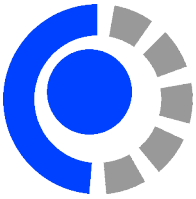
Universidad Nacional del Comahue

Facultad de Informática



Tesis de Licenciado en Ciencias de la Computación

Una Arquitectura Cliente-Servidor para Modelado Conceptual Asistido por Razonamiento Automático

Gimenez, Christian Nelson

Cecchi, Laura Andrea

Braun, Germán Alejandro

Neuquén Argentina

2018

Prefacio

Esta tesis es presentada como parte de los requisitos finales para optar al grado académico de *Licenciado/a en Ciencias de la Computación*, otorgado por la Universidad Nacional del Comahue, y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otras. La misma es el resultado de la investigación llevada a cabo en el Departamento Teoría de la Computación, de la Facultad de Informática, en el período comprendido entre ..... y ..... , bajo la dirección de Cecchi, Laura Andrea y la codirección de Braun, Germán Alejandro.

Gimenez, Christian Nelson
Facultad de Informática
Universidad Nacional del Comahue
Neuquén, .. de ..... del .... .

UNIVERSIDAD NACIONAL DEL COMAHUE

Facultad de Informática

La presente tesis ha sido aprobada el día ....., mereciendo la calificación de .....

Agradecimientos

A mi familia, a Romi y a la familia Otte-Fernandez, por la paciencia y el apoyo.

A Laura C., a Germán B., a mis profesores y a mis compañeros, por ayudarme y guiarme en mi educación y en la realización de esta tesis.

A los miembros del Jurado de esta tesis, por sus valorables sugerencias a la versión original del manuscrito del presente trabajo.

A los que me han ayudado a lo largo de mi vida, por acompañarme a alcanzar mis más deseadas metas.

Resumen

Los sistemas de información basados en ontologías, en especial la Web Semántica, han tenido un gran impulso en los últimos tiempos, aumentando en cuanto a cantidad de datos e información relacionada. La calidad de los mismos está ampliamente determinada por el nivel conceptual, por lo que el diseño de ontologías es clave para la posterior implementación y mantenimiento. Asimismo, la ingeniería ontológica necesita de metodologías y herramientas gráficas para la creación, edición y actualización de modelos que cumplan con criterios de calidad altos y medibles.

Existen diversas herramientas orientadas a la ingeniería ontológica. Sin embargo, éstas no pueden ser utilizadas sin grandes modificaciones para llevar a cabo una integración gráfico-lógica con soporte de servicios de razonamiento. Esto permite brindarle asistencia al usuario en el diseño ontológico, por medio de la utilización de consultas automatizadas que permitan obtener propiedades conceptuales implícitas o explícitas. Particularmente, y tema central para esta tesis, es la posibilidad de consultar la consistencia de una ontología, y por ende, del modelo conceptual asociado.

En base a lo expuesto, se plantea el diseño de una arquitectura para una herramienta Web colaborativa que utilice lenguajes gráficos de modelado conceptual para la creación de ontologías. A fin de asegurar la calidad de los diseños, se incluye un servicio de razonamiento subyacente para resolver consultas acerca de la consistencia del modelo de usuario y sus clases. Para poder llevar a cabo el mapeo gráfico-lógico, y brindar al razonador la ontología necesaria, se utiliza una codificación que formaliza el lenguaje de modelado conceptual en Lógica Descriptiva. Asimismo, para determinar la consistencia del modelo (y de su ontología asociada) se definen las consultas necesarias para que el razonador pueda responderlas.

Esto resulta en una herramienta denominada *crowd* que implementa todos los criterios establecidos en el diseño presentado. Se opta por el uso de un subconjunto de primitivas de UML como lenguaje gráfico. Esto requiere de una biblioteca gráfica para la interfaz, por lo que se han relevado varias disponibles, optando por JointJS. *crowd* puede mostrar al usuario las inconsistencias en un modelo conceptual gracias a la codificación de UML a Lógica Descriptiva escrita en sintaxis OWL 2 y que, con un conjunto de consultas propias del protocolo OWLlink, alimentan al razonador RACER. Las respuestas, son mostradas al usuario bajo el mismo lenguaje gráfico, con notación gráfica resaltada para indicar cuáles elementos gráficos requieren de atención.

Abstract

There is a great impulse on ontology-based information systems, specially the Semantic Web, raising the amount of data and their relations between them in the last years. The quality of these data is widely determined by the conceptual level, making the conceptual modelling a key process for the latest implementation and maintenance of such systems. To this end, this level requires new methodologies and tools that enable domain experts to create, edit and maintain their models that comply with measurable high level quality criteria.

Although there exists a diversity of modelling tools oriented to the ontologic engineering, they cannot be employed without great modifications to achieve a graphical-logical integration with reasoning services support. These services assist users in the ontological design, through automatic queries about the conceptual implicit or explicit properties of the model. Particularly, and of great interest for this thesis, the ontology can be queried about its consistency, and thus, the consistency of its associated conceptual model.

Altogether, a design for a new architecture is proposed, as a collaborative Web tool that supports ontology authoring using visual conceptual modelling languages. To ensure good design qualities, the proposed tool includes reasoning services that can solve consistency queries about the user's models and their classes. To accomplish the graphical-logical mapping and feed to the reasoner with ontologies, a formalisation in Description Logic (DL) of a conceptual model language is evaluated and considered for this work. Moreover, for proving the soundness of models, the respective reasoner queries are described.

Based on the above, this thesis concludes with an implementation called *crowd*, which complies all the established baselines described in the presented design. *crowd* uses a subset of UML primitives as its visual language. For this purpose, JointJS has been selected as graphical library according to predefined criteria defined in this work. This tool can display inconsistencies in user's conceptual models through an UML codification in DL together with a set of queries in the OWLlink protocol for communicating with the RACER reasoning system. Afterwards, the answers are displayed in the same graphical language, with the element's style modified where the user's attention is required.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El avance de las ontologías sobre el terreno de la Web Semántica [20], como su incorporación en otros ámbitos como la medicina [2, 32, 89], la agronomía [27] y la biología [71, 87, 94, 97], ha hecho necesario el desarrollo de herramientas que faciliten al diseño de estas bases de conocimiento. Muchos de estos programas poseen una interfaz textual como Protégé [66], otros utilizan un lenguaje visual similar a los de modelado conceptual o uno completamente innovador (OntoUML [50], Graphol [31], WebVOWL [73], etc.).

También, estas innovaciones promueven la creación de programas razonadores (RACER [51], Hermit [95], etc.) para la utilización de estas bases de conocimiento y para realizar consultas sobre ellas acerca de su diseño conceptual como de sus instancias de datos.

Sin embargo, una herramienta que una varias de estas características es difícil de encontrar, en especial que contemple la representación gráfica del modelo ontológico, su posterior razonamiento sobre consultas que deduzcan posibles fallas en la calidad del modelo. Por consiguiente, en este trabajo, se presenta una arquitectura para la creación de una herramienta Web colaborativa para el diseño de ontologías con capacidades de razonamiento. La finalidad de esta herramienta es doble. Por un lado, permite diseñar en el lenguaje de modelado conceptual gráfico UML, que es de amplio uso en ambientes de desarrollo de software. Por el otro, permite a través de consultas al razonador detectar inconsistencias, para posteriormente señalarlas al usuario en el mismo lenguaje gráfico.

1.1. Motivación

Existe un incremento en la complejidad de los sistemas de información derivado de nuevos conceptos, como por ejemplo, la Web Semántica [20], Big Data [62], E-government [76], etc. los que requieren soluciones de calidad alta. La misma, está ampliamente determinada por el nivel conceptual, por lo tanto, el diseño de ontologías es clave para la posterior implementación y mantenimiento de dichos sistemas. En este contexto, los enfoques basados en lógicas para la representación de información y la adopción de técnicas de razonamiento automático deben asistir a los desarrolladores a obtener modelos conceptuales consistentes. Esta ingeniería ontológica necesita de metodologías y herramientas gráficas para la creación, edición y actualización de modelos que permitan establecer criterios de calidad claros y medibles, con una curva de aprendizaje equiparable con respecto al de modelado conceptual gráfico de amplio uso.

En consecuencia, herramientas gráficas y automáticas que asistan a los modeladores en obtener modelos conceptuales consistentes son esenciales para una integración exitosa entre las intenciones de éstos y la semántica formal de una ontología.

En el desarrollo de ontologías se deben tener en cuenta el conocimiento de los expertos, los sistemas de información preexistentes y los nuevos requerimientos para poder obtener ontologías de calidad que sean útiles.

Las herramientas existentes (véase el capítulo 5) difieren de nuestro principal objetivo, que

es el de enfocarnos en una metodología para crear y entender ontologías, considerando esto como una aproximación al usuario, a la utilización de lenguajes de modelados conceptuales gráficos (UML [21], EER [45], ORM [54]) y la asistencia de servicios de razonamientos para la realización de tareas de diseños automáticos.

La arquitectura presentada en este trabajo concluye en una herramienta Web colaborativa denominada *crowd*. Ésta otorga al usuario la posibilidad de crear sus ontologías por medio de un lenguaje gráfico, con asistencia de un servicio de razonamiento que verifica la consistencia de su modelo.

1.2. Contribuciones

El principal resultado de esta tesis, es el desarrollo una arquitectura Web que permita el diseño de una ontología utilizando el lenguaje de modelado conceptual, con asistencia de un servicio de razonamiento automático a partir de una representación formal del modelo, a fin de poder deducir detalles acerca de la consistencia del modelo y presentarlas en la interfaz bajo el mismo lenguaje conceptual.

Por consiguiente, las principales contribuciones de este trabajo son:

- Diseño de una arquitectura Web que permite el modelado ontológico de forma gráfica con integración a servicios de razonamiento.
- Implementar una codificación del modelo conceptual proveído por el usuario, en una representación formal utilizando Lógicas Descriptivas, basándonos en la descripta por [19]. La motivación de esta implementación es realizar procesos de razonamiento sobre el modelo en busca de inconsitencias.
- Herramienta *crowd*: Implementación funcional de la arquitectura, con soporte para un subconjunto de primitivas del lenguaje de modelado conceptual UML, en particular el diagrama de clases, para poder generar un modelo y obtener su estado de consistencia.
- Explicitar cómo se ha implementado cada módulo y qué se tuvo en cuenta al momento de llevarlo a cabo.

Algunos resultados, relacionados con esta tesis, han formado parte de las siguientes publicaciones:

- Germán Braun, Christian Gimenez, Pablo Fillottrani y Laura Cecchi. “Towards Conceptual Modelling Interoperability in a Web Tool for Ontology Engineering”. *46 Jornadas Argentinas de Informática(*46° *JAIIO) - Simposio Argentino de Ontologías y sus Aplicaciones (SAOA)*. 2017. Córdoba Capital, Córdoba, Argentina.
- Germán Braun, Christian Gimenez, Laura Cecchi y Pablo Fillottrani. “Towards a Visualisation Process for Ontology-Based Conceptual Modelling”. *International Workshop on Description Logic 2017*. 2017. Universidad de Montpellier. Montpellier, Francia.
- Christian Gimenez, Germán Braun, Laura Cecchi y Pablo Fillottrani.“Interoperabilidad entre Lenguajes de Modelado Conceptual en *crowd*”. *XIX Workshop de Investigadores en Ciencias de la Computación (XIX WICC)*. Abril del año 2017. Ciudad Autónoma de Buenos Aires, Buenos Aires, Argentina.
- Germán Braun, Christian Gimenez, Laura Cecchi y Pablo Fillottrani. “Towards a Visualisation Process for Ontology-Based Conceptual Modelling”. *Brazilian Ontology Research Seminar (ONTOBRAS)*. 2016. Curitiba-PR, Brazil.

- Christian Gimenez, Germán Braun, Laura Cecchi y Pablo Fillottrani. "crowd: A Tool for Conceptual Modelling assisted by Automated Reasoning - Preliminary Report". *45 Jornadas Argentinas de Informática (45° JAIIO) - Simposio Argentino de Ontologías y sus Aplicaciones (SAOA)*. Septiembre del 2016. Ciudad Autónoma de Buenos Aires, Buenos Aires, Argentina.

- Christian Gimenez, Germán Braun, Laura Cecchi y Pablo Fillottrani. "Una Arquitectura Cliente-Servidor para Modelado Conceptual Asistido por Razonamiento Automático". *XVIII Workshop de Investigadores en Ciencias de la Computación (XVIII WICC)*. Abril del año 2016. Concordia, Entre Ríos, Argentina. ISBN: 978-950-698-377-2.

1.3. Estructura de la Tesis

La estructura del presente trabajo es la siguiente: En el capítulo 2 se describe la familia de las Lógicas Descriptivas (DL) que se utiliza para expresar las ontologías. Posteriormente, se explica el lenguaje OWL 2 utilizado para representar ontologías en la Web y su relación con las DL. Luego, se describe la utilización de UML como un lenguaje gráfico para ontologías, comentando cómo puede formalizarse este lenguaje de modelado conceptual en Lógica Descriptiva y, por consecuencia, en OWL 2.

En el capítulo 3, se detalla el Proceso de Visualización para Modelados Conceptuales Ontológicos. Luego, se describe la arquitectura propuesta, los módulos y los diseños preliminares que cumplen con los objetivos propuestos por esta tesis y cuya implementación brinde soporte al Proceso de Visualización.

En el capítulo 4, se describe a *crowd*, una implementación de la arquitectura. Se describe cada módulo implementado, el funcionamiento, las tecnologías y los protocolos utilizados en cada uno de ellos. Se concluye el capítulo con un ejemplo de uso de la implementación donde se muestra su funcionamiento con un modelo conceptual satisfacible y otro insatisfacible.

En el capítulo 5 se introducen una serie de herramientas relevadas que permitan visualizar y editar ontologías. Se las compara con *crowd* considerando los lineamientos del Proceso de Visualización, el uso de lenguajes gráficos de modelado y de la utilización de razonamientos subyacentes.

Por último, en el capítulo 6, se presentan las conclusiones de esta tesis, los resultados obtenidos, las contribuciones realizadas y se proponen algunas ideas para trabajos futuros.

Capítulo 2

Lógicas Descriptivas y Lenguajes de Ontologías

En Ciencias de la Computación, una *ontología* es la descripción del conocimiento sobre un dominio de interés. Sin embargo, a diferencia de un modelo conceptual de datos, una ontología provee una representación independiente de una determinada aplicación. Es posible representar el conocimiento de una aplicación de dominio, de forma estructurada y formalmente bien comprendida, expresándola por medio de una Lógica Descriptiva (DL) [10]. Esto dota a los modelos resultantes de capacidades de razonamiento para validarlos y consultar sobre sus propiedades.

Las DLs han sido adoptadas como el principal paradigma para la descripción ontológica, que luego redundó en la estandarización del lenguaje de ontologías Web OWL [47, 68], por parte del *World Wide Web Consortium (W3C)*. Además, dicha adopción posibilitó la construcción de herramientas para inferencias automáticas, definiendo así la base teórica para los sistemas de información. Las DLs son una evolución de las redes semánticas [88] y los sistemas basados en frames [75] que, aunque carentes de una semántica precisa, su ventaja era la facilidad de comprensión [6]. En este contexto, éstas lógicas surgen para dotar a estas representaciones de una semántica formal. Además, las DLs son una familia de lenguajes de representación de conocimiento que pueden ser consideradas como un subconjunto decidible de la lógica de primer orden ($\mathcal{FOL}$). Ellas pueden ser usadas para describir el dominio de una aplicación en términos de clases y relaciones y admitiendo razonamiento decidible. En la misma dirección, surge OWL [84, 3] como un lenguaje formal de la Web Semántica [20], para expresar ontologías. Basado en la semántica de DL, OWL ha sido diseñado para aplicaciones que requieren procesar información, en lugar de sólo presentarla a los humanos, además de incrementar la interoperabilidad Web.

El propósito del presente capítulo es hacer un breve repaso de las lógicas descriptivas básicas, tomando como referencia las lógicas $\mathcal{ALC}$ y $\mathcal{ALCQI}$, y el lenguaje OWL, a través de sus características principales y sus sublenguajes subyacentes. En la sección 2.1 detallamos la sintaxis y semántica de las lógicas $\mathcal{ALC}$ y $\mathcal{ALCQI}$. También, describimos algunas lógicas más expresivas y resumimos sus propiedades computacionales. A continuación, en la sección 2.2, analizamos OWL 1 y OWL 2 y realizamos una breve comparación de ambos. Finalmente, en la sección 2.3 describimos cómo codificar los diagramas de clases UML utilizando las lógicas descriptivas y el lenguaje OWL 2.

2.1. Lógicas Descriptivas

Investigaciones en el campo de la representación de conocimiento y el razonamiento, usualmente se enfocan en métodos que provean una descripción de alto nivel de un mundo, que pueda ser usado de manera efectiva, para construir aplicaciones inteligentes. En este caso, con "inteligente" nos referimos a la posibilidad de encontrar consecuencias implícitas de una representación de conocimiento [6]. A pesar de que las lógicas de primer orden ($\mathcal{FOL}$) ofrecen una maquinaria

muy poderosa y general, para la representación de conocimientos en dominios reales no es preciso utilizar toda su expresividad, mas bien de fragmentos de ella, como queda demostrado por la utilización de redes semánticas y frames descripto en [22]. A partir de esto, se deduce que son posibles los razonamientos basados en diferentes fragmentos de $\mathcal{FOL}$ que conllevan a problemas de diversas complejidades.

En consecuencia, surgen las Lógicas Descriptivas (DLs), las cuales son una familia de lenguajes para la representación de conocimiento de una forma estructural y formalmente bien comprendida [10]. Su nombre proviene del hecho de que las nociones más importantes del dominio son descriptos como *descripciones* de conceptos, en otras palabras, de expresiones que se construyen por medio de conceptos y roles atómicos usando constructores proveídos por la DL particular.

Semánticamente, un *concepto* es interpretado como un conjunto de individuos y los *roles* como conjuntos de pares de individuos. En el aspecto terminológico, una base de conocimiento, llamada TBox, describe nociones relevantes de un dominio de aplicación estableciendo propiedades sobre los conceptos y roles, y relaciones entre ellos. La parte de aserciones de la base de conocimiento, denominada ABox, es usada para describir una situación concreta detallando las propiedades para individuos. Ambos, TBox y ABox definen una ontología en DL.

2.1.1. $\mathcal{ALC}$ y $\mathcal{ALCQI}$

$\mathcal{ALC}$ (Attributive concept Language with Complements) significa "lenguaje de conceptos con atributos y complementos". Se introdujo inicialmente en [91], donde investigaron la incorporación de la unión ($\sqcup$), intersección ($\sqcap$) y complemento ($\neg$) a un lenguaje de conceptos con atributos y su impacto en la complejidad computacional.

A continuación, se brindarán definiciones formales de la sintaxis y semántica de los constructores según se encuentran en [10].

Definición 2.1 (Sintaxis $\mathcal{ALC}$) : Sea N_C un conjunto de nombres de conceptos y N_R un conjunto de nombres roles. El conjunto de descripciones de $\mathcal{ALC}$-conceptos es el conjunto más pequeño de forma que:

1. $\top$, $\bot$ y cada concepto $A \in N_C$ es un $\mathcal{ALC}$-concepto atómico.
2. Si C y D son $\mathcal{ALC}$-conceptos y $r \in N_R$, entonces $C \sqcap D, C \sqcup D, \neg C, \forall r.C$ y $\exists r.C$ son $\mathcal{ALC}$-conceptos.

■

Definición 2.2 (Semántica $\mathcal{ALC}$) : Una interpretación $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consiste de un conjunto no vacío $\Delta^{\mathcal{I}}$ denominado el dominio de $\mathcal{I}$ y la función $\cdot^{\mathcal{I}}$ que asocia cada $\mathcal{ALC}$-concepto con un subconjunto de $\Delta^{\mathcal{I}}$, y cada nombre de rol a un subconjunto de $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, para todos los $\mathcal{ALC}$-conceptos C, D y para todo nombre de rol r:

$$\begin{aligned}
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \qquad \bot^{\mathcal{I}} = \emptyset, \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \qquad (C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}} \qquad \neg C^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}} \\
(\exists r.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} | \text{ existe algún } y \in \Delta^{\mathcal{I}} \quad (x,y) \in r^{\mathcal{I}},\ y \in C^{\mathcal{I}}\} \\
(\forall r.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} | \text{ para todo } y \in \Delta^{\mathcal{I}},\ \text{ si } (x,y) \in r^{\mathcal{I}}, \text{ entonces } y \in C^{\mathcal{I}}\}
\end{aligned}$$

Decimos que $C^{\mathcal{I}}$ ($r^{\mathcal{I}}$) es una extensión del concepto C (rol r) en la interpretación $\mathcal{I}$. Si $x \in C^{\mathcal{I}}$, entonces decimos que x es una instancia de C en $\mathcal{I}$.

■

El dominio de la interpretación puede ser elegido de forma arbitraria y puede ser infinito. Las características de los dominios, en relación a finito/infinito y la consideración del mundo abierto, son distintivas de las lógicas descriptivas, con respecto a los lenguajes de modelado desarrollados en el estudio de bases de datos. Los conceptos atómicos son interpretados como subconjuntos del dominio de interpretación, mientras que la semántica para los otros constructores son especificados definiendo un conjunto de individuos denotado por cada constructor.

Definición 2.3 (GCI y Sintaxis y Semántica TBox) : La inclusión general de conceptos (GCI por sus siglas en inglés "General Concept Inclusion") es de la forma $C \sqsubseteq D$, donde C y D son $\mathcal{ALC}$-conceptos. Un conjunto finito de GCIs es llamado un TBox. Una interpretación $\mathcal{I}$ es un modelo de un GCI $C \sqsubseteq D$ si $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ también lo es de un TBox T si éste lo es de cada GCI en T.

■

El uso de $C \equiv D$ es una abreviación para el par simétrico de GCIs $C \sqsubseteq D$ y $D \sqsubseteq C$.

Por otra parte, los axiomas para indicar que un individuo es una instancia de un concepto dado y para indicar que un par de individuos es una instancia de un rol serán incluidos en lo que se denomina ABox.

Definición 2.4 (Axioma de Aserción) : Un axioma de aserción es de la forma $x\!:\!C$ o $(x, y)\!:\!r$ donde C es un $\mathcal{ALC}$-concepto, r es un $\mathcal{ALC}$-rol y x e y son nombres de individuos. Un conjunto finito de axiomas de aserción es llamado un ABox. Una interpretación $\mathcal{I}$ es un modelo de un axioma de aserción $x : C$ si $x^{\mathcal{I}} \in C^{\mathcal{I}}$ e $\mathcal{I}$ es un modelo de un axioma de aserción $(x, y) : r$ si $(x^{\mathcal{I}}, y^{\mathcal{I}}) \in r^{\mathcal{I}}$; $\mathcal{I}$ es un modelo de un ABox $\mathcal{A}$ si es un modelo para todos los axiomas en $\mathcal{A}$.

■

Existen varias anotaciones para describir un ABox en la literatura, en este trabajo utilizaremos en forma indistinta $C(x)$ o $x : C$ para conceptos, y $r(x, y)$ o $\langle x, y\rangle : r$ para roles.

Definición 2.5 (Base de Conocimiento) : Una base de conocimiento (KB) es un par (T, A), donde T es un TBox y A es un ABox. Una interpretación $\mathcal{I}$ es un modelo de la KB $K = (T, A)$ si $\mathcal{I}$ es un modelo para T e I es un modelo para A.

■

Definición 2.6 (Consecuencia Lógica) : Un axioma α es una consecuencia de una KB, escrito como KB $\models \alpha$ si todo modelo de la KB es también un modelo de α. Por lo tanto, para todo $\mathcal{I}$, donde $\mathcal{I}$ satisface a KB, también se cumple $\mathcal{I} \models \alpha$.

■

Ejemplo 2.1 Consideremos los siguientes $\mathcal{ALC}$-conceptos que describen una KB (un TBox y un ABox).

TBox:
$T = \{\, Padre \equiv Humano \sqcap Hombre \sqcap \exists tieneHijos$
$PadreAlegre \sqsubseteq Padre \sqcap \forall tieneHijos.(Doctor \sqcup Abogado \sqcup PersonaAlegre)$
$AlegreAnc \sqsubseteq \forall descendiente.PadreAlegre$
$Maestro \sqsubseteq \neg Doctor \sqcap \neg Abogado \,\}$

ABox:
$A = \{\, Maestro(mary), tieneHijos(john, mary), AlegreAnc(john) \,\}$

En este ejemplo se presenta una ontología expresada en $\mathcal{ALC}$-conceptos. En la primer línea se expresa que el concepto "padre" coincide con el de los humanos y que son de género masculino, además que deben tener hijos. Luego, un "padre alegre" es un subconjunto de los padres en los que todos sus hijos son doctores, abogados o son personas alegres.

■

La lógica descriptiva $\mathcal{ALCQI}$ aumenta la expresividad de la lógica $\mathcal{ALC}$ ampliando los constructores que define utilizando los roles inversos, denotado como $\mathcal{I}$, y con restricción numérica cualificada, notada como $\mathcal{Q}$. En el primer caso, en esta lógica más expresiva, ahora es posible modelar la inversa de una relación. Por ejemplo, la definición $\exists hijo^{-}.Doctor$ establece que alguien tiene un padre doctor, mediante el uso de la inversa del rol "hijo". Asimismo, las restricciones numéricas cualificadas son una forma de restricciones numéricas, permitiendo la definición de cardinalidades sobre roles, en particular, sobre roles atómicos y su inversa. Usando esta nueva primitiva podemos modelar relaciones como $BuenPadre \sqsubseteq Padre \sqcap (\geq 2\ tieneHijo.Hijo)$, dónde expresamos que un "buen padre" es un padre con al menos 2 hijos.

De forma análoga a las definiciones de sintaxis y semántica $\mathcal{ALC}$, podemos describir la extensión $\mathcal{ALCQI}$ incorporando las definiciones de $\mathcal{I}$ y $\mathcal{Q}$.

Definición 2.7 (Sintaxis $\mathcal{ALCQI}$) : Sea N_C un conjunto de nombres de conceptos y N_R un conjunto de nombre de roles. El conjunto de descripciones de $\mathcal{ALCQI}$-conceptos es el conjunto más pequeño de forma que:

1. $\top$, $\bot$ y cada concepto $A \in N_C$ es un $\mathcal{ALCQI}$-concepto atómicos.

2. Si C y D son $\mathcal{ALCQI}$-conceptos y $r \in N_R$, entonces $C \sqcap D, C \sqcup D, \neg C, \forall r.C$ y $\exists r.C$ son $\mathcal{ALCQI}$-conceptos.

3. Si C es un $\mathcal{ALCQI}$-concepto, $r \in N_R$ y $n \in \mathbb{Z}^{+}$, entonces r^{-}, $(\geq\ n\ r.C)$, $(\leq\ n\ r.C)$ y $(=\ n\ r.C)$ también son $\mathcal{ALCQI}$-conceptos.

■

Definición 2.8 (Semántica $\mathcal{ALCQI}$) : Una interpretación $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consiste de un conjunto no vacío $\Delta^{\mathcal{I}}$ denominado el dominio de $\mathcal{I}$ y la función $\cdot^{\mathcal{I}}$ que asocia cada $\mathcal{ALCQI}$-concepto con un subconjunto de $\Delta^{\mathcal{I}}$, y cada nombre de rol a un subconjunto de $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ de forma que, para todos los $\mathcal{ALCQI}$-conceptos C, D y para todo nombre de rol r:

$$
\begin{aligned}
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \qquad \bot^{\mathcal{I}} = \emptyset, \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \qquad (C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}} \qquad \neg C^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}} \\
(\exists r.C)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} |\ \text{existe algún } b \in \Delta^{\mathcal{I}} \qquad (a,b) \in r^{\mathcal{I}} \text{ e } b \in C^{\mathcal{I}}\} \\
(\forall r.C)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} |\ \text{para todo } b \in \Delta^{\mathcal{I}},\ \text{si } (a,b) \in r^{\mathcal{I}}, \text{ entonces } b \in C^{\mathcal{I}}\} \\
(r^{-})^{\mathcal{I}} &= \{(b,a) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} | (a,b) \in r^{\mathcal{I}}\} \\
(\geq n\ r.C)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} |\ \left|\{b \in \Delta^{\mathcal{I}} | (a,b) \in r^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\right| \geq n\} \\
(\leq n\ r.C)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} |\ \left|\{b \in \Delta^{\mathcal{I}} | (a,b) \in r^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\right| \leq n\} \\
(= n\ r.C)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} |\ \left|\{b \in \Delta^{\mathcal{I}} | (a,b) \in r^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\right| = n\}
\end{aligned}
$$

Decimos que $C^{\mathcal{I}}$ ($r^{\mathcal{I}}$) es una extensión del concepto C (rol r) en la interpretación $\mathcal{I}$. Si $a \in C^{\mathcal{I}}$, entonces decimos que a es una instancia de C en $\mathcal{I}$.

■

Las definiciones subsecuentes de la sección 2.1.1, correspondiente a la inclusión general de conceptos, al axioma de aserción, a la base de conocimiento y consecuencias lógicas en $\mathcal{ALC}$ (definición 2.3, 2.4, 2.5 y 2.6 respectivamente), son análogas para $\mathcal{ALCQI}$.

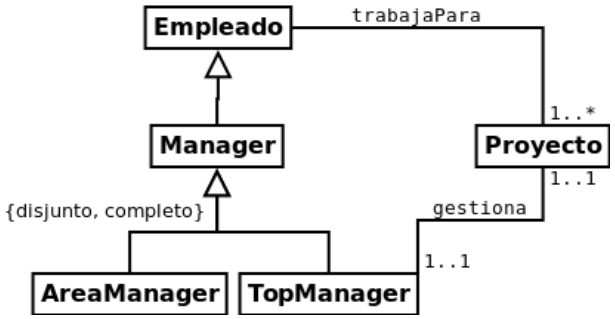


Figura 2.1: Representación de los conceptos y sus relaciones en el ejemplo 2.2.

Ejemplo 2.2 Considere el UML de la Figura 2.1. Una representación $\mathcal{ALCQI}$ de dicho modelo es la siguiente:

TBox:

(1) $Manager \sqsubseteq Empleado$
(2) $AreaManager \sqsubseteq Manager$
(3) $TopManager \sqsubseteq Manager$
(4) $Manager \sqsubseteq AreaManager \sqcup TopManager$
(5) $AreaManager \sqcap TopManager \sqsubseteq \bot$
(6) $\exists trabajaPara \sqsubseteq Empleado$
(7) $\exists trabajaPara^{-} \sqsubseteq Proyecto$
(8) $Proyecto \sqsubseteq \exists trabajaPara^{-}$
(9) $\exists gestiona \sqsubseteq TopManager$
(10) $\exists gestiona^{-} \sqsubseteq Proyecto$
(11) $Proyecto \sqsubseteq \exists gestiona^{-}$
(12) $TopManager \sqsubseteq \exists gestiona$
(13) $(\geq 2\ gestiona) \sqsubseteq \bot$
(14) $(\geq 2\ gestiona^{-}) \sqsubseteq \bot$

Un posible **ABox** sería:

$A = \{\, Empleado(german), Empleado(laura), Empleado(christian), Proyecto(crowd),$
$Manager(german), AreaManager(laura), TopManager(geman),$
$gestiona(german, crowd), trabajaPara(christian, crowd)\,\}$

En este ejemplo se define un ABox el cual cumple con la representación $\mathcal{ALCQI}$. Sin embargo, es posible que un ABox genere inconsistencias en la KB. Considere los siguientes casos:

- $A \cup \{Manager(juan)\}$ debido a que $juan$ debe ser también un $Empleado$ por la regla en 1, pero no está presente $Empleado(juan)$ en el ABox.
- $A \cup \{TopManager(laura)\}$ debido a que $AreaManager(laura)$ y $TopManager(laura)$ existen en el ABox, hace que $AreaManager$ y $TopManager$ no sean disjuntos contradiciendo la regla 5 en el TBox.
- $A \cup \{Empleado(juan), Manager(juan), TopManager(juan), gestiona(juan, crowd)\}$ debido a que hay 2 gestores para $crowd$ contradiciendo la regla 13.
- $A \cup \{Proyecto(linkeddata), gestiona(german, linkeddata)\}$ debido a que hay 2 proyectos gestionados por $german$ contradiciendo la regla 14.

En los dos primeros casos de ABox inconsistentes, la expresividad de la lógica $\mathcal{ALC}$ puede representar las restricciones necesarias. Sin embargo, para los dos últimos, esto no es posible: no se puede limitar la cantidad de elementos que están dentro de un rol, debido a la falta de constructores cuantificadores como los expresados en las reglas 13 y 14.

■

2.1.2. Familia de Lógicas Descriptiva

Para muchas aplicaciones, el poder expresivo de las lógicas $\mathcal{ALC}$ y $\mathcal{ALCQI}$ puede no ser suficiente. Por lo tanto, para dar lugar a lógicas más expresivas, otros constructores fueron introducidos en [91] junto con una propuesta de un primer esquema de nombres para DLs. Partiendo de los proveídos por la lógica $\mathcal{AL}$, la incorporación de otros constructores es indicado, mediante la concatenación al nombre de la lógica, las letras que representan a cada constructor agregado. Así, $\mathcal{ALC}$ es obtenido desde $\mathcal{AL}$ junto con el constructor de complemento ($\neg$). Sin detallar totalmente la sintaxis y semántica de cada una, en la Tabla 2.1 se resumen dichas lógicas junto con sus símbolos representativos. Además, en [7] se describen lógicas que no poseen todos los constructores dados por $\mathcal{AL}$, por ejemplo, la DL $\mathcal{FL}^-$ se obtiene a partir de $\mathcal{AL}$ pero sin permitir la negación atómica, $\mathcal{FL}_0$ se obtiene a partir de $\mathcal{FL}^-$, pero quitando los cuantificadores existenciales limitados. Las DLs mencionadas hasta ahora poseen como constructores de conceptos la intersección y restricciones de valor como núcleo común, pero la familia de lógicas $\mathcal{EL}$ excluye las restricciones de valor, sólo la intersección de conceptos y el cuantificador existencial se pueden utilizar. Con la finalidad de evitar nombres muy largos para expresar DLs, la abreviatura $\mathcal{S}$ fue introducida para referirse a las lógicas $\mathcal{ALC}_{R^+}$ (en otras palabras, la lógica $\mathcal{ALC}$ extendida con roles transitivos).

Otros constructores y propiedades de relevancia se describen también con letras, como la letra $\mathcal{H}$ que representa subroles o jerarquía de roles, $\mathcal{O}$ representa nominales, $\mathcal{I}$ representa roles inversos, $\mathcal{N}$ restricciones y $\mathcal{Q}$ restricciones numéricas calificadas. La integración con un dominio/tipo de dato concreto es indicado agregando su nombre entre paréntesis, o una **D** "genérica". Por ejemplo, la DL correspondiente para el lenguaje de ontologías OWL DL que incluye todos los constructores es denominado $\mathcal{SHOIN}(\mathbf{D})$.

Nombre	Sintaxis	Semántica	Símbolo
Top	$\top$	$\Delta^{\mathcal{I}}$	$\mathcal{AL}$ / $\mathcal{EL}$
Bottom	$\bot$	$\emptyset$	$\mathcal{AL}$ / $\mathcal{EL}$
Intersección	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$	$\mathcal{AL}$ / $\mathcal{EL}$
Cuantificador existencial limitado	$\exists R.\top$	$\{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}}\}$	$\mathcal{AL}$ / $\mathcal{EL}$
Restricción de valor	$\forall R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\}$	$\mathcal{AL}$
Negación	$\neg C$	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$	$\mathcal{C}$
Restricción num. calificada	$\geq nR.C$ $\leq nR.C$ $= nR.C$	$\{a \in \Delta^{\mathcal{I}} \mid \mid \{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\mid \geq n\}$ $\{a \in \Delta^{\mathcal{I}} \mid \mid \{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\mid \leq n\}$ $\{a \in \Delta^{\mathcal{I}} \mid \mid \{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\mid = n\}$	$\mathcal{Q}$
Inverso	r^-	$\{(b,a) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (a,b) \in r^{\mathcal{I}}\}$	$\mathcal{I}$
Unión	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$	$\mathcal{U}$
Cuantificador existencial	$\exists R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$	$\mathcal{E}$ / $\mathcal{EL}$
Restricción num. no calificada	$\geq nR$ $\leq nR$ $= nR$	$\{a \in \Delta^{\mathcal{I}} \mid \mid \{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\}\mid \geq n\}$ $\{a \in \Delta^{\mathcal{I}} \mid \mid \{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\}\mid \leq n\}$ $\{a \in \Delta^{\mathcal{I}} \mid \mid \{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\}\mid = n\}$	$\mathcal{N}$
Mapeo rol-valor	$R \subseteq S$ $R = S$	$\{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow (a,b) \in S^{\mathcal{I}}\}$ $\{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R^{\mathcal{I}} \leftrightarrow (a,b) \in S^{\mathcal{I}}\}$	
Acuerdo y desacuerdo	$u_1 \doteq u_2$ $u_1 \neq u_2$	$\{a \in \Delta^{\mathcal{I}} \mid \exists b \in \Delta^{\mathcal{I}}.u_i^{\mathcal{I}}(a) = b = u_2^{\mathcal{I}}(a)\}$ $\{a \in \Delta^{\mathcal{I}} \mid \exists b_1, b_2 \in \Delta^{\mathcal{I}}.u_i^{\mathcal{I}}(a) = b_1 \neq b_2 = u_2^{\mathcal{I}}(a)\}$	$\mathcal{F}$
Nominal	I	$I^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ con $\mid I^{\mathcal{I}} \mid = 1$	$\mathcal{O}$

Tabla 2.1: Sintaxis y semántica de constructores para la familia de DL. Extraída de [6] y ampliada según su texto.

2.1.3. Servicios de Razonamiento y Propiedades Computacionales

La importancia de las DLs y su tratabilidad se expresa en la capacidad de consultar el conocimiento representado a través de un conjunto de tareas o servicios de razonamiento para tal fin. Dichas tareas, además pueden ser reducidas, desde un punto de vista lógico, a otras tareas facilitando el desarrollo de herramientas como la que presentamos en este trabajo. A continuación detallaremos los servicios más relevantes, a partir de los cuales, clasificaremos las propiedades computacional de cada lógica.

- **Satisfacibilidad de la Base de Conocimiento**. Una KB es satisfacible (o consistente) si tiene un modelo. Por lo tanto, existe una interpretación $\mathcal{I}$ tal que $\mathcal{I}$ es modelo de la KB [8, 51].
- **Inferencia de Axiomas**. Una KB infiere un axioma α si todo modelo de la KB es también un modelo de α. El problema de chequear la inferencia de axiomas puede reducirse a decidir si una KB es satisfacible [51].
- **Satisfacibilidad de Conceptos**. Dado una KB, un concepto C es satisfacible con respecto a la KB si puede ser instanciado. Por lo tanto, existe un modelo I de KB que mapea C a un conjunto no vacío: $C^{\mathcal{I}} \neq \emptyset$. El problema de decidir la satisfacibilidad de un concepto C puede ser reducido al servicio anterior determinando si KB $\models C \sqsubseteq \bot$ [8, 51].

Decidir si una base de conocimiento es consistente es importante *per se*, ya que un KB inconsistente puede llevar a errores de modelado severos y a la derivación de axiomas inválidos. Una KB donde todos los conceptos son satisfacibles es llamada *coherente*.

Es posible clasificar la complexidad computacional de las lógicas descriptivas en términos de satisfacibilidad de conceptos y axiomas generales de la forma $C \sqsubseteq D$, para conceptos arbitrarios C y D. La Tabla 2.2 presenta una clasificación no exhaustiva con sus respectivas referencias.

DL	Complejidad	Comentario
$\mathcal{ALC}$	PSPACE-completo [91]	
$\mathcal{ALCI}$	PSPACE-completo [100]	
$\mathcal{ALCQI}$	PSPACE-completo [100]	
$\mathcal{SHIF}$	ExpTime-completo [100, 57]	
$\mathcal{SHIN}$	ExpTime-completo [100, 57]	OWL-Lite [10]
$\mathcal{SHOIQ}(\mathcal{D})_n^-$	$\mathcal{SHOIQ}$ es NExpTime-completo [100, 99]	Usado para DIG [14]
$\mathcal{SHOIN}$	NExpTime-completo [100, 99]	Usado para OWL-DL [10]
$\mathcal{SROIQ}$	NExpTime-hard [99, 61, 60]	Usado para OWL 1.1 [85, 60]

Tabla 2.2: Complejidad Computacional para satisfacibilidad de conceptos de algunas Lógicas Descriptivas

2.2. Lenguaje de Ontologías Web: OWL 2

La Web Semántica es una evolución de la Web actual donde la información tiene un significado explícito posibilitando el procesamiento automático por parte de las máquinas y la integración de la información disponible. La Web Semántica intenta integrar la capacidad del lenguaje XML para definir etiquetas y la flexibilidad de RDF para representar datos. Sin embargo, un nivel superior es necesario para describir el significado de los términos usados en estos nuevos documentos Web. En este contexto y, debido a la necesidad de ejecutar tareas de razonamiento sobre tales documentos, surge el Lenguaje de Ontologías Web (OWL). OWL es un lenguaje de representación de conocimiento para la Web Semántica con un significado definido formalmente, ya que está

basado en Lógicas Descriptivas. A partir de él, es posible definir ontologías especificando clases, individuos y valores de datos que pueden ser almacenados como documentos Web semánticos. Además, estas ontologías pueden incluir información escrita en otros lenguajes, por ejemplo RDF, para la descripción de recursos Web, versionado y anotaciones.

Debido a sus grados de expresividad, el estándar OWL fue dividido en los siguientes sublenguajes: OWL Full, OWL DL y OWL Lite. El primero, OWL Full, contiene a los restantes sublenguajes y su expresividad lo vuelve indecidible. Una de las razones de ésta indecidibilidad es la falta de restricciones al momento de combinar individuos, clases y roles. Diferente es el caso de OWL DL el cual es soportado por varias herramientas de software debido a su decidibilidad. Si bien OWL DL incluye todos los constructores OWL, ellos no pueden ser combinados de maneras aleatorias. En este contexto, definiciones de clases como instancias de otras clases no se permiten, entre otras. Finalmente OWL Lite, mantiene la decidibilidad pero es aún menos expresivo que los anteriores. OWL Lite restringe aún más sus definiciones permitiendo sólo, por ejemplo, valores de cardinalidades 0 y 1. Las definiciones completas de estos sublenguajes pueden ser consultadas en [105, 104].

En particular, OWL 2 [56, 84] es esencialmente una pequeña extensión de su predecesor, OWL 1. Desde un punto de vista sintáctico, OWL 2 introduce una nueva sintaxis llamada *Sintaxis Funcional*, que reemplaza a la sintaxis abstracta de OWL 1. El propósito de esta nueva sintaxis es facilitar la lectura estructural de las ontologías. Además, OWL 2 mantiene la compatibilidad hacia atrás de OWL 2 DL con respecto al sublenguaje OWL 1 DL, mientras que define los siguientes perfiles explorando otras expresividades y propiedades computacionales, cuyos resultados son tres perfiles explayados en [78]: OWL 2 EL, OWL 2 QL y OWL 2 RL. En tanto que OWL 2 Full continua con las mismas limitaciones que el OWL 1 Full en referencia a su indecidibilidad.

Con respecto a los nuevos perfiles OWL 2 [78], OWL EL es utilizado en aplicaciones con ontologías que contienen un gran número de propiedades y/o clases. Captura un poder expresivo utilizado por la mayoría de las ontologías siendo un subconjunto de OWL 2 por lo que las tareas básicas de razonamiento pueden ser realizadas en tiempo polinomial con respecto al tamaño de la ontología. El acrónimo "EL" refleja la familia $\mathcal{EL}$ [5, 11] de la lógica descriptiva, las cuales proveen solamente cuantificadores existenciales. OWL QL se utiliza especialmente para aplicaciones que usan un gran número de instancias de datos, donde las consultas es la tarea de razonamiento más importante. Basado en la familia DL-Lite, usando técnicas de razonamiento consistentes y completas, una consulta conjuntiva puede ser realizada en LOGSPACE [25] con respecto al tamaño de los datos. El acrónimo "QL" refleja el hecho de que, responder consultas en este perfil, puede ser implementado por medio de una reescritura de las consultas a un lenguaje de consultas("Query Language" en inglés) relacional estándar. Por último, el objetivo de OWL 2 RL es tratar con aplicaciones que requieren razonamiento escalable sin sacrificar demasiado poder expresivo. Está diseñado para una fácil adopción por parte de los sistemas de inferencia basados en reglas y restringe el uso de constructores a ciertas posiciones sintácticas. El acrónimo "RL" refleja el hecho de que el razonamiento en este perfil puede ser implementado usando un lenguaje de reglas estándares.

Sintaxis y Semántica de OWL

La sintaxis primaria para almacenar ontologías es la sintaxis *RDF/XML* [16], también basada sobre RDF y por lo tanto, orientada al intercambio de datos. OWL 2 también incluye otras definiciones sintácticas: serializaciones RDF tales como *Turtle* [18] cuyo propósito es facilitar la lectura/escritura de tripletas RDF; una serialización XML, *OWL/XML* [84, 79] para un manejo eficiente de herramientas XML y una sintaxis llamada *Manchester* [84, 59] la cual intenta hacer más entendibles los documentos ontológicos.

Para poder razonar sobre las ontologías OWL 2, es necesario especificar su semántica. Actualmente, OWL 2 presenta dos semánticas estrechamente relacionadas: *Semántica Directa* (Direct Semantics) y *Semántica Basada en RDF* (RDF-based Semantics).

$\mathcal{ALCQI}$	OWL 2	$\mathcal{ALCQI}$	OWL 2

$C, \top$	`<owl:Class IRI="C"/>`
	`<owl:Class`
	`abbreviatedIRI="owl:Thing"/>`
		$A \sqsubseteq B$	`<owl:SubClassOf>`
			`<owl:Class IRI="A"/>`
			`<owl:Class IRI="B"/>`
			`</owl:SubClassOf>`

$A \sqcap B$	`<owl:ObjectIntersectionOf>`
	`<owl:Class IRI="A"/>`
	`<owl:Class IRI="B"/>`
	`</owl:ObjectIntersectionOf>`
		$A \sqcup B$	`<owl:ObjectUnionOf>`
			`<owl:Class IRI="A"/>`
			`<owl:Class IRI="B"/>`
			`</owl:ObjectUnionOf>`

R^-	`<owl:ObjectInverseOf>`
	`<owl:ObjectPropertyOf IRI="R"/>`
	`</owl:ObjectInverseOf>`
		$\neg C$	`<owl:ObjectComplementOf>`
			`<owl:Class IRI="C"/>`
			`</owl:ObjectComplementOf>`

$(\leq n\, R.\top)$	`<owl:ObjectMinCardinality`
	`cardinality="n">`
	`<owl:ObjectProperty IRI="R"/>`
	`<owl:Class`
	`abbreviatedIRI="owl:Thing"/>`
	`</owl:ObjectMinCardinality>`
		$(\geq n\, R.\top)$	`<owl:ObjectMaxCardinality`
			`cardinality="n">`
			`<owl:ObjectProperty IRI="R"/>`
			`<owl:Class`
			`abbreviatedIRI="owl:Thing"/>`
			`</owl:ObjectMaxCardinality>`

$\forall R.C$	`<owl:ObjectAllValuesFrom>`
	`<owl:ObjectProperty IRI="R"/>`
	`<owl:Class IRI="C"/>`
	`</owl:ObjectAllValuesFrom>`
		$\exists R.C$	`<owl:ObjectSomeValuesFrom>`
			`<owl:ObjectProperty IRI="R"/>`
			`<owl:Class IRI="C"/>`
			`</owl:ObjectSomeValuesFrom>`

Tabla 2.3: Referencias de los axiomas OWL 2 y su significado en $\mathcal{ALCQI}$. Extraído de [12, 56].

La semántica directa define el significado de los axiomas OWL mediante relaciones directas a las Lógicas Descriptivas. Esto resulta en una semántica compatible con la semántica de modelos de $\mathcal{SROIQ}$ y en la posibilidad de utilizar herramientas para modelado ontológico integradas a sistemas de razonamiento. Por otro lado, la semántica basada en RDF primero traduce axiomas OWL en grafos dirigidos expresados en RDF. Esta semántica es completamente compatible con la de RDF y puede ser también aplicada a cualquier ontología OWL 2 sin restricciones. Finalmente, la relación entre ambas es estrecha ya que dada una ontología OWL 2 DL, las inferencias hechas usando la semántica directa serán también válidas si dicha ontología es mapeada a grafos RDF e interpretada usando la semántica basada en RDF. Este último resultado ha sido publicado en [92]. La Tabla 2.3 muestra un correspondencia entre la lógica $\mathcal{ALCQI}$ y los constructores de OWL 2.

Ejemplo 2.3 Considere la Tabla 2.4 donde se identifica las partes relevantes de un documento OWL/RDF por medio de una ontología usando dicha sintaxis. Los nombres e identificadores que se utilizan son mayormente IRIs. La primer sección son los espacios de nombres XML a utilizar y las abreviaturas asociadas a ellas. Los espacios de nombres de este ejemplo corresponden a OWL, RDFS (RDF Schema), RDF, XSD (XML Schema). En "Información Básica" se encuentra información que se considera útil para varias aplicaciones. También se provee el nombre, que usualmente es la ubicación en la Web en la que está disponible. Es común reutilizar información que está dentro de otra ontología importándola. En el ejemplo la ontología se denomina `http://example.com/owl/families` y se reutiliza los datos contenidos en `http://example.org/otherOntologies/families.owl`. Luego, se describe el nivel conceptual como los conceptos (`owl:Class`) y los roles (`owl:ObjectProperty` y `owl:DatatypeProperty`). En este caso, se define el concepto "Woman" como incluido dentro de "Person", los roles "hasParent" y su inverso "hasChild", el cual debe ser disjunto con "hasSpouse". También, se describe otro rol denominado "hasAge" cuyo dominio es "Person" y rango un tipo de dato "nonNegativeInteger" descripto en XML Schema. Finalmente, el ejemplo describe un individuo del concepto "Woman" llamado "Mary". Indica que, en la ontología importada, "Mary" es el mismo individuo descripto allí como "MaryBrown".

En DL, se describen los conceptos y roles de la siguiente manera:
$Woman \sqsubseteq Person$
$hasParent^- \equiv hasChild$

$hasParent \sqcap hasSpouse \equiv \emptyset$
$(\exists hasAge.\top \sqsubseteq Person)$
$(\exists hasAge^{-}.\top \sqsubseteq nonNegativeInteger)$

$Woman(Mary)$

Espacios de Nombres	`<rdf:RDF xml:base="http://example.com/owl/families/"`
	`  xmlns="http://example.com/owl/families/"`
	`  xmlns:otherOnt=`
	`  "http://example.org/otherOntologies/families/"`
	`  xmlns:owl="http://www.w3.org/2002/07/owl#"`
	`  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"`
	`  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"`
	`  xmlns:xsd="http://www.w3.org/2001/XMLSchema#">`

Información Básica	`<owl:Ontology rdf:about="http://example.com/owl/families">`
	`  <owl:imports rdf:resource=`
	`  "http://example.org/otherOntologies/families.owl" />`
	`</owl:Ontology>`

Conceptos	`<owl:Class rdf:about="Woman">`
	`  <rdfs:subClassOf rdf:resource="Person"/>`
	`</owl:Class>`

Roles	`<owl:ObjectProperty rdf:about="hasParent">`
	`  <owl:inverseOf rdf:resource="hasChild"/>`
	`  <owl:propertyDisjointWith rdf:resource="hasSpouse"/>`
	`</owl:ObjectProperty>`
	`<owl:DatatypeProperty rdf:about="hasAge">`
	`  <rdfs:domain rdf:resource="Person"/>`
	`  <rdfs:range rdfs:Datatype=`
	`  "http://www.w3.org/2001/XMLSchema#nonNegativeInteger"/>`
	`  <owl:equivalentProperty rdf:resource="&otherOnt;age"/>`
	`</owl:DatatypeProperty>`

Individuos	`<Person rdf:about="Mary">`
	`  <rdf:type rdf:resource="Woman"/>`
	`  <owl:sameAs rdf:resource="&otherOnt;MaryBrown"/>`
	`</Person>`

	`</rdf:RDF>`

Tabla 2.4: Partes de un documento OWL RDF para una ontología simple. Extractos de OWL RDF obtenidos de [56].

■

Para finalizar esta sección, resumimos en la Tabla 2.5, los distintos perfiles de OWL junto con su complejidad computacional para cada situación.

Lenguaje	Problema de Razonamiento	Complejidad Taxonómica	Complejidad Datos	Complejidad Consultas	Complejidad Combinada
OWL 2 con semántica basada en RDF	Consistencia ontológica, Subsume en expresiones de clases, Chequeo de instancia, Consulta conjuntiva	Indecidible	Indecidible	Indecidible	Indecidible
OWL 2 con semántica directa	Consistencia ontológica, Satisfacibilidad de expresiones de clases, Subsume en expresiones de clases, Chequeo de instancias	N2EXPTIME-completo	Decidible pero de complejidad abierta (NP-Hard)	No Aplicable	N2EXPTIME-completo
	Consulta conjuntiva	Decibilidad Abierta	Decibilidad Abierta.	Decibilidad Abierta	Decibilidad Abierta
OWL 2 EL	Consistencia ontológica, Satisfacibilidad de expresiones de clases, Subsume en expresiones de clases, Chequeo de instancias	PTime-completo	PTime-completo	No Aplicable	PTime-completo
	Consulta conjuntiva	EXPTIME	PTime-completo	NP-completo	EXPTIME
OWL 2 QL	Consistencia ontológica, Satisfacibilidad de expresiones de clases, Subsume en expresiones de clases, Chequeo de instancias	NLogSpace-completo	AC^0	No Aplicable	NLogSpace-completo
	Consulta conjuntiva	NLogSpace-completo	AC^0	NP-completo	NP-completo
OWL 2 RL	Consistencia ontológica	PTime-completo	PTime-completo	No Aplicable	PTime-completo
	Satisfacibilidad de expresiones de clases, Subsume en expresiones de clases, Chequeo de instancias	PTime-completo	PTime-completo	No Aplicable	co-NP-completo
	Consulta conjuntiva	PTime-completo	PTime-completo	NP-completo	NP-completo
OWL 1 DL	Consistencia ontológica, Satisfacibilidad de expresiones de clases, Subsume en expresiones de clases, Chequeo de instancias	NEXPTIME-completo	Decidible pero de complejidad abierta (NP-Hard)	No Aplicable	NEXPTIME-completo
	Consulta conjuntiva	Decibilidad Abierta	Decibilidad Abierta	Decibilidad Abierta	Decibilidad Abierta

Tabla 2.5: Complejidad computacional para los distintos perfiles de OWL 2. Tabla extraída de [83].

2.3. UML como Lenguaje Gráfico para Ontologías

El modelado conceptual usando Lógicas Descriptivas (y OWL) permite eliminar ambigüedades en dichos modelos dando interpretaciones precisas del dominio, además de proveer capacidades de razonamiento para validarlo. En ambos casos, el inconveniente principal es que se requieren modeladores y usuarios con conocimientos en estos lenguajes para poder interpretar y comunicar los modelos. En este contexto, lenguajes de modelado conceptual gráfico tales como UML [21], EER [45] o ORM [54], podrían ser de ayuda. Estos lenguajes estándar permiten modelar la información de un dominio de interés en términos de clases y relaciones entre ellos, aunque no están exentos de introducir información implícita, inconsistencias o redundancias que pueden estar ocultas, principalmente, en modelos complejos. Por lo tanto, equipar a las herramientas para el diseño conceptual con razonamiento automático y permitir la detección de propiedades relevantes en los diagramas, es altamente deseable.

Desde un punto de vista teórico, diferentes trabajos han mostrado que el razonamiento sobre UML [19], EER [4] y ORM 2 [53, 43] es posible, con ciertas restricciones propias de la expresividad de cada uno de los formalismos. En todos los casos, la expresividad es $\mathcal{ALCQI}$, para la cual la complejidad del razonamiento es EXPTIME-Completo. La Tabla 2.3 muestró la correspondencia entre los constructores de OWL 2 y la lógica $\mathcal{ALCQI}$.

En el presente trabajo utilizaremos UML y, en particular, los diagramas de clases como lenguaje gráfico de modelado. UML es el estándar *de-facto* para análisis y diseño de software. Es ampliamente usado en aplicaciones de software a escala industrial y diversas herramientas CASE ("Computer Aided Software Engineering"[1]) lo soportan, como por ejemplo, ArgoUML [90, 29] y Poseidon, entre otras. Una primera aproximación al razonamiento en UML es presentado en [19], codificando en lógica las principales primitivas estáticas de UML relevantes conceptualmente y obviando aquellas relevantes a nivel implementación. De esta manera, se tomarán en cuenta diagramas que incluyan clases, relaciones binarias, n-arias y generalizaciones (totales y disjuntas) y atributos. La Tabla 2.6 muestra cómo estas primitivas son codificadas en $\mathcal{ALCQI}$. Dicha codificación es la que consideraremos para el desarrollo de nuestra herramienta.

Ejemplo 2.4 Consideremos el modelo de la Figura 2.1 y su representación en lógica descriptiva $\mathcal{ALCQI}$ dadas en el Ejemplo 2.2, un documento OWL que represente el modelo indicado se puede observar en formato OWL/XML en la Figura A.1 y Turtle en la Figura A.2 del Apéndice A.1.

■

[1] Según [69], CASE se define como la aplicación científica de un conjunto de herramientas y métodos a un software resultando en un producto de alta calidad, libre de defectos y mantenible.

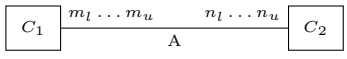
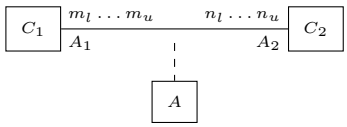
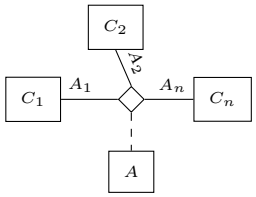
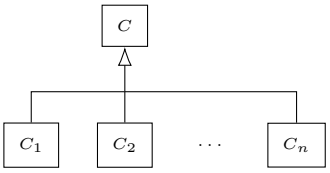
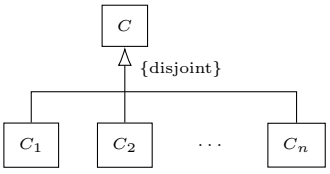
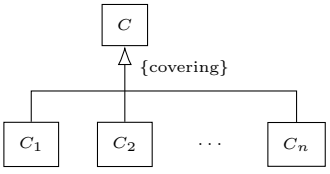
Expresión	UML	$\mathcal{ALCQI}$
Clase		C
Clase con atributo		$C \sqsubseteq \forall a.T$
Asociación binaria sin clase asociativa		$\top \sqsubseteq \forall A.C_2 \sqcap \forall A^-.C_1$ $C_1 \sqsubseteq (\geq n_l\, A.\top) \sqcap (\leq n_u\, A.\top)$ $C_2 \sqsubseteq (\geq m_l\, A^-.\top) \sqcap (\leq m_u\, A^-.\top)$
Asociación binaria con clase asociativa		Usar reificación. $A \sqsubseteq \exists A_1.C_1 \sqcap \exists A_2.C_2 \sqcap$ $(\leq 1\, A_1) \sqcap (\leq 1\, A_2)$ $C_1 \sqsubseteq (\geq n_l A_1^-.A) \sqcap (\leq n_u A_1^-.A)$ $C_2 \sqsubseteq (\geq m_l A_2^-.A) \sqcap (\leq m_u A_2^-.A)$
Asociación n-aria con clase asociativa		Usar reificación. $A \sqsubseteq \exists r_1.C_1 \sqcap \ldots \sqcap \exists r_n.C_n \sqcap$ $(\leq 1\, r_1) \sqcap \ldots \sqcap (\leq 1\, r_n)$
Herencia		$C_1 \sqsubseteq C, \ldots, C_n \sqsubseteq C$
Herencia disjunta		$C_i \sqsubseteq (\sqcap_{j=i+1}^{n} \neg C_j)$ para $1 \leq i \leq n-1$
Herencia completa		$C \sqsubseteq (\sqcup_{i=1}^{n} C_i)$

Tabla 2.6: Resumen de las primitivas UML y su codificación a $\mathcal{ALCQI}$. Tabla elaborada según la codificación explicada en [19].

Capítulo 3

crowd: Su Arquitectura

En el presente capítulo vamos a sintetizar los conceptos presentados anteriormente en una arquitectura Web, la cual está enmarcada en el Proceso de Visualización presentado en [23], cuya finalidad es la de mejorar la calidad de los diagramas reduciendo las inconsistencias y/o anomalías.

crowd es una herramienta que hemos desarrollado para implementar dicha arquitectura. La misma fue diseñada con la posibilidad de que funcione como un servicio Web de capacidades escalables y extensibles, en especial con respecto al lenguaje gráfico y a su codificación a un lenguaje formal. Por ello, la arquitectura está dividida en dos partes: el cliente y el servidor. En este capítulo, detallaremos sus estructuras, la comunicación entre ambas partes y las relaciones entre sus módulos internos.

Se utilizará las Lógicas Descriptivas como un medio para dotar a esta herramienta de una codificación de los modelos que posibilita el razonamiento automático. Esto nos permite detectar propiedades formales relevantes como las inconsistencias o las redundancias [19]. Para hacer posible esto, la arquitectura debe contemplar la codificación y el razonamiento de una serie de consultas sobre la ontología resultante a partir del modelo proveído por el usuario.

La organización del presente capítulo es la siguiente: primero, se presenta el Proceso de Visualización para Modelado Conceptual Ontológico y la relación con *crowd*. Luego, en la sección 3.2, se describe la arquitectura junto con los módulos que la componen, partiendo de una descripción del cliente y posteriormente del servidor, para finalmente detallar cómo se realiza el servicio de razonamiento para chequear la satisfacibilidad del modelo del usuario.

3.1. *crowd* y el Proceso de Visualización para Modelado Conceptual Ontológico

En [23] hemos presentado el Proceso de Visualización de conocimiento que se concibe con la intención de mejorar la calidad de los diagramas reduciendo las inconsistencias y/o anomalías. Además, posee los siguientes objetivos: la integración entre las representaciones visuales y las capacidades de razonamientos lógicos; la definición de qué es relevante y cómo debe ser representado visualmente y el uso de las representaciones gráficas como fuente de evaluación de la calidad de los modelos y su correspondencia con el dominio que se está modelando.

Este proceso consta de una iteración cíclica que comienza por convertir hechos propios del dominio en formas visuales. Luego, se describen los hechos en un lenguaje de modelado bien definido. Estas formas visuales se mapean a una ontología, para poder posteriormente razonar sobre ella. Finalmente, se muestran los resultados asociándolos al mismo lenguaje gráfico.

Como se muestra en la Figura 3.1, el proceso se divide en tres niveles: nivel de datos, de conceptos y lógico. El primero, incluye los hechos ontológicos extraídos de los datos del dominio a ser representado en un modelo conceptual. El segundo, es el nivel conceptual que envuelve a los usuarios en la obtención y exploración de conceptos de las diferentes formalizaciones lógicas

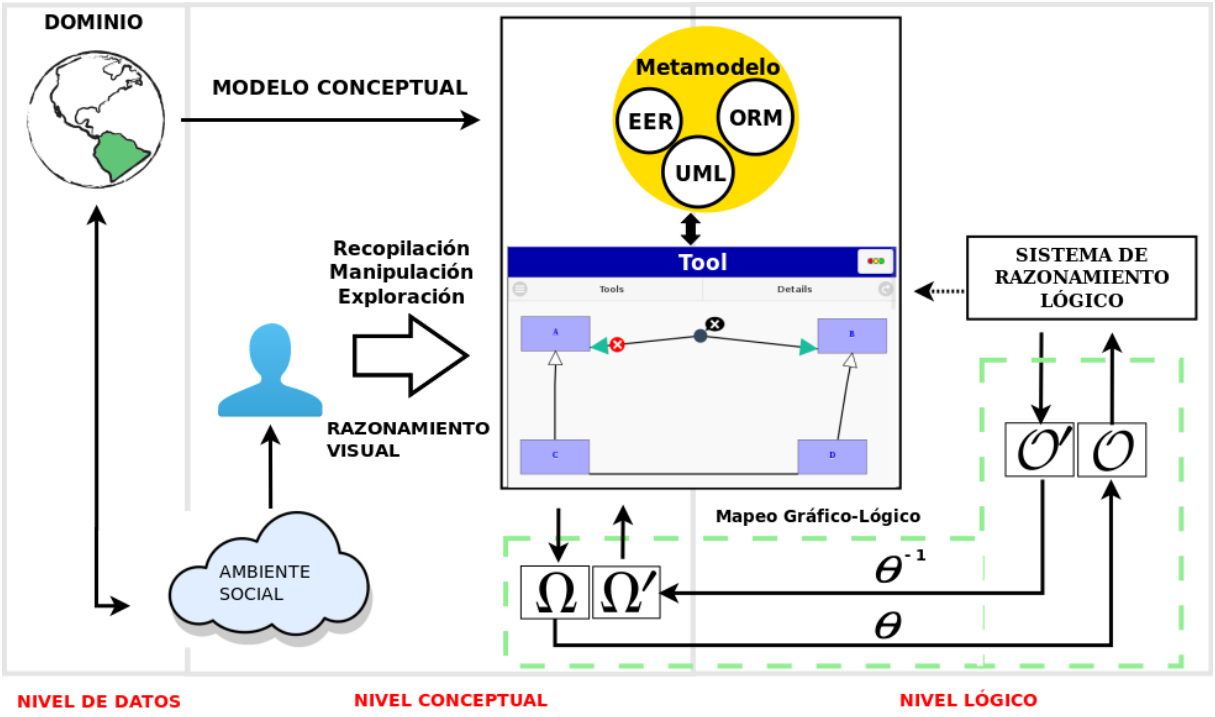


Figura 3.1: Esquema del Proceso de Visualización para Modelado Conceptual Ontológico.

para representar los diagramas, siendo estos manipulados por motores gráficos. Tercero, el nivel lógico incluye el mapeo gráfico-lógico aportando un soporte formal para guiar el proceso.

El núcleo del proceso se encuentra en el mapeo gráfico-lógico, cuyo objetivo es coordinar las diferentes formas de codificar las primitivas gráficas en un formalismo lógico decidible. La función θ se define como la unión entre las representaciones lógicas que codifican cada elemento gráfico. Por medio de Ω y Ω', y sus respectivas ontologías $\mathcal{O}$ y $\mathcal{O}'$ a través de θ y θ^{-1}, se define la integración del soporte gráfico con razonamiento mostrando los resultados de este último en la misma notación visual. Por ejemplo, Ω podría ser una representación gráfica de una ontología en un lenguaje de modelado conceptual, supóngase UML, θ corresponde con un mapeo de la ontología UML a un lenguaje formal como Lógica Descriptiva. $\mathcal{O}$ en este caso es la traducción del modelo gráfico UML Ω en DL. Una vez que se llama al razonador, éste devuelve una nueva formalización en DL con posibles cambios, que denominamos $\mathcal{O}'$. En el proceso inverso, la formalización en DL se traduce a su representación gráfica UML, por medio de θ^{-1}, resultando en un nuevo modelo gráfico UML representado por Ω'.

crowd fue desarrollado teniendo como base el Proceso de Visualización incorporándolo dentro de las guías básicas propuestas para este último. Asimismo, el diseño de la arquitectura y sus diferentes módulos corresponden con sus distintas etapas, los cuales serán descriptos en las secciones sucesivas, resultando así en una implementación posible. *crowd* es una herramienta necesaria para llevar a cabo los tres niveles planteados como receptor de los hechos, como herramienta visual para mostrar la representación de estos diagramas y como herramienta para realizar el mapeo gráfico-lógico final.

Además, el proceso describe la utilización de un metamodelo Keet-Fillottrani (KF metamodel) orientado a ontologías [65, 42, 41]. Éste, está diseñado y formalizado para permitir diferentes vistas del dominio por medio de reconstrucciones lógicas y aserciones intermodelos. Su intención es la de proveer la interoperabilidad, integración y conversión de modelos de datos conceptuales representado en diferentes lenguajes. Para ello, especifica una aproximación para transformar un modelo de un lenguaje a otro de manera que semánticamente permita crear vínculos apropiados entre ellos. Actualmente, esta especificación unifica UML, EER y ORM 2, que aunque aparentan

ser similares, poseen sus distinciones semánticas los cuales son contempladas por el metamodelo KF. La formalización del metamodelo se describe en Lógica de Primer Orden y/o DL para unificar las entidades estructurales y sus limitaciones de los lenguajes.

3.1.1. Modelado Conceptual Asistido por Razonamiento Automático

El nivel lógico del proceso descripto nos brinda la posibilidad de realizar razonamientos cuyos resultados pueden afectar al modelo conceptual, si el usuario así lo permitiese.

Los modelos conceptuales utilizados a escala industrial pueden resultar largos y complejos de diseñar, analizar y mantener. La expresividad de los lenguajes gráficos pueden llevar a consecuencias implícitas que no son detectadas por el diseñador en diagramas complejos, causando así varias formas de inconsistencias o redundancias. Esto resulta en una degradación en la calidad del diseño y/o en el incremento del tiempo y del costo. En especial, si estos modelos son utilizados como parte de un desarrollo de tipo Model-Driven, la calidad de los modelos pueden influenciar a la de los sistemas implementados (especialmente, si se utiliza un generador de código o se usan los modelos para generar casos de tests) [19].

No sólo la búsqueda de inconsistencias a través del razonamiento es el principal fundamento de la utilización de una formalización en el modelo, sino también, y como se expresa en [82], la posibilidad de compartir la estructura de la información entre personas o agentes de software; reusar el dominio; hacer las asunciones del dominio explícitas; separar el conocimiento del dominio del conocimiento operacional y analizar el conocimento del dominio.

3.2. Diseño de la Arquitectura

Previamente se subrayó la importancia de herramientas que provean de soluciones de calidad para asistir a los desarrolladores en el diseño de ontologías a nivel conceptual. Por ello, se incorpora un soporte lógico y de razonamiento automático, el cual ayudará en el proceso de modelado, así se pueden establecer criterios de calidad claros y medibles. En nuestro caso, explotaremos el criterio de la coherencia del modelo, generando una codificación formal del modelo gráfico proveído por el usuario a un razonador, con las consultas necesarias que representarán la calidad deseada. El diseño que presentamos es expansible en varios aspectos importantes, como por ejemplo, se espera a futuro brindar soportes a varios lenguajes gráficos, distintas alternativas de razonadores, varias posibilidades de traducción a DL y varias estrategias de consultas posibles.

Para el diseño se propone una arquitectura cliente-servidor presentada en la Figura 3.2. Ésta, se realizó teniendo en consideración la posibilidad de utilizar tecnologías Web como servidores HTTP, lenguajes HTML, CSS y Javascript que serán detallados en el capítulo 4. Dicha arquitectura incluye una interfaz de usuario para la cual se podrán definir nuevas primitivas gráficas de modelado o se utilizarán primitivas fácilmente reconocibles por el usuario (como por ejemplo: aquellas provenientes de los diagramas de clases UML [21], o correspondientes a EER [45]).

También, se incluye un módulo de traducción a un lenguaje formal, cuyo objetivo es la representación de los modelos conceptuales a una codificación que pueda ser procesada por una herramienta razonador. Hemos optado como lenguaje formal a las Lógicas Descriptivas debido a la semántica precisa y sus diversos niveles de expresividad. La utilización de lógicas más complejas, como la Lógica de Primer Orden ($\mathcal{FOL}$), es excluida de este trabajo ya que las típicas formas de razonamiento usadas en representaciones basadas en estructuras pueden llevarse a cabo usando fragmentos de la expresividad de $\mathcal{FOL}$, reduciendo la complejidad computacional, conclusión que surge a partir de las representaciones usando Frames ó Redes Semánticas puesto que requieren fragmentos de esta lógica [9, 22].

El servicio de razonamiento se lleva a cabo cargando al razonador con la representación formal del modelo, y posteriormente, realizando una serie de consultas generadas estratégicamente por su correspondiente módulo. Éste requiere del modelo original para determinar cuáles consultas

generar, puesto que estas pueden depender de las primitivas usadas por el usuario y las relaciones entre ellas. Las respuestas generadas por parte del razonador deben ser procesadas dependiendo de las consultas introducidas al razonador. Por consiguiente, el módulo encargado de procesar la salida del razonador estará relacionado al módulo generador de consultas para llevar a cabo la interpretación de la estrategia utilizada al momento de crear las consultas. Por ejemplo, si nuestra intención es determinar si el modelo ofrecido por el usuario tiene una posible instancia, nuestra estrategia puede consistir en consultar al razonador si cada concepto de la representación formal en DL asociado a una primitiva es consistente. Para ello, se generará una consulta de consistencia por cada primitiva y, cuando el razonador las procese, se deberá determinar si existe una respuesta negativa.

La arquitectura presentada permite definir un proceso de verificación de los modelos conceptuales creados por el usuario. Cuando el razonamiento es invocado, la traducción a lógica descriptiva de los modelos es enviada al razonador junto con un conjunto de consultas generadas por el módulo pertinente. A continuación, el sistema mostrará al usuario todas las deducciones relevantes modificando la apariencia del diagrama gráfico original y, dejando a éste, la decisión de preservar o descartar dichos cambios.

El usuario interactúa con un *front-end* que es ejecutado en el explorador Web, el cual provee un ambiente gráfico con las funcionalidades necesarias para crear y editar ontologías junto con un conjunto de primitivas gráficas. Además, muestra las respuestas del razonador usando el mismo conjunto de representaciones gráficas e incluso de forma textual. El *back-end* se ejecuta del lado del servidor, y se compone de módulos que traducen la representación gráfica a una ontología, que generan consultas al sistema de razonamiento, que razona sobre las especificaciones del modelo y además procesa las salidas.

La posibilidad de utilizar razonamientos automáticos es permitido por una definición semántica precisa de todos los elementos de los diagramas de clases y, por consecuencia, las limitaciones de los diagramas son internamente traducidas en un formalismo lógico basado en clases, capturando las características típicas de los modelos de datos conceptuales.

Más aún, puesto que *crowd* está basado en una noción de completa deducción para un soporte de razonamiento relativo a la sintaxis de los diagramas de clases, el usuario observará la ontología original de forma gráfica y completa con todas las deducciones que tienen sentido, dado lo proveído por la ontología y lo expresado en el lenguaje de diagrama de clases gráfico mismo. Esto incluye chequear clases y relaciones por consistencias, descubrir clases implicadas y limitaciones de cardinalidad. Otras limitaciones no gráficas pueden ser modeladas en *crowd*, pero deben ser escritas en forma textual, por ejemplo, utilizando expresiones en OWLlink [72] o en $\mathcal{DLR}$ [26].

Por último, *crowd* solamente se enfoca en el modelado gráfico de un esquema ontológico, mientras que no considera a los individuos.

3.2.1. Cliente

Del lado del cliente se propone el uso de una librería gráfica que es controlada por una serie de módulos que se basan en un estilo arquitectónico denominado Modelo-Vista-Controlador. En el modelo se representa en forma estática las distintas primitivas gráficas posibles a realizarse. De esta forma, estos objetos pueden responder a diversos métodos afectando a una o a un conjunto de primitivas en el modelo e incluso puede realizarse una traducción textual para ser enviada al servidor. La vista, posee aspectos visuales de la primitiva, definiendo su color, forma, fuentes del texto, etc. El controlador, si éste existiera para la vista y el modelo dado, respondería a las acciones del usuario y a la entrada de datos del mismo, como por ejemplo, las acciones a realizarse ante un evento del usuario sobre una primitiva.

Además de esto, se debe proveer del código necesario para enviar al servidor la información que requiere para que éste determine la satisfacibilidad y luego para analizar sus respuestas.

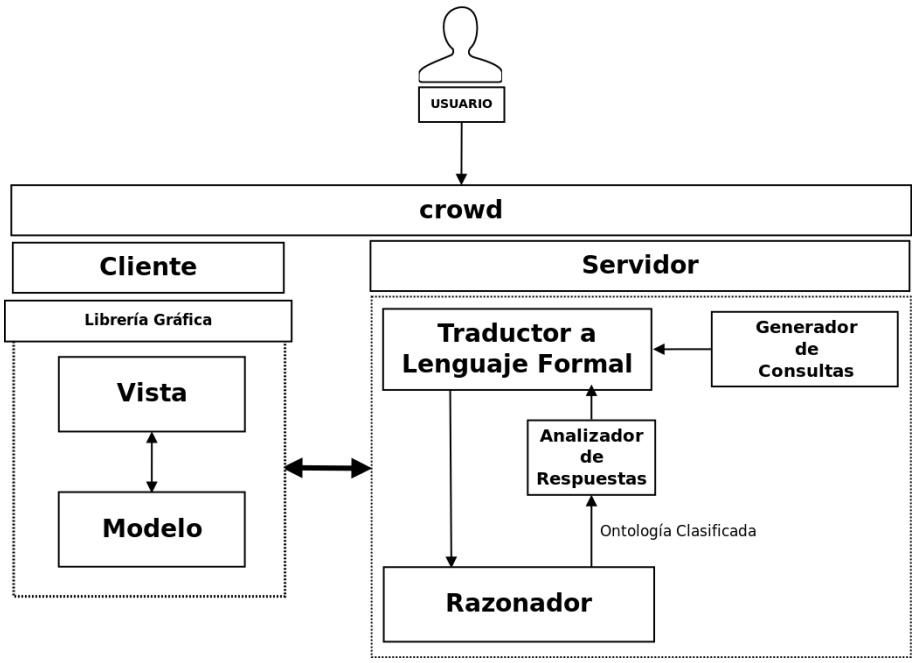


Figura 3.2: Arquitectura de *crowd*.

3.2.2. Servidor

En cuanto al servidor, su funcionamiento básicamente consiste en estar a la espera de órdenes por parte del cliente. Para esto, debe proveer una API (Interfaz de Programación de Aplicaciones) que permita recibir las distintas instrucciones, en especial, recibir el modelo conceptual, determinar qué acción realizar en él y emitir la respuesta esperada, todo bajo un formato predeterminado consensuado entre ambas partes. Consecuentemente, es preciso dividir al servidor en una serie de módulos cuyos roles en conjunto lograrán llevar a cabo las solicitudes del cliente.

A continuación, se enumeran los distintos módulos necesarios y sus respectivas funciones:

`Traductor a DL:` Provee de la representación en Lógica Descriptiva bajo un lenguaje computable (por ejemplo, en OWL 2 [84]) desde el modelo de entrada, que luego será enviada al módulo razonador.

En nuestro relevo, nos hemos encontrado con varios métodos para formalizar el modelo conceptual de entrada a Lógica Descriptiva, incluso para un mismo lenguaje gráfico. Además, hemos considerado que existen varias representaciones textuales para un mismo lenguaje computacional que describe a la misma ontología(por ejemplo: una ontología OWL 2 puede expresarse utilizando Turtle [17], RDF [86, 16] e incluso con una sintaxis funcional [80]).

Para que la arquitectura pueda ser extendida bajo estas consideraciones, hemos realizado un diseño, basado en dos patrones denominados "Builder" y "Strategy" [44, 46], presentado en la Figura 3.3. Cada una de las subclases de "AbstractStrategy" implementan un método de formalización determinado, donde su salida es una secuencia de instrucciones con la Lógica Descriptiva a generar. Para las distintas sintaxis y protocolos, se utiliza una subclase de "DocumentBuilder" la cual recibe las instrucciones provenientes de la estrategia seleccionada y genera el documento apropiado. Por ejemplo, "XMLOWLBuilder" permite la generación de un documento de sintaxis XML con la lógica generada por una de las estrategias bajo las etiquetas definidas por el estándar OWL. El proceso completo se explica en la sección 4.4 y puede observarse su diagrama de secuencia detallado en la Figura 4.6.

`Generador de Consultas:` Aporta una serie de consultas que serán ejecutadas en el razonador sobre el modelo inicial. Estas consultas permiten obtener propiedades de la ontología, ya

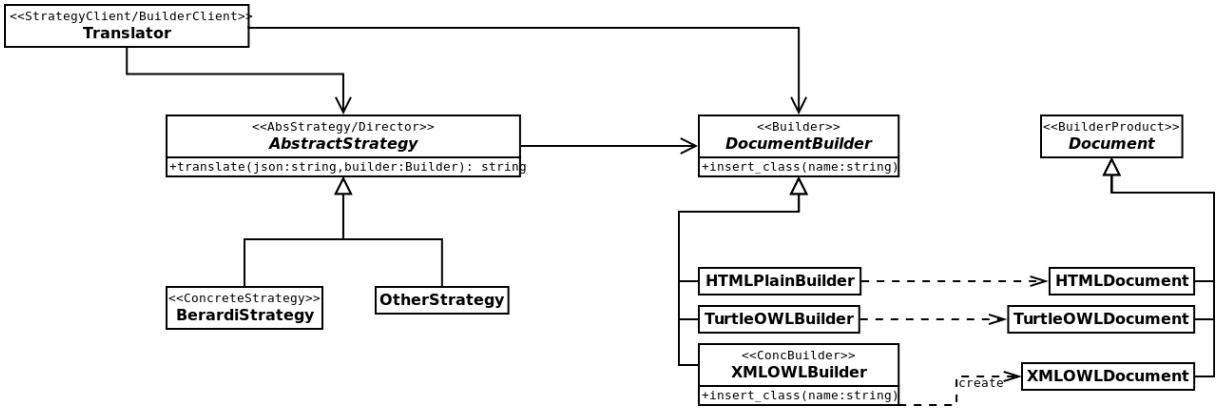


Figura 3.3: Diseño del módulo traductor.

incorporados en el razonamiento. Diferentes conjuntos de consultas pueden ser configurados en este componente dependiendo de la metodología o el servicio invocado desde el *front-end*. Sin embargo, sólo describiremos qué consultas deben realizase al razonador y cómo generarlas para detectar inconsistencias en las clases UML [21] del modelo ingresado.

Las consultas son generadas como instrucciones que serán incorporadas a la traducción dependiendo de la subclase "DocumentBuilder" seleccionada por el módulo `Traductor a DL`. En el diagrama de secuencia de la Figura 3.4 se observa en qué momento se realiza la inclusión de estas consultas para que el razonador pueda recibirlas y procesarlas.

`Razonador:` El razonamiento basado en lógica realizado sobre el modelo, el cual está compuesto por el modelo del usuario junto con el conjunto de consultas agregados por el módulo previo, es ejecutado por un razonador, descripto en la arquitectura como el módulo `Razonador`. El resultado en conjunto con el modelo y su traducción a DL, al que denominaremos como "ontología clasificada", son una serie de respuestas obtenidas a partir de las conclusiones realizadas por el razonador y su entrada. Existen varios programas razonadores disponibles en la actualidad los cuales, en su mayoría, se comunican por medio de uno o varios de los protocolos estandarizados existentes como DIG [14] y OWLlink [72]. Ejemplos de estos razonadores son Pellet [96], RACER [51], konclude [98], entre otros [102, 77, 95].

Este módulo se encarga de activar estos programas apropiadamente y configurarlos para poder enviar y recibir los datos por cada razonador que se desee utilizar. Para ello, se diseñó una clase denominada "Runner" con una interfaz simplificada. Dicha clase debe ser configurada con el razonador que se desee utilizar por medio de una de las subclases de "Connector". Esta última es la que se encarga de inicializar el razonador y de los detalles de comunicación de interproceso necesarios para enviar y recibir información con él. La Figura 3.5 muestra el diagrama de clases realizado para este módulo.

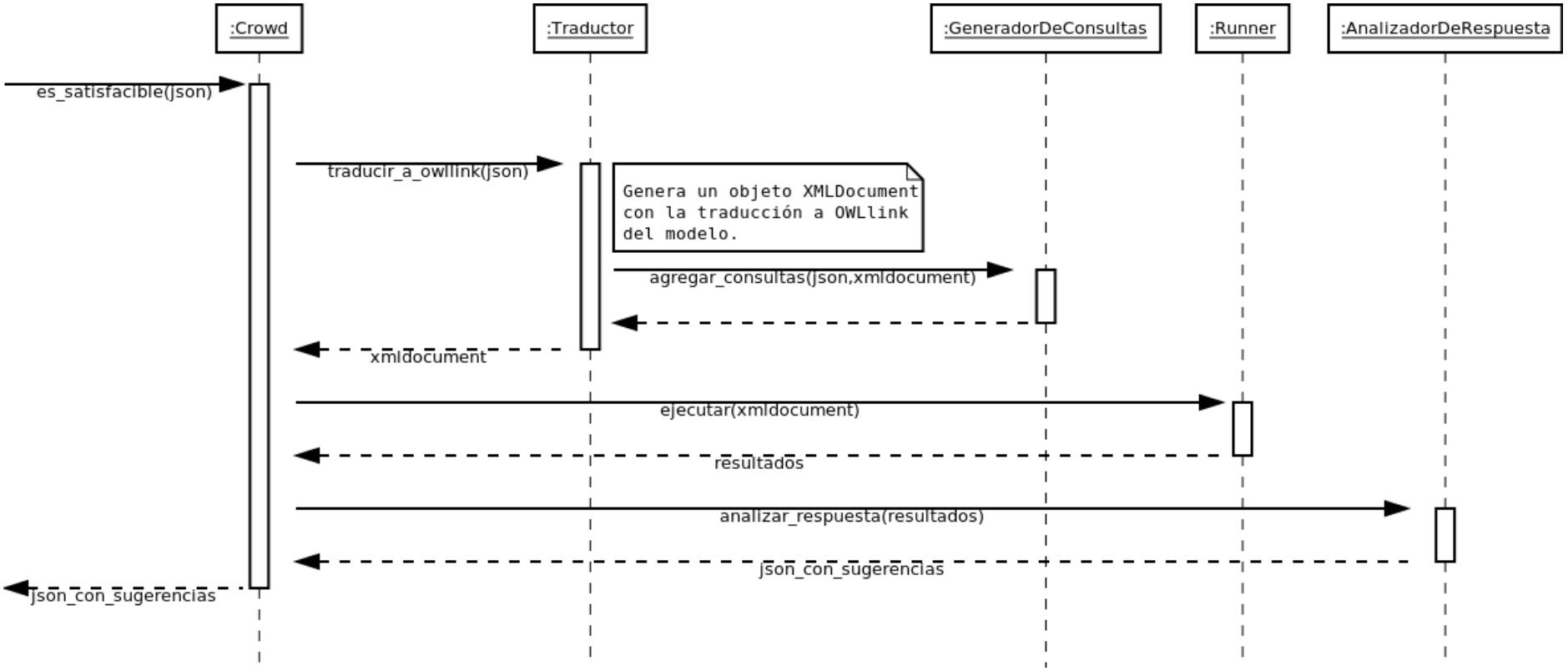


Figura 3.4: Diagrama de secuencia que muestra el servicio de razonamiento para determinar la satisfacibilidad del modelo.

`Analizador de Respuestas:` Finalmente, la ontología clasificada es usada como entrada para el módulo `Analizador de Respuestas`, el cual procesa la salida del razonador para proveer al usuario de las conclusiones obtenidas de esta información en un formato entendible por el cliente.

Existe una relación entre éste y el módulo `Generador de Consultas`, debido a que en la implementación se debe programar al sistema con una estrategia generadora de consultas compatible con el analizador de respuesta. Además, es preciso que el analizador comprenda la salida de razonador. Primero, para buscar las respuestas y relacionarlas con cada una de las consultas realizadas. Segundo, para que la implementación pueda interpretar estas respuestas y generar la salida esperada por el cliente.

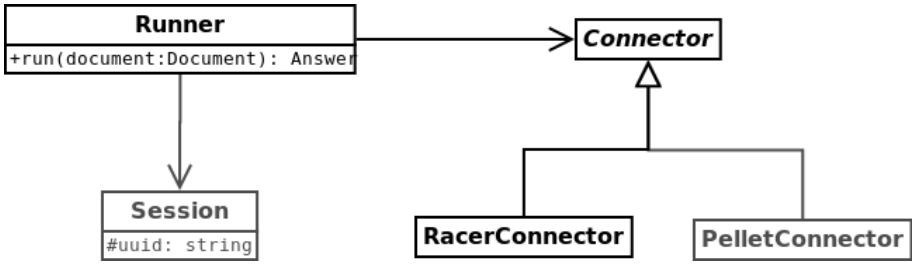


Figura 3.5: Diseño del módulo Razonador para soportar varios tipos de programas razonadores.

3.2.3. Servicios de Razonamiento

El servicio de razonamiento, para determinar la satisfacibilidad del modelo, comienza a partir de una petición del usuario desde la interfaz gráfica. Luego, el cliente traduce el modelo conceptual a una representación intermedia que pueda ser enviada al servidor para su posterior procesado. Esto se realiza por medio de la interacción entre los objetos del modelo dentro de la arquitectura modelo-vista-controlador, el cual producirá el código intermedio del diagrama, por ejemplo, recorriendo todos los elementos del modelo de las primitivas gráficas y solicitando a estos objetos su representación intermedia.

Desde el lado del servidor, el `Traductor a DL` recibe esta representación y comienza su traducción a una Lógica Descriptiva por medio de una codificación determinada (por ejemplo, la codificación UML a $\mathcal{ALCQI}$ propuesta en [19]), el cual generará reglas y axiomas de dicha lógica a partir de cada primitiva gráfica y su semántica. Esta representación debe ser conocida y computable por el `Razonador`. A continuación, el módulo `Generador de Consultas` producirá una serie de consultas entendibles por el razonador, la forma y la cantidad de éstas dependen del modelo conceptual ofrecido por el usuario y deben cumplir con el objetivo de chequear si la base de conocimiento en conjunto con las primitivas del modelo son consistentes y satisfacibles. En caso de que no lo sean, se determina dónde se encuentra el problema en el modelo.

En la Figura 3.4, se representa de forma parcial y por medio de un diagrama de secuencia el proceso descripto.

Capítulo 4

Implementación de un Prototipo de *crowd*

El prototipo de *crowd* fue implementado basándonos en tecnología web. La elección de los lenguajes a utilizar corresponde a la funcionalidad de los módulos donde se utilizan, y son los siguientes:

- Del lado del cliente, utilizamos HTML, CSS, y Javascript. Estos lenguajes son los estándares para realizar páginas web con ciertos efectos y nos permiten aplicar la técnica AJAX, la que posibilita enviar al servidor el diagrama de forma asincrónica y de fondo, permitiendo al usuario seguir trabajando. JSON es el lenguaje utilizado para codificar el modelo y transmitirlo, como así también para recibir las respuestas del servidor; es de fácil lectura y altamente compatible con Javascript.

  Asimismo, decidimos utilizar el lenguaje CoffeeScript, ya que permite el desarrollo orientado a objetos utilizando clases en vez de objetos prototipados, sin perder la alta compatibilidad con Javascript, puesto que sus implementaciones proveen de un compilador que traduce los códigos desarrollados a éste último. La facilidad y agilidad que provee sumado a diversas soluciones ofrecidas por éste, permite trabajar en un código más limpio y entendible capaz de implementar los diseños UML que realizamos para nuestro cliente.

  Además, se brindará al usuario de un subconjunto de primitivas de los diagramas de clases UML para que pueda elaborar su modelo conceptual gráfico. Para ello, es preciso utilizar una biblioteca gráfica implementada en los lenguajes de programación mencionados. En la sección 4.1 se presenta una descripción de la biblioteca JointJS, como así también una comparación de varias bibliotecas gráficas estudiadas.

- En el servidor, los diversos módulos se programan utilizando el lenguaje PHP usando los nombres de espacios ("*namespaces*") para distinguirlos. El `Razonador` se divide en dos partes: el razonador propiamente dicho el cual es un programa separado que puede comprender el protocolo OWLlink y la interfaz que permite ejecutar el programa, inicializarlo y entregarle la información. Actualmente, se utiliza el razonador RACER [51] aunque puede utilizarse cualquier otro si se modifica la interfaz adecuadamente.

En las secciones siguientes presentamos la biblioteca gráfica seleccionada para el cliente. Luego, se detallará el cliente para que mantenga una representación interna del modelo conceptual del usuario. En la sección 4.3, se describe los protocolos usados, la API del servidor expuesta al cliente y los protocolos disponibles actualmente para la interacción con el razonador. En las secciones 4.4, 4.5 y 4.6 se explica cómo fueron implementados los módulos del servidor, sus entradas y salidas y las consideraciones que hemos tenido en cuenta. Finalmente, en la sección 4.7 se muestra un ejemplo con un modelo conceptual satisfacible y una modificación del mismo para hacerlo insatisfacible, mostrando así el funcionamiento del prototipo para ambos casos.

4.1. Biblioteca Gráfica

Las características del cliente de *crowd* son soportadas por una biblioteca gráfica realizada en Javascript denominada JointJS[1], cuyo principal objetivo es el de proveer instrucciones sencillas para dibujar las primitivas y, actualmente, es utilizada en herramientas relacionadas para visualización de datos como [81]. Una revisión preliminar acerca de las diferentes bibliotecas gráficas y sus características demostró que este tipo de bibliotecas utilizan una de las dos principales tecnologías de dibujo denominadas Canvas y SVG, y por medio del uso de estos comandos básicos puede realizar diagramas como los de UML. También, pueden gestionar los eventos del usuario tales como el *drag-and-drop*, entre otros.

En particular, JointJS presenta algunas ventajas claves: se enfoca en el dibujo de primitivas gráficas, provee *plug-ins* para UML, EER, etc. y brinda la posibilidad de crear nuevos de forma simple. Asimismo, usa la biblioteca Backbone.js para dar estructura a la aplicación Web. Backbone.js es una biblioteca Javascript que asiste en el desarrollo de aplicaciones Web proveyendo una arquitectura Modelo-Vista-Controlador del lado del cliente.

Otras bibliotecas como Raphael[2], Processing[3], p5js[4] proveen una interfaz más simple para dibujar primitivas como curvas y figuras geométricas, pero no otorga al desarrollador una API dedicada al dibujo de primitivas de forma directa desde el punto de vista que nosotros consideramos para dar soporte al lenguaje de modelado conceptual. Finalmente, jsPlumb[5] posee otro enfoque asistiendo al dibujo desde distintos tipos de conexiones entre elementos pero no provee primitivas para la creación de diagramas.

4.2. Vista y Modelo del Cliente

Basándonos en la arquitectura propuesta por Backbone.js y utilizando Coffeescript hemos desarrollado dos módulos del lado del cliente, la `Vista` que posee la representación visual del diagrama y el `Modelo` que contiene una representación abstracta de las primitivas aceptadas por la interfaz.

Dentro del aspecto visual, se consideran retazos de interfaces como el botón del semáforo, los paneles y demás elementos que utilizan la API de Backbone.js para poder funcionar.

Para el modelo, se ha realizado un diseño, mezclando los patrones "Abstract Factory" y "Observer" [44, 46], que permite detallar las distintas primitivas gráficas que el usuario puede manipular considerando la posibilidad de utilizar otros lenguajes gráficos, como ORM ó EER, e incluso se pueden mezclar las de varios lenguajes en un mismo diagrama. La Figura 4.1 muestra el diagrama de clases que fue implementado utilizando Coffeescript, mostrando la posibilidad de utilizar varias primitivas de varios lenguajes proveídas por JointJS, por ejemplo las clases poseen un comportamiento similar para las de UML (`UMLClass` o `joint.shapes.uml.Class` en JointJS) con respecto a las Entidades EER. En el lado derecho de la figura, se observa las subclases de `Factory` que permiten la creación de estas primitivas.

4.3. Protocolos de Comunicación

El servidor expone al cliente una sencilla API basada en el el protocolo HTTP accesible por medio de direcciones URLs. Éstas pueden ser utilizadas con fines de desarrollo y debuging por medio de bibliotecas o de programas que pueden realizar solicitudes GET y PUT (por ejemplo: `wget` y `curl`).

[1] http://www.jointjs.com/
[2] http://raphaeljs.com/
[3] http://processingjs.org
[4] http://p5js.org
[5] https://jsplumbtoolkit.com/

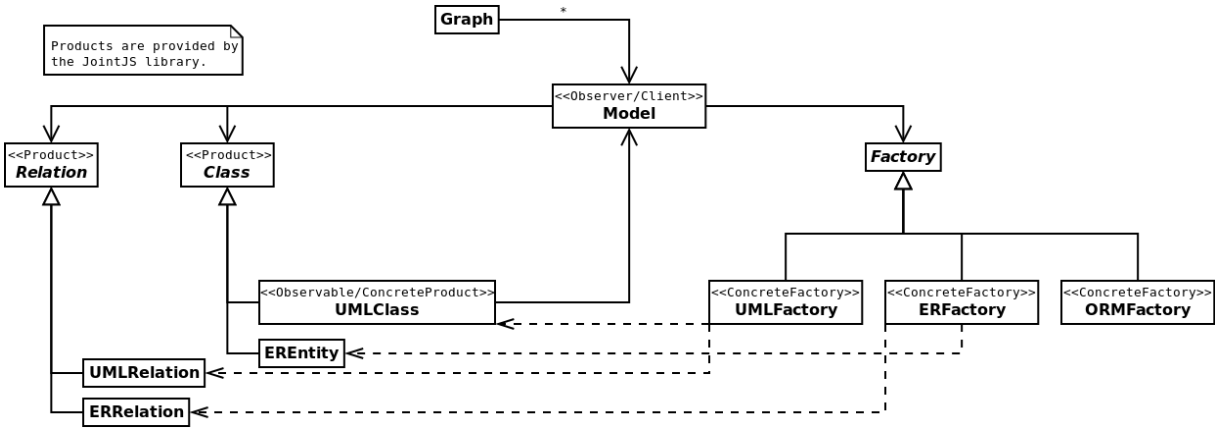


Figura 4.1: Diseño UML que podrá brindar soporte a más de un lenguaje gráfico en el Front-End.

La Figura 4.2 muestra dos sufijos de URL de dicha API: `/query.php` para realizar una consulta de satisfacibilidad y `/translators/TRANSLATOR.php` para solicitar una codificación del modelo a un lenguaje formal. Las consultas, requieren como parámetro el modelo en formato JSON. Luego, un script en PHP, determinado por el sufijo, se activa para trasladar la información necesaria al módulo del servidor necesario.

Los razonadores que hemos relevado aceptan como entrada textos por medio de un archivo, una tubería o un socket escritos bajo protocolos estándares. Actualmente, los dos protocolos más importantes son DIG y OWLlink. Ambos, están basados en HTTP y XML y pueden usarse tanto para brindar la ontología como para realizar consultas acerca de ella. A continuación, se describirán estos protocolos.

4.3.1. Protocolo DIG

La interfaz DIG provee un acceso uniforme a los razonadores de lógica descriptiva definiendo un protocolo simple basado en HTTP PUT/GET, en conjunto con esquemas XML, que describen un lenguaje conceptual y las operaciones que lo acompañan.

La especificación de DIG 1.1, publicada en el año 2003 en [14], ofrece una expresividad que no es suficiente para capturar las ontologías OWL-DL en general. El lenguaje conceptual está basado en $\mathcal{SHOIQ}\mathcal{D}_n^-$, el cual consiste en operadores booleanos estándares ($\sqcap, \sqcup, \neg$), restricciones cuantificadas universales y existenciales, limitadores de cardinalidad, jerarquía de roles, roles inversos, el constructor "uno-de" ("one-of") y dominios concretos.

Herramientas, como por ejemplo ICOM [39], utilizan únicamente este protocolo para comunicarse con el razonador. DIG se encuentra discontinuado al día de la fecha haciendo que estas herramientas queden sin soporte para los nuevos razonadores.

En la Figura 4.3 donde se muestra, a modo de comparación, su representación en OWL. Más detalles se presentan en un caso de uso en [34].

Formato de los Mensajes

La comunicación con el razonador se realiza por medio de mensajes codificados como funciones `TELL` y `ASK`.

La sintaxis para la petición `TELL` contiene varios elementos `tells` que en sí mismos consisten en un número de enunciados "tell", los cuales son monotónicos (información que una vez proveída a la base de conocimiento no puede ser retractada ni removida). Deben ser realizadas en el contexto de una base de conocimiento particular y el orden de estos enunciados no es importante.

En el caso de las peticiones `ASK`, éstas contienen varios elementos `asks`, los cuales se conforman de un número de enunciados "ask", donde cada uno posee un atributo identificatorio para una

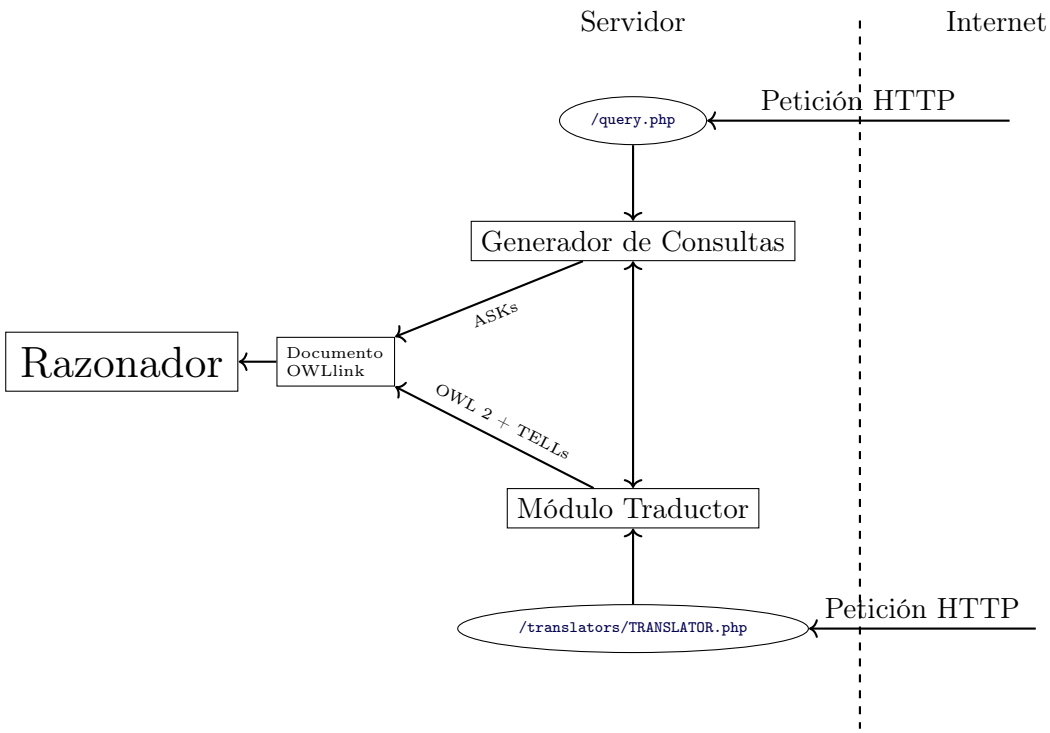


Figura 4.2: Diseño general del *back-end* realizado en PHP.

colección de preguntas, permitiendo la presentación de múltiples consultas en una sola solicitud.

El esquema XML que detalla a DIG, incluye una descripción de las respuestas esperadas por cada petición `ASK`. Éstas, contienen un elemento `response`, que a su vez posee un número de respuestas (una por cada consulta en la petición `ASK` realizada). Cada respuesta particular posee un identificador que coincide con el de la consulta realizada.

4.3.2. Protocolo OWLlink

OWLlink provee de una interfaz declarativa para razonadores basados en OWL. Se propone en [72] como una evolución de la interfaz DIG 1.1, que en su borrador [15] fue originalmente presentado como DIG 2.0 pero posteriormente renombrado como OWLlink, para reflejar la importancia del cambio. Su intención es la de proveer a las aplicaciones clientes con un mecanismo liviano para acceder a las funciones de razonamiento proveídas por un servidor. Por estas razones, se utiliza este protocolo para la comunicación entre el razonador y el código PHP del servidor de *crowd*, enviando y recibiendo documentos OWLlink con la representación del modelo en DL y las consultas necesarias para determinar su consistencia y obteniendo las respuestas en el mismo formato. De hecho, los módulos `Traductor a DL` y `Generador de Consultas` recrean la sintaxis del protocolo para generar una representación formal y computable.

Aunque la sintaxis y semántica de las primitivas de OWLlink están basadas en OWL 2 [84], los esfuerzos para dar soporte al núcleo del protocolo con respecto al parseo es reducido. Consecuentemente, OWLlink hereda todos los conceptos del lenguaje subyacente y las nociones de equivalencias estructurales de OWL 2. Sin embargo, no soporta ninguna de las partes de OWL 2 más allá de los niveles de axiomas.

Además, la especificación de OWLlink no resuelve los inconvenientes referidos a la conexión, transacción, autenticación, cifrado, compresión, concurrencia y múltiples clientes, entre otros. Pero, algunas de estas características pueden ser proveídas transparentemente por protocolos de accesos [72].

La estructura sintáctica de OWLlink se basa en un documento XML que debe cumplir con el

RDF/XML

```
<owl:Class rdf:ID=''A''>
  <owl:complementOf>
    <owl:Class rdf:ID='B'' />
  </owl:complementOf>
  <rdfs:subClassOf>
    <owl:Class rdf:ID='C'' />
  </rdfs:subClassOf>
</owl:Class>
```

DIG

```
<?xml version="1.0"?>
<tells
  xmlns=http://dl.kr.org/dig/lang
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:schemaLocation="http://dl.kr.org/dig/lang
    http://potato.cs.man.ac.uk/dig/level0/dig.xsd">
  <defconcept name="http://example.org/foo#A" />
  <defconcept name="http://example.org/foo#B" />
  <defconcept name="http://example.org/foo#C" />
  <impliesc>
    <catom name="http://example.org/foo#A" />
    <catom name="http://example.org/foo#C" />
  </impliesc>
  <equalc>
    <catom name="http://example.org/foo#A" />
    <not>
      <catom name="http://example.org/foo#B"/>
    </not>
  </equalc>
</tells>
```

Figura 4.3: Ejemplo de DIG y su representación en OWL.

esquema disponible en la URL `https://www.w3.org/Submission/owllink-httpxml-binding/owllink.xsd`. Básicamente, este esquema detalla algunos elementos de la Figura 4.4, dónde deben estar y qué otras etiquetas y propiedades pueden o deben incluir. Existen dos tipos de etiquetas raíz: `RequestMessage` para enviar información y para realizar consultas a un razonador y `ResponseMessage` que se utiliza en la salida del razonador para contener las respuestas de las operaciones solicitadas. Es necesario incluir a éstas etiquetas la propiedad `xsi:schemaLocation` con la URL donde se encuentra el esquema de OWLlink. En el caso del documento dado al razonador, típicamente se le solicita crear una o varias bases de conocimientos por medio de la etiqueta `CreateKb` asignándoles una URL como identificador, luego se le envía una ontología incorporándola dentro de una etiqueta `Tell`, se realizan consultas y, si la KB ya no es necesaria, se solicita que se la libere con la etiqueta `ReleaseKB`. En el Apéndice A, en la Figura A.6 y en la Figura A.7, se muestra un template de la estructura de la consulta descripta y un documento de respuesta respectivamente.

Tell y Aks

OWLlink se basa en primitivas del lenguaje OWL 2 en lo que respecta a la representación de la ontología dentro de las peticiones `TELL`. En la Figura 4.4, se presenta un diagrama de clases UML con los objetos básicos soportados por el protocolo, donde los nombres de todas las clases correspondientes a OWL 2 poseen el prefijo `ox.` para indicar que no están definidos en la especificación OWLlink. La composición "axioms" que observamos en el diagrama de clase indica que uno o más axiomas OWL 2 (`ox.Axiom`) deben estar contenidos en las etiquetas `Tell`. Esto significa que el protocolo permite usar varios axiomas acerca de clases, propiedades y hechos de la especificación de OWL 2 [80], para representar la ontología en DL sobre la que el razonador debe trabajar.

OWLlink posee un conjunto de consultas generales para obtener información de las entidades de la base de conocimiento, como así también de los hechos deducibles. Las llamadas preguntas básicas ("basic ask" en inglés) sólo cubren las consultas más comunes con respecto a los axiomas dados e inferidos de la KB. OWLlink extiende considerablemente la interfaz de consulta proveída por DIG 1.1, además de estar más alineados con OWL 2. La Tabla 4.1 muestra la interfaz definida en OWLlink para consultas `ASK`s y el tipo de respuesta esperada. En la Tabla, la abreviatura "`[O|D]`" indica que existen dos instrucciones en vez de una, reemplazando con `Object` ó con `DataProperty` según corresponda.

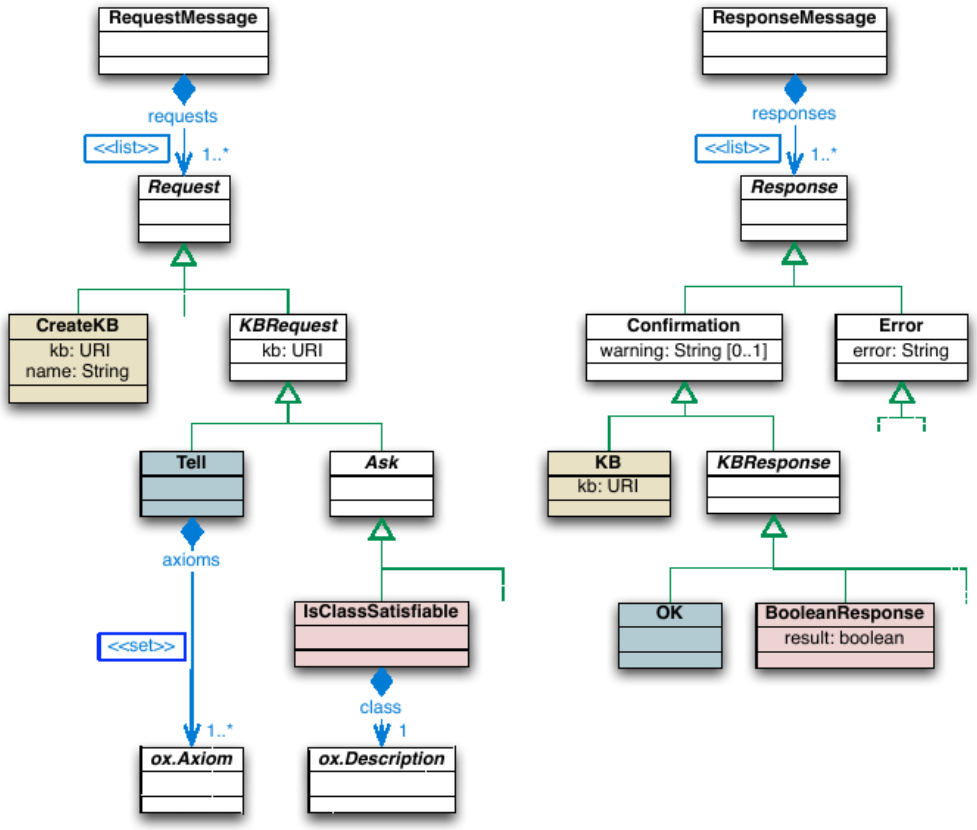


Figura 4.4: Objetos Básicos del Protoloco OWLlink [72]

En la Figura 4.5, puede observarse un código OWLlink donde se le solicita al razonador la creación de una KB, se le incorpora mediante una petición `TELL` una ontología simple de tres clases donde se cumple $A \sqsubset B$ y $B \sqsubset C$, usando una petición de tipo `ASK` se consulta si la clase A es satisfacible y finalmente se libera la KB.

4.4. Módulo Traductor

El módulo traductor se encarga de procesar un modelo realizado por el usuario y transformado a JSON por el cliente, para posteriormente obtener un documento traducido a OWLlink, que será utilizado como entrada para el razonador. Esta transformación debe ser consistente con alguna codificación existente que transforme un modelo realizado bajo el lenguaje gráfico UML a una DL, como el propuesto por [19], y que es representada por JSON y OWL 2 [56] respectivamente. El archivo OWLlink producido contendrá los comandos para inicializar una KB y los enunciados OWL 2 que representan la ontología. Luego se anexarán una serie de consultas creadas por el módulo `Generador de Consultas`. La Figura 4.2 muestra que, por medio de una solicitud HTTP del cliente, el módulo `Traductor` y el módulo `Generador de Consultas` crean el documento OWLlink para que el módulo `Razonador` lo utilice.

Asimismo, se le brinda la posibilidad al usuario de incluir más restricciones de mayor expresividad que las del lenguaje gráfico utilizando la sintaxis OWLlink. Dichas restricciones son anexadas al final de la representación formal generada por este módulo.

El módulo fue diseñado de modo tal que puedan implementarse a futuro diferentes codificaciones de UML a DL, e incluso desde otros lenguajes de modelado conceptual gráfico a DL. Asimismo, es posible expandir la implementación para producir otro formato de salida diferente al de OWLlink basado en XML, como por ejemplo un formato HTML con la representación textual

	Ask	KBResponse
Entidades de la KB	GetAllClasses	SetOfClasses
	GetAllClasses	SetOfClasses
	GetAllObjectProperties	SetOfObjectProperties
	GetAllDataProperties	SetOfDataProperties
	GetAllAnnotationProperties	SetOfAnnotationProperties
	GetAllIndividuals	SetOfIndividuals
	GetAllDatatypes	SetOfDatatypes
Estado	IsKBSatisfiable	BooleanResponse
	IsKBStructurallyConsistent	BooleanResponse
	GetKBLanguage	StringResponse
Esquema	IsClassSatisfiable	BooleanResponse
	IsClassSubsumedBy	BooleanResponse
	AreClassesDisjoint	BooleanResponse
	AreClassesEquivalent	BooleanResponse
	GetSubClasses	SetOfClassSynsets
	GetSuperClasses	SetOfClassSynsets
	GetEquivalentClasses	SetOfClasses
	GetClassHierarchy	ClassHierarchy
	Are[O|D]PropertiesEquivalent	BooleanResponse
	Is[O|D]PropertySatisfiable	BooleanResponse
	Are[O|D]PropertiesDisjoint	BooleanResponse
	Is[O|D]PropertySubsumedBy	BooleanResponse
	Is[O|D]PropertyXXX	BooleanResponse
	GetSub[O|D]Properties	SetOf[O|D]PropertySynsets
	GetSuper[O|D]Properties	SetOf[O|D]PropertySynsets
	GetEquivalent[O|D]Properties	SetOf[O|D]Properties
	Get[O|D]PropertyHierarchy	[O|D]PropertyHierarchy
Hechos	AreIndividualsEqual	BooleanResponse
	AreIndividualsDifferent	BooleanResponse
	IsInstanceOf	BooleanResponse
	GetTypes	SetOfClassSynsets
	GetEquivalentIndividuals	SetOfIndividuals
	Get[O|D]PropertyOfSource	SetOf[O|D]PropertySynsets
	GetObjectPropertyOfFiller	SetOfObjectPropertySynsets
	GetDataPropertyOfConstant	SetOfDataPropertySynsets
	Get[O|D]PropertiesBetween	SetOf[O|D]PropertySynsets
	GetInstances	SetOfIndividualSynsets
	GetObjectPropertyFillers	SetOfIndividualSynsets
	GetDataPropertyFillers	SetOfConstants
	Get[O|D]PropertySources	SetOfIndividualSynsets
	GetFlattenInstances	SetOfIndividuals
	GetFlattenObjectPropertyFillers	SetOfIndividuals
	GetFlattenObjectPropertySources	SetOfIndividuals

Tabla 4.1: Consultas ASKs posibles para OWLlink. Tabla extraída de [72].

```
<?xml version="1.0" encoding="UTF-8"?>
<RequestMessage xmlns="http://www.owllink.org/owllink#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.owllink.org/owllink#
    http://www.owllink.org/owllink-20091116.xsd">
  <CreateKB kb="http://www.owllink.org/examples/KB_1"/>
  <Tell kb="http://www.owllink.org/examples/KB_1">
    <owl:SubClassOf>
      <owl:Class IRI="http://www.owllink.org/examples/myOntology#A"/>
      <owl:Class IRI="http://www.owllink.org/examples/myOntology#B"/>
    </owl:SubClassOf>
    <owl:SubClassOf>
      <owl:Class IRI="http://www.owllink.org/examples/myOntology#B"/>
      <owl:Class IRI="http://www.owllink.org/examples/myOntology#C"/>
    </owl:SubClassOf>
  </Tell>
  <IsClassSatisfiable kb="http://www.owllink.org/examples/KB_1">
    <owl:Class IRI="http://www.owllink.org/examples/myOntology#A"/>
  </IsClassSatisfiable>
  <ReleaseKB kb="http://www.owllink.org/examples/KB_1"/>
</RequestMessage>
```

Figura 4.5: Ejemplo de código OWLlink.

y *human-readable* (legible para los humanos) de la DL resultante de la traducción. Estas ideas se reflejan en el diseño del diagrama de clases UML, presentado en la Figura 3.3, donde se muestra que del lado izquierdo están los métodos de transformación como subclases de `AbstractStrategy` y del lado derecho los diferentes formatos resultantes como subclases de `Document`.

La traducción que se realizó en *crowd* desde UML a DL se puede observar en la Tabla 2.6 y se explicó en la sección 2.3. Nótese que traducir $\mathcal{ALCQI}$ a OWL 2 consiste en representar una primitiva en una notación inmediata (ver Tabla 2.3). Así, podemos traducir cada primitiva de UML a $\mathcal{ALCQI}$ y finalmente representarla en un archivo XML en OWL 2.

La selección del tipo de estrategia a utilizar y el formato de salida se realiza creando instancias de la clase `BerardiStrategy` y `OWLlinkBuilder` respectivamente. Luego, la instancia del primero se comunica con la del segundo para darle instrucciones de qué debe incorporar en el documento XML, a su vez, éste último va generando el string OWLlink incorporando las etiquetas XML según las peticiones del constructor. En la Figura 4.6 se puede observar de forma detallada el diagrama de secuencia del proceso de traducción.

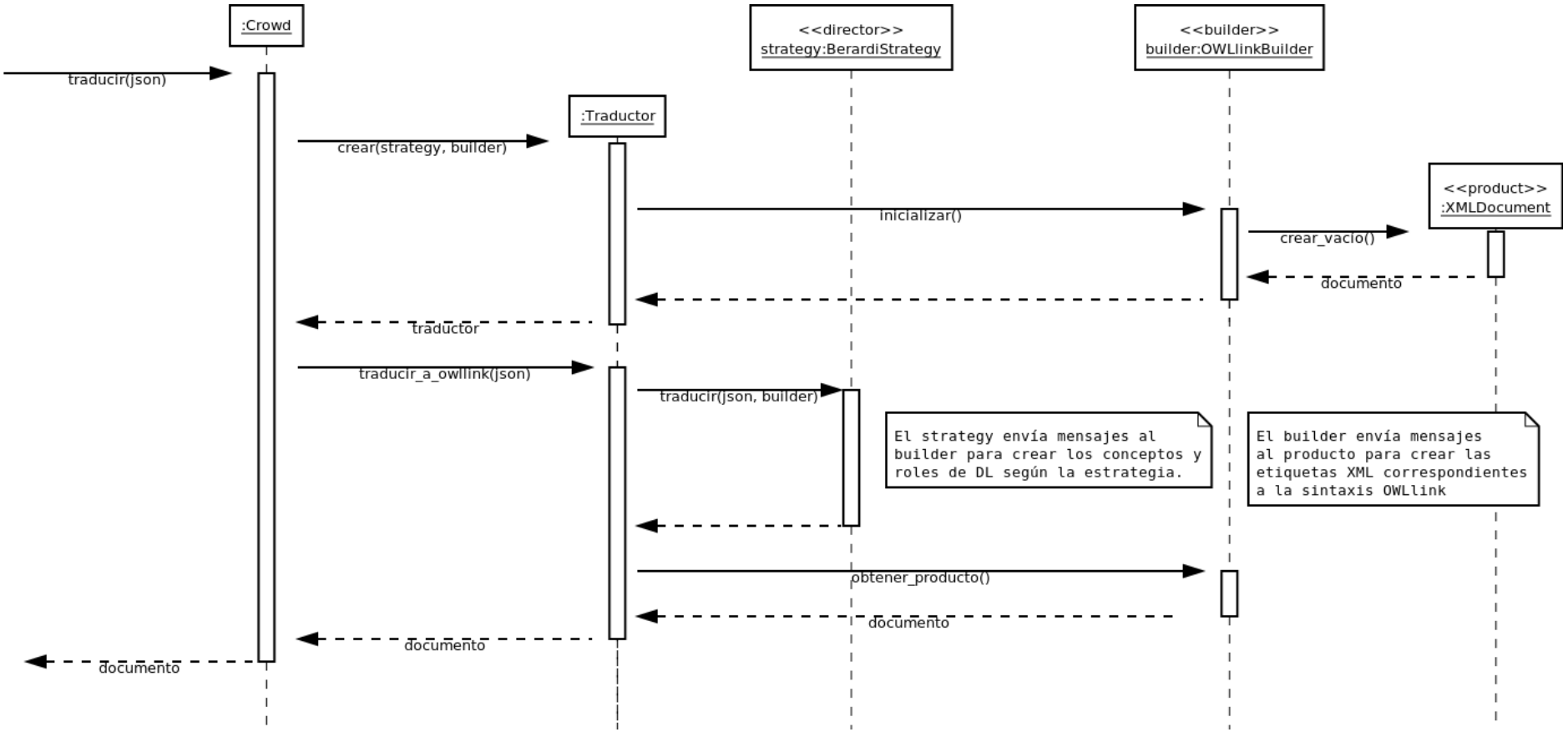


Figura 4.6: Diagrama de secuencia detallando el proceso de traducción.

4.5. Módulo Generador de Consultas y Analizador de Respuestas

El módulo `Generador de Consultas` se encarga de crear consultas que se envían al razonador para chequear la satisfacibilidad de la KB en general y de cada clase del modelo del usuario. Luego, estas se anexarán a la salida del `Módulo Traductor` para que el archivo final sea dado al razonador para su procesamiento. Para poder realizar su función, el módulo requiere del modelo creado por el usuario para crear una consulta OWLlink "`IsClassSatisfiable`" por cada clase del mismo, siguiendo la estructura de la Tabla 4.2 y reemplazando el texto "NombreClase" por el valor correspondiente.

En cuanto al módulo `Analizador de Respuestas`, su función es la de obtener los resultados generados por el razonador de las consultas enviadas, procesarlas y generar un texto en formato JSON para ser enviado a la interfaz del cliente.

El orden de las consultas es importante al momento de generarlas, puesto que el razonador basado en OWLlink emitirá una respuesta por cada consulta y en el mismo orden que se realizan. Por consecuencia, el analizador requiere de la consulta realizada para determinar cuál respuesta corresponde con cuál consulta y así generar un JSON apropiado indicando cuáles clases son inconsistentes.

Un ejemplo de estas consultas con sus respectivas respuestas pueden verse en el Apéndice A, en las Figuras A.5 y A.7. En el caso del `Analizador de Respuestas`, un ejemplo de entrada y de salida se encuentra en el mismo apéndice, en la Figura A.7 y en la Figura A.8 respectivamente.

Consulta	OWLlink	Cantidad
KB Satisfacible	`<IsKBSatisfiable kb="http://localhost/kb1" />`	1

Clase satisfacible	`<IsClassSatisfiable kb="http://localhost/kb1">`
	`  <owl:Class IRI="NombreClase" />`
	`</IsClassSatisfiable>`
		Uno por cada clase

Tabla 4.2: Consultas generadas por el módulo `Generador de Consultas`.

4.6. Módulo Razonador

Es preciso utilizar un razonador al que se le provee de la formalización del modelo conceptual y de sus consultas para determinar la satisfacibilidad del mismo. Por consiguiente, la implementación poseerá un razonador en funcionamiento de fondo.

Con respecto al `Módulo Razonador`, su objetivo es el de proveer de una interfaz para comunicarse con un programa razonador. Es decir, inicializarlo, proveerle de la información de entrada, ejecutarlo y recuperar su salida. En el diseño propuesto (Figura 3.5), la herencia definida sobre la clase `Conector` permite la selección de un razonador u otro.

Cabe aclarar que es posible utilizar otro programa razonador creando una subclase de `Conector`. Ésta debe reimplementar los métodos heredados para que el programa razonador se ejecute y se configure, además de alimentarlo con la salida de los módulos anteriores por algún método IPC (Inter-Process Communication) existente.

En el prototipo de *crowd* implementado se decidió utilizar únicamente el razonador RACER [51], el cual puede trabajar con entradas en formato DIG [14] y OWLlink [72]. Para ejecutar el programa RACER el módulo utiliza la operación de PHP `exec()` ejecutando el proceso. Previo a esto, se prepara un archivo temporal con el texto OWLlink que ha sido generado por el `Traductor` y que será utilizado como entrada indicándole esto al razonador por medio de un parámetro de ejecución. Al finalizar, sus resultados serán capturados de la salida estándar del programa en una variable para que el siguiente módulo la procese. La Figura 4.7 muestra cómo es el formato de un archivo de entrada, cómo se realiza el llamado al programa RACER y su salida utilizando

```
> cat ../owllink-examples/complete.xml
<?xml version="1.0" encoding="UTF-8"?>
<RequestMessage xmlns="http://www.owllink.org/owllink#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.owllink.org/owllink#
    http://www.owllink.org/owllink-20091116.xsd">
  <CreateKB kb="http://localhost/kb1" />
  <Tell kb="http://localhost/kb1">
    <owl:SubClassOf>
      <owl:Class IRI="Person" />
      <owl:Class abbreviatedIRI="owl:Thing" />
    </owl:SubClassOf>
  </Tell>  <!-- Queries -->
  <IsKBSatisfiable kb="http://localhost/kb1" />
  <IsClassSatisfiable kb="http://localhost/kb1">
    <owl:Class IRI="Person" />
  </IsClassSatisfiable>
  <ReleaseKB kb="http://localhost/kb1" />
</RequestMessage>

> ./Racer -- -owllink ../owllink-examples/complete.xml
<?xml version="1.0" encoding="UTF-8"?>
<ResponseMessage xmlns="http://www.owllink.org/owllink#"
                 xmlns:owl="http://www.w3.org/2002/07/owl#">
  <KB kb="http://localhost/kb1"/>
  <OK/>
  <BooleanResponse result="true"/>
  <BooleanResponse result="true"/>
  <OK/>
</ResponseMessage>
```

Figura 4.7: Entrada y salida del razonador RACER usando el protocolo OWLlink.

la línea de comandos y manteniendo la misma instrucción de ejecución dada a la operación de PHP.

El archivo ejecutable del razonador, proveído por la versión actual del prototipo, y el archivo temporal se encuentran designados en un directorio especial denominado "run". De esta forma, al momento de instalar el sistema, el administrador podrá proveerle los permisos necesarios para ejecutarse por el código PHP cada vez que éste último lo requiera.

4.6.1. Razonadores

Para el caso del razonador a incorporar al prototipo, decidimos utilizar RACER debido a su disponibilidad en Internet, su licencia compatible con las Licencias Libres , la compatibilidad con ICOM [40], el soporte a protocolos de comunicación más actuales y la posibilidad de funcionamiento con ontologías lo suficientemente grandes para expresar modelos con muchas primitivas gráficas. Con respecto a los protocolos, optamos por la utilización de OWLlink y OWL 2 [84] por ser los más actuales, por sus expresividades y por sus propiedades computacionales. En la Figura 4.8 se muestra una consulta al razonador RACER, que está funcionando de fondo, por medio de un programa de interfaz gráfica denominado RacerPorter.

Existen varios razonadores disponibles en la actualidad que, en su mayoría, se comunican por medio de uno o varios de los protocolos estandarizados existentes. Ejemplos de estos razonadores son Pellet [96], RACER y RacerPro [52], TopBraid [101] (el cual también es un entorno de desarrollo completo) y aún más que se encuentran listados por la W3C en la siguiente URL: `https://www.w3.org/2001/sw/wiki/Category:Reasoner`. Si bien, la implementación del prototipo no cuenta con el soporte de estos razonadores, es posible utilizarlos en conjunto con RACER desarrollando un adaptador en PHP.

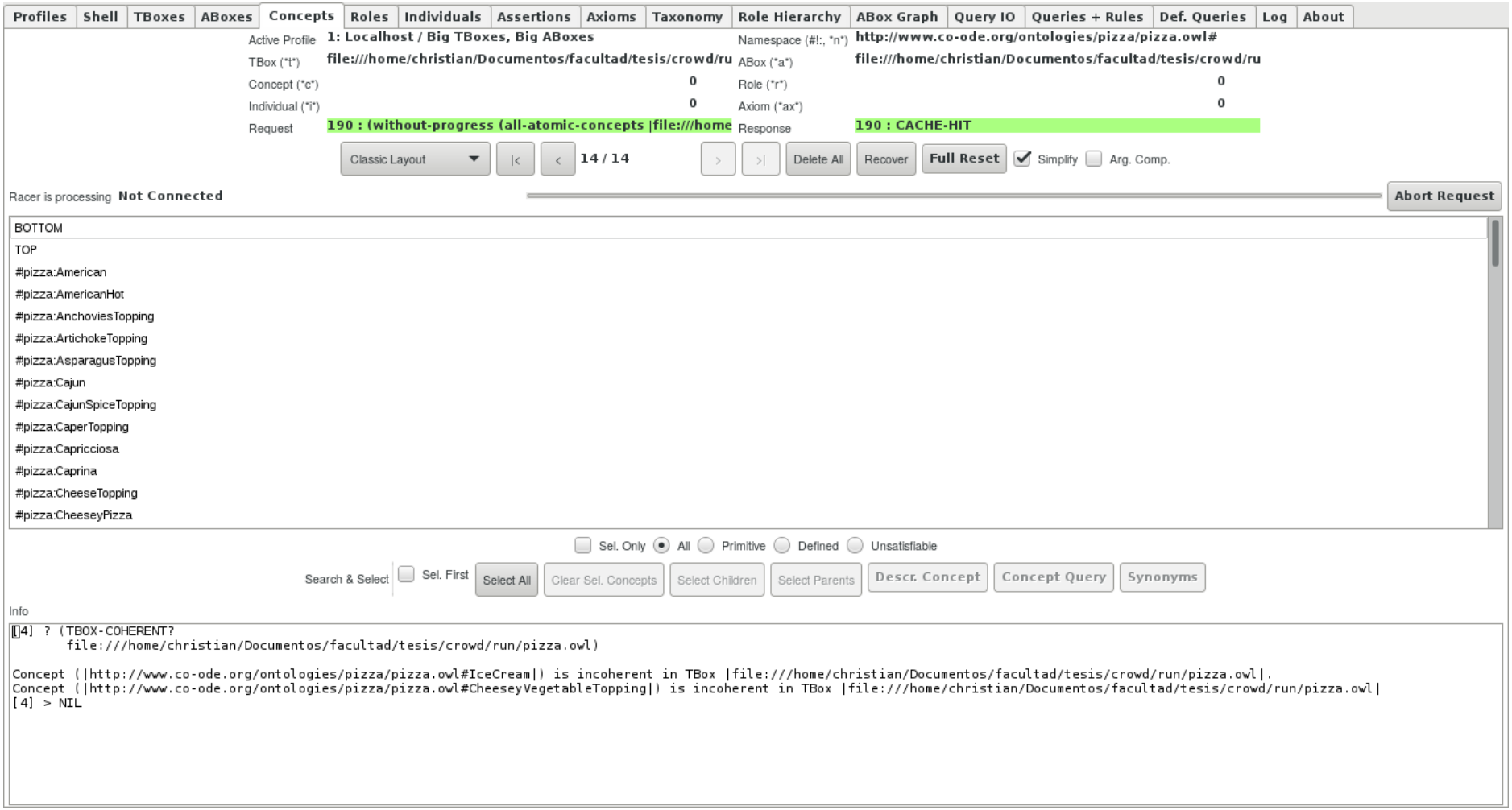


Figura 4.8: RacerPorter, la interfaz de RACER mostrando una ontología y consultando la consistencia de la misma.

4.7. Ejemplo de Funcionamiento de Prototipo de *crowd*

En esta sección se mostrará la arquitectura en funcionamiento utilizando el prototipo explicado previamente. Se explicarán dos casos, uno satisfacible y el otro insatisfacible, para comparar la reacción del programa.

En la sección 4.7.1 se podrá visualizar la interfaz y se explicará el proceso interno que el prototipo realiza. El modelo conceptual, en este caso, será consistente y por ende, el prototipo deberá responder de la misma manera, por lo que se aprovechará a explicar las respuestas descriptas en la interfaz gráfica.

Luego, se presentará el mismo modelo conceptual, pero con unas leves modificaciones para que resulte inconsistente, con la intención de mostrar cómo se comporta la interfaz gráfica ante esta situación y las diferencias en el funcionamiento interno del prototipo. Es interesante y de importancia para el usuario obtener cómo se representa esta inconsistencia en la interfaz, además de cómo analizar los datos proveídos por el prototipo, para poder comprender qué sucede y corregir su modelo rápidamente.

Con la intención de facilitar la lectura, los códigos completos a los que se hará referencia podrán observarse en el Apéndice A, ordenados según como se van generando en el proceso de traducción y verificación de satisfacibilidad.

4.7.1. Modelo Conceptual Satisfacible

En [19] se presenta un modelo conceptual sencillo que es consistente y una posible traducción a $\mathcal{ALCQI}$ del mismo. Dicho modelo se muestra en la Figura 4.9.

El usuario puede desarrollar este mismo modelo. Primeramente, por medio de un panel de herramientas se puede crear todas las clases del mismo. Para facilitar el modelado, las relaciones se crean utilizando un menú contextual que se activa sobre la clase creada, en él se puede brindar el nombre, seleccionar la multiplicidad y determinar las clases relacionadas. Actualmente, se soportan las multiplicidades de cero a muchos $(0 \ldots *)$, de uno a muchos $(1 \ldots *)$, de uno a uno $(1 \ldots 1)$ y de cero a uno $(0 \ldots 1)$. La herencia también puede ser dibujada por medio del mismo menú, activando el tipo de restricción (disjunto y/o completa) y seleccionando las clases hijas.

Al diseñar el modelo, por cada acción del usuario se modifica una representación de éste en el motor Javascript del explorador web, el cual no sólo controla y responde las acciones del usuario, sino también almacena la información para luego ser traducida en una sintaxis transportable al servidor.

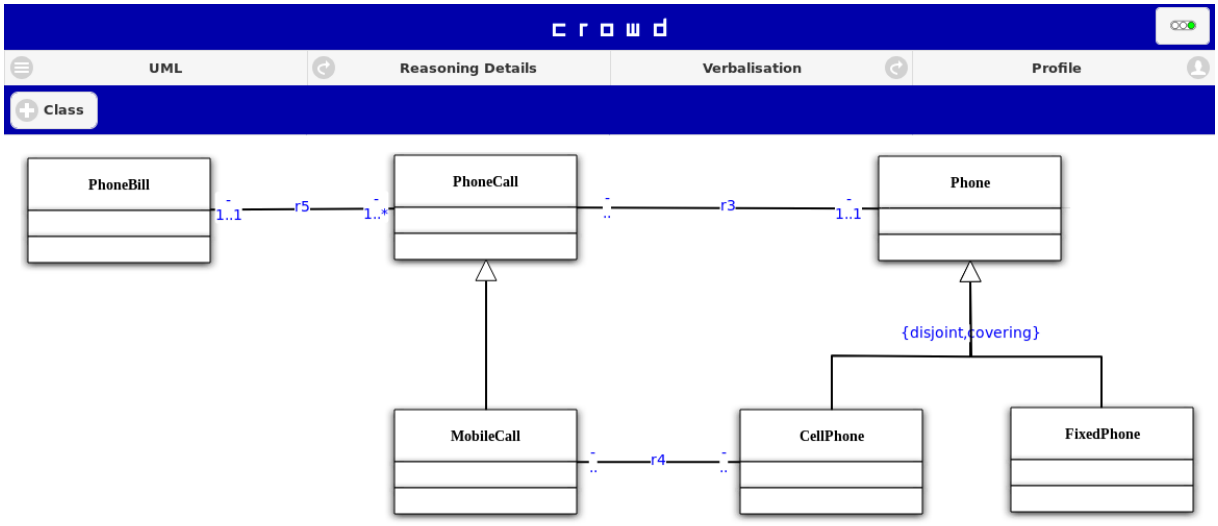


Figura 4.9: Modelo conceptual consistente extraído de [19].

Una vez finalizada la creación del modelo de la Figura 4.9, el usuario puede solicitar al prototipo su evaluación por consistencia. Al hacerlo, el modelo Javascript genera una representación en JSON para ser enviada al servidor y posteriormente traducido a lógica descriptiva. El explorador web queda a la espera de la respuesta a su petición HTTP para procesarla y mostrar los resultados. En el apéndice en la Figura A.3 puede verse la representación en JSON generada y enviada al servidor.

En el servidor, se recibe esta petición y el modelo se traduce a DL generando una sintaxis OWL 2 como la que se muestra en la Figura A.4. A este texto se le incorpora una serie de consultas para determinar la consistencia del modelo, las cuales son anexadas al final cuando se genera el documento OWLlink. La Figura A.5 y la Figura A.6 presentan las consulta para el ejemplo dado y un template donde puede visualizarse el lugar donde aparecen las distintas secciones presentadas.

Una vez generado el documento OWLlink completo, éste es enviado al razonador para que sea procesado y genere una respuesta, que poseerá el formato que se describe en el estándar OWLlink[6] y en [72]. Se presenta una posible respuesta del razonador RACER usado para este ejemplo en la Figura A.7.

Esta respuesta es analizada para determinar qué clases son las inconsistentes y producir un texto en formato JSON para el cliente. La respuesta, presentada en la Figura A.8, posee los datos que requiere la interfaz: qué clases son inconsistentes, cuáles no lo son, y los textos en formato XML de entrada y salida del razonador.

Finalmente, la interfaz recibe la respuesta del servidor y la procesa mostrando al usuario los resultados: colorea una tonalidad rojiza las clases que figuran en el campo "unsatisfiable", restaura el color a las clases que se nombran en "satisfiable", agrega la entrada y salida del razonador en las áreas de textos que corresponden. Por último, carga la imagen de un semáforo con el color correspondiente que indica la consistencia del modelo. Actualmente hemos decidido que el color verde será utilizado cuando todo el modelo es consistente y rojo cuando alguna clase sea inconsistente.

4.7.2. Modelo Conceptual Insatisfacible

Para mostrar el funcionamiento del prototipo sobre un modelo insatisfacible, partiremos del modelo presentado en la sección anterior con unas ligeras modificaciones que lo harán inconsistente, ambos fueron extraídos de [19].

A partir de la Figura 4.9, se le incorpora una asociación de herencia entre la clase "FixedPhone" y la clase "CellPhone" para hacer inconsistente al modelo, debido a que la restricción de completa y disjunta no puede permitir que una clase hija tenga relación Is-A de otra clase hija. El resultado se muestra en la Figura 4.10.

Al realizar el dibujo, el JSON que representa al modelo y que será enviado al servidor, incluirá dentro del campo "links" una relación más al final. La Figura A.9 muestra una porción del código agregado junto con unas líneas del código anterior para poder determinar el contexto donde fue modificado. Esto, por consecuencia, será traducido por el servidor junto con el resto del modelo utilizado en la sección anterior, pero agregando lo necesario para respetar la semántica de la nueva generalización. Puede observar en la Figura A.10 la representación OWL 2 de la nueva asociación agregada al resto de la traducción de la Figura A.4.

En cuanto a las consultas generadas por el módulo respectivo no sufren ninguna modificación pues no hay nuevas clases en el modelo. Por ello, el OWLlink resultante es similar al anterior siguiendo el mismo formato presentado en la Figura A.6 con el texto OWL 2 modificado y las mismas consultas de la Figura A.5.

El razonador, al recibir este texto, emitirá una respuesta diferente. Si observamos las últimas líneas de la representación OWL 2 nos encontramos con los siguiente predicados $\mathcal{ALCQI}$:

[6] https://www.w3.org/Submission/2010/SUBM-owllink-structural-specification-20100701/

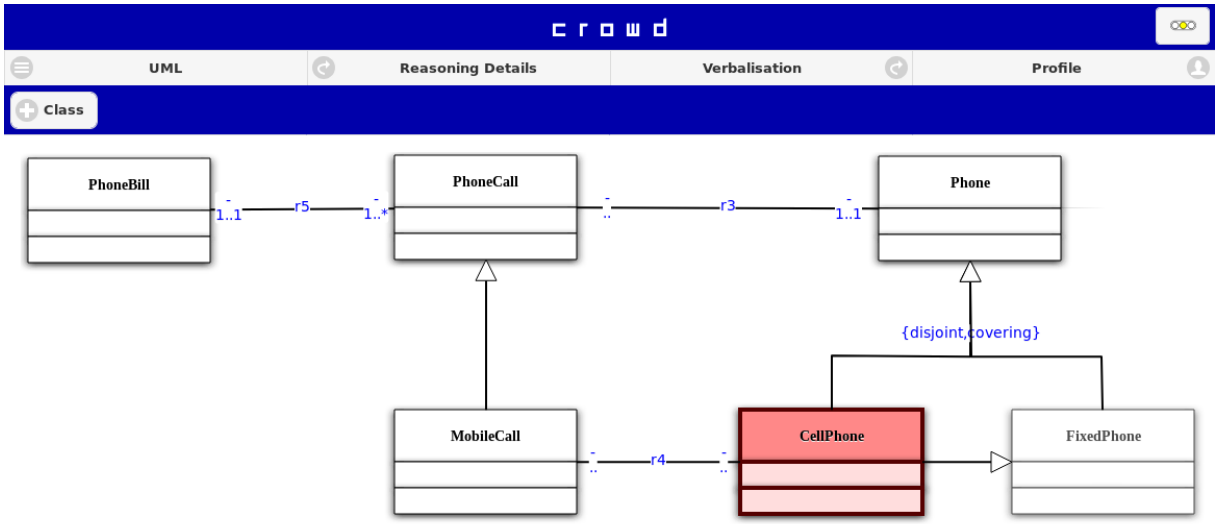


Figura 4.10: Modificación de la Figura 4.9 para que el modelo sea insatisfacible.

$CellPhone \sqsubseteq Phone$
$FixedPhone \sqsubseteq Phone$
$CellPhone \sqsubseteq (\neg FixedPhone)$
$Phone \sqsubseteq (CellPhone \sqcup FixedPhone)$
$CellPhone \sqsubseteq FixedPhone$ (Sentencia agregada)

El absurdo se encuentra al intentar indicar que las instancias de CellPhone están en el complemento de FixedPhone por la tercer sentencia, y también se encuentran dentro de FixedPhone por la última sentencia agregada. Esto no puede ser posible, una instancia es un FixedPhone o es su complemento, no es posible ambas.

Esto lleva a que el razonador responda lo expresado en la Figura A.11. Obsérvese que la anteúltima respuesta booleana, que corresponde con la consulta si CellPhone es satisfacible (véase las consultas en la Figura A.5), indica que la clase no es consistente.

Luego, el módulo `Analizador de Respuestas` producirá el texto en formato JSON de la Figura A.12 para ser enviado a la interfaz, la que podrá indicarle al usuario los resultados del razonador.

La Figura 4.10, posee la clase CellPhone coloreado de una tonalidad rojiza y un reborde más grueso, indicando que no puede poseer instancias puesto que es inconsistente bajo el modelo conceptual dado por el usuario. De esta manera, el usuario deberá enfocar su atención a las clases señaladas por la interfaz para corregir su diseño.

Capítulo 5

Comparación con Otras Herramientas

Durante el transcurso de este trabajo hemos relevado y estudiado algunas herramientas existentes que posibilitan la creación, edición y visualización de ontologías. Este análisis ha sido realizado siguiendo los lineamientos del Proceso de Visualización detallado en la sección 3.1 y considerando principalmente, los lenguajes gráficos de modelado, el uso de razonamiento para validar modelos y, en caso que sea posible, cuán integrados trabajan los lenguajes y el razonamiento automático.

En este capítulo se detallan las herramientas investigadas indicando los puntos resaltados en el párrafo anterior y las principales diferencias de cada herramienta con nuestra propuesta. Las implementaciones consideradas son las siguientes: Protégé [66] y WebProtégé, VOWL [73] y WebVOWL, OWLGrEd [28], NORMA [33], TopBraid Composer [101], NeOn toolkit [55], Graphol [31] y OntoUML [50]. Otras herramientas [67, 64] fueron revisadas pero no serán detalladas aquí puesto que han quedado discontinuadas.

5.1. Protégé y WebProtégé

Protégé es una plataforma libre, desarrollada por la Universidad de Stanford. Ofrece un conjunto amplio de estructuras de modelado para la creación y manipulación de ontologías, junto con interfaces con razonadores automáticos. La herramienta puede ser extendida mediante el desarrollo de *plug-ins*. La edición en Protégé es principalmente textual y un escaso soporte gráfico es provisto por *plug-ins* como OWLViz [58] y SOVA [70]. De manera similar el razonamiento es completamente soportado de manera textual, pero las deducciones obtenidas por razonadores externos son limitadas a las de tipo Is-A de manera gráfica. Otras extensiones gráficas, como por ejemplo OntoGraf [35] dan soporte interactivo para navegar sólo las relaciones de subclasificación, individuos, dominios y rangos de las propiedades y equivalencias en OWL. En cuanto a la integración con el razonador, es limitada y levemente soportada. Asimismo, ninguna nueva deducción a nivel textual es reflejada gráficamente usando OntoGraf. OWLViz visualiza la jerarquía de clases de ontologías OWL, las cuales pueden ser además navegadas de manera incremental, mientras que los conceptos inconsistentes y las jerarquías inferidas pueden ser visualizadas luego de razonar sobre el modelo. Por último, SOVA (Simple Ontology Visualization API) posee dos modos de visualización. Por un lado, una vista de la ontología completa como se especifica en OWL y, por otro, presenta la jerarquía de clases e individuos inferidas luego de que los servicios de razonamiento son ejecutados.

Esta importante limitación también se presenta en la versión Web de la herramienta denominada WebProtégé [103], en la cual la interacción con razonadores es nula y tampoco se provee un soporte gráfico, siendo principalmente un repositorio de ontologías con capacidades de edición colaborativa.

En la Figura 5.1 se puede observar la interfaz de Protégé con una ontología conocida como MUTO (Modular Unified Tagging Ontology) que se puede cargar directamente desde la URL

http://purl.org/muto/core# y mostrando la descripción de una de sus clases.

crowd se basa en la utilización de lenguajes gráficos para modelado conceptual para representar una ontología y el uso del razonamiento en una representación formal del modelo proveído por el usuario. Sin embargo, estos objetivos no son compartidos por Protégé y WebProtégé por lo que su soporte gráfico y de razonamiento son limitados.

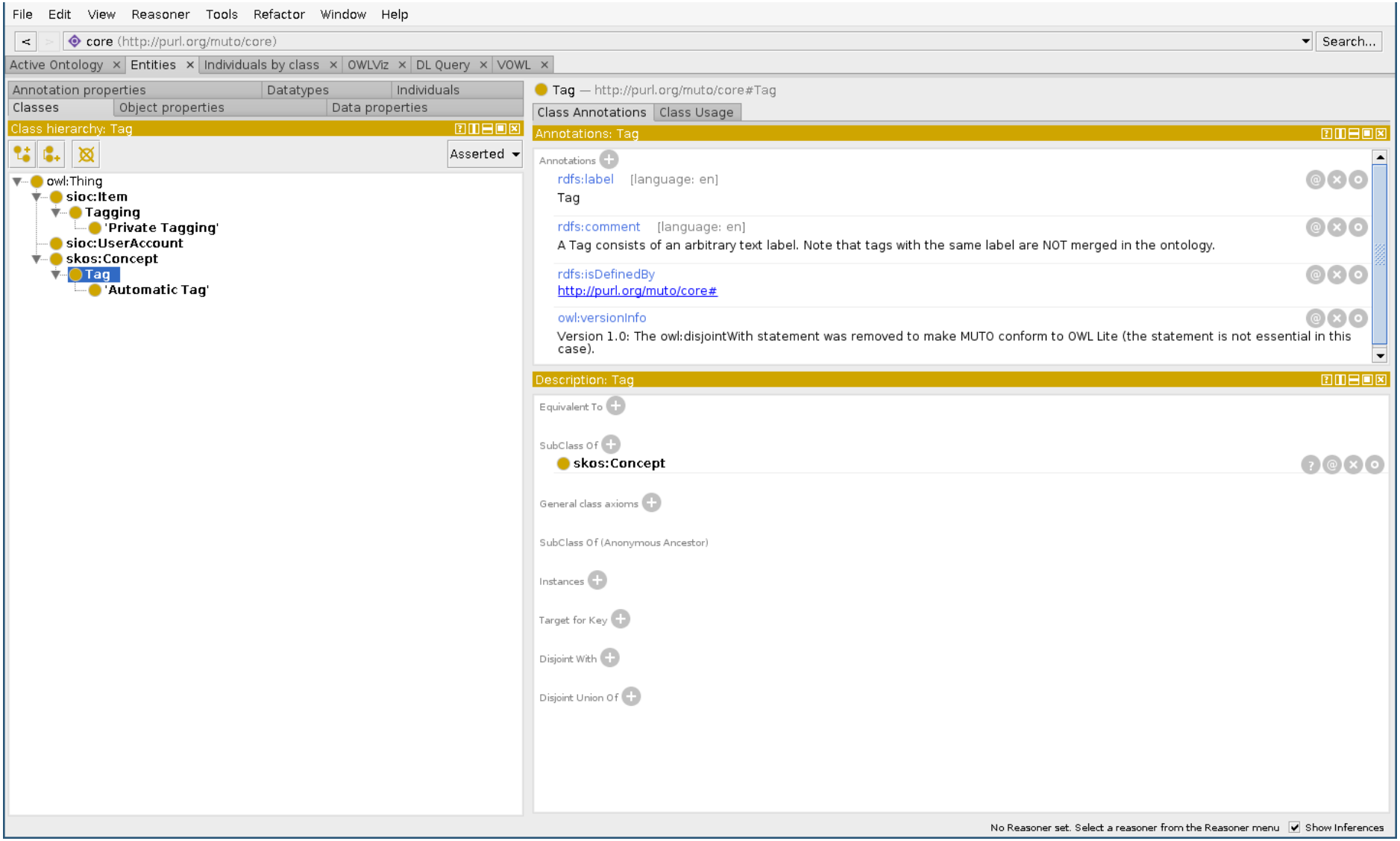


Figura 5.1: Captura de pantalla de Protégé.

5.2. VOWL y WebVOWL

VOWL ("Notación Visual para Ontologías OWL") [74] es básicamente un lenguaje visual para la representación de ontologías orientado al usuario. Define símbolos gráficos para los elementos de OWL que son combinados con un grafo dirigido visualmente distribuido. Se enfoca en la visualización de los esquemas de ontología (TBox), asimismo incluye recomendaciones en cómo dibujar individuos y valores de datos correspondientes al ABox de la ontología. Este lenguaje fue implementado en dos herramientas, un *plug-in* para Protégé denominado ProtégéVOWL y una aplicación Web llamada WebVOWL. Ambos realizan una representación visual, que no es editable en ese mismo lenguaje de la ontología que el usuario provee por medio de un archivo en formato OWL. La Figura 5.2 muestra la interfaz de ProtégéVOWL donde, de forma automática y animada, presenta las clases de la ontología MUTO.

A pesar de su facilidad de uso y de que distribuye las primitivas automáticamente en pantalla, existen grandes diferencias con *crowd*: Primeramente, es requisito estar familiarizado con el lenguaje OWL y las demás tecnologías de la Web Semántica para poder utilizarlo, puesto que utiliza un lenguaje visual que representa la DL y sus operadores, en vez de utilizar aquellos propios del dominio de la ingeniería de software. Segundo, tanto el *plug-in* como WebVOWL, no permiten la edición de las ontologías en el mismo lenguaje gráfico, sólo la visualización de las mismas. Tercero, no se provee ningún tipo de servicios de razonamiento en la interfaz web, y en cuanto al *plug-in*, sólo lo que posibilita la herramienta Protégé.

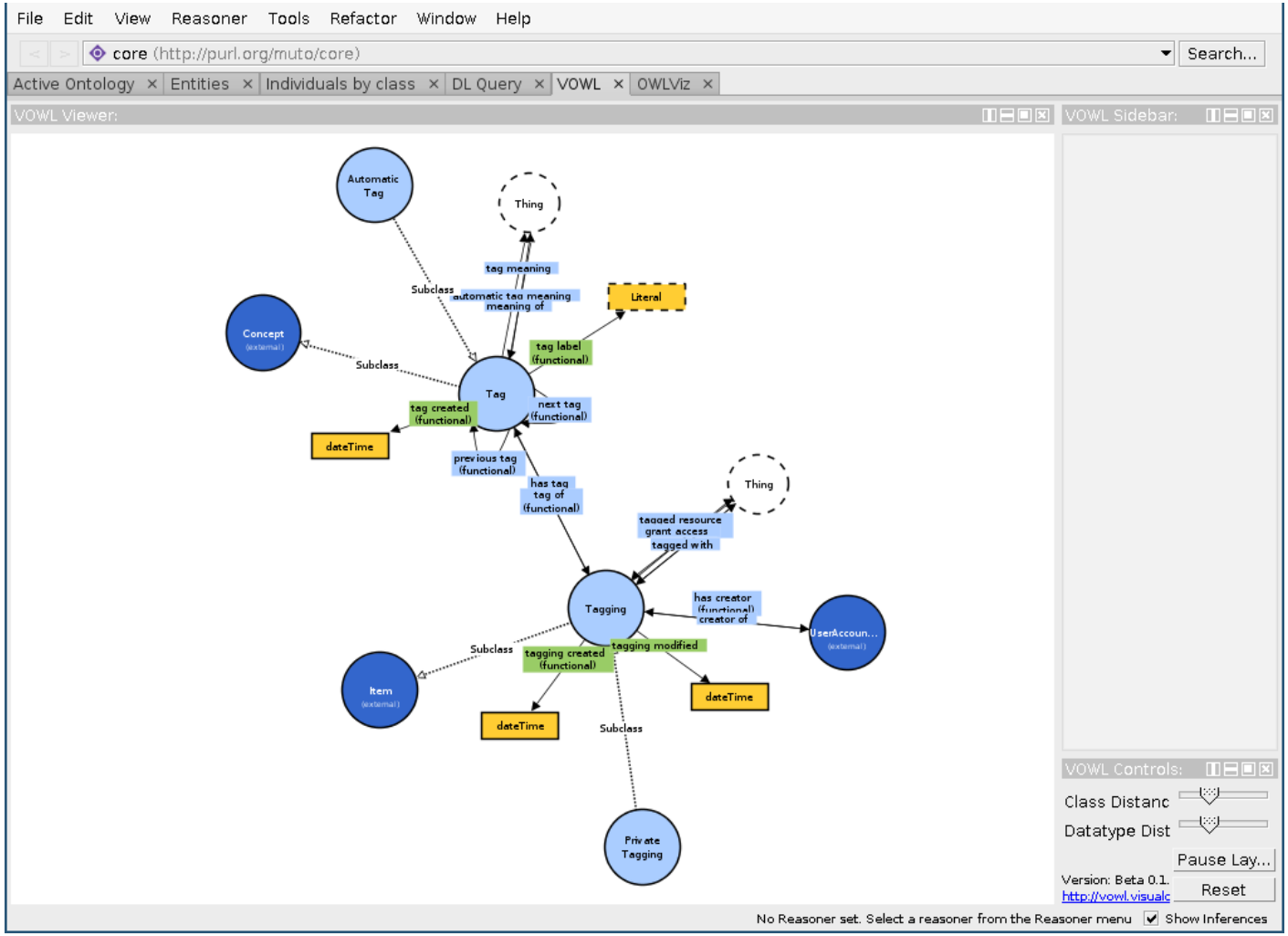


Figura 5.2: VOWL Protégé mostrando una ontología OWL.

5.3. TopBraid Composer y NeOn Toolkit

TopBraid Composer y NeOn Toolkit son herramientas desarrolladas sobre la plataforma Eclipse para la ingeniería de ontologías. Permiten la carga y edición textual de ontologías desde archivos en formato OWL. Similar a Protégé, también muestran inferencias gráficamente, pero limitadas a relaciones Is-A. El primero, sin embargo, también provee otro formato de visualización por medio de gráficos RDF [93], en tanto que el soporte de visualización en NeOn Toolkit es mediante el *plug-in* "OWL Ontology Visualization". En la Figura 5.3 se muestra la herramienta NeOn Toolkit con la ontología FOAF (Friend of a Friend) cargada desde la URI `http://xmlns.com/foaf/0.1` y con el *plug-in* de visualización centrado en la clase "Agent" mostrando sus distintas subclases.

A diferencia de *crowd*, estas herramientas no proveen de servicios de razonamiento *out-of-the-box*. Sin embargo, algunas se apoyan de *plug-ins* y de la posibilidad de exportar a OWL para poder interactuar con razonadores y realizar consultas, incluso utilizan la sintaxis RDF y soportan SparQL para hacer consultas específicas solamente en cuanto a sus individuos. En cuanto a la visualización de ontologías, *crowd* permite la edición en el mismo lenguaje gráfico y no requiere que el usuario posea conocimiento acerca de DL ni de la sintaxis OWL, en contraposición con TopBraid Composer y NeOn Toolkit que se enfoca en una edición textual en archivos OWL.

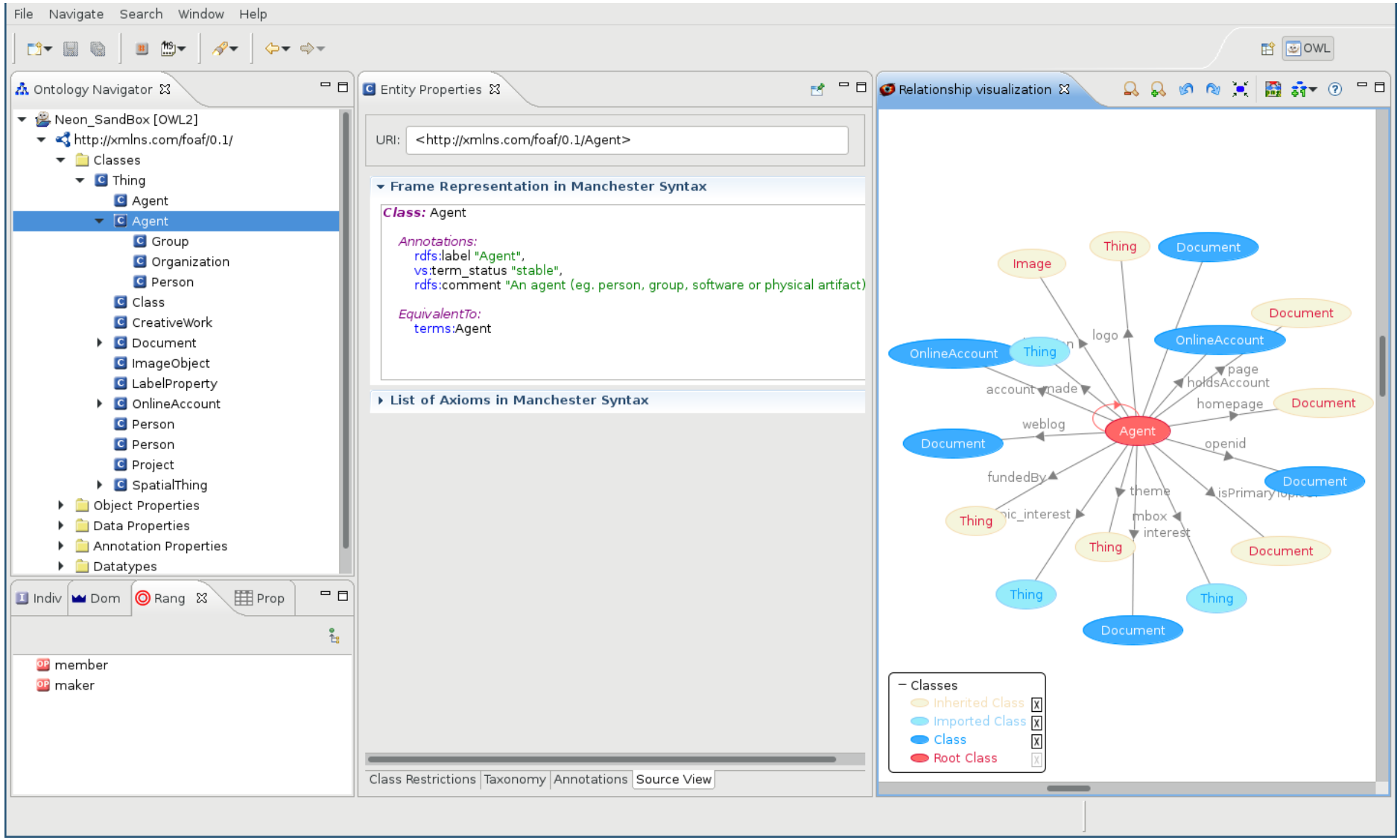


Figura 5.3: Captura de Pantalla de NeOn Toolkit mostrando un Agente de la ontología FOAF.

5.4. OWLGrEd

El editor de ontologías OWLGrEd [13] provee un entorno gráfico para OWL 2 que, como se explica en [13], utiliza una sintaxis gráfica de estilo similar al de los diagramas de clases UML. Dicha sintaxis consiste en un mapeo entre conceptos OWL que son similares a los de UML, por ejemplo: de clases de OWL a clases UML, de subclases OWL (`subclassOf`) a herencia UML, etc. Sin embargo, existen conceptos únicos en OWL por lo que también define una extensión de la notación UML con símbolos adicionales y expresiones textuales para abarcarlos. El entorno completo es capaz de mostrar ontologías OWL y/o RDF en el lenguaje descripto, además de editarlas bajo el mismo lenguaje y guardarlas en el formato con el que se cargó. Asimismo, otras ontologías pueden ser importadas como paquetes UML a un único proyecto.

El *plug-in* OWLGrEd permite interoperar entre Protégé y el entorno completo habilitando la capacidad de exportar el gráfico a esta última y viceversa. Sin embargo, el *plug-in* no soporta las últimas versiones de Protégé, limitándose a versiones previas a la 5.0.0 beta. En su versión online, disponible en la URL `http://owlgred.lumii.lv/online_visualization`, sólo permite visualizar una ontología a partir de un archivo OWL, RDF, etc, aunque existen limitaciones de tamaño y sólo se observa la ontología directa y no las que se hacen referencia.

Finalmente, si bien OWLGrEd, ya sea en su versión de escritorio o Web, soportan un lenguaje basado en el UML estándar, no poseen ningún servicio de razonamiento integrado más allá de los provistos por Protégé, cuyas posibles nuevas inferencias no son inmediatamente graficadas. Sin embargo, la utilización de un modelado gráfico similar al utilizado por la ingeniería de software hace a la herramienta más accesible, a pesar de considerar los aspectos propios de OWL que hacen a la extensión del lenguaje.

5.5. NORMA

NORMA es una herramienta principalmente gráfica para la visualización y edición de ontologías por medio del lenguaje ORM. Este lenguaje es gráfico y de representación en forma de diagrama cercana a la ontología de la Lógica Descriptiva. Está formalmente descripto en [53] como un modelado orientado a hechos para especificar modelos, transformaciones y consultas de un domino de negocios a nivel conceptual.

NORMA, aunque provee de un lenguaje ampliamente descripto, no posee ningún soporte de razonamiento y, además, requiere de la instalación de Visual Studio 2015 o inferior, haciendo de esta herramienta dependiente de versiones desactualizadas de esta IDE. Sin embargo, el soporte al lenguaje visual es próximo a la Lógica Descriptiva y de interés por promover la orientación a los hechos.

5.6. Eddy para Graphol

Graphol [31] es un lenguaje visual para ontologías que posean una expresividad en sus TBoxes de tipo $\mathcal{SROIQ}(\mathcal{D})$. Incluso, como se describe en [30], permite representar ontologías, como si fueran en OWL 2, pero por medio de una representación completamente gráfica evitando la escritura que requiere de sintaxis complejas.

La herramienta ofrecida en su sitio web[1] se denomina Eddy y permite importar y exportar OWL 2 para ser graficado, y luego visualizar y editar este archivo por medio del lenguaje visual. La Figura 5.4 muestra al editor Eddy con una ontología de ejemplo.

Cabe notar que se requiere la comprensión de un nuevo lenguaje puesto que no se provee soporte para los lenguajes estándares en la industria del desarrollo de software (i.e. UML, EER, etc.). Además, no posee ningún tipo de servicio de razonamiento para determinar la consistencia

[1] `http://www.dis.uniroma1.it/~graphol/`

o resolver consultas. A pesar de que permite la incorporación de *plug-ins*, no se han encontrado ninguno que permita conectar la herramienta a tal servicio.

La posibilidad de utilizar OWL 2 como lenguaje permite la utilización de un razonador *a posteriori* si la ontología exportada es previamente encapsulada en un texto OWLlink junto con las consultas. Sin embargo, esto debe realizarse por separado de forma manual, escribiendo dicho texto y ejecutando el razonador.

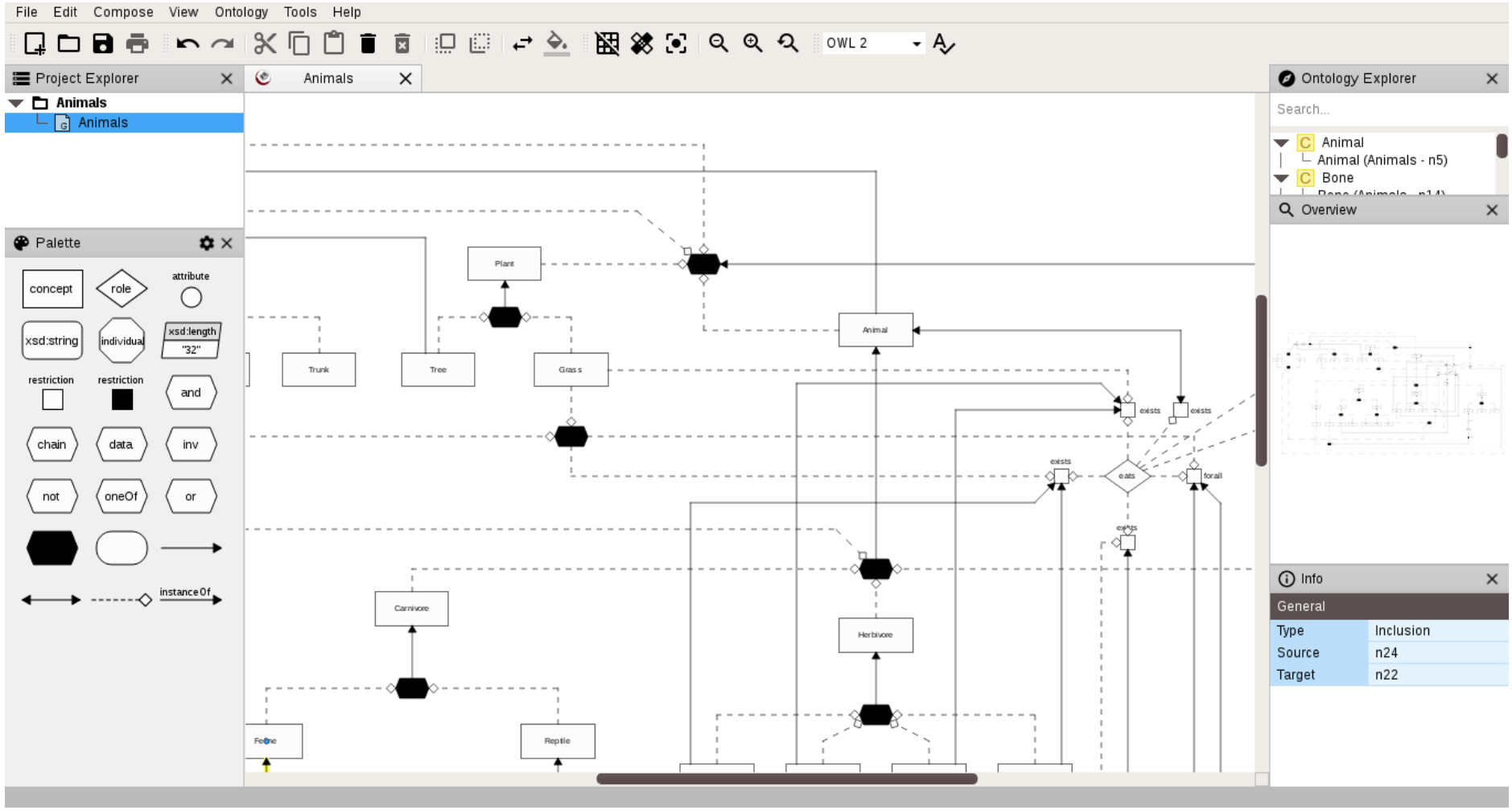


Figura 5.4: Editor gráfico Eddy mostrando una ontología en Graphol.

5.7. Menthor y OntoUML

OntoUML [50] es una versión formal basada en patrones del lenguaje UML. Su metamodelo ha sido diseñado en conformidad con las distinciones ontológicas de una teoría bien fundada denominada Unified Foundational Ontology (UFO) propuesta en [49].

Menthor es un editor gráfico para OntoUML, que tiene la posibilidad de incluir reglas OCL (Object Constraint Language) [48] al modelo y también de verificar la consistencia de los diagramas por medio de un razonador denominado Alloy Analizer. También, permite exportar el modelo a una ontología en varios formato, incluyendo OWL, y utilizando varias codificaciones denominadas "aproximaciones".

Tanto *crowd* como Menthor utilizan lenguajes visuales y pueden generar una representación en DL de los modelos proveídos por el usuario. Sin embargo, *crowd* se basa en lenguajes de modelado conceptual ampliamente conocidos por la ingeniería del software, como el UML, para el desarrollo de ontologías, evitando la necesidad de aprender un lenguaje nuevo. Asimismo, para la verificación de satisfacibilidad del modelo del usuario, el servicio de razonamiento de *crowd* utiliza el programa razonador RACER [51] cuya expresividad se corresponde con la DL $\mathcal{SHIQ}$. Sin embargo, el razonador utilizado por Menthor, Alloy Analizer, está basado en un algoritmo capaz de resolver SAT (Problema Booleano de Satisfacibilidad). Alloy Analizer recibe como entrada modelos escritos en Alloy [63], un lenguaje para describir estructuras, basado en lógica e inspirado por el lenguaje para especificaciones de software Z [1], requiriendo así, una transformación del modelo para su posterior verificación de satisfacibilidad.

5.8. ICOM

Finalmente, la más relacionada a nuestro trabajo es ICOM [37, 39, 36, 38], considerándola como la herramienta fundacional de *crowd*. ICOM adopta un lenguaje gráfico neutral para representar ontologías por medio de diagramas similares al UML o ER, aprovechando la ventaja de que sea claro aún para personas que no están familiarizados con los lenguajes ontológicos más clásicos.

Según [40], la herramienta posee una interfaz que brinda apoyo a la ingeniería ontológica para crear modelos claros y de criterios de calidad medibles. Esto se logra por medio de la formalización interna, traduciendo los diagramas y restricciones entre modelos a una lógica basada en clases.

En la Figura 5.5 se presenta una captura de pantalla de la herramienta con una ontología de ejemplo. ICOM ofrece una opción para conectarse a un razonador y chequear la consistencia del modelo.

A pesar de tener la posibilidad de utilizar servicios de razonamientos, el protocolo DIG, junto con sus limitaciones para modelar ontologías más expresivas, y las bibliotecas gráficas que utiliza, ya se encuentran discontinuados haciendo que estos aspectos críticos de la herramienta dificulten su continuidad en el desarrollo. *crowd* está desarrollada con tecnología web para poder ser utilizada por varias personas y promover así el uso colaborativo de la misma descartando las dificultades de instalación y puesta en marcha, diferencia sustancial con ICOM la cual está escrita en Java con la intención de que sea solamente una aplicación de escritorio.

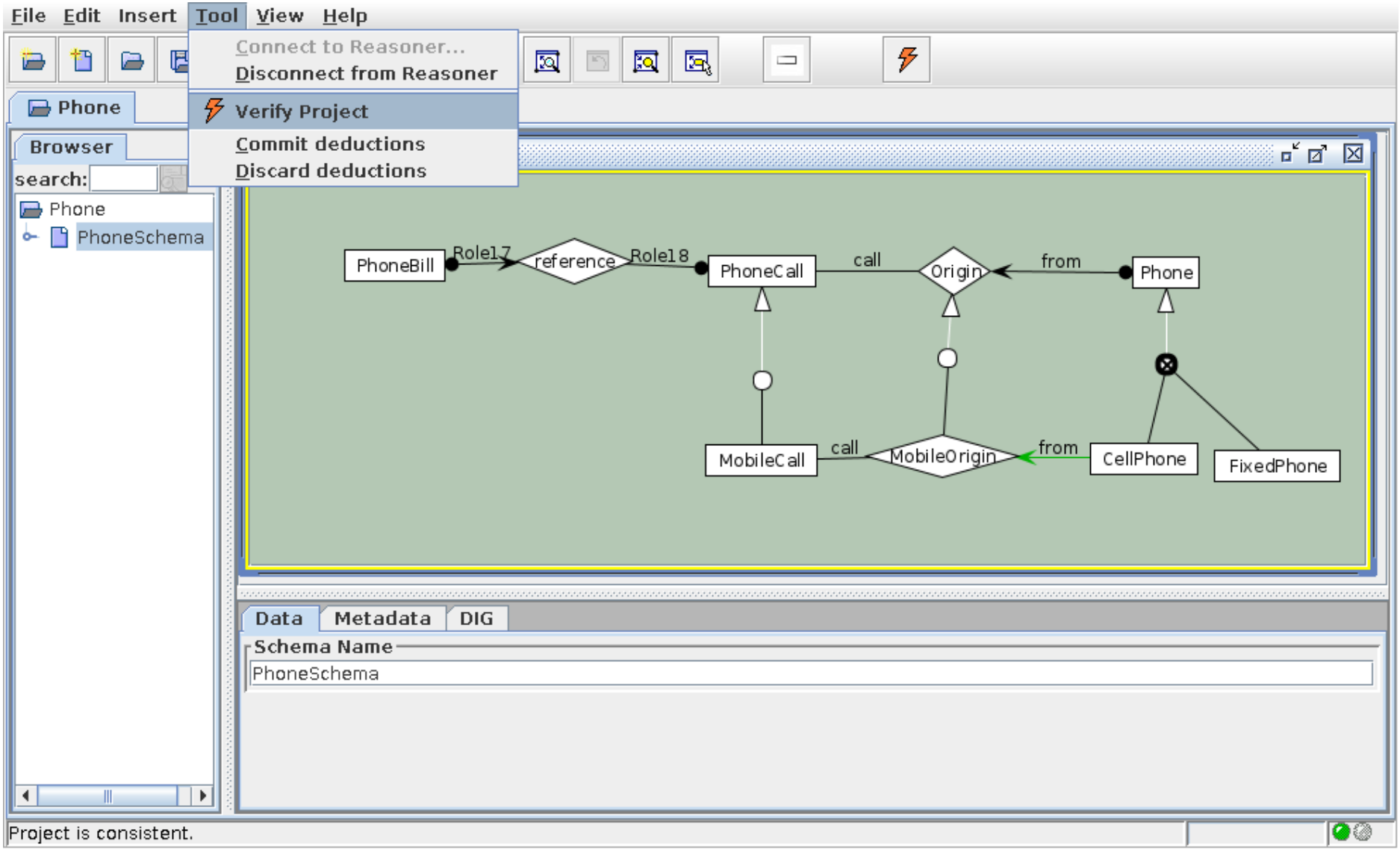


Figura 5.5: ICOM con una ontología de ejemplo.

Capítulo 6

Conclusiones

La necesidad de una herramienta Web colaborativa que ofrezca soporte gráfico a los diseñadores, con razonamiento subyacente para mejorar la calidad de los modelos ontológicos fue la motivación más fuerte en este trabajo, resultando en la creación de una arquitectura y de una implementación de la misma. En este desarrollo, nos hemos centrado en formalizaciones de modelos conceptuales de un subconjunto de primitivas de un lenguaje muy utilizado en la industria del desarrollo de software como el UML, dejando abierta las posibilidades para la incorporación de otros lenguajes de modelado conceptual que sean considerados de preferencia en la industria del software.

Es importante para la arquitectura la integración entre las representaciones visuales y las capacidades de razonamientos lógicos. Los modelos conceptuales utilizados a escala industrial pueden resultar largos y complejos de diseñar, analizar y mantener. La expresividad de los lenguajes gráficos pueden llevar a consecuencias implícitas que no son detectadas por el diseñador en estos diagramas, causando así varias formas de inconsistencias o redundancias. Esto resulta en una degradación en la calidad del diseño y/o en el incremento del tiempo y del costo. La herramienta resultante de la implenetación de la arquitectura propuesta, nos brinda la posibilidad de realizar razonamientos sobre modelos conceptuales cuyos resultados pueden afectarlos. Por ejemplo, puede detectar dichas consecuencias implícitas del modelo que pueden degradar la calidad del mismo, las cuales pueden ser mostradas y contrarrestadas si el usuario lo desea.

En consecuencia, se presentó una arquitectura Web que se basa en el Proceso de Visualización explicado, en Lógicas Descriptiva como forma de representación ontológica y en la formalización de un lenguaje visual de modelado conceptual a esta lógica. También, se planteó el diseño y la interacción de cada módulo para que el usuario pueda crear su modelo y finalmente verificar la consistencia del mismo.

Posteriormente, basado en dicha arquitectura se implementó una herramienta Web colaborativa denominada *crowd*. En esta, se eligió al lenguaje UML para el modelado conceptual gráfico, en el que se consideró un subconjunto de primitivas de los diagramas de clase. Asimismo, se optó por los lenguajes OWL 2 y OWLlink para representar en Lógica Descriptiva los modelos y las consultas necesarias para verificar su consistencia y de la potencia del razonador RACER para ejecutarlas. Para brindar soporte visual, se seleccionó basándonos en ciertos criterios, la biblioteca gráfica JointJS de entre varias que hemos encontrado disponible, y del lenguaje CoffeeScript para realizar una implementación lo más parecida a los diseños propuestos. Además, se presentó un ejemplo que describe el funcionamiento de *crowd* y cómo se muestran las inconsistencias en el mismo lenguaje gráfico.

Para finalizar, se realizó una comparación de las distintas herramientas disponibles que hemos relevado con respecto a *crowd*, describiendo sus diferencias y semejanzas enfocado especialmente al lenguaje para modelar y si posee un servicio de razonamiento de fondo. Concluyendo de esta manera, que a diferencia de *crowd*, las herramientas relevadas que trabajan con servicios de razonamiento vinculados en ellas y que utilicen lenguaje gráfico son escasas, además que no se

fundamentan fuertemente en el Proceso de Visualización para Modelado Conceptual Ontológico.

En resumen, *crowd*, el producto resultante de la arquitectura propuesta, se establece como una herramienta Web que favorece a la colaboración de diseños ontológicos, minimizando los tiempos cognitivos al permitir la utilización de lenguajes visuales de preferencia para el diseñador y que son de amplio uso en la industria del desarrollo de software haciéndolo aún más accesible. Además, la integración con el servicio de razonamiento, basándose en el Proceso de Visualización para Modelado Conceptual Ontológico y que comparte sus objetivos, permite asistir al usuario verificando el modelo en desarrollo evitando las inconsistencias y, por consecuencia, la degradación de la calidad del mismo.

6.1. Trabajos Futuros

Al momento de diseñar e implementar esta arquitectura se tuvo en cuenta los cambios más importantes que pueden surgir a futuro. Por ejemplo, el uso de diferentes razonadores requerirá de una interfaz que permita adaptar la ejecución y la puesta en marcha de estos. El diseño de *crowd* brinda soporte para realizar estos cambios.

En el lado del cliente, se consideró la posibilidad de utilizar varias primitivas gráficas de otros lenguajes como ORM y EER e incluso interoperar entre estas primitivas para aumentar la expresividad visual. Aunque sólo se brinde la posibilidad de programar o de realizar estas acciones, la semántica y la traducción junto con los factores que se tuvieron en cuenta para la interacción de varios lenguajes visuales no corresponden con el objetivo de este trabajo.

En cuanto a la interoperabilidad y la posibilidad de generar diferentes representaciones del modelo del usuario en los lenguajes mencionados, se logra con la implementación del metamodelo expuesto en la sección 3.1. Este trabajo está en desarrollo y algunos avances preliminares se han presentado en [24].

En cuanto al servidor, la traducción puede ser expandida para soportar diversas primitivas gráficas de los lenguajes proveídos por el cliente, varias estrategias o metodologías de codificación de un lenguaje gráfico a DL y/o el uso de varios lenguajes textuales como de varios formatos de salida. Por ejemplo, el programador puede crear una traducción que genere la formalización en Lógica Descriptiva en formato HTML y en OWL 2 para que sea legible para el usuario. Además, las consultas a realizar al razonador pueden ser expandidas para ofrecer otros servicios y, por consecuencia, la interpretación de las respuestas por parte del módulo `Analizador de Respuestas` contempla el análisis de estas nuevas consultas y la intención del programador. Finalmente, el módulo `Razonador` puede cambiar y/o incorporar más razonadores con el fin de comparar los resultados de cada uno de ellos o de brindar soporte a cualidades específicas que poseen estos.

Estos cambios pueden ser realizables en la implementación propuesta de *crowd*, dado que el diseño es extensible con estas características, con impacto sobre la menor cantidad de módulos posibles. Sin embargo, todas estas propuestas quedaron fuera del alcance de esta tesis y sólo han sido contempladas para próximos trabajos.

Apéndice A

Secuencia de Entradas y Salidas de la Implementación del Prototipo

A.1. Ejemplo de un Documento OWL en Varias Sintaxis

Considere el Ejemplo 2.2 donde se presenta una ontología en lógica descriptiva $\mathcal{ALCQI}$. En las Figuras A.1 y A.2 se presenta la misma lógica en sintaxis XML, Turtle.

A.2. Ejemplo Satisfacible

En este capítulo se describirá la secuencia de entradas y salidas de cada módulo, considerando como el modelo conceptual de entrada dado en la Figura 4.9.

A.2.1. Modelo JSON Mantenido por El Cliente

El motor Javascript mantendrá el siguiente modelo JSON, el cual será enviado al servidor ante la petición de chequeo de consistencia.


```
{"classes":[
  {"name":"PhoneBill","attrs":[],"methods":[],
   "position":{"x":20,"y":20}},
  {"name":"PhoneCall","attrs":[],"methods":[],
   "position":{"x":352,"y":28}},
  {"name":"Phone","attrs":[],"methods":[],
   "position":{"x":685,"y":28}},
  {"name":"MobileCall","attrs":[],"methods":[],
   "position":{"x":352,"y":191}},
  {"name":"CellPhone","attrs":[],"methods":[],
   "position":{"x":608,"y":183}},
  {"name":"FixedPhone","attrs":[],"methods":[],
   "position":{"x":780,"y":187}}],
"links":[
  {"name":"r5",
   "classes":["PhoneBill","PhoneCall"],
   "multiplicity":["1..1","1..*"],
   "type":"association"},
  {"name":"r6","classes":["PhoneCall","Phone"],
   "multiplicity":[null,"1..1"],
    "type":"association"},
  {"name":"r7","classes":["MobileCall"],
   "multiplicity":null,"type":"generalization",
   "parent":"PhoneCall","constraint":[]},
  {"name":"r8",
    "classes":["MobileCall","CellPhone"],
   "multiplicity":[null,null],
   "type":"association"},
  {"name":"r9",
   "classes": ["CellPhone","FixedPhone"],
   "multiplicity":null,"type":"generalization",
   "parent":"Phone","constraint":
      ["disjoint","covering"]}],
"owllink":""}
```


Figura A.3: Representación JSON del modelo conceptual de la Figura 4.9

A.2.2. Representación OWL 2

Posteriormente, el modelo conceptual es traducido a lógica descriptiva utilizando la sintaxis XML basada en OWL 2.

```
<!DOCTYPE Ontology [
<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
<!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
]>

<Ontology xml:base="http://example.com/owl/families/" ontologyIRI="http://example.com/owl/families"
  xmlns="http://www.w3.org/2002/07/owl#">
  <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
  <Prefix name="mydom" IRI="http://myhost.com/myontology#"/>

  <Declaration><Class abbreviatedIRI="mydom:Empleado"/></Declaration>
  <Declaration><Class abbreviatedIRI="mydom:Manager"/></Declaration>
  <Declaration><Class abbreviatedIRI="mydom:AreaManager"/></Declaration>
  <Declaration><Class abbreviatedIRI="mydom:TopManager"/></Declaration>
  <Declaration><Class abbreviatedIRI="mydom:Proyecto"/></Declaration>
  <Declaration><ObjectProperty abbreviatedIRI="mydom:trabajaPara"/></Declaration>
  <Declaration><ObjectProperty abbreviatedIRI="mydom:gestiona"/></Declaration>

  <SubClassOf><!-- 1 -->
    <Class abbreviatedIRI="mydom:Manager"/><Class abbreviatedIRI="mydom:Empleado"/></SubClassOf>
  <SubClassOf><!-- 2 -->
    <Class abbreviatedIRI="mydom:AreaManager"/><Class abbreviatedIRI="mydom:Manager"/></SubClassOf>
  <SubClassOf><!-- 3 -->
    <Class abbreviatedIRI="mydom:TopManager"/><Class abbreviatedIRI="mydom:Manager"/></SubClassOf>
  <SubClassOf><!-- 4 -->
    <Class abbreviatedIRI="mydom:Manager"/>
    <ObjectUnionOf><Class abbreviatedIRI="mydom:AreaManager"/><Class abbreviatedIRI="mydom:TopManager"/>
  </ObjectUnionOf></SubClassOf>
  <SubClassOf><!-- 5 -->
    <ObjectIntersectionOf><Class abbreviatedIRI="mydom:AreaManager"/>
    <Class abbreviatedIRI="mydom:TopManager"/></ObjectIntersectionOf>
    <Class abbreviatedIRI="Nothing"/></SubClassOf>
  <SubClassOf><!-- 6 -->
    <ObjectSomeValuesFrom><ObjectProperty abbreviatedIRI="mydom:trabajaPara"/>
    <Class abbreviatedIRI="Thing"/></ObjectSomeValuesFrom>
    <Class abbreviatedIRI="mydom:Empleado"/>
  </SubClassOf>
  <SubClassOf><!-- 7 -->
    <ObjectSomeValuesFrom><ObjectInverseOf><ObjectProperty abbreviatedIRI="mydom:trabajaPara"/>
      </ObjectInverseOf><Class abbreviatedIRI="Thing"/></ObjectSomeValuesFrom>
    <Class abbreviatedIRI="mydom:Proyecto"/></SubClassOf>
  <SubClassOf><!-- 8 -->
    <Class abbreviatedIRI="mydom:Proyecto"/><ObjectSomeValuesFrom>
      <ObjectInverseOf><ObjectProperty abbreviatedIRI="mydom:trabajaPara"/></ObjectInverseOf>
      <Class abbreviatedIRI="Thing"/></ObjectSomeValuesFrom></SubClassOf>
  <SubClassOf><!-- 9 -->
    <ObjectSomeValuesFrom><ObjectProperty abbreviatedIRI="mydom:gestiona"/>
      <Class abbreviatedIRI="Thing"/></ObjectSomeValuesFrom>
    <Class abbreviatedIRI="mydom:TopManager"/></SubClassOf>
  <SubClassOf><!-- 10 -->
    <ObjectSomeValuesFrom><ObjectInverseOf><ObjectProperty abbreviatedIRI="mydom:gestiona"/>
      </ObjectInverseOf><Class abbreviatedIRI="Thing"/></ObjectSomeValuesFrom>
      <Class abbreviatedIRI="mydom:Proyecto"/></SubClassOf>
  <SubClassOf><!-- 11 -->
    <Class abbreviatedIRI="mydom:Proyecto" /><ObjectSomeValuesFrom><ObjectInverseOf>
<ObjectProperty abbreviatedIRI="mydom:gestiona"/></ObjectInverseOf>
      <Class abbreviatedIRI="Thing"/></ObjectSomeValuesFrom></SubClassOf>
  <SubClassOf><!-- 12 -->
    <Class abbreviatedIRI="mydom:TopManager" /><ObjectSomeValuesFrom>
      <ObjectProperty abbreviatedIRI="mydom:gestiona"/><Class abbreviatedIRI="Thing"/>
      </ObjectSomeValuesFrom></SubClassOf>
  <SubClassOf><!-- 13 -->
    <ObjectMaxCardinality cardinality="2"><ObjectProperty abbreviatedIRI="mydom:gestiona"/>
      <Class abbreviatedIRI="Thing" /></ObjectMaxCardinality>
    <Class abbreviatedIRI="Nothing" /></SubClassOf>
  <SubClassOf><!-- 14 -->
    <ObjectMaxCardinality cardinality="2"><ObjectInverseOf>
      <ObjectProperty abbreviatedIRI="mydom:gestiona"/></ObjectInverseOf>
      <Class abbreviatedIRI="Thing"/></ObjectMaxCardinality><Class abbreviatedIRI="Nothing"/>
   </SubClassOf></Ontology>
```

Figura A.1: Ejemplo de un documento OWL que representa la KB dada en el Ejemplo 2.2.

```
@prefix : <http://myhost.com/myontology#> .
@prefix otherOnt: <http://example.org/otherOntologies/families/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

:Empleado rdf:type owl:Class .
:Manager rdf:type owl:Class .
:AreaManager rdf:type owl:Class .
:TopManager rdf:type owl:Class .
:Proyecto rdf:type owl:Class .
:gestiona rdf:type owl:ObjectProperty .
:gestiona rdfs:domain :TopManager ;
    rdfs:range :Proyecto .
:trabajaPara rdf:type owl:ObjectProperty .
:trabajaPara rdfs:domain :Empleado ;
    rdfs:range :Proyecto .

:Manager rdfs:subClassOf :Empleado .                                                    # 1
:AreaManager rdfs:subClassOf :Manager .                                                 # 2
:TopManager rdfs:subClassOf :Manager .                                                  # 3
:Manager rdfs:subClassOf [                                                              # 4
   rdf:type owl:Class ; owl:unionOf ( :AreaManager :TopManager ) ] .
[] rdf:type owl:Class ; owl:intersectionOf ( :AreaManager :AreaManager ) ;              # 5
    rdfs:subClassOf [ rdf:type owl:Class ; owl:Nothing ] .
[] rdf:type owl:Class ;                                                                 # 6
    [ rdf:type owl:Restriction ; owl:onProperty :trabajaPara ; owl:someValuesFrom owl:Thing ]
    rdfs:subClassOf [ rdf:type owl:Class ; :Empleado ] .
[] rdf:type owl:Class ;                                                                 # 7
    [ rdf:type owl:Restriction ; owl:onProperty [ owl:inverseOf :trabajaPara ] ;
      owl:someValuesFrom owl:Thing ]
    rdfs:subClassOf [ rdf:type owl:Class ; :Proyecto ] .
:Proyecto rdfs:subClassOf [                                                             # 8
     rdf:type owl:Class ; [ rdf:type owl:Restriction ;
          owl:onProperty [ owl:inverseOf :trabajaPara ] ; owl:someValuesFrom owl:Thing ] ] .
[] rdf:type owl:Class ;                                                                 # 9
    [ rdf:type owl:Restriction ; owl:onProperty :gestiona ; owl:someValuesFrom owl:Thing ]
    rdfs:subClassOf [ rdf:type owl:Class ; :TopManager ] .
[] rdf:type owl:Class ; [ rdf:type owl:Restriction ;                                    # 10
      owl:onProperty [ owl:inverseOf :gestiona ] ; owl:someValuesFrom owl:Thing ]
    rdfs:subClassOf [ rdf:type owl:Class ; :Proyecto ] .
:Proyecto rdfs:subClassOf [                                                             # 11
    rdf:type owl:Class ; [ rdf:type owl:Restriction ;
        owl:onProperty [ owl:inverseOf :gestiona ] ; owl:someValuesFrom owl:Thing ] ] .
:TopManager rdfs:subClassOf [                                                           # 12
     rdf:type owl:Class ;
     [ rdf:type owl:Restriction ; owl:onProperty :gestiona ; owl:someValuesFrom owl:Thing ] ] .
[] rdf:type owl:Class ;                                                                 # 13
    [ rdf:type owl:Restriction ; owl:maxCardinality  "2"^^xsd:nonNegativeInteger ;
      owl:onProperty :gestiona ] ; rdfs:subClassOf owl:Nothing .
[] rdf:type owl:Class ;                                                                 # 14
    [ rdf:type owl:Restriction ; owl:maxCardinality  "2"^^xsd:nonNegativeInteger ;
        owl:onProperty [ owl:inverseOf :gestiona ] ] ; rdfs:subClassOf owl:Nothing .
```

Figura A.2: Ejemplo de un documento OWL usando la sintaxis Turtle que representa la KB dada en el Ejemplo 2.2.

```
<SubClassOf><Class IRI="PhoneBill"></Class><Class abbreviatedIRI="Thing"></Class></SubClassOf>
<SubClassOf><Class IRI="PhoneCall"></Class><Class abbreviatedIRI="Thing"></Class></SubClassOf>
<SubClassOf><Class IRI="Phone"></Class><Class abbreviatedIRI="Thing"></Class></SubClassOf>
<SubClassOf><Class IRI="MobileCall"></Class><Class abbreviatedIRI="Thing"></Class></SubClassOf>
<SubClassOf><Class IRI="CellPhone"></Class><Class abbreviatedIRI="Thing"></Class></SubClassOf>
<SubClassOf><Class IRI="FixedPhone"></Class><Class abbreviatedIRI="Thing"></Class></SubClassOf>
<SubClassOf>
  <Class abbreviatedIRI="Thing"></Class>
  <ObjectIntersectionOf>
    <ObjectAllValuesFrom>
      <ObjectProperty IRI="r5"></ObjectProperty><Class IRI="PhoneBill"></Class>
    </ObjectAllValuesFrom>
    <ObjectAllValuesFrom>
      <ObjectInverseOf><ObjectProperty IRI="r5"></ObjectProperty></ObjectInverseOf>
      <Class IRI="PhoneCall"></Class>
    </ObjectAllValuesFrom>
  </ObjectIntersectionOf>
</SubClassOf>
<SubClassOf><Class IRI="PhoneBill"></Class>
  <ObjectMinCardinality cardinality="1"><ObjectProperty IRI="r5"></ObjectProperty></ObjectMinCardinality>
</SubClassOf>
<SubClassOf><Class IRI="PhoneCall"></Class>
  <ObjectIntersectionOf><ObjectMinCardinality cardinality="1">
    <ObjectInverseOf><ObjectProperty IRI="r5"></ObjectProperty></ObjectInverseOf>
    </ObjectMinCardinality><ObjectMaxCardinality cardinality="1">
      <ObjectInverseOf><ObjectProperty IRI="r5"></ObjectProperty></ObjectInverseOf>
    </ObjectMaxCardinality>
  </ObjectIntersectionOf>
</SubClassOf>
<SubClassOf><Class abbreviatedIRI="Thing"></Class>
  <ObjectIntersectionOf>
    <ObjectAllValuesFrom>
      <ObjectProperty IRI="r6"></ObjectProperty><Class IRI="PhoneCall"></Class>
    </ObjectAllValuesFrom>
    <ObjectAllValuesFrom>
      <ObjectInverseOf><ObjectProperty IRI="r6"></ObjectProperty></ObjectInverseOf>
      <Class IRI="Phone"></Class>
    </ObjectAllValuesFrom>
  </ObjectIntersectionOf>
</SubClassOf>
<SubClassOf>
  <Class IRI="PhoneCall"></Class>
  <ObjectIntersectionOf>
    <ObjectMinCardinality cardinality="1"><ObjectProperty IRI="r6"></ObjectProperty></ObjectMinCardinality>
    <ObjectMaxCardinality cardinality="1"><ObjectProperty IRI="r6"></ObjectProperty></ObjectMaxCardinality>
  </ObjectIntersectionOf>
</SubClassOf>
<SubClassOf><Class IRI="MobileCall"></Class><Class IRI="PhoneCall"></Class></SubClassOf>
<SubClassOf>
  <Class abbreviatedIRI="Thing"></Class>
  <ObjectIntersectionOf>
    <ObjectAllValuesFrom>
      <ObjectProperty IRI="r8"></ObjectProperty><Class IRI="MobileCall"></Class>
    </ObjectAllValuesFrom>
    <ObjectAllValuesFrom>
      <ObjectInverseOf><ObjectProperty IRI="r8"></ObjectProperty></ObjectInverseOf>
    <Class IRI="CellPhone"></Class>
    </ObjectAllValuesFrom>
  </ObjectIntersectionOf>
</SubClassOf>
<SubClassOf><Class IRI="CellPhone"></Class><Class IRI="Phone"></Class></SubClassOf>
<SubClassOf><Class IRI="FixedPhone"></Class><Class IRI="Phone"></Class></SubClassOf>
<SubClassOf>
  <Class IRI="CellPhone"></Class>
  <ObjectComplementOf><Class IRI="FixedPhone"></Class></ObjectComplementOf>
</SubClassOf>
<SubClassOf>
  <Class IRI="Phone"></Class>
  <ObjectUnionOf><Class IRI="CellPhone"></Class><Class IRI="FixedPhone"></Class></ObjectUnionOf>
</SubClassOf>
```

Figura A.4: Transformación del modelo de la Figura 4.9 a XML OWLlink.

A.2.3. Incorporación de las Consultas

Posteriormente el módulo `Generador de Consultas` completa el OWL 2 con consultas OWLlink. La Figura A.5 muestra las consultas incorporadas. Finalmente, se completa el OWLlink incorporando las etiquetas XML restantes que se observan en la Figura A.6, donde cada comentario XML son sustituidos por los códigos respectivos.

```
<IsKBSatisfiable kb="http://localhost/kb1"></IsKBSatisfiable>
<IsClassSatisfiable kb="http://localhost/kb1"><owl:Class IRI="PhoneBill"></owl:Class></IsClassSatisfiable>
<IsClassSatisfiable kb="http://localhost/kb1"><owl:Class IRI="PhoneCall"></owl:Class></IsClassSatisfiable>
<IsClassSatisfiable kb="http://localhost/kb1"><owl:Class IRI="Phone"></owl:Class></IsClassSatisfiable>
<IsClassSatisfiable kb="http://localhost/kb1"><owl:Class IRI="MobileCall"></owl:Class></IsClassSatisfiable>
<IsClassSatisfiable kb="http://localhost/kb1"><owl:Class IRI="CellPhone"></owl:Class></IsClassSatisfiable>
<IsClassSatisfiable kb="http://localhost/kb1"><owl:Class IRI="FixedPhone"></owl:Class></IsClassSatisfiable>
```

Figura A.5: Consultas a realizarse para el modelo de la Figura 4.9 en OWLlink.

```
<?xml version="1.0" encoding="UTF-8"?>
<RequestMessage xmlns="http://www.owllink.org/owllink#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.owllink.org/owllink# http://www.owllink.org/owllink-20091116.xsd">
    <CreateKB kb="http://localhost/kb1"></CreateKB>
    <Tell kb="http://localhost/kb1">
        <!-- Representación del modelo en OWL 2. -->
    </Tell>
    <!-- OWLlink escrito por el usuario. -->
    <!-- Consultas OWLlink al razonador. -->
</RequestMessage>
```

Figura A.6: Template general para localizar las distintas secciones dentro del documento OWLlink.

A.2.4. Respuesta del Razonador y Análisis de la Misma

El razonador recibe como entrada todo el OWLlink generado, procesando las consultas completas y emitiendo la respuesta de la Figura A.7. Éste es procesado junto con las consultas de la Figura A.5 en el módulo `Analizador de Respuestas` generando una salida JSON de la Figura A.8, que será enviada al cliente.

```
<?xml version="1.0" encoding="UTF-8"?>
<ResponseMessage xmlns="http://www.owllink.org/owllink#"
    xmlns:owl="http://www.w3.org/2002/07/owl#">
    <KB kb="http://localhost/kb1"></KB>
    <OK></OK>
    <BooleanResponse result="true"></BooleanResponse>
    <BooleanResponse result="true"></BooleanResponse>
    <BooleanResponse result="true"></BooleanResponse>
    <BooleanResponse result="true"></BooleanResponse>
    <BooleanResponse result="true"></BooleanResponse>
    <BooleanResponse result="true"></BooleanResponse>
    <BooleanResponse result="true"></BooleanResponse>
</ResponseMessage>
```

Figura A.7: Respuesta del razonador ante las consultas de la Figura A.5.

```
{"satisfiable":{"kb":true, "classes": [
     "PhoneBill","PhoneCall", "Phone","MobileCall",
     "CellPhone","FixedPhone"]},
   "unsatisfiable":{"classes":[]},
   "suggestions":{"links":[]},
   "reasoner":{
     "input":"XML OWLlink que recibe el razonador.",
     "output":"XML OWLlink que responde el razonador."}}
```


Figura A.8: JSON de respuesta para el cliente procesado a partir de los datos de la Figura A.7.

A.3. Ejemplo Insatisfacible

Este ejemplo es similar al anterior, pero incorpora una generalización entre las clases "CellPhone" y "FixedPhone" como ilustra la Figura 4.10. En las siguientes secciones se presentan las diferencias y los códigos completos de entrada y salida para cada módulo.

A.3.1. Modelo JSON Mantenido por El Cliente

En la Figura A.9 se muestra el retazo de código necesario para incorporar la generalización entre "CellPhone" y "FixedPhone", y a continuación, el código completo para enviarse al servidor.


```
// ------------------------------------------------
// Insertado:
{"name":"r10","classes":["CellPhone"],
 "multiplicity":null,"type":"generalization","parent":"FixedPhone","constraint":[]}],
// ------------------------------------------------
```

```
{"classes":[
  {"name":"PhoneBill","attrs":[],"methods":[],
   "position":{"x":20,"y":20}},
  {"name":"PhoneCall","attrs":[],"methods":[],
   "position":{"x":352,"y":28}},
  {"name":"Phone","attrs":[],"methods":[],
   "position":{"x":685,"y":28}},
  {"name":"MobileCall","attrs":[],"methods":[],
   "position":{"x":352,"y":191}},
  {"name":"CellPhone","attrs":[],"methods":[],
   "position":{"x":608,"y":183}},
  {"name":"FixedPhone","attrs":[],"methods":[],
   "position":{"x":780,"y":187}}],
```

```
"links":[
  {"name":"r5",
   "classes":["PhoneBill","PhoneCall"],
   "multiplicity":["1..1","1..*"],
   "type":"association"},
  {"name":"r6",
   "classes":["PhoneCall","Phone"],
   "multiplicity":[null,"1..1"],
   "type":"association"},
  {"name":"r7",
   "classes":["MobileCall"],
   "multiplicity":null,"type":"generalization",
   "parent":"PhoneCall","constraint":[]},
  {"name":"r8",
   "classes":["MobileCall","CellPhone"],
   "multiplicity":[null,null],
   "type":"association"},
  {"name":"r9",
   "classes":["CellPhone","FixedPhone"],
   "multiplicity":null,"type":"generalization",
   "parent":"Phone",
   "constraint":["disjoint","covering"]}
  {"name":"r10","classes":["CellPhone"],
   "multiplicity":null,
   "type":"generalization",
   "parent":"FixedPhone","constraint":[]}],
"owllink":""}
```


Figura A.9: Retazo de código JSON insertado para representar la generalización agregada a la Figura 4.9 y el código resultante.

A.3.2. Representación OWL 2

El código XML a introducirse al razonador también sufre modificaciones, puesto que la nueva generalización también debe ser representada en lógica descriptiva. La Figura A.10 muestra estas

modificaciones donde sólo incorpora tres líneas al final representando la siguiente sentencia DL respetando el método de traducción utilizado por el prototipo: $CellPhone \sqsubseteq FixedPhone$.

```
<!-- [...] -->
<SubClassOf><Class IRI="CellPhone"></Class><Class IRI="Phone"></Class></SubClassOf>
<SubClassOf><Class IRI="FixedPhone"></Class><Class IRI="Phone"></Class></SubClassOf>
<SubClassOf>
    <Class IRI="CellPhone"></Class>
    <ObjectComplementOf><Class IRI="FixedPhone"></Class></ObjectComplementOf>
</SubClassOf>
<SubClassOf>
    <Class IRI="Phone"></Class>
    <ObjectUnionOf><Class IRI="CellPhone"></Class><Class IRI="FixedPhone"></Class></ObjectUnionOf>
</SubClassOf>
<!-- Se inserta el siguiente código: -->
<SubClassOf>
    <Class IRI="CellPhone"></Class><Class IRI="FixedPhone"></Class>
</SubClassOf>
<!-- Final de la sección OWL 2 -->
```

Figura A.10: Código OWl 2 insertado para representar la modificación de la Figura 4.10.

A.3.3. Respuesta del Razonador y Análisis de la Misma

El razonador responderá de manera similar a la Figura A.7, exceptuando por la consistencia en la clase "CellPhone", la cual retorna como falso. Las consultas son las mismas que las anteriores (Figura A.5 puesto que no hay modificaciones en la cantidad de clases. El módulo `Analizador de Respuestas` indica esta diferencia agregando la clase al conjunto del campo "unsatisfiable".

```
<?xml version="1.0" encoding="UTF-8"?>
<ResponseMessage xmlns="http://www.owllink.org/owllink#"
    xmlns:owl="http://www.w3.org/2002/07/owl#">
    <KB kb="http://localhost/kb1"></KB>
    <OK></OK>
    <BooleanResponse result="true"  warning=
"Unsatisfiable classes: (*BOTTOM* BOTTOM file:///var/www/html/wicom/run/input-file.owllinkCellPhone)">
    </BooleanResponse>
    <BooleanResponse result="true"></BooleanResponse>
    <BooleanResponse result="true"></BooleanResponse>
    <BooleanResponse result="true"></BooleanResponse>
    <BooleanResponse result="true"></BooleanResponse>
    <BooleanResponse result="false"></BooleanResponse>
    <BooleanResponse result="true"></BooleanResponse>
</ResponseMessage>
```

Figura A.11: Respuesta del razonador ante la consulta de satisfacibilidad correspondiente al modelo de la Figura 4.10.

```
{"satisfiable":{"kb":true, "classes": [
     "PhoneBill","PhoneCall","Phone","MobileCall",
     "FixedPhone"]},
   "unsatisfiable":{"classes":["CellPhone"]},
   "suggestions":{"links":[]},
   "reasoner":{
     "input":"XML OWLlink que recibe el razonador.",
     "output":"XML OWLlink que responde el razonador."}}
```

Figura A.12: Texto en formato JSON producida a partir de la respuesta del razonador de la Figura A.11

Bibliografía

[1] Abrial, Jean-Raymond, Stephen A. Schuman y Bertrand Meyer: *Specification Language*. En *On the Construction of Programs*, páginas 343–410. 1980.

[2] Agüero, Matías y Carlos Alvez: *Ontología para la Recuperación Semántica de Imágenes DICOM en Bases de Datos Objeto-Relacional*. En *the 2nd Simposio Argentino de Ontologías y sus Aplicaciones SAOA '16 JAIIO '16*, 2016.

[3] Allemang, Dean y James A. Hendler: *Semantic Web for the working ontologist effective modeling in RDFS and OWL*. Morgan Kaufmann/Elsevier, Waltham, MA, 2011, ISBN 9780123859662 0123859662. http://site.ebrary.com/id/10468915.

[4] Artale, Alessandro, Diego Calvanese, Roman Kontchakov, Vladislav Ryzhikov y Michael Zakharyaschev: *Complexity of Reasoning in Entity Relationship Models*. En *Proceedings of the 2007 International Workshop on Description Logics (DL)*, 2007.

[5] Baader, Franz, Sebastian Brandt y Carsten Lutz: *Pushing the EL Envelope*. En *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, páginas 364–369, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc. http://dl.acm.org/citation.cfm?id=1642293.1642351.

[6] Baader, Franz, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi y Peter F. Patel-Schneider (editores): *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003, ISBN 0-521-78176-0.

[7] Baader, Franz, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi y Peter F. Patel-Schneider (editores): *The Description Logic Handbook: Theory, Implementation, and Applications*, capítulo Appendix 1. Cambridge University Press, New York, NY, USA, 2003, ISBN 0-521-78176-0.

[8] Baader, Franz, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi y Peter F. Patel-Schneider (editores): *The Description Logic Handbook: Theory, Implementation, and Applications*, capítulo 2. Cambridge University Press, New York, NY, USA, 2003, ISBN 0-521-78176-0.

[9] Baader, Franz, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi y Peter F. Patel-Schneider (editores): *The Description Logic Handbook: Theory, Implementation, and Applications*, capítulo 1. Cambridge University Press, New York, NY, USA, 2003, ISBN 0-521-78176-0.

[10] Baader, Franz, Ian Horrocks y Ulrike Sattler: *Description Logics*. En *Handbook of Knowledge Representation*, capítulo 3, páginas 135–180. Elsevier, 2008. download/2007/BaHS07a.pdf.

[11] Baader, Franz, Carsten Lutz y Sebastian Brandt: *Pushing the EL Envelope Further*. En *Proceedings of the Fourth OWLED Workshop on OWL: Experiences and Directions, Washington, DC, USA, 1-2 April 2008.*, 2008. http://ceur-ws.org/Vol-496/owled2008dc_paper_3.pdf.

[12] Bao, Jie, Elisa F. Kendall, Deborah L. McGuinness y Peter F. Patel-Schneider (editores): *OWL 2 Web Ontology Language Quick Reference Guide (Second Edition)*. W3C Recommendation, 2ª edición, Diciembre 2012. Disponible en https://www.w3.org/TR/owl-quick-reference/.

[13] Barzdins, Janis, Guntis Barzdins, Karlis Cerans, Renars Liepins y Arturs Sprogis: *UML Style Graphical Notation and Editor for OWL 2*. En *Perspectives in Business Informatics Research - 9th International Conference, BIR 2010, Rostock Germany, September 29-October 1, 2010. Proceedings*, páginas 102–114, 2010. https://doi.org/10.1007/978-3-642-16101-8_9.

[14] Bechhofer, S., R. Moller y P. Crowther: *The DIG Description Logic Interface*. En *In Proc. of International Workshop on Description Logics (DL2003)*, 2003.

[15] Bechhofer, Sean, Thorsten Liebig, Marko Luther, Olaf Noppens, Peter F. Patel-Schneider, Boontawee Suntisrivaraporn, Anni-Yasmin Turhan y Timo Weithöner: *DIG 2.0 – Towards a Flexible Interface for Description Logic Reasoners*. En Grau, Bernardo Cuenca, Pascal Hitzler, Conor Shankey y Evan Wallace (editores): *Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions, Athens, Georgia, USA, November 10-11, 2006*, volumen 216 de *CEUR Workshop Proceedings*. CEUR-WS.org, 2006. http://ceur-ws.org/Vol-216/submission_3.pdf.

[16] Beckett, Dave y Brian McBride (editores): *RDF/XML Syntax Specification (Revised)*. W3C Recommendation, Febrero 2004. Disponible en https://www.w3.org/TR/rdf-syntax-grammar/.

[17] Beckett, David y Tim Berners-Lee: *Turtle - Terse RDF Triple Language*. W3C Recommendation, Marzo 2011. Disponible en https://www.w3.org/TeamSubmission/turtle/.

[18] Beckett, David, Tim Berners-Lee, Eric Prud'hommeaux y Gavin Carothers: *RDF 1.1 Turtle*. World Wide Web Consortium, February 2014. https://www.w3.org/TR/turtle/.

[19] Berardi, Daniela, Diego Calvanese y Giuseppe De Giacomo: *Reasoning on UML class diagrams*. Artif. Intell., 168(1-2):70–118, 2005. http://dx.doi.org/10.1016/j.artint.2005.05.003.

[20] Berners-Lee, T., J. Hendler y O. Lassila: *The Semantic Web*. Scientific American, Mayo 2001. http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21.

[21] Booch, Grady, James Rumbaugh y Ivar Jacobson: *Unified Modeling Language User Guide*. Addison-Wesley Professional, 2005, ISBN 0321267974.

[22] Brachman, Ronald J. y Hector J. Levesque (editores): *Readings in Knowledge Representation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1985, ISBN 093461301X.

[23] Braun, Germán, Christian Gimenez, Laura Cecchi y Pablo Fillottrani: *Towards a Visualisation Process for Ontology-Based Conceptual Modelling*. En Baracho, Renata Maria Abrantes, Seiji Isotani y Maurício Barcellos Almeida (editores): *ONTOBRAS – Brazilian Ontology Research Seminar (ONTOBRAS)*, número 1862 en *CEUR Workshop Proceedings*, páginas 107–118, Aachen, 2016. http://ceur-ws.org/Vol-1862/#paper-09.

[24] Braun, Germán, Christian Gimenez, Pablo Fillottrani y Laura Cecchi: *Towards Conceptual Modelling Interoperability in a Web Tool for Ontology Engineering.* En *the 3rd Simposio Argentino de Ontologías y sus Aplicaciones SAOA '17 JAIIO '17*, 2017.

[25] Calvanese, Diego, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini y Riccardo Rosati: *Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family.* Journal of Automated Reasoning, 39(3):385–429, Oct 2007, ISSN 1573-0670. http://dx.doi.org/10.1007/s10817-007-9078-x.

[26] Calvanese, Diego, Giuseppe De Giacomo, Maurizio Lenzerini, Daniele Nardi y Riccardo Rosati: *Description Logic Framework for Information Integration.* En *KR-98*, 1998.

[27] Caracciolo, Caterina, Armando Stellato, Ahsan Morshed, Gudrun Johannsen, Sachit Rajbhandari, Yves Jaques y Johannes Keizer: *The AGROVOC Linked Dataset.* Semantic Web, 4(3):341–348, 2013. https://doi.org/10.3233/SW-130106.

[28] Cerans, Karlis, Julija Ovcinnikova, Renars Liepins y Arturs Sprogis: *Advanced OWL 2.0 Ontology Visualization in OWLGrEd.* En *DB&IS*, Frontiers in Artificial Intelligence and Applications. IOS Press, 2012.

[29] Complak, Wojciech, Adam Wojciechowski, Alok Mishra y Deepti Mishra: *Use Cases and Object Modelling Using ArgoUML.* En *On the Move to Meaningful Internet Systems: OTM 2011 Workshops - Confederated International Workshops and Posters: EI2N+NSF ICE, ICSP+INBAST, ISDE, ORM, OTMA, SWWS+MONET+SeDeS, and VADER 2011, Hersonissos, Crete, Greece, October 17-21, 2011. Proceedings*, páginas 246–255, 2011. https://doi.org/10.1007/978-3-642-25126-9_35.

[30] Console, Marco, Domenico Lembo, Valerio Santarelli y Domenico Fabio Savo: *Graphical Representation of OWL 2 Ontologies Through Graphol.* En *Proceedings of the 2014 International Conference on Posters & Demonstrations Track - Volume 1272*, ISWC-PD'14, páginas 73–76, Aachen, Germany, Germany, 2014. CEUR-WS.org. http://dl.acm.org/citation.cfm?id=2878453.2878472.

[31] Console, Marco, Domenico Lembo, Valerio Santarelli y Domenico Fabio Savo: *Graphol: Ontology Representation through Diagrams.* En *Informal Proceedings of the 27th International DL*, 2014.

[32] Consortium, The Gene Ontology: *Expansion of the Gene Ontology knowledgebase and resources.* Nucleic Acids Research, 45(Database-Issue):D331–D338, 2017. https://doi.org/10.1093/nar/gkw1108.

[33] Curland, Matthew y Terry A. Halpin: *The NORMA Software Tool for ORM 2.* En *CAiSE Forum*, Lecture Notes in Business Information Processing. Springer, 2010.

[34] Dickinson, Ian: *Implementation Experience with the DIG 1.1 Specification.* Informe técnico, Hewlett Packard, Digital Media Systems Laboratory, Bristol, May 2004.

[35] Falconer, Sean: *OntoGraf.* http://protegewiki.stanford.edu/wiki/OntoGraf accedido en julio 2017.

[36] Fillottrani, P., E. Franconi y S. Tessaris: *The new ICOM Ontology Editor.* En *Description Logics*, CEUR Workshop Proceedings. CEUR-WS.org, 2006. http://dblp.uni-trier.de/db/conf/dlog/dlog2006.html#FillottraniFT06.

[37] Fillottrani, P., E. Franconi y S. Tessaris: *The ICOM 3.0 intelligent conceptual modelling tool and methodology.* Semantic Web, 2012.

[38] Fillottrani, Pablo R., Enrico Franconi y Sergio Tessaris: *The new ICOM Ontology Editor*. En Parsia, Bijan, Ulrike Sattler y David Toman (editores): *Proceedings of the 2006 International Workshop on Description Logics (DL2006), Windermere, Lake District, UK, May 30 - June 1, 2006*, volumen 189 de *CEUR Workshop Proceedings*. CEUR-WS.org, 2006. http://ceur-ws.org/Vol-189/submission_38.pdf.

[39] Fillottrani, Pablo R., Enrico Franconi y Sergio Tessaris: *Ontology Design and Integration with ICOM 3.0 - Tool Description and Methodology*. En *Proceedings of the 24th International Workshop on Description Logics (DL)*, 2011.

[40] Fillottrani, Pablo R., Enrico Franconi y Sergio Tessaris: *The ICOM 3.0 intelligent conceptual modelling tool and methodology*. Semantic Web, 2012.

[41] Fillottrani, Pablo R. y C. Maria Keet: *KF metamodel formalization*. CoRR, abs/1412.6545, 2014. http://arxiv.org/abs/1412.6545.

[42] Fillottrani, Pablo Rubén y C. Maria Keet: *Conceptual Model Interoperability: A Metamodel-driven Approach*. En *Rules on the Web. From Theory to Applications - 8th International Symposium, RuleML 2014, Co-located with the 21st European Conference on Artificial Intelligence, ECAI 2014, Prague, Czech Republic, August 18-20, 2014. Proceedings*, 2014.

[43] Franconi, Enrico, Alessandro Mosca y Dmitry Solomakhin: *ORM2: Formalisation and Encoding in OWL2*. En *On the Move to Meaningful Internet Systems: OTM 2012 Workshops, Confederated International Workshops: OTM Academy, Industry Case Studies Program, EI2N, INBAST, META4eS, OnToContent, ORM, SeDeS, SINCOM, and SOMOCO 2012*, 2012.

[44] Gamma, Erich, Richard Helm, Ralph Johnson y John Vlissides: *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995, ISBN 0-201-63361-2.

[45] Gogolla, Martin: *Extended Entity-Relationship Model: Fundamentals and Pragmatics*. Springer-Verlag, 1994.

[46] Grand, Mark: *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, volumen 1. John Wiley & Sons, Inc., New York, NY, USA, 2nd edición, 2002, ISBN 0471227293, 9780471227298.

[47] Grau, Bernardo Cuenca, Ian Horrocks, Boris Motik, Bijan Parsia, Peter Patel-Schneider y Ulrike Sattler: *OWL 2: The Next Step for OWL*. Web Semant., 2008.

[48] Group, Oject Management: *Object Constraint Language*. Object Management Group, February 2014.

[49] Guizzardi, G.: *Ontological foundations for structural conceptual models*. Tesis de Doctorado, University of Twente, Enschede, The Netherlands, Enschede, October 2005, ISBN 90-75176-81-3.

[50] Guizzardi, G. y G. Wagner: *Conceptual Simulation Modeling with onto-UML*. En *Proceedings of the WSC'12*, 2012.

[51] Haarslev, V. y R. Möller: *RACER System Description*. En Goré, R., A. Leitsch y T. Nipkow (editores): *International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, Siena, Italy*, páginas 701–705. Springer-Verlag, 2001.

[52] Haarslev, Volker, Kay Hidde, Ralf Möller y Michael Wessel: *The RacerPro knowledge representation and reasoning system*. Semantic Web, 2012.

[53] Halpin, Terry: *ORM 2*, páginas 676–687. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, ISBN 978-3-540-32132-3. http://dx.doi.org/10.1007/11575863_87.

[54] Halpin, Terry y Tony Morgan: *Information Modeling and Relational Databases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2ª edición, 2008, ISBN 0123735688, 9780123735683.

[55] Hasse, P., H. Lewen, R. Studer y M. Erdmann: *The NeOn Ontology Engineering Toolkit*. 2008.

[56] Hitzler, Pascal, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider y Sebastian Rudolph (editores): *OWL 2 Web Ontology Language: Primer*. W3C Recommendation, 27 October 2009. Available at http://www.w3.org/TR/owl2-primer/.

[57] Hladik, Jan: *A tableau system for the Description Logic $\mathcal{SHIO}$*. En Sattler, Ulrike (editor): *Contributions to the Doctoral Programme of the 2nd Int. Joint Conf. on Automated Reasoning (IJCAR 2004)*, volumen 106 de *CEUR Workshop Proceedings*, páginas 21–25. CEUR-WS.org, 2004.

[58] Horridge, Matthew: *OWLViz*. http://protegewiki.stanford.edu/wiki/OWLViz accedido en julio 2017.

[59] Horridge, Matthew y Peter F. Patel-Schneider: *OWL 2 Web Ontology Language Manchester Syntax (Second Edition)*. W3C Recommendation, 2ª edición, Diciembre 2012. Disponible en https://www.w3.org/TR/owl2-manchester-syntax/.

[60] Horrocks, Ian, Oliver Kutz y Ulrike Sattler: *The even more irresistible $\mathcal{SROIQ}$*. En Doherty, Patrick, John Mylopoulos y Christopher A. Welty (editores): *Proc. of the 10th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2006)*, páginas 57–67. AAAI Press, 2006.

[61] Horrocks, Ian y Ulrike Sattler: *Decidability of $\mathcal{SHIQ}$ with complex role inclusion axioms*. Artificial Intelligence, 160(1–2):79–104, 2004.

[62] IBM, Paul Zikopoulos y Chris Eaton: *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 1st edición, 2011, ISBN 0071790535, 9780071790536.

[63] Jackson, D.: *Software Abstractions: Logic, Language, and Analysis*. Books24x7.com, 2012, ISBN 9780262017152. https://books.google.com.ar/books?id=DDv8Ie_jBUQC.

[64] Kalyanput, A., B. Parsia, E. Sirin, B.C. Grau y J. Hendler: *Swoop: A "Web" Ontology Editing Browser*. Journal of Web Semantics, June 2005.

[65] Keet, C. Maria y Pablo Rubén Fillottrani: *An ontology-driven unifying metamodel of UML Class Diagrams, EER, and ORM2*. Data & Knowledge Engineering, 2015.

[66] Knublauch, H., R. Fergerson, N. Noy y M. Musen: *The Protégé OWL plugin: An open development environment for semantic web applications*. 2004.

[67] Krivov, Sergey, Richard Williams y Ferdinando Villa: *GrOWL: A tool for visualization and editing of OWL ontologies*. J. Web Sem., 2007.

[68] Krötzsch, Markus: *OWL 2 Profiles: An Introduction to Lightweight Ontology Languages*. En *Reasoning Web*, páginas 112–183, 2012.

[69] Kuhn, D.L.: *Selecting and effectively using a computer aided software engineering tool*. Jan 1989.

[70] Kunowski, Piotr y Tomasz Boiński: *SOVA*. http://protegewiki.stanford.edu/wiki/SOVA accedido en julio 2017.

[71] Lambrix, Patrick, He Tan, Vaida Jakoniene y Lena Strömbäck: *Biological Ontologies*, capítulo 4, páginas 85–99. Springer US, Boston, MA, 2007, ISBN 978-0-387-48438-9. https://doi.org/10.1007/978-0-387-48438-9_5.

[72] Liebig, Thorsten, Marko Luther, Mariano Rodriguez, Diego Calvanese, Michael Wessel, Ralf Möller, Matthew Horridge, Sean Bechhofer, Dmitry Tsarkov y Evren Sirin: *OWLlink: DIG for OWL 2*. En *In Proceedings of the 5th OWL Experiences and Directions Workshop (OWLED-2008)*, página 48, 2008.

[73] Lohmann, Steffen, Stefan Negru y David Bold: *The ProtégéVOWL Plugin: Ontology Visualization for Everyone*. En *Proceedings of ESWC 2014 Satellite Events*, volumen 8798 de *LNCS*, páginas 395–400. Springer, 2014.

[74] Lohmann, Steffen, Stefan Negru, Florian Haag y Thomas Ertl: *Visualizing Ontologies with VOWL*. Semantic Web, 7(4):399–419, 2016. http://dx.doi.org/10.3233/SW-150200.

[75] Minsky, Marvin: *A Framework for Representing Knowledge*. Informe técnico, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[76] Misra, Chandra: *1 Defining E-government: A Citizen-centric Criteria-based Approach*, 2006.

[77] Motik, B. y R. Studer: *KAON2–A Scalable Reasoning Tool for the Semantic Web*. En *Proceedings of the 2nd ESWC'05*, 2005.

[78] Motik, Boris, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue y Carsten Lutz (editores): *OWL 2 Web Ontology Language Profiles*. World Wide Web Consortium, second edition edición, December 2012. https://www.w3.org/TR/2012/REC-owl2-profiles-20121211/.

[79] Motik, Boris, Bijan Parsia y Peter F. Patel-Schneider (editores): *OWL 2 Web Ontology Language XML Serialization (Second Edition)*. W3C Recommendation, 2ª edición, Diciembre 2012. Disponible en https://www.w3.org/TR/owl2-xml-serialization/.

[80] Motik, Boris, Peter F. Patel-Schneider y Ian Horrocks: *OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax*. World Wide Web Consortium, April 2008. https://www.w3.org/TR/2008/WD-owl2-syntax-20080411/.

[81] Mouromtsev, Dmitry, Dmitry Pavlov, Yury Emelyanov, Alexey Morozov, Daniil Razdyakonov y Mikhail Galkin: *The Simple Web-based Tool for Visualization and Sharing of Semantic Data and Ontologies*. En *Proceedings of the ISWC 2015 Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC-2015), Bethlehem, PA, USA, October 11, 2015.*, 2015.

[82] Noy, Natalya F. y Deborah L. Mcguinness: *Ontology Development 101: A Guide to Creating Your First Ontology*. Informe técnico, Stanford University, 2001.

[83] OWL Working Group, W3C: *OWL 2 Web Ontology Language Profiles (Second Edition)*. W3C Recommendation, 11 December 2012. Available at http://www.w3.org/TR/2012/REC-owl2-profiles-20121211/.

[84] OWL Working Group, W3C: *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 27 October 2009. Available at http://www.w3.org/TR/owl2-overview/.

[85] Patel-Schneider, Peter F. y Ian Horrocks: *OWL 1.1 Web Ontology Language Overview*. W3C Recommendation, Diciembre 2006. Available at https://www.w3.org/Submission/owl11-overview/#ref-owl-1.1-specification.

[86] Patel-Schneider, Peter F. y Boris Motik (editores): *OWL 2 Web Ontology Language Mapping to RDF Graphs*. W3C Recommendation, 2ª edición, Diciembre 2012. Disponible en http://www.w3.org/TR/owl2-mapping-to-rdf/.

[87] Pereira, Rafael y Rui Mendes: *Current Trends in Bio-Ontologies and Data Integration*, páginas 579–586. Springer International Publishing, Cham, 2013, ISBN 978-3-319-00551-5. https://doi.org/10.1007/978-3-319-00551-5_69.

[88] Quillian, M. Ross: *Word concepts: a theory and simulation of some basic semantic capabilities*. Behavioral Science, 1967.

[89] Rosse, Cornelius y José L. V. Mejino: *The Foundational Model of Anatomy Ontology*, capítulo 4, páginas 59–117. Springer London, London, 2008, ISBN 978-1-84628-885-2. https://doi.org/10.1007/978-1-84628-885-2_4.

[90] Schach, Stephen R.: *Intro to Object-Oriented Analysis and Design with UML CD*. McGraw-Hill/Irwin, 1ª edición, Junio 2003, ISBN 978-0072939842, 0072939842.

[91] Schmidt-Schaußß, Manfred y Gert Smolka: *Attributive Concept Descriptions with Complements*. Artif. Intell., 48(1):1–26, Febrero 1991, ISSN 0004-3702. http://dx.doi.org/10.1016/0004-3702(91)90078-X.

[92] Schneider, Michael (editor): *OWL 2 Web Ontology Language RDF-Based Semantics (Second Edition)*. W3C Recommendation, 2ª edición, Diciembre 2012. Disponible en https://www.w3.org/TR/owl-rdf-based-semantics/.

[93] Schreiber, Guus, Yves Raimond, Frank Manola Eric Miller y Brian McBride (editores): *RDF 1.1 Primer*. World Wide Web Consortium, June 2014. https://www.w3.org/TR/rdf11-primer/.

[94] Schulz, Stefan, Holger Stenzhorn y Martin Boeker: *The ontology of biological taxa*. Bioinformatics, 24(13):i313–i321, 2008. +http://dx.doi.org/10.1093/bioinformatics/btn158.

[95] Shearer, Rob, Boris Motik y Ian Horrocks: *HermiT: A Highly-Efficient OWL Reasoner*. En Dolbear, Catherine, Alan Ruttenberg y Ulrike Sattler (editores): *Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions, collocated with the 7th International Semantic Web Conference (ISWC-2008), Karlsruhe, Germany, October 26-27, 2008*, volumen 432 de *CEUR Workshop Proceedings*. CEUR-WS.org, 2008. http://ceur-ws.org/Vol-432/owled2008eu_submission_12.pdf.

[96] Sirin, Evren, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur y Yarden Katz: *Pellet: A Practical OWL-DL Reasoner*. Web Semant., 5(2):51–53, Junio 2007, ISSN 1570-8268. http://dx.doi.org/10.1016/j.websem.2007.03.004.

[97] Smith, Barry, Michael Ashburner, Cornelius Rosse, Jonathan Bard, William Bug, Werner Ceusters, Louis J Goldberg, Karen Eilbeck, Amelia Ireland, Christopher J Mungall, The OBI Consortium, Neocles Leontis, Philippe Rocca-Serra, Alan Ruttenberg, Susanna Assunta Sansone, Richard H Scheuermann, Nigam Shah, Patricia L Whetzel, y Suzanna Lewis: *The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration*. Nature biotechnology, 2007.

[98] Steigmiller, Andreas, Thorsten Liebig y Birte Glimm: *Konclude: System Description*. Journal of Web Semantics (JWS), 27:78–85, 2014.

[99] Tobies, Stephan: *The complexity of reasoning with cardinality restrictions and nominals in expressive Description Logics.* Journal of Artificial Intelligence Research, 12:199–217, 2000.

[100] Tobies, Stephan: *Complexity Results and Practical Algorithms for Logics in Knowledge Representation.* CoRR, cs.LO/0106031, 2001. http://dblp.uni-trier.de/db/journals/corr/corr0106.html#cs-LO-0106031.

[101] TopQuadrant: *TopQuadrant / Products / TopBraid Composer*, 2011. http://www.topquadrant.com/products/TB_Composer.html, http://www.topquadrant.com/products/TB_Composer.html accedido en agosto del 2011.

[102] Tsarkov, Dmitry y Ian Horrocks: *FaCT++ description logic reasoner: System description.* En *In Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006*, páginas 292–297. Springer, 2006.

[103] Tudorache, Tania, Csongor Nyulas, Natalya Fridman Noy y Mark A. Musen: *WebProtégé: A collaborative ontology editor and knowledge acquisition tool for the Web.* Semantic Web, 4(1):89–99, 2013. http://dx.doi.org/10.3233/SW-2012-0057.

[104] (W3C), World Wide Web Consortium: *OWL Web Ontology Language Overview*, Febrero 2004. https://www.w3.org/TR/owl-features/, accedido en Junio del 2017.

[105] World Wide Web Consortium (W3C): *OWL Web Ontology Language Reference*, 2004. http://www.w3.org/TR/owl-ref/, accedido en Junio de 2013.