Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Proyecto Fin de Carrera
Ingeniería en Informática

Visualizador de ontologías basado en un razonador de Lógica Descriptiva

Jorge Bobed Lisbona

Directores: Carlos Bobed, Eduardo Mena

Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza

Septiembre de 2012

Visualizador de ontologías basado en un razonador de Lógica Descriptiva

RESUMEN

Durante los últimos años, la Web está evolucionando en lo que se denomina la Web Semántica. En esta nueva Web, los contenidos pasan a ser entendidos y procesados por los ordenadores para automatizar y mejorar distintas tareas. Esto se consigue mediante la adición de información semántica a dichos contenidos. Dicha información es ofrecida por las ontologías, definidas por Tomas Grüber como la "especificación de una conceptualización"[9], que permiten modelar y definir distintas vistas del mundo de manera que el ordenador pueda entenderlas y manipularlas. Actualmente, el formalismo más extendido para la definición de ontologías son los llamados lenguajes Description Logics (DLs). Los DLs ofrecen un marco teórico que permite la existencia de razonadores, programas que permiten razonar e inferir nuevos hechos sobre las ontologías, encontrar relaciones no explícitas, clasificar los conceptos e instancias de las ontologías, o encontrar inconsistencias en los modelos, entre otras tareas. Este formalismo es el que se encuentra tras el estándar W3C de representación de ontologías: OWL (*Web Ontology Language*).

Asimismo, el correcto modelado y uso de una ontología recae en el conocimiento del conjunto de conceptos y relaciones a representar, y en su dominio sobre la alta complejidad de los formalismos de los DLs. Por tanto, es vital un visualizador que ayude tanto al creador encontrando posibles fallos de modelado (inconsistencias) como a desarrolladores que pretendan entender y utilizar ontologías de terceros.

El objetivo de este PFC ha sido por tanto la realización de un visualizador de ontologías semánticamente expresivo. El problema de encontrar una representación visual acorde a la complejidad de estos esquemas no es nuevo. Existen diversos visualizadores con aproximaciones muy diferentes con sus puntos fuertes y débiles. Con el visualizador propuesto se ha pretendido suplir esas carencias que van desde la ausencia de apoyo sobre un razonador a la omisión de entidades como clases anónimas o la representación de la jerarquía de propiedades en el mismo esquema. Para llevarlo a cabo ha sido vital la experiencia del grupo SID en el campo.

Con el visualizador se ha conseguido:

- Ofrecer un lenguaje visual expresivo y semánticamente correcto que facilite el entendimiento de la ontología.
- Permitir observar la jerarquía de conceptos o clases, así como las clases anónimas que puedan surgir de la definición de éstos.
- Permitir al usuario ocultar parcialmente el grafo, de modo que pueda centrarse en los aspectos que más le interesen.

- Modificar la disposición de los elementos en pantalla para dejarlos según las preferencias personales.
- Ser capaz de guardar el estado visual y la configuración en ficheros XML, para poder restaurarlo en futuras sesiones.
- Exportar como distintos tipos de imagen (JPG y PNG) para poder visualizar posteriormente con un visor de imágenes.
- Añadir la jerarquía de roles y propiedades sobre el grafo.
- Poder listar instancias de las distintas clases.
- Interactuar con los distintos elementos: Desplazar nodos a voluntad, ocultar subníveles de estos, las propiedades asociadas, mostrar información adicional al de la entidad que tiene el foco de atención, realizar zoom sobre el grafo, etc.

Además se ha planteado el desarrollo del mismo como aplicación propia y como plug-in de la plataforma Protégé. La decisión de la integración en Protégé, como plug-in viene justificada por el aumento de usabilidad que supone en un entorno de edición ampliamente extendido en la comunidad de la Web Semántica. Como lenguaje de programación, se ha utilizado Java, dado que tanto Protégé como la API más extendida (OWLAPI) están desarrollados en dicho lenguaje. Por otro lado, se hace uso intensivo de razonadores DL explotando sus capacidades para ofrecer información relevante para la explicación de la ontología. La selección de éstos viene dada por su compatibilidad con OWLAPI (Pellet, Hermit, Fact++ y RacerPro, entre otros).

Agradecimientos

Es mucha la gente que me ha ayudado tanto de forma directa con su apoyo y consejos (mis directores, Roberto, Gorka, Ricardo ....) como indirecta con el ánimo y apoyo que me han proporcionado en momentos buenos y malos (familia, amigos, Marina...). A todos ellos, va dedicada esta sección.

Nobody expects the spanish inquisition![16].

Índice general

1. Introducción **1**

1.1. Motivación 1

1.2. Objetivo y alcance 3

1.3. Contenido de la memoria 4

2. Contexto Tecnológico **5**

2.1. Conceptos relacionados con el proyecto 5

2.1.1. Ontologías 5

2.1.2. Description Logics 6

2.2. Lenguajes de representación 7

2.2.1. RDF 7

2.2.2. RDFS 8

2.2.3. OWL 9

2.3. Herramientas ontológicas 10

2.3.1. Razonadores DL 10

2.3.2. OWLAPI 11

2.4. Software de apoyo 11

3. Estado del arte **13**

3.1. Herramientas de visualización 13

3.1.1. GrOwl 13

3.1.2. OWLViz 14

3.1.3. OntVis 15

3.1.4. Ontograf 17

3.1.5. Resumen comparativo 18

4. Planificación del visualizador **21**

4.1. Análisis de requisitos 21

4.1.1. Requisitos 21

4.1.2. Metodología seguida 21

4.2. Elementos a visualizar 23

4.3. Diseño propuesto 27

5. Implementación del visualizador **31**

5.1. Algoritmo de visualización 31

5.1.1. Obteniendo la taxonomía 31

5.1.2. Añadiendo clases definidas 32

5.1.3. Reorganizar el grafo 33

5.1.4. Ajuste de posición y anchura 34

5.1.5. Expansión del grafo : Añadiendo restricciones 34

5.1.6. Distribución de los nodos en un plano 2D (Algoritmo de Sugiyama) 36

5.2. Características generales 37

5.2.1. Diferencias Plugin-Applet 37

5.2.2. Instancias 37

5.2.3. Características configurables 37

5.3. Utilización del prototipo 38

5.3.1. Obtención de la aplicación y uso 39

5.3.2. Probando la aplicación 39

6. Conclusiones **43**

6.1. Conclusiones generales 43

6.2. Línea de posibles mejoras 44

6.3. Conclusiones personales 45

6.4. Marco temporal del proyecto 45

A. Evaluación **49**
A.1. Ontologías pequeñas . . . 50
A.1.1. Proyectos . . . 50
A.1.2. Family . . . 51
A.2. Ontologías medianas . . . 52
A.2.1. Animals . . . 52
A.2.2. SemanticGranules . . . 54
A.3. Ontologías grandes . . . 55
A.3.1. Snomed . . . 55
A.3.2. DBpedia . . . 56
A.3.3. Sumo . . . 57

B. Manual de usuario **67**
B.1. Instalación del plug-in . . . 67
B.2. Utilización del plug-in . . . 67
B.2.1. Opciones del plug-in . . . 67
B.3. Applet . . . 73

C. Web Semántica e integración de datos **77**
C.1. Componentes de la web semántica . . . 78
C.2. Description Logics . . . 79
C.2.1. Definición del formalismo . . . 80
C.2.2. Lenguajes de descripción . . . 81
C.3. Web Ontology Language (OWL) . . . 82
C.3.1. Clases simples con nombre . . . 82
C.3.2. Individuos . . . 82
C.3.3. Propiedades . . . 83
C.3.4. Clases complejas . . . 84

D. Uso del razonador **85**

E. Disposición visual del grafo (Algoritmo de Sugiyama) **91**

F. Gestión de versiones **93**

G. Preparación del entorno **97**

G.1. Setup de Eclipse 97

G.1.1. Estructura de los ficheros 97

G.2. Distribución del plug-in 99

H. Diagrama de clases **101**

Bibliografía **120**

Índice de figuras

1.1. Arquitectura de los lenguajes semánticos. 2

2.1. Ejemplo de tripleta RDF. 7
2.2. Ejemplo de grafo RDF. 8
2.3. Extracto de RDF en XML. 8
2.4. Ejemplo de construcción OWL. 9
2.5. Diagrama de clases de un subconjunto de OWLAPI. 12

3.1. Vista de ontología del visualizador GrOwl. 14
3.2. Captura de OWLViz. 15
3.3. Captura de ontVis. 16
3.4. Captura de Ontograf. 18

4.1. Clase con nombre. 23
4.2. Clase definida. 24
4.3. Clase anónima. 24
4.4. Restricciones. 24
4.5. Propiedades. 25
4.6. Tipos de conectores en el grafo. 27
4.7. Conectores de herencia en propiedades. 27
4.8. Primer boceto de diseño. 28
4.9. Segundo prototipo de diseño (panel de control). 28
4.10. Prototipo con propiedades. 29
4.11. Clases disjuntas y clases equivalentes. 29
4.12. Resultado visual final. 30

5.1. Definición (clase anónima) junto a clase definida. 33
5.2. Reordenando clases. 34
5.3. Expansión de nodo. 35
5.4. Grafo con layout Sugiyama. 36
5.5. Diferencias plug-in y applet. 37
5.6. Visualizando las instancias. 38
5.7. Configuración de parámetros sobre fichero XML. 38
5.8. Carga y clasificación de una ontología. 39
5.9. Visualización de proyectos.owl en OWLViz. 40
5.10. Visualización de proyectos.owl en OntoGraf. 41
5.11. Visualización expandida de proyectos.owl en el visualizador desarrollado. 41

6.1. Posible mejora con instancias. 44
6.2. Diagrama de Gantt y reparto de tiempo. 46

A.1. Visualización de proyectos.owl en el visualizador desarrollado. 50
A.2. Visualización expandida de proyectos.owl en el visualizador desarrollado. 51
A.3. Visualización de proyectos.owl en OWLViz. 51
A.4. Visualización de proyectos.owl en OntoGraf. 52
A.5. Visualización de family.owl en OWLViz. 53
A.6. Visualización de family.owl en OntoGraf. 54
A.7. Visualización de family.owl en el visualizador desarrollado. 55
A.8. Visualización de animals.owl en el visualizador desarrollado. 56
A.9. Visualización de animals.owl en el OWLViz. 57
A.10. Visualización de animals.owl en el OntoGraf. 58
A.11. Visualización de SemanticGranules.owl en OWLViz. 59
A.12. Visualización de SemanticGranules.owl en OntoGraf. 60
A.13. Visualización de SemanticGranules.owl en el visualizador desarrollado. 61
A.14. Visualización de DBPedia.owl en el visualizador desarrollado. 62
A.15. Visualización de DBPedia.owl en OWLViz. 63
A.16. Visualización de DBpedia.owl en OntoGraf. 64

A.17.Visualización de SUMO.owl. 65

B.1. Ventana principal de Protégé. 68
B.2. Añadiendo la pestaña del visualizador. 68
B.3. Vista general de plug-in. 69
B.4. Selección de razonador. 69
B.5. Opciones del panel del plug-in. 70
B.6. Menú contextual sobre un punto cualquiera. 70
B.7. Tooltip de una clase. 71
B.8. Cerrando una clase. 71
B.9. Ocultando una clase. 71
B.10.Arrastrando figura. 72
B.11.Empujando clases mediante repulsión. 73
B.12.Ocultando propiedades de una clase. 73
B.13.Panel delApplet. 74
B.14.Cargar ontología 1. 74
B.15.Cargar ontología desde diálogo. 74
B.16.Cargar razonador. 75

C.1. Arquitectura de los lenguajes semánticos. 78
C.2. Arquitectura de un sistema de representación del conocimiento basado en DL. 80
C.3. Conjunto de definiciones. 82

D.1. Propiedades ordenadas dentro de una clase. 88

E.1. Cruces entre aristas en algoritmo de Sugiyama. 92

F.1. Gestión de versiones con CVS en Eclipse. 95

H.1. Diagrama completo. 101
H.2. Diagrama 1 103
H.3. Diagrama 2. 105
H.4. Diagrama 3. 109
H.5. Diagrama 4. 113
H.6. Diagrama 5. 117
H.7. Diagrama 6. 119

Índice de cuadros

2.1. Constructores RDFS. 9

3.1. Resumen de características de los visualizadores. 19

4.1. Requisitos del sistema. 22
4.2. Símbolos utilizados en el renderizado de restricciones. 25

A.1. Clasificación de ontologías por complejidad y tamaño. 49

C.1. Reglas de sintaxis ALC. 81
C.2. Características de las propiedades. 83
C.3. Restricciones de propiedades. 84

Capítulo 1

Introducción

El presente documento presenta la memoria de trabajo del proyecto de fin de carrera "*Visualizador de ontologías basado en un razonador de Lógica Descriptiva*", realizado por el alumno Jorge Bobed Lisbona bajo la dirección de Eduardo Mena Nieto y Carlos Bobed Lisbona. Este proyecto se ha realizado dentro del Departamento de Informática e Ingeniería de Sistemas de la Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza, concretamente dentro del grupo de Sistemas de Información Distribuidos (SID).

1.1. Motivación

Durante los últimos años estamos observando la progresiva evolución de la Web en lo que se denomina la Web Semántica. En esta nueva Web, los contenidos pasan a ser entendidos y procesados por los ordenadores para automatizar y mejorar distintas tareas. Esto se consigue mediante la adición de información semántica a dichos contenidos. Esta información semántica es ofrecida por las ontologías, definidas por Tomas Grüber como la "especificación de una conceptualización explícita y formal"[12]. Las ontologías permiten modelar y definir distintas vistas del mundo de manera que el ordenador pueda entenderlas y manipularlas.

Actualmente, el formalismo más extendido para la definición de ontologías son los llamados lenguajes de Lógica Descriptiva (DLs, de su nombre en inglés *Description Logics*). Los DLs ofrecen un marco teórico que permite la existencia de razonadores, programas que permiten: razonar e inferir nuevos hechos a partir de lo asertado en las ontologías encontrar relaciones no explícitas, clasificar los conceptos e instancias de las ontologías, o encontrar inconsistencias en los modelos, entre otras. Este formalismo es el que se encuentra tras el estándar W3C de representación de ontologías, OWL (*Web Ontology Language*). Se trata de un lenguaje que se sustenta sobre más lenguajes de diferente expresividad decreciente como son: RDFS (*RDF Schema*), RDF (*Resource Description Framework*) y, finalmente, XML (*eXtensible*

Markup Language). En la Figura 1.1 se puede comprobar la relación existente entre los lenguajes mencionados

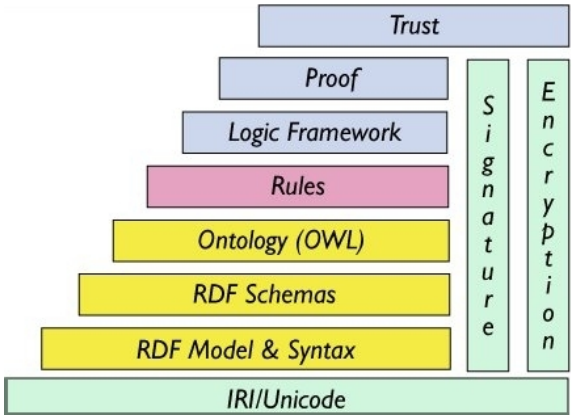


Figura 1.1: Arquitectura de los lenguajes semánticos.

Asímismo, el correcto modelado y uso de una ontología requiere el conocimiento del conjunto de conceptos y relaciones a representar, y dominio sobre la alta complejidad de los formalismos de los DL. Por tanto es vital un visualizador que ayude tanto al creador encontrando posibles fallos de modelado (inconsistencias) como a desarrolladores que pretendan entender y utilizar ontologías de terceros. Esta última tarea es crucial para la explotación y puesta en valor de iniciativas como la de Linked Data, que ofrece una ingente cantidad de información en formato RDF anotada con una relativamente pequeña cantidad de ontologías.

En este proyecto se utiliza la experiencia del grupo SID en el campo y se plantea la realización de un visualizador desarrollado como aplicación propia y como plug-in de la plataforma Protégé. La decisión de la integración en Protégé viene justificada por el aumento de usabilidad que supone en un entorno de edición ampliamente extendido en la comunidad de la Web Semántica.

El problema de encontrar una representación visual acorde a la complejidad de estos esquemas no es nuevo. Hasta la fecha han aparecido una gran cantidad de visualizadores/editores con representaciones muy distintas y opciones de usuario variables. Con el presente visualizador se aspira a dar una representación más intuitiva que aquellas que utilizan los visualizadores existentes y que además sea capaz de suplir otras carencias más concretas que van desde la ausencia de apoyo sobre un razonador a la omisión de entidades como clases anónimas o la representación de la jerarquía de propiedades en el mismo esquema.

1.2. Objetivo y alcance

El objetivo del presente proyecto es el desarrollo de un visualizador de ontologías que tenga en cuenta la semántica de las mismas para ofrecer una representación visual más comprensible por parte del usuario. De esta manera, el visualizador a desarrollar debe tener las siguientes características:

- Ofrecer un lenguaje visual expresivo y semánticamente correcto que facilite el entendimiento de la ontología.
- Permitir observar la jerarquía de conceptos o clases, así como las clases anónimas que puedan surgir de la definición de éstos.
- Permitir al usuario ocultar parcialmente el grafo, de modo que pueda centrarse en los aspectos que más le interesen.
- Modificar la disposición de los elementos en pantalla para dejarlos según las preferencias personales.
- Ser capaz de guardar el estado visual y la configuración en ficheros XML, para poder restaurarlo en futuras sesiones.
- Exportar como distintos tipos de imagen (JPG y PNG) para poder visualizar posteriormente con un visor de imágenes.
- Añadir la jerarquía de roles y propiedades sobre el grafo.
- Poder listar instancias de las distintas clases.
- Interactuar con los distintos elementos: Desplazar nodos a voluntad, ocultar subníveles de estos, las propiedades asociadas, mostrar información adicional al de la entidad que tiene el foco de atención, realizar zoom sobre el grafo, etc.

Como lenguaje de programación, se utilizará Java, dado que tanto Protégé como la API más extendida (OWLAPI) están desarrollados en dicho lenguaje. Para la realización del plug-in mencionado se utilizará el framework OSGI del que hace uso Protégé para la inclusión de plug-ins. Por otro lado, se hará uso intensivo de razonadores DL[1] explotando en lo posible sus capacidades para ofrecer información relevante para la explicación de la ontología. La selección de éstos vendrá dada por su compatibilidad con OWLAPI siendo Pellet[2] , Hermit[3], Fact++[4] y RacerPro[5] los principales razonadores que la soportan. En el desarrollo se utilizará principalmente

[1]Razonadores `http://owlapi.sourceforge.net/reasoners.html`
[2]Pellet `http://clarkparsia.com/pellet/`
[3]Hermit `http://hermit-reasoner.com/`
[4]Fact++ `http://owl.man.ac.uk/factplusplus/`
[5]RacerPro `http://www.franz.com/agraph/racer/`

Pellet, aunque se pueda utilizar cualquiera de los anteriormente mencionados. Finalmente, para guardar la información, se hará uso de ficheros xml tratados con las librerías DOM y XPATH.

1.3. Contenido de la memoria

El contenido de la memoria contempla los siguientes capítulos:

- El **capítulo 2** tiene como objetivo dar una visión de los conceptos básicos sobre las ontologías además de presentar herramientas que faciliten su manipulación y por tanto su desarrollo y evolución. Dentro del contexto tecnológico se citan además las herramientas utilizadas.
- El **capítulo 3** trata el estado del arte. Se seleccionan diversos visualizadores de ontologías existentes para posteriormente enmarcar la propuesta del proyecto presente y destacar los puntos fuertes de la misma.
- El **capítulo 4** aborda la planificación del proyecto. Se especifican los requisitos, la metodología seguida, los elementos a visualizar y para acabar las distintas fases de diseño por las que ha pasado
- El **capítulo 5** abarca los aspectos más importantes de la implementación, centrándose en el algoritmo de visualización.
- Para finalizar, el **capítulo 6** reune las conclusiones obtenidas al final del proyecto junto con la planificación temporal.

Capítulo 2

Contexto Tecnológico

El objetivo de este capítulo es dar una visión de los conceptos básicos sobre las ontologías además de presentar herramientas que facilitan su manipulación y por tanto su desarrollo y evolución. Aparte se citará también el software que ha sido necesario para la realización del proyecto.

2.1. Conceptos relacionados con el proyecto

En esta sección se hablará sobre los conceptos que van a ser mencionados continuamente a lo largo de toda la memoria y cuya comprensión resulta indispensable para el correcto seguimiento de ella.

2.1.1. Ontologías

Una ontología es la especificación formal y explicita de una conceptualización compartida [12]. En otras palabras, se trata de un modelo de conocimiento consensuado sobre un determinado dominio, con intención de ser revisado y reutilizado en diferentes aplicaciones. Para poder explotar su potencial, una ontología ha de satisfacer el mayor número posible de los siguientes puntos:

- No ser exclusiva de una persona o de una institución sino que sea consensuada o compartida por un grupo de personas, instituciones, etc.
- El modelo a formalizar deber constar de de un conjunto de conceptos organizados jerárquicamente, los cuales se definen a través de un conjunto de propiedades y de sus relaciones con otros conceptos y de un conjunto de axiomas o reglas que permiten inferir conocimientos nuevos a través de los ya existentes.

- Proporcionar términos que tengan una semántica bien definida y se expresen utilizando lenguajes de programación, que a su vez tienen una semántica bien formada.
- Ser formal y por tanto interpretable por un ordenador.

Elementos de una ontología

Las ontologías se componen de los siguientes elementos:

- **Conceptos/Clases**: Son esencialmente conjuntos de elementos o individuos que comparten ciertas propiedades. Normalmente se consideran dos clases que estarán en todo modelo: Top/Thing y Bottom/Nothing.*Thing* es superclase de todas las demás y Bottom es una clase que no puede tener ningún elemento, es por definición subclase de todas las demás y engloba todas las inconsistencias.
- **Roles/Propiedades**: Representan las relaciones de los conceptos. Relacionan siempre dos conceptos con papeles diferenciados. Uno será el dominio de la propiedad y el otro el rango. Un ejemplo de rol o propiedad podría ser *tiene_ consola,* con dominio supuesto de *Persona* y rango *Consola*.
- **Instancias/Individuos**: Representan individuos pertenecientes a un concepto.

Estos tres elementos son los más importantes del modelo. Sin embargo, también disponemos de las **restricciones,** elementos que pueden ser vistos como operadores sobre roles/propiedades.

2.1.2. Description Logics

Los DL son "*lenguajes formales para representar conocimiento y razonamiento sobre él*" [4]. Están formados por una capa intensional llamada TBox y otra ABox.

La *TBox* es la capa de conocimiento intensional y está compuesta por un conjunto de axiomas terminológicos. Son fórmulas del tipo $C \equiv D$ o $C \sqsubseteq D$, donde C y D son conceptos.

La *ABox* es un conjunto de aserciones que describen un estado específico del mundo representado en la *TBox* asociada.

En otras palabras, se diferencia el conocimiento estático sobre los conceptos (TBox) de aquello que se sabe de las instancias (ABox). La descripción de los lenguajes DL es extendida en la sección C.2 de los anexos.

2.2. Lenguajes de representación

En esta sección se va a dar una reseña de los lenguajes que sirven para representar y publicar información semántica. Se puede encontrar una visión en la Figura 1.1 que indica la relación que existe entre los elementos descritos a continuación.

2.2.1. RDF

RDF (*Resource Description Framework*) es un lenguaje para representar y publicar información semántica de recursos en la Web. En primer lugar, estaba enfocado a representar metadatos de cualquier elemento identificable en la red mediante una URI (*Uniform Resource Identifier*) como pueden ser autor, fecha de modificación, etc. Sin embargo, se ha extendido el uso de RDF para representar información sobre cualquier tipo de recurso en la red.

RDF se basa en la idea de la identificación de elementos a través de URIs y de describir recursos en función de propiedades simples y valores. Todo ello contribuye a que sea capaz de representar afirmaciones simples como un grafo de nodos y arco, cuya mínima expresión es una tripleta $a\ R\ b$ (Figura 2.1), por la cual dos elementos quedan relacionados por la propiedad R. De esta manera se permiten formar grafos con información estructurada.

Con RDF se pretende conseguir un vehículo para lograr los objetivos de la Web Semántica. Podría ser considerado su lenguaje ensamblador. Está orientado principalmente para publicar datos de manera interoperable, y en particular, es el lenguaje de representación elegido por la iniciativa Linked Data[6].

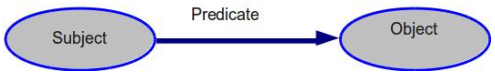


Figura 2.1: Ejemplo de tripleta RDF.

El ejemplo mostrado en la Figura 2.2 ilustra un grafo RDF obtenido a partir de tripletas. Para el intercambio de información RDF es comúnmente visto sobre XML con una representación como la que vemos en la Figura 2.3. A partir de esas líneas podemos extraer lo siguiente:

- #**me** *es_de_tipo* **Person**
- #**me** *tiene_nombre* **Eric Miller**
- #**me** *tiene_correo* **"mailto:em@w3.org"**
- #**me** *tiene_titulo* **"Doctor"**

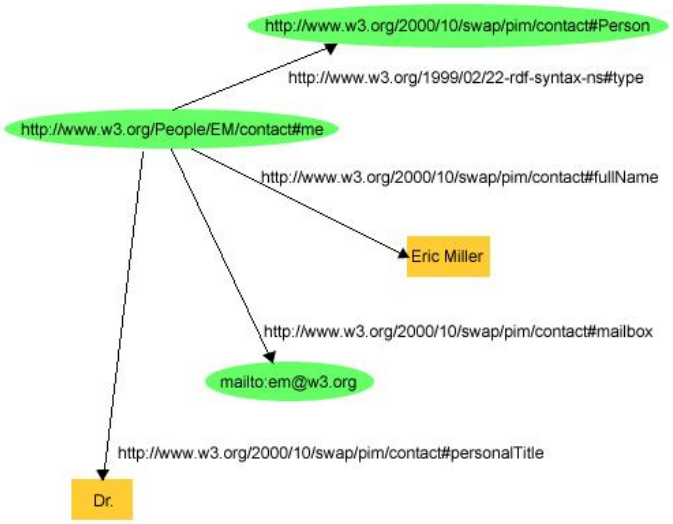


Figura 2.2: Ejemplo de grafo RDF.

```
<?xml version="1.0"?> <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-
    ns#"                xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#">
  <contact:Person rdf:about="http://www.w3.org/People/EM/contact#me">
    <contact:fullName>Eric Miller</contact:fullName>
    <contact:mailbox rdf:resource="mailto:em@w3.org"/>
    <contact:personalTitle>Dr.</contact:personalTitle>
  </contact:Person>
</rdf:RDF>
```

Figura 2.3: Extracto de RDF en XML.

2.2.2. RDFS

RDFS (*RDF Schema*) es un lenguaje de descripción de vocabularios RDF. Construido sobre RDF, puede verse como un sistema de tipos parecido al de los lenguajes orientados a objetos: recursos como ejemplares de una o más clases, jerarquía de clases, propiedades asociadas a clases a través de dominio y rango.

También puede verse como una sintaxis construida sobre RDF con los constructores del Cuadro 2.1

Esta relación RDF con RDFS supone una simplificación haciendo equivalentes casos como el siguiente:

```
<rdf:Description rdf:ID="Computador">
  <rdf:type
    rdf:resource="...../rdf-schema#Class"/>
</rdf:Description>
```

```
<rdfs:Class rdf:ID="Computador"/>
```

Classes	*Properties*	*Utility Properties*
rdfs:Resource	rdfs:domain	rdfs:seeAlso
rdfs:Class	rdfs:range	rdfs:isDefinedBy
rdfs:Literal	rdf:type	
rdfs:Datatype	rdfs:subClassOf	
rdf:XMLLiteral	rdfs:subPropertyOf	
rdf:Property	rdfs:label	
	rdfs:comment	

Cuadro 2.1: Constructores RDFS.

2.2.3. OWL

OWL, lenguaje propuesto por el W3C, se ha convertido en el lenguaje estándar de facto para modelar ontologías. Está basado en DLs y proporciona más expresividad que RDFS para describir propiedades y clases tales como: relaciones entre clases (por ejemplo "clases disjuntas"), cardinalidad (por ejemplo "exactamente uno"), igualdad, más tipos para las propiedades, y clases enumeradas. En la Figura 2.4 tenemos una contrucción OWL en formato RDFS/XML. En ella se esta especificando que la clase *Student* es intersección de *Persona* y *someValuesFrom(takesCourse,Course)* que en lenguaje natural sería "alguien que hace al menos un curso"

```
<owl:Class rdf:ID="Student">
  <rdfs:label>student</rdfs:label>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Person" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#takesCourse" />
      <owl:someValuesFrom>
        <owl:Class rdf:about="#Course" />
      </owl:someValuesFrom>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

Figura 2.4: Ejemplo de construcción OWL.

Hay que tener en cuenta la expresividad del subconjunto escogido de OWL para representar nuestro dominio porque las propiedades computacionales del razonamiento dependen de la misma. Por ello, existía en la primera versión una distinción entre OWL Lite, OWL DL y OWL Full, siendo cada uno de ellos superconjunto del anterior. A raíz del uso exhaustivo por expertos, salieron a la luz nuevas necesidades, como el uso de expresividades reducidas en ciertos entornos o aumento de expresividad en otros. Como respuesta se actualizó a OWL 2 (versión utilizada en este proyecto) que añade nuevas características. Las más relevantes para nuestro proyecto son las siguientes:

1. Nuevas construcciones que añaden expresividad a las propiedades, por ejemplo

la inclusión de cadenas de propiedades (property chain).

2. Soporte de tipos de datos aumentado.

3. Capacidad de metamodelado simple para expresar información metalógica sobre las entidades de una ontología.

En el la sección C.3 del anexo correspondiente a OWL se detallan con más detenimiento los constructores que ofrece OWL.

2.3. Herramientas ontológicas

En esta sección se abordan los razonadores como herramienta que trabajan sobre las ontologías y el API que independiza el uso de razonadores y parseadores de ontologías.

2.3.1. Razonadores DL

Un razonador semántico, motor de reglas, o simplemente un razonador, es una pieza de software capaz de inferir consecuencias lógicas a partir de un conjunto de hechos asertados o axiomas. La noción de razonador semántico generaliza aquella de un motor de inferencia, ya que provee un conjunto más variado de mecanismos con los que trabajar.

Las reglas de inferencia son comunmente especificadas en un lenguaje de ontologías (generalmente OWL). Muchos razonadores utilizan lógica de predicados de primer orden para razonar y esta inferencia suele ser realizada mediante encadenamiento hacia delante o hacia detrás (forward/backward chaining).

Entre los gratuitos que usaremos por el hecho de ser compatibles con OWLAPI tenemos: FaCT[1], FaCT++ [2], Pellet [3], HermiT[4].

Sobre las tareas que puede realizar un razonador podemos destacar:

- Comprobar la consistencia: Mira si existe un modelo lógico que satisfaga todos los axiomas de la ontología.

- Recuperación de instancias: Obtener todos los individuos que pertenecen a una clase.

[1]FaCT - `http://www.cs.man.ac.uk/~horrocks/FaCT/`
[2]FaCT++ -`http://owl.man.ac.uk/factplusplus/`
[3]Pellet - `http://clarkparsia.com/pellet/`
[4]Hermit - `http://hermit-reasoner.com/`

- Satisfacibilidad de conceptos/clases: Comprueba si una clase puede tener instancias. Si no puede, denota necesariamente al conjunto vacío.
- Subsumición: Permite saber si una clase puede ser considerara más general que otra clase ($classA \sqsubseteq classB$?).
- Clasificar: Crea una jerarquía de clases y propiedades basándose en las relaciones de subsumición.

2.3.2. OWLAPI

La API de *OWL* (*OWL API* [11]) es una API de Java y una implementación de referencia para crear, manipular y serializar ontologías *OWL*. La versión utilizada en el proyecto esta pensada para utilizar el lenguaje OWL 2 en su interior. Se trata además de una API de código abierto y está disponible bajo licencias *LGPL* o de *Apache*.

Entre lo que nos ofrece *OWLAPI* podemos destacar los siguientes puntos.

- Escritura y parseado de ficheros en formato *RDF/XML, OWL/XML, OWL Functional Syntax, Turtle.*
- Parseado de ficheros en formato KRSS parser y OBO.
- Interfaces para trabajar con razonadores semánticos como FaCT++, HermiT, Pellet o Racer.

Para hacernos una idea del alcance que tiene, se muestra en la Figura 2.5 un diagrama de clases de un conjunto pequeño pero importante de clases que nos proporciona. Los ejemplos mostrados y la documentación proporcionada en `http://owlapi.sourceforge.net/javadoc/index.html` han sido inestimables para completar el proyecto.

2.4. Software de apoyo

Para el desarrollo del proyecto se ha utilizado el siguiente software de apoyo:

- Eclipse IDE 3.7.0: Entorno de desarrollo
- Java 1.6: Lenguaje de programación del proyecto.
- Lyx: Entorno gráfico para escribir documentos Latex.
- Gimp: Edición de imágenes de la memoria.

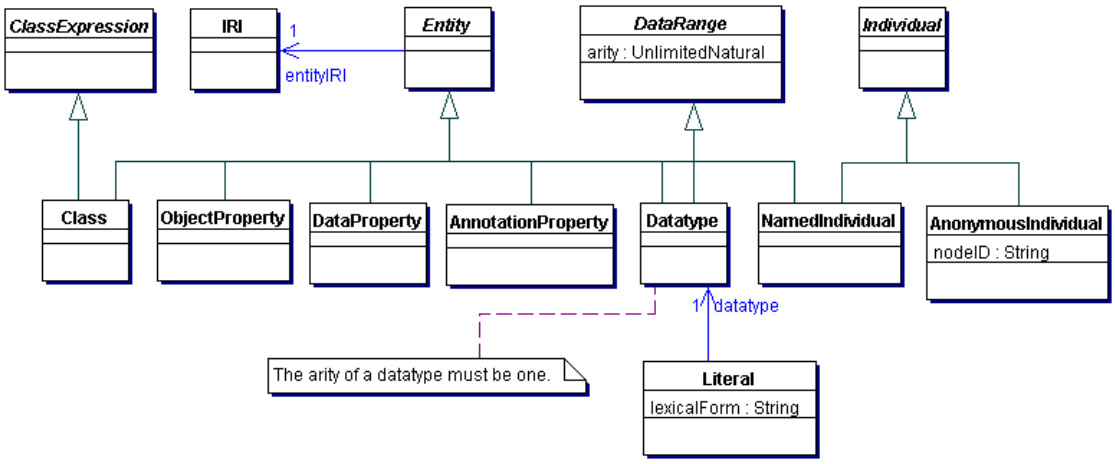


Figura 2.5: Diagrama de clases de un subconjunto de OWLAPI.

- ArgoUml: Diseño de los diagramas de clase.
- OWLAPI : API de interacción con elementos de OWL.
- Pellet: Razonador semántico.
- Protégé 4.1: Editor de ontologías.
- Pedviz.jar: Librería que provee una implementación del algoritmo de Sugiyama[15] de ordenación de grafos planares.

Capítulo 3

Estado del arte

En este capítulo se van a estudiar distintas soluciones existentes al problema de la visualización de ontologías para posteriormente enmarcar la propuesta del proyecto presente y destacar los puntos fuertes de la misma.

3.1. Herramientas de visualización

Vamos a resumir la información obtenida de distintas fuentes (especificaciones del autor,opiniones de terceros, experiencia propia...) de un conjunto de visualizadores de referencia.

3.1.1. GrOwl

Este visualizador posee tanto plug-in como aplicación aparte. De su uso podemos extraer las siguientes características y conclusiones:

Comentarios positivos:

- Tiene opciones de edición.
- Permite varios filtros para visualizar parcialmente.
- Representa las clases como rectángulos y las propiedades como elipsoides que apuntan a su dominio.

Comentarios negativos:

- No utiliza el razonador para completar el esquema.

- Las restricciones aparecen como roles asociados a una clase en vez de ser realmente clases (se ve en la Figura 3.1).
- No representan la jerarquía de roles
- No tienen representación para las restricciones sin cualificar.

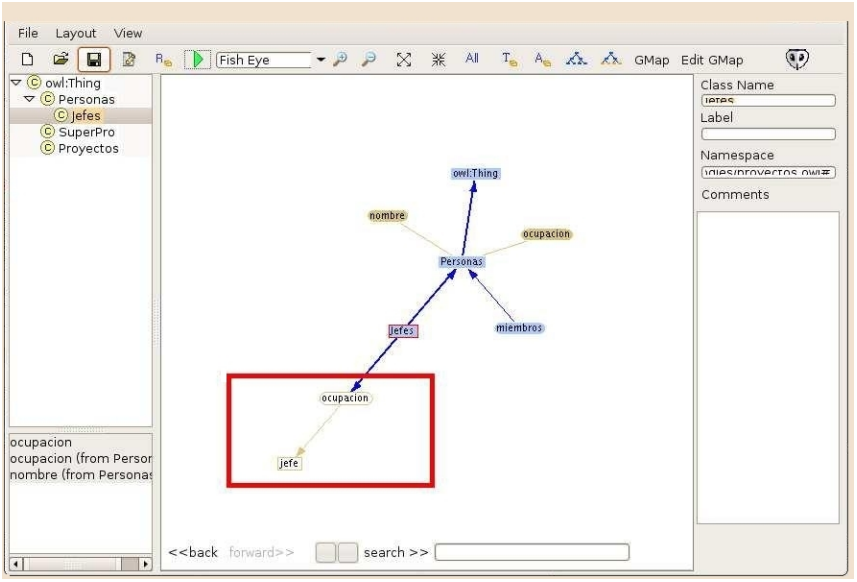


Figura 3.1: Vista de ontología del visualizador GrOwl.

3.1.2. OWLViz

Esta herramienta viene incluída como plug-in de Protégé. Necesita que esté instalado el paquete GraphViz para su funcionamiento. Después de usarlo con unas cuantas ontologías se puede resaltar distintos aspectos.

En cuanto a lo positivo podemos destacar:

- Permite mostrar ramas del grafo (sin necesidad de ver el camino entero desde Thing)
- Representa la jerarquía de clases con flechas 'is-a' que apuntan a la superclase.
- En cuanto a la disposición del grafo (*layout*), es estático y se muestra como en la Figura 3.2 (permite cambiarlo a vertical/horizontal).
- Se apoya en el razonador para separar conocimiento asertado de inferido.

En cuanto a los aspectos negativos podemos observar

- La posición de las formas no se puede modificar.
- La interacción con ellas se reduce a mostrar u ocultar.
- No muestra información de propiedades ni de clases anónimas ni de instancias.

En resumen, éste se parece mucho a la idea que se propone en el proyecto. Sin embargo, pese a mostrar bien la información que muestra, resulta escasa.

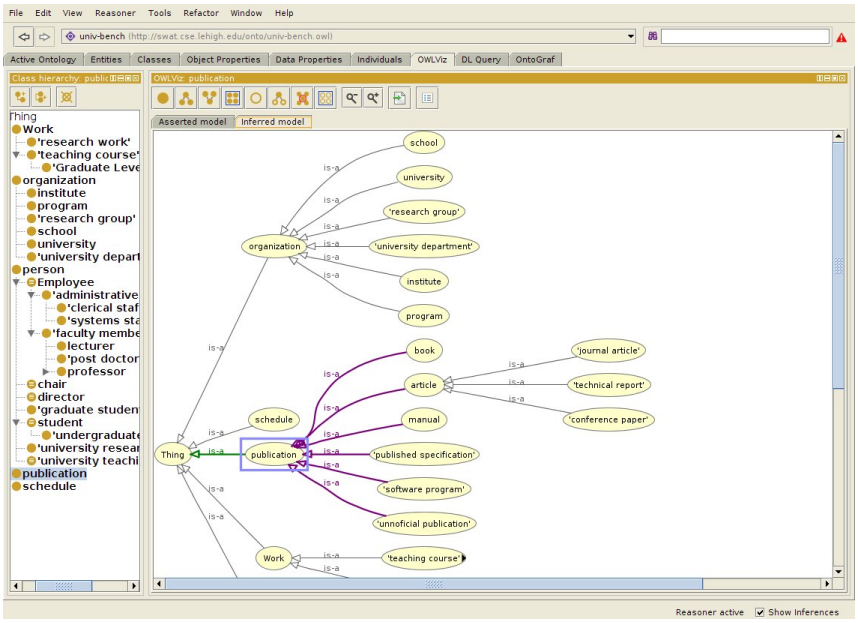


Figura 3.2: Captura de OWLViz.

3.1.3. OntVis

Teniendo en cuenta que la recopilación de herramientas encontradas en el enlace `http://techwiki.openstructs.org/index.php/Ontology_Tools` podría estar desactualizada, se ha buscado más exhaustivamente y se ha dado con un visualizador no perteneciente a dicha lista que además parece más nuevo y avanzado. Sin embargo se procede a analizarlo y logramos extraer las siguientes conclusiones separadas en pros y contras.

Pros:

- Se trata de una aplicación web (Flash) en una comunidad.
- Permite guardar estado visual para luego restaurarlo.
- Permite exportar a formato de imagen.

- Es visualmente agradable si se logra encontrar una disposición adecuada.

Contras:

- Se expande el grafo de forma manual a partir de una entidad seleccionada también manualmente. No parece haber forma de visualizar la ontología completa si no se ha desplegado manualmente antes.
- El layout del grafo se realiza de forma manual. Da más libertad que el visualizador del proyecto, pero delega en el usuario la responsabilidad de encontrar una disposición satisfactoria de los elementos del grafo.
- No incluye como elementos a las clases anónimas.
- Sobre las propiedades se muestra cual es su dominio al ser mostradas junto a él, pero es imposible ver en la misma vista la jerarquía de propiedades o el rango de estas.
- No tenemos evidencia de que se apoye en el uso de un razonador para encontrar implicaciones o comprobar la consistencia.
- La accesibilidad a la vista del visualizador es a través de una pestaña difícil de ver.

Como referencia se puede observar una imagen de un grafo parcial en la Figura 3.3

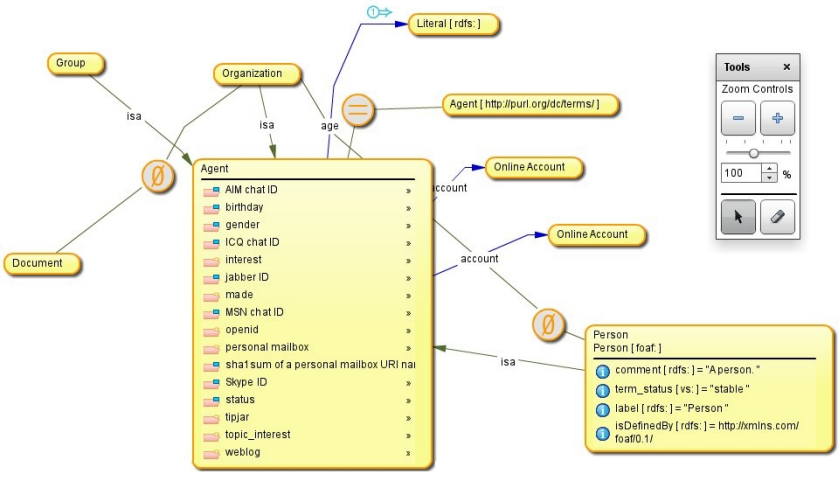


Figura 3.3: Captura de ontVis.

3.1.4. Ontograf

En Ontograf (Figura 3.4) encontramos otro visualizador popular y que viene incluido como plug-in por defecto en Protégé. Después de probarlo podemos destacar los siguientes aspectos:

- Añade instancias como nodos identificados con un rombo morado.
- Distingue clases definidas del resto y muestra la definición en un tooltip.
- Permite exportar como imágen.

Sin embargo, también tiene carencias:

- En la primera visualización de una ontología hay que extenderla a mano, lo cual puede ser laborioso si es suficientemente grande. Por lo menos tiene soporte para guardar el estado y restaurarlo en una sesión posterior.
- La disposición del grafo queda a responsabilidad del usuario. Si bien los nodos se desplazan tras abrirlos o cerrarlos, el resultado no es completamente satisfactorio.
- Las propiedades vienen reflejadas como arcos con un color propio. Es necesario ir a mirar la leyenda o esperar al tooltip del arco.
- No añade clases anónimas como elementos visuales.
- No hace uso de un razonador en ningún momento.
- De las propiedades sólo obtenemos el dominio y el rango. No se dice nada del tipo de propiedad ni de su relación con otras propiedades.

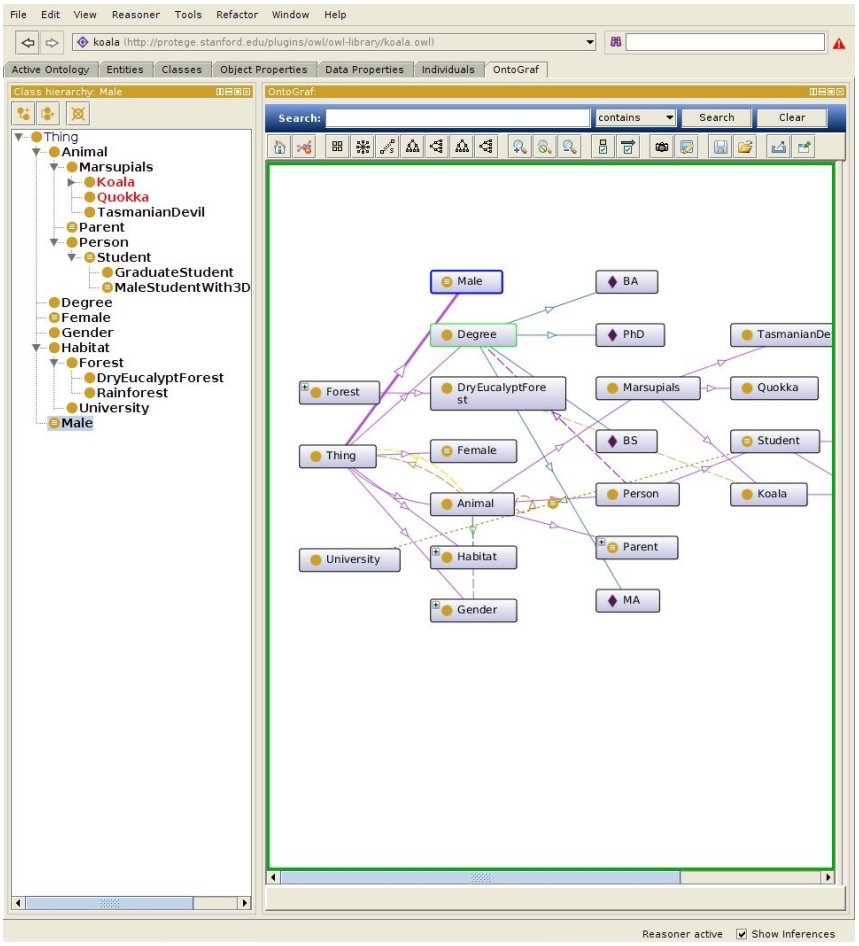


Figura 3.4: Captura de Ontograf.

3.1.5. Resumen comparativo

En el Cuadro 3.1 vamos a agrupar en distintos criterios las aproximaciones tomadas por los visualizadores previamente analizados para compararlo con el nuestro:

	GrOWL	OWLViz	OntVis	OntoGraf	Propuesto
Visualización inicial	La sección seleccionada en el panel lateral.	Vista completa desde Thing con sentido de izquierda a derecha (como propuesto).	Nodo seleccionado.	Un nodo elegido para expandir manualmente.	Vista completa con distribución de Sugiyama.
Tratamiento de propiedades	✓ Sin jerarquía.	✗	✓	✓ Mediante códigos de colores. Sin jerarquía de propiedades.	✓ Información de dominio, rango, tipo de propiedad y jerarquía de propiedades.
Uso de razonador	✗	✓ Diferencia grafo asertado de inferido.	✗	✗	✓
Clases anónimas	✗	✗	✗	✓ Muestra las definiciones pero no como elementos visuales.	✓ Representadas como elementos visuales y expandidas en subexpresiones atómicas.
Permite desplazar nodos	✓	✗	✓	✓	✓ En el mismo nivel.
Visualización parcial	✓	✓	✓	✓	✓
Guardar estado	✗	✗	✓	✓	✓
Exportar imágen	✗	✓	✓	✓	✓
Edición	✓	✗	✗	✗	✗

Cuadro 3.1: Resumen de características de los visualizadores.

Capítulo 4

Planificación del visualizador

En este capítulo se abordarán los aspectos más importantes de la planificación del proyecto. Empezando por una lista de requisitos, la metodología utilizada y las distintas fases del diseño.

4.1. Análisis de requisitos

4.1.1. Requisitos

En la Tabla 4.1 se muestra el estado final de la especificación de requisitos del visualizador. Se especifica como final ya que ha habido una evolución en esta lista. Comenzando desde el objetivo principal, creación de un visualizador de ontologías, los requisitos han sido añadidos, modificados e incluso eliminados en distintas etapas del proyecto. Sin embargo, es algo que se ha tenido en cuenta desde un principio, de modo que el enfoque ha sido el de implementar el sistema de forma que un cambio en un requisito no suponga desechar todo el trabajo realizado. En los requisitos se pueden distinguir aquellos que incentivan la utilización del visualizador a partir de características no vitales (aunque importantes) de los que hacen hincapié en representar el modelo de OWL (códigos de la tabla de requisitos 1 o 6 por poner ejemplos).

4.1.2. Metodología seguida

La metodología que ha guiado el desarrollo de este PFC se ha basado en un modelo Incremental-Evolutivo. En éste se combinan características de ambos métodos, como la variación en la especificación de requisitos o el incremento de funcionalidades adicionales al sistema. Sobre una especificación inicial abstracta se ha ido añadiendo nuevas características de forma incremental con la previsión de la existencia de

Código	Descripción
1	Permitir observar la jerarquía de conceptos o clases, así como las clases anónimas que puedan surgir de la definición de éstos.
2	Permitir al usuario ocultar parcialmente el grafo, de modo que pueda centrarse en los aspectos que más le interesen.
2.1	Se debe poder cerrar/abrir una clase (ocultar subniveles en el estado en el que estén).
2.2	Se debe poder ocultar una clase.
2.3	Se debe poder ocultar y mostrar conectores de disjunción.
2.4	Se debe poder ocultar y mostrar conectores de rango de propiedades.
2.5	Se debe poder ocultar las propiedades asociadas a una clase dada.
2.6	Se debe poder ocultar toda la información referente a las propiedades en un sólo paso.
3	Modificar la disposición de los elementos en pantalla para dejarlos según las preferencias personales.
3.1	Debe permitir desplazar en el mismo nivel las clases y restricciones existentes.
4	Guardar el estado visual en XML.
5	Exportar el estado visual en formato de imagen (jpg,png).
6	Añadir la jerarquía de roles/propiedades sobre el grafo.
7	Listar instancias de las distintas clases.
7.1	Se listarán las instancias en una ventana aparte.
8	Interactuar con los distintos elementos.
9	Se debe poder realizar zoom sobre el grafo.
10	Se desarrollará como plug-in de Protégé y como aplicación independiente.

Cuadro 4.1: Requisitos del sistema.

cambios. En cada incremento ha sido necesario feedback por parte de usuarios de las distintas versiones.

A destacar del modelo evolutivo se pueden enumerar los siguientes puntos que lo definen:

- Los requerimientos no son completamente conocidos al inicio.
- El software evoluciona con el tiempo.
- Los requerimientos son cuidadosamente examinados, y sólo esos que son bien comprendidos son seleccionados para un primer incremento.
- La implementación parcial es usada y testeada por los usuarios y proveen feedback a los desarrolladores.
- En función de esta retroalimentación los requisitos evolucionan con cambios.

En definitiva, lo que se ha buscado ha sido encontrar una metodología entre las existentes que satisfaciera el requisito de agilidad y adaptabilidad. La mezcla de ambas ha sido el resultado de dicha búsqueda.

4.2. Elementos a visualizar

En esta sección se van a describir los elementos visuales del grafo resultante. Empezando por las clases y restricciones (los elementos más visibles), continuando con las propiedades, y finalizando con las aristas.

Clases (conceptos)

Representan las clases del modelo de OWL. En el lenguaje visual definido podremos distinguir entre clases con nombre, anónimas y clases definidas. Visualmente se corresponden con rectángulos y dependiendo del tipo de clase variarán las características visuales de los mismos.

Clases con nombre

Son las que aparecen como la Figura 4.1. Se corresponden con las *NamedClasses* no definidas de OWLAPI y desde el punto de vista lógico, son aquellas que dan condiciones suficientes para que una instancia pertenezca a ellas. Visualmente se caracterizan por tener un nombre simple (sin prefijos de URIS), un fondo grisáceo y fuente plana.

Figura 4.1: Clase con nombre.

Clases definidas

Son clases con nombre con la particularidad de que han sido definidas a partir de una expresión. En otras palabras, se han dado condiciones necesarias y suficientes de pertenencia (ver Figura 4.2). Visualmente son idénticas a las clases con nombre salvo por la fuente, que es cursiva precisamente para diferenciar este caso. Su definición viene asociada como clase anónima (Figura 4.3).

Figura 4.2: Clase definida.

Clases anónimas

Clases que no tienen nombre en el esquema. Se corresponden con expresiones que representan clases y son referenciadas por el razonador para clasificar correctamente los conceptos de la T-Box. Un ejemplo se tiene en la Figura 4.3. El fondo del rectángulo es blanco en contraposición de los anteriores tipos con la intención de resaltar la mayor importancia que poseen aquellas con nombre.

Figura 4.3: Clase anónima.

Restricciones

Son constructures que nos permiten definir clases mediante la imposición de una restricción sobre una propiedad. En el grafo vienen representadas como círculos (Figura 4.4) asociados a una expresión formal. Surgen de la descomposición de las clases anónimas que han aparecido en el grafo en un primer lugar. Completan el grafo, pero a la vez aumentan el número de elementos visibles.

El método de renderizado de la etiqueta correspondiente viene determinado por el tipo de expresión asociada. Por ejemplo, en la Figura 4.4, tenemos dos casos: uno genérico que mantiene la etiqueta encima, y el caso de intersección (que es mostrado con su símbolo dentro del círculo).

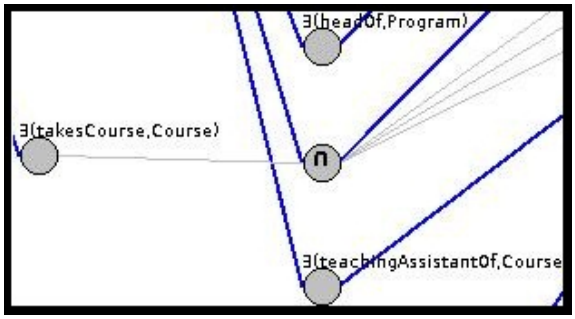


Figura 4.4: Restricciones.

Además, en dicha figura se aprecia la composición de restricciones para formar expresiones de mayor nivel y podemos observar una intersección de la restricción existencial de *takesCourse* con otra clase (o expresión).

Símbolo	Explicación
$\cap$	Intersección
$\cup$	Unión
$\exists$	Some values
$\forall$	All values from
$\leq$	Cuantificador
$\neg$	Complemento

Cuadro 4.2: Símbolos utilizados en el renderizado de restricciones.

Propiedades

Son elementos subordinados a la existencia de clases, ya que describen comportamientos o relaciones característicos de los individuos pertenecientes a una clase. Para su completa definición precisa de dos elementos, el *dominio* y el *rango*. Si no se especifican explícitamente, se entiende que serán *Thing*. En la representación visual del grafo, una propiedad es vista como una cadena en una "caja" invisible presente debajo del dominio de dicha propiedad. En la Figura 4.5 podemos observar

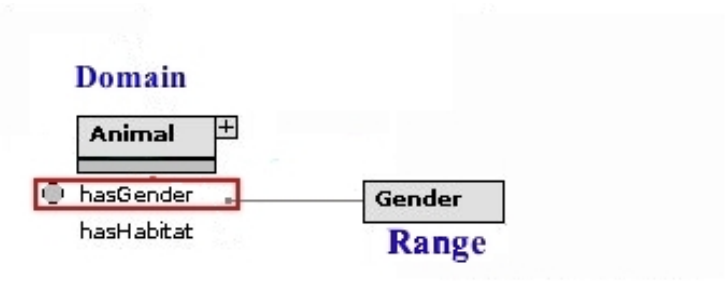


Figura 4.5: Propiedades.

los distintos elementos que se visualizan en el grafo por añadir las propiedades.

En primer lugar, de la propiedad *hasGender* se puede decir que por el hecho de estar debajo de la clase *Animal* será una propiedad que se aplique sobre animales (su dominio); en segundo lugar, que el rango no es la propia clase sino que es *Gender*. Con esto concluimos que *hasGender* relaciona individuos de *Animal* con individuos de *Gender*.

Además de los elementos principales comentados, se aprecia también un círculo gris junto a la propiedad. Esto indica que la descripción de la propiedad es más extensa que la definición de dominio y rango. Esto es debido a que las propiedades o

roles pueden tener una caracterización especial, por ejemplo pueden ser transitivas, funcionales, simétricas, etc. Para visualizar la descripción sólo hay que mirar el tooltip asociado al círculo gris.

Por otra parte, las propiedades de datos (*OWLDatatypeProperty*) comparten representación, aunque con la salvedad que el rango no es un elemento de una clase sino un literal de un tipo de datos de XMLSchema. El tipo de dato asociado a la propiedad es puesto junto al nombre de ésta junto a unos ':' de separación.

Conectores/aristas

En el grafo se pueden ver distintas clases de conectores con distintas características que desempeñan distintas funciones.

- **conector is-a**: van de concepto a concepto. Indican relación jerárquica.
- **conector de equivalencia**: conjunto de 3 líneas que unen para indicar la relación de equivalencia. Se evita añadir en la definición de una clase a partir de una expresión anónima. Además se evita la conexión todos con todos en un conjunto de clases equivalentes, dejándolo a la mínima expresión sin perder significado.
- **conector disjunto**: como el anterior, salvo que se trata de un conjunto de 2 líneas que expresan que las clases son disjuntas.
- **conector a rayas (dashed)**: es un conector is-a especial que aparece para indicar que hay una relación indirecta. Si una clase A es superclase de la clase B y ésta a su vez es superclase de otra llamada C, entendemos que existe una relación indirecta entre A y C.
- **conector de herencia**: de propiedad a propiedad con una flecha apuntando a la propiedad padre.
- **conector de rango**: de propiedad a concepto. Apunta al rango de la propiedad.
- **conector de herencia múltiple**: En el caso de que una propiedad sea subpropiedad de más de una propiedad.

En la Figura 4.6 tenemos los conectores más comunes, y aparte, en la Figura 4.7 aquellos relacionados con la herencia en propiedades.

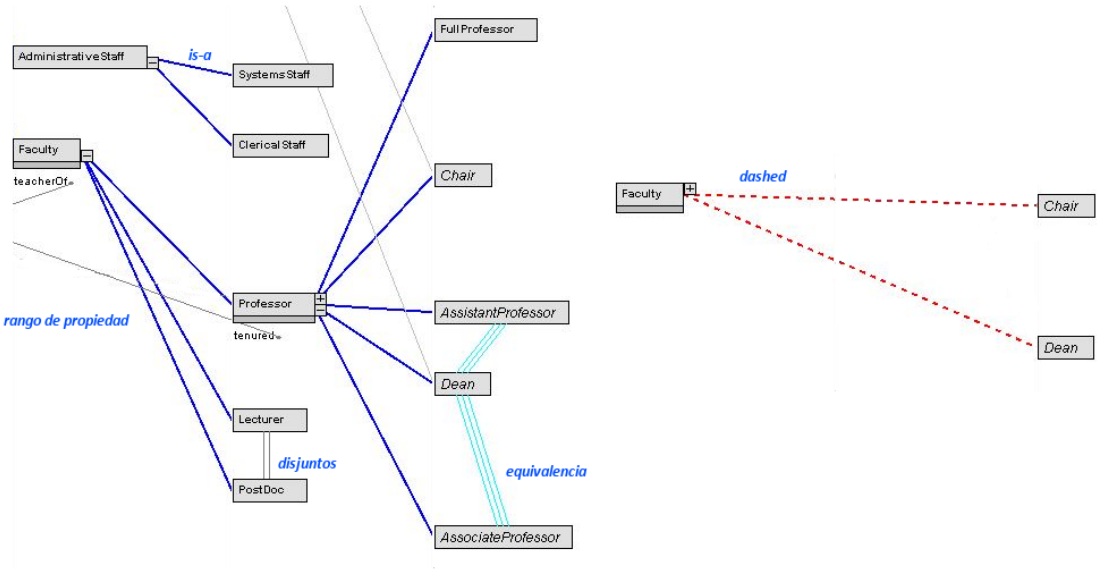


Figura 4.6: Tipos de conectores en el grafo.

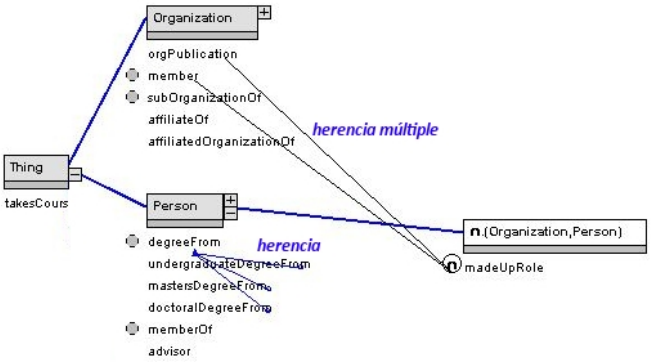


Figura 4.7: Conectores de herencia en propiedades.

4.3. Diseño propuesto

En esta sección se va a presentar la evolución del aspecto gráfico del visualizador a lo largo del proyecto. Se empezará por un primer prototipo que se centra en la representación de las clases y a partir de ahí surgirán nuevas propuestas que se irán adaptando a los requisitos que se van añadiendo. Esta sección abarca tanto el plug-in propuesto como la versión *standalone* (Cuadro 4.1, código 10) dado que tienen mucho en común.

Primer prototipo

La primera aproximación al diseño que iba a tener el grafo consistía en un grafo DAG (*directed acyclic graph*) de cajas con los nombres de las clases en horizontal. Se puede observar un boceto de la idea abstracta en la Figura 4.8 . Poner el grafo en

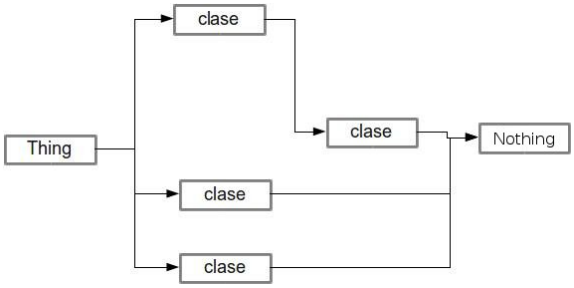


Figura 4.8: Primer boceto de diseño.

horizontal permite mostrar más información en el marco de la pantalla (la longitud de los nombres de las clases hacen que los rectángulos que las contienen sean mas anchos que altos).

La jerarquía se representa siempre de izquierda a derecha con conectores *is-a* y se decide bloquear la posición horizontal para alinear clases del mismo nivel. En este primer prototipo no se tienen en cuenta todavía las propiedades ni las instancias, dado que en esta fase aún se desconocen los riesgos que puedan aparecer en este primer objetivo. Otro punto que se tiene claro, es la existencia de un panel de control para ir añadiendo funcionalidades. Sin embargo, en esta primera fase, no se especifica aún que va a tener ese panel de control.

Expansión y panel de control

En la Figura 4.9 tenemos un prototipo más avanzado. Hemos incluido las restricciones, un panel de control funcional que permite cargar ontologías y razonadores (necesario para la aplicación propia, pero omitido en el plug-in de Protégé por estar incluído). A partir de este punto el panel cambia poco. Se maquetará ligeramente, se añadirá algún icono y más opciones.

Figura 4.9: Segundo prototipo de diseño (panel de control).

Propiedades

En la Figura 4.10 ya podemos ver como vamos a incluir las propiedades en el esquema. Decidimos mostrar las propiedades y sus características evitando añadir mucho ruido en el grafo. Dado que cada propiedad ha de referenciar dos clases (Dominio, Rango), situaremos la propiedad debajo de su dominio y el rango se indicará con una flecha hacia la clase. La herencia de propiedades se indicará con una flecha que vaya de subpropiedad a propiedad.

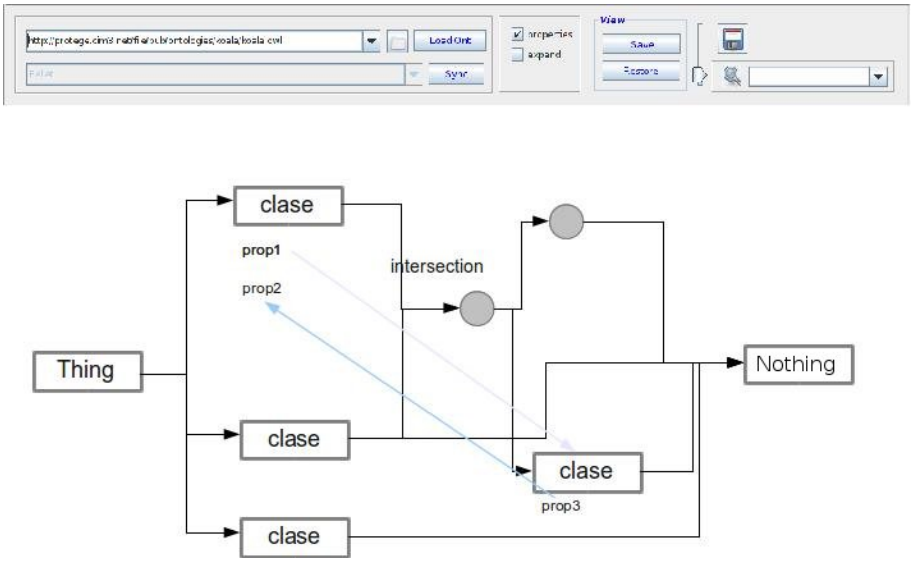


Figura 4.10: Prototipo con propiedades.

Equivalencia y disjunción

En cuanto a representación visual de las relaciones de equivalencia y disjunción de clases, se representará como en la Figura 4.11.

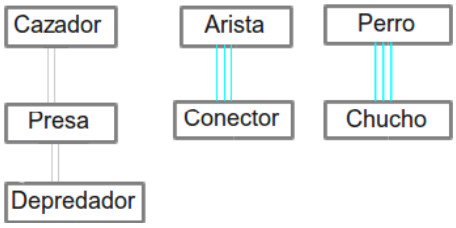


Figura 4.11: Clases disjuntas y clases equivalentes.

Estado final

El estado final del grado se aprecia en la Figura 4.12.

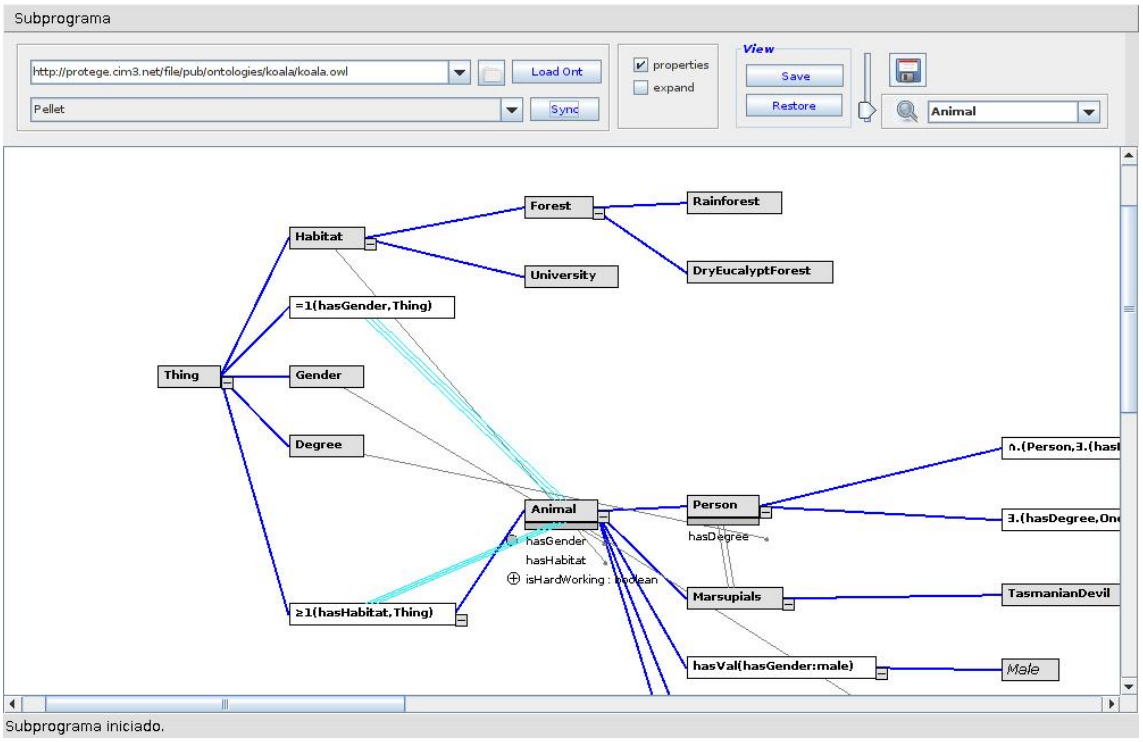


Figura 4.12: Resultado visual final.

Capítulo 5

Implementación del visualizador

En este capítulo se abordarán los puntos más importantes de la implementación, dejando a la parte de anexos aquellos que se consideren menos relevantes.

5.1. Algoritmo de visualización

A la hora de representar la ontología, el paso más importante es la creación del grafo a visualizar. Para ello utilizamos el razonador para diversas tareas como pedir clases relacionadas (subclases, superclases, clases equivalentes), buscar dominios, etc. En el algoritmo, asumimos la ontología y el razonador cargados.

En el Algoritmo 5.1 se muestra el algoritmo en alto nivel, el cual incluye creación y la distribución de los nodos en el espacio. El algoritmo se detalla a lo largo de esta sección.

5.1.1. Obteniendo la taxonomía

Para crear nuestra estructura que refleje las clases que hay en la ontología y las relaciones existentes entre ellas necesitaremos una estrategia. Esta estrategia consiste en primer lugar en buscar a partir de las clases que van estar en toda ontología (*Thing* y clases equivalentes a *Thing).*

A partir de ahí tendremos que realizar un recorrido que obtenga la jerarquía de clases, tanto anónimas como con nombre. El recorrido es realizado en todas las direcciones, es decir, hacia las subclases, superclases y equivalentes (*en expansión*). Así pues ayudado del razonador en conjunción de OWLAPI que permite extraer información explícita de la ontología cargada, podemos obtener todas las expresiones que hay en el esquema.

Algoritmo 5.1 Creación del grafo.

```
AccionCrear(){
  Nodo<OWLClass> nodo = razonador.obtenerNodoTop()
  Set<OWLClassExpression>  set = nodeToSet(nodo)
  crearGrafo(set,expandido) //expandido es un booleano obtenido del panel de
      control
}
```

```
crearGrafo(){
  recorrido(set)
  tratarClasesDefinidas()
  ReorganizarGrafo()
  ReajustarPosicionYAnchura()
  Si (expandido){
    expandirGrafo()
    ReorganizarGrafo()
    EncogerGrafo()
  }

  ReajustarPosicionYAnchura()
  AñadirConectoresEquivalencia()
  AñadirConectoresDisjunción()
  AñadirPropiedades()
  EliminarConectoresRedundantes()
  ReordenacionGrafo(Sugiyama)
}
```

La expansión en todas las direcciones obliga a comprobar que la clase actual no ha sido añadida. Para optimizar esta búsqueda, se utiliza una tabla hash como índice para optimizar la búsqueda de expresiones ya añadidas (para no repetir nodos).

Cada nodo que se añada al grafo tendrá asociado un nivel que se corresponderá con una posición horizontal sobre la que se alinearán sus elementos. Los niveles y los nodos asociados a ellos serán fijos una vez se haya finalizado el algoritmo de visualización (puede variar en los procesos intermedios que se explican en la sección). En alto nivel el algoritmo sería como se indica en el Algoritmo 5.2

5.1.2. Añadiendo clases definidas

Un ejemplo de clase definida viene ilustrado por:

$$Student \equiv Person \cap (Some(takesCourse, Course))$$

Este es un caso específico que necesita ser tratado a parte por varios motivos:

- Se puede dar el caso en que se forme un ciclo al darse que la definición de igualdad venga dada por $(Definición \sqsubseteq A) \bigwedge (A \sqsubseteq Definición)$, provocando que la reordenación posterior acabe por número máximo de iteraciones en vez de por conseguir el objetivo.

Algoritmo 5.2 Recorrido de clases.

```
procedimiento recorrido(set_expresiones){ //set_expresiones ha sido iniciado con
    las clases del nodo top
    para cada (expresion: set_expresiones){
        si (buscar_shape(expresion) == null){
            nuevaClaseVisual = añadirClaseVisual(expresion)
            set_subclases <- owlapi.subclases(expresion)
            set_subclases <- razonador.subclases(expresion)
            //igual con superclases y clases equivalentes

            recorrido(set_subclases)
            recorrido(set_superclases)
            recorrido(set_equivalentes)

            añadirConectores(nuevaClaseVisual)
        }
    }
}
```

- Se quiere mantener visualmente las definiciones siempre en un nivel superior a la clase que define como en Figura 5.1

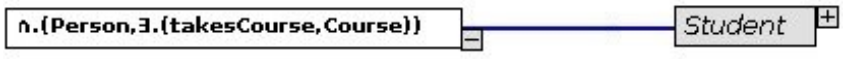


Figura 5.1: Definición (clase anónima) junto a clase definida.

Por ello, estas instancias concretas de conector se añadirán después del recorrido inicial.

5.1.3. Reorganizar el grafo

El recorrido en expansión hasta ahora permite que haya aristas salientes de una clase a otra clase que esta en el mismo nivel o anterior. Esta es una característica que se quiere evitar para mantener una dirección del grafo, con motivo de mejorar la legibilidad. Si encuentra alguna situación en la que el sentido de la arista no sea el deseado, colocara la clase destino en el nivel siguiente para corregirlo (ver Figura 5.3).

La reordenación es satisfactoria en caso de que no haya ciclos, pero se trata de un grafo dirigido y no existe problema en ese sentido. Sin embargo,se ha mostrado en la subsección de clases definidas que pueden aparecer ciclos evitables.

Figura 5.2: Reordenando clases.

Algoritmo 5.3 Código reorganizando el grafo.

```
procedimiento reorganizarGrafo(){
  completado = false
  Mientras_que (not completado){
    completado = true
    Para cada (Figura: set_de_shapes){
      Para cada (conector_saliente: Figura.conectores_salientes){
        Figura2 = conector_saliente.Figura_en_extremo
        si (nivel(Figura2) <= nivel(Figura)){
          nivel(Figura2) = nivel(Figura)+1
          completado = false
        }
      }
    }
  }
}
```

5.1.4. Ajuste de posición y anchura

Mover clases de un nivel a otro como se ha hecho en el paso anterior deja al grafo construido, pero con información visual desactualizada. En la situación actual podríamos tener dos niveles distintos con la misma coordenada X o tener un nivel con una anchura mucho menor que la forma más ancha del nivel. Este proceso consta de dos pasos:

1. Actualizar las anchuras de los niveles en función de la forma de mayor anchura que contengan.

2. A partir del primer nivel, actualizar la posición de los niveles y de las formas que hay en ellos.

5.1.5. Expansión del grafo : Añadiendo restricciones

La idea visual de expandir un nodo de clase anónima se ilustra en la Figura 5.3. Las clases anónimas que aparecen en el esquema pueden llegar a ser expresiones complejas construidas a partir de más expresiones.

El proceso de expansión sustituye la clase anónima por su descomposición en átomos lógicos (restricciones).

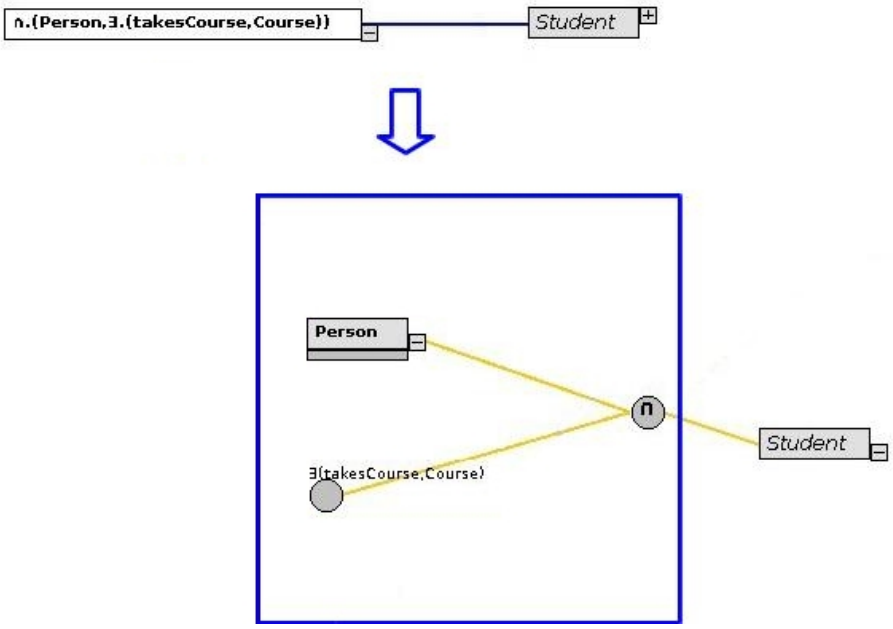


Figura 5.3: Expansión de nodo.

Añadiendo Restricciones

Como se observa en la Figura 5.3 el conjunto de expresiones simples que dan lugar a la clase expandida, ocupan más niveles. En otras palabras, el grafo crece en profundidad a la vez que se expanden nodos. La solución más directa pasa por añadir un nuevo nivel en medio (actualizando aquellos que tuviesen un id mayor o igual al nivel introducido) y añadir en el la restricción. Sin embargo, esta solución no es gratis, ya que permite que de nuevo vuelvan a aparecer situaciones como las que queríamos evitar en (ver Figura 5.2). Tiene fácil solución si aplicamos el mismo procedimiento.

Además se crea otra nueva situación: en grafos con expresiones anónimas muy largas se introducen demasiados niveles semivacíos que extienden horizontalmente el grafo más de lo deseable. En este punto nos falta comprimir al máximo el grafo. Para ello, los niveles con sólo restricciones son plegados. Con plegar nos referimos a traer al nivel actual todas las restricciones que estén en niveles superiores y que estén libres de dependencias. Se considerará libre de dependencias si traer el elemento mantiene la restricción de sentido (izquierda a derecha). Después de este realojamiento es muy probable que queden niveles completamente vacíos, así que conviene eliminarlos.

En resumen: primero, se realojan las restricciones al nivel más bajo posible, y segundo, se eliminan los niveles vacíos resultante.

5.1.6. Distribución de los nodos en un plano 2D (Algoritmo de Sugiyama)

El último paso del algoritmo de visualización consiste en disponer los nodos en el plano 2D de forma que se minimicen los cruces entre aristas y la distancia entre un nodo padre y su hijo. De esta manera, se presentará al usuario una primera disposición que resulte agradable por la minimización de cruces. Para este problema se ha utilizado la solución de Sugiyama [15] (Anexo E) con el paquete *pedviz.jar*[1] que la implementa. En este punto se ha ajustado con los siguiente pasos:

1. Eliminar conectores is-a redundantes (para que el algoritmo de Sugiyama sea eficiente).
2. Clonar la estructura del grafo (los nodos de clases) a un grafo del paquete pedviz. En la Figura 5.4 se observa el grafo clonado.

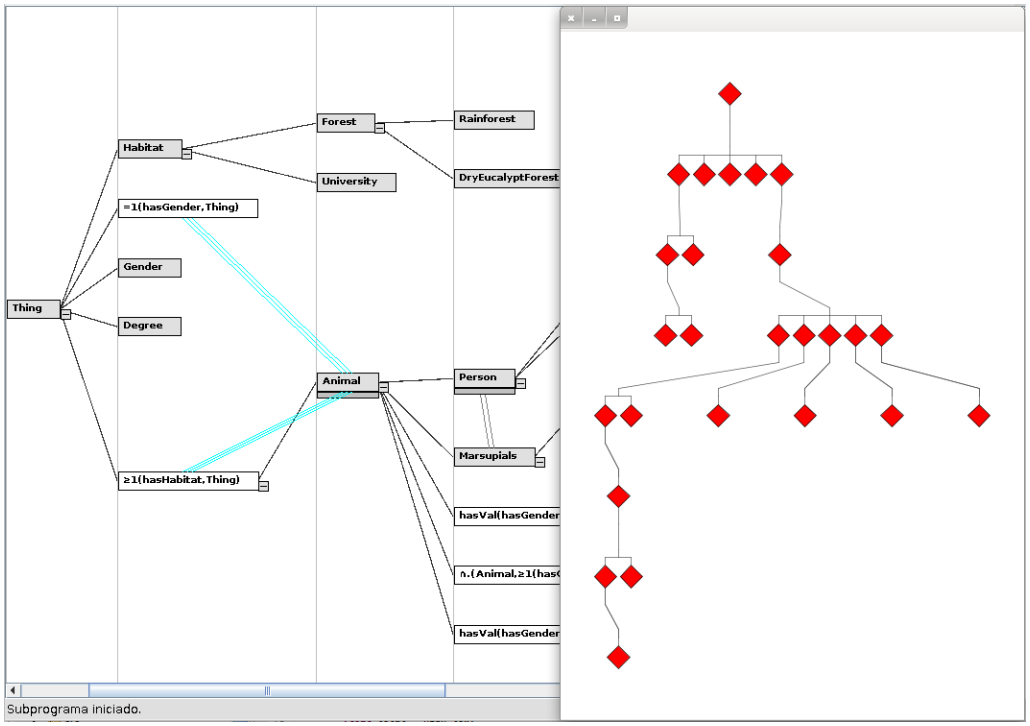


Figura 5.4: Grafo con layout Sugiyama.

3. Aplicar su implementación de Sugiyama al grafo clonado.
4. Acceder a la información visual del grafo ordenado.
5. Traducir esa información absoluta a una relativa (al tamaño del panel) para usarla en el grafo principal.

[1] pedviz - `http://www.pedvizapi.org/`

5.2. Características generales

5.2.1. Diferencias Plugin-Applet

En la realización de las dos versiones del visualizador han surgido diferencias debidas la propia API de Protégé que ofrece para el desarrollo de plug-ins. En definitiva, el objetivo ha sido intentar independizar al máximo el visualizador de los servicios de Protégé (Figura 5.5) y hacerlo dependiente del factor común: *OWLAPI*. Estos servicios en la práctica se concretan en la carga de la ontología y del razonador y para ello se han abstraído en ambos casos en métodos propios (*loadOntology y loadReasoner*).

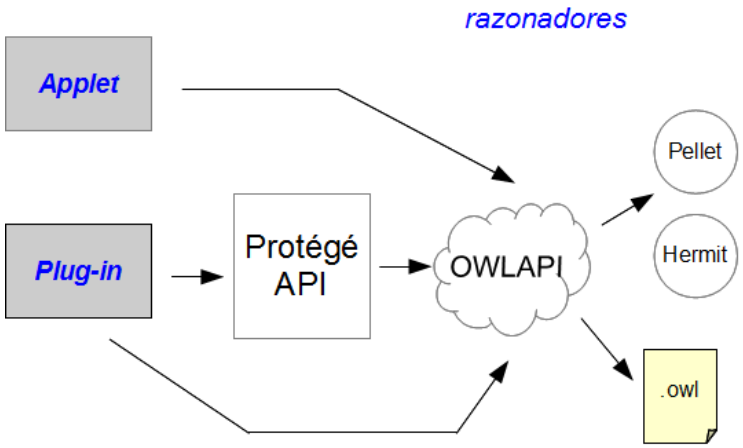


Figura 5.5: Diferencias plug-in y applet.

5.2.2. Instancias

Las instancias o individuos pertenecientes a una clase son representados como una lista de nombres en una ventana aparte (Figura 5.6). No se añaden como elementos visuales para no añadir ruido a la visualización swl esquema de la ontología.

5.2.3. Características configurables

Con el objetivo de dejar configurables características tales como el grosor de los conectores o los colores, se ha optado por un módulo de configuración programado con el patrón *Singleton* como referencia. En la actualidad soporta el la configuración del color y grosor de los conectores principales, pero se ha pensado en la proyección

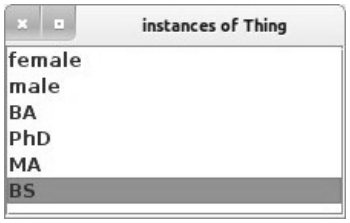


Figura 5.6: Visualizando las instancias.

futura del proyecto (que será continuado) en añadir más parámetros en función de las necesidades que vayan surgiendo.

Un ejemplo de fichero de configuración se ve reflejado en la Figura 5.7 .Para el

```
1  <?xml version="1.0" ?>
2
3  <information>
4
5      <connector>
6          <width>3.0f</width>
7          <color>blue</color>
8      </connector>
9
10     <dashedConnector>
11         <color>red</color>
12     </dashedConnector>
13
14 </information>
15
```

Figura 5.7: Configuración de parámetros sobre fichero XML.

parseo del fichero XML se ha hecho uso de un API de XPATH[3] para Java, un lenguaje de consulta sobre ficheros XML.

5.3. Utilización del prototipo

En esta sección se va a presentar el prototipo implementado. Se va a empezar dando consejos para poder dar los primeros pasos con la aplicación. En segundo lugar, se presenta una evaluación breve frente a distintas ontologías de complejidad y tamaño variables (más detalles de sobre el uso del prototipo en el Anexo B y sobre la evaluación en el Anexo A).

5.3.1. Obtención de la aplicación y uso

El visualizador en su estado actual puede ser probado como applet[2], descargado como aplicación independiente[3], o como plug-in para la plataforma Protégé[4].

La iniciación varía para cada una de las versiones de la aplicación.

- En caso del applet sólo es preciso un navegador con soporte de Java apuntando a la URL.
- Para la aplicación, basta con ejecutar el archivo .jar de las distintas formas posibles (doble-click, java -jar ...).
- En cuanto a Protégé, hay que copiar plug-in a la carpeta plugins de la instalación de Protégé. Será detectado en el inicio del programa.

Para mostrar una ontología clasificada el proceso es igual en el caso del applet y la aplicación independiente (Figura 5.8).

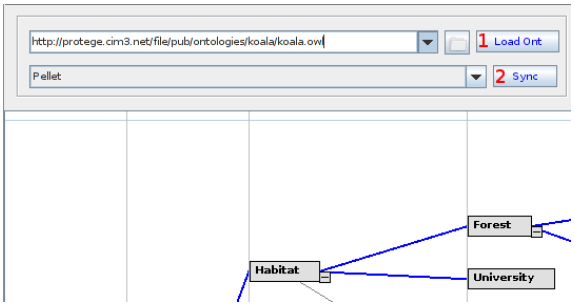


Figura 5.8: Carga y clasificación de una ontología.

5.3.2. Probando la aplicación

En el anexo A se encuentra una evaluación del visualizador frente ontologías con expresividades y tamaños variables junto a la comparación con el resultado obtenido de otros visualizadores. Esta subsección pretende resumir las conclusiones que se pueden obtener de un uso normal de la aplicación realizada.

Con el ejemplo más sencillo de los utilizados, pero no por ello incompleto, podemos observar grandes diferencias. La ontología proyectos.owl abarca muchos aspectos de OWL con apenas unos pocos axiomas. Tiene propiedades, subpropiedades, y

[2]Applet -`http://sid.cps.unizar.es/OntView/OntView.html`
[3]Standalone - `http://sid.cps.unizar.es/OntView/OntView.jar`
[4]Plug-in de Protégé -`http://sid.cps.unizar.es/OntView/OntViewPlugin.jar`

clases definidas a partir de expresiones complejas. En la visualización de este esquema, el primero de los visualizadores (OWLViz) probados pierde demasiada información por el hecho de quedarse con la taxonomía. Apenas se ven cuatro relaciones de subsumición en la Figura 5.9.

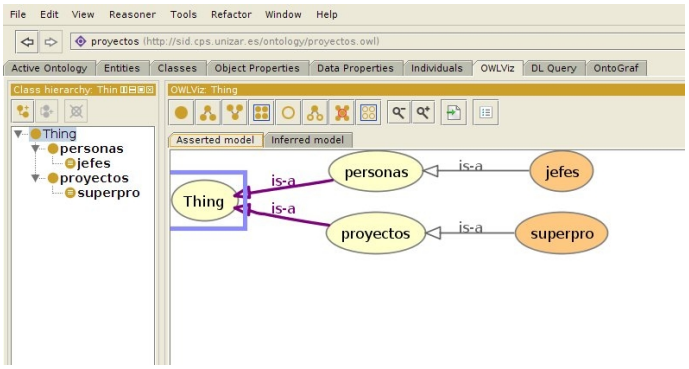


Figura 5.9: Visualización de proyectos.owl en OWLViz.

En OntoGraf, la situación mejora en cuanto a expresividad (Figura 5.10). Con la ayuda de la leyenda de colores de los conectores se intuyen las relaciones existentes. Pese a todo, falla al no poder identificar que jefe es subpropiedad de miembros. Se queda corto en este aspecto. Por contra muestra la A-Box gráficamente, es decir, muestra como nodos a las instancias de las clases.

Finalmente, entra en juego el visualizador presente (Figura 5.11). No pierde expresividad en el campo de la representación de las propiedades como hacía OntoGraf, ni en el de los conceptos, ya que no se conforma con añadir los conceptos sino que añade las definiciones de aquellos que no sean primitivos

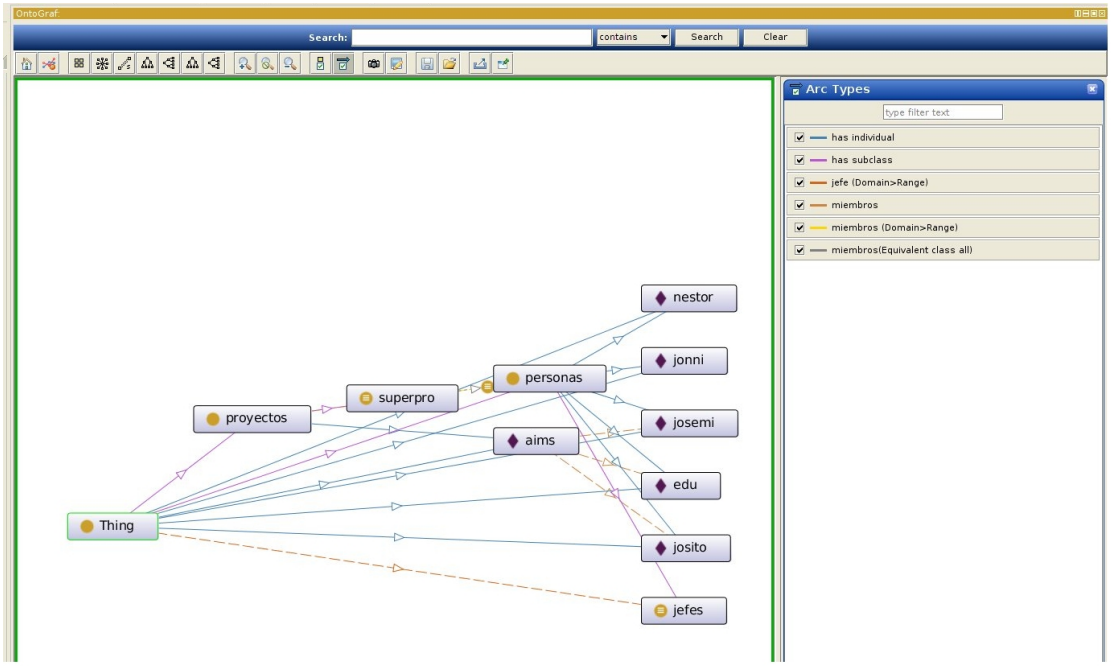


Figura 5.10: Visualización de proyectos.owl en OntoGraf.

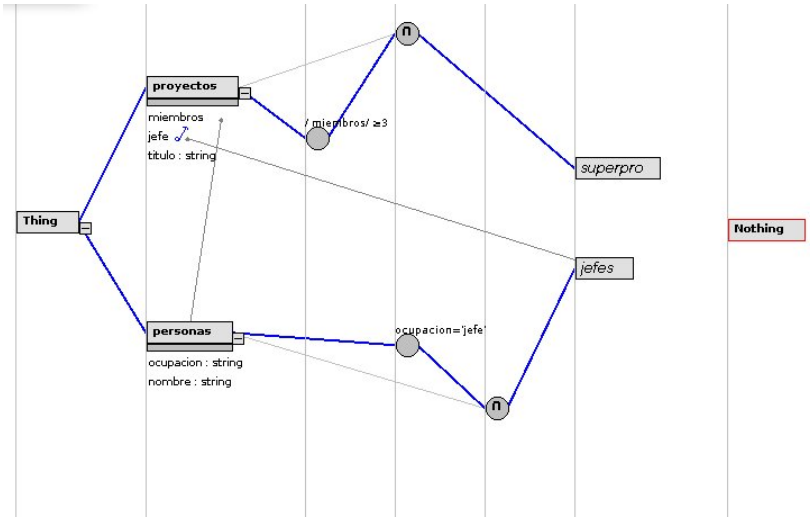


Figura 5.11: Visualización expandida de proyectos.owl en el visualizador desarrollado.

Capítulo 6

Conclusiones

En este capítulo se reflexiona sobre las conclusiones generales que conciernen al desarrollo del proyecto, se apuntan líneas de trabajo futuro para el visualizador, se realiza una breve conclusión personal y para finalizar se muestra la planificación temporal del proyecto.

6.1. Conclusiones generales

La conclusión más importante del desarrollo de este proyecto es que se ha conseguido dar forma dentro del marco de tiempo estimado a una herramienta que está siendo ya utilizada por diseñadores de ontologías. Se ha conseguido realizar un visualizador que:

- Abarca visualmente de forma compacta e intuitiva los aspectos más importantes de las ontologías.
- Permite trabajar con partes de la ontología.
- Permite al usuario continuar la sesión en el estado en el que la dejó.
- Se apoya en el uso de un razonador para encontrar las implicaciones subyacentes.

Desde otro punto de vista, me ha servido para empezar a tocar temas de RDF/RDF-S/OWL/SPARQL y ponerme al día en tecnologías de la Web Semántica que van a ser cada más vez más requeridas en el ámbito empresarial por la interoperabilidad que consiguen.

En resumen, se ha cumplido el objetivo genérico que representaba al principio realizar un visualizador de ontologías.

6.2. Línea de posibles mejoras

En el desarrollo del proyecto han surgido diferentes decisiones que han sido descartadas, no por malas ideas, sino por salirse de la planificación del proyecto o no haber consenso en la aprobación de la actualización.

Entre ellas:

- La primera posible mejora concierne a la adición de las instancias como entidades gráficas.Una forma sencilla de realizar esta mejora se ilustra en la Figura 6.1. La idea consistiría en añadir niveles intermedios donde incluir solamente las instancias con una forma característica y apuntando a la clase a la que pertenecen. Entre los posibles inconvenientes que pueden surgir está la sobrecarga visual. Este es el motivo con más peso para dejar esta mejora en el tintero. Sin embargo, se podría llegar a un compromiso, como por ejemplo, ocultándolas a voluntad o bien con una muestra parcial de las instancias de una clase bajo solicitud.

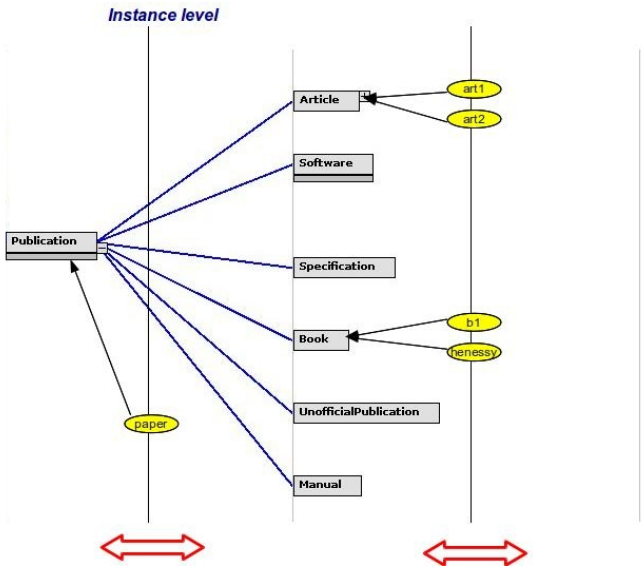


Figura 6.1: Posible mejora con instancias.

- Otra mejora más costosa sería convertir el visualizador en editor/visualizador. No tiene mucho sentido realizar esta mejora en el plug-in de Protégé, ya que Protégé es en sí mismo un editor de ontologías. Sin embargo, para la versión *standalone* habría que estudiar más detenidamente si invertir tiempo/esfuerzo en esta mejora es mejor que centrarse en la especialización como visualizador.

- Con el terreno ya allanado en el módulo de configuración, no sería difícil añadir más elementos configurables desde el xml correspondiente.

- Añadir algo de "*eye-candy*". Las animaciones de los elementos o las formas son planas. Si bien es cierto que se trata de un diseño funcional y efectivo, se podría pulir ligeramente para que fuese más atractivo a la vista del usuario.
- Añadir la posibilidad de asociar un endpoint SPARQL que esté etiquetado con la ontología cargada para poder mostrar estadísticas de las instancias que contiene de manera gráfica.

6.3. Conclusiones personales

El proyecto presente me ha aportado muchos conocimientos a muchos niveles. En primer lugar, desde el punto de vista del trabajo previo, me ha provisto de un conocimiento del lenguaje OWL, una especificación realmente compleja y que puede darme una ventaja competitiva en el mercado laboral dada la escasez de gente con conocimientos de tecnologías semánticas. Aunque el proyecto se haya centrado en la visualización del modelo, se han ido consolidando los conceptos que habían quedado sueltos en los primeros pasos del trabajo previo.

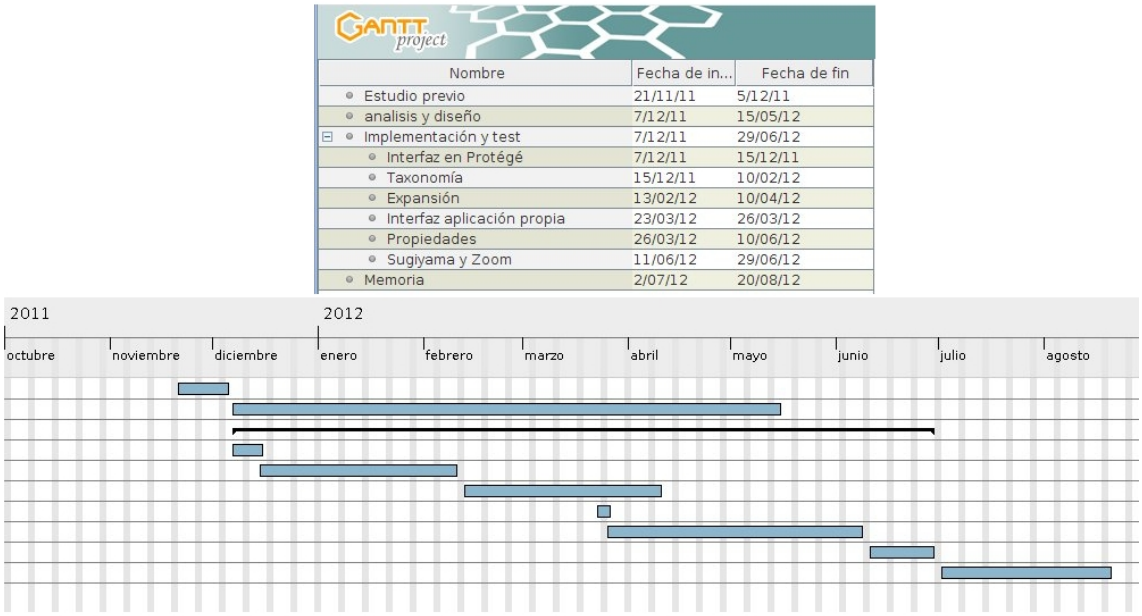
También he aprendido de la metodología utilizada, ya que ha conseguido mejorar mi capacidad de preveer evoluciones de requisitos, de modo que un cambio no signifique desechar una cantidad importante de trabajo.

Desde el punto de vista de las herramientas utilizadas también he ganado mucha experiencia. El lenguaje Java, por ejemplo, es un lenguaje del cual ya poseía cierta experiencia ya que lo he utilizado para alguna pequeña práctica en la carrera. Sin embargo, el desarrollo del proyecto me ha obligado a profundizar más tanto en aspectos técnicos como de entorno. La gestión de versiones es otro concepto del que tenía cierta experiencia y que ha sido consolidado con el uso del CVS proporcionado.

En resumen, el proyecto ha hecho que consiguiera adquirir conocimientos interesantes de cara a mi futuro profesional.

6.4. Marco temporal del proyecto

En la Figura 6.2 tenemos la planificación inicial sobre el reparto del tiempo para las tareas y que se ha ajustado de forma bastante fiel al cabo del proyecto. Respecto a esta planificación se tiene en cuenta el tipo de metodología usada que obliga a reanalizar y rediseñar; así que se solapa durante la mayor parte de la vida del proyecto. Es preciso comentar que en el diagrama se tiene en cuenta un margen para riesgos o incluso imprevistos ajenos que puedan frenar el desarrollo del proyecto.



Tiempo dedicado por tarea

Tarea	Tiempo
Estudio Previo	60
Análisis y Diseño	190
Implementación y tests	380
Memoria	110

Figura 6.2: Diagrama de Gantt y reparto de tiempo.