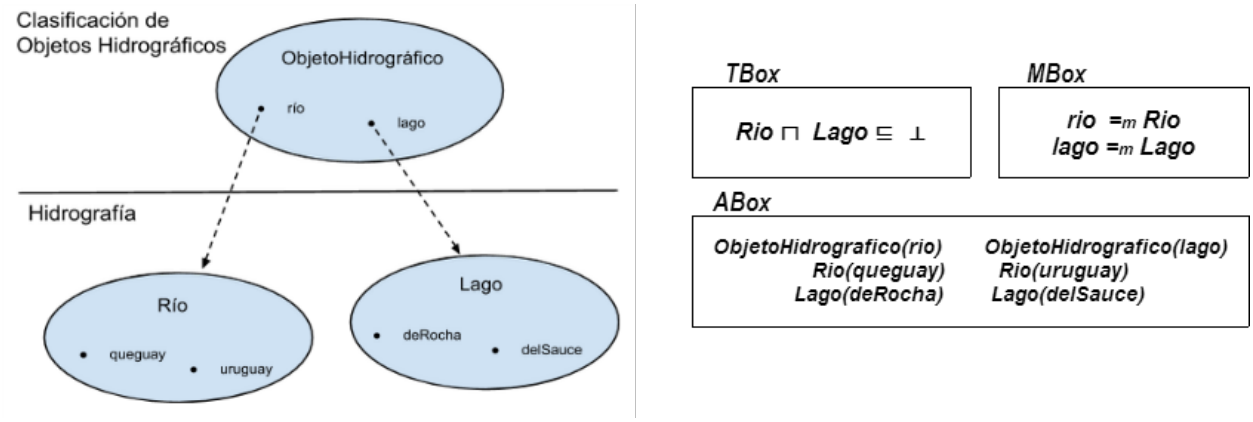


Figura 4. Ontología Hidrográfica descrita en ALCQM

Consistencia en ALCQM:

Para chequear la consistencia de las ontologías en la nueva lógica es necesario definir un algoritmo de *tableau*. El nuevo algoritmo parte del *tableau* de ALCQ y lo extiende con tres nuevas reglas, de esta forma para desarrollar un razonador será suficiente con implementar las nuevas reglas sobre un razonador OWL.

Las tres nuevas reglas de expansión se ocupan de las igualdades y desigualdades entre individuos con *metamodelling* los cuales deben ser transferidos al nivel de conceptos como igualdades y desigualdades entre los conceptos correspondientes.

$\approx$: Sea $a =_m A$ y $b =_m B$ en M.
Si $a \approx b$ y $A \sqcap \neg B,\ B \sqcap \neg A$ no pertenecen a T entonces:
$T \leftarrow A \sqcup \neg B,\ B \sqcup \neg A$.

$\not\approx$: Sea $a =_m A$ y $b =_m B$ en M.
Si $a \not\approx b$ y no existe z tal que $A \sqcap \neg B \sqcup B \sqcap \neg A \in L(z)$ entonces:
se crea un nodo z con $L(z) = \{A \sqcap \neg B \sqcup B \sqcap \neg A\}$.

close: Sea $a =_m A$ y $b =_m B$ donde $a \approx x$, $b \approx y$, $L(y)$ están definidos.
Si ni $x \approx y$ ni $x \not\approx y$ existen, se agrega: $a \equiv b$ ó $a \not\equiv b$.

Reglas de expansión para ALCQM

Inicialmente el *tableau* de ALCQM es casi el mismo que el de ALCQ. Los nodos del grafo inicial se crean a partir de los individuos de la ABox y de la MBox. A continuación el algoritmo procede de forma no determinística aplicando las reglas de expansión para ALCQM. Estas reglas son el conjunto de reglas ALCQ más las tres definidas anteriormente.
A medida que el algoritmo avanza el conjunto de axiomas de la Tbox y el grafo L, denotado como (T, L) va cambiando.
El algoritmo termina cuando se alcanza algún (T, L), donde (T, L) es *ALCQM-completo*, (es decir, no se pueden aplicar más reglas de expansión), L tiene una contradicción ó L tiene un ciclo. La ontología

(T, A, M) es consistente si existe un (T, L) *ALCQM-completo* tal que L no tiene contradicciones ni ciclos. De lo contraria, es inconsistente.

2.3. Protégé

Protégé[1] es una plataforma open-source desarrollada principalmente por la Universidad de Standford, con el apoyo de varias agencias gubernamentales y privadas. Provee un conjunto de herramientas para la construcción de modelos de dominio y bases de conocimiento aplicadas a ontologías.
La herramienta más popular de Protégé es su editor de ontologías el cual implementa un potente entorno para el diseño, modelado, implementación, manipulación y visualización de ontologías en varios formatos. Soporta completamente el lenguaje OWL 2 en sus tres variantes (Lite, DL y Full).

En Protégé se distinguen 3 componentes fundamentales para el desarrollo de ontologías:

- *Individuos*: representan los objetos del dominio en que estamos interesados, se puede decir que son las instancias de las clases.
- *Propiedades*: son las relaciones binarias entre individuos.
- *Clases*: las clases son los conjuntos donde se contienen los individuos, también llamados conceptos.

La plataforma está desarrollada apuntando a la extensibilidad, implementa una arquitectura basada en la *infraestructura OSGi*[2]. De esta manera se logra una plataforma desacoplada sobre la que pueden instalarse nuevas funcionalidades fácilmente mediante plugins.
Un caso particular de extensión de funcionalidad son los razonadores de lógicas descriptivas. Protégé brinda a través de su API herramientas con las cuales distintos desarrolladores han construido sus razonadores. Algunos de los más populares son Pellet, HermiT o FacT++.

2.4. Razonador Pellet

Pellet es una herramienta de razonamiento para ontologías especificadas en OWL. Está implementado en Java y es de distribución libre a través de su sitio web. Provee una variedad de tareas de razonamiento y posee una gran compatibilidad con otras herramientas de razonamiento como es el caso de JENA.

En la Figura 5 se pueden ver los principales componentes de su arquitectura. Básicamente se trata de un *core* que implementa el razonador de lógica descriptiva mediante diferentes implementaciones del algoritmo de *tableau*, para chequear la consistencia de la base de conocimiento y otros servicios.

[1] "Protégé." 2002. 19 Ene. 2015 <http://protege.stanford.edu/>
[2] "OSGi Alliance | Technology / HomePage." 2012. 20 Ene. 2015 <http://www.osgi.org/Technology/HomePage>

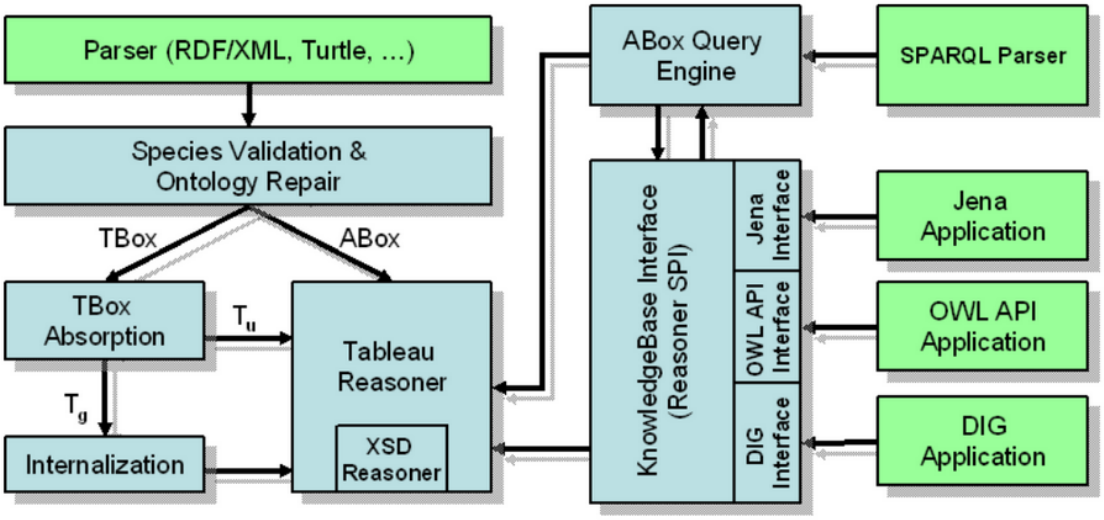


Figura 5. Arquitectura de los componentes de Pellet

Junto a esto, Pellet implementa distintas técnicas de optimización para DL como son la *normalización, simplificación, absorción, ramificación semántica,* entre otros.

Algunas de las características a destacar son:

- ABox query: se incluye un sistema de búsqueda que puede responder consultas en SPARQL.
- Razonamiento con Data Types: Pellet soporta razonamiento sobre Data Types construidos por el usuario.
- Refinamiento de axiomas: Posibilita mejorar y corregir las ontologías.

2.5. Lenguaje OWL

Con el objetivo de expresar la información contenida en una ontología y transmitirla a los distintos agentes semánticos, se desarrollaron diferentes lenguajes.
Uno de los principales lenguajes es RDF:
Resource Description Framework (RDF) es un marco para metadatos en la Web. Posee clases y propiedades y jerarquías de estas, rango y dominio de propiedades.
Pero RDF no es lo suficientemente expresivo para describir recursos, ya que , no tiene restricciones de dominio y rango, no tiene propiedades simétricas ni inversas, entre otras.
Sumado a esto, RDF tiene semánticas no estándares lo que dificulta soportar el razonamiento.

Debido a esto, en busca de mayor expresividad, se desarrollaron dos nuevos lenguajes, por un lado OIL, desarrollado por un grupo de investigadores europeos y por otro lado DAML-ONT, desarrollado por investigadores de Estados Unidos (programa DARPA DAML).
Ambos grupos de investigadores aunaron esfuerzos para producir juntos el lenguaje DAML+OIL que extendió a RDF logrando una semántica clara y bien definida.

Finalmente el grupo WebOnt desarrolló el ***lenguaje OWL***[3] basado en DAML+OIL, el cual es ahora una recomendación W3C, es decir un estándar.
Dentro del lenguaje OWL se encuentran tres variantes: *OWL Lite* el más sencillo y menos expresivo, un subconjunto, *OWL DL* está construido de tal forma que toda sentencia pueda ser resuelta en tiempo finito *y OWL Full* la variante más expresiva y completa de OWL puede resultar en bucles infinitos.

Algunos de los elementos principales que componen la sintaxis de OWL son:

Sintaxis OWL	Sintaxis DL	Ejemplo
subClassOf	$C_1 \sqsubseteq C_2$	$Humano \sqsubseteq Animal \sqcap Bipedo$
equivalentClass	$C_1 \equiv C_2$	$Hombre \equiv Humano \sqcap Masculino$
subPropertyOf	$P_1 \sqsubseteq P_2$	$tieneGato \sqsubseteq tieneMascota$
equivalentProperty	$P_1 \equiv P_2$	$costo \equiv precio$
transitiveProperty	$P^+ \sqsubseteq P$	$ancestro^+ \sqsubseteq ancestro$
type	$a : C$	$Juan : Padre$
property	$< a, b >: R$	$< Maria, Juan >: tieneHijo$

La versión actual del lenguaje OWL es OWL 2, la misma es una extensión de la versión 1, agrega pequeñas pero útiles características solicitadas por los usuarios principalmente. Provee una considerable mejora en cuanto al manejo de *datatypes* y soluciona ciertos problemas de expresividad de OWL 1.

[3] "OWL Web Ontology Language Reference." 2002. 23 Ene. 2015 <http://www.w3.org/TR/owl-ref/>

3. Desarrollo de la solución

En el presente capítulo se realiza el análisis de la solución implementada, partiendo del problema hasta su implementación final. Dado que la implementación se realizó en 3 niveles diferentes, se describe cada paso por separado: en primer lugar se describen las modificaciones y agregados hechos sobre la API de OWL; en segundo lugar la implementación del plugin sobre Protégé, la interfaz para el usuario; y finalmente, se presenta el desarrollo realizado sobre el razonador Pellet para obtener la solución requerida.

Analizando el problema planteado se observa que una ***red de ontologías*** está compuesta por varias ontologías y diferentes relaciones entre los objetos de las mismas.
El editor de ontologías Protégé sobre el cual se implementa el plugin **OntoRed**, tiene la capacidad de importar varias ontologías dentro de una nueva y a su vez permite crear relaciones en ALCQ entre los objetos. Utilizando esto, la red de ontologías se modela como una ontología normal de Protégé sobre la cual se importan las ontologías de la red.
Como se vió en 2.1, las relaciones para las redes son de 4 tipos, 3 de estos tipos se pueden modelar mediante relaciones normales de Protégé y OWL restringiendo el dominio y rango de la relación a objetos de ontologías diferentes.
Por lo tanto tenemos que:

- relación **Linking** se modela como una relación object*Property* entre clases de ontologías diferentes.
- relación **Mapping** es modelada como una relación *subClassOf* o *equivalentTo* donde las clases pertenecen a ontologías diferentes.
- relación **Extension** es un ***import*** de Protégé con el cual se importan las ontologías dentro de la red.

Para la relación de **Metamodelling** es necesario extender la API de OWL sobre la cual está desarrollado Protégé de forma que permita relacionar una clase y un individuo de distintas ontologías como conceptos equivalentes.

Dado que ni Protégé ni la API de OWL están construidos para trabajar directamente con redes de ontologías y mucho menos en **ALCQM** es necesario mantener de alguna forma las relaciones de *metamodelling* de la red. Una de las opciones es mantener un archivo separado donde se persista esa información, lo cual presenta la desventaja del mantenimiento y lectura del mismo desde el plugin. Por otro lado, está la opción de aprovechar el desarrollo requerido sobre la API de OWL e implementar el mecanismo para que las relaciones de tipo *metamodelling* se persistan y mantengan en el mismo archivo que se persiste la ontología (en lenguaje OWL/XML). Está segunda opción es la escogida, aunque requiere mayor trabajo, crea una herramienta más estándar y menos compleja al no depender de otras fuentes de datos.

Por el otro lado está el razonador que debe procesar las relaciones dentro de la red para chequear la consistencia e inferir nuevo conocimiento. Aquí de nuevo encontramos que no existe ningún razonador capaz de trabajar sobre **ALCQM**, es decir con relaciones de metamodelling. Por lo tanto es necesario personalizar un razonador existente implementando las nuevas reglas presentadas en [1].
Los razonadores existentes en Protégé funcionan como plugins y a su vez utilizan la API de OWL para procesar las ontologías. Entre los razonadores más populares encontramos Pellet, HermiT y FaCT++.

A continuación se detalla para cada etapa de la implementación cual fue el desarrollo realizado. Para todos los casos el entorno utilizado fue ***Eclipse version Kepler SR 2*** para Java y ***JDK y JRE de Java versión 7***.

3.1. OWL API versión modificada

La OWL API[4] es una API implementada en Java para la creación, manipulación y serialización de Ontologías OWL. Desde la versión 3.1 la misma está diseñada para OWL 2.
Se distribuye bajo licencia Open Source LGPL y Apache.

Algunos de los componentes que incluye la API son:

- interfaz para OWL 2.
- implementación de referencia de OWL
- parsers y escritores para RDF/XML, OWL/XML, OWL Functional Syntax, Manchester OWL Syntax, Turtle, KRSS y OBO.
- interfaces para trabajar con razonadores, como por ejemplo FaCT++, HermiT, Pellet o Racer.

Mediante sus interfaces la OWLAPI brinda acceso a los distintos objetos y componentes de las ontologías basadas en este lenguaje. Es por esto que resulta un elemento central en el desarrollo de herramientas que trabajan con ontologías OWL.
En el caso de este trabajo, tanto el editor Protégé como el razonador Pellet utilizan la API.

Dado que la relación de metamodelling es un agregado del lenguaje OWL y no existe como tal, será necesario implementarla dentro de la OWLAPI.

Para extender la API se escoge la versión 3.4.5 de la misma, ya que es compatible con las versiones tanto de Protégé como de Pellet.

La API OWL fue desarrollada siguiendo unos principios de diseño básicos, los cuales se resumen en:

- interfaces que proveen acceso de solo lectura al modelo.
- manipulación mediante operaciones explícitas.
- independencia de las diferentes sintaxis.
- separación clara entre componentes que proveen funcionalidades particulares, como son parsing, renderizado, manipulación o representación.
- separación entre aserciones e inferencias.

Uno de sus principales componentes es el package *org.semanticweb.owlapi.model*, en el cual se encuentran las interfaces de los elementos del lenguaje modelados, como son los axiomas, los objetos y las ontologías. En la Figura 6 se ve un diagrama UML que modela la relación entre los elementos principales.

[4] "OWL API." 2004. 18 Mar. 2015 <http://owlapi.sourceforge.net/>

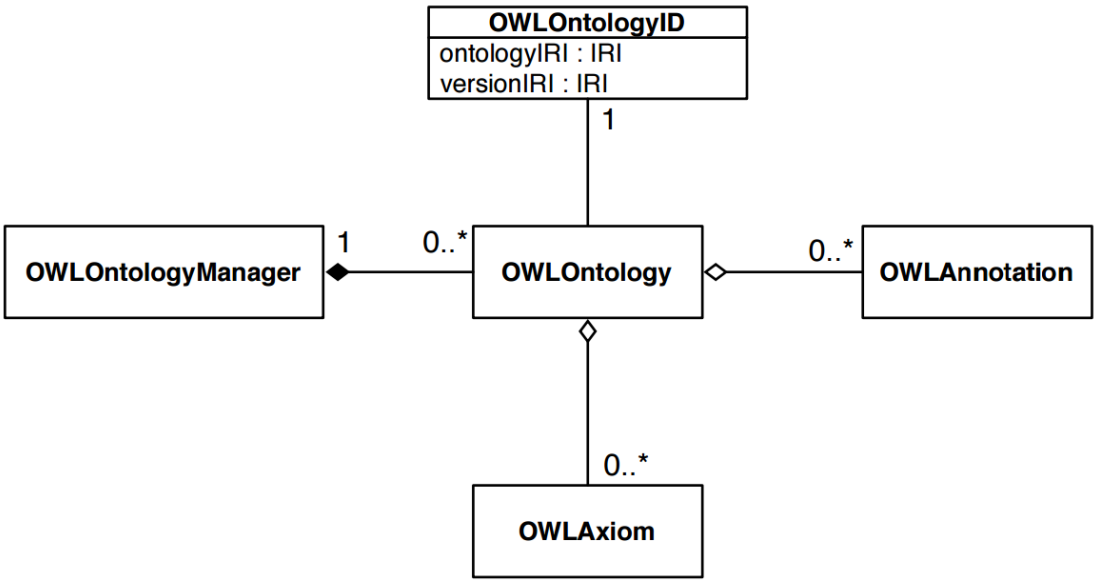


Figura 6. Modelo de los elementos principales de la API.

Modelo

El modelo de la OWL API brinda acceso a la ontología OWL a través de varias clases e interfaces. Respetando los principios de diseño estas interfaces son de solo lectura y no proveen ninguna funcionalidad explícita para modificar la estructura de datos.
Siguiendo la especificación de OWL 2, el modelo implementado proporciona una visión de la ontología centrada en los axiomas, donde una OWLOntology contiene varios objetos del tipo OWLAxiom. Implementa además métodos útiles para responder a las consultas de contención como por ejemplo "¿Contiene la ontología O a la clase C?".

El modelo de estructura de datos desarrollado en la API hace un amplio uso del patrón Visitor[5], el cual permite separar la estructura de datos de la funcionalidad. Dicho patrón facilita la tarea de agregar funcionalidades a una jerarquía de clases (y es especialmente adecuado para situaciones en donde las estructuras de datos representan sintaxis abstractas). Sin embargo, tiene el inconveniente de que los cambios en la estructura de datos son costosos, ya que requieren cambios en las implementaciones de los visitors.

OWLOntology

Además de brindar interfaces para representar entidades, expresiones de clases y axiomas, es necesario permitir a las aplicaciones gestionar las ontologías y trabajar con estas. La interfaz OWLOntology que se ve en la figura, provee un punto de acceso a los axiomas contenidos en una ontología. Como ya se mencionó, la interfaz OWLOntology puede tener diferentes implementaciones con mecanismos de almacenamiento distintos según la sintaxis de la ontología.

[5] "Visitor Pattern | Object Oriented Design - Design Patterns." 2008. 14 Ene. 2015 <http://www.oodesign.com/visitor-pattern.html>

En este sentido la interfaz OWLOntologyManager proporciona un punto central para crear, cargar, modificar y guardar ontologías, que son instancias de la interfaz OWLOntology. Cada ontología se crea o se carga mediante un gestor de ontología. Cada instancia de una ontología es única y exclusiva de un gestor en particular, y todos los cambios realizados se aplican a través de este.
Este diseño permite a las aplicaciones cliente la gestión centralizada a través de un único punto de acceso a las ontologías. El gestor también oculta gran parte de la complejidad asociada con la elección de los programas de parseo y renderizado adecuados para cargar y guardar ontologías, haciendo más simple el uso de la API.

Extensión de OWLAPI con axiomas Metamodelling:

El primer paso para extender la API de OWL consiste en agregar al modelo la implementación del nuevo axioma para metamodelling. Para esto, primero se implementa la interfaz del axioma modelado extendiendo a la clase OWLClassAxiom (clase de la que heredan todos los axiomas que relacionan clases de OWL) como se ve a continuación:

```
/*  OWLMetamodellingAxiom.java  */
    public interface OWLMetamodellingAxiom extends OWLClassAxiom {

        /**
         * Gets the ModelClass in this axiom
         * @return The class expression that represents the Model in this axiom.
         */
        OWLClassExpression getModelClass();

        /**
         * Gets the Metamodel in this axiom.
         * @return The individual that represents the Metamodel in this axiom.
         */
        OWLIndividual getMetamodelIndividual();

        /**
         * @return <code>true</code> if this axiom is a GCI, other wise <code>false</code>.
         */
        boolean isGCI();

        @Override
        OWLMetamodellingAxiom getAxiomWithoutAnnotations();
    }
```

Interfaz OWLMetamodellingAxiom.

Como segundo paso, dentro del package *uk.ac.manchester.cs.owl.owlapi* se crea la clase OWLMetamodellingAxiomImpl, la cual implementa la interfaz antes creada y a su vez extiende a OWLClassAxiomImpl. Como se puede observar en el código a continuación, el axioma se modela definiendo dos atributos, OWLClassExpression y OWLIndividual los cuales refieren a la clase y al individuo de OWL relacionados por el axioma de metamodelling.

```java
/* OWLMetamodellingAxiomImpl.java */
    public class OWLMetamodellingAxiomImpl extends OWLClassAxiomImpl implements OWLMetamodellingAxiom {

        @SuppressWarnings("unused")
        private static final Logger logger = Logger.getLogger(OWLMetamodellingAxiomImpl.class.getName());
        private static final long serialVersionUID = 30402L;

        private final OWLClassExpression modelClass;
        private final OWLIndividual metamodelIndividual;

        @SuppressWarnings("javadoc")
        public OWLMetamodellingAxiomImpl(OWLClassExpression modelClass, OWLIndividual metamodel,
            Collection<? extends OWLAnnotation> annotations) {
            super(annotations);
            this.modelClass = modelClass;
            this.metamodelIndividual = metamodel;
        }

        @Override
        public OWLMetamodellingAxiom getAnnotatedAxiom(Set<OWLAnnotation> annotations) {
            return getOWLDataFactory().getOWLMetamodellingAxiom(modelClass, metamodelIndividual, mergeAnnos(annotations));
        }

        @Override
        public OWLMetamodellingAxiom getAxiomWithoutAnnotations() {
            if (!isAnnotated()) {
                return this;
            }
            return getOWLDataFactory().getOWLMetamodellingAxiom(modelClass, metamodelIndividual);
        }

        @Override
        public OWLMetamodellingAxiom getAxiomWithoutAnnotations() {
            if (!isAnnotated()) {
                return this;
            }
            return getOWLDataFactory().getOWLMetamodellingAxiom(modelClass, metamodelIndividual);
        }

        @Override
        public OWLClassExpression getModelClass() {
            return modelClass;
        }

        @Override
        public OWLIndividual getMetamodelIndividual() {
            return metamodelIndividual;
        }


        @Override
        public AxiomType<?> getAxiomType() {
            return AxiomType.METAMODELLING;
        }
```

Clase OWLMetamodellingAxiomImpl, parte 1.

Se implementan a su vez los métodos *compareObjectOfSameType* y *equals* y *isGCI*. Los dos primeros son utilizados por la API para comparar los distintos axiomas. Mientras que el método *isGCI()* determina si el axioma es de tipo general (General Concept Inclusions) o no, y es usado por ejemplo por los razonadores OWL.

```java
/* OWLMetamodellingAxiomImpl.java */
    public class OWLMetamodellingAxiomImpl extends OWLClassAxiomImpl implements OWLMetamodellingAxiom {

        (...)

        @Override
        protected int compareObjectOfSameType(OWLObject object) {
            OWLMetamodellingAxiom other = (OWLMetamodellingAxiom) object;
            int diff = modelClass.compareTo(other.getModelClass());
            if (diff != 0) {
                return diff;
            }
            return metamodelIndividual.compareTo(other.getMetamodelIndividual());
        }

        @Override
        public boolean equals(Object obj) {
            if (!(obj instanceof OWLMetamodellingAxiom)) {
                return false;
            }
            if(super.equals(obj)) {

            OWLMetamodellingAxiom other = (OWLMetamodellingAxiom) obj;
            return other.getModelClass().equals(modelClass) && other.getMetamodelIndividual().equals(metamodelIndividual);
            }
            return false;
        }
        @Override
        public boolean isGCI() {
            return modelClass.isAnonymous();
        }
```

Clase OWLMetamodellingAxiomImpl, parte 2.

Finalmente se implementan varios métodos visitors, correspondientes a los “visitantes” de la API.

```java
/* OWLMetamodellingAxiomImpl.java */
    public class OWLMetamodellingAxiomImpl extends OWLClassAxiomImpl implements OWLMetamodellingAxiom {

        (...)

        @Override
        public void accept(OWLAxiomVisitor visitor) {
            visitor.visit(this);
        }
        @Override
        public void accept(OWLObjectVisitor visitor) {
            visitor.visit(this);
        }
        @Override
        public <O> O accept(OWLAxiomVisitorEx<O> visitor) {
            return visitor.visit(this);
        }
        @Override
        public <O> O accept(OWLObjectVisitorEx<O> visitor) {
            return visitor.visit(this);
        }

    }
```

Clase OWLMetamodellingAxiomImpl, parte 3

Siguiendo con la implementación es necesario crear métodos para cargar y guardar los axiomas de metamodelling. Para esto se implementa la clase OWLMetamodellingAxiomElementHandler que funciona

como manejador y permite utilizar el axioma de metamodelling a partir de elementos OWLXML. Esta clase extiende a la clase abstracta AbstractOWLAxiomElementHandler.

```java
/* OWLMetamodellingAxiomElementHandler.java */
    public class OWLMetamodellingAxiomElementHandler extends AbstractOWLAxiomElementHandler {

        private static final Logger logger = Logger.getLogger(OWLMetamodellingAxiomElementHandler.class.getName());

        private OWLClassExpression modelClass;
        private OWLIndividual metaIndividual;

        public OWLMetamodellingAxiomElementHandler(OWLXMLParserHandler handler) {
            super(handler);
        }

        @Override
        public void startElement(String name) throws OWLXMLParserException {
            super.startElement(name);
            modelClass = null;
            metaIndividual = null;
        }

        @Override
        public void handleChild(AbstractClassExpressionElementHandler handler) {
               modelClass = handler.getOWLObject();
        }

        @Override
        public void handleChild(OWLIndividualElementHandler handler) throws OWLXMLParserException {
            metaIndividual = handler.getOWLObject();
        }

        @Override
        protected OWLAxiom createAxiom() throws OWLXMLParserException {
            return getOWLDataFactory().getOWLMetamodellingAxiom(modelClass, metaIndividual, getAnnotations());
        }
    }
```

Como último paso importante se agrega al vocabulario de OWL/XML la sintaxis para metamodelling como se ve en la siguiente imagen, y se implementan los métodos para cargar y grabar los axiomas en un archivo.

```java
/* OWLXMLVocabulary.java */
    public enum OWLXMLVocabulary {

        (...)

        CLASS("Class"),

        DATA_PROPERTY("DataProperty"),

        OBJECT_PROPERTY("ObjectProperty"),

        NAMED_INDIVIDUAL("NamedIndividual"),

        METAMODELLING("MetaModelling"),

        (...)
    }
```

Tipo METAMODELLING en OWLXMLVocabulary.

En la clase OWLXMLObjectRenderer el método para grabar el axioma según la sintaxis definida:

```
/* OWLXMLObjectRenderer.java */
    public class OWLXMLObjectRenderer implements OWLObjectVisitor {
        (...)
        @Override
        public void visit(OWLMetamodellingAxiom axiom) {
            writer.writeStartElement(METAMODELLING);
            writeAnnotations(axiom);
            axiom.getMetamodelIndividual().accept(this);
            axiom.getModelClass().accept(this);
            writer.writeEndElement();
        }
        (...)
    }
```

y el método en OWLXMLParserHandler que permite cargar los axiomas desde un archivo:

```
/* OWLXMLParserHandler.java */
    public class OWLXMLParserHandler extends DefaultHandler {
        (...)
        addFactory(new AbstractElementHandlerFactory(METAMODELLING) {
            @Override
            public OWLElementHandler<?> createHandler(OWLXMLParserHandler handler) {
                return new OWLMetamodellingAxiomElementHandler(handler);
            }
        });
        (...)
    }
```

Las 3 nuevas clases, junto con los métodos agregados para el manejo , parseo y el renderizado del axioma dentro de una ontología son la parte principal de la extensión. Además de esto es necesario implementar otros métodos como por ejemplo los visitors para metamodelling en distintas partes de la API. La lista completa de las clases modificadas se presenta en el anexo I.

Con la versión extendida de la API de OWL es posible cargar y grabar ontologías que utilicen axiomas de metamodelling, dichos axiomas serán definidos en el archivo con la siguiente sintaxis:

```
<MetaModelling>
    <NamedIndividual IRI="#a"/>
    <Class IRI="#A"/>
</MetaModelling>
```

Relación metamodelling entre individuo ***a*** y clase ***A*** en OWL/XML.

Definiendo un elemento de tipo *NamedIndividual* que será el meta-modelo del otro elemento definido del tipo *Class*. Ambos contenidos entre las etiquetas de *MetaModelling.*

3.2. OntoRed plugin

Protégé como se mencionó antes, es un editor de ontologías con soporte completo de OWL 2 (*Web Ontology Language)* que además brinda, mediante software de terceros, razonadores de lógica descriptiva. La versión de Protégé más reciente al momento de realizarse el presente trabajo es la versión 5.0, pero la misma se encuentra en fase beta. Por lo tanto se escogió trabajar sobre la última versión estable, la 4.3 de abril de 2013.

La arquitectura de Protégé 4.3 está diseñada de forma modular en base a la infraestructura OSGi[6] logrando extensibilidad que le permite agregar nuevas funcionalidades fácilmente mediante plugins. De esta forma los servicio principales de Protégé 4.3 se encuentran dentro de un "core" y las demás funcionalidades como son los editores de clases, vistas de ontologías, razonadores o consultas SWRL se distribuyen como plugins.

Para este trabajo la extensión de Protégé que permite trabajar con redes ontológicas y axiomas de metamodelling, se implementa mediante un plugin del tipo "Tab", el cual se presenta en el programa como una nueva pestaña.
En dicho plugin se busca presentar de la forma más clara, directa y simple posible las herramientas que tiene el usuario disponible para la creación y edición de las redes de ontologías.

Estructura de un plugin en Protégé:

Todo plugin en Protégé tiene una estructura básica común basada en el modelo OSGi.
En la Figura 7 se puede observar cómo está compuesto el plugin OntoRed a modo de ejemplo.

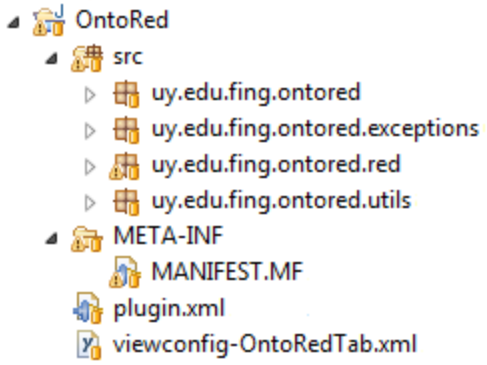


Figura 7. Estructura de un plugin de Protégé.

Todo plugin está compuesto por un lado por las clases Java agrupadas en los diferentes packages dentro de la carpeta *src*, y por otro lado ciertos archivos de configuración.
Estos archivos de configuración tienen distintos objetivos.

El archivo *MANIFEST.MF* es en general igual en todos los plugins. En él se anotan las librerías o package que se utilizan en el proyecto y los plugins requeridos de Protégé para ser ejecutado. También se definen ciertos campos que si varían, referidos al nombre del plugin, su descripción o el desarrollador.

[6] "OSGi Alliance | Main / OSGi Alliance." 12 Mar. 2015 <http://www.osgi.org/>

La función del archivo *plugin.xml* es declarar de qué forma el nuevo plugin extiende a Protégé, y a su vez, declarar de qué forma puede este plugin ser extendido por otros.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?eclipse version="3.0"?>

<plugin>
   <extension id="View"
              point="org.protege.editor.core.application.ViewComponent">
      <label value="OntoRed"/>
      <class value="uy.edu.fing.ontored.OntoRedViewComponent"/>
      <category value="@org.protege.ontologycategory"/>
   </extension>
   <extension id="ExampleWorkspaceTab"
              point="org.protege.editor.core.application.WorkspaceTab">
      <label value="OntoRed Plugin"/>
      <class value="org.protege.editor.owl.ui.OWLWorkspaceViewsTab"/>
      <index value="X"/>
      <editorKitId value="OWLEditorKit"/>
      <defaultViewConfigFileName value="viewconfig-OntoRedTab.xml"/>
   </extension>
</plugin>
```

Archivo plugin.xml de OntoRed

En la imagen anterior se muestra el archivo plugin.xml de OntoRed. En él se declaran dos extensiones sobre Protégé. La primera parte hace referencia a que se implementa un *ViewComponent* mediante la clase *OntoRedViewComponent*. La segunda extensión declara que se trata de un plugin de tipo pestaña (tab) que tiene una vista definida en el archivo *viewconfig-OntoRedTab.xml* y que el nombre en la lista de tabs de Protégé es *"OntoRed Plugin"*.

Finalmente los plugins de tipo *tab*, deben incluir un archivo XML que configure la vista del mismo. En el caso de OntoRed el archivo se llama *viewconfig-OntoRedTab.xml* y se puede ver en la siguiente imagen:

```
<?xml version="1.0" encoding="UTF-8"?>
<layout>
    <VSNode splits="0.15 0.85">

        <CNode>
            <Component label="Asserted hierarchy">
                <Property id="pluginId"
                    value="org.protege.editor.owl.OWLAssertedClassHierarchy" />
            </Component>
        </CNode>
        <CNode>
            <Component label="OntoRed Plugin">
                <Property id="pluginId" value="OntoRed.View" />
            </Component>
        </CNode>
    </VSNode>
</layout>
```

Archivo *viewconfig-OntoRedTab.xml.*

Este archivo es referenciado en el *plugin.xml* y es el encargado de configurar de qué forma se visualiza el plugin dentro de Protégé. En el caso de OntoRed la vista está configurada en dos secciones verticales

(<*VSNode*>) una de 15% y la otra del 85% del tamaño de la pantalla de Protégé. En la sección de la izquierda se reutiliza un componente propio de Protégé como es la vista de jerarquía de clases, para que el usuario del plugin pueda ver el árbol de clases sin tener que cambiar de pestaña. En la sección derecha se muestra la vista implementada por la clase OntoRedViewComponent cómo se define en el archivo plugin.xml, referenciada mediante *OntoRed.View*. Con la propiedad *label* se define el título que tiene cada sección.

Arquitectura del plugin OntoRed:

La arquitectura interna del plugin OntoRed está organizada basicamente en la clase OntoRedViewComponent encargada de la interfaz y su comportamiento, y el package *uy.edu.fing.red* en el cual se encuentran las clases que modelan la red y sus relaciones. Además están los packages *exceptions* encargado de definir las excepciones que lanza el programa y el package *utils* donde se implementan métodos auxiliares del plugin.

Figura 8. Distribución de packages

Clase OntoRedViewComponent

A través de la clase **OntoRedViewComponent** se realiza toda la interacción entre el usuario y el plugin. La clase extiende a *AbstractOWLViewComponent*, que es la clase encargada de la interfaz en los plugins dentro de la arquitectura de Protégé. Los métodos que necesariamente se deben implementar son *disposeOWLView()* y *initialiseOWLView()* ya que son los encargados de inicializar y destruir la vista del plugin. El resto de los métodos que se implementan se encargan de generar los elementos de la interfaz y el comportamiento de los mismos. Dichos componentes son elementos estándar de las librerías de **Java AWT** y **Swing**.

Package Red:

En el package ***red*** encontramos, la clase ***Red*** que modela la red de ontologías, la interfaz ***Relacion*** que define el comportamiento común de todas las relaciones y finalmente una clase por cada relación que implementa el comportamiento de estas.

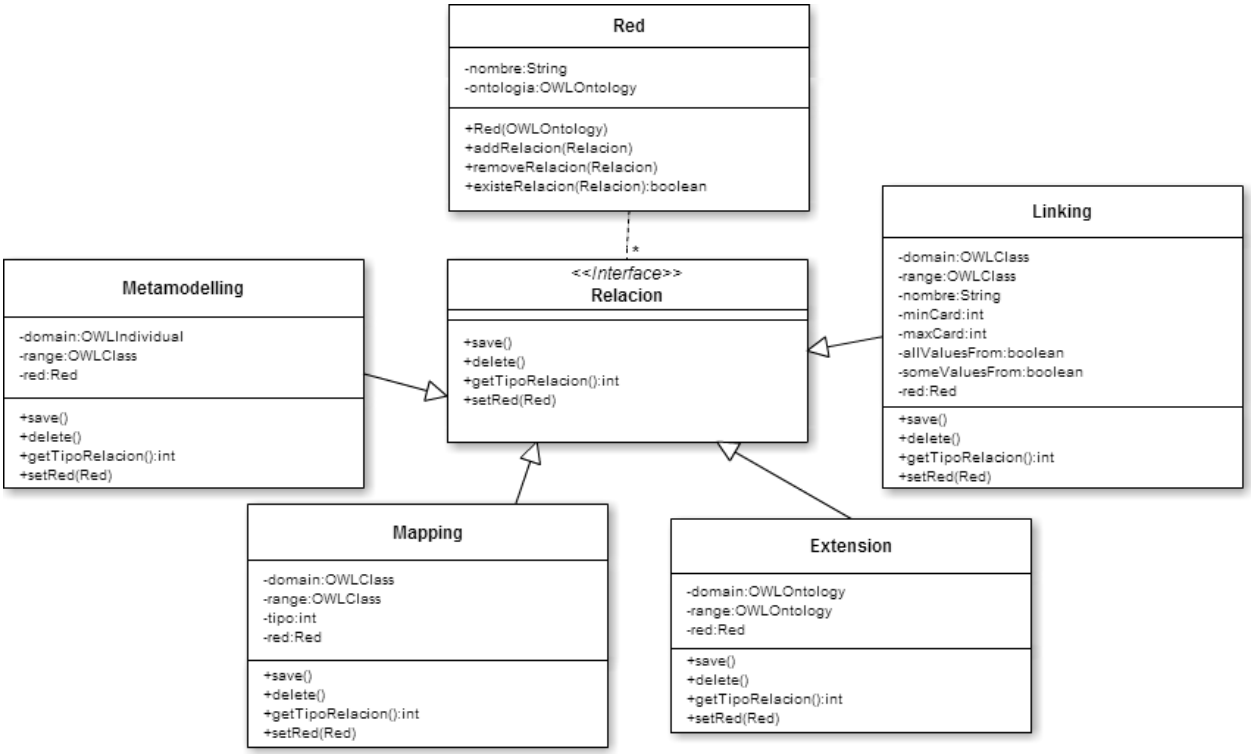


Figura 9. Modelo de clases

La clase ***Red*** tiene como atributos un nombre, la ontología activa en Protégé que modela la red (tipo OWLOntology) y una lista de tipo ***Relacion*** que representa las relaciones de la red. Aparte del constructor los métodos de la clase son útiles para el manejo de la colección de relaciones.

La interfaz ***Relacion*** y las clases que la implementan poseen la operación ***save()*** y ***delete()***. El método *save* se encarga de crear la relación comprobando las restricciones definidas y agregarla a la red, mientras que el método *delete* se utiliza para eliminar una relación, el mismo se encarga de destruir las estructuras asociadas y quitar la relación de la red.

Las clases que implementan las relaciones mantienen como atributos solo lo necesario, esto es el objeto dominio, el objeto rango de la relación y la red a la que pertenecen. Para el caso del Linking se agregan además el nombre de la relación y atributos para definir los cuantificadores. También se agrega el atributo tipo para la clase Mapping ya que dicha relación puede ser *subClassOf* ó *equivalentTo.*

La implementación más relevante de cada relación son los constructores, y los métodos save y delete. A continuación se describe para cada una de las relaciones dichos métodos.

Extension

La relación de extensión relaciona dos ontologías, sus métodos de save y delete están vacíos ya que se utiliza el servicio de importación de ontologías de Protégé para crear y eliminar relaciones de este tipo.

```
public Extension(OWLOntology domain, OWLOntology range, Red red) {
    super();
    this.domain = domain;
    this.range = range;
    this.red = red;
}

public void save() throws ExtensionException {

}

public void delete() throws EliminarException {

}
```

Clase Extension

Mapping

La relación mapping puede ser de dos tipos, el tipo de la misma es indicado por un entero. Según el tipo, se creará un axioma de subClass o de equivalentTo tanto en save como en delete para agregarlo o quitarlo de la ontologia. Como se puede observar los cambios realizados sobre las ontologías se hacen a través de una instancia de la clase OWLDataFactory la cual centraliza todas las modificaciones.

```
public Mapping(OWLClass domain, OWLClass range, int tipo, Red red) {
    super();
    this.domain = domain;
    this.range = range;
    this.tipo = tipo;
    this.red = red;
}

public void save() {
    OWLOntologyManager manager = FactoryOWLManager.getOWLManager().getOWLOntologyManager();
    OWLDataFactory factory = manager.getOWLDataFactory();
    //Creao la relacion MAPPING segun el tipo elegido
    OWLAxiom axiom;
    if(this.tipo==TIPO_EQUIVALANTE)
        axiom = factory.getOWLEquivalentClassesAxiom(this.domain, this.range);
    else
        axiom = factory.getOWLSubClassOfAxiom(this.domain, this.range);
    //Agrego el axioma de Mapping a la red y aplico los cambios.
    AddAxiom addAxiom = new AddAxiom(this.red.getOntologia(), axiom);
    manager.applyChange(addAxiom);
}

public void delete() {
    OWLOntologyManager manager = FactoryOWLManager.getOWLManager().getOWLOntologyManager();
    OWLDataFactory factory = manager.getOWLDataFactory();
    // Obtengo la relacion de Mapping segun su tipo
    OWLAxiom axiom;
    if(this.tipo==TIPO_EQUIVALANTE)
        axiom = factory.getOWLEquivalentClassesAxiom(this.domain, this.range);
    else
        axiom = factory.getOWLSubClassOfAxiom(this.domain, this.range);
    //Elimino la relacion de la red
    manager.removeAxiom(this.red.getOntologia(), axiom);
    this.red.removeRelacion(this);
}
```

Clase Mapping

Linking

La relación linking es la que posee más cantidad de atributos, ya que se trata de una relación que maneja restricciones. Esta es la única relación en la cual es necesario crear más de un axioma, dado que para definir cada restricción se define uno nuevo. Los métodos save y delete funcionan de forma análoga, primero se obtienen los axiomas y luego se crean o se eliminan. A diferencia de las otras relaciones, aquí utilizan los métodos *addAxioms* y *removeAxioms* por tratarse de una lista de axiomas.

```
public Linking(String nombre, OWLClass domain, OWLClass range, Red red, int min, int max, boolean some, boolean all) {
    super();
    this.nombre = nombre;
    this.domain = domain;
    this.range = range;
    this.red = red;
    this.allValuesFrom = all;
    this.someValuesFrom = some;
    this.minCard = min;
    this.maxCard = max;
}

public void save() {
    OWLOntologyManager manager = FactoryOWLManager.getOWLManager().getOWLOntologyManager();
    IRI ontoIRI = this.red.getOntologia().getOntologyID().getOntologyIRI();
    OWLDataFactory factory = manager.getOWLDataFactory();
    Set<OWLAxiom> axioms = new HashSet<OWLAxiom>();
    //Creo la nueva propiedad-LINKING/ObjectProperty-
    OWLObjectProperty prop = factory.getOWLObjectProperty(IRI.create(ontoIRI + "#" + this.nombre));
    //Defino el Dominio
    axioms.add(factory.getOWLObjectPropertyDomainAxiom(prop, this.domain));
    //++++ RESTRICCIONES ++++
    boolean sinRestricciones = true;
    if(this.someValuesFrom){
        sinRestricciones = false;
        OWLClassExpression propSome = factory.getOWLObjectSomeValuesFrom(prop, range);
        axioms.add(factory.getOWLObjectPropertyRangeAxiom(prop, propSome));
    }
    if(this.allValuesFrom){
        sinRestricciones = false;
        OWLClassExpression propAll = factory.getOWLObjectAllValuesFrom(prop, range);
        axioms.add(factory.getOWLObjectPropertyRangeAxiom(prop, propAll));
    }
    if(this.minCard>-1){
        sinRestricciones = false;
        OWLClassExpression propMinCard = factory.getOWLObjectMinCardinality(minCard, prop, range);
        axioms.add(factory.getOWLObjectPropertyRangeAxiom(prop, propMinCard));
    }
    if(this.maxCard>-1){
        sinRestricciones = false;
        OWLClassExpression propMaxCard = factory.getOWLObjectMaxCardinality(maxCard, prop, range);
        axioms.add(factory.getOWLObjectPropertyRangeAxiom(prop, propMaxCard));
    }
    //Defino el Rango
    if(sinRestricciones){
        axioms.add(factory.getOWLObjectPropertyRangeAxiom(prop, this.range));
    }
    //Agrego el axioma "Linking" a la red.
    manager.addAxioms(this.red.getOntologia(), axioms);
}

public void delete() {
    OWLOntologyManager manager = FactoryOWLManager.getOWLManager().getOWLOntologyManager();
    OWLDataFactory factory = manager.getOWLDataFactory();
    IRI ontoIRI = this.red.getOntologia().getOntologyID().getOntologyIRI();
    Set<OWLAxiom> axioms = new HashSet<OWLAxiom>();
    //Obtengo la propiead -LINKING-
    OWLObjectProperty prop = factory.getOWLObjectProperty(IRI.create(ontoIRI + "#" + this.nombre));
    //Obtengo el Axioma de dominio
    axioms.add(factory.getOWLObjectPropertyDomainAxiom(prop, this.domain));
    //Obtengo los Axiomas de rango
    boolean sinRestricciones = true;
    if(this.someValuesFrom){
        sinRestricciones = false;
        OWLClassExpression propSome = factory.getOWLObjectSomeValuesFrom(prop, range);
        axioms.add(factory.getOWLObjectPropertyRangeAxiom(prop, propSome));
    }
    if(this.allValuesFrom){
        sinRestricciones = false;
        OWLClassExpression propAll = factory.getOWLObjectAllValuesFrom(prop, range);
        axioms.add(factory.getOWLObjectPropertyRangeAxiom(prop, propAll));
    }
    if(this.minCard>-1){
        sinRestricciones = false;
        OWLClassExpression propMinCard = factory.getOWLObjectMinCardinality(minCard, prop, range);
        axioms.add(factory.getOWLObjectPropertyRangeAxiom(prop, propMinCard));
    }
    if(this.maxCard>-1){
        sinRestricciones = false;
        OWLClassExpression propMaxCard = factory.getOWLObjectMaxCardinality(maxCard, prop, range);
        axioms.add(factory.getOWLObjectPropertyRangeAxiom(prop, propMaxCard));
    }
    if(sinRestricciones){
        axioms.add(factory.getOWLObjectPropertyRangeAxiom(prop, this.range));
    }
    // Elimino la relacion Linking de la red
    manager.removeAxioms(this.red.getOntologia(), axioms);
    this.red.removeRelacion(this);
}
```

Clase Linking

Metamodelling

La última relación, el metamodelling relaciona un individuo y una clase. Para sus métodos save y delete se utiliza la extensión realizada sobre la API de OWL. Aquí es necesario invocar a los nuevos métodos implementados para crear el axioma de metamodelling. El axioma se agrega o elimina a la ontología de la misma forma que en las demás relaciones.

```
public Metamodeling(OWLIndividual domain, OWLClass range, Red red) {
    super();
    this.domain = domain;
    this.range = range;
    this.red = red;
}

public void save() {
    OWLOntologyManager manager = FactoryOWLManager.getOWLManager().getOWLOntologyManager();
    OWLDataFactory factory = manager.getOWLDataFactory();
    //Creo la relacion de Meta-modeling
    OWLAxiom axiom = factory.getOWLMetamodellingAxiom(range, domain);
    //Agrego la relacion a la red y aplico los cambios
    AddAxiom addAxiom = new AddAxiom(this.red.getOntologia(), axiom);
    manager.applyChange(addAxiom);
}

public void delete() {
    OWLOntologyManager manager = FactoryOWLManager.getOWLManager().getOWLOntologyManager();
    OWLDataFactory factory = manager.getOWLDataFactory();
    //Obtengo la relacion
    OWLAxiom axiom = factory.getOWLMetamodellingAxiom(range, domain);
    //Elimino el metamodeling de la red
    manager.removeAxiom(this.red.getOntologia(), axiom);
    this.red.removeRelacion(this);
}
```

Clase Metamodelling.

3.3. Razonador Pellet modificado

El razonador Pellet es desarrollado y mantenido por el grupo Clark&Parsia[7]. Su código fuente es accesible a través de su repositorio público en GitHub.

El proyecto Pellet, en su versión 2.3.1, se compone de varios módulos como se muestra en la Figura 10.

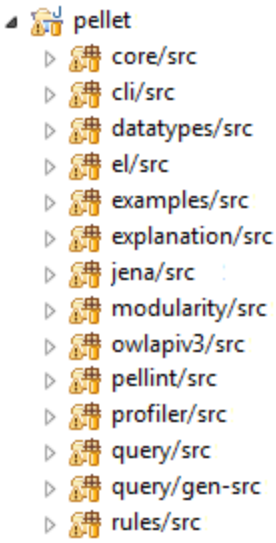


Figura 10. Módulos de Pellet.

La clase principal del razonador se encuentra en el módulo *cli*. En este módulo se implementa el cliente standalone de Pellet, el cual se encarga de dialogar con el usuario mediante los comandos definidos:

- classify
- consistency
- realize
- unsat
- explain
- query
- modularity
- trans-tree
- extract

En el caso del presente trabajo, la extensión de Pellet apunta a modificar el algoritmo de chequeo de consistencia de las ontologías. Este chequeo no solo se realiza al ejecutar el comando *consistency*, la consistencia de las ontologías también se comprueba de forma implícita al ejecutar los demas comandos.

El módulo principal de Pellet, donde se implementa la mayor parte de la lógica, es el módulo *core*. En este módulo se encuentra modelada la o las ontologías y sus relaciones sobre las cuales se realizan los chequeos.

[7] "Pellet: OWL 2 Reasoner for Java - Clark & Parsia." 2008. 20 Mar. 2015 <http://clarkparsia.com/pellet/>

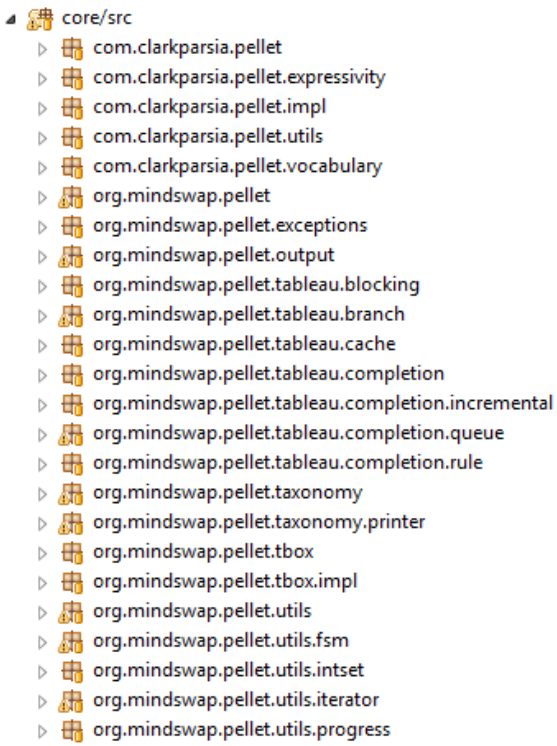


Figura 11. Packages del módulo Core.

El punto central del módulo es la clase *KnowledgeBase*. Esta clase centraliza todos los elementos de las ontologías cargadas. La ontología es cargada en un objeto *KnowledgeBase* mediante la clase KBLoader que identifica el tipo de la ontología y define qué loader debe utilizar. En nuestro caso, las ontologías son OWL y específicamente en formato OWL/XML, por lo tanto se cargan utilizando el módulo *owlapiv3* de Pellet.
En dicho módulo existe una clase llamada *PelletVisitor* que como su nombre indica, hace uso del patrón Visitor y se encarga de cargar las relaciones de la ontología en la *KnowledgeBase.*

La primer modificación para extender el razonador consta de sustituir la API OWL que trae Pellet, por la versión modificada. Luego, para cargar los axiomas de metamodelling que posee la ontología, se implementa el método del Visitor para OWLMetamodellingAxiom.

```
public class PelletVisitor implements OWLObjectVisitor {

    (...)

    public void visit(OWLMetamodellingAxiom axiom) {
        axiom.getModelClass().accept(this);
        ATermAppl clase = term;
        axiom.getMetamodelIndividual().accept(this);
        ATermAppl ind = term;

        if (addAxioms) {
            kb.addMetaModeling(ind, clase);
        }
        else {
            ATermAppl metaAxiom = ATermUtils.makeMeta(ind, clase);
            // reload is required if remove fails
            reloadRequired = !kb.removeAxiom(metaAxiom);
        }
    }
}
```

Implementación del Visitor para metamodelling.

Los axiomas se procesan a través de *PelletVisitor* y se cargan en la *KnowledgeBase.* *KnowledgeBase* divide los axiomas en dos conjuntos según su tipo, por un lado las aserciones (ABox) y por otro lado los términos (TBox). En el modelo de Pellet cada conjunto se modela con una clase, y además la clase *TBox* posee dos tipos de conjuntos, *TuBox* (para los términos de unfolding) y *TgBox* para los axiomas generales.

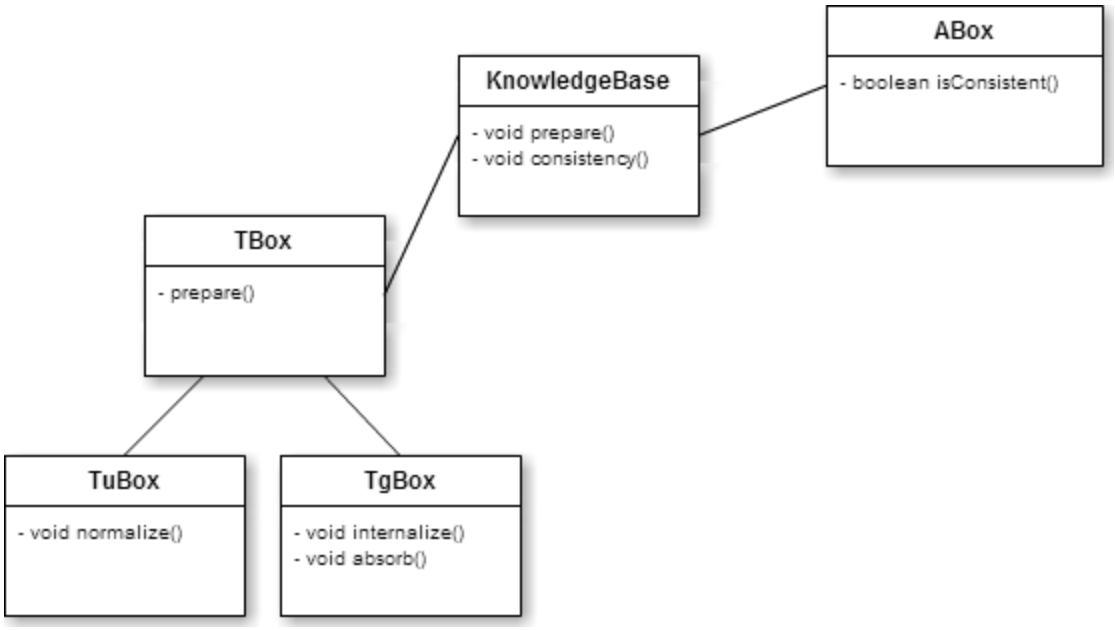


Figura 12. Modelo de clases resumido en Core sin modificar.

En este punto la extensión al modelo consiste en asociar a la *KnowledgeBase* el nuevo conjunto de axiomas llamado *MBox*, para el cual se crea una clase con el mismo nombre.
Este último agregado da como resultado el modelo de la Figura 13.

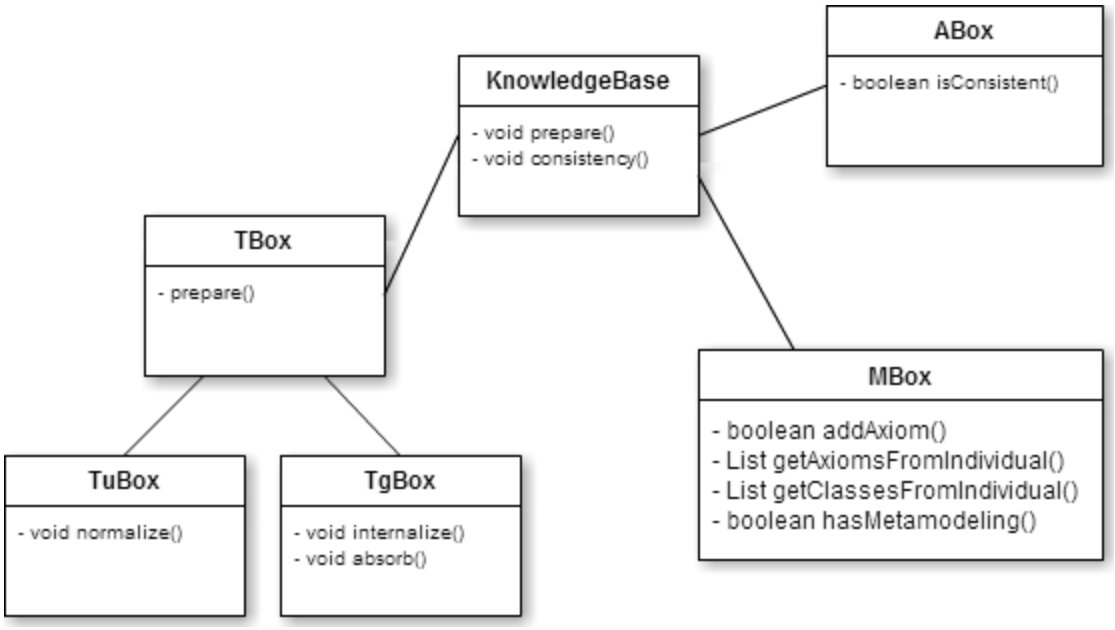


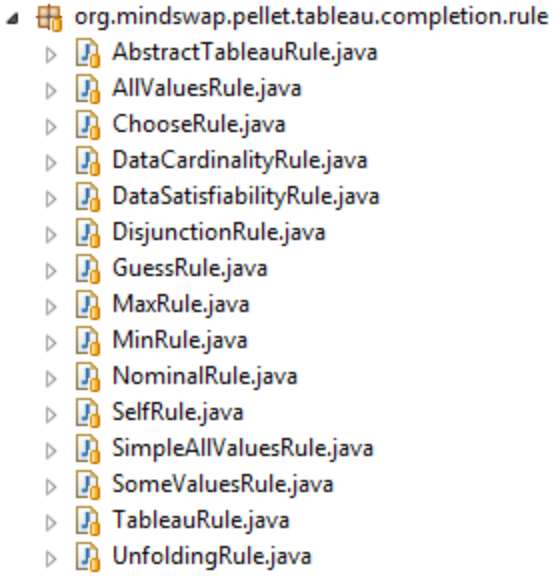
Figura 13. Modelo de clases resumido en Core modificado.

Mediante la clase *MBox* se mantienen los axiomas de metamodelling y se trabajan con los mismos. A su vez, la clase brinda operaciones útiles para aplicar las nuevas reglas.

Una vez que el razonador carga la ontología y sus axiomas en las clases correspondientes se prepara para ejecutar las reglas de expansión y chequear la consistencia. Para esto, en primer lugar calcula la expresividad de la ontología, es decir, la lógica descriptiva en la que está expresada (por ejemplo ALCQ), y con esto define el conjunto de reglas que debe aplicar. En este paso se modifica tanto el cálculo de expresividad como la definición de reglas de forma que, si existen axiomas de metamodelling, se cambie la lógica (por ejemplo ALCQM) y se incorporen las nuevas reglas.
A continuación, Pellet ejecuta el conjunto de reglas definido continuamente hasta alcanzar un estado final consistente ó hasta que se sature el tableau y no pueda ejecutar más reglas. Si alcanza un estado inconsistente (encuentra un *clash*), el razonador intentará volver a un estado consistente anterior y continuar por otra rama del razonamiento, descartando la última.
Todas las reglas se intentan ejecutar para todos los nodos. Al comienzo del razonamiento los únicos nodos son los individuos definidos en la ontología, pero a medida que se ejecutan las reglas se pueden crear nuevos nodos anónimos que no corresponden a un individuo. Esto ocurre por ejemplo al ejecutar la regla *MinRule* ($\geq$).
Cada nodo en *Pellet* (clase *Node*), mantiene un conjunto de dependencias de los axiomas que tiene relacionados y a los que se les aplicó una regla. Esto, es utilizado al momento de aplicar una regla para saber si esta es o no aplicable sobre un nodo determinado. Además del conjunto de dependencias sobre los axiomas, los nodos mantienen otro tipo de dependencias. Estas últimas marcan en qué branch del razonamiento se ejecutó una regla y que otros branches dependen de ese nodo. Estas dependencias son

útiles para que el razonador pueda mantener el árbol de deducción y pueda hacer backtrack para volver a buscar un estado consistente anterior.

Las reglas implementadas en Pellet se encuentran en el módulo *Core* en el package *org.mindswap.pellet.tableau.completion.rule*. Para extender el Pellet y razonar sobre axiomas con metamodelling, se extiende el conjunto implementando las reglas *EqualRule, NotEqualRule* y *CloseRule.* El comportamiento de estas reglas está definido en la sección "*2.2 Lógica ALCQM"*.



Reglas originales de Pellet.

Regla Equal:

La regla Equal, implementada en la clase *EqualRule* se ejecuta solo para aquellos nodos que tengan un axioma de metamodelling asociado y posean otro nodo con metamodelling relacionado por la igualdad. Si esto ocurre, y las clases a las que están asociados por metamodelling son distintas, dichas clases deben ser equivalentes. En este caso para optimizar el algoritmo no se agrega un nuevo axioma a la TBox, en vez de eso se agrega la igualdad de clases a cada individuo directamente.

```
public class EqualRule extends AbstractTableauRule{
    public void apply(Individual ind) {
        if (!strategy.getMBox().hasMetamodeling(ind.getTerm()))
            return;
        List<Node> sames = new ArrayList<Node>(ind.getMerged());
        sames.add(ind);
        for (Node x: sames)
            if (strategy.getMBox().hasMetamodeling(x.getTerm()))
                for(ATermAppl c1: strategy.getMBox().getClassesFromIndividual(ind.getTerm()))
                    for(ATermAppl c2: strategy.getMBox().getClassesFromIndividual(x.getTerm())){
                        if((!c1.equals(c2)) && (!strategy.getMBox().existsEqualClasses(c1, c2))){
                            DependencySet ds = DependencySet.INDEPENDENT;
                            ATermAppl aux= ATermUtils.makeValue(x.getTerm());
                            ds = ds.union(ind.getDepends(aux), strategy.getABox().doExplanation());
                            if(addEquivalentClass(c1, c2, ds)){
                                strategy.getMBox().addEqualClasses(c1, c2);
                            }
                        }
                        if( strategy.getABox().isClosed() ){
                            return;
                        }
                    }
    }
    /**
     * Retorna TRUE si existe algun individuo Z que contenga la clase c en su grafo
     */
    public boolean addEquivalentClass(ATermAppl c1, ATermAppl c2, DependencySet ds){
        IndividualIterator iter =strategy.getABox().getIndIterator();
        while(iter.hasNext()){
            Individual x = iter.next();
            if(x.getTypes().contains(c1))
                //Agrego la clase c2 al grafo del nodo x.
                strategy.addType(x, c2, ds);
            else if(x.getTypes().contains(ATermUtils.negate(c1)))
                //Agrego la clase notC2 al grafo del nodo x.
                strategy.addType(x, ATermUtils.negate(c2), ds);
            else if(x.getTypes().contains(c2))
                //Agrego la clase c1 al grafo del nodo x.
                strategy.addType(x, c1, ds);
            else if(x.getTypes().contains(ATermUtils.negate(c2)))
                //Agrego la clase notC1 al grafo del nodo x.
                strategy.addType(x, ATermUtils.negate(c1), ds);
        }
        return false;
    }
}
```

Clase *EqualRule*.

Regla NotEqual:

La clase que implementa esta regla, es *NotEqualRule*. Para esta regla se toman en cuenta los individuos que poseen una relación de metamodelling y a su vez tiene definido como diferente algun otro individuo que tambien posee metamodelling. Bajo estas condiciones las clases relacionados por metamodelling a estos individuos deben ser diferentes. Para lograr esto, como indica la definición de la regla, se debe crear un nuevo nodo z tal que $L(z) = \{A \sqcap \neg B \sqcup B \sqcap \neg A\}$.

En la imagen siguiente se puede ver como en el método *generarClase(a, b)*, se crea la expresión mencionada para posteriormente agregarla al grafo del nuevo nodo.

```
public class NotEqualRule extends AbstractTableauRule{
    public void apply(Individual ind) {
        List<Node> diffs = new ArrayList<Node>(ind.getDifferents());
        if (!strategy.getMBox().hasMetamodeling(ind.getTerm()) || diffs.isEmpty())
            return;
        for (Node x: diffs)
            if (strategy.getMBox().hasMetamodeling(x.getTerm()))
                for(ATermAppl c1: strategy.getMBox().getClassesFromIndividual(ind.getTerm()))
                    for(ATermAppl c2: strategy.getMBox().getClassesFromIndividual(x.getTerm())){
                        if (!strategy.getMBox().existsNotEqualClasses(c1, c2)){
                            ATermAppl clase = generarClase(c1, c2);
                            DependencySet ds = new DependencySet(strategy.getABox().getBranch()-1);
                            if(!existsZ(clase)){
                                Individual z = strategy.createFreshIndividual( ind, ds);
                                strategy.addType( z, clase, ds);
                                strategy.getMBox().addNotEqualClasses(c1, c2);
                            }
                        }
                        if( strategy.getABox().isClosed() )
                            return;
                    }
    }
    /**
     * Genera una clase de la fomra (A ^ -B v B ^ -A) a partir de A y B
     */
    public ATermAppl generarClase(ATermAppl a, ATermAppl b){
        ATermAppl notA = ATermUtils.makeNot(a);
        ATermAppl notB = ATermUtils.makeNot(b);
        ATermAppl clase = ATermUtils.makeOr(ATermUtils.makeAnd(a, notB), ATermUtils.makeAnd(notA, b));
        return ATermUtils.normalize(clase);
    }
    /**
     * Retorna TRUE si existe algun individuo Z que contenga la clase c en su grafo
     */
    public boolean existsZ(ATermAppl c){
        IndividualIterator iter =strategy.getABox().getIndIterator();
        while(iter.hasNext()){
            Individual x = iter.next();
            if(x.getTypes().contains(c))
                return true;
        }
        return false;
    }
}
```

Clase *NotEqualRule*.

Regla Close:

Por su parte la regla Close, se encuentra implementada en dos clases, *CloseRule* y *CloseBranch*. Como ocurre con algunas de las reglas originales, como por ejemplo la regla *MaxRule* o *DisjunctionRule*, la regla Close se divide en varios branches, en este caso dos.

Esta regla, según su definición, va a generar un axioma de diferencia ó de igualdad para todos aquellos nodos que posean metamodelling y no tengan una “relación” entre ellos. Esta relación hace referencia a que para dos nodos exista un axioma que los iguale o que los diferencie. Esta relación es detectada por el método *hayRelacion(a, b)* implementado en *CloseRule* como se ve en la imagen siguiente.

```
public class CloseRule extends AbstractTableauRule{

    public CloseRule(CompletionStrategy strategy) {
        super(strategy, NodeSelector.CLOSE, AbstractTableauRule.BlockingType.INDIRECT);
    }
    public void apply(Individual ind) {
        if (!strategy.getMBox().hasMetamodeling(ind.getTerm()))
            return;

        IndividualIterator iter =strategy.getABox().getIndIterator();
        while(iter.hasNext()){
            Individual x = iter.next();
            if(x.equals(ind) || !strategy.getMBox().hasMetamodeling(x.getTerm())){
                continue;
            }
            // x tiene metamodeling
            if (!hayRelacion(ind, x)){
                // Agrego nueva relacion entre ind y x, equate o diff
                CloseBranch newBranch = new CloseBranch(strategy.getABox(), strategy, ind, x, ind.getDepends(x.getTerm()), 2);
                strategy.addBranch( newBranch );
                newBranch.tryNext();
                if( strategy.getABox().isClosed() )
                    return;
            }
        }
    }
    public boolean hayRelacion(Individual a, Individual b){
        List<Node> inds= new ArrayList<Node>();
        inds.add(a);
        for (int i= 0; i< inds.size(); i++){
            Set<Node> diffs =a.getDifferents();
            Set<Node> sames =a.getMerged();
            if(diffs.contains(b) || sames.contains(b))
                return true;
            agregarNuevos(inds, sames);
        }
        return false;
    }
    public void agregarNuevos(List<Node> inds, Set<Node> sames){
        if (sames == null || sames.isEmpty())
            return;
        for(Node n: sames)
            if(!inds.contains(n))
                inds.add(n);
    }
}
```

Clase *CloseRule.*

Si no existe relación entre dos nodos, se les aplica la regla, generando el branch implementado por *CloseBranch*.
CloseBranch prueba primero generando un axioma de igualdad para los nodos, y en el caso que se llegue a un resultado inconsistente, mediante *backtrack* se vuelve a ejecutar el branch pero esta vez generado el axioma de diferencia entre los nodos.

```
public class CloseBranch extends Branch{
    private Node a;
    private Node b;

    public CloseBranch(ABox abox, CompletionStrategy strategy, Node a, Node b, DependencySet ds, int n) {
        super(abox, strategy, ds, n);
        this.a= a;
        this.b= b;
    }
    protected void tryBranch() {
        abox.incrementBranch();
        ATermAppl sameAxiom = ATermUtils.makeSameAs( a.getTerm(), b.getTerm() );
        if( PelletOptions.USE_INCREMENTAL_DELETION ) {
            strategy.getABox().getKB().getSyntacticAssertions().add( sameAxiom );
        }
        if (a.isMerged())
            a = a.getSame();
        if (b.isMerged())
            b = b.getSame();

        DependencySet ds = new DependencySet(getBranch());
        // El branch se llama dos veces, tryNext = 0 -> hago merge. tryNext = 1 -> hago diff.
        if(getTryNext() == 0)
            strategy.mergeTo(a, b, ds); // a = b
        else
            a.setDifferent(b, ds); // a != b
    }
    public Node getNode() {
        return this.a;
    }
}
```

Clase *CloseBranch.*

Chequeo de ciclos:

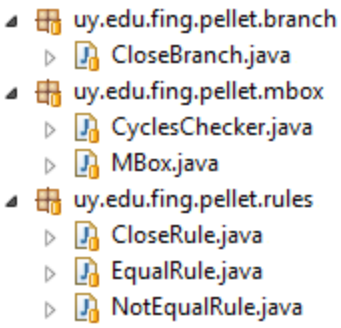
El último paso para extender el razonador es implementar el chequeo de ciclos de metamodelling. Un ciclo de metamodelling se da, cuando un individuo que es meta-modelo de una clase pertenece a ésta o alguna subclase suya. Por ejemplo en la ontología definida por: $\{B(a),\ B \sqsubseteq A,\ a =_m A\}$ existe un ciclo, ya que el individuo a es meta-modelo de la clase A y pertenece a la misma.

Para evitar los ciclos, se implementa un algoritmo que chequea su existencia antes de declarar una ontología como consistente. Si la ontología no posee ciclos se continúa normalmente, de lo contrario se setea un *clash* que identifica el estado de inconsistencia por ciclos y se intenta realizar *backtrack* para continuar con el razonamiento desde un estado consistente anterior. Este funcionamiento es análogo a lo que ocurre cuando se detecta una inconsistencia en la ejecución de una regla.

```java
/**
 * Devuelve TRUE si existen ciclos de metamodeling entre los individuos.
 * FALSE si esta libre de ciclos.
 */
public boolean cyclesChecker(){
    IndividualIterator i = (PelletOptions.USE_COMPLETION_QUEUE)
            ? abox.getCompletionQueue()
            : abox.getIndIterator();
    boolean ciclo = false;
    while (!ciclo && i.hasNext()) {
        Individual ind = i.next();
        if(this.mbox.hasMetamodeling(ind.getTerm()))
            ciclo = check(ind, ind);
    }
    return ciclo;
}
private boolean check(Individual x, Individual y){
    if (this.mbox.hasMetamodeling(y.getTerm())){
        for (ATermAppl c : this.mbox.getClassesFromIndividual(x.getTerm())) {
            if (x.hasType(c) || hasEquivalentType(x, c)){
                return true;
            else
                for (Individual z : nodosConClase(c))
                    return check(x, z);
        }
    }
    return false;
}
```

Implementación del algoritmo de chequeo de ciclos.

Con esto, el razonador Pellet es capaz de chequear consistencia e inferir conocimiento en ontologías con axiomas de metamodelling. Es decir, es capaz de razonar en la nueva lógica *ALCQM.*
Las clases nuevas agregadas al razonador Pellet se implementaron dentro de package *uy.edu.fing.pellet* de forma que se mantuviera de forma separada a las clases de la versión original.



Clases nuevas del package *uy.edu.fing.pellet.*

Además de las modificaciones principales mencionadas en esta sección, se realizaron otras menos relevantes a distintas clases, la lista completa de modificaciones se puede consultar en el anexo II.

3.4. Plugin Metamodelling View

Durante el transcurso del proyecto se decidió implementar un nuevo plugin para gestionar las relaciones de metamodelling.
Si bien el plugin OntoRed provee una interfaz para la creación de este nuevo tipo de axioma, el mismo está diseñado para trabajar con redes de ontologías y no con ontologías individuales. Debido a que se trata de una relación no existente en el estándar OWL, Protégé no posee una forma de gestionar este tipo de axiomas, es por esto que resulta útil tener una pestaña donde los metamodelling de una ontología se puedan crear y eliminar al igual que ocurre con las clases o los individuos.

El nuevo plugin se implementa con una estructura igual y una arquitectura similar a la del plugin OntoRed. La interfaz implementada es muy sencilla y similar a la presentada por el otro plugin. En la misma se listan las clases e individuos que pueden relacionarse y la lista de todos los axiomas de metamodelling ya creados. También se reutilizan las vistas de Protégé de inviduos y jerarquía de clases para insertarlas en la vista del plugin. De esta forma el usuario puede crear y visualizar dentro del plugin los individuos y las clases.

3.5. Problemas encontrados

Durante el desarrollo de la solución se encontraron problemas debido principalmente a la poca o mala documentación y a la complejidad de algunos algoritmos.
En el caso de los plugins para Protégé, la documentación es muy escasa, incompleta y desactualizada. Los ejemplos provistos no eran suficientes ni demasiado claros.
La API de OWL por su parte, contaba con bastante documentación y ejemplos de uso importantes. Igualmente dentro del código existían métodos utilizados que no brindaban explicación de su funcionalidad.
Por otra parte, no resultó sencillo la extensión, debido al uso del patrón de diseño Visitor, fue necesario modificar muchas clases. Si bien resulta un buen diseño para un modelo que no varía tanto como es OWL, en nuestro caso dificultó la tarea.
Finalmente el razonador Pellet, no cuenta con una buena documentación y no está diseñado pensando en la extensibilidad. El hecho de ser un razonador muy eficiente y optimizado hace que algunos de sus algoritmos sean bastante complejos. En definitiva la poca documentación, sumada a la complejidad del código hizo complicado entender su funcionamiento para implementar la extensión.

4. Producto logrado

4.1. Instalación y uso

El producto final de este trabajo consiste de los siguientes archivos:

- dos plugins para Protégé (dos archivos formato jar)
- una versión extendida de OWLAPI 3.4.5 (un archivo formato jar)
- una versión del razonador Pellet 2.3.1 modificada (versión standalone con varios archivos y un ejecutable, y versión plugin para Protégé en un archivo formato jar).

Estos archivos se pueden descargar desde la siguiente pagina web www.cs.le.ac.uk/people/ps56/pelletM.

El desarrollo está realizado para la versión 4.3 de Protégé. Para trabajar con los nuevos plugins es necesario pegar los archivos jar de OWLAPI, los plugins OntoRed y metamodelling View, y el plugin de Pellet dentro del directorio *plugins* de Protégé. De esta forma se reemplazan la API de OWL existente y el plugin de Pellet si existe alguno, por los modificados.
Una vez hecho esto, iniciando Protégé se va a disponer de la versión modificada. Los nuevos plugins se pueden acceder desde la opción *windows-> tabs* cliqueando sobre los nombres de los plugins se agrega una nueva pestaña al programa. Para usar el razonador Pellet solo se debe seleccionar en el menú *reasoners* la opción *Pellet* y a continuación la opción *start* que inicia el razonamiento.

El plugin OntoRed esta especialmente desarrollado para trabajar con redes de ontologías. La manera de generar esta red es importando varias ontologías dentro de otra. El plugin carga relaciones de tipo extensión por cada ontología que se importe. La creación de relaciones funciona de la misma forma en cada pestaña, primero se seleccionan los objetos (clases o individuos) que son el dominio y el rango de la relación y luego se selecciona el botón *Crear*. Todas las relaciones creadas pueden eliminarse seleccionando las mismas en las tablas inferiores. Los objetos de la red se presentan con su nombre y con el nombre de la ontología a la que pertenecen como prefijo, de esta forma se puede identificar fácilmente.

Para crear nuevas relaciones del tipo metamodelling sobre una ontología alcanza con utilizar el plugin Metamodelling view de forma similar a como se crea un propiedad en Protégé. Se selecciona un individuo, una clase y se selecciona *Crear.* De ser necesario una relación creada puede ser eliminada siendo seleccionada en la lista inferior.

Las siguientes figuras muestran a forma de ejemplo, algunas impresiones de los plugins y del razonador Pellet sobre Protégé.

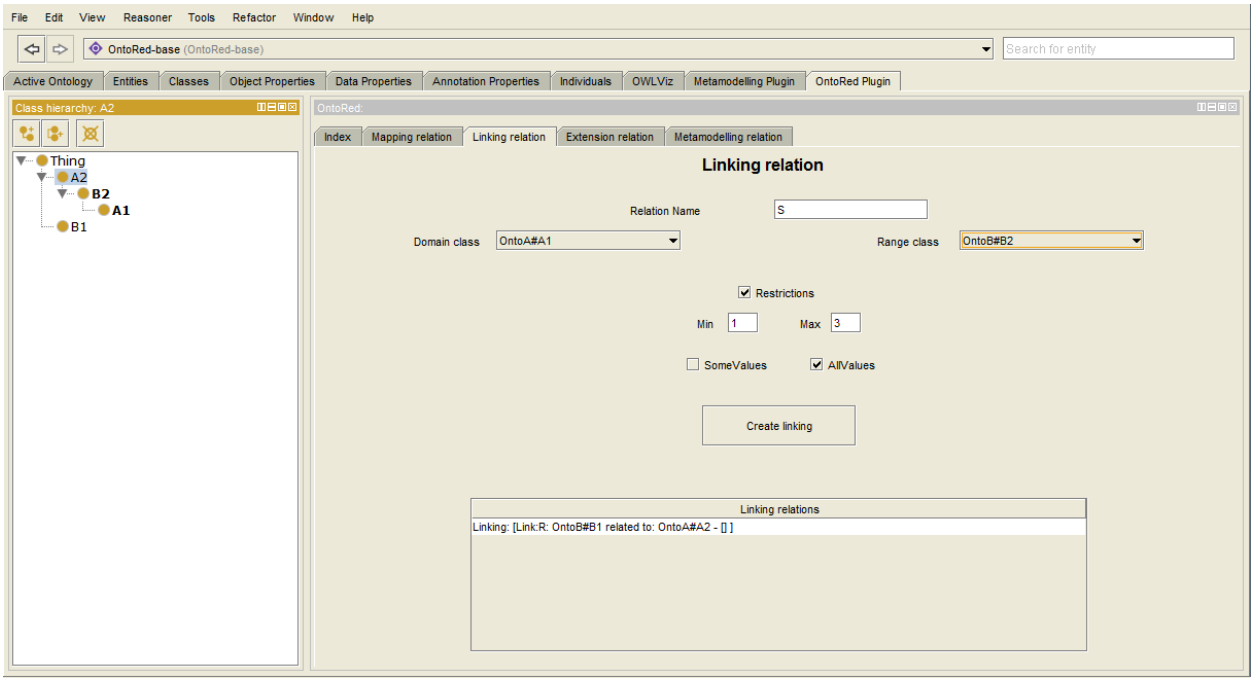


Figura 14. Vista del plugin OntoRed, tab Linking.

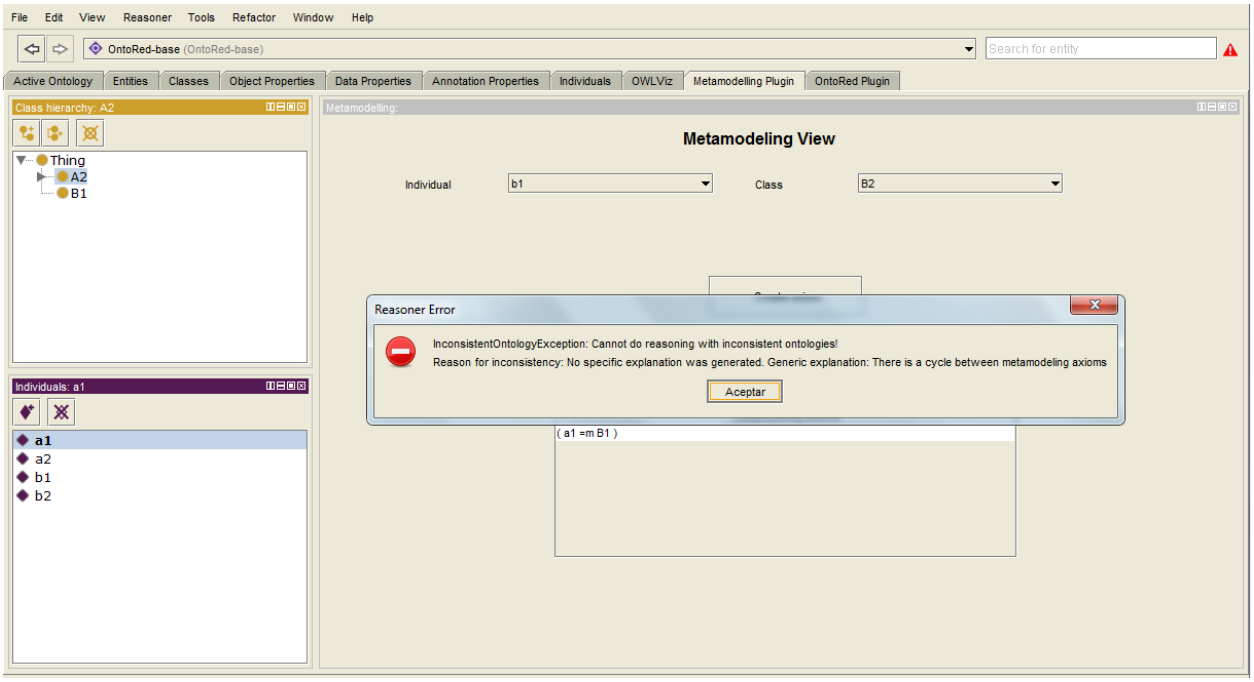


Figura 15. Ejecución de Pellet desde Protégé.

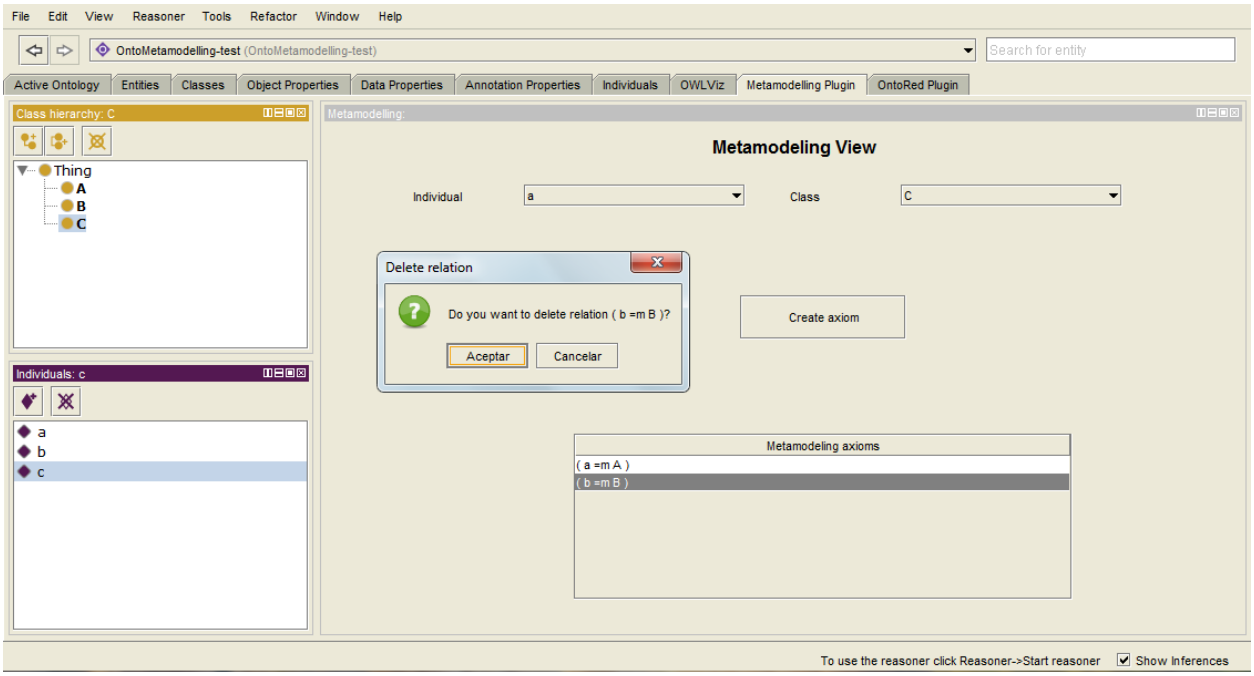


Figura 16. Vista del plugin Metamodelling View.

Si se desea trabajar con Pellet con la versión standalone, se debe descomprimir el archivo zip de esa versión en un directorio. Dentro de dicho directorio y mediante comandos de consola se ejecuta la opción deseada. Por ejemplo, si se desea chequear la consistencia de una ontología guardada en *ontologia1.owl* se debe ejecutar: "*pellet consistency ontologia.owl",* como se muestra en el ejemplo de la Figura 17.

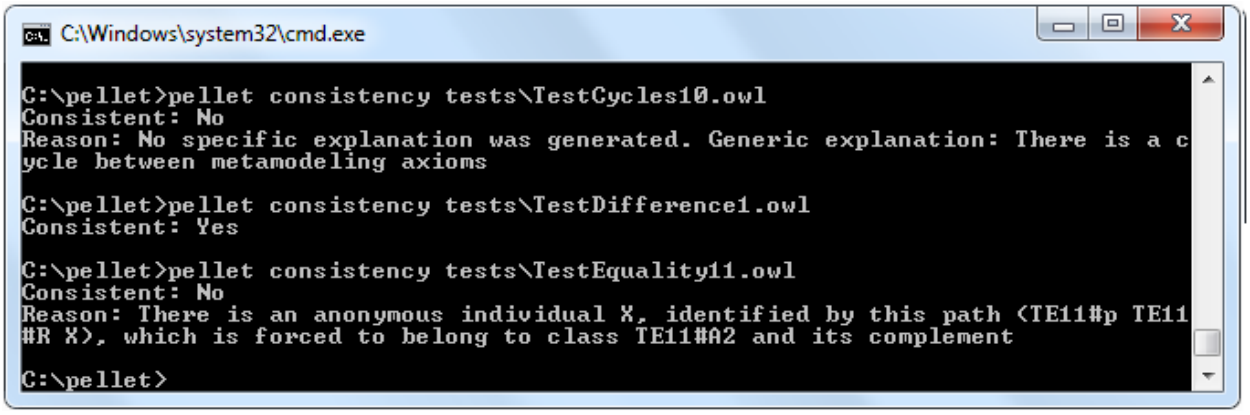


Figura 17. Ejecución de Pellet standalone para chequeo de consistencia.

Una explicación más extensa de todo el funcionamiento junto con un tutorial de uso de los plugins con Protégé se puede consultar en el anexo III.

4.2. Verificación de la solución

La verificación de la solución desarrollada se llevó a cabo en dos pasos. Primero se probaron los plugins junto con la OWLAPI, y luego se comprobó que el razonador Pellet funcionará correctamente con la nueva API y los axiomas de metamodelling.

Para probar los plugins generados y la API de OWL extendida, se crearon, cargaron y modificaron ontologías y redes de ontologías con y sin axiomas de metamodelling, tanto desde Protégé como a través de un editor de texto. De esta forma se comprobó que OWLAPI era estable y trabajaba de la manera deseada con el nuevo axioma sin interferir con los ya existentes. Se comprobó también que los plugins y el propio Protégé no presentaban problemas de compatibilidad con la nueva OWLAPI modificada. El modelado de las redes de ontologías diseñado para el plugin OntoRed funcionó sin ningún problema creando, eliminando, guardando y cargando las relaciones correctamente.

Con respecto al razonador Pellet se tuvo especial cuidado y se realizó una verificación bastante amplia y minuciosa.
Para ello, se definieron dos etapas de verificación. En primer lugar se definieron alrededor de 70 casos de prueba. Cada uno de los casos presenta una ontología diseñada para probar un aspecto específico del algoritmo de chequeo de Pellet. Los casos de prueba se dividen en 4 escenarios según las reglas o condición que apuntan a verificar. Además de comprobar que el resultado sea el esperado se verificó mediante logs que los pasos del razonamiento sean los correctos.
Luego de obtener resultados satisfactorios en la primer etapa, se verificó la solución utilizando un caso de estudio real de forma de comprobar si efectivamente se puede trabajar con dicho razonador.

En el anexo IV se detallan los casos de prueba de la etapa uno.

5. Conclusiones

5.1. Evaluación de la solución

El objetivo principal de este proyecto era implementar en la práctica el concepto de metamodelling, generando una herramienta que permita crear y editar redes de ontologías y razonar sobre ellas en lógica ALCQM.
El conjunto de herramientas desarrollado cumple con todos los objetivos del alcance del proyecto e incluso complementa lo requerido con la implementación del plugin Metamodelling View.
El producto final brinda las siguientes funcionalidades:

- creación y edición de ontologías y redes de ontologías con axiomas de metamodelling.
- exportación e importación de ontologías ALCQM en formato OWL/XML.
- extensión del lenguaje OWL así como de su API Java, con compatibilidad con metamodelling.
- modificación del razonador Pellet para dar soporte al razonamiento sobre ontologías con metamodelling.

Se trata de un producto bastante amplio que consta de varias partes implementadas a distintos niveles, lo que resultó costoso y complejo de lograr. Aunque superó gran cantidad de tests con solvencia y es una solución en general estable, quedan varias mejoras y ajustes que hacer, principalmente en cuanto a usabilidad de las interfaces.
En el desarrollo de cada una de las partes se siguieron las buenas prácticas de desarrollo. Se documentando los métodos importantes, se estructurando las nuevas clases en package separados y se diseñando pensando en la extensibilidad y el mantenimiento futuro. Las partes del software fueron probadas individualmente y de forma conjunta en test de integración.
Se considera que el producto logrado brinda una buena base para continuar con el desarrollo hacia una versión final.

En el aspecto académico, el proyecto requirió profundizar el aprendizaje acerca de ontologías, tanto en lo teórico como en la práctica. Investigando, evaluando y aprendiendo acerca de los distintos agentes de software que trabajan sobre éstas (razonadores, editores, etc.). El desarrollo y extensión de las distintas herramientas permitió experimentar con APIs e integración de sistemas. Finalmente, Pellet en particular, brindó la posibilidad de trabajar y aprender de algoritmos complejos y optimizados que apuntan a mejorar la eficiencia.

5.2. Limitaciones y trabajo a futuro

De acuerdo al alcance del proyecto, se definió trabajar con ontologías en OWL/XML y extender las herramientas solamente para este formato. Algunos formatos de ontologías a los cuales sería interesante extender la herramienta son: RDF/XML, Manchester Syntax, Latex, Turtle, OBO, entre otros. Dicha extensión consiste en implementar los métodos de las interfaces de parseo de cada lenguaje en la API OWL. Se trata de un trabajo relativamente sencillo, ya que los nuevos métodos funcionan de forma análoga a los ya implementados y el desarrollo más importante para el soporte del metamodelling ya está implementado en el *core* de la API y es reutilizable.

El proyecto deja tres líneas principales a seguir trabajando en el futuro:

- **Verificación:** si bien se realizó una verificación amplia, es necesario continuar probando la herramienta, sobre todo el razonador y hacer pruebas con usuarios finales que aporten otra visión del producto.

- **Mejoras de interfaz:** la interfaz de los plugins se diseñó sencilla y apuntando a la funcionalidad específica. Existen varias mejoras para realizar en este aspecto por ejemplo, reutilizar más componentes de Protégé para crear una interfaz similar a la original, brindar otras funcionalidades relacionadas en la misma pantalla, hacerla más atractiva y fácil de utilizar.
- **Optimización:** especialmente en Pellet la eficiencia del código resulta clave, ya que los chequeos que realiza generan árboles lógicos potencialmente muy grandes. Debido a que dentro de los requerimientos la optimización no era un objetivo principal, ésta es una área en la que se puede profundizar para mejorar la eficiencia.

Sumado a lo anterior, la herramienta podría extenderse a más formatos y plataformas. La versión está generada y probada para Protégé y para Pellet en sus dos versiones, pero sería bueno continuar trabajando con la API OWL extendida para que otros programas implementen metamodelling. Sería muy útil para los usuarios y la comunidad extender otros razonadores como HermiT o FaCT++ que poseen características distintas a Pellet.

6. Bibliografía

1. Motz, Regina, Edelweis Rohrer, and Paula Severi. **"Reasoning for ALCQ extended with a flexible meta-modelling hierarchy."** Semantic Technology - 4th Joint International Conference, JIST 2014, Chiang Mai, Thailand, November 9-11, 2014. Revised Selected Papers. Lecture Notes in Computer Science 8943, Springer 2015, ISBN 978-3-319-15614-9, 2014.

2. Diaz, Alicia, Regina Motz, and Edelweis Rohrer. **"Making ontology relationships explicit in a ontology network."** Proceedings of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management, Santiago, Chile, May 9-12, 2011. CEUR Workshop Proceedings 749, CEUR-WS.org 2011

3. Hitzler, Pascal, Markus Krotzsch, and Sebastian Rudolph. **Foundations of semantic web technologies.** CRC Press, 2011..

4. **Protege4DevDocs - Protege Wiki.** 2010. Accesible en: <http://protegewiki.stanford.edu/wiki/Protege4DevDocs> Ultimo acceso: 26 Nov. 2014

5. **OSGi Alliance | Technology.** 2012 Accesible en: <http://www.osgi.org/Technology/HomePage> Ultimo acceso: 30 Ene. 2015.

6. Matthew Horridge, Sean Bechhofer **The OWL API: A Java API for OWL Ontologies** Semantic Web Journal 2(1), Special Issue on Semantic Web Tools and Systems, pp. 11-21, 2011.

7. Sirin, Evren et al. **"Pellet: A practical owl-dl reasoner."** Web Semantics: science, services and agents on the World Wide Web 5.2 (2007): 51-53.

8. Baader, Franz. **The description logic handbook: theory, implementation, and applications**. Franz Baader. Cambridge university press, 2003.

9. Glimm, Birte, Ian Horrocks, and Boris Motik. **"Optimized description logic reasoning via core blocking."** Automated Reasoning (2010)

10. Motik, Boris, Rob Shearer, and Ian Horrocks. **"Hypertableau reasoning for description logics."** Journal of Artificial Intelligence Research 36.1 (2009): 165-228.

11. Horrocks, Ian R. **"Optimising tableaux decision procedures for description logics."** Tesis de doctorado, Universidad de Manchester, 1997.

12. **Vista General del Lenguaje de Ontologías Web (OWL)**. 2007. Accesible en: <http://www.w3.org/2007/09/OWL-Overview-es.html> Ultimo acceso: 26 Abr. 2014

13. **Protege 3.x vs 4.x - UNIK wiki**. 2014. Accesible en: <http://cwi.unik.no/images/0/0b/Protege3_vs_4.pdf> Ultimo acceso: 26 Abr. 2014

14. **Comment créer un plugin sous protégé 4.1.** 2012. Accesible en: <http://maximefavier.fr/comment-creer-un-plugin-pour-protege-4-1.php> Ultimo acceso: 5 Mayo 2014

15. Glimm, Birte et al. **"HermiT: an OWL 2 reasoner."** Journal of Automated Reasoning 53.3 (2014): 245-269.

16. **Pellet: OWL 2 Reasoner for Java - Clark & Parsia.** 2008. Accesible en: <http://clarkparsia.com/pellet/> Ultimo acceso: 5 Abr. 2015.

17. **OWL API - SourceForge.** 2004. Accesible en: <http://owlapi.sourceforge.net/> Ultimo acceso: 5 Abr. 2015.

18. **Protégé.** 2002. Accesible en: <http://protege.stanford.edu/> Ultimo acceso: 5 Abr. 2015

19. **Ontology by Tom Gruber**. Encyclopedia of Database Systems, Ling Liu and M. Tamer Özsu (Eds.), Springer-Verlag, 2009. Accesible en: <http://tomgruber.org/writing/ontology-definition-2007.htm> Ultimo acceso: 11 Abr. 2015

7. Anexos

Anexo I: Modificaciones a OWLAPI

A continuación se listan todas las clases modificadas en la extensión de la API de OWL.

Package OWLAPI-API:	**Package OWLAPI-IMPL:**
OWLMetamodelingAxiom.java*	OWLMetamodelingAxiomImpl.java*
AxiomType.java	SatisfiabilityReducer.java
OWLAxiomVisitor.java	AbstractEntityRegistrationManager.java
OWLAxiomVisitorEx.java	AbstractInternalsImpl.java
OWLDataFactory.java	InitVisitorFactory.java
AxiomSubjectProvider.java	Internals.java
AxiomTypeProvider.java	InternalsImpl.java
DelegatingObjectVisitorEx.java	OWLDataFactoryImpl.java
DLExpressivityChecker.java	OWLEntityCollectionContainerCollector.java
HashCode.java	
HornAxiomVisitorEx.java	**Package OWLAPI-PARSERS:**
MaximumModalDepthFinder.java	OWLMetamodelingAxiomElementHandler.java*
NNF.java	KRSS2OWLObjectRenderer.java
OWLAxiomVisitorAdapter.java	KRSS2Vocabulary.java
OWLClassExpressionCollector.java	KRSSObjectRenderer.java
OWLEntityCollector.java	OWLObjectRenderer.java
OWLObjectComponentCollector.java	LatexObjectVisitor.java
OWLObjectDuplicator.java	ManchesterOWLSyntax.java
OWLObjectTypeIndexProvider.java	OWLXMLObjectRenderer.java
OWLObjectVisitorAdapter.java	OWLXMLParserHandler.java
OWLObjectVisitorExAdapter.java	AbstractTranslator.java
OWLObjectWalker.java	ManchesterOWLSyntaxObjectRenderer.java
SimpleRenderer.java	DLSyntax.java
StructuralTransformation.java	DLSyntaxObjectRenderer.java
OWLRDFVocabulary.java	
OWLXMLVocabulary.java	**Package OWLAPI-TOOLS:**
	SatisfiabilityConverter.java
Package OWLAPI-CONTRACT	SyntacticLocalityEvaluator.java
OWLTutorialSyntaxObjectRenderer.java	DebuggerClassExpressionGenerator.java
ContractOwlapiUtilTest.java	ExplanationOrdererImpl.java
ContractRdfModelTest.java	
Utils.java	
OWLTutorialSyntaxObjectRenderer.java	

(*): clases nuevas.

Anexo II: Modificaciones a Pellet

A continuación se listan todas las clases modificadas en la extensión del razonador Pellet.

Módulo *OWLAPI3V:*	**Módulo *CORE:***
EntailmentChecker.java	CloseBranch.java*
PelletVisitor.java	CloseRule.java*
	EqualRule.java*
Módulo *EL*:	NotEqualRule.java*
ELExpressivityChecker.java	MBox.java*
	CyclesChecker.java*
Módulo *EXPLANATION*:	KnowledgeBase.java
ManchesterSyntaxObjectRenderer.java	ATermUtils.java
	Branch.java
	NodeSelector.java
	CompletionStrategy.java
	Expressivity.java
	DLExpressivityChecker.java
	Clash.java
	SROIQStrategy.java

(*): clases nuevas.

Anexo III: Manual de uso

Tutorial plugin OntoRed y razonador Pellet

Requisitos

- Protégé 4.3
- Java 7 (JRE) o superior
- ontologías en formato OWL/XML
- plugin OntoRed
- OWLAPI versión modificada
- plugin Pellet versión modificada

Instalación

Los archivos necesarios para instalar la herramienta se pueden obtener en el siguiente enlace www.cs.le.ac.uk/people/ps56/pelletM.

OntoRed y OWLAPI

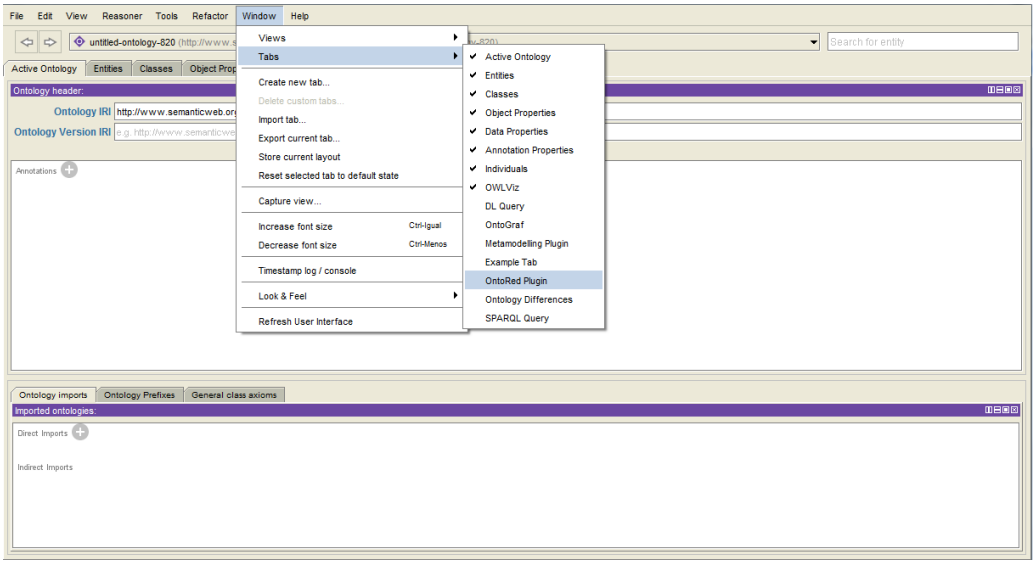
El plugin OntoRed está desarrollado para la versión 4.3 de Protégé y la versión 7 o superior de Java RE. Para instalarlo es necesario copiar los archivos jar OntoRed.jar y org.semanticweb.owl.owlapi.jar dentro de la carpeta “plugins” que se encuentra en la carpeta de instalación de Protégé. Será necesario reemplazar el jar org.semanticweb.owl.owlapi.jar que viene por defecto con Protégé por la nueva versión modificada.

Pellet plugin

Para utilizar el razonador Pellet en Protégé es necesario instalar la versión modificada del plugin. Al igual que OntoRed, se debe copiar el archivo jar **com.clarkparsia.protege.plugin.pellet.jar** en la carpeta “*plugins”* de Protégé.

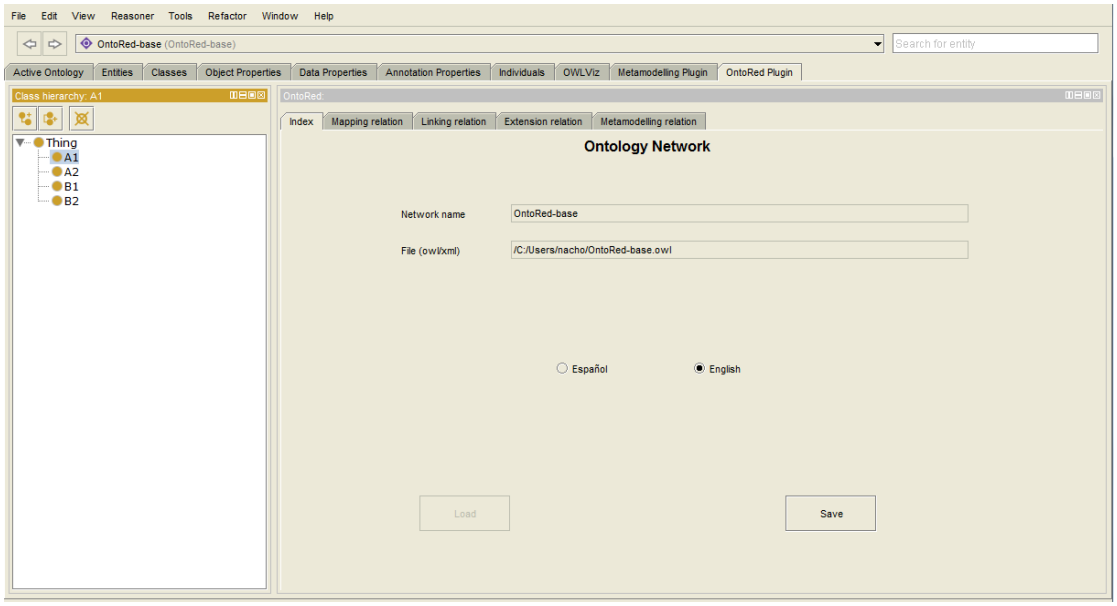
Inicio

Para iniciar el plugin en Protégé hay que seleccionar en **Window** -> **Tabs** la opción **OntoRed** Plugin. Con eso se abrirá una nueva Tab correspondiente al plugin.
La red de ontologías, se crea como cualquier ontología normal de **Protégé**, utilizando el formato **OWL/XML** para grabar la misma.



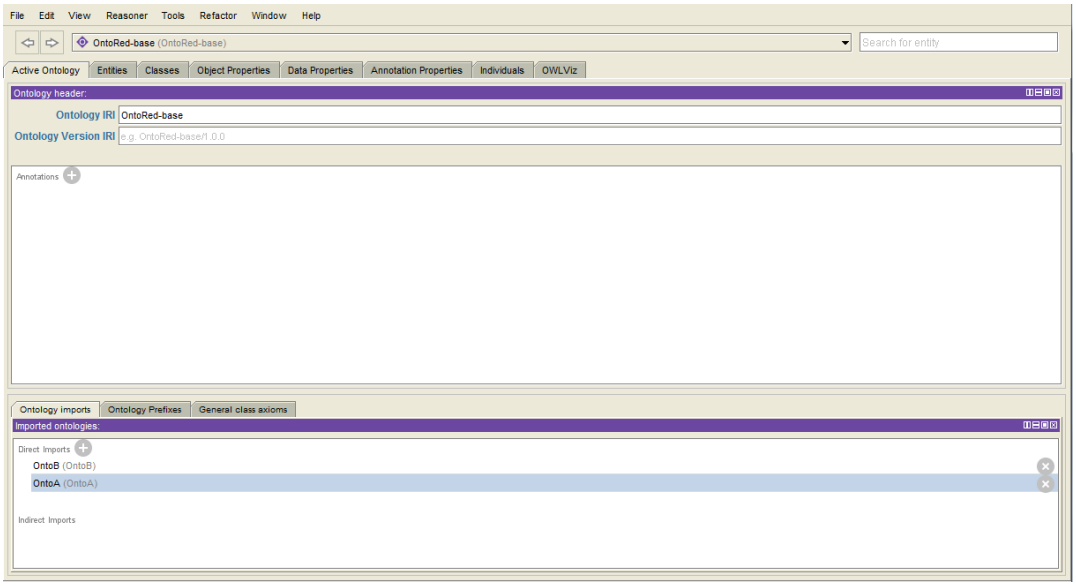
Plugin

El plugin consiste de 5 tabs. En la primera, la tab inicio, se muestra la información de la ontología (red) cargada, su nombre y el archivo donde esta guardada. Desde aquí se puede cargar una red y guardar la misma. También es posible seleccionar el idioma entre Español e Inglés.
Las otras 4 tabs son para el manejo de las relaciones.



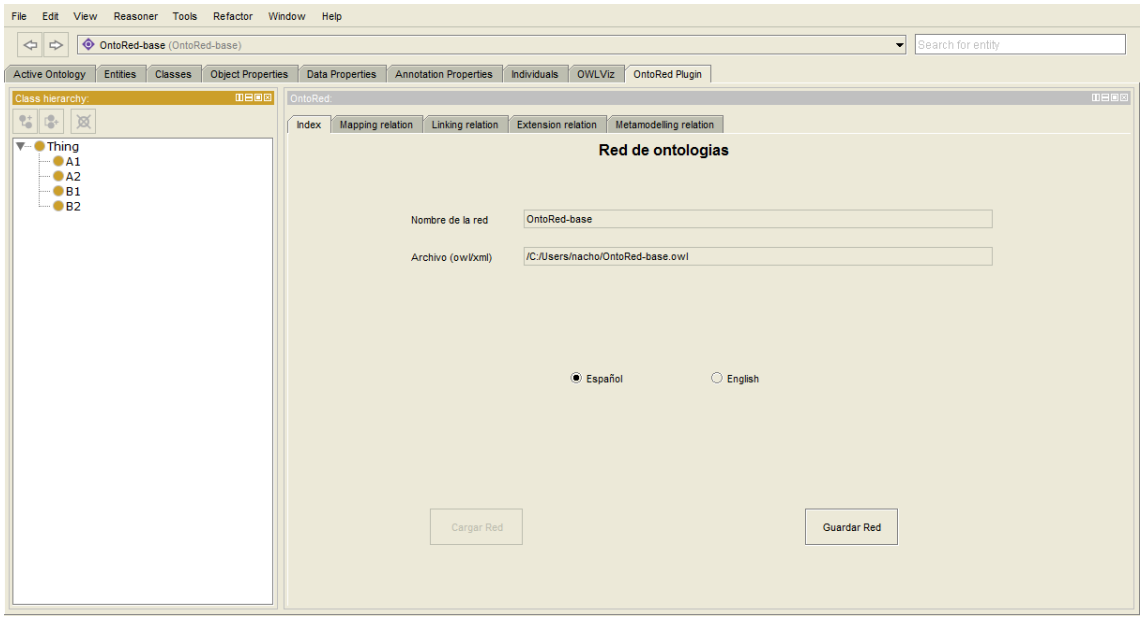
Imports

Una **red de ontologías** es una ontología que importa otras ontologías. Para importar las mismas, se utiliza el ***import*** que provee Protégé. En la tab inicial, en la parte inferior donde dice ***import*** se seleccionan las ontologías que se desea importar. Desde aquí también, se pueden quitar las ontologías que no queramos que pertenezcan a la red.



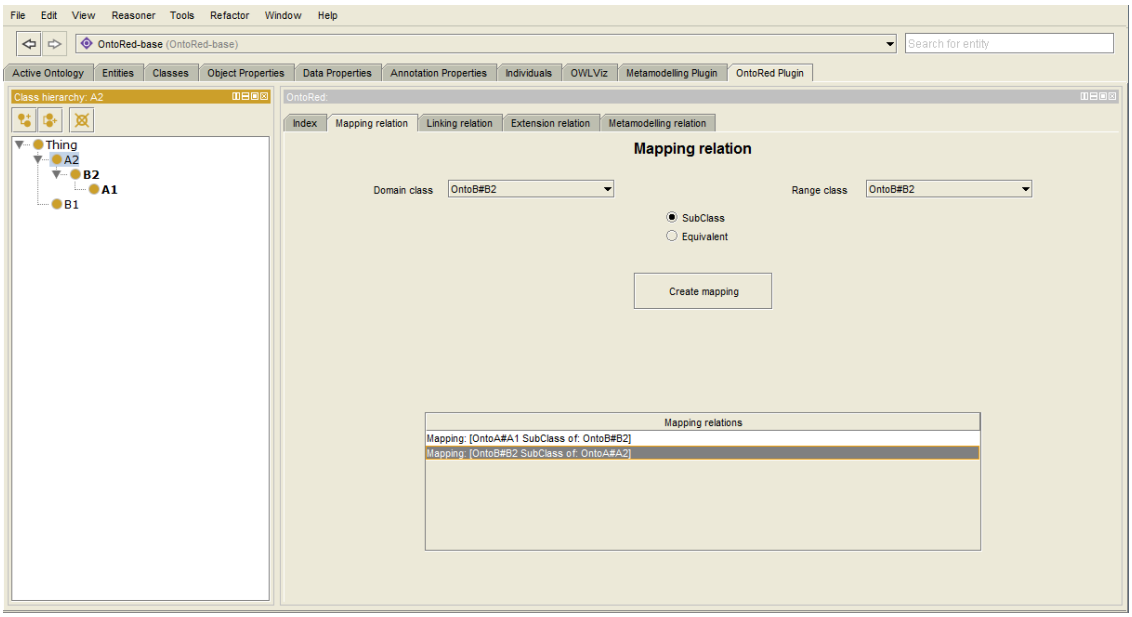
crear/cargar red

Una vez que tenemos creada o cargada en Protégé la ontología que representa a la red (formato OWL/XML), vamos al tab del plugin (OntoRed) y damos click en **Cargar Red**. Con esto el plugin cargará las relaciones que existan en sus tabs correspondientes.



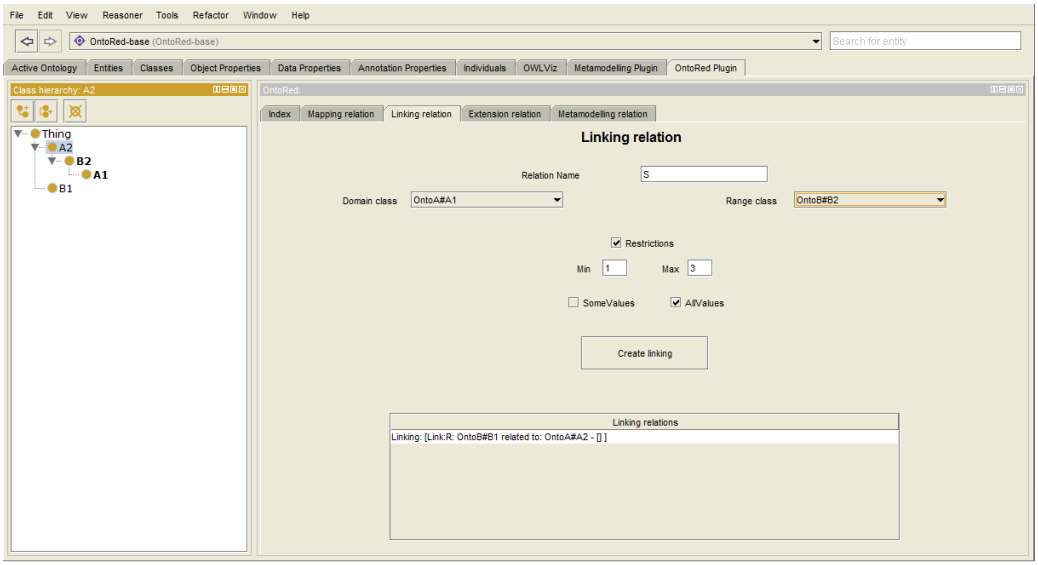
tab mapping

El tab mapping permite crear relaciones entre clases de distintas ontologías del tipo **subclass** o **equivalent**.
Para esto se selecciona una clase origen y una destino, el tipo de mapping, y se da click en *Crear Mapping*.
Todas las relaciones creadas de este tipo se pueden visualizar en la lista de más abajo. Para eliminar una relación creada, se da click sobre la misma.



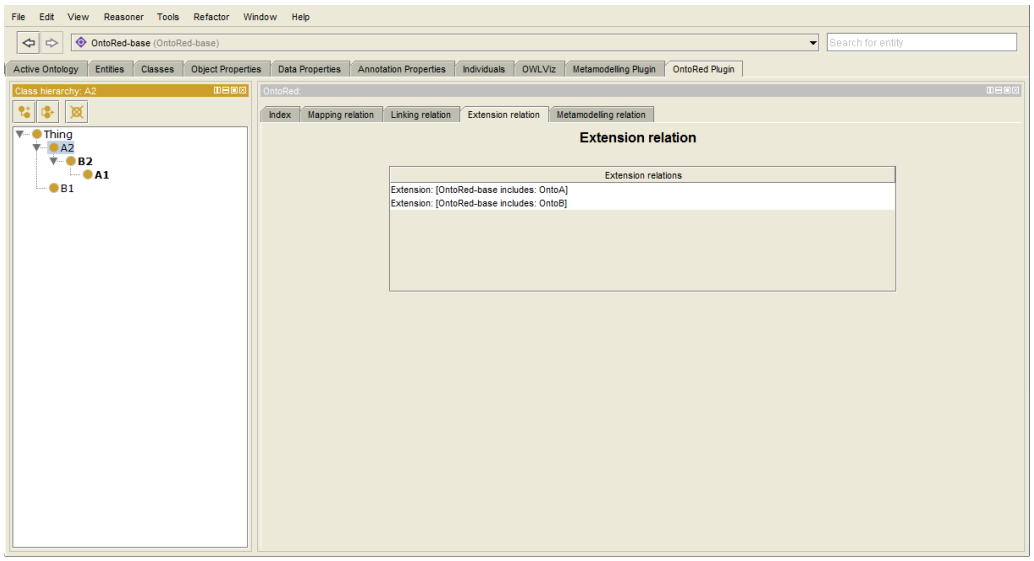
tab linking

El tab linking permite crear relaciones entre clases de diferentes ontologías del tipo ObjectProperty, con restricciones tanto de **cardinalidad** como **someValues** y **allValues** si se desea.
Para ello, se seleccionan las clases de la relación y se marcan las restricciones deseadas.
En la lista de la parte inferior también se pueden visualizar y eliminar las relaciones de tipo linking.



tab extension

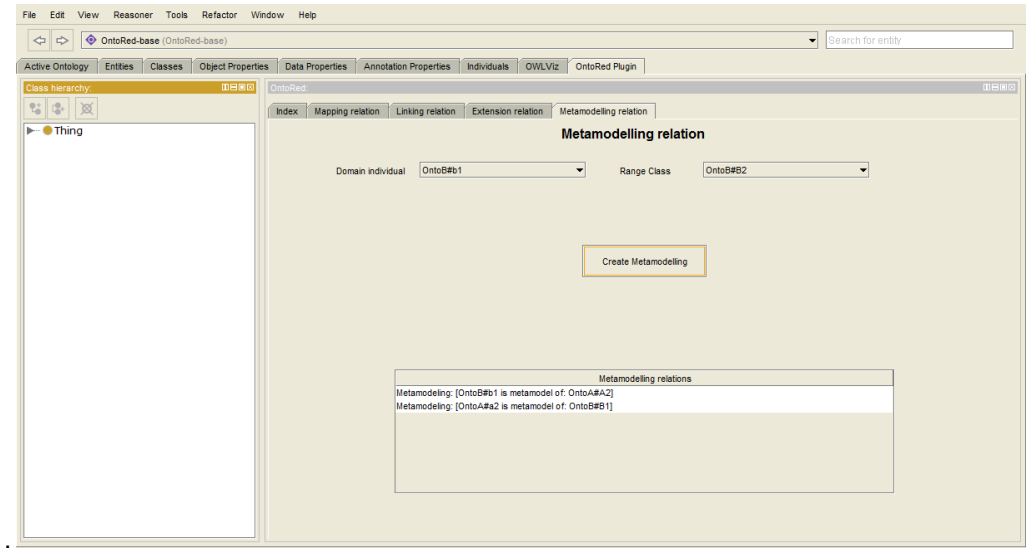
Las relaciones del tipo extensión corresponden a las importaciones de ontologías en la red. En esta tab se visualizan las relaciones, pero el manejo (creación y eliminación) se realiza como se vió antes, mediante la funcionalidad de Protégé.



tab metamodelling

Finalmente está el tab metamodelling. Este nos permite crear relaciones entre instancias y clases de ontologías diferentes. Representa una relación que no existe como tal en el lenguaje OWL, una equivalencia entre una instancia y una clase.

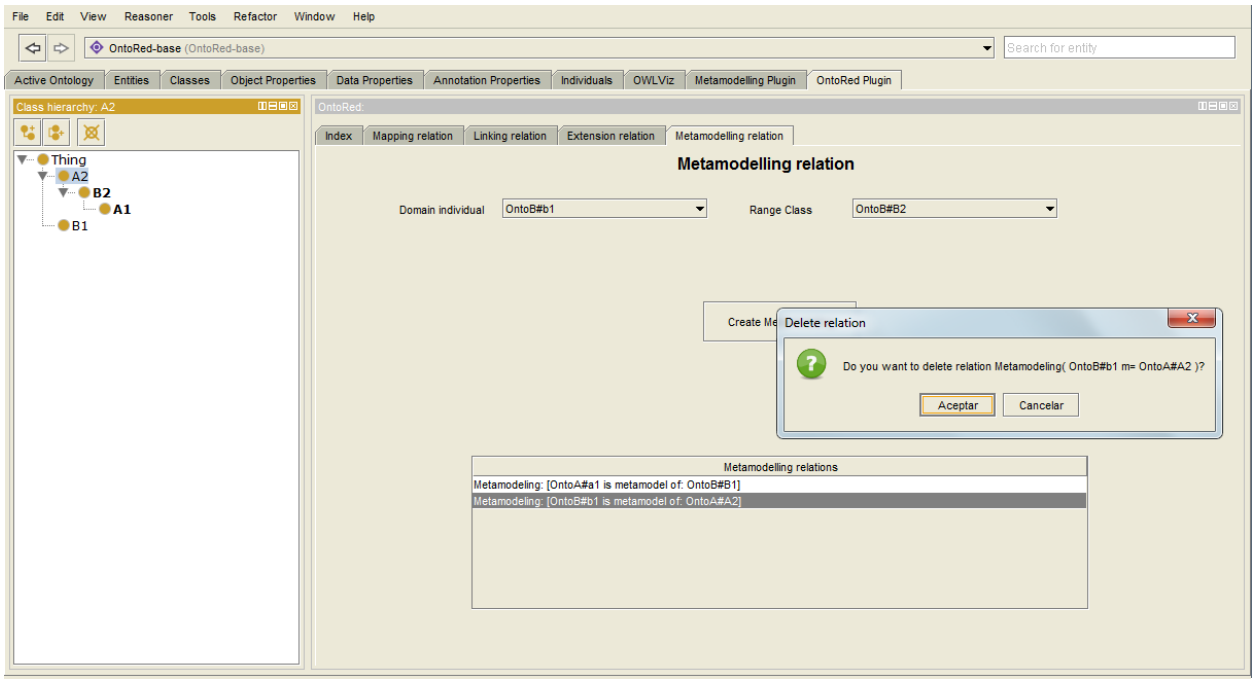
Funciona igual que mapping y linking, se selecciona origen y destino de la relación y se da click en **Crear**. También se pueden visualizar y eliminar las relaciones creadas en la lista de la parte inferior



.

eliminar relación

Como se mencionó, para eliminar una relación, se da click sobre la misma.

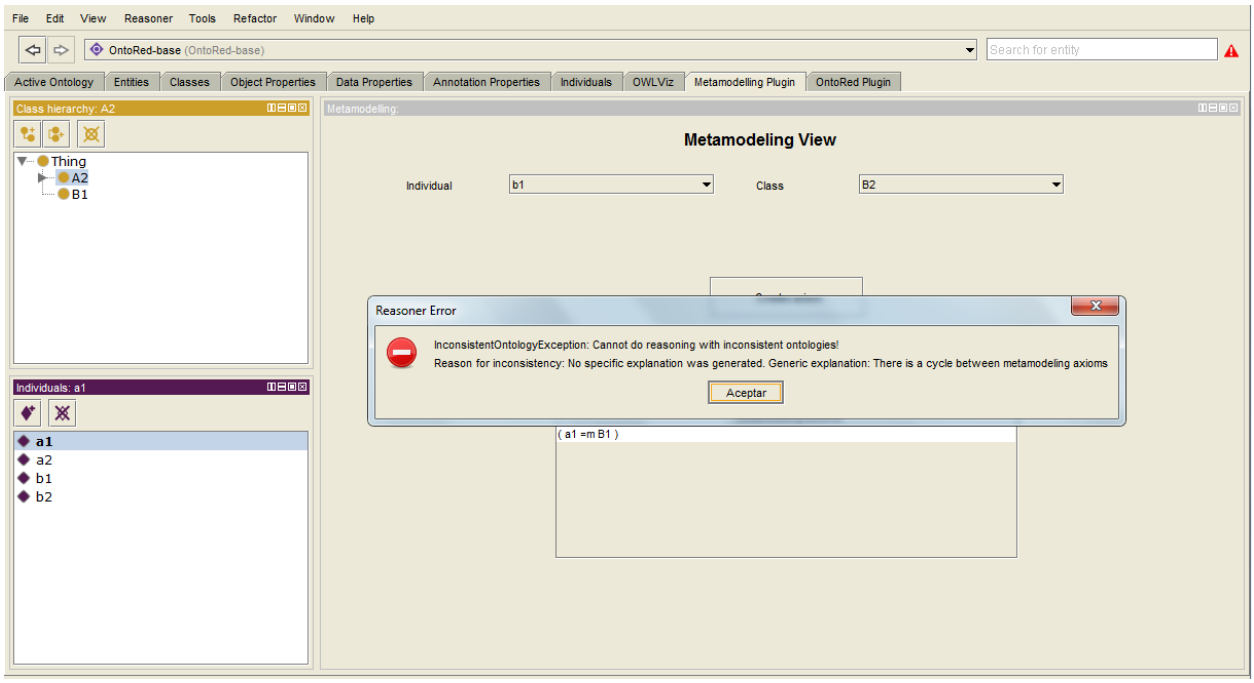


razonador Pellet

Para utilizar el razonador Pellet se debe configurar e iniciar desde el menú *Reasoner.*
Primero se selecciona el razonador haciendo *Reasoner-> Pellet,* y luego se inicia el razonamiento haciendo *Reasoner->Start reasoner.*

Pellet realiza el chequeo de consistencia de la red cargada e infiere nuevo conocimiento sobre esta. Si se modifica la red, es necesario sincronizar el razonador para hacer un nuevo chequeo, esto se logra seleccionado *Reasoner->Synchronize reasoner.*

Si existe una inconsistencia en la red, se muestra un mensaje de error que indica esto.

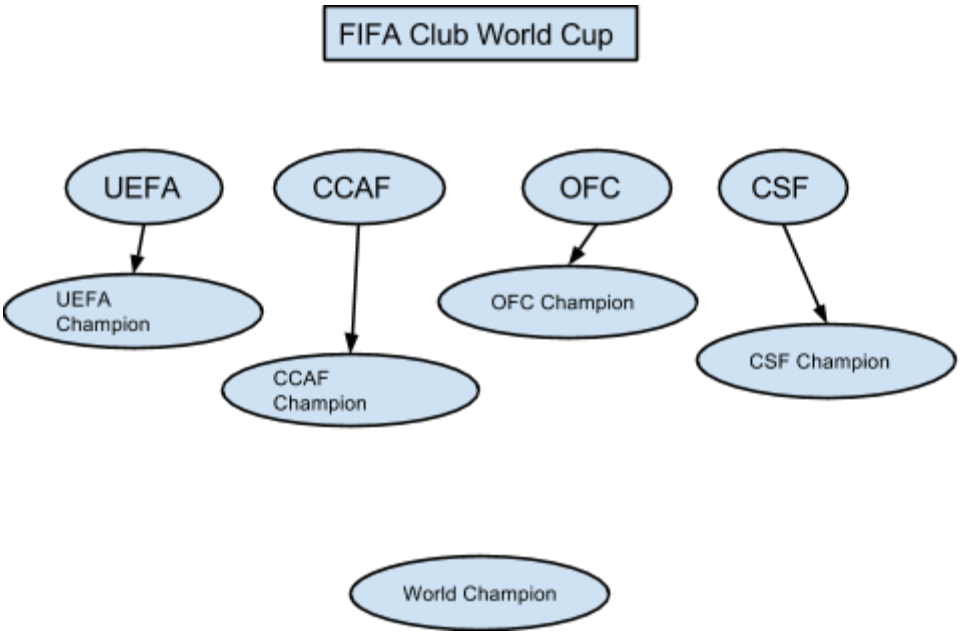


Mensaje de error por inconsistencia en la red.

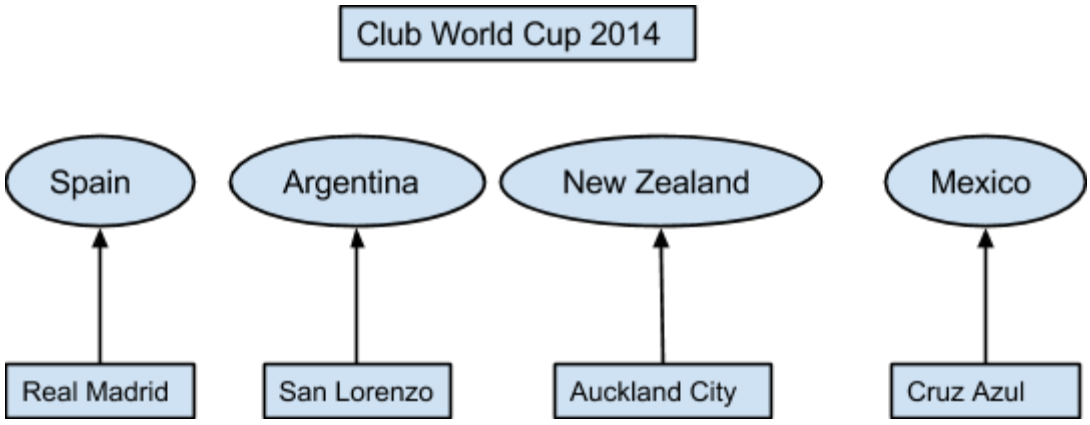
Red de ejemplo

Para mostrar el funcionamiento básico de OntoRed plugin, utilizaremos una red que modela el Mundial de clubes FIFA.
La red estará formada por una ontología que modela el formato del torneo, y otra ontología que representa la edición del año 2014.

En primer lugar, tenemos la ontologia **FIFA_WC** que modela el torneo de forma simplificada. En el mismo compiten los campeones de las confederaciones FIFA de cada continente (UEFA, CONCACAF, CSF y OFC). El resultado del torneo es un ***Campeón Mundial***. Todos los conceptos son modelados como clases.



Por otro lado, tenemos al ontología con los representantes de la edición 2014 del torneo. En este caso modelamos los países de los equipos con clases, y a los equipos como individuos.



Como primer paso, creamos las ontologías **FIFA_WC** y **WC_2014** normalmente usando Protégé (o usamos las adjuntas en los archivos correspondientes). Luego, iniciamos Protégé y en una nueva ontología vamos a importar las dos anteriores, de esta forma creamos la red.

Ahora, vamos al plugin, y damos click en “cargar red”, nos cargará la red vacía, solo con los import realizados.

Vamos a crear las siguientes relaciones:

En la **pestaña de Mapping,** **modelamos que los países de los equipos pertenecen a una Confederación:**

- **<Spain subClassOf UEFA>**
- **<Mexico subClassOf CCAF>**
- **<Argentina subClassOf CSF>**
- **<New_Zealand subClassOf OFC>**

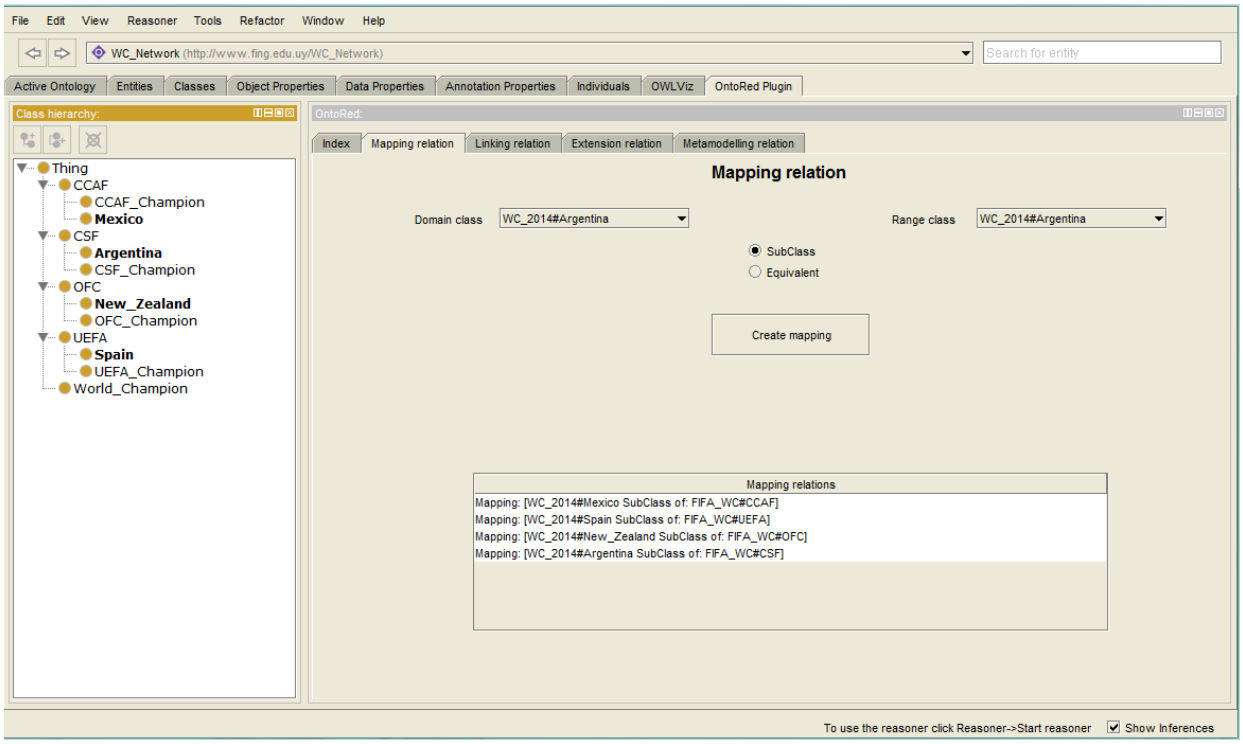
En la pestaña de Linking, modelamos que el campeón mundial es español:

- **<World_Champion isFrom Spain>**

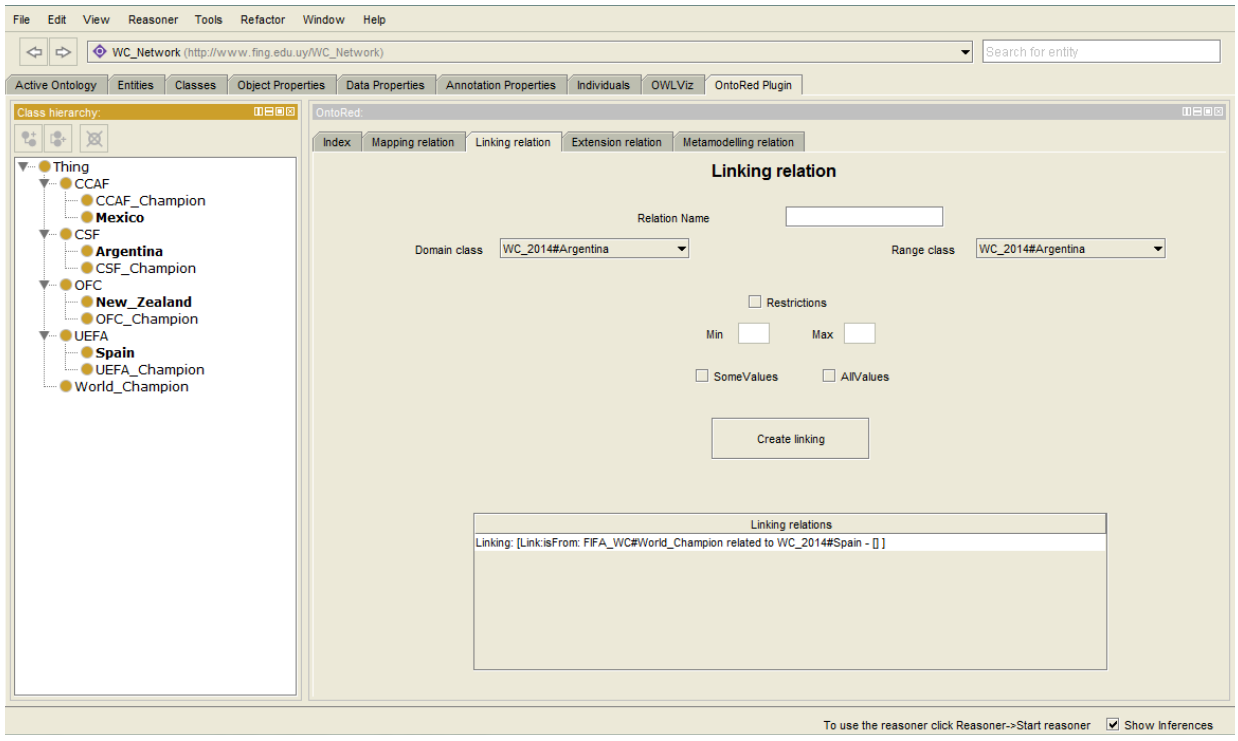
Finalmente, en la pestaña de Metamodelling, vamos a crear la equivalencia entre los clubs que son individuos en WC_2014, con las clases que los representan en la ontologia **FIFA_WC:**

- <Real_Madrid $=_m$ UEFA_Champion>
- <Auckland_City $=_m$ OFC_Champion>
- <San_Lorenzo $=_m$ CSF_Champion>
- <Cruz_Azul $=_m$ CCAF_Champion>
- <Real_Madrid $=_m$ World_Champion>

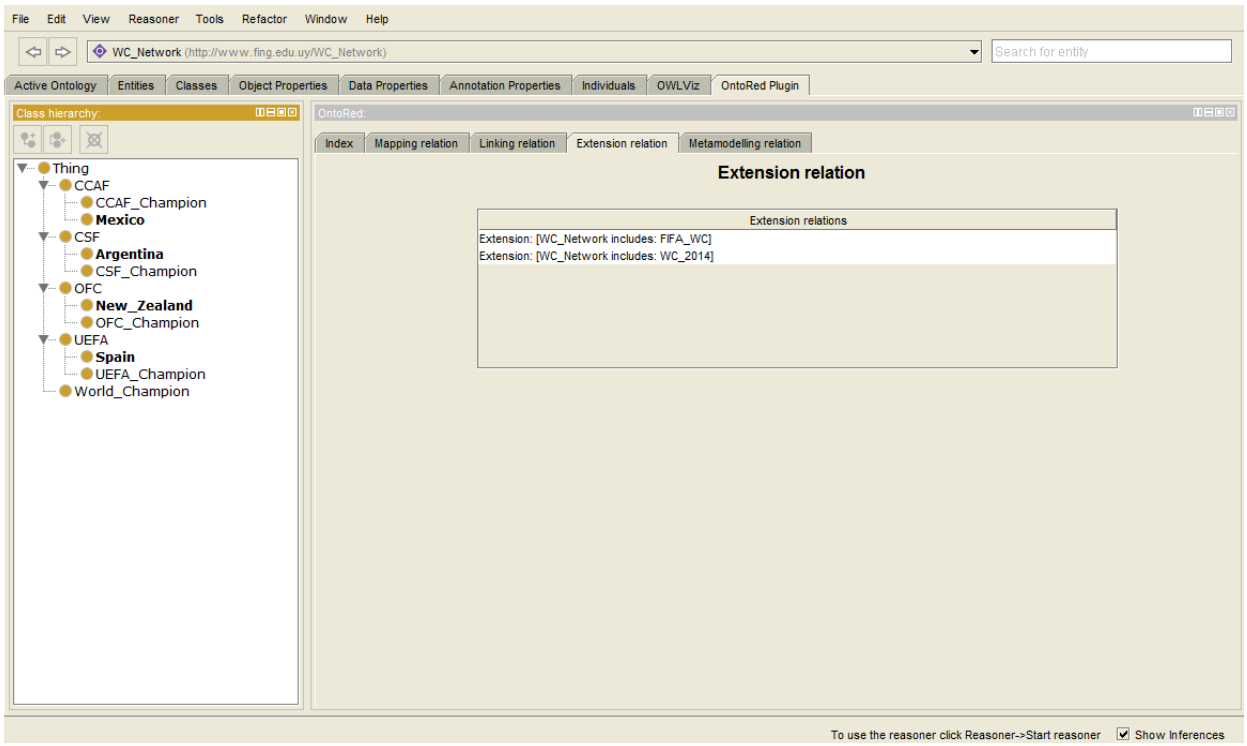
Relaciones Mapping



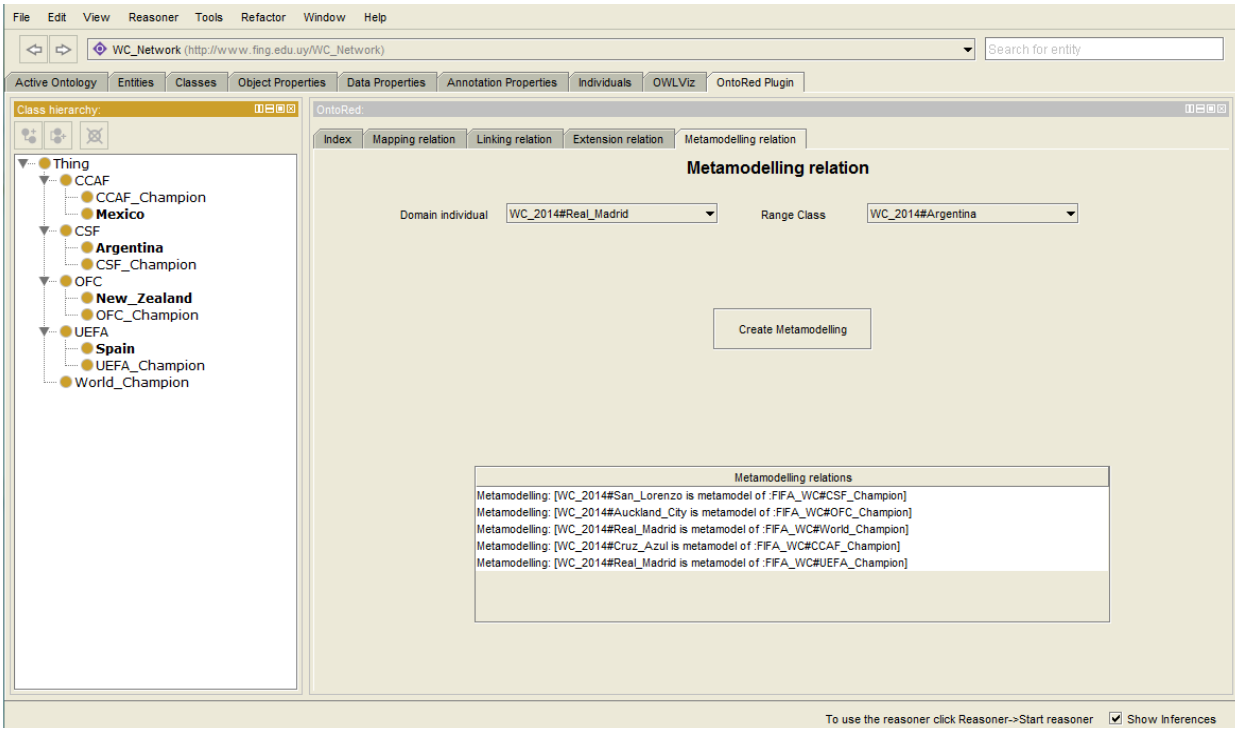
Relaciones Linking



Relaciones Extension



Relaciones Metamodelling



Anexo IV: Casos de prueba

Casos de prueba - Equality:

Nº	Ontología	Res.Esperado	Res.Obtenido
1	$a =_m A1$, $a =_m A2$, $A1(p)$, $\neg A2(p)$.	Inconsistente	Correcto
2	$a1 =_m A1$,$a2 =_m A2$, $A1(p)$,$\neg A2(p)$.	Consistente	Correcto
3	$a1 =_m A1$, $a2 =_m A2$, $A1(p)$, $\neg A2(p)$, $a1 = a2$.	Inconsistente	Correcto
4	$a1 =_m A1$, $a2 =_m A2$, $A1(p)$, $\neg A2(p)$, $P(q, a1)$, $P(q, a2)$.	Consistente	Correcto
5	$a1 =_m A1$, $a2 =_m A2$, $A1(p)$, $\neg A2(p)$, $P(q, a1)$, $P(q, a2)$, P funcional	Inconsistente	Correcto
6	$a1 =_m A1$, $a2 =_m A2$, $A1(p)$, $A1 \sqcap A2 \sqsubseteq \bot$, $P(q, a1)$, $P(q, a2)$.	Consistente	Correcto
7	$a1 =_m A1$,$a2 =_m A2$, $A1(p)$, $A1 \sqcap A2 \sqsubseteq \bot$, $P(q, a1)$, $P(q, a2)$, P funcional	Inconsistente	Correcto
8	$a1 =_m A1$,$a2 =_m A2$, $a3 =_m A3$, $a4 =_m A4$, $A3(p)$, $A3 \sqcap A4 \equiv \bot$, $P(a1, a3)$, $P(a2, a4)$, P funcional	Consistente	Correcto
9	$a1 =_m A1$,$a2 =_m A2$, $a3 =_m A3$, $a4 =_m A4$, $A1 \equiv B \sqcup C$, $A2 \equiv B \sqcup C$, $A3(p)$, $A3 \sqcap A4 \equiv \bot$, $P(a1, a3)$, $P(a2, a4)$, P funcional	Inconsistente	Correcto
10	$A1 \equiv \exists R.A2$, $A1 \equiv \forall R.\neg A1$, $a1 =_m A1$,$a2 =_m A2$, $P(p, a1)$, $P(p, a2)$.	Consistente	Correcto
11	$A1(q)$, $A1 \equiv \exists R.A2$, $A1 \equiv \forall R.\neg A1$, $a1 =_m A1$, $a2 =_m A2$, $P(p, a1)$, $P(p, a2)$, P funcional	Inconsistente	Correcto
12	$A1(q)$, $A1 \equiv \exists R.A2$, $A2 \equiv \forall R.\neg A1$, $a1 =_m A1$,$a2 =_m A2$, $P(p, a1)$, $P(p, a2)$.	Consistente	Correcto
13	$A1(q)$, $A1 \equiv \exists R.A2$, $A2 \equiv \forall R.\neg A1$, $a1 =_m A1$,$a2 =_m A2$, $P(p, a1)$, $P(p, a2)$, P funcional	Inconsistente	Correcto
14	$a1 =_m A1$,$a2 =_m A2$, $A1(p)$, $(\neg A2 \sqcup A3)(p)$, $a1 = a2$.	Consistente	Correcto
15	$a1 =_m A1$, $a2 =_m A2$, $A1(p)$,$(A3 \sqcup \neg A2)(p)$, $a1 = a2$.	Consistente	Correcto
16	$a1 =_m A1$,$a2 =_m A2$, $a3 =_m A3$, $R(p, a1)$, $R(p, a2)$, $R(p, a3)$, $A1(s)$, $A2(t)$, $A3(u)$, $T \sqsubseteq \leq 2R.T$, $A1 \sqsubseteq A2$, $A2 \sqcap A3 \sqsubseteq \bot$.	Consistente	Correcto
17	$a1 =_m A1$,$a2 =_m A2$, $a3 =_m A3$, $R(p, a1)$, $R(p, a2)$, $R(p, a3)$, $S(a1, a2)$, $S(a2, a3)$, $A1(s)$, $A2(t)$, $A3(u)$, $T \sqsubseteq \leq 2R.T$, $T \sqsubseteq \leq 1S.T$, $A1 \equiv A2$, $A2 \ A3 \sqsubseteq \bot$.	Inconsistente	Correcto
18	$a1 =_m A1$,$a2 =_m A2$, $a3 =_m A3$, $R(p, a1)$, $R(p, a2)$, $R(p, a3)$, $S(a1, a2)$, $S(a2, a3)$, $A1(s)$, $A2(t)$, $A3(u)$, $T \sqsubseteq \leq 2R.T$, $T \sqsubseteq \leq 1S.T$, $A2 \sqsubseteq A3$, $A1 \sqcap A2 \sqsubseteq \bot$.	Consistente	Correcto
19	$a =_m A$, $b =_m B$, $C \sqsubseteq \exists R.A$, $C \sqsubseteq \forall R.\neg B$, $A(p)$, $B(q)$, $C(r)$, $a = b$.	Inconsistent.	Correcto
20	$p = a$, $q = b$, $p = q$, $A \equiv A1 \sqcap A2 \sqcup \forall R.A3$, $A \equiv A4 \sqcup \neg A5$, $B \equiv B1 \sqcap \exists R.B2$, $B \equiv \exists R.B3 \sqcap B4$, $\neg A4(p)$, $A5(p)$, $\exists R.B3(p)$, $B4(p)$	Consistente	Correcto

21	p = a, q = b, p = q, A ≡ A1 ⊓ A2 ⊔ ∀R.A3, A ≡ A4 ⊔ ¬A5, B ≡ B1⊓ ∃R.B2, B ≡ ∃R.B3⊓B4, ¬A4(p), A5(p), ∃R.B3(p), B4(p), a $=_m$ A, b $=_m$ B.	Inconsistente	Correcto
22	A ≡ A1 ⊔ A2, a = b, A1(p), ¬B(p) a $=_m$ A, b $=_m$ B.	Inconsistente	Correcto
23	A ≡ X, a = b, X(p), ¬B(p) a $=_m$ A, b $=_m$ B.	Inconsistente	Correcto

Casos de prueba - Difference:

1	a1 $=_m$ A1,a2 $=_m$ A2, A1 ≡ A2, A1(p).	Consistente	Correcto
2	a1 $=_m$ A1,a2 $=_m$ A2, A1 ≡ A2, A1(p), a1 ≠ a2.	Inconsistente	Correcto
3	a1 $=_m$ A1,a2 $=_m$ A2, A1(p),A1 ⊑ B, B ⊑ A2, A2 ⊑ A1 ⊓ C, a2 ≠ a3, P(q, a1), P(q, a3).	Consistente	Correcto
4	a1 $=_m$ A1,a2 $=_m$ A2, A1(p),A1 ⊑ B, B ⊑ A2, A2 ⊑ A1 ⊓ C, a2 ≠ a3, P(q, a1), P(q, a3), P funcional	Inconsistente	Correcto
5	a1 $=_m$ A1, a2 $=_m$ A2, R(c, a1), R(c, a2), C(c), C ⊑≥ 2R.T.	Consistente	Correcto
6	a1 $=_m$ A1, a2 $=_m$ A2, R(c, a1), R(c, a2), C(c), a1 = a2, C ⊑≥ 2R.T.	Consistente	Correcto
7	a1 $=_m$ A1, a2 $=_m$ A2, R(c, a1), R(c, a2), C(c), D(c), A1(d), ¬A2(d), C ⊑≥ 2R.T, D ⊑≤ 2R.T.	Consistente	Correcto
8	a1 $=_m$ A1, a2 $=_m$ A2, A1 ≡ A2, R(c, a1), R(c, a2), C(a1), D(a2), T ⊑≥ 2R.(C ⊔ D), T ⊑≤ 1R.C ⊓ ≤ 1R.D.	Inconsistente	Correcto
9	A1(p), A2(q), ¬A3(s), R(r, s), a1 $=_m$ A1,a2 $=_m$ A2, a1 ≠ a2, ∃R.A3 ⊑ (¬A1 ⊔ A2) ⊓ (¬A2 ⊔ A1).	Consistente	Correcto
10	A1(p), A2(q), T ⊑ ∃R.A3, a1 $=_m$ A1,a2 $=_m$ A2, a1 ≠ a2, ∃R.A3 ⊑ (¬A1 ⊔ A2) ⊓ (¬A2 ⊔ A1).	Inconsistente	Correcto
11	A1(p), A2(q), A3(s), A4 ⊑ ∃R.A3, a1 $=_m$ A1,a2 $=_m$ A2, a1 ≠ a2, ∃R.A3 ⊔ ¬A4 ⊑ (¬A1 ⊔ A2) ⊓ (¬A2 ⊔ A1).	Inconsistente	Correcto
12	p = a, q = b, p ≠ q, A(p), A ≡ A1⊓ A2 ⊔ ∀R.A3, A ≡ ∃S.A4 ⊓ (A6 ⊔ A7), B ≡ B1 ⊔ B2, B ≡ ∃R.B3 ⊔ ∀R.¬B3, T ≡ ∃S.A4, T ≡ (A6 ⊔ A7),	Consistente	Correcto
13	p = a, q = b, p ≠ q,A(p),A ≡ A1 ⊓ A2 ⊔ ∀R.A3, A ≡ ∃S.A4 ⊓ (A6 ⊔ A7), B ≡ B1 ⊔ B2, B ≡ ∃R.B3 ⊔ ∀R.¬B3,T ≡ ∃S.A4, T ≡ (A6 ⊔ A7), a $=_m$ A, b $=_m$ B.	Inconsistente	Correcto
14	a ≠ b, X ≡ A, X ⊑ B, T(p).	Inconsistente	Fallo

Casos de prueba - Cycles:

1	A(a), a $=_m$ A.	Inconsistente	Correcto
2	2. A(b), B(a), a $=_m$ A, b $=_m$ B	Inconsistente	Correcto

3	B(a), B ⊑ A, a $=_m$ A.	Inconsistente	Correcto
4	A3(a), ∃R.A2(a), A ≡ A1 ⊓ A2, A ≡ A3 ⊓ ∃R.A2.	Consistente	Correcto
5	A3(a), ∃R.A2(a), A ≡ A1 ⊓ A2, A ≡ A3 ⊓ ∃R.A2, a $=_m$ A.	Inconsistente	Correcto
6	p = a, q = b, A3(q), ∃R.A2(q), B3(p), ∀R.B2(p),A ≡ A1 ⊓ A2, A ≡ A3 ⊓ ∃R.A2,B ≡ B1 ⊔ B2, B ≡ B3 ⊓ ∀R.B2,	Consistente	Correcto
7	7. p = a, q = b,A3(q), ∃R.A2(q),B3(p), ∀R.B2(p),A ≡ A1 ⊓ A2, A ≡ A3 ⊓ ∃R.A2,B ≡ B1 ⊔ B2, B ≡ B3 ⊓ ∀R.B2,a $=_m$ A, b $=_m$ B.	Inconsistente	Correcto
8	B(a'), P(p, a), P(p, a'), P funcional, B ⊑ A, a $=_m$ A.	Inconsistente	Correcto
9	A3(a), ∃R.A2(a), A ≡ A1 ⊓ A2, X ≡ A3 ⊓ ∃R.A2, X ⊑ A.	Consistente	Correcto
10	A3(a), ∃R.A2(a), A ≡ A1 ⊓ A2, X ≡ A3 ⊓ ∃R.A2, X ⊑ A, a $=_m$ A.	Inconsistente	Correcto
11	p = a, q = b, A3(q), ∃R.A2(q), B3(p), ∀R.B2(p), A ≡ A1 ⊓ A2, X ≡ A3 ⊓ ∃R.A2, X ⊑ A, B ≡ B1 ⊔ B2, Y ≡ B3 ⊓ ∀R.B2, Y ⊑ B	Consistente	Correcto
12	p = a, q = b, A3(q), ∃R.A2(q), B3(p), ∀R.B2(p), A ≡ A1 ⊓ A2, X ≡ A3 ⊓ ∃R.A2, X ⊑ A, B ≡ B1 ⊔ B2, Y ≡ B3 ⊓ ∀R.B2, Y ⊑ B, a $=_m$ A, b $=_m$ B.	Inconsistente	Correcto
13	a $=_m$ A,b $=_m$ B, c $=_m$ A, A(b), B(c).	Inconsistente	Correcto
14	a $=_m$ A,b $=_m$ B, a1 $=_m$ A1, A(b), B(a1), A ≡ A1.	Inconsistente	Correcto
15	a $=_m$ A,b $=_m$ B, a1 $=_m$ A1, A(b), B(a1), X ≡ A, X ≡ A1.	Inconsistente	Correcto
16	a $=_m$ A,b $=_m$ B, a1 $=_m$ A1, A(b), B(a1), X ≡ A2 ⊔ A3, X ≡ A1, A ⊑ A2 ⊔ A3, A2 ⊑ A, A3 ⊑ A,.	Inconsistente	Correcto

Casos de prueba - Conservativity:

1	A1(p),¬A2(p), a1 = a2.	Consistente	Correcto
2	A1 ≡ A2, A1(p), a1 ≠ a2.	Consistente	Correcto
3	A(b), B(a)	Consistente	Correcto
4	A ≡ B, C ⊑ ∃R.A, C ⊑ ∀R.¬B, A(p), B(q), C(r).	Inconsistente	Correcto

Tutorial plugin OntoRed y razonador Pellet

Requisitos

- Protégé 4.3
- Java 7 (JRE) o superior
- ontologías en formato OWL/XML
- plugin OntoRed
- OWLAPI versión modificada
- plugin Pellet versión modificada

Instalación

Los archivos necesarios para instalar la herramienta se pueden obtener en la carpeta con nombre *Archivos* ó en el siguiente enlace www.cs.le.ac.uk/people/ps56/pelletM.

OntoRed y OWLAPI

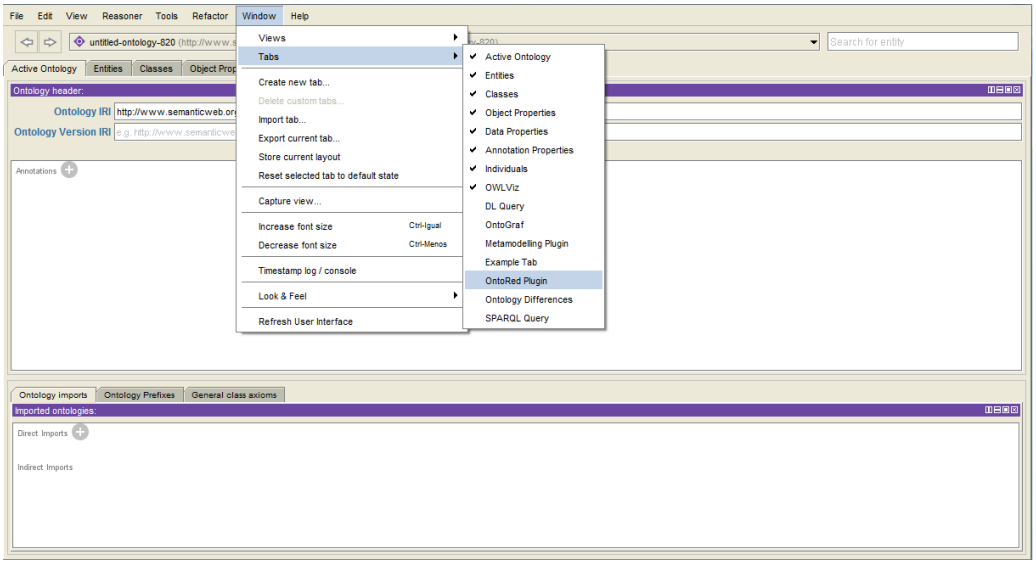
El plugin OntoRed está desarrollado para la versión 4.3 de Protégé y la versión 7 o superior de Java RE. Para instalarlo es necesario copiar los archivos jar OntoRed.jar y org.semanticweb.owl.owlapi.jar dentro de la carpeta “plugins” que se encuentra en la carpeta de instalación de Protégé. Será necesario reemplazar el jar org.semanticweb.owl.owlapi.jar que viene por defecto con Protégé por la nueva versión modificada.

Pellet plugin

Para utilizar el razonador Pellet en Protégé es necesario instalar la versión modificada del plugin. Al igual que OntoRed, se debe copiar el archivo jar **com.clarkparsia.protege.plugin.pellet.jar** en la carpeta “*plugins”* de Protégé.

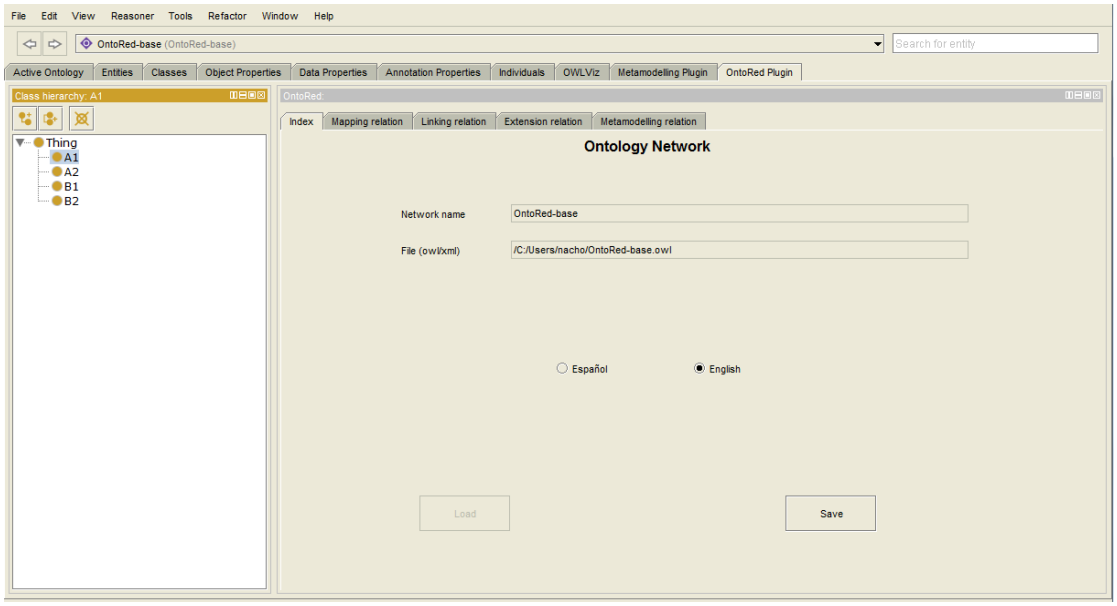
Inicio

Para iniciar el plugin en Protégé hay que seleccionar en **Window** -> **Tabs** la opción **OntoRed** Plugin. Con eso se abrirá una nueva Tab correspondiente al plugin.
La red de ontologías, se crea como cualquier ontología normal de **Protégé**, utilizando el formato **OWL/XML** para grabar la misma.



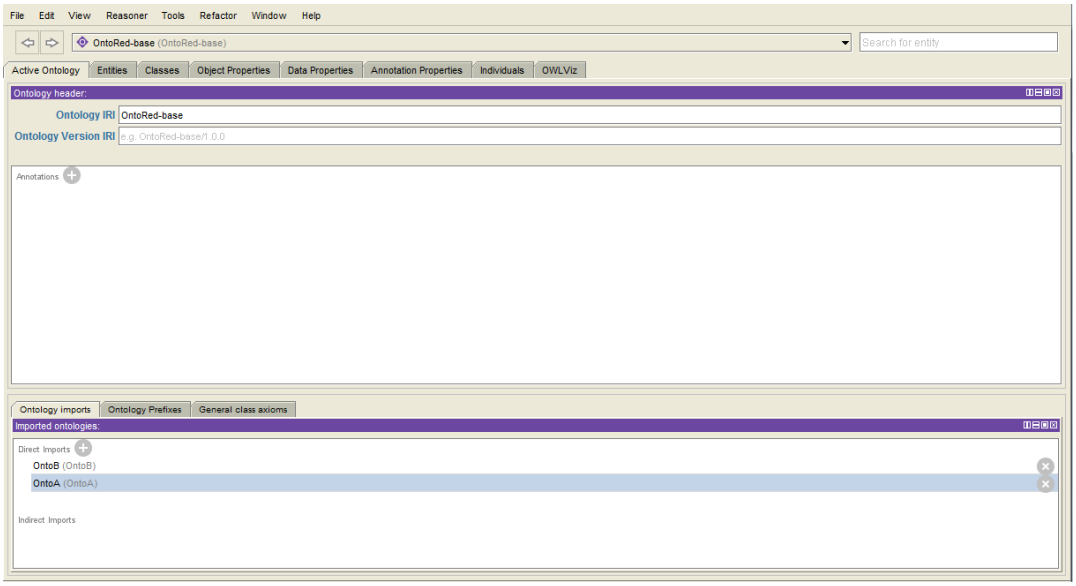
Plugin

El plugin consiste de 5 tabs. En la primera, la tab inicio, se muestra la información de la ontología (red) cargada, su nombre y el archivo donde esta guardada. Desde aquí se puede cargar una red y guardar la misma. También es posible seleccionar el idioma entre Español e Inglés.
Las otras 4 tabs son para el manejo de las relaciones.



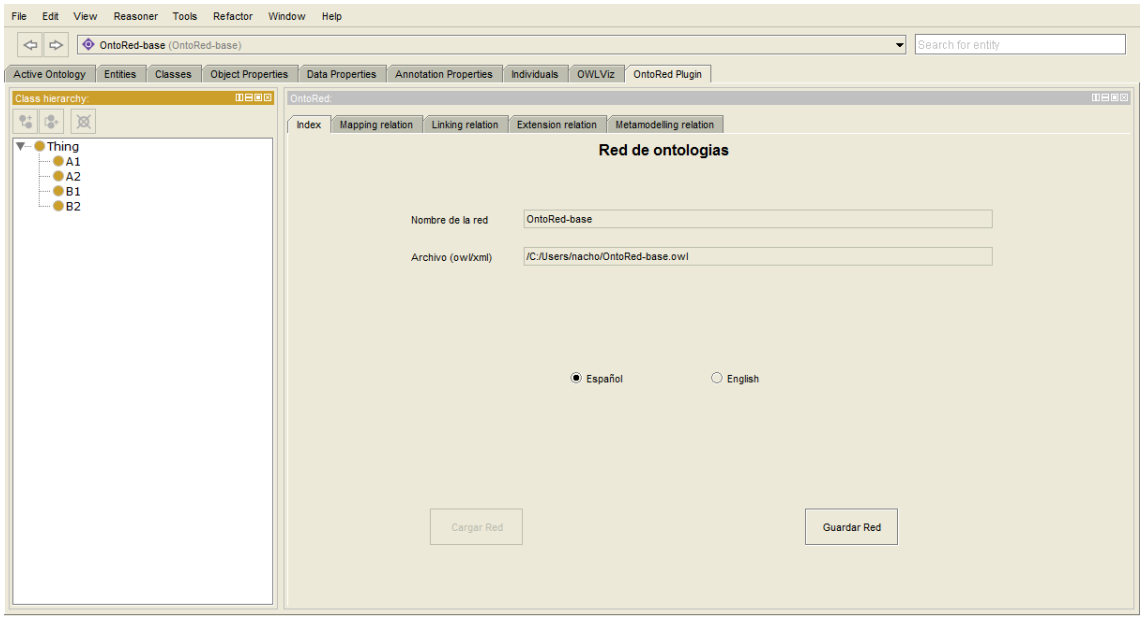
Imports

Una **red de ontologías** es una ontología que importa otras ontologías. Para importar las mismas, se utiliza el ***import*** que provee Protégé. En la tab inicial, en la parte inferior donde dice ***import*** se seleccionan las ontologías que se desea importar. Desde aquí también, se pueden quitar las ontologías que no queramos que pertenezcan a la red.



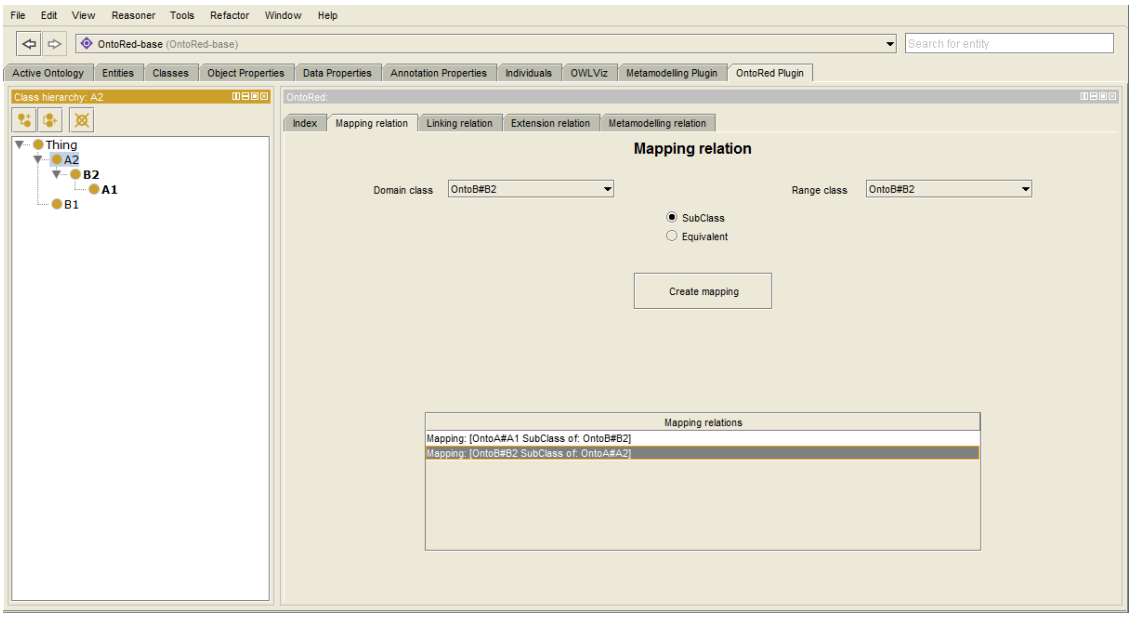
crear/cargar red

Una vez que tenemos creada o cargada en Protégé la ontología que representa a la red (formato OWL/XML), vamos al tab del plugin (OntoRed) y damos click en **Cargar Red**. Con esto el plugin cargará las relaciones que existan en sus tabs correspondientes.



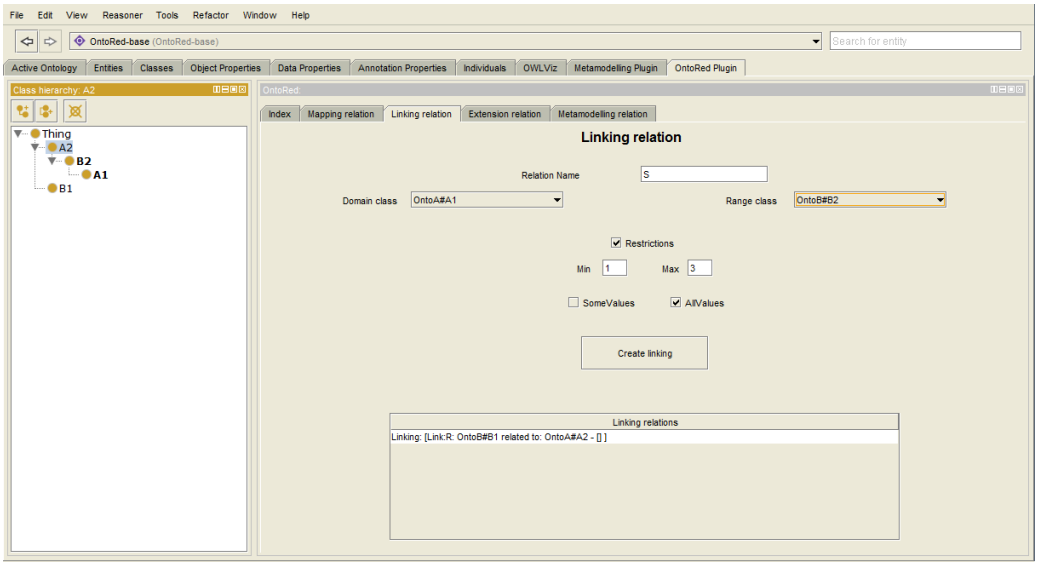
tab mapping

El tab mapping permite crear relaciones entre clases de distintas ontologías del tipo **subclass** o **equivalent**.
Para esto se selecciona una clase origen y una destino, el tipo de mapping, y se da click en *Crear Mapping*.
Todas las relaciones creadas de este tipo se pueden visualizar en la lista de más abajo. Para eliminar una relación creada, se da click sobre la misma.



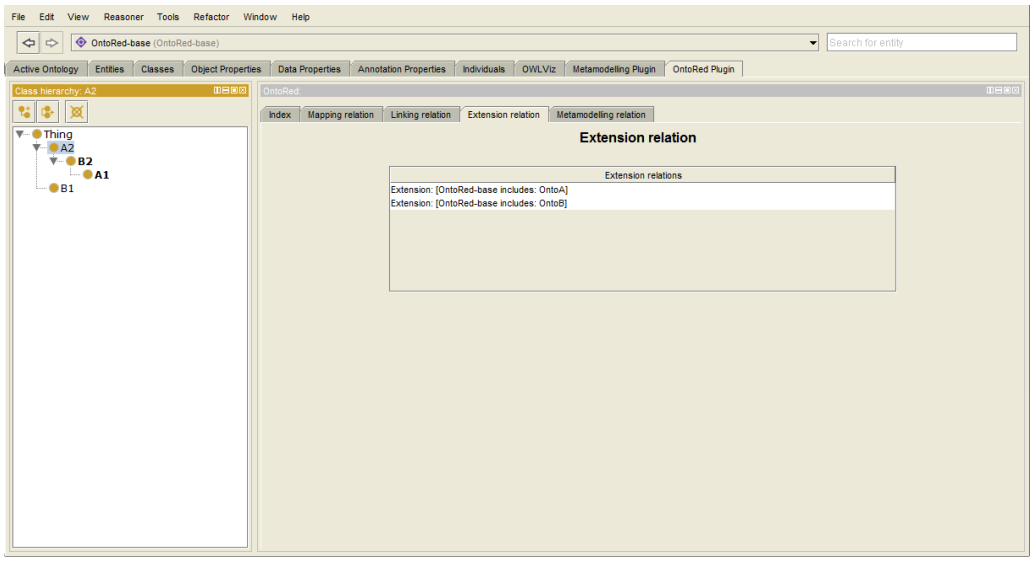
tab linking

El tab linking permite crear relaciones entre clases de diferentes ontologías del tipo ObjectProperty, con restricciones tanto de **cardinalidad** como **someValues** y **allValues** si se desea.
Para ello, se seleccionan las clases de la relación y se marcan las restricciones deseadas.
En la lista de la parte inferior también se pueden visualizar y eliminar las relaciones de tipo linking.



tab extension

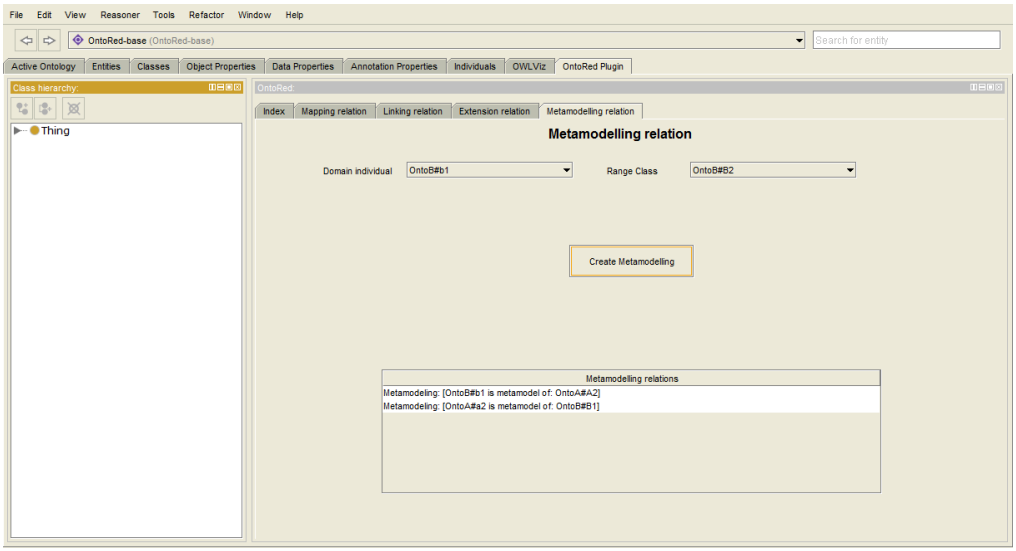
Las relaciones del tipo extensión corresponden a las importaciones de ontologías en la red. En esta tab se visualizan las relaciones, pero el manejo (creación y eliminación) se realiza como se vió antes, mediante la funcionalidad de Protégé.



tab metamodelling

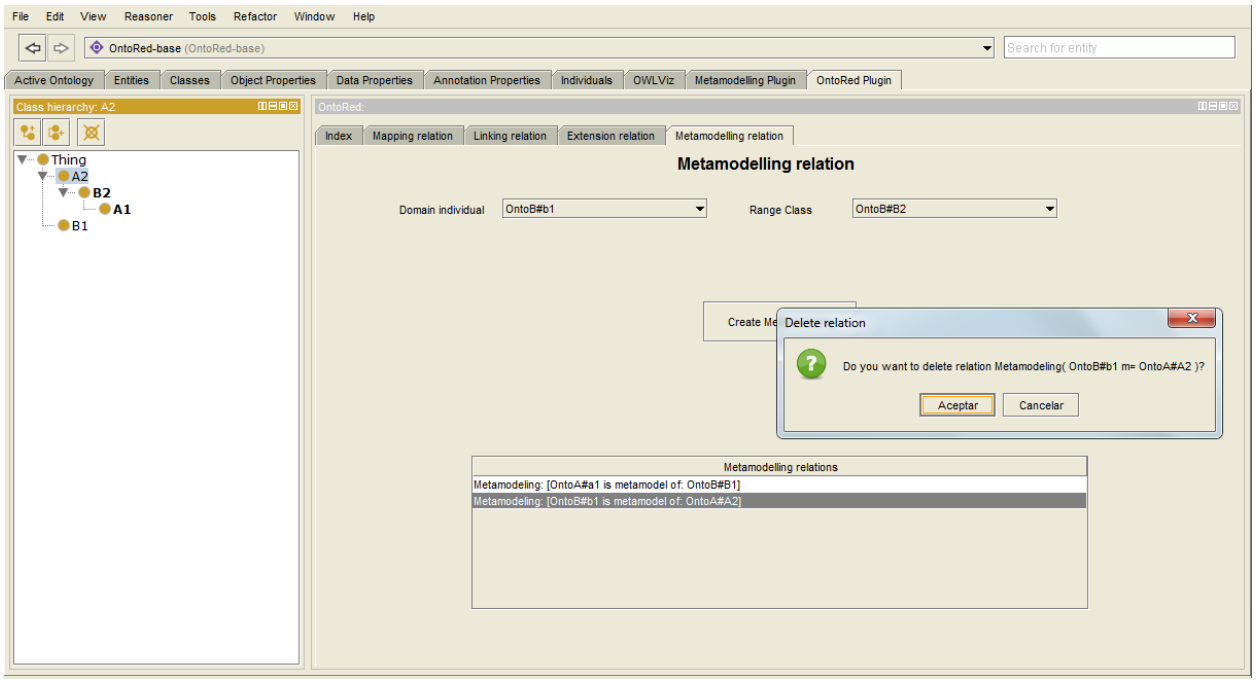
Finalmente está el tab metamodelling. Este nos permite crear relaciones entre instancias y clases de ontologías diferentes. Representa una relación que no existe como tal en el lenguaje OWL, una equivalencia entre una instancia y una clase.

Funciona igual que mapping y linking, se selecciona origen y destino de la relación y se da click en **Crear**. También se pueden visualizar y eliminar las relaciones creadas en la lista de la parte inferior



eliminar relación

Como se mencionó, para eliminar una relación, se da click sobre la misma.

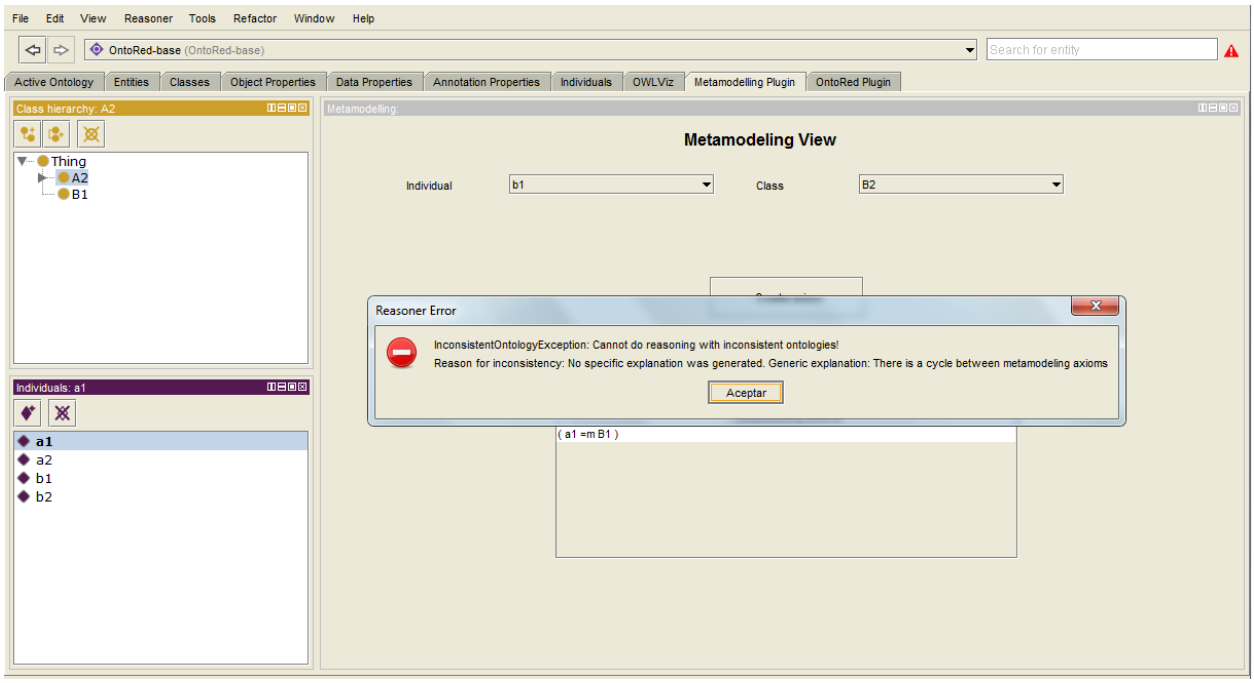


razonador Pellet

Para utilizar el razonador Pellet se debe configurar e iniciar desde el menú *Reasoner.*
Primero se selecciona el razonador haciendo *Reasoner-> Pellet,* y luego se inicia el razonamiento haciendo *Reasoner->Start reasoner.*

Pellet realiza el chequeo de consistencia de la red cargada e infiere nuevo conocimiento sobre esta. Si se modifica la red, es necesario sincronizar el razonador para hacer un nuevo chequeo, esto se logra seleccionado *Reasoner->Synchronize reasoner.*

Si existe una inconsistencia en la red, se muestra un mensaje de error que indica esto.

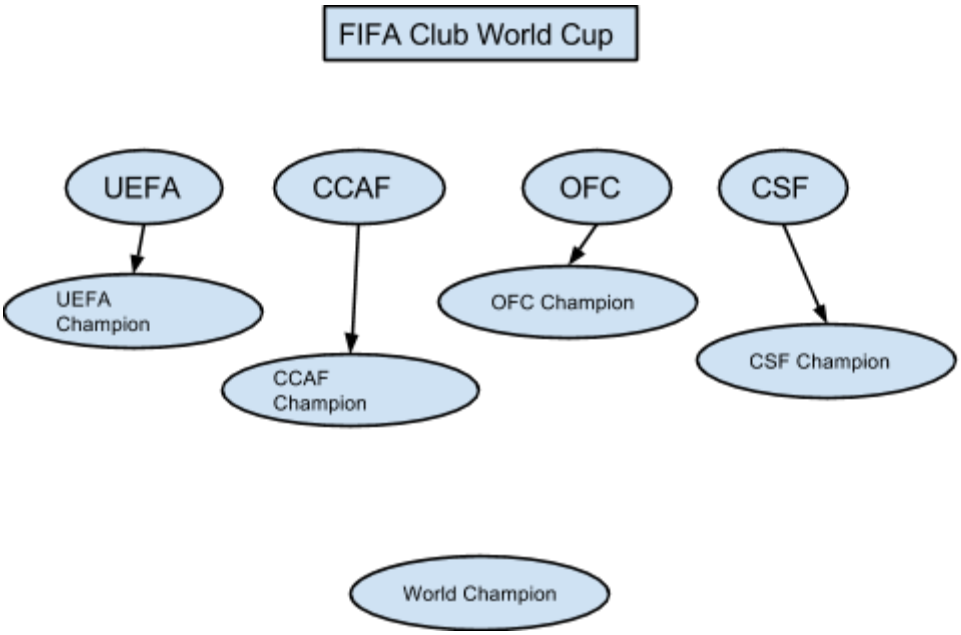


Mensaje de error por inconsistencia en la red.

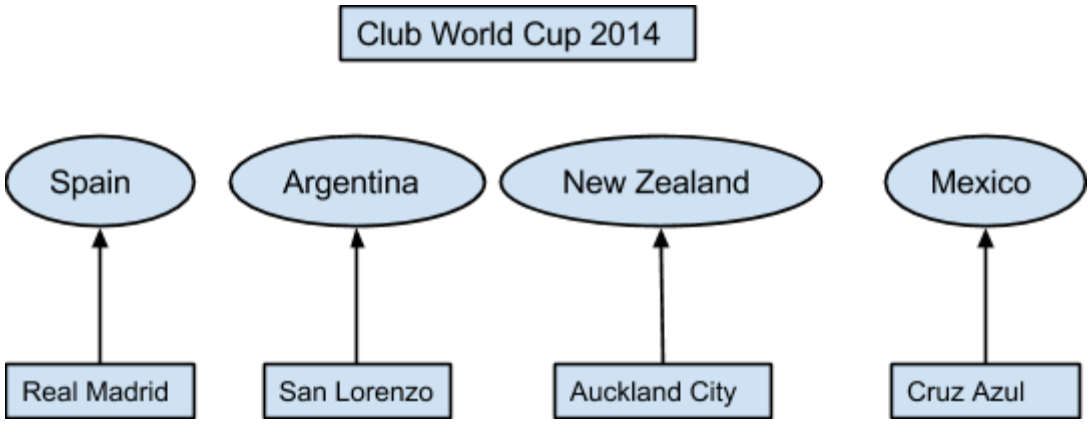
Red de ejemplo

Para mostrar el funcionamiento básico de OntoRed plugin, utilizaremos una red que modela el Mundial de clubes FIFA.
La red estará formada por una ontología que modela el formato del torneo, y otra ontología que representa la edición del año 2014.

En primer lugar, tenemos la ontologia **FIFA_WC** que modela el torneo de forma simplificada. En el mismo compiten los campeones de las confederaciones FIFA de cada continente (UEFA, CONCACAF, CSF y OFC). El resultado del torneo es un ***Campeón Mundial***. Todos los conceptos son modelados como clases.



Por otro lado, tenemos al ontología con los representantes de la edición 2014 del torneo. En este caso modelamos los países de los equipos con clases, y a los equipos como individuos.



Como primer paso, creamos las ontologías **FIFA_WC** y **WC_2014** normalmente usando Protégé (o usamos las adjuntas en los archivos correspondientes). Luego, iniciamos Protégé y en una nueva ontología vamos a importar las dos anteriores, de esta forma creamos la red.

Ahora, vamos al plugin, y damos click en “cargar red”, nos cargará la red vacía, solo con los import realizados.

Vamos a crear las siguientes relaciones:

En la **pestaña de Mapping,** **modelamos que los países de los equipos pertenecen a una Confederación:**

- **<Spain subClassOf UEFA>**
- **<Mexico subClassOf CCAF>**
- **<Argentina subClassOf CSF>**
- **<New_Zealand subClassOf OFC>**

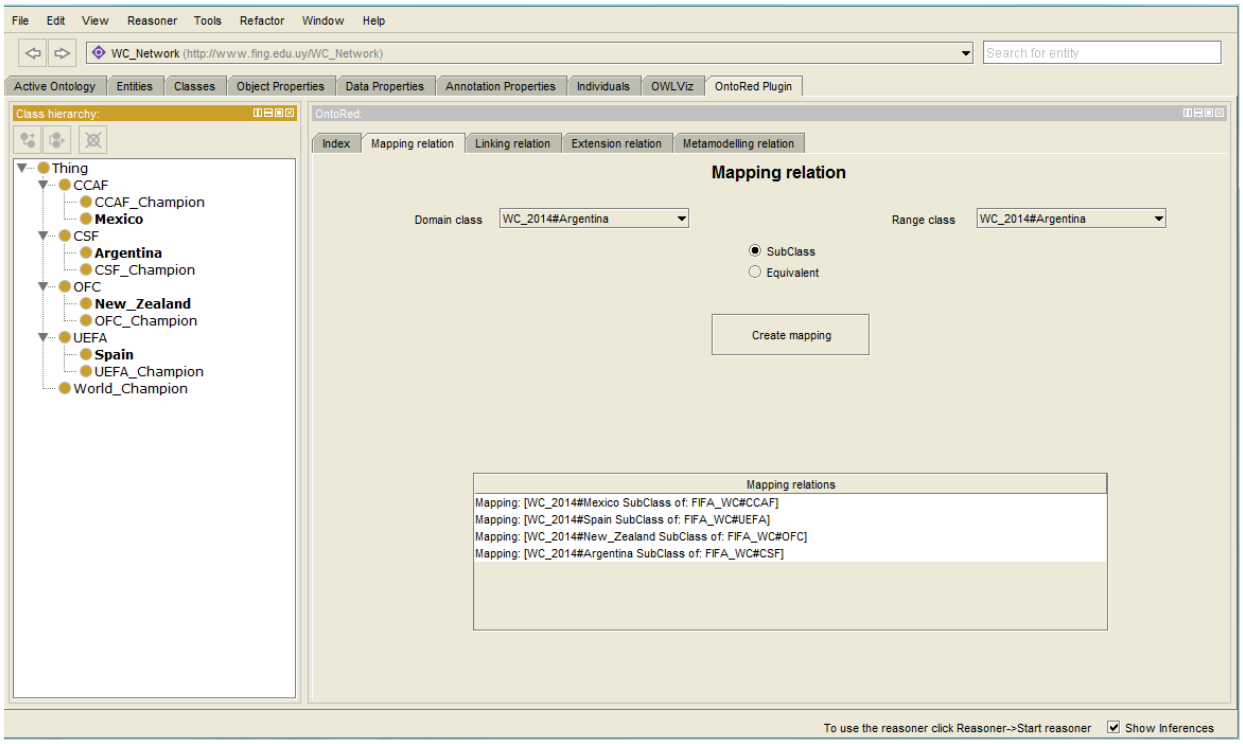
En la pestaña de Linking, modelamos que el campeón mundial es español:

- **<World_Champion isFrom Spain>**

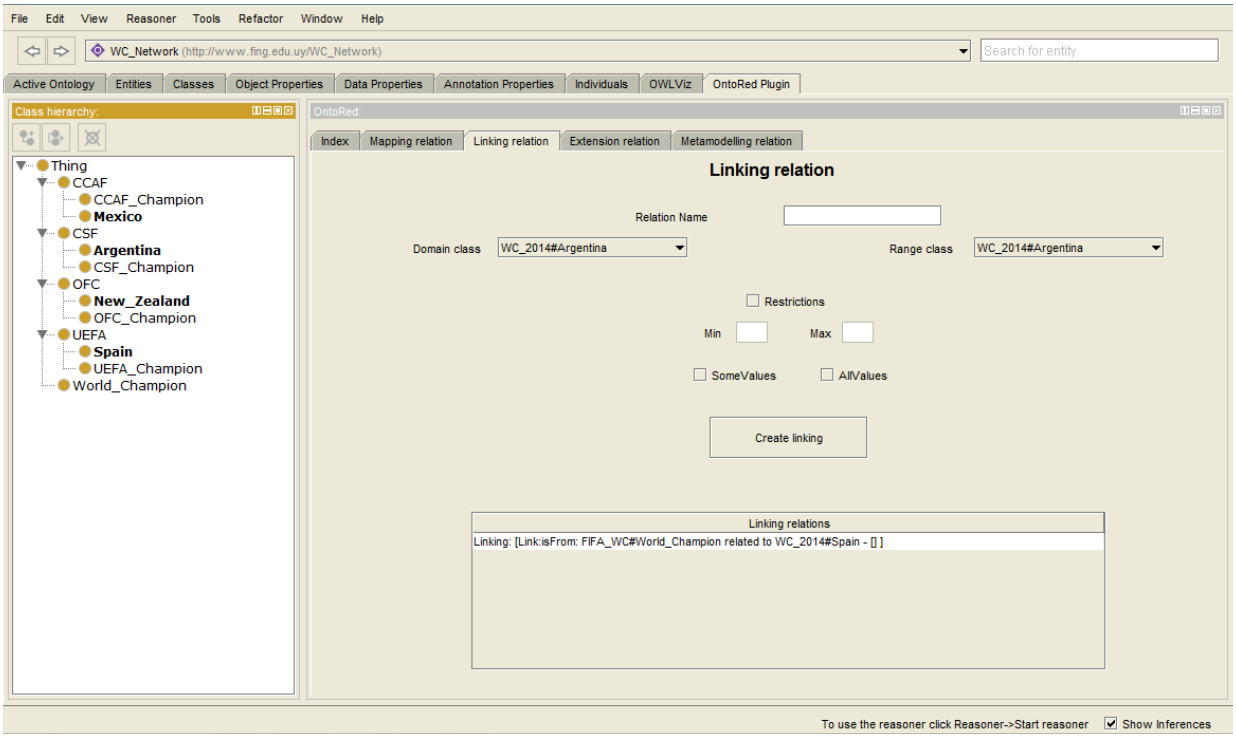
Finalmente, en la pestaña de Metamodelling, vamos a crear la equivalencia entre los clubs que son individuos en WC_2014, con las clases que los representan en la ontologia **FIFA_WC:**

- <Real_Madrid $=_m$ UEFA_Champion>
- <Auckland_City $=_m$ OFC_Champion>
- <San_Lorenzo $=_m$ CSF_Champion>
- <Cruz_Azul $=_m$ CCAF_Champion>
- <Real_Madrid $=_m$ World_Champion>

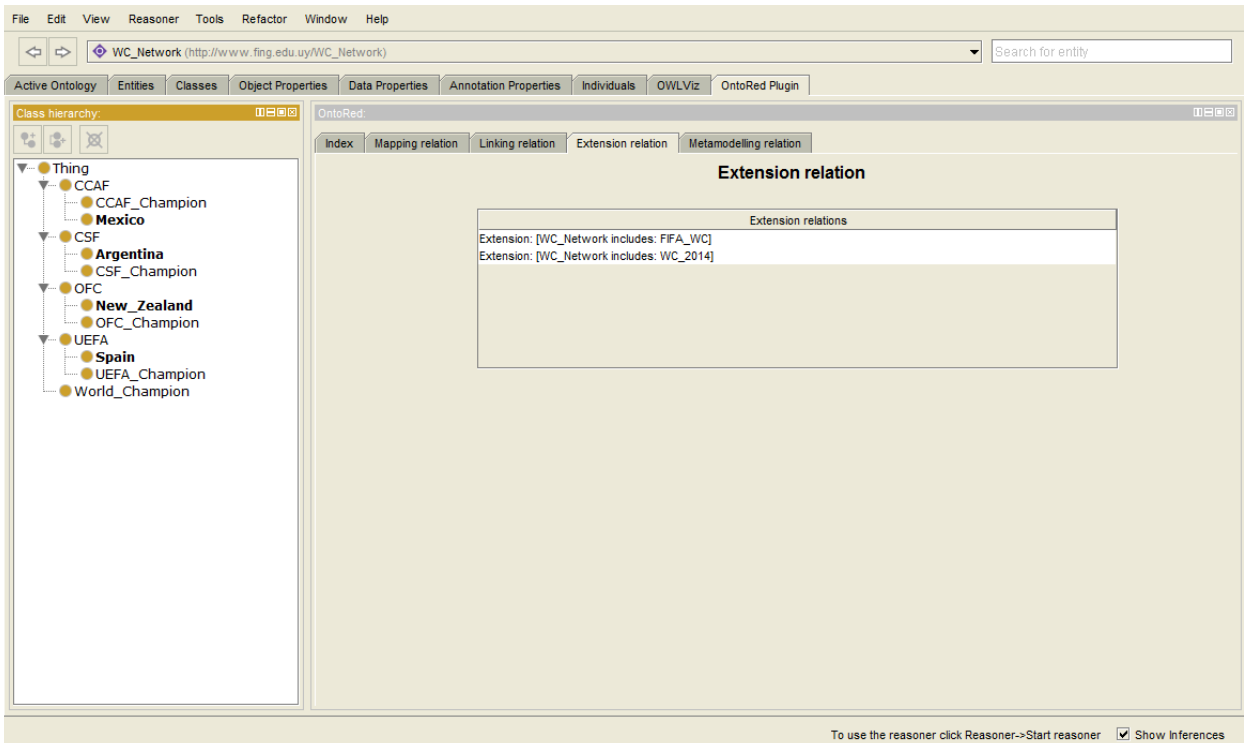
Relaciones Mapping



Relaciones Linking



Relaciones Extension



Relaciones Metamodelling