Universidad Nacional del Comahue

Facultad de Economía y Administración

Departamento de Ciencias de la Computación

Tesis para la Carrera
Licenciatura en Ciencias de la Computación

Extensión de Protégé para la Integración de Ontologías

Laura Haide Perez

Dra. Alejandra Cechich

Mg. Agustina Buccella

Neuquén Argentina

Agosto 2008

Página para los Evaluadores

Calificación:

Comentarios:

....

.....

.....
Lugar para la Fecha de la Evaluación

Dedicatorias

Esta tesis se la dedico a mi familia que me apoyaron e incentivaron durante todos estos años de estudio. Les agradezco por su su compañia, paciencia y consejos, y por respaldarme en las decisiones que finalmente tomo. Ellos son de alguna manera coautores de este trabajo.

Con todo mi corazón, Laura.

Prefacio

Esta tesis es presentada como parte de los requisitos final para optar al grado académico *Licenciada en Ciencias de la Computación*, de 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 una investigación llevada a cabo en el *Departamento de Ciencias de la Computación* en el período comprendido entre febrero de 2006 y agosto de 2008, bajo la dirección de *Dr. Alejandra Cechich* y la codirección de *Mg. Agustina Buccella.*

Laura Haide Perez
Departamento de Ciencias de la Computación
Universidad Nacional del Comahue
Neuquén, 28 de Agosto de 2008.

Agradecimientos

Utilizo este espacio para agradecer a todas las personas cercanas a mi en el transcurso de mis estudios.

A las directoras de esta tesis, tanto a Alejandra Cechich, como a Agustina Buccella por su dedicación, gran apoyo y enseñanza.
A los docentes de la Universidad Nacional del Comahue por su contribución en mi formación.
Y por último, a mis amigos por su incondicional amistad, cariño y los momentos vividos.

A todos ellos muchas gracias.

Resumen

La integración de datos que son almacenados en sistemas diferentes es un tema activo de investigación desde hace varios años. En los años 90 emergió el término *sistemas federados* o *bases de datos federadas* para caracterizar el conjunto de técnicas que proveen acceso a un conjunto distribuido, heterogéneo y autónomo de bases de datos. A pesar de que existen diferentes arquitecturas de sistemas de información federados, todas ellas comparten la misma tarea semántica, llamada *Correspondencia Semántica*. Esta tarea determina las correspondencias entre elementos de distintas fuentes. La tarea de integración es compleja, ya que se deben tener en cuenta algunas dificultades en cuanto a la heterogeneidad semántica donde se debe decidir acerca de conceptos semánticamente equivalentes o semánticamente relacionados o no relacionados.

El uso de ontologías para la *integración* de fuentes de información heterogéneas es una estrategia viable para solucionar el problema de la heterogeneidad semántica. Las ontologías que fueron introducidas por Gruber [35] como una "especificación explicita de una conceptualización", son ampliamente utilizadas en procesos de integración para describir la semántica de las fuentes de información haciendo su contenido explícito.

Actualmente, con el creciente uso de las ontologías y la distribuida naturaleza de su desarrollo, el problema de conocimiento solapado sobre un dominio ocurre con frecuencia. Las ontologías sobre dominios específicos son modeladas por diferentes autores y para distintos entornos. Para que estas ontologías puedan ser *reutilizadas*, primero deben ser mezcladas o alineadas. La cual es una tarea dificultosa, laboriosa y propensa a la introducción de errores.

En este trabajo de tesis, presentamos la arquitectura propuesta en [12] para un sistema de información federado, en la cual la *capa de federación* es el centro de la misma. Precisamente, nos enfocamos en la estrategia semiautomática de tres niveles (sintáctico, semántico y usuario), que permite la construcción de axiomas de similitud entre dos ontologías. Estos axiomas son expresados como correspondencias entre conceptos de las mismas, para la construcción de un vocabulario global.

En primer lugar, se presenta una propuesta de extensión del método para la detección de definiciones cíclicas en las ontologías. En segundo lugar, introducimos, como un aporte a la eficiencia, una verificación de valores ya calculados para evitar comparar nuevamente un mismo par de clases. En tercer lugar, se realiza el diseño e implementación del método semiautomático de busqueda de similitudes como un plugin, que llamamos OWLSim, para el editor de ontologías Protégé. En la etapa de diseño, aplicamos la metodología diseño guiado por responsabilidades (RDD,

Responsability-Driven Design). Finalmente, como el plugin será implementado como una extensión de Protégé usamos el lenguaje Java para su implementación.

Summary

The integration of data that is stored in different systems is an active research topic since many years. As a result of that research, in the 90th the term *federated system* or *federated database* emerged to characterize techniques for providing an integrated access to a set of distributed, heterogeneous and autonomous databases. Besides the differences existing among the various federated system architectures, all of them share the same semantic task, namely *Semantic Matching*). This task determines the mappings between elements of different sources. The integration task is not easy, some difficulties regarding the semantic heterogeneity should be faced where we should decide about semantically equivalent concepts or semantically related/unrelated concepts.

The use of ontologies for the *integration* of heterogeneous information sources is a possible approach to overcome the problem of semantic heterogeneity. Ontologies that were introduced by [35] as an "explicit specification of a conceptualization", are widely used in integration processes to describe the semantics of the information sources and to make the content explicit.

Currently, with the growing usage of ontologies and distributed nature of ontology development, the problem of overlapping knowledge in a common domain occurs more often. Domain-specific ontologies are modeled by multiple authors in multiple settings. In order for these ontologies to be *reused*, they first need to be merged or aligned to one another. Manual ontology merging using conventional editing tools without support is difficult, labor intensive and error prone.

Therefore, in this work, we introduce an architecture for federated systems in which the *federation layer* is the core of the system [12]. More precisely, we focus in the proposed three-level approach, that allows to build similarities expressed as concept mappings, for construction of a global vocabulary. The concepts of an ontology are compared using three comparison levels: syntactic, semantic and user level.

Firstly, we will make an extension to the method for it to take into account cyclic definition within ontologies. Secondly, in order to improve efficiency, we introduce a new verification in order to avoid calculating twice the similarities for the same pair of concepts. In the third place, we design and implement the semi-automatic method for searching similarities as a plugin, namely OWLSim, for the ontology editor Protégé. We make use of the Responsability-Driven Design (DDR) as a methodology for guiding the design process. Finally, as the plugin will be a Protégé extension we use the Java language for the implementation.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

En este capítulo, se presenta una breve descripción sobre el contexto de la tesis, su objetivo y su estructura.

1.1. Motivación

Es común hoy en día encontrar empresas que posean diferentes bases de datos no integradas entre si, cada una con su propia interfase de usuario, su propio lenguaje de consulta y su propia implementación. Así, el usuario debe adaptar la misma consulta a las características de cada una de estas interfaces, teniendo que aprender la sintaxis en la que se expresan las consultas para cada una de ellas.

El inconveniente que supone consultar una a una cada base de datos relevante hace imprescindible el trabajar con un sistema que integre varias bases de datos bajo una única interfase de usuario. Un sistema de estas características permitirá al usuario obtener información de todas las bases de datos integradas sin más que escribir una consulta. Será el sistema el que redirigirá dicha consulta a todas las bases de datos implicadas y presentará las respuestas obtenidas de cada una de ellas al usuario.

Así, en la actualidad, dentro de las empresas o instituciones existen también gran cantidad de proyectos de integración que tienden a solucionar estos nuevos problemas. Los mismos, en cada caso, se presentan con una forma y restricciones diferentes; por ejemplo, muchos optan por la construcción de un *Depósito de Datos* (Data Warehouse[1]) o de un sistema *middleware* [3] lo cual involucra una cantidad significativa de tareas de integración.

Otra opción es la construcción de un *Sistema Federado* como estructura principal para el proceso de integración. El término *Sistema Federado* o *Bases de Datos Federadas* ha surgido como un medio para caracterizar las técnicas que proveen acceso a los datos integrados, dado un conjunto de fuentes de información (o bases de datos) que están físicamente distribuidas, tienen autonomía y son heterogéneas [37]. Entendemos por *autonomía* que los usuarios y las aplicaciones puedan acceder a los datos a través del sistema federado o por medio de su interfase local.

[1]Un almacén de datos (del inglés data warehouse) es una colección de datos orientada a un determinado ámbito (empresa, organización, etc.), integrado, no volátil y variable en el tiempo, que ayuda a la toma de decisiones en la entidad en la que se utiliza.

La *distribución* está íntimamente ligada al concepto de Internet y a la posibilidad de contar con una gran variedad de información en distintas ubicaciones geográficas. Por último, la *heterogeneidad* se refiere tanto a representaciones diferentes de los modelos de datos como diferente hardware y software.

A pesar de las formas de implementación posibles de un sistema integrado, todas confluyen en una misma tarea semántica llamada, *correspondencias de datos semánticamente heterogéneos* o simplemente *correspondencia semántica* (*Semantic Matching*). Es fácil ver que la etapa de análisis de esta tarea se centra especialmente en identificar entidades que describen los datos, lo que comúnmente es llamado *semántica del mundo real* (*real-world semantics*).

En general existen tres pasos para efectuar la integración:

1. Tomar como entrada múltiples bases de datos (esquema y datos).

2. Para cada una de ellas, obtener información semántica sobre qué describe cada una de las entidades, es decir, lo que llamamos su semántica de mundo real.

3. Producir como salida una base de datos unificada (no necesariamente almacenada físicamente) y un conjunto de correspondencias de las bases de datos individuales con el esquema unificado.

La tarea de crear las correspondencias se lleva a cabo durante el análisis efectuado en las primeras etapas de la integración. Esto determina la correspondencia entre los diferentes elementos de múltiples bases de datos.

Es importante destacar las implicancias del concepto *integración de datos* ya que involucra una serie de decisiones que se deben tomar en forma correcta para lograr un resultado consistente. Cuando hablamos de consistencia nos referimos a que la respuesta que se le brinda a un usuario, luego de efectuar una consulta, debe ser coherente y por lo tanto satisfacer su necesidad. Para esto, el sistema debe recuperar la información de las fuentes de información o bases de datos relacionadas con la consulta que estén disponibles en ese momento. Lamentablemente, la integración no es una tarea fácil debido a que en este punto nos encontramos con problemas que corresponden a la heterogeneidad semántica y que involucran decidir sobre conceptos que son sinónimos, conceptos dentro de una generalización/especialización, etc.

Nótese que la integración y la correspondencia semántica están íntimamente ligadas al concepto de *Web Semántica* [43]. La Web Semántica es una extensión de la web actual en la que la información tiene un significado bien definido, posibilitando que los ordenadores y las personas trabajen en cooperación. Esta provee acceso automático a la información basada en una semántica procesable por la computadora junto con teorías de dominio. La Web Semántica se basa en dos conceptos fundamentales:

- La descripción del significado que tienen los contenidos en la Web.

- La manipulación automática de estos significados.

La descripción del significado requiere conceptos ligados a la Semántica, entendida como significado procesable por máquinas; a los *Metadatos*, como contenedores de información semántica sobre los datos; y a las *Ontologías*, que son un conjunto de términos y relaciones entre ellos que describen un dominio de aplicación concreto.

En el trabajo de investigación en [12], se analizaron las distintas arquitecturas diseñadas para federar bases de datos, en especial aquellas arquitecturas que son dirigidas por ontologías, y se diseñó una arquitectura con capacidad para solucionar problemas de heterogeneidad semántica. La misma tiene el proposito de integrar bases de datos:

- físicamente dispersas, es decir, ubicadas en diferentes lugares geográficos;
- heterogéneas, en cuanto a las máquinas que las albergan, al sistema operativo de dichas máquinas, al SGBD (Sistema Gestor de Base de Datos) que las soporta, a su estructura y al tipo de información que almacenan;
- autónomas, es decir, totalmente independientes en funcionamiento. Esto implica, por un lado, que no es posible cambiar el esquema de ninguna de las bases de datos para adaptarlo al esquema global de la federación. Por otro lado, es posible que estos esquemas cambien para acomodar los requisitos de las aplicaciones asociadas a dichas bases de datos.

Se definieron también métodos y técnicas asociados que en forma semiautomática mejoren los resultados del análisis de correspondencia semántica entre las fuentes de información. Aquí pueden verse problemas relacionados con la heterogeneidad semántica y más específicamente con la heterogeneidad ontológica.

Por otro lado, actualmente las tareas de encontrar correspondencias entre ontologías, alinear ontologías ó la mezcla de ontologías son realizadas, en su mayoría, a mano, ya que existen pocas herramientas para automatizar el proceso completa o parcialmente. Estas tareas realizadas manualmente son tediosas y costosas en lo que a tiempo se refiere, además de ser propensas a la introducción de errores. Por ello, con el motivo de proveer una herramienta más, proponemos la implementación del *método de analisis de correspondencias* entre ontologías, que fue diseñado en [12], como un plugin para el editor de bases de conocimiento Protégé (ver Apéndice A), desarrollado en el Departamento de Informatica Médica (SMI Stanford Medical Informatics) de la Universidad de Stanford. Protégé es una herramienta de código abierto para el desarrollo de bases de conocimiento. Éste posee una *arquitectura basada en componentes* que permite a los usuarios agregar nuevas funcionalidades por medio del desarrollo de plugins. En nuestro caso, el plugin permitiría que los usuarios que han creado ontologías en este entorno puedan analizar dos ontologías en búsqueda de similitudes entre ellas para construir una ontología global. Así mismo, al implementar varias funcionalidades del método propuesto podríamos mostrar su funcionamiento y proveer una implementación para la evaluación del mismo.

1.2. Objetivos

El objetivo de este trabajo de tesis es diseñar e implementar un plugin para el editor de ontologías Protégé-OWL [5, 50]. El plugin implementará el método semi-automático para la búsqueda de similitudes entre dos ontologías [12], incluyendo las funcionalidades del mismo. El modelo de conocimiento subyacente del método, y por lo tanto el plugin, es el provisto por el lenguaje ontológico OWL (Web Ontology Language, lenguaje web ontológico) [4], específicamente por el sublenguaje OWL-Lite.

Los objetivos específicos de esta tesis son principalmente tres:

- Estudio de la definición de ontologías y del contexto de bases de datos federadas. De esto se desprende el estudio del lenguaje ontológico OWL que será utilizado como base para la integración y el análisis de las tareas necesarias para la construcción del sistema.
- Estudio del editor de ontologias Protégé. En este caso, se deberá estudiar la herramienta en sí, así como la forma de implementar los plugins dentro de la misma. Para esto se requiere un análisis de los plugins existentes, además de la documentación de Protégé.
- Implementación del plugin en Java junto con una especificación precisa del sistema.

1.3. Estructura

Los temas de esta tesis están organizados de la siguiente manera:

- En el Capítulo 2 se introduce el concepto de sistema de información federado describiendo sus características. Luego, se describe cómo surgieron las ontologías para ayudar en la tarea de integración o construcción de éstos sistemas junto con los posibles enfoques existentes en la literatura.
- En el Capítulo 3 se describe la arquitectura creada para implementar un sistema federado, y el enfoque de tres niveles [12] para realizar la correspondencia entre ontologías por medio de la construcción del vocabulario común. Luego, se explican una ampliación del método de búsqueda de similitudes completo incluyendo el manejo de ciclos y algunas mejoras para evitar recalcular valores.
- El Capítulo 4 se centra específicamente en nuestro trabajo de tesis. Allí, describimos la herramienta de software que implementa el método semiautomático de búsqueda de similitudes, introducido en el capítulo anterior. La misma será implementada como un plugin para el editor de ontologías Protégé.
- En el Capítulo 5 se presentan varios casos de estudio para describir el funcionamiento del plugin y los resultados que se obtienen del método de búsqueda de similitudes.

- El Capítulo 6 presenta las conclusiones y los trabajos futuros destacando las contribuciones y limitaciones de esta tesis.
- El Apéndice A describe brevemente el editor de ontologías Protégé para el cual hemos desarrollado nuestra herramienta de software. También, se describen los detalles de implementación del plugin para Protégé, así como también las elecciones de otros componentes de software que utiliza nuestro plugin, por ejemplo parser de owl ó interfaces para acceso a recursos lingüísticos.
- En el Apéndice B se describen las características principales del lenguaje ontológico OWL. Nos enfocamos principalmente en el lenguaje OWL-Lite describiendo las primitivas principales de construcción del mismo.

Capítulo 2

Ontologías en la integración de información

En este capítulo describimos los tipos de sistemas de información centrándonos principalmente en las características principales de los sistemas de información federados. Luego, explicamos el significado de las ontologías dentro del área de ciencias de la computación y en particular dentro de la integración de las bases de datos o sistemas de información federados. Veremos las ventajas que nos provee el uso de las mismas junto con diferentes enfoques posibles para su utilización. También introducimos OWL [56] (Web Ontology Language, lenguaje web ontológico) un lenguaje para la representación de ontologías.

2.1. Sistemas de información federados

Un tema de investigación activo desde hace varios años es la integración de los datos [21] que se encuentran almacenados en diferentes sistemas. La primera aproximación fueron los Almacenes de Datos (Data Warehouses) donde los esfuerzos en investigación estaban dirigidos principalmente a conseguir que los datos dispersos entre distintas aplicaciones propietarias e incompatibles se almacenan en un único sistema de gestión de bases de datos central. Estos sistemas poseen los datos replicados en el almacén y las consultas se procesan sobre ese almacén. El problema que deben resolver es el mantenimiento de la consistencia de los datos.

En los '90 surgió la necesidad de combinar los datos almacenados en distintos sistemas de gestión de bases de datos. De hecho, aunque algunos de los primeros trabajos sobre estos sistemas multi-base de datos aparecieron en 1985, no fue hasta 1990 cuando se definió el término de *Bases de Datos Federadas* para caracterizar a las técnicas utilizadas para proveer de un sistema integrado de acceso a un conjunto distribuido y heterogéneo de bases de datos autónomas. En esa época se definieron también otros conceptos de importancia, como la distinción entre sistemas multi-base de datos y sistemas federados, y entre sistemas fuertemente acoplados y débilmente acoplados.

Sin embargo, los requisitos para la integración de datos fueron cambiando durante los últimos años. Por ejemplo, los problemas técnicos de la integración producidos por incompatibilidades de redes de datos han desaparecido, práctica-

mente con la llegada de Internet, y la distribución física se ha convertido en algo manejable gracias a herramientas como CORBA y Java. Por otro lado, el número potencial de fuentes de datos se ha incrementado en gran medida, principalmente debido al uso del *World Wide Web* como un sistema de publicación de datos. Esto último ha producido una gran heterogeneidad, provocando cambios en distintos aspectos, como pueden ser la integración de esquemas o el acceso a la información a través de lenguajes de consulta.

Aparte de todo esto, se han producido cambios en los paradigmas de desarrollo de software, lo que modifica los requisitos de las técnicas de integración de información. Las técnicas actuales se basan en el uso de componentes de integración llamados *mediadores*, que acceden a las fuentes o a otros mediadores bajo demanda. Las fuentes, a su vez, se encuentran encapsuladas por componentes llamados *envoltorios* (wrappers), que son capaces de presentar la información en un formato que el cliente (mediador) necesita. Estas ideas de sistemas basados en conjuntos de pequeños componentes interactivos semiautomáticos, se encuentran en muchos de los proyectos de investigación actuales.

Como efecto colateral se ha producido una redefinición de los conceptos comúnmente usados en estos contextos. El primer cambio importante es denominar a estos sistemas *"Sistemas de información federados"* en lugar de *"Sistemas de Bases de Datos Federadas"*, ya que muchas veces las fuentes no serán solo bases de datos.

Los sistemas de información federados tienen tres características principales que los diferencian de otros sistemas, estas son: *heterogeneidad*, *autonomía* y *distribución* [38]:

- *Heterogeneidad.* Se divide en 4 categorías [27, 33]:

  1. *Heterogeneidad de sistema*, se refiere al hardware y al sistema operativo en el que se ejecuta.
  2. *Heterogeneidad de sintaxis*, se refiere a los diferentes lenguajes y representaciones de los datos.
  3. *Heterogeneidad de estructura*, se refiere a diferentes modelos de datos.
  4. *Heterogeneidad semántica*, se refiere al significado de las palabras, es decir, el significado de los términos usados dentro de un modelo de datos.

- *Autonomía.* Se clasifica en tres tipos [21, 52]:

  1. *Autonomía de Diseño.* El diseño de cada base de datos involucrada en la federación es independiente con respecto al modelo de datos, nombres de conceptos, etc. También se refiere a la autonomía para cambiar su diseño en cualquier momento que se requiera.
  2. *Autonomía de Comunicación.* Cada base de datos puede decidir en forma independiente con cuáles sistemas se comunica, es decir, puede salir y entrar en la federación en cualquier momento.
  3. *Autonomía de Ejecución.* Se refiere a la independencia de las bases de datos con respecto a ejecución y planificación de requerimientos.

Cada base de datos involucrada en la federación posee sus propias interfaces. Por lo tanto, las aplicaciones y los usuarios pueden acceder a los datos ya sea a través del sistema federado o directamente desde su sistema local. Además, es importante que la federación no comprometa el rendimiento de las operaciones locales.

- *Distribución.* Actualmente muchas computadoras están conectadas por algún tipo de red, especialmente por Internet. Así, es natural pensar en combinar aplicaciones y datos que están ubicados físicamente en diferentes nodos pero que pueden comunicarse a través de una red de comunicaciones.

2.1.1. Clasificación de los sistemas de información

Basándose en los principios de distribución y heterogeneidad, los sistemas de información se clasifican en tres clases:

- **Sistemas de Información Simples (monolítico y centralizado, SIS)**: se ejecuta como una aplicación monolítica sobre una computadora ofreciendo una o mas interfaces. Los SIS pueden ser:
  - **Sistemas de bases de datos**: usan un SGBD para almacenar y manipular datos. Están basados en un modelo de datos (relacional, jerárquico u orientado a objetos), poseen un esquema para estructurar los datos y se acceden mediante un lenguaje de consulta (como por ejemplo, SQL u OQL).
  - **Sistemas sin bases de datos**: tales como archivos y colecciones de documentos. Generalmente no están basados en un modelo de datos estándar y no ofrecen un lenguaje de consulta. Estos sistemas se denominan semi-estructurados ya que los datos generalmente no poseen una estructura predefinida.
- **Sistemas de Información Distribuidos (DIS)**: los datos están físicamente distribuidos en diferentes nodos pero conectados mediante alguna clase de red de comunicación.
- **Sistemas de Información Heterogéneos (HIS)**: los sistemas difieren en aspectos como hardware, modelo de datos o semántica.

Si agregamos el principio de autonomía, obtenemos la definición de **Sistemas de Información Federados (FIS)**. La Figura 2.1 muestra gráficamente esta clasificación. Un FIS puede estar construido a partir de cualquier tipo y cantidad de los sistemas de información descriptos anteriormente.

La Figura 2.2 muestra la arquitectura general de un FIS [21]. La *capa de federación* es un componente de software que ofrece un mecanismo uniforme de acceso a los datos por parte de la aplicación y de los usuarios. Esta uniformidad es lograda mediante una estrategia específica de interoperación, es decir, esta capa puede ofrecer un esquema federado, un lenguaje de consulta uniforme o una descripción de las fuentes como conjuntos de metadatos. Generalmente la integración de las fuentes de datos se logra a través de sistemas de envoltorio (wrappers) que resuelven las diferencias técnicas.

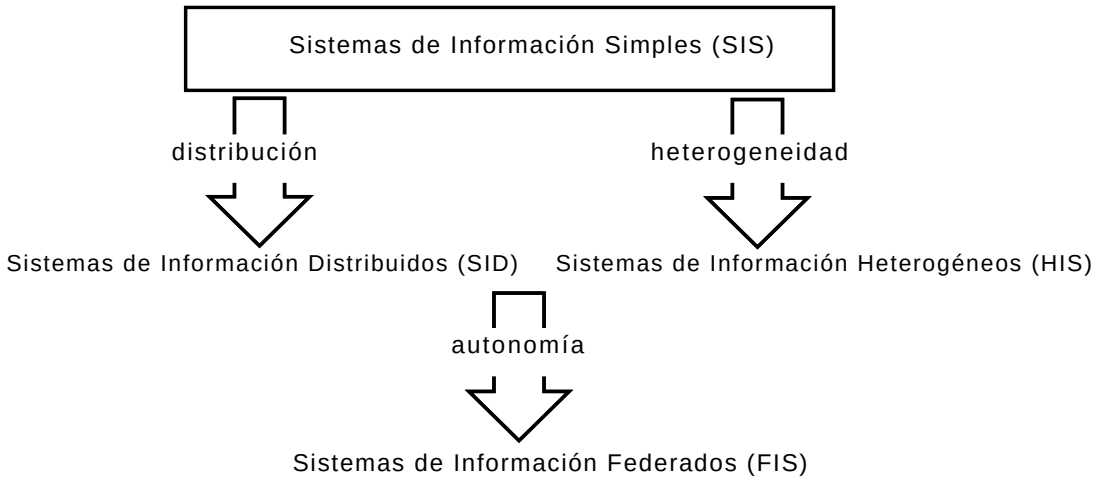

Figura 2.1: Clasificación de los sistemas de información

2.1.2. Tipos de Sistemas de Información Federados

Los sistemas de información federados se dividen en tres tipos según [21]: *sistemas de información débilmente acoplados*, *sistemas de bases de datos federadas* y *sistemas de información basados en mediadores*. Los criterios utilizados para realizar esta clasificación son [8]:

- *Clases de componentes: estructurados, semi-estructurados y no estructurados.* Los FIS difieren en los tipos de componentes que pueden integrar. Las fuentes estructuradas son aquellas que poseen un esquema predefinido. Una fuente de datos semi-estructurada también tiene una estructura pero no esta predefinida en un esquema estricto, es decir, cada dato puede tener su propia definición semántica y la suma de todas ellas puede ser considerada un esquema. Las fuentes de datos no estructuradas no poseen ninguna estructura, por ejemplo, los documentos de texto.

- *Federación débil versus fuerte.* La federación fuerte tiene un esquema global unificado por el cual acceden todos los usuarios. La federación débil no posee ese esquema sino que ofrece sólo un lenguaje de consulta unificado para consultar los datos de las fuentes. En un sistema de federación fuerte, la federación misma puede tratar de compensar capacidades no contenidas en las fuentes; en cambio en un sistema débil depende de la consulta formulada por el usuario.

- *Modelo de datos de un FIS.* La capa de federación de un FIS debe basarse en un modelo de datos específico llamado modelo de datos canónico o modelo de datos común. Los esquemas de la federación fuerte son esquemas en este modelo. Algunos de los modelos de datos más comunes son orientados a objetos o relacionales.

- *Clases de integración semántica.* Hay diferentes tipos de integración semántica a nivel de datos:

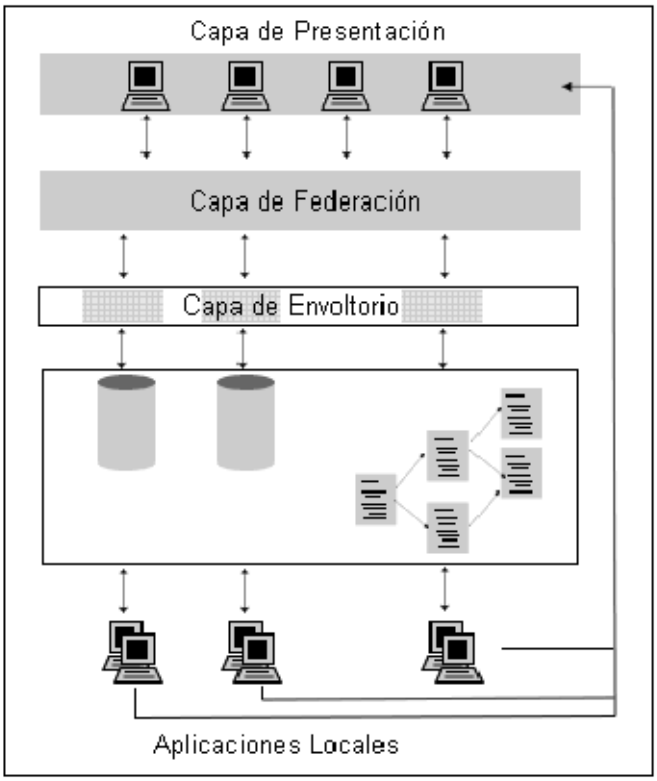

Figura 2.2: Arquitectura general de un sistema de información federado

- *Colección*. Los datos de las fuentes se almacenan tal cual están sin relacionarlos con datos equivalentes en otras fuentes.
- *Fusión*. Simplemente se extraen los datos de las fuentes sin realizar ninguna abstracción. En contraste con el caso anterior, la fusión de los objetos se realiza para identificar entidades equivalentes semánticamente desde las diferentes fuentes. La fusión de datos es una tarea muy difícil ya que involucra decidir qué valores son correctos.
- *Abstracción*. La necesidad de abstracción es generalmente causada por conflictos semánticos.
- *Adición*. Se pueden agregar datos (metadatos), que no se encuentran directamente en las fuentes, sino que describen el contenido o semántica de los datos.

- *Transparencia*. Un sistema federado es transparente cuando el usuario ve al conjunto de fuentes federadas como un simple sistema homogéneo y consistente. Se pueden distinguir entre tres tipos de transparencia:
  - *Transparencia en ubicación*: el usuario no necesita saber la ubicación física de la información.
  - *Transparencia en esquema*: el usuario no necesita conocer los nombres que las entidades o atributos tienen en las fuentes de datos.

  - *Transparencia en lenguaje*: el usuario no necesita manejar diferentes tipos de lenguajes, es decir, existe un único lenguaje de consulta.

- *Paradigma de Consulta.* Los Sistemas de Información permiten tres tipos de consultas:

  - *Consultas estructuradas*: que asumen alguna estructura en la información. Los SGBD son sistemas de este tipo.
  - *Recuperación de la información (IR, Information Retrieval)*: que realiza búsquedas en documentos. El motor de búsqueda de la Web es un ejemplo de un sistema basado en IR.
  - *Metadatos*: como los metadatos describen los objetos de datos, se pueden pensar en consultas que, por ejemplo, busquen un documento que cumpla con algún criterio como tamaño o fecha de creación, los cuales no están almacenados explícitamente con el documento.

- *Bottom-up vs. Top-down* [22]:

  - *Estrategia Top-down*: el sistema se construye en base a la información global necesitada. El esquema global puede ser generado ad-hoc o ser el resultado de un proceso de análisis mas formal, comenzando en casos de uso y terminando con técnicas de integración de vistas. Las aplicaciones que usan ontologías comunes o esquemas estándares son inherentemente top-down. El esquema de componentes es considerado en un segundo paso, es decir, cuando han sido establecidas las correspondencias entre el esquema global y el esquema fuente para permitir la traducción de las consultas.
  - *Estrategia Bottom-up*: se necesita el requerimiento inicial para tener un acceso integrado a un número dado de fuentes de datos. Un escenario típico es por ejemplo, la necesidad de un acceso detallado y uniforme a todas las bases de la compañía para construir aplicaciones globales. Según [22] esta estrategia es apropiada para FIS pequeños o medianos donde la configuración de la infraestructura y las fuentes son relativamente estables.

- *Integración material versus virtual.* Estas dos integraciones se diferencian en el almacenamiento de las fuentes, que puede ser en forma persistente o no persistente dentro de la capa de federación. En el caso de la integración virtual (no persistente), el resultado se almacena sólo en el momento que se efectúa una consulta. En cambio, en el caso de la integración material, las fuentes se almacenan parcial o completamente en la capa de federación (por ejemplo data warehousing). Las ventajas de esta última integración son: un alto rendimiento para las consultas y estricto control de los datos almacenados. Pero sus desventajas son el alto costo para mantener los datos actualizados y la gran cantidad de espacio de almacenamiento necesitado. Además, existen en la literatura algunos enfoques híbridos.

- *Accesos de escritura o sólo lectura.* Hay FIS que permiten inserciones o actualizaciones de los datos a través de la capa de federación y otros que sólo permiten lecturas. El acceso a escritura no tiene importancia en los proyectos de integración debido a una serie de factores:
  - muchas interfaces, como la Web, no permiten escritura,
  - surgen gran cantidad de problemas cuando se quiere escribir a través de vistas integradas [9],
  - la escritura a través de un sistema integrado es compleja ya que si una clase se encuentra representada en más de una fuente no se sabría cuál de estas usar,
  - las transacciones globales requieren protocolos complejos.

La Tabla 2.1 muestra una clasificación de estos tres tipos sistemas de información federados según algunas de las características vistas.

Los *Sistemas de Información débilmente acoplados* no ofrecen un esquema federado, sino solamente un lenguaje de consulta para acceder a los componentes. Esto tiene la ventaja de que los componentes no pierden autonomía para participar en una federación. Por otro lado, no se ofrece transparencia en esquema ni en ubicación, es decir, cuando el usuario efectúa una consulta, se ve obligado a seleccionar el componente involucrado y el campo en particular en el esquema de ese componente.

Al proporcionar un lenguaje de consulta uniforme, el sistema federado soluciona los problemas técnicos y de lenguaje. Los conflictos lógicos tienen que ser resueltos por el usuario de los servicios en la capa de presentación. El usuario va a ser el responsable de toda la integración de los datos, en todos los aspectos de colecciones, fusiones y abstracciones.

La capa de federación es independiente del diseño lógico de los componentes. Como no existe un esquema global, los cambios de los componentes del esquema no afectan al sistema. En la literatura, los sistemas de información débilmente acoplados se denominan sistemas *multi-base de datos.*

Los *sistemas de bases de datos federadas* proporcionan la funcionalidad clásica de los sistemas de bases de datos. Esto incluye acceso de lectura y escritura para el manejo de los datos. El término *bases de datos* indica la relación con sistemas clásicos de bases de datos, es decir, los componentes de los sistemas de bases de datos federadas son fuentes estructuradas, accedidas a través de lenguajes de consulta.

Los sistemas de bases de datos federadas, son sistemas de información fuertemente acoplados construidos con técnicas *Bottom-Up* mediante la aplicación de alguna técnica de integración de esquemas. Como los sistemas fuertemente acoplados, éstos ofrecen a los usuarios transparencia completa en ubicación y en esquema. Sin embargo, suelen tener una arquitectura estática que genera que la evolución se convierta en una tarea realmente compleja. Esto se debe principalmente a que adicionar o sustraer nuevas fuentes o modificar las existentes provoca en muchas ocasiones que el proceso de integración se deba volver a realizar por completo.

	S.I. débilmente acoplados	**Bases de Datos Federadas**	**S.I. basados en mediadores**
Tipos de heterogeneidad solucionados	Técnicos y de lenguaje	Todos excepto heterogeneidad de restricciones. Dificultades en integración de heterogeneidades de esquema	Todos
Transparencia en consulta	Lenguaje	Localización, esquema y parcialmente lenguaje	Localización, esquema y lenguaje
Tipo de componentes	Estructurados	Estructurados	Cualquiera
Métodos de acceso	Lenguaje de consulta	Lenguaje de consulta	Cualquiera
Restricciones de acceso	No	No	Si
Acceso de escritura	Si	Si	No
Acoplamiento	Débil	Fuerte	Fuerte
Tipos de integración semántica	Colecciones	Colecciones y fusiones	Colecciones fusiones y a veces abstracciones
Bottom-Up vs. Top-Down		Bottom-Up	Top-Down
Capacidad de evolución	Alta	Baja	Alta

Cuadro 2.1: Características de los tres tipos de Sistemas de Información Federados

Por supuesto, que esto dependerá de la implementación de cada Base de Datos Federada.

Los *Sistemas de Información basados en Mediadores* son sistemas fuertemente acoplados, por lo tanto, necesitan un esquema federado para proveer acceso integrado a la información de los distintos componentes (heterogeneidad semántica). Una diferencia obvia con los Sistemas de Bases de datos Federadas, es el acceso de sólo lectura a los datos. Además, de acuerdo a las necesidades de información los sistemas basados en mediadores suelen construirse *Top-Down*. Por lo tanto, añadir o eliminar componentes debería ser sencillo.

2.2. Integración de sistemas de información federados

La *integración de los datos* es uno de los problemas principales que afronta el modelado de un sistema federado. El concepto de integración involucra una serie de decisiones que se deben tomar en forma correcta para lograr un resultado consistente. Cuando hablamos de consistencia nos referimos a que la respuesta que se le brinda a un usuario, luego de efectuar una consulta, debe ser coherente y por lo tanto satisfacer su necesidad. Para esto, el sistema debe recuperar la información de las fuentes de información relacionadas con la consulta que estén disponibles en ese momento. Lamentablemente, la integración no es una tarea fácil debido a que en este punto nos encontramos con problemas que corresponden a la

heterogeneidad semántica que explicaremos en secciones siguientes.

Existen en la literatura gran cantidad de métodos y proyectos que surgen para solucionar los problemas de la integración. En general, estos proyectos se dividen en dos ramas dependiendo del uso o no de ontologías.

A continuación describimos el significado de las ontologías junto con las ventajas que proveen dentro del campo de los sistemas de información federados y en particular en el proceso de integración.

2.2.1. Ontologías

A través de los años el término *ontología* ha sido usado en muchas formas distintas y en diferentes dominios [24]. Por ejemplo, tiene una larga historia en filosofía, en el cual se refiere a la temática existencial.

Dentro del mundo de las Ciencias de la Computación las ontologías fueron introducidas por Gruber [35] como una "especificación formal y explícita de una conceptualización compartida". El término *conceptualización* se refiere a un modelo abstracto de cómo el ser humano piensa comúnmente cosas del mundo real, como por ejemplo, *una mesa* (objetos, conceptos, y otras entidades cuya existencia es asumida en algún área de interés y las relaciones que se mantienen entre ellas). El término *especificación explícita* se refiere a que se ha dado un nombre y una definición explícita a los conceptos y relaciones creados en el modelo abstracto. *Formal* se refiere al hecho de que la ontología debe ser entendida por una computadora (machine readable) [30, 60]. Y *compartida* refleja la noción de que una ontología captura conocimiento consensuado, es decir, es aceptada por un grupo. Por lo tanto, una ontología:

- nombra y describe las entidades que existen en un dominio por medio de predicados que representan relaciones entre esas entidades,
- provee un vocabulario para representar y comunicar conocimiento sobre el dominio,
- provee de un conjunto de relaciones que contienen los términos del vocabulario a un nivel conceptual.

Dependiendo del nivel de generalidad, se pueden identificar diferentes tipos de ontologías. Cada una de ellas cumple diferentes roles en el proceso de construcción de sistemas basados en conocimiento [30]. Algunos de ellos son:

- *Ontologías de dominio*: capturan el conocimiento válido para un tipo particular de dominio, por ejemplo, dominio médico, electrónico, digital, etc.
- *Ontologías de meta datos*: proveen un vocabulario para describir el contenido de fuentes de información en línea [62].
- *Ontologías genéricas*: capturan el conocimiento general sobre el mundo proveyendo nociones básicas y conceptos para cosas como tiempo, espacio, evento, estado, etc. Son válidas para varios dominios. Por ejemplo una ontología sobre relaciones *parte-de* es aplicable a cualquier dominio técnico.

- *Ontologías de representación*: no se refieren a ningún dominio. Solo proveen entidades de representación sin especificar lo que las mismas representan. Un ejemplo de este tipo de ontología es la *Frame-Ontology* [35] que define conceptos como marcos (frames), estantes (slots), etc. para permitir la expresión de conocimiento.

- *Ontologías de tareas o métodos*: proveen el razonamiento en un dominio de conocimiento. Las ontologías de tareas proveen términos específicos para una tarea particular, por ejemplo las *hipótesis* pertenecen al diagnóstico de las ontologías de tareas. Las ontologías de métodos proveen términos específicos para métodos de solución de problemas, por ejemplo el *estado correcto* pertenece a una ontología de método revise-y-proponga.

Así, las ontologías nos proveen las herramientas para describir la información basada en su contexto semántico permitiéndonos definir en forma sencilla la semántica de la información fuente. En el trabajo en [12] se usaron las ontologías para integrar fuentes de información las cuales pueden poseer distintas clases de diferencias. Estas clases de diferencias se denominan *heterogeneidad* [27, 33] (describimos más adelante los distintos tipos). En particular, la heterogeneidad semántica involucra una serie de conceptos en cuanto a la analogía en la forma y significado de las palabras o términos asociados a los conceptos de una ontología. Estos son:

- **Sinónimos**: son términos iguales o semejantes en significado pero su escritura y por lo tanto su pronunciación son diferentes, por ejemplo, casa y hogar.

- **Homónimos**: son términos que poseen la misma pronunciación pero su significado es diferente y algunas veces también su escritura, por ejemplo, arroyo (riachuelo) y arrollo (atropellar). Existen dos clases de homónimos: homófonos y homógrafos. Los homónimos homófonos son aquellos términos de igual pronunciación pero significado y escritura distinta como por ejemplo, ola (del mar) y hola (saludo). Por otro lado, los homónimos homógrafos son aquellos términos de igual escritura pero que significan cosas distintas, por ejemplo: capital (de un país) y capital (de dinero).

- **Antónimos**: son términos que se escriben diferente y su significado es exactamente lo opuesto, por ejemplo, día y noche.

- **Hiperónimos/Hipónimos**: son términos que poseen un nivel diferente de abstracción. Por ejemplo, animal es hiperónimo de gato y a su vez gato es un hipónimo de animal.

En la integración de ontologías se deben solucionar problemas centrados en términos que son sinónimos y homónimos homógrafos. Los homónimos homófonos, palabras con la misma pronunciación pero con distinta ortografía y significado, y antónimos no generan problemas a la hora de comparar dos términos dentro de la integración, dado que éstos, tienen palabras (o cadenas de caracteres) que difieren

entre sí. Además, la búsqueda de estas palabras en un Tesauro [1] confirmaría que se trata de conceptos de diferente significado. A diferencia, para determinar que dos palabras, probablemente con muy diferente ortografía, son sinónimos basta con efectuar una búsqueda en un Tesauro. Mas difícil todavía, es el caso de los homónimos homógrafos, ya que además de las palabras tener la misma ortografía, la búsqueda en Tesauro no alcanza para distinguir sus diferencias en el significado. Se necesitan otros mecanismos, para desambigüar los sentidos de las palabras.

Una clasificación posible de la heterogeneidad semántica es la siguiente:

- Conceptos equivalentes semánticamente:
  - Los modelos utilizan términos distintos para referirse al mismo concepto. Estos generalmente son sinónimos.
  - Los sistemas modelan las propiedades de manera diferente, por ejemplo, para el mismo producto, un catálogo incluye el color y el otro no.
  - Tipos de propiedades distintos, por ejemplo, el concepto *longitud* puede estar en metros o en millas.
- Conceptos no relacionados semánticamente:
  - Los sistemas pueden elegir el mismo término para especificar conceptos completamente diferentes.
- Conceptos relacionados semánticamente:
  - Generalización/Especialización: los sistemas representan los términos con diferentes niveles de abstracción; por ejemplo, un sistema tiene los conceptos *fideos*, *asado*, *arroz*, etc. y el otro sólo el concepto *comida* (relaciones de hiperonimia/hiponimia).
  - Diferente clasificación: los sistemas clasifican en forma diferente términos de un mismo nivel de abstracción; por ejemplo, un sistema clasifica el término persona como *femenino* y *masculino* y otro como *empleado* y *desempleado*.

2.2.2. Lenguajes para representación de ontologías

La palabra ontología ha sido usada para nombrar recursos con distintos niveles estructurales. Éstos van desde simples jerarquías (e.g. jerarquía en Yahoo), hasta esquemas de metadatos (e.g. Dublin Core [2]), o teorías lógicas. La Web Semántica, por ejemplo, necesita de ontologías con un significativo grado de estructuración. Ésta necesita especificar descripciones para la siguiente clase de conceptos:

- Clases, entidades generales, en los diversos dominios de interés.

[1] La palabra tesauro, derivado del neo latín que significa tesoro, se refiere a listado de palabras o términos empleados para representar conceptos.

[2] Es un conjunto de metadatos estandar para descripción de recursos de información. Provee conjunto simple y estandarizado de convenciones para describir recursos online de manera que sea mas fácil encontralos.

- Relaciones que pueden existir entre las entidades.
- Propiedades, o atributos, que dichas entidades puedan tener.

Los lenguajes ontológicos son lenguajes formales usados para construir ontologías, permitiendo la codificación de conocimiento acerca de dominios específicos. Éstos lenguajes son usualmente declarativos, son generalizaciones de lenguajes basados en marcos (frame-based), y se basan en FOL (First Order Logic, lógica de primer orden) ó DL (Description Logic, lógica descriptiva). Existe una amplia variedad de lenguajes para generar una "especificación explicita", [35] definición introducida en la sección anterior. Y la mayoría de los lenguajes basan las representaciones en:

- Objetos/Instancias/Individuos

  Elementos del dominio del discurso

  Equivalente a constantes en FOL

- Tipos/Clases/Conceptos

  Conjunto de objetos que comparten ciertas características

  Equivalente a predicados unarios en FOL

- Relaciones/Propiedades/Roles

  Conjunto de pares (duplas) de objetos

  Equivalente a predicados unarios en FOL

Cuando las ontologías están expresadas en lenguajes basados en lógica se pueden hacer distinciones entre clases, relaciones y propiedades que sean detalladas, precisas, consistentes y significativas. Por ello, algunas herramientas de manipulación de ontologías pueden hacer razonamiento automático con estas ontologías y así proveer servicios avanzados a aplicaciones inteligentes como: búsqueda y recuperación conceptual/semántica, agentes de software, soporte de decisiones, entendimiento de lenguaje natural, entre otras.

Lenguajes Web de especificación de ontologías

Como se mencionó en el Capítulo 1, el lenguaje que será utilizado en este trabajo para la representación de ontologías es el lenguaje ontológico OWL, ya que es un estándar y por lo tanto su uso dentro de la comunidad de la web es cada vez más amplio. Antes de describir las características del lenguaje ontológico OWL, describiremos en pocas palabras los dos estándares existentes mas conocidos en los cuales se basaron los lenguajes web de especificación de ontologías, llamados XML [10] (eXtensible Markup Language) y RDF [42] (Resource Description Framework). También, explicamos los lenguajes web para representar ontologías en los cuales se basó el lenguaje OWL, llamados *OIL* [40] (Ontology Interchange Language) y *DAML+OIL* [26] (Darpa Agent Markup Language).

- *XML*: este metalenguaje deriva de SGML (Standard General Markup Language) y fue desarrollado por el grupo W3C[3] . Algunas ventajas de este lenguaje son la facilidad de depuración y su sintaxis bien definida y comprensible por cualquier persona. Existen en el mercado muchas herramientas para depurar y trabajar con XML. Un documento XML puede usar una declaración del tipo de documento ya sea conteniendo su definición o apuntando a ella. Esta declaración define la gramática para los documentos XML y es llamada DTD (Document Type Definition). Cuando un documento XML se usa como base de un lenguaje de especificación de ontologías posee las siguientes ventajas:

  - permite la definición de una especificación sintáctica común por medio de los DTDs,
  - es fácil de ser leído por cualquier persona,
  - puede ser usado para representar conocimiento distribuido a través de varias páginas web como si estuviera embebido en ellas.

  Pero presenta las siguientes desventajas:

  - es difícil encontrar componentes de una ontología dentro de un documento debido a que permite la especificación de información no estructurada,
  - no posee herramientas para realizar inferencias.

  XML es un lenguaje que no ofrece características especiales para la especificación de ontologías, sólo ofrece una simple pero poderosa manera de especificar la sintaxis. Por lo tanto los lenguajes usan como base a XML para utilizar esa sintaxis y explotar las facilidades de comunicación de la Web.

- *RDF*: es un lenguaje desarrollado también por W3C para describir recursos de web creando metadatos. Posee una fuerte relación con XML ya que estos dos lenguajes han sido definidos como complementarios: una de las metas principales de RDF es especificar en manera estándar la semántica de los datos basados en XML. El modelo de datos consiste en tres tipos de objetos: recursos (entidades que pueden ser referenciadas por una dirección en la web), propiedades (o predicados, que definen características o atributos usados para describir un recurso) y sentencias (que asignan un valor a una propiedad en un recurso específico). Pero para proveer mecanismos de definición de relaciones entre atributos y recursos existe el lenguaje de especificación RDFS (RDF Schema) [23]. Este es más sencillo pero menos expresivo que cualquier otro lenguaje de cálculo de predicados. En conclusión, una ontología definida en RDFS puede carecer de funciones y axiomas pero será posible definir conceptos, relaciones e instancias.

  Aunque los lenguajes XML DTDs y XML Schema [11] son suficientes para el intercambio de datos, entre partes que hayan acordado definiciones de

[3] Web Ontology Working Group `www.w3.org`

antemano, su falta de semántica impide que las máquinas puedan realizar automáticamente esta tarea a medida que se agregan nuevos vocabularios. El mismo término puede ser usado con diferente significado, muchas veces muy sutil, en diferentes contextos, y diferentes términos pueden ser usados para ítems con el mismo significado. XML Schemas no es considerado como un lenguaje ontológico. RDF y RDFS apuntan a este problema permitiendo que una semántica simple sea asociada con los identificadores. Con RDFS [23], se pueden definir clases que pueden tener múltiples sub-clases y super-clases, y se pueden definir propiedades, que pueden tener sub-propiedades, dominios y rangos. En este sentido RDF Schema se puede ver como un lenguaje simple para ontologías. Sin embargo, para lograr interoperación entre numerosos, esquemas autonomamente desarrollados y manipulados, es necesaria una semántica más rica. Por ejemplo, en RDFS no se puede especificar que las clases *Persona* y *Auto* son disjuntas; o que un cuarteto musical tiene exactamente cuatro músicos como miembro.

- *OIL* [40] (Ontology Interchange Language): es un lenguaje diseñado para proveer muchas de las primitivas usadas en lenguajes basados en marcos y Lógica Descriptiva (DL). OIL hereda de la Lógica Descriptiva su semántica formal y el soporte de razonamiento eficiente desarrollado para estos lenguajes (FACT [39], RACER [36], etc.). De los sistemas basados en marcos, OIL incorpora las primitivas de modelado esenciales en su lenguaje. Está basado en OKBC, XOL y RDF con una semántica clara y bien definida. Una ontología en OIL es una estructura que posee componentes organizados en tres capas: el nivel de objetos (que maneja instancias), el primer meta nivel o definición de la ontología y el segundo meta nivel o contenedor de la ontología (que contiene información sobre las características de la ontología). Pueden definirse conceptos, relaciones, funciones y axiomas. Se basa en la noción de asociar a un concepto la definición de sus superclases y atributos. También pueden definirse relaciones no sólo como atributos de una clase sino como una entidad independiente con un dominio y rango. Al igual que las clases, las relaciones también pueden pertenecer a una jerarquía.

- *DAML+OIL* [26]: fue construido teniendo en cuenta a OIL. Provee primitivas de modelado encontradas comúnmente en los lenguajes basados en marcos y posee una semántica bien definida. Esta basado en lógica descriptiva SHIQ [41] (el cual adiciona propiedades inversas y restricciones de cardinalidad generalizadas) y en RDFS. La presencia de una semántica bien definida en términos de SHIQ permite el uso de los razonadores de Lógica Descriptiva tal como FACT y RACER, para soportar tareas de clasificación y detección de inconsistencias.

El Lenguaje Web Ontológico - OWL

El lenguaje OWL fue desarrollado como una extensión de RDF y deriva del lenguaje para ontologías DAML+OIL. Este lenguaje fue propuesto como estándar y es ampliamente aceptado como lenguaje ontológico para la web semántica. Posee una sintaxis y semántica bien definida, soporte para razonamiento automático

y flexibilidad para la elección de distintos grados de expresividad. Existen tres sublenguajes con distinto grado de expresividad: OWL-Lite, OWL-DL, y OWL-Full. OWL-Lite es el menos expresivo, mientras que OWL-Full es el de mayor expresividad.

OWL-Lite OWL-Lite es el sub-lenguaje de sintaxis más simple. El propósito del mismo es que sea usado en situaciones en las cuales sólo se necesita una jerarquía de clases simple y restricciones simples. Por ejemplo, éste provee restricciones de cardinalidad, pero solo permite valores de cardinalidad de 0 o 1. De esta manera, debería ser más simple proveer herramientas de soporte para OWL-Lite que para los otros lenguajes de la familia.

OWL-DL OWL-DL es mucho mas expresivo y se basa en Lógicas Descriptivas (Description Logics, DL). Las Lógicas Descriptivas son un fragmento decidible de la lógica de primer orden.y por lo tanto permiten el razonamiento automático. Por lo tanto es posible calcular automáticamente la jerarquía de clases y chequear inconsistencias en una ontología.

OWL-Full OWL-Full es el sub-lenguaje mas expresivo. La intensión es que sea usado en situaciones donde un alto grado de expresividad sea mas importante que ser capaz de garantizar decidabilidad [4] y completitud computacional del lenguaje [5]. Por lo tanto no es posible realizar razonamiento automático sobre ontologías en OWL-Full.

En el Apéndice B se describen con más detalle las características del lenguaje OWL mencionadas en esta sección, junto con la sintaxis y la explicación de las construcciones mas importantes.

2.2.3. Enfoques para el uso de las ontologías en la integración de sistemas de información federados

En [61] se describen tres enfoques diferentes para aplicar el uso de ontologías: a) *enfoque simple*, b) *enfoque múltiple* y c) *enfoque híbrido*. La Figura 2.3 muestra a cada uno en forma gráfica. El *enfoque simple* usa una ontología global que provee un vocabulario compartido para la especificación de la semántica. Todas las fuentes de información están relacionadas mediante una única ontología. Este enfoque tiene como principal ventaja el tiempo de desarrollo ya que puede ser realizado en un período corto. La principal desventaja se encuentra en el difícil mantenimiento de la ontología en cuanto a cambios en las fuentes o a la adición de alguna nueva.

Por otro lado, en el *enfoque múltiple* cada fuente de información está descripta mediante su propia ontología y cada ontología puede ser desarrollada independientemente de las demás. Al no crearse una ontología compartida o global, no se deben realizar tareas de modificación de la misma cuando se agregan o modifican nuevas fuentes de información. Pero la falta de un vocabulario común es la

[4] todas las computaciones terminarán en un tiempo finito

[5] se garantiza el computo de todas las conclusiones

principal desventaja ya que hace difícil la tarea de comparar las ontologías entre sí.

Finalmente, para solucionar esta desventaja surge el *enfoque híbrido* ya que además de las múltiples ontologías se define un vocabulario compartido. Este contiene los términos básicos del dominio que están combinados en las ontologías locales para describir una ontología mas compleja. Muchas veces el vocabulario compartido también es una ontología. La ventaja principal de este enfoque es que las nuevas fuentes de información pueden ser fácilmente adicionadas a la integración sin la necesidad de efectuar mayores modificaciones. Una desventaja importante de este enfoque es el almacenamiento del vocabulario compartido ya que muchas veces puede traer problemas de rendimiento y espacio en algunos casos.

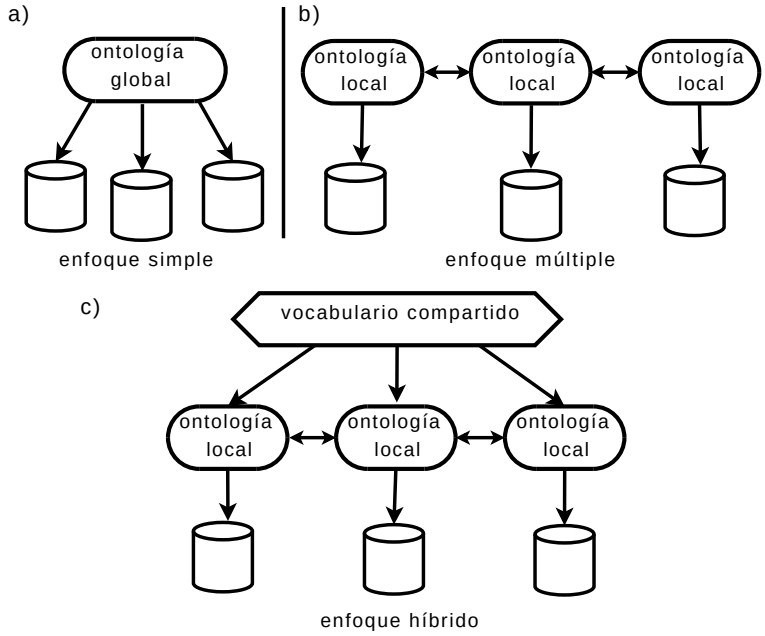

Figura 2.3: Tres formas de usar ontologías

En el enfoque híbrido, e incluso en el enfoque múltiple, se genera otro tipo de heterogeneidad llamada *heterogeneidad ontológica* [59]. Para describir este tipo de heterogeneidad, Visser [59] define a una ontología como una 5-tupla

$$O =< CD, RD, FD, ID, AD >$$

donde CD es un conjunto de definiciones de clases, RD es un conjunto de definiciones de relaciones, FD es un conjunto de definiciones de funciones, ID es un conjunto de definiciones de instancias y AD es un conjunto de definiciones de axiomas. Por ejemplo, las clases *Empleado* y *Departamento* pertenecen al conjunto CD, *trabajaEn* pertenece a FD y *Juan Perez*, *Pedro Lopez* y *Compras* pertenecen a ID.

Teniendo en cuenta estos elementos de las ontologías, Visser clasificó a las incompatibilidades de *heterogeneidad ontológica* en dos tipos básicos: *incompatibilidades de conceptualización* e *incompatibilidades de explicación*. Las Figuras 2.4 y 2.5 muestran las clasificaciones del primer y segundo tipo respectivamente.

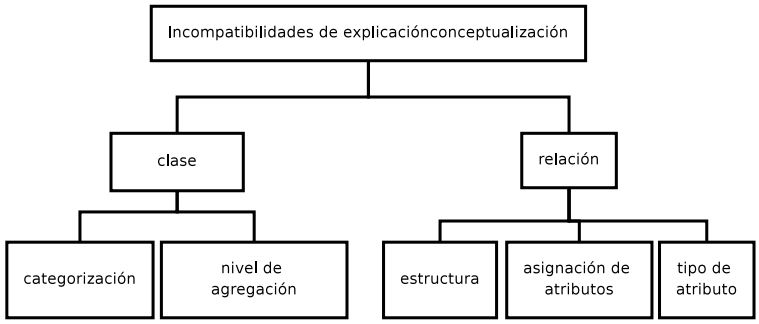

Figura 2.4: Clasificación de las incompatibilidades de conceptualización

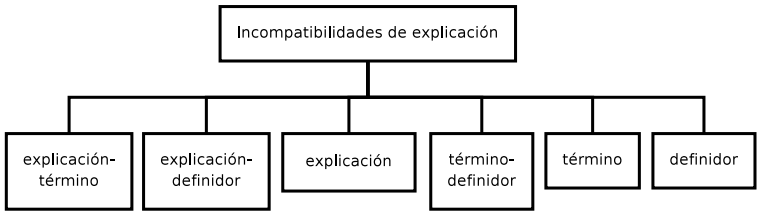

Figura 2.5: Clasificación de las incompatibilidades de explicación

Las *incompatibilidades de conceptualización* son incompatibilidades que se refieren a dos (o mas) términos/conceptos del dominio. Éstos difieren en la definición de los conceptos ontológicos o en la forma en que se relacionan los términos.

Las incompatibilidades de conceptualización se dividen en dos tipos: *incompatibilidad de clase* e *incompatibilidad de relación*.

- *Incompatibilidad de clase*. Relaciona algunas clases definidas en la conceptualización. En particular, este tipo de incompatibilidad se refiere a las clases y sus subclases y se divide en dos tipos: *incompatibilidad de categorización* e *incompatibilidad de nivel de agregación*.

  - *Incompatibilidad de categorización*: ocurre cuando dos conceptualizaciones definen la misma clase, pero cada una la divide en diferentes subclases. La Figura 2.6 muestra un ejemplo de dos conceptualizaciones modelando conocimiento acerca de animales. Cada una divide la clase *Animal* en diferentes subclases.

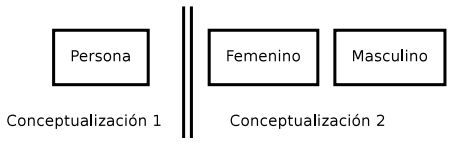

Figura 2.6: Un ejemplo de incompatibilidad de categorización

  - *Incompatibilidad de nivel de agregación*: ocurre cuando dos conceptualizaciones definen clases con diferentes niveles de abstracción. La Figura 2.7 muestra un ejemplo de dos conceptualizaciones modelando conocimiento acerca de personas. Una de ellas define la clase *persona* y la

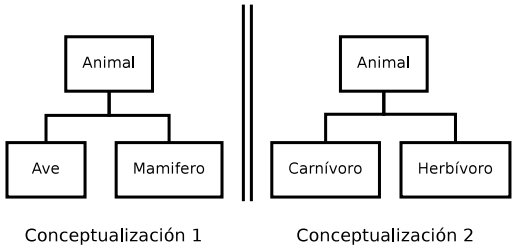

Figura 2.7: Un ejemplo de incompatibilidad de nivel de agregación

otra sólo las clases *femenino* y *masculino* sin poseer como superclase a persona.

- *Incompatibilidad de relación.* Conecta las relaciones definidas en la conceptualización. Este tipo de incompatibilidad se refiere, por ejemplo, a las relaciones de herencia entre dos clases, a la asignación de atributos a las clases, etc. Se divide en tres tipos: *incompatibilidad de estructura*, *incompatibilidad de asignación de atributos* e *incompatibilidad de tipo de atributo.*

  - *Incompatibilidad de estructura*: ocurre cuando dos conceptualizaciones definen el mismo conjunto de clases pero difieren en la forma de estructuración de esas clases por medio de relaciones. La Figura 2.8 muestra un ejemplo de dos conceptualizaciones modelando las mismas clases, pero cada una definiendo una relación diferente. Las dos relaciones *esta-compuesta* y *es-hecha* tienen un significado similar pero no son completamente iguales ya que no podemos decir que una casa esta hecha de ladrillo, pero sí que la casa esta compuesta por ladrillos.

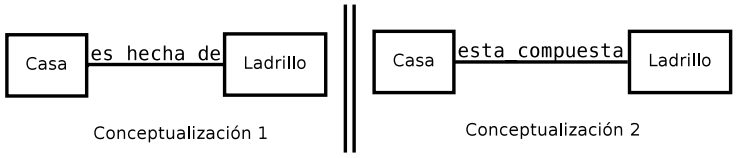

Figura 2.8: Un ejemplo de incompatibilidad de estructura

  - *Incompatibilidad de asignación de atributos*: ocurre cuando dos conceptualizaciones difieren en la forma en que asignan un atributo (clase) a otras clases. La Figura 2.9 muestra un ejemplo de dos conceptualizaciones modelando una asignación de atributos diferente.
  - *Incompatibilidad de tipo de atributo*: ocurre cuando dos conceptualizaciones definen la misma clase (atributo) pero difieren en las instancias que generan. Por ejemplo, la clase (atributo) *longitud* contenida en dos conceptualizaciones posee una instancia de tipo *metros* en una de ellas y en la otra de tipo *millas.*

Por otro lado, las *Incompatibilidades de explicación* no están definidas sobre la conceptualización del domino sino sobre la forma en que es especificada la con-

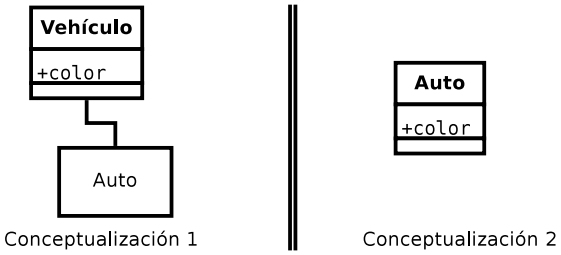

Figura 2.9: Un ejemplo de incompatibilidad de asignación de atributos

ceptualización. Cada definición en *CD*, *RD* e *ID* es una 3-tupla

$$Def =< T, D, C >$$

donde T es un término, D es un definidor y C es el concepto ontológico que es explicado desde la conceptualización, por ejemplo en la sección de DOCUMENTACIÓN definida en lenguajes como Ontolingua [34]. T y D están definidas en algún lenguaje formal por medio de una fórmula atómica y una compuesta, respectivamente. Un ejemplo de una definición es: *auto(X) negro(X) grande(X)* con la descripción del concepto "un medio de transporte que sirve para trasladar personas de un lugar a otro". En este caso T es el término *auto(X)*, D es el definidor que posee las propiedades del término *negro(X)* y *grande(X)* y C es la descripción, en este caso en lenguaje natural, del término que es explicado.

Seis tipos de incompatibilidades surgen de estos tres componentes de una definición ya que una incompatibilidad de explicación ocurre cuando dos ontologías tienen diferentes definiciones pero algunos de sus términos, sus definidores[6] o sus explicaciones son idénticas:

- *Incompatibilidad CT*: ocurre cuando dos ontologías usan los mismos definidores D pero la explicación C y el término T son distintos. Por ejemplo, si una ontología posee la definición *auto(X) negro(X) grande(X)* para definir el concepto *auto*, y otra la definición *tiburón(X) negro(X) grande(X)* para definir el concepto *tiburón*.

- *Incompatibilidad CD*: ocurre cuando dos ontologías usan el mismo término T pero difieren en la explicación C y los definidores D. Por ejemplo, una ontología posee la definición *araña(X) peluda(X) venenosa(X)* para definir el insecto araña y otra ontología usa el mismo término pero los definidores son diferentes: *araña(X) con_velas(X) grande(X)* definiendo el concepto araña que se utiliza para describir a las lámparas colgantes antiguas. Una incompatibilidad *CD* implica que los términos T (*araña(X)*) son homónimos.

- *Incompatibilidad C*: ocurre cuando dos ontologías usan el mismo término T y definidores D pero difieren en la explicación C. Por ejemplo, ambas ontologías podrían usar la expresión *araña(X) grande(X) negra(X)* pero cada una toma al concepto araña con un significado diferente. Una de ella entiende

[6] palabra utilizada para traducir *definers*

por araña al insecto y otra a la lámpara. Una incompatibilidad C implica que los términos T (*araña(X)*) son homónimos.

- *Incompatibilidad TD*: ocurre cuando dos ontologías definen la misma explicación C pero difieren en la manera en que lo definen, es decir, definen diferentes términos T y definidores D. Por ejemplo, dos ontologías para definir el concepto barco utilizan dos definiciones totalmente diferentes. En una caso se define como *barco(X) grande(X) flota(X)* y en el otro caso como *navío(X) amplio(X) navega(X)*. Una incompatibilidad TD implica que los términos T (*barco(X)* y *navío(X)*) son sinónimos y posiblemente las propiedades D también lo sean.

- *Incompatibilidad T*: ocurre cuando dos ontologías definen la misma explicación C usando los mismos definidores D pero se refieren al término T en forma distinta. Por ejemplo, una ontología para definir el concepto barco utilizan la definición *barco(X) grande(X) flota(X)* y otra ontología lo define como *navío(X) grande(X) flota(X)*. Una incompatibilidad T implica que los términos T (*barco(X)* y *navío(X)*) son sinónimos.

- *Incompatibilidad D*: ocurre cuando dos ontologías definen la misma explicación C usando el mismo término T pero usa diferentes definidores D. Por ejemplo, una ontología para definir el concepto barco utiliza la definición *barco(X) grande(X) flota(X)* y otra ontología utilizando el mismo término lo define como *barco(X) amplio(X) navega(X)*.

2.3. Resumen

En este capítulo hemos descripto las características generales de los sistemas de información federados y la influencia de las ontologías como herramienta para facilitar el proceso de integración.

Como vimos, una ontología describe conceptos y relaciones que son importantes en un dominio particular, proveyendo un vocabulario para ese dominio, además de una especificación computarizada del significado de los términos usados en el vocabulario. En los últimos años, las ontologías han sido adoptadas en muchos aplicaciones y comunidades científicas como un modo de compartir, reusar y procesar conocimiento del dominio. Las ontologías son ahora centrales para muchas aplicaciones tales como portales de conocimiento científico, sistemas de manejo e integración de información, comercio electrónico, y servicios de la web semántica.

El siguiente capítulo, describe una arquitectura propuesta para un sistema federado teniendo en cuenta ventajas y desventajas analizadas en este capítulo. Se centra principalmente en la capa de federación y en el método para la búsqueda de similitudes.

Capítulo 3

La Capa de Federación

En este capítulo describimos la propuesta en [12], para solucionar los problemas que surgen de un sistema federado. Específicamente mostramos cómo el uso de las ontologías puede ayudar a solucionar problemas con respecto a la heterogeneidad semántica existente entre los diferentes sistemas que participan en la integración. Además de describir la arquitectura basada en un enfoque híbrido, describimos el método de tres niveles para búsqueda de similitudes entre conceptos, el cual es parte fundamental en la creación del vocabulario compartido. Finalmente, explicamos la extensión que realizamos en este trabajo a dicho método: detección de ciclos.

3.1. Arquitectura de un Sistema Federado

El proceso de integración de datos es uno de los problemas principales que posee todo sistema federado. El usuario de un sistema de este tipo debe utilizarlo de la misma forma que utiliza un sistema centralizado, es decir, no debe notar que existen varias fuentes subyacentes de donde se obtiene la información. A su vez, la información obtenida debe ser consistente y completa. Consistente se refiere a que debe corresponderse con todas las fuentes y completa se refiere a que debe obtenerse toda la información almacenada en las diferentes fuentes. Estos dos conceptos están vistos desde el punto de vista del usuario. Desde el punto de vista del diseñador o desarrollador de un sistema federado la consistencia se refiere a los mecanismos que se deben implementar para que los datos se muestren de una única manera. Por ejemplo, una fuente puede tener información del género de las personas como F y M y otra como *Femenino* y *Masculino*. El diseñador debe decidir cuál de las dos formas se utilizará para mostrar esa información al usuario. La completitud se refiere a que la información que está distribuida en las fuentes se corresponda de forma correcta con la información que provee el sistema federado. Por ejemplo, la información que se representa en un vocabulario compartido contenga toda la información distribuida en las diferentes fuentes. Así, cuando el usuario efectúa una consulta, el sistema sea capaz de devolver toda la información relacionada con la misma.

Con esto en mente, podemos ver que la integración es un proceso complejo que implica el conocimiento semántico de cada una de las fuentes de información involucradas, de manera de realizar correspondencias entre ellas en forma correcta.

Para brindar el conocimiento semántico necesario, en [12] se propone el uso de ontologías como herramientas para lograr esa integración ayudando a comprender y comparar datos de diversas fuentes. Las ontologías que ya fueron introducidas en la Sección 2.2.1, proveen el nombre y las descripciones de las entidades específicas del dominio a través del uso de predicados unarios, o conceptos y predicados binarios, o roles. Una ontología brinda un vocabulario para representar y comunicar conocimiento del dominio y un conjunto de relaciones que contienen, a nivel conceptual, los términos del vocabulario.

En [12] se define una ontología por cada fuente de información (*ontologías fuentes*) que convergen en un mismo *vocabulario compartido* (u ontología global). Luego, la propuesta se basa en el enfoque de ontología híbrido [61]. Por ello, se adiciona un tercer componente dentro de la federación que maneje la información entre las ontologías fuentes y el vocabulario compartido. Este componente se denomina *OM* (Ontology Mapping, correspondencia entre ontologías) [13, 15].

Al realizar las correspondencias entre las ontologías fuentes y el vocabulario compartido surgen problemas inherentes a la *heterogeneidad ontológica* [59], debido a que cada ontología corresponde sólo a la fuente de información en la cual se originó y puede haberse creado en forma independiente. La Figura 3.1 muestra los componentes principales de la arquitectura propuesta.

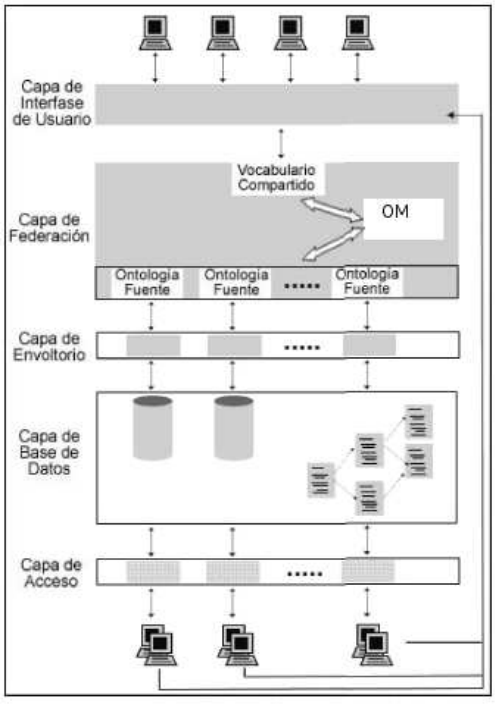

Figura 3.1: Arquitectura del sistema federado

Como podemos ver, esta arquitectura se basa en la arquitectura presentada en la Figura 2.2 del capítulo anterior. El trabajo se centra principalmente en la *capa de federación* adicionando los componentes necesarios para lograr la integración. La *capa de base de datos* comprende las fuentes de información que están involucradas en la federación. Las mismas pueden ser páginas HTML o textos en general, además de bases de datos. La *capa de envoltorio* comprende un conjunto

de módulos que corresponden a una organización de datos específica y que conocen cómo recuperar esos datos desde las fuentes subyacentes ocultando dicha organización. Así, el sistema accede a esta información abstrayéndose de los aspectos de implementación particulares de cada fuente.

Existe una comunicación entre las ontologías fuentes y cada uno de estos módulos (envoltorios) en el sistema, ya que juntos interactúan para recuperar la información necesitada. Como el sistema federado es autónomo, los usuarios locales acceden a su base de datos local por medio de la *capa de acceso* en forma independiente de los usuarios de otros sistemas. Por otro lado, para acceder al sistema federado deben utilizar la *capa de interfase de usuario*.

3.2. La capa de federación

El modelo de la *Capa de Federación* (CF) [13, 15, 16] es la parte central del trabajo en [12]. Dentro del mismo se deben solucionar los problemas relacionados con la heterogeneidad ontológica y para ello la capa posee tres componentes (Figura 3.1): las *ontologías fuentes*, el *vocabulario compartido* y el *OM*.

Como explicamos en la sección anterior, se considera una ontología fuente por cada fuente de información involucrada en la federación. Las relaciones de similitud (o correspondencias) entre conceptos de cada ontología fuente, son las que facilitan la tarea de crear el componente *vocabulario compartido*. Este componente es el que poseerá los conceptos genéricos, que involucran a todas las ontologías fuentes y los resultantes de la integración de las ontologías fuentes. Decimos resultantes ya que del componente *OM* se obtienen todos los *axiomas de igualdad* necesarios para indicar las relaciones de similitud que existen entre conceptos de dos o mas ontologías. Así, estas relaciones darán lugar a conceptos en el vocabulario compartido, obtenidos desde los conceptos en las fuentes.

Por lo tanto el vocabulario compartido contendrá una serie de conceptos construidos a partir de estos axiomas incluidos en el *OM*. Los usuarios utilizarán el vocabulario definido allí para consultar y obtener respuestas del sistema, por medio de la capa de interfase. Una vez formulada la consulta, el sistema usará el componente *OM* para determinar qué conceptos están relacionados con la consulta y así podrá acceder a las fuentes de información involucradas.

3.2.1. Método para la construcción de la capa de federación

En esta sección explicamos el método diseñado para construir los componentes definidos en la capa de federación [14, 15, 16, 18]. La Figura 3.2 muestra, en forma gráfica, dicho algoritmo. El método tiene tres etapas principales: *construir las ontologías fuentes*, *definir las correspondencias entre las ontologías fuentes* (componente *OM*) y *construir el vocabulario compartido*.

La primera etapa consiste en una actividad, la *búsqueda de propiedades y clases* (conceptos y roles). Ésta realiza un análisis completo de las fuentes de información, es decir, la información almacenada, el significado de esa información (la semántica), las restricciones, etc. Con toda esta información se puede iniciar el proceso de construcción de la ontología.

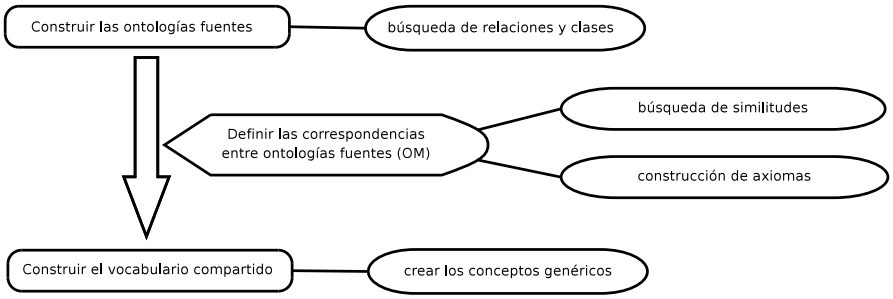

Figura 3.2: Método para la construcción del sistema

Existen en la literatura varias herramientas semi-automáticas para construir ontologías. Generalmente estas herramientas combinan aprendizaje computacional (machine learning), extracción de información y técnicas lingüísticas. Las principales tareas son: extracción de conceptos relevantes, construcción de jerarquías es-un y extracción de relaciones entre conceptos.

La segunda etapa, definir las correspondencias entre las ontologías fuentes (*OM*), contiene dos pasos: *búsqueda de similitudes* y *construcción de los axiomas de igualdad*. El primer paso, *búsqueda de similitudes*, es uno de los pasos mas importantes ya que es donde se buscan las similitudes entre los conceptos. Para esto, se utiliza el método para búsqueda de similitudes diseñado en [12]. Éste será descripto detalladamente en la Sección 3.3.1. En el segundo paso, *construcción de los axiomas de igualdad*, se definen los axiomas de igualdad según los resultados devueltos en el paso anterior. Es decir, las comparaciones que han producido valores de similitud altos formarán parte de estos.

La tercera etapa, *construir el vocabulario compartido*, consiste en *crear los conceptos genéricos*. En este paso, dado los axiomas de igualdad definidos en la etapa anterior, se escogen los conceptos genéricos que formaran parte del vocabulario compartido. Los usuarios utilizarán el vocabulario definido en esta etapa para consultar y obtener respuestas del sistema, por medio de la capa de interfase. Una vez formulada la consulta, el sistema usará el componente *OM* para determinar qué conceptos están relacionados con la consulta y así poder acceder a las fuentes de información involucradas.

3.3. El Proceso de Búsqueda de Similitudes

La búsqueda de similitudes entre conceptos de diferentes ontologías es una tarea muy compleja, ya que en general no es posible determinar automáticamente todas las correspondencias que existan entre ellos. Por ello, el método de búsqueda de similitudes determina sólo correspondencias candidatas que el usuario puede aceptar, rechazar o cambiar. A su vez, el usuario podrá especificar correspondencias para conceptos donde el sistema fue incapaz de encontrar similitudes.

La mayoría de las ontologías que existen, no especifican su conceptualizacion sólo a través de estructuras lógicas, sino que, en gran medida, hacen referencia (o nombran) conceptos y relaciones con terminología generada haciendo uso del lenguaje natural. Es decir, cada concepto y relación de la ontología tiene asociado

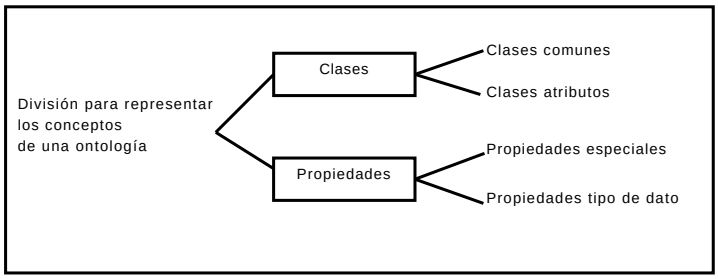

Figura 3.3: División para representar ontologías

uno ó mas términos. Luego, el método propuesto apunta a capturar las similitudes entre ontologías a dos niveles diferentes, léxico y conceptual. Primero, a nivel léxico se considera cómo los términos asociados a los componentes de las ontologías son usados para transmitir significados (comparación semántica) y las distintas formas asociadas a un término (sintaxis de la palabra). Segundo, a nivel conceptual se evalúan las similitudes entre las relaciones que existen asociadas a los conceptos.

El método se basa en los distintos componentes de las ontologías, como clases, propiedades y restricciones, y tres niveles de análisis: *sintáctico*, *semántico* y *usuario*, además del análisis de la estructura conceptual de las ontologías implícito en el método, a través de la comparación de relaciones.

3.3.1. Metodología

Con el propósito de comparar los conceptos de dos ontologías, el método hace una distinción entre los componentes de las mismas. La Figura 3.3 muestra como se dividen los diferentes componentes de una ontología. La primer división se refiere a dos componentes diferentes: *clases* (conceptos) y *propiedades* (relaciones). Primero, analizaremos la rama de las *clases*, la cual es también dividida en dos nuevas ramas: *clases comunes* y *clases atributo*. Ambas son clases definidas en la ontología para representar cosas acerca del mundo. El rol especifico definido en la ontología hace la diferencia entre ellas. Las *clases comunes* tienen el rol de representar cosas acerca del dominio y las *clases atributo* tienen el rol de representar información acerca de las *clases comunes*. Ambos roles existen porque algunos conceptos de la ontología actúan como atributos. Por ejemplo, una ontología podría tener la clase *Animal* como una clase común y la clase *Órgano* como una clase atributo porque ésta última existe para describir una característica acerca de una clase común. La clase *Órgano* no tiene propiedades.

En la otra rama, la Figura 3.3 muestra la rama de *propiedades* la cual también se divide en dos nuevas ramas: *propiedades tipo de dato* y *propiedades especiales*. Una propiedad es un conjunto de tuplas que representan una relación entre objetos en el universo del discurso. Cada tupla es una una secuencia finita y ordenada de objetos (por ejemplo listas). Las propiedades tienen restricciones para denotar por ejemplo funciones, cardinalidad, dominio, rango, etc. Las *propiedades tipo de dato* son propiedades que relacionan clases de un tipo de dato. Por ejemplo, el nombre de un animal es una propiedad tipo de dato entre la clase *Animal* y el tipo de dato *String*. Por otro lado, las *propiedades especiales* son propiedades que relacionan clases. Por ejemplo, la relación entre *Animal* y *Órgano* para denotar los órganos

de un animal.

La Figura 3.4 muestra el enfoque de tres niveles, presentado en [17, 19], creado para el proceso de búsqueda de similitudes. Según este enfoque los conceptos de las ontologías se analizarán usando tres niveles de comparación: *sintáctico*, *semántico* y *usuario*. Como podemos ver, además de los módulos que involucran alguno de estos niveles, existen otros tres módulos que son necesarios para el funcionamientos de todo el proceso. El primero de ellos, *Recuperar propiedades y clases atributo*, es el encargado de recuperar las clases atributos y las propiedades especiales y tipo de dato de los dos conceptos ingresados por el usuario. Los dos primeros pasos que realiza el proceso son:

1. *Entrada del usuario*: El usuario indica la primer correspondencia entre dos conceptos.

2. *Obtención de conceptos relacionados*: El módulo *Recuperar propiedades y clases atributos* se encarga de obtener toda la información respecto a las relaciones con otros conceptos (clases atributos y las propiedades especiales y tipo de dato).

Los dos módulos restantes, son aquellos representados mediante rectángulos doblemente encerrados. Estos son módulos externos que los otros módulos requieren para recuperar información y así completar su funcionamiento. El módulo *Instanciación de la Ontología*, es el encargado de devolver las instancias de las clases y propiedades solicitadas por el módulo *Recuperar propiedades y clases atributos*. Cuando el usuario eligió las dos ontologías a comparar, el proceso, mediante otros módulos no representados en la Figura 3.4, crea la estructura de objetos mediante la instanciación de los mismos en las clases que representan el dominio de las ontologías (clases, propiedades, restricciones, etc.). Justamente de esta estructura se obtiene la información solicitada.

El módulo Tesauro es el encargado de buscar información dentro de un Tesauro. El método utiliza WordNet [53] (versión ingles) ya que nos provee con la información semántica deseada y cumple con los requerimientos de una fuente de información semántica. Específicamente este módulo se usa para la búsqueda de sinónimos entre las palabras ó términos asociados a los conceptos dados. La función de similitud $sim_{tesauro}(x, y)$ retorna 1 cuando x e y (términos asociados a conceptos) son sinónimos, es decir, se encuentran en el tesauro como posibles sinónimos, y 0 cuando no lo son.

A continuación, se explican los módulos que son parte de cada uno de los tres niveles de comparación.

3.3.2. Comparación Sintáctica

El módulo *Comparación Sintáctica*, que se encuentra dentro del nivel sintáctico de nuestro enfoque, toma la información recuperada por el módulo *Recuperar propiedades y clases atributos* y compara las clases y propiedades de los conceptos ingresados por el usuario mediante dos funciones sintácticas: *editar distancia* (*edit distance*) [45, 47] y *trigrama* (*trigram*) [46], y las funciones *tipo de dato* y

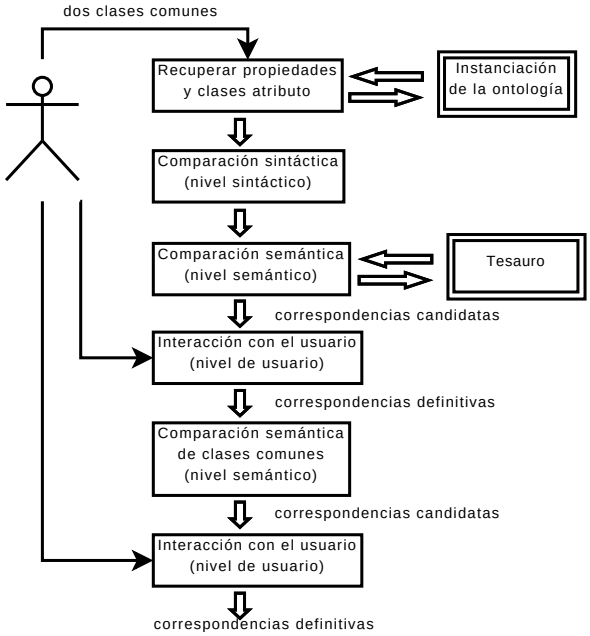

Figura 3.4: Método para búsqueda de similitudes

restricciones que analizan las características y restricciones de las propiedades, o relaciones entre los conceptos de la ontología.

La función *editar distancia*, es un método para ponderar la diferencia entre dos cadenas de caracteres. Ésta calcula el mínimo número de cambios necesarios para convertir una cadena en la otra. Por ejemplo, $ed(hamaca, hamacas) = 1$, porque debe hacerse una inserción para convertir la cadena *hamaca* en *hamacas*. En base a la función *editar distancia* de Levenshtein, en [47] se propone una medida de similitud léxica para strings, $sim_{ed}(x,y)$ (3.1), que compara los términos x e y asociados a los conceptos.

sim_{ed}, retorna un valor de similitud entre 0 y 1, donde 1 denota una igualdad perfecta y 0 una desigualdad. Esta función considera el número de cambios necesarios para convertir una cadena en la otra y pesa el número de esos cambios contra la longitud de la cadena mas corta.

$$sim_{ed} = max\left(0, \frac{min(\mid x \mid, \mid y \mid) - ed(x,y)}{min(\mid x \mid, \mid y \mid)}\right) \in [0,1] \tag{3.1}$$

Luego, siguiendo con el ejemplo, $min(|hamaca|, |hamacas|) = min(6,7) = 6$, por lo tanto $sim_{ed}(hamaca,hamacas) = max(0,5/6) = 5/6$.

La función *trigrama* (3.2) esta basada en el número de diferentes trigramas en dos cadenas de caracteres. La función *tri(x)* representa el conjunto de trigramas de x. Por ejemplo, $tri(hamaca) = \{$ham,ama,mac,aca$\}$. Si comparamos la palabra *hamaca* con la palabra *hamacas*, tenemos $|tri(hamaca)| = 4, |tri(hamacas)| = 5$ y $(|tri(hamaca)| \cap |tri(hamacas)|) = 4$, por lo tanto $sim_{tri}(hamaca, hamacas) = 1/2$.

$$sim_{tri}(x,y) = \frac{1}{1+ \mid tri(x) \mid + \mid tri(y) \mid) - 2 \times \mid tri(x) \cap tri(y) \mid} \tag{3.2}$$

La función *tipo de dato* o *compatibilidad de tipo de dato* (3.3) es muy sencilla ya que compara los tipos de datos de los rangos de las propiedades tipo de dato en las que participan los dos conceptos dados. Por ejemplo, cadena de caracteres contra cadena de caracteres, o entero contra cadena de caracteres, etc. La función retorna 1 si existe una conversión lógica [11] de un tipo de dato en el otro, de otra forma retorna 0.

$$sim_{dtc}(dtp_1, dtp_2) = \begin{cases} 1 & \text{si } \exists \text{ compatibilidad de tipo de dato entre r}(dtp_1) \text{ y r}(dtp_2) \\ 0 & \text{si } \nexists \text{ compatibilidad de tipo de dato entre r}(dtp_1) \text{ y r}(dtp_2) \end{cases} \tag{3.3}$$

donde $r(dtp_1)$ y $r(dtp_2)$ son los rangos de las propiedades dadas dtp_1 y dtp_2 respectivamente. Por ultimo, la función para restricciones sim_{rest} compara las restricciones aplicadas a las propiedades dadas (por ejemplo, mínima cardinalidad, máxima cardinalidad, todos lo valores, etc.) y los axiomas de propiedades que definen características sobre las mismas (por ejemplo, funcional, transitiva, simétrica, etc). Estas restricciones son las permitidas por el lenguaje OWL (Apéndice B). La función retorna un valor igual a 1, sólo cuando ambas propiedades poseen las mismas restricciones. De otra forma retorna un porcentaje de acuerdo al número de restricciones que son iguales. Además, utilizamos una tabla de compatibilidad para encontrar restricciones que posean los mismos significados semánticos. La Tabla 3.1 muestra las compatibilidades entre las restricciones aplicadas a las propiedades. El significado de algunos datos de la tabla son:

- Cardinality(1,1) significa *exactamente uno*

$$Cardinality(1,1) \begin{cases} \text{minCardinality(1) y maxCardinality(1)} \\ \text{or} \\ \text{cardinality(1)} \end{cases}$$

- Cardinality(1,-) significa *al menos uno*

$$Cardinality(1,-) \Longrightarrow minCardinality(1)$$

- Cardinality(0,1) significa *no mas de uno*

$$Cardinality(0,1) \begin{cases} \text{minCardinality(0) y maxCardinality(1)} \\ \text{or} \\ \text{maxcardinality(1)} \end{cases}$$

Por ejemplo, si una propiedad se define como funcional y otra posee una restricción de cardinalidad "no mas de uno", el porcentaje de similitud entre ambas es igual a 1 ya que denotan el mismo significado.

La Tabla 3.2 muestra las tres últimas compatibilidades entre las restricciones `allValuesFrom` y `someValuesFrom`.

```
Dadas dos clases comunes c1 y c2 de dos ontologías
    Por cada propiedad tipo de dato dtp_i ∈ c1 y dtp_j ∈ c2
        proceso_de_limpieza(dtp_i)
        proceso_de_limpieza(dtp_j)
        sim_dt_ed(dtp_i, dtp_j) = sim_ed(dtp_i, dtp_j)
        sim_dt_tri(dtp_i, dtp_j) = sim_tri(dtp_i, dtp_j)
        sim_dtc(rango_de(dtp_i), rango_de(dtp_j))
    Por cada propiedad especial sp_i ∈ c1 y sp_j ∈ c2
        proceso_de_limpieza(sp_i)
        proceso_de_limpieza(sp_j)
        sim_sp_ed(sp_i, sp_j) = sim_ed(sp_i, sp_j)
        sim_sp_tri(sp_i, sp_j) = sim_tri(sp_i, sp_j)
        sim_rest(sp_i, sp_j) = chequea_restricciones(sp_i, sp_j)
        sim_rango_total(sp_i, sp_j) = búsqueda de similitudes para
        (rango_de(sp_i), rango_de(sp_j))
    Para c1 y c2
        proceso_de_limpieza(c1)
        proceso_de_limpieza(c2)
        sim_clase_ed(c1, c2) = sim_ed(c1, c2)
        sim_clase_tri(c1, c2) = sim_tri(c1, c2)
```

Figura 3.5: Algoritmo para el módulo de comparación sintáctica.

Se debe aclarar, que las restricciones sobre propiedades que se tienen en cuenta en esta función sim_{rest} son sólo aquellas que definen una superclase restricción para la clase dominio de la propiedad especial.

Para comparar las restricciones tenemos que R y R' son las restricciones de las propiedades especiales P y P' respectivamente. $R \cap R'$ posee las restricciones que son iguales o similares (extrayendo los valores de la Tabla 3.1). Estos valores son sumados, pero cuando no hay relación entre las restricciones no se efectúa ninguna suma (o se suma 0). Luego, para el valor final se divide sobre la cantidad mayor de restricciones. La función es la siguiente (3.4):

$$sim_{rest}(\mathcal{R}, \mathcal{R}') = \frac{\mid \mathcal{R} \cap \mathcal{R}' \mid}{max(\mid \mathcal{R} \mid, \mid \mathcal{R}' \mid)} \tag{3.4}$$

La Figura 3.5 muestra el algoritmo usado para la comparación en el nivel sintáctico.

El método comienza con las dos clases seleccionadas por el usuario. Como vimos anteriormente, existen varios módulos encargados de obtener la información que proveen estas dos clases, es decir, sus propiedades y sus clases atributos. Obviamente, toda esta información podrá obtenerse sólo si las clases son clases comunes; recordemos que las clases atributos no poseen propiedades.

Cuando el usuario selecciona dos clases comunes, el algoritmo ingresa dentro del primer ciclo en donde se comparan la propiedades tipo de dato de ambas clases. Previo a la comparación, el algoritmo posee una función llamada *proceso_de_limpieza* que realiza un proceso de eliminación de artículos, preposiciones (entre, de, en, etc.) y caracteres no relevantes (_, ., -, etc.). Por ejemplo, *proceso_de_limpieza*("fecha_de_nacimiento") = fecha_nacimiento.

Este proceso de limpieza es sumamente necesario ya que las preposiciones y caracteres extraños están presentes en la mayoría de las clases y propiedades de las ontologías. Si nosotros no realizáramos ningún proceso de limpieza, los nombres

Axiomas-Restricciones	**Symmetric**	**Functional**	**Inverse Of**	**Inverse Functional**	**Cardinality(1,1)**	**Cardinality(1,-)**	**Cardinality(0,1)**
Symmetric	1	0	1	0	0	0	0
Functional	0	1	0	0.8	0.8	0	1
Inverse Of	1	0	1	0	0	0	0
Inverse Functional	0	0.8	0	1	0.8	0	1
Cardinality(1,1)	0	0.8	0	0.8	1	0	0.8
Cardinality(1,-)	0	0	0	0	0	1	0
Cardinality(0,1)	0	1	0	1	0.8	0	1

Cuadro 3.1: Compatibilidad de restricciones y axiomas de propiedades especiales.

Restricciones	SomeValuesFrom	AllValuesFrom
SomeValuesFrom	1	0.5
AllValuesFrom	0.5	1

Cuadro 3.2: Compatibilidad de otras restricciones de propiedades especiales.

utilizados para representar los conceptos en la ontología no serían encontrados por el tesauro ya que la información que este provee se encuentra de una determinada forma. Por ejemplo, si quisiéramos buscar en inglés la palabra “domestic dog" deberíamos hacerlo de la forma “domestic_dog" para obtener un resultado favorable.

Luego el algoritmo realiza la comparación utilizando las funciones de similitud que vimos en párrafos anteriores. Primero compara las propiedades usando la función (3.1) la cual puede devolver 0 en el caso de que los items léxicos asociados a las propiedades sean cadenas de caracteres diferentes. La función (3.2) también compara los items léxicos asociados a las propiedades en forma sintáctica pero teniendo en cuenta su composición en trigramas. Lo particular de esta función es que nunca devuelve 0, por lo tanto cuando dos items léxicos son diferentes pero sinónimos esto se convierte en una ventaja y se debe tener en cuenta para determinar el umbral (que veremos mas adelante). Por último, las propiedades se comparan con la función (3.3) que evalúa los tipos de datos ($sim_{dtc}(rango\_de(dtp_i), rango\_de(dtp_j))$) de los rangos de las mismas.

En el segundo ciclo, se vuelve a repetir el proceso anterior sólo con las propiedades especiales de las clases comunes ingresadas por el usuario. Se vuelven a utilizar las funciones (3.1) y (3.2) pero no la función (3.3) ya que los rangos aquí no son tipos de datos sino clases.

La siguiente función invocada en el algoritmo es la función $sim_{rest}(sp_i, sp_j)$ que verifica cuáles restricciones se aplicaron a cada propiedad especial [56], tal como, funcional, simétrica, cardinalidades, etc.; y devuelve 1 cuando se aplicaron exactamente las mismas restricciones y un porcentaje estimativo cuando existe alguna intersección entre ellas. Luego, la función $sim_{rango\_total}(sp_i, sp_j)$ calcula la similitud de las propiedades especiales teniendo en cuenta el rango de las mismas. Recordemos que el rango de las propiedades especiales son clases comunes o clases atributos. En cualquier caso, estas clases rango deben compararse para poder determinar si son realmente similares y afirmar que los rangos de las propiedades concuerdan. Para realizar esta comparación, se debe volver a ejecutar todo el método desde el principio, es decir, si las clases rango son clases comunes se deben obtener todas las propiedades de las mismas y realizar los pasos descriptos anteriormente. Por lo tanto, el método es recursivo y terminará cuando los rangos sean clases atributos (ya que no poseen propiedades) ó sean clases comunes sólo con propiedades tipo de dato. La comparación de las propiedades especiales terminará cuando se comparen el conjunto de clases rangos de cada una de ellas; notar que a su vez una clase rango común también puede tener propiedades especiales con rangos del mismo tipo.

Por último, se comparan las clases entre ellas, es decir, usando las funciones (3.1) y (3.2) se calcula la similitud de las mismas a nivel sintáctico. Nótese aquí, que si las dos clases (o al menos una de ellas) que seleccionó el usuario hu-

bieran sido clases atributos, el método realizaría directamente la comparación de las mismas, ya que estas clases no poseen propiedades.

Es importante destacar aquí que éste análisis recursivo que realiza el método comparando las propiedades que vinculan dos pares de clases, la clase dominio y la clase rango, implica recorrer la ontología como grafo. Por lo tanto, debemos tener en cuenta que siguiendo el análisis de las propiedades podemos retroceder volviendo a un par de clases previamente visto durante el análisis. Luego en la Sección 3.4, explicamos en detalle cómo el método recorre la ontología en forma de grafo y en qué casos podemos encontrar ciclos. En la Sección 3.5 vemos una ampliación del método de búsqueda de similitudes completo incluyendo el manejo de ciclos y algunas mejoras para evitar recalcular valores.

3.3.3. Comparación Semántica

Los resultados de las comparaciones efectuadas en cada ciclo se envían al módulo siguiente, *Comparación Semántica*. Este módulo compara las clases y propiedades a nivel semántico. Al igual que el módulo anterior, se analiza cada *propiedad tipo de dato* y cada *propiedad especial* de las clases dadas y luego se comparan las *clases*; sólo cambian las funciones utilizadas para la comparación. Se utilizan dos tipos de información en común: la información que se obtiene del tesauro, semántica léxica, y los resultados generados en el nivel sintáctico para cada elemento de la ontología. La Figura 3.6 muestra el algoritmo para este módulo.

En el primer ciclo, los items léxicos asociados a las propiedades tipo-de-dato se buscan en el tesauro para saber si son sinónimos. Como habíamos visto, la función $sim_{tesauro}(x,y)$ retorna 1 cuando los items léxicos son sinónimos y 0 cuando no lo son. Luego, la función $sim_{dt\_total}(dtp_i, dtp_j)$ se calcula utilizando el resultado de aplicar la función tesauro y los resultados de las funciones a nivel sintáctico. Esta retorna un valor entre 0 y 1 obtenido a partir de pesos que le colocamos a la función para cada tipo de comparación efectuada. Por último, si el resultado de esta función excede un umbral ($th_{aceptar}$), se agregará una correspondencia candidata entre las dos propiedades tipo-de-dato (dtp_i y dtp_j).

El segundo ciclo utiliza las mismas funciones que la anterior sólo que los resultados de las funciones utilizadas aquí son las correspondientes a las propiedades especiales del módulo sintáctico. Note que aquí se utiliza también el resultado de haber comparado las clases rango de cada propiedad. Nuevamente, si el resultado de la función total (sim_{sp_total}) excede un umbral ($th_{aceptar}$) se agregará una correspondencia candidata entre las propiedades especiales (sp_i y sp_j).

Finalmente, se comparan las clases entre ellas a nivel semántico. Como en los dos ciclos anteriores, primero se determina si las clases son sinónimos y luego con los resultados del módulo sintáctico para las clases, se calcula la función $sim_{clase\_total}(c_1,c_2)$. Si las clases son clases atributo, se agrega una correspondencia candidata entre las mismas cuando el resultado de dicha función excede un umbral ($th_{aceptar}$). Si las clases son clases comunes, la comparación no generará correspondencias candidatas sino que el resultado será enviado al módulo *Comparación Semántica de clases comunes*. Esto se debe a que las clases comunes poseen ambos tipos de propiedades (especiales y tipo de datos) que proveen mayor información semántica, a diferencia de las clases atributos que no poseen propiedades y

```
Por cada propiedad tipo de dato dtp_i ∈ c_1 y dtp_j ∈ c_2
      sim1_tesauro(dtp_i, dtp_j) = busqueda_en_tesauro(dtp_i, dtp_j)
      sim_dt_total(dtp_i, dtp_j) = w_ed * sim_dt_ed(dtp_i, dtp_j)+
      w_tri * sim_dt_tri(dtp_i, dtp_j) + w_dtc*
      sim_dtc(rango_de(dtp_i), rango_de(dtp_j)) + w_tesauro*
      sim1_tesauro(dtp_i, dtp_j)
      si sim_dt_total(dtp_i, dtp_j) ≥ th_aceptar
          agregar_correspondencia(dtp_i, dtp_j)
Por cada propiedad especial sp_i ∈ c_1 y sp_j ∈ c_2
      sim2_tesauro(sp_i, sp_j) = busqueda_en_tesauro(sp_i, sp_j)
      sim_sp_total(sp_i, sp_j) = w_ed * sim_sp_ed(sp_i, sp_j)+
      w_tri * sim_sp_tri(sp_i, sp_j) + w_tesauro*
      sim2_tesauro(sp_i, sp_j) + w_rest * sim_rest(sp_i, sp_j) + w_total*
      sim_total(sp_i, sp_j)
      si sim_sp_total(sp_i, sp_j) ≥ th_aceptar
          agregar_correspondencia(sp_i, sp_j)
Para c_1 y c_2
      sim3_tesauro(c_1, c_2) = busqueda_en_tesauro(c_1, c_2)
      sim_clase_total(c_1, c_2) = w_ed * sim_clase_ed(c_1, c_2)+
      w_tri * sim_clase_tri(c_1, c_2) + w_tesauro * sim3_tesauro(c_1, c_2)
      si c_1 y c_2 son clases atributo
          si sim_clase_total(c_1, c_2) ≥ th_aceptar
              agregar_correspondencia(c_1, c_2)
```

Figura 3.6: Algoritmo para el módulo de comparación semántica.

por lo tanto no es necesaria otra instancia de comparación. Otra vez, si las clases que se comparan son clases atributos (o al menos una de ellas) el método sólo ejecutará la comparación de las clases y no ingresará en los ciclos para comparación de propiedades.

Según la Figura 3.4, el módulo siguiente es *Interacción con el Usuario* que se encuentra dentro del Nivel de Usuario. En este módulo, se muestran al usuario todas las correspondencias candidatas encontradas en los dos módulos anteriores (sintáctico y semántico) para que decida cuáles son correctas y cuales no, es decir, debe determinar si realmente los conceptos encontrados como equivalentes por el sistema realmente lo son. Las correspondencias que acepta el usuario se clasifican como correspondencias definitivas y se almacenan en el sistema. Las clases comunes que se compararon en el módulo anterior no se muestran al usuario ya que todavía deben pasar una instancia más de comparación.

Comparación semántica de clases comunes

El siguiente módulo es *comparación semántica de clases comunes*, que como su nombre lo indica, compara las clases comunes a nivel semántico. Recordemos que en el módulo semántico anterior las clases se compararon usando la información del tesauro y la comparación sintáctica. En este módulo, las clases comunes se comparan mediante una nueva instancia semántica, precisamente se consideran las estructuras semánticas de la ontología analizando las relaciones entre conceptos. Es decir, aquí se contemplan las propiedades en común, tanto especiales como tipo de dato, de las clases comunes. La función (3.5) que utilizamos para la comparación es parte de un modelo computacional basado en las relaciones semánticas definido en [54, 55]. Éste combina el proceso de *correspondencia de características* con la *medición de distancia semántica*, ya que además de considerar las características comunes y diferentes entre dos conceptos, define la relevancia de las características

en términos de la distancia entre los conceptos dentro de una estructura jerárquica. Hemos elegido esta función ya que se corresponde perfectamente con nuestros requerimientos. La misma está basada en el modelo del cociente (*ratio model*) del proceso basado en características [58].

En la función $sim_{att}(c_1,c_2)$, *att* denota atributos (o propiedades) de clases, c_1 y c_2 son clases y C_1 y C_2 son los conjuntos de características (o atributos) respectivas a c_1 y c_2. El análisis de correspondencias se efectúa mediante las diferencias e intersecciones de conjuntos que se ven en la función. Se calcula la cardinalidad (||) del conjunto resultante entre la intersección ($C_1 \cap C_2$) que denotan el conjunto de elementos que pertenecen a C_1 y también a C_2; y las diferencias (C_1 - C_2) y (C_2 - C_1) que denota el conjunto de elementos que pertenecen a C_1 y no a C_2 y viceversa, respectivamente.

$$sim_{att}(c_1, c_2) = \frac{\mid C_1 \cap C_2 \mid}{\mid C_1 \cap C_2 \mid + \alpha(c_1, c_2) \mid C_1/C_2 \mid + (1 - \alpha(c_1, c_2)) \mid C_2/C_1 \mid} \quad (3.5)$$
$$para\ 0 \leq \alpha \leq 1$$

Donde, la función α se determina mediante la distancia entre las clases y la superclase inmediata que incluye ambas clases. En una evaluación de múltiples ontologías, como es nuestro caso, no existe una superclase común entre los conceptos. Una aproximación para obtener el nivel de generalización consiste en conectar dos ontologías mediante una raíz imaginaria y mas general llamada *una_ clase* (*thing*). La Figura 3.7 muestra dos ontologías conectadas por medio de esta superclase.

Por ejemplo, si el concepto *animal* se describe mediante tres atributos (color, peso y edad) y el concepto *animales* también se describe mediante tres atributos (color, edad y mamífero), obtenemos:

$$\mid C_1 \cap C_2 \mid = \mid \{color, peso, edad\} \cap \{color, edad, mamfero\} \mid = 2;$$
$$\mid C_1/C_2 \mid = \mid \{color, peso, edad\}/\{color, edad, mamfero\} \mid = 1;$$
$$\mid C_2/C_1 \mid = \mid \{color, edad, mamfero\}/\{color, peso, edad\} \mid = 1;$$

Siguiendo, si en este caso asumimos que $\alpha(animal, animales) = 0{,}5$. Luego, $sim_{att}(animal, animales) = 2/3$.

La función (3.6) es la ecuación utilizada para calcular estas distancias. Para esto, se utiliza el concepto de profundidad (*depth*) que mide el nivel en que cada concepto esta dentro de una jerarquía. Por ejemplo, en la Figura 3.7, si quisiéramos comparar el concepto automóvil de la ontología (a) con el concepto *automóvil* de la ontología (b) tendríamos que la profundidad (*depth*) de $automóvil^{(a)}$ es de 3 y la de $automóvil^{(b)}$ es de 2.

$$\alpha(a^p, b^q) = \begin{cases} \frac{profundidad(a^p)}{profundidad(a^p)+profundidad(b^q)} & profundidad(a^p) \leq profundidad(b^q) \\ 1 - \frac{profundidad(a^p)}{profundidad(a^p)+profundidad(b^q)} & profundidad(a^p) > profundidad(b^q) \end{cases} \quad (3.6)$$

La Figura 3.8 muestra el algoritmo de este módulo en el cual se usan las funciones 3.5 y 3.6. Nótese, que hemos dividido la Función 3.5 ($sim_{att}(c_1,c_2)$) en dos

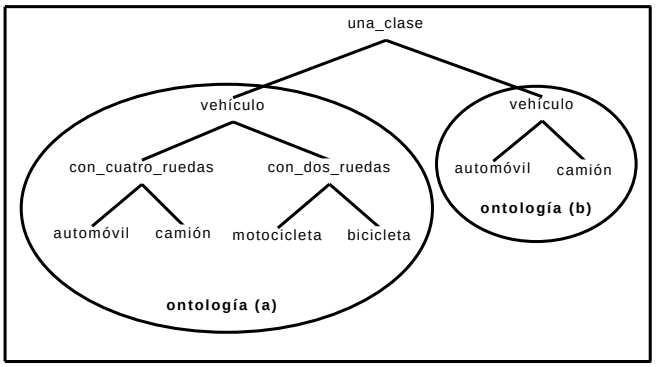

Figura 3.7: Dos ontologías unidas por la superclase *una_ clase*

Si c_1 y c_2 son clases comunes
$sim_{clase\_comun\_total}(c_1, c_2) = w_{clase\_total} * sim_{clase\_total}(c_1, c_2)+$
$sim_{propiedades\_especiales}(c_1, c_2)+$
$w_{propiedades\_tipo\_tipo\_de\_dato} * sim_{propiedades\_tipo\_tipo\_de\_dato}(c_1, c_2)$
si $sim_{clase\_comun\_total}(c_1, c_2) \geq th_{aceptar}$
agregar_correspondencia(c_1, c_2)

Figura 3.8: Algoritmo para el módulo de comparación semántica de clases comunes

funciones nuevas, $sim_{propiedades\_especiales}(c_1,c_2)$ y $sim_{propiedades\_tipo-de-dato}(c_1,c_2)$ dependiendo del tipo de propiedades. Esta división permite separar el cálculo de las propiedades especiales y tipo de dato para incrementar los pesos cuando así se lo requiera. Por ejemplo podríamos querer incrementar el peso en las propiedades comunes, ya que poseen mas información semántica que las tipo de dato.

Nuevamente, si el resultado de la función total ($sim_{clase\_comun\_total}$) excede un umbral ($th_{aceptar}$) se agregará una correspondencia candidata entre c_i y c_j. Es importante señalar, que aquí se utilizan las correspondencias definitivas que seleccionó el usuario en el módulo anterior. Sólo las correspondencias entre propiedades que fueron seleccionadas como correctas ingresarán en este módulo y se compararán en esta función. La función $sim_{clase\_comun\_total}(c_1,c_2)$ también posee el resultado de la comparación semántica de las clases comunes que se realizó en el módulo de comparación semántica. Recordemos que el resultado de esta comparación no se mostró al usuario.

Luego de que se efectúan todos los cálculos y se obtienen las correspondencias candidatas, se muestran al usuario en el módulo de *Interacción con el Usuario* para que el usuario decida si las correspondencias son correctas. Si lo son, se almacenan en forma permanente.

Es importante remarcar que el enfoque de tres niveles para la búsqueda de similitudes (Figura 3.4) genera correspondencias uno-a-uno ya que compara un concepto de una ontología con algún otro concepto de otra ontología.

3.4. Detección de ciclos

En este trabajo, estudiamos los casos en que las definiciones de las ontologías pueden causar ciclos en el método de búsqueda de similitudes. En consecuencia,

proponemos una extención al método para hacerlo rubusto ante estos casos agregando el manejo de ciclos.

Como explicamos en la sección anterior, sabemos que el método de análisis hace un recorrido en profundidad cuando analiza las propiedades especiales de las clases de las ontologías. Dadas dos clases, de dos ontologías distintas, se comparan todas las propiedades especiales de cada una de ellas. Comparar un par de propiedades especiales implica, a su vez, que se comparen las clases de sus rangos. Para la comparación de las clases en los rangos, nuevamente se comparan sus propiedades especiales. Y así sucesivamente se avanza de forma recursiva, en búsqueda de valores de similitud para pares de clases. Este análisis recursivo termina cuando, por lo menos, alguna de las dos clases comparadas no posea propiedades especiales.

De esta manera, la tarea del análisis de propiedades especiales se puede ver como un grafo que se recorre en profundidad. Informalmente, un grafo consiste de un conjunto de nodos, y un conjunto de arcos entre nodos. En nuestro caso, un nodo representa un problema de similitud entre dos clases a ser resuelto y los arcos, partiendo desde un nodo, representan la descomposición en subproblemas para la resolución del nodo.

Definición. Un grafo (dirigido), consiste de un conjunto de N nodos y un conjunto A de pares (ordenados) de nodos llamados arcos (dirigidos). Un nodo n_2 es un vecino del nodo n_1 si hay un arco del nodo n_1 al n_2. Esto es, $(n_1, n_2) \in A$.

Definimos ahora el grafo $G = (Nodos, Arcos)$ que recorre el método de análisis. G es aquel donde cada nodo se puede definir como la comparación entre un par de clases y cada arco que parte del nodo como las propiedades especiales que se deben comparar para dichas clases. Es decir, de un nodo saldrán tantos arcos como combinaciones de a dos de todas las propiedades especiales de ambas clases en comparación haya. Luego, cada nodo vecino estará formado por las clases que integren el rango de las dos propiedades especiales que definen los arcos hacia ese nodo vecino. Dicho en estos términos, el proceso continúa hasta que todos lo nodos alcanzables desde el nodo origen han sido descubiertos. Cuando se ha buscado sobre todo un subárbol para un nodo (par de clases en comparación), la búsqueda retrocede ("backtracks") para explorar comenzando con el siguiente nodo del mismo nivel. Se busca en profundidad tanto como sea posible en busca de valores de similitud para pares de clases.

La forma de evaluación del método establece un *orden*, es decir, hay una precedencia en el análisis. De lo mencionado hasta ahora, resaltamos que para obtener los valores de similitud entre dos clases (resolver un nodo del grafo), primero hay que comparar y obtener los valores de similitud entre sus propiedades especiales (resolver primero todos los hijos ó vecinos del nodo). Implícitamente se esta definiendo un orden ó precedencia de evaluación.

Por otro lado, recordemos las definiciones de camino y ciclo:
Camino. Un camino desde un nodo s a un nodo g es una secuencia de nodos $n_0, n_1, ..., n_k$, tal que $s = n_0, g = n_k$ y hay un arco desde n_{i-1} a n_i. Otra forma de expresar el camino es $< n_0, n_1 >, < n_1, n_2 >, ..., < n_{k-1}, n_k >$.
Ciclo. Un ciclo es un camino no vacío donde el nodo final es igual al nodo comienzo, es decir $n_0 = n_k$, y $k <> 0$. Un grafo dirigido sin ciclos es llamado Grafo

Dirigido Acíclico, DAG.

En un grafo dirigido acíclico un orden lineal de todos los nodos es tal que si G contiene un arco $< n_1, n_2 >$, luego n_1 aparece antes de n_2 en el ordenamiento. Si el grafo no es acíclico el orden lineal no es posible. Del mismo modo, el grafo de búsqueda de similitudes sobre propiedades especiales tampoco debe contener ciclos para que sea posible la obtención de valores de similitud y el algoritmo termine. A continuación, analizamos algunos casos y veremos en cuáles de ellos es posible que el recorrido en profundidad contenga ciclos.

Cuando los rangos de propiedades especiales (clases) de ambas ontologías están en el camino de análisis

En este caso, el análisis en profundidad que hace el método sobre la ontología es posible que encuentre ciclos en el grafo de búsqueda de similitudes. La causa de la existencia de ciclos tiene que ver con la definición de clases comunes y propiedades especiales que haya en la ontología. En las Figuras 5.7 y 5.6, se muestra una parte de dos ontologías donde la comparación de clases de estas mismas podría generar ciclos. En las figuras sólo se grafican las clases y las propiedades especiales.

- La ontología denominada *"Travel Ontolgy"* [1] modela vuelos, agencias de viajes, alquiler de autos, hoteles, etc. y posee alrededor de 40 clases junto con gran cantidad de propiedades para describirlas, ya sea propiedades de tipo de dato o especiales. De esta ontología sólo describimos aquí la parte que tomamos para nuestro ejemplo. La clase *Airport* que tiene dos propiedades especiales: la propiedad *hasFlightTo* que tiene como clase rango a *Airport* y la propiedad *hasCity* que tiene como clase rango a *City*. Esta última, a su vez, tiene una propiedad especial que es la propiedad *hasAirport* con la clase *Airport* como rango.

- La ontología *"Location Ontology"* [2] posee cinco clases y un conjunto de propiedades para representar un dominio de ubicación. De esta ontología sólo tomamos una parte para nuestro análisis, que describimos a continuación. La clase *Continent* sin propiedades especiales. La clase *Country* con dos propiedades especiales: *is_part_of* con la clase *Continent* de rango, y la propiedad *has_parts* que tienen a las clases *State* y *City* en el rango. Luego la clase *City* tiene la propiedad *is_part_of_Country* que la relaciona con la clase *Country* y la propiedad *is_part_of_State* con la clase *Sate*. Y por ultimo, la clase *State* con la propiedad *is_part_of* que tiene como rango a la clase *Country* y la propiedad *has_parts* con la clase rango *City*.

Cuando evaluamos un nodo $A = (c_1, c_2)$, estamos buscando el valor de similitud de dos clases. Este valor depende a su vez de los valores de similitud hallados para los nodos vecinos (hijos). Los nodos hijos $(A_1, A_2, A_3, ..., A_n)$ también representan una comparación de dos clases. Si al analizar alguno de los nodos $A_i = (c_i, c_j)$ tenemos algún arco (para nuestro dominio una propiedad especial) que nos lleva

[1] `www.ilby.net/travel.owl`

[2] `http://www.liacs.nl/CS/DLT/pickups/sjoerd/for\%20Protege/Science.zip`

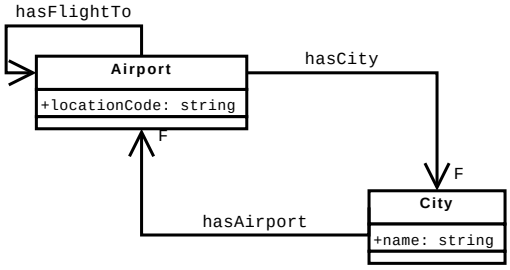

Figura 3.9: Parte de la ontología *"Travel"*

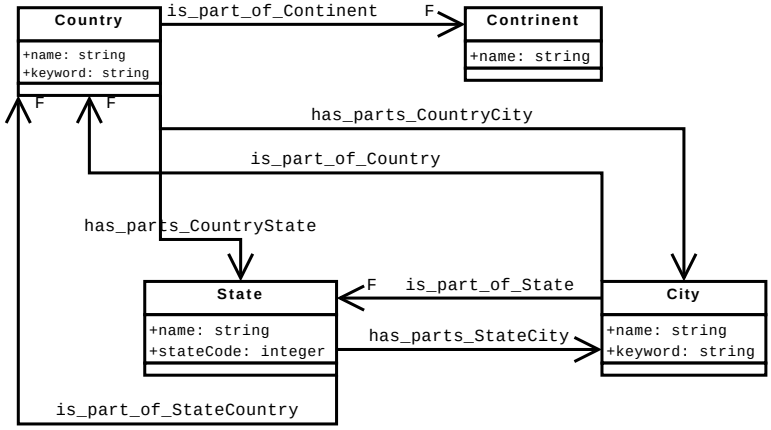

Figura 3.10: La ontología *"Location"*

desde A_i al nodo A nuevamente, significa que la similitud de dos clases (c_1, c_2) , depende de la evaluación de dos clases (c_i, c_j) y que a su vez esta depende de (c_1, c_2). Así, el método entra en un ciclo infinito ya que ambas evaluaciones dependen de la resolución de la otra.

Como ejemplo, para mostrar cómo se llega en el método a analizar un nodo ya visitado, véase la Figura 3.11. En la misma se comparan las clases *City* de ambas ontologías y tenemos que resolver la similitud entre $A = (Travel.City, Location.City)$ dentro de los vecinos (hijos) a evaluar tenemos $B = (Travel.Ariport, Location.Country)$. Para resolver este último tenemos que evaluar todos sus nodos adyacentes dentro de los cuales tenemos a $C = (Travel.City, Location.City)$. Luego no podemos resolver A porque antes tenemos que resolver B, y a su vez no podemos resolver B porque antes tenemos que resolver $A = C$.

Cuando los rangos de propiedades especiales (clases) de una de las ontologías están en el camino de análisis

En el caso anterior vimos que en ambas ontologías había definiciones cíclicas de propiedades especiales, es decir, un par de clases donde ambas son clase rango de la otra. A diferencia de ésto, si una de las ontologías tiene que en el recorrido en profundidad el rango de una clase nos lleva a una clase que ya se visitó, pero no sucede lo mismo en las propiedades especiales de la otra ontología, entonces no habrá problemas ciclos. Por ejemplo, veamos que pasaría en la comparación de

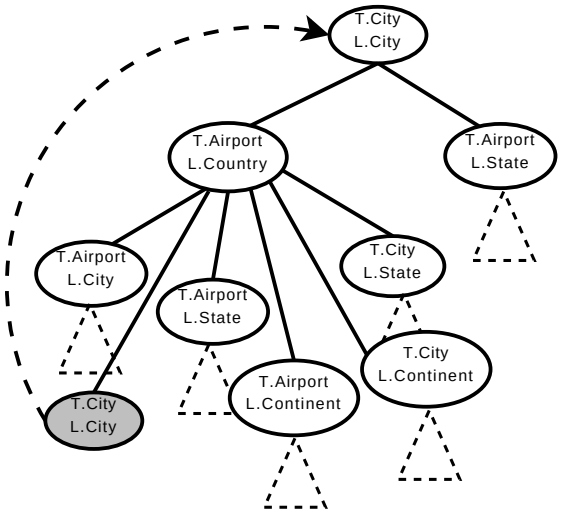

Figura 3.11: Grafo parcial para búsqueda de similitud entre las clases *City* de las ontologías *Travel* (T) y *Location* (L). Ciclo.

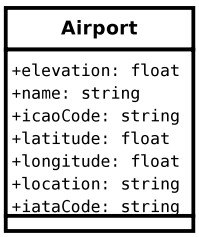

Figura 3.12: La ontología *"Airport"*

la ontología *Travel* del punto anterior (Figura 5.7) y la ontología *Airport* (Figura 3.12).

- La ontología *"Airport Ontology"*[3] es muy sencilla y sólo modela la clase *Airport*. La clase tiene siete propiedades tipo de dato y ninguna propiedad especial. Por ejemplo, la propiedad `name` tiene como rango el tipo de dato *String* y la propiedad `longitude` con el tipo de dato *float* como rango.

En este ejemplo, comparamos las clases *Airport* de ambas ontologías. como la clase *Airport* de la ontología *"Airport Ontology"* no tiene propiedades especiales, no habrá análisis de dichas propiedades y por lo tanto no existe posibilidad de ciclos. Si bien, al analizar directamente el rango de la propiedad especial *hasFlightTo* de la clase *Airport* en la ontología *"Travel Ontology"* e indirectamente al analizar el rango de la propiedad *hasCity*, que nos llevaría a analizar la clase *City*, y al analizar el rango de la propiedad especial *hasAirport* de *City* volvemos a la clase *Airport* de donde partimos, no habrá ciclo ya que en la otra ontología no tenemos propiedades especiales cuyos rangos nos lleven a clases ya visitadas.

[3] `http://www.daml.org/2001/10/html/airport-ont`

Cuando los rangos de propiedades especiales (clases) no están en el camino de análisis para ninguna de las ontologías

En el recorrido en profundidad ninguna de las combinaciones de propiedades especiales analizadas llevan al análisis de un par de clases que ya se visitó. En este caso no habrá posibilidad de ciclos ya que siempre se estarán analizando nuevos pares de clases, ó visitando nuevos nodos.

Manejo de ciclos

Para el primer caso mencionado, el único que contiene ciclos, es necesario que se detecten y eliminen los ciclos para evitar bucles en el recorrido en profundidad. Si un nodo u aparece antes que v en el camino, el objetivo es resolver siempre v antes que u eliminando los ciclos. Luego, es necesario modificar el método de búsqueda de similitudes agregando primero la detección de ciclos y segundo la eliminación y determinación de valores de similitud ante estos casos.

En primer lugar, la estrategia para la detección de ciclos genérica, es marcar como visitados los nodos durante el recorrido transversal del grafo con el propósito de evitar visitar el mismo nodo dos veces en el recorrido en profundidad de un camino del grafo. Para ello, se puede mantener el camino que va desde un nodo raíz al nodo actual, es decir, todos los nodos que lo componen. Si al avanzar en el recorrido alcanzamos un nodo que ya estaba en el camino, significa que nos encontramos con un ciclo. Esta estrategia es muy simple y es efectiva para la detección de ciclos.

En segundo lugar, se define una forma para la eliminación ciclos y determinación del valor de similitud en el caso de hallarlo. Al detectar un ciclo en un nodo del grafo, no se iniciará el proceso de análisis de los descendientes y se devolverá un valor parcial. Este valor parcial, será calculado sólo con la información del nodo mismo. Es decir, por medio de un análisis sintáctico y semántico, pero sin el análisis estructural que provee el análisis de propiedades especiales. Teniendo en cuenta que este nodo es a su vez adyacente de otro nodo que respeta un orden de evaluación, hay otro nodo ancestro cuyo valor depende de la resolución del nodo actual donde se corta el ciclo. Así, el valor parcial devuelto tendrá incidencia en los valores hallados para los nodos ancestros. Se observa que podrá influir tanto de manera negativa si eran estructuralmente similares (es decir, se podría haber obtenido un valor de similitud alto a partir de la evaluación de las propiedades) ó erróneamente positiva si sintácticamente son similares pero sus estructuras muy diferentes (es decir, se podría haber obtenido un valor de similitud muy bajo si se hubieran evaluado las propiedades especiales).

3.5. Método de búsqueda de similitudes extendido

A continuación, en la Figura 3.13 mostramos el algoritmo completo para el método de búsqueda de similitudes incorporando el control de ciclos, según el análisis realizado en la sección anterior. También describimos una extensión, que tiene como objetivo reducir el número de pasos en el proceso de búsqueda de similitudes, al pasar por nodos que ya fueron resueltos.

Similarity(O1,O2)
el usuario selecciona dos clases c_1 y c_2
si c_1 y c_2 son clases comunes
si$< c_1, c_2 > \notin Visitados$
$agregar\_visitado(c_1, c_2)$
para cada propiedad tipo de dato $dtp_i \in c_1$ y $dtp_j \in c_2$
$proceso\_de\_limpieza(dtp_i, dtp_j)$
$sim1_{tesauro}(dtp_i, dtp_j) = busqueda\_en\_tesauro(dtp_i, dtp_j)$
$sim1_{lex}(dtp_i, dtp_j) = w_{ed} * sim_{ed}(dtp_i, dtp_j) + w_{tri}*$
$sim_{tri}(dtp_i, dtp_j) + w_{tesauro} * sim1_{tesauro}(dtp_i, dtp_j)$
$w_{tesauro} * sim1_{tesauro}(dtp_i, dtp_j)$
$sim_{dtp}(dtp_i, dtp_j) = w_{lex} * sim1_{lex}(dtp_i, dtp_j) + w_{dtc}*$
$sim_{dtc}(rango\_de(dtp_i), rango\_de(dtp_j))$
si $sim_{dtp}(dtp_i, dtp_j) \geq th_{aceptar}$
$agregar\_correspondencia(dtp_i, dtp_j)$
por cada propiedad especial $sp\_i \in c_1$ y $sp_j \in c_2$
$proceso\_de\_limpieza(sp_i, sp_j)$
$sim2_{tesauro}(sp_i, sp_j) = busqueda\_en\_tesauro(sp_i, sp_j)$
$sim_{rest}(sp_i, sp_j) = verificar\_restricciones(sp_i, sp_j)$
$sim2_{lex}(sp_i, sp_j) = w_{ed} * sim_{ed}(sp_i, sp_j) + w_{tri} * sim_{tri}(sp_i, sp_j)+$
$w_{tesauro} * sim2_{tesauro}(sp_i, sp_j) + w_{rest} * sim_{rest}(sp_i, sp_j)$
si $(rango\_de(sp_i), rango\_de(sp_j)) \notin Correspondencias\_Encontradas$
$sim_{total}(sp_i, sp_j) =$
$calcular\_todo\_el\_proceso\_para(rango\_de(sp_i), rango\_de(sp_j))$
si $(rango\_de(sp_i), rango\_de(sp_j)) \in Correspondencias\_Encontradas$
$sim_{total}(sp_i, sp_j) = obtener\_valor(rango\_de(sp_i), rango\_de(sp_j))$
$sim_{spp}(sp_i, sp_j) = w_{lex} * sim2_{lex}(sp_i, sp_j) + w_{total} * sim_{total}(sp_i, sp_j)$
si $sim_{spp}(sp_i, sp_j) \geq th_{aceptar}$
$agregar\_correspondencia(sp_i, sp_j)$
$quitar\_visitado(c_1, c_2)$
usando las correspondencias encontradas
$proceso\_de\_limpieza(c_1, c_2)$
$sim3_{tesauro}(c_1, c_2) = busqueda\_en\_tesauro(c_1, c_2)$
$sim3_{lex}(c_1, c_2) = w_{ed} * sim_{ed}(c_1, c_2) + w_{tri} * sim_{tri}(c_1, c_2)+$
$w_{tesauro} * sim3_{tesauro}(c_1, c_2)$
si c_1 y c_2 son clases atributo
$sim_{clase}(c_1, c_2) = sim3_{lex}(c_1, c_2)$
si c_1 y c_2 son clases comunes
$sim_{clase}(c_1, c_2) = w_{lex} * sim3_{lex}(c_1, c_2) + w_{propiedadtes\_especiales}$
$*sim_{propiedadtes\_especiales} + w_{propiedadtes\_tipo\_de\_dato} * sim_{propiedadtes\_tipo\_de\_dato}$
si $sim_{clase}(c_1, c_2) \geq th_{aceptar}$
$agregar\_correspondencia(c_1, c_2)$

Figura 3.13: Método de búsqueda de similitudes extendido.

En el algoritmo del cuadro 3.13 vemos la implementación de estas extensiones. Con la verificación si $< c_1, c_2 > \notin Visitado$, se verifica si el nodo actual *no* está en el camino de análisis. De ser así, se procede con la verificación de las clases rango, análisis de propiedades especiales, para hallar el valor de sim_{total}. En caso contrario, si el nodo ya esta en el conjunto de visitados, si $< c_1, c_2 > \in Visitado$, no se analizarán sus propiedades especiales. Nótese, que en el cálculo del valor de similitud total para las clases c_1 y c_2, el operando $w_{propiedadtes\_especiales} * sim_{propiedadtes\_especiales}$ será nulo. Sólo se comparan las propiedades tipo de dato, igualmente, el valor resultante de esta comparación junto con el valor de la comparación sintáctica de las clases puede resultar mayor que el umbral. Luego, aunque no estemos considerando las propiedades especiales, igualmente este par de clases estaría en condiciones de generar una correspondencia candidata entre clases. Teniendo en cuenta que estamos frente a un ciclo, es decir no es la primera vez que se intenta analizar este par de clases, y la primer instancia de análisis esta todavía pendiente, sólo se utilizará este valor de similitud parcial, sin incluir las propiedades especiales. Se consultará al usuario por el mismo, con la salvedad de que aunque el usuario acepte dicho valor no se agregará correspondencia ninguna, sólo se retornara el valor para continuar con el análisis. La correspondencia definitiva se agregará finalmente si el usuario la acepta al momento de la resolución completa de la similitud entre las clases en la primer instancia de análisis para estas. Es decir, la primera vez que se visitaron en el recorrido del grafo.

Cuando se completa el análisis de un par de clases en el grafo, si el valor hallado no supera el umbral o es rechazado, no se agregará correspondencia alguna y tampoco se registra ninguna información acerca de la fallida comparación de las clases. Si más adelante en el proceso se vuelve a intentar el análisis del mismo par de clases, se hará el cálculo nuevamente ya que no quedó ningún dato registrado de análisis anteriormente realizado.

La segunda extensión, además de la detección de ciclos, es que antes de ejecutar nuevamente todo el proceso para las clases rango, se consulta si las clases rango están en el conjunto de correspondencias encontradas en algún recorrido de otro subárbol, si $(rango\_de(sp_i), rango\_de(sp_j)) \in Correspondencias\_Encontradas$. En caso afirmativo, se recupera el valor hallado con anterioridad para evitar hacer el mismo análisis nuevamente. Como se puede ver en la Figura 3.11, el factor de ramificación (arcos que salen de un nodo) suele ser muy alto cuando ambas clases en comparación tienen un cierto número de propiedades especiales, esto nos provee medidas de la complejidad del recorrido del grafo.

3.6. Trabajos relacionados

Existen otros métodos para mezcla y alineación de ontologias en la literatura. Entre ellos, en [51], se describe la herramienta PROMPT la cual implementa una herramienta interactiva de mezcla de ontologias que guia al usuario a lo largo del proceso. PROMPT comienza identificando las correspondencias entre nombres de clases para realizar actualizaciones automáticas, encontrando resultados que generan conflictos y haciendo sugerencias al usuario para remover estos conflictos. Cuando dos ontologías contienen clases con nombres similares (es decir, coinciden

los términos asociados a los conceptos) y propiedades tipo de dato nombradas de manera diferente, esta herramienta no sugiere si las propiedades son las mismas o no, porque no hay analisis de similitudes entre conceptos que son llamados de manera diferente. Así, este método es altamente dependiente de los nombres dados a los conceptos y de las herramientas de comparación que se configuren para el analisis de similitud entre estos nombres.

En [48] los autores presentan Chimaera como una una herramienta interactiva para mezcla de ontologias basada en el editor de ontologias Ontolingua [29]. Chimaera provee soporte para la mezacla de conceptos de ontologías de diferentes fuentes, verificando la cobertura y correctitud de las ontologias y manteniendo las ontologias en el tiempo. Chimaera al contrario de PROMPT no hace sugerencias al usuario. Las unicas relaciones que éste herramienta tiene en cuenta son relaciones subclase-superclase.

Otra estrategia es FCA-MERGE [57], en la cual se propone un método totalmente diferente. La comparación de ontologías se hace usando un conjunto compartido de instancias o un conjuto compartido de documentos anotados con conceptos de las ontologías fuente.

Finalmente, el trabajo presentado en [47] es similar al método de búsqueda de similitudes ya que una capa léxico-sintáctica y una capa conceptual son usadas para encontrar similitudes. Sin embargo, una diferencia es que en éste no se utilizan Tesauros u otras fuentes de información semántica. Al nivel de analisis léxico, el método utiliza una función léxica llamada medida de similitud (Similarity Measure, SM). A nivel conceptual, los concpetos de las ontologías (clases y propiedades) se comparan teniendo en cuenta la taxonomía en la que aparecen. Además, el concepto de cotopología semantica (Semantic Cotopy, SC) es usado para definir todos los super y subconceptos de un concepto específico en la jerarquía. Una función diferente se utiliza cuando se comparan las relaciones o propiedades. Para esto, los autores comparan los dominios y los rangos de las propiedades usando otro concpeto llamado cotopología ascendente (Upwards Cotopy, UC), el cual define los superconceptos de un concepto especifíco. Las propiedades tipo de dato no son consideradas por este método. Según [47], no hay sugerencias al usuario.

El método base para nuestro trabajo de tesis compara los dos tipos de propiedades. En cuanto a las propiedades tipo de dato, analiza las compatibilidades de tipo de dato y la similitud de los términos asociados a los conceptos (nombres de los conceptos). En cuanto a las propiedades especiales, se analiza también sintáctica y semánticamente tanto el dominio como el rango (que en este caso son clases). Además, en el método diseñado en [12] se consideran los términos asociados a las propiedades utlizando un Tesauro y se consideran las restricciones aplicadas a las propiedades. Todos los factores inflencian en la búsqueda de similitudes, porque dos ontologías pueden tener dos propiedades con el mismo dominio y rango, pero con diferentes significados.

3.7. Resumen

Se describió la arquitectura por niveles propuesta para un sistema federado. Específicamente, se profundizó la capa de federación ya que es aquí en donde

se encuentran los problemas de mayor nivel de complejidad. Es dentro de esta misma capa donde se desarrolló el método de tres niveles que permite encontrar similitudes entre conceptos.

Seguidamente, introducimos una extensión al método de búsqueda de similitudes para que pueda manejar ontologías con definiciones cíclicas. Para evitar el recalculo de valores incorporamos un control de correspondencias almacenadas.

Como vimos en este capítulo la propuesta para la integración de información se basa en una herramienta fundamental: las *ontologías*. Éstas se ven reflejadas en todo el proceso ya que el método, se basa principalmente en la integración de las mismas utilizando las cualidades semánticas que ellas proveen.

En el siguiente capítulo describimos el diseño de una herramienta de software que implementa el método de búsqueda de similitudes entre ontologías. La misma será implementada en forma de plugin para el editor de ontologías Protégé.

Capítulo 4

Diseño de la herramienta de software

El presente capítulo describe el diseño de la herramienta de software cuyo objetivo es la búsqueda de correspondencias entre dos ontologías. Ésta herramienta será un plugin para el editor de ontologías Protégé [5], el cual posee una arquitectura basada en plugins (Sección A.1). Así, nuestro diseño estará ligado en algunos aspectos con el diseño de Protégé.

En primer lugar, describimos la metodología de diseño utilizada como guía en el proceso de desarrollo, el cual es un diseño guiado por responsabilidades (Responsability-Driven Design - RDD)[63, 64]. Luego, mencionamos brevemente los estilos arquitecturales ó patrones de sistema en los cuales basamos el diseño de la arquitectura, MVC modelo-vista-controlador y PAC presentación-abstracción-control [20, 44].

Se describe el diseño del plugin haciendo uso de artefactos de UML (Unified Modeling Language, Lenguaje Unificado de Modelado) [31] para la representación de las clases que componen el diseño y el modelo de colaboraciones entre las mismas. Los nombres utilizados para las clases en el diseño están en idioma ingles. Esto se debe a que nuestra herramienta es un plugin para el editor de ontologías Protégé, y el mismo es una herramienta de software internacional. Así, si queremos que nuestro código este disponible, pensamos que es conveniente que tenga los nombres en dicho idioma. También creemos conveniente que en la explicación del diseño llamemos a las clases con los mismo nombres, es decir sin traducir.

4.1. Introducción

4.1.1. Método de diseño

La estrategia de diseño guiado por responsabilidades es un método semi-formal, que ofrece distintas técnicas para dar forma al pensamiento acerca de cómo dividir las responsabilidades de una aplicación en objetos y coordinar su trabajo. El nombre del método enfatiza el hecho que se mantiene a través de todas las actividades: el enfoque en las responsabilidades de la aplicación. Las responsabilidades describen *qué* debe hacer la aplicación para lograr su propósito. De esta manera,

se deja de pensar en los objetos como datos más algoritmos para pensar acerca de ellos como roles y responsabilidades.

La estrategia de diseño guiada por responsabilidades es un proceso de clarificación. Los requerimientos iniciales se transforman en descripciones y modelos iniciales; las descripciones iniciales en descripciones y modelos de objetos mas detallados; y los modelos de objetos candidatos en modelos detallados de sus responsabilidades y patrones de colaboración. El proceso de diseño se divide en dos fases: *diseño exploratorio* el cual crea un diseño inicial, y *refinamiento del diseño* el cual incluye actividades para hacer el diseño mas consistente, flexible, predecible y entendible.

En un modelo basado en responsabilidades, los objetos juegan roles específicos y ocupan posiciones bien definidas en la arquitectura de la aplicación. Es una comunidad de objetos donde cada uno es responsable de realizar cierta parte del trabajo. Los objetos colaboran de manera bien definida, haciendo contratos para alcanzar los objetivos globales de la aplicación. Con tal comunidad de objetos y asignando responsabilidades especificas a cada uno, se construye un modelo colaborativo de la aplicación.

Describimos a continuación algunos conceptos de diseño que componen la estrategia utilizados en nuestro proceso:

- Roles. Ningún objeto existe asilado, siempre es parte de alguna estructura mayor. Como parte integrante tiene un propósito especifico: el rol que juega dentro de un contexto. Un rol es un conjunto de responsabilidades que pueden intercambiarse, si hay mas de una clase de objeto que contempla el mismo conjunto de responsabilidades. Es útil pensar acerca de un objeto preguntandonos cual es su rol. Esto nos ayuda a concentrarnos en qué debe *ser* y qué debe *hacer*.

- Estereotipos de roles de objetos (Objet Role Stereotypes). Un objeto bien definido soporta claramente un rol bien definido. Se utilizan simplificaciones útiles, o *estereotipos de roles*, para focalizar las responsabilidades de un objeto. *Los estereotipos de roles* son caracterizaciones de los roles necesitados por una aplicación. Estos nos permiten caracterizar y pensar acerca de los objetos a un nivel mas alto ignorando detalles. Un objeto puede caracterizarse en mas de un rol. Se proponen como útiles los estereotipos siguientes:

  1. Contenedores de información ("information-holders"): conocen y proveen información.
  2. Estructuradores ("structurers"): mantienen relaciones entre objetos e información acerca de esas relaciones.
  3. Proveedores de servicios ("service-providers"): realizan una tarea, y en general proveen servicios de calculos.
  4. Coordinadores ("coordinators"): reaccionan a eventos delegando tareas a otros.
  5. Controladores ("controllers"): toman decisiones y dirigen las acciones de otros.

6. Interfaces ("interfacers"): transforman información y requerimientos entre distintas partes del sistema.

- Roles, Responsabilidades y Colaboraciones. Las responsabilidades son asignadas a roles, los cuales colaboran para llevar adelante sus responsabilidades. Comenzamos el diseño descubriendo objetos, asignandoles responsabilidades de conocimiento de información y de realización del trabajo de la aplicación. Los objetos trabajan en conjunto para lograr responsabilidades mas grandes. El modelo Roles-Responsabilidades-Colaboraciones es utilizado para mantener el enfoque en el comportamiento del software.

Como estrategia, para encontrar las clases candidatas se proponen los siguientes pasos de descubrimiento:

- Escribir una historia de diseño breve (Design Story) describiendo *qué* es importante para nuestra aplicación.
- Usar esta historia e identificar los temas principales que definen asuntos centrales de la aplicación.
- Buscar objetos candidatos que se relacionan y soportan cada tema.
- Verificar que estos candidatos representen conceptos clave.
- Buscar candidatos que representen mecanismos adicionales y parte de funcionamiento de la aplicación.
- Dar nombres, describir y caracterizar los objetos candidatos.
- Organizar los candidatos, busqueda de formas naturales de dividir la aplicación en grupos.
- Testear que sean apropiados verificando si representan abstracciones razonables.
- Defender la razón de inclusión de cada candidato.
- Cuando el descubrimiento disminuye, pasar a modelar responsabilidades y colaboraciones.

Los objetos candidatos generalmente representan el trabajo que realiza el sistema, es decir tareas directamente afectadas o conectadas a la aplicación. Por ejemplo, actividades de toma de decisiones, control y coordinación, estructuras y grupos de objetos, representaciones del mundo real acerca de las cuales la aplicación necesite saber algo, etc. Este tipo de abstracciones están intimamente relacionadas con los estereotipos de roles.

La estrategia para la formulación de responsabilidades, incluye las siguientes actividades:

- Identificar las responsabilidades del sistema especificadas o implicadas por los casos de uso.

- Identificar responsabilidades adicionales para completar algunos vacíos en casos de uso u otras descripciones del sistema.
- Extraer comportamiento del sistema aparte de los temas e historias de software.
- Seguir la cadena de razonamiento: *Qué si...luego...* y *cómo.*
- Formular responsabilidades que naturalmente surgen de estereotipos de roles.
- Identificar responsabilidades privadas necesarias para dar soporte a responsabilidades públicas.
- Indicar qué responsabilidades surgen a partir de las relaciones existentes ente clases de objetos candidatas.
- Formular responsabilidades asociadas con eventos importantes durante la vida de un objeto.
- Identificar responsabilidades asumidas por un objeto cuando se coloca en un entorno técnico determinado.

Por último, en cuanto al lenguaje de modelado para describir el diseño se utilizan los artefactos de UML siguientes:

- Diagrama de Estructura Estática: muestra el conjunto de clases y objetos que forman parte de un sistema, junto con las relaciones existentes entre estas clases y objetos. El mismo exhibe de una manera estática la estructura de información del sistema y la visibilidad que presenta cada una de las clases, dada por sus relaciones con las demás en el modelo.
- Diagramas de Colaboración: Los diagramas de interacción son modelos que describen la manera en que colaboran grupos de objetos para cierto comportamiento. La elección de este tipo de artefactos radica en su capacidad de exponer el contexto de la operación y ciclos en la ejecución.

4.1.2. Estilo arquitectural

La elección de una arquitectura de software [20, 44] es una parte crucial del proceso de diseño. La arquitectura de software es la realización de decisiones tempranas de diseño que tienen que ver con la descomposición del sistema en partes.

Se considera que hay por lo menos dos funciones distintas que un sistema con una interfase de usuario debe proveer: *presentación* la cual involucra la interacción con el usuario, y *aplicación* la cual se refiere al propósito subyacente del sistema. Hay aspectos del dominio Interacción Hombre-Maquina (HCI Human-Computer interaction) por lo que la tarea principal realizada por el usuario es dividida en subtareas, que serán sucesivamente divididas hasta llegar a nivel de acciones físicas que el usuario realiza en la presentación. Esta descomposición y secuenciamiento puede ser pensado como un diálogo entre el usuario y la aplicación. Luego *dialogo* es un tercer tipo de función que cada sistema interactivo debe soportar.

El patrón arquitectural MVC, Modelo-Vista-Controlador (Model View Controller), divide una aplicación interactiva en componentes, persiguiendo los atributos de calidad modificabilidad y transportabilidad. Se divide la funcionalidad en distintos componentes conformados por tres capas, *presentación*, *aplicación*, y *diálogo*. A su vez se separan entrada / salida no sólo de aplicación y dialogo, sino que también entre ellas mismas. El *modelo* contiene la funcionalidad y datos especificos de la aplicación . Las *vistas* muestran información al usuario a través de la presentación. Los *controladores*, manejan la entrada del usuario y la traducen a requerimientos de servicios. Las vistas y controladores conforman la interfase de usuario donde un mecanismo de propagación de cambios asegura la consistencia entre la interfase de usuario y el modelo.

El MVC evolucionó en el estilo arquitectural PAC (*presentación-abstracción-control*). Hay una diferencia en los atributos de calidad priorizados ya que en PAC son modificabilidad y escalabilidad. Cada terna PAC tendrá asignada cierta funcionalidad y consiste de tres componentes. Esta división separa aspectos de la interacción hombre-maquina de la funcionalidad y comunicación con otras ternas PAC. Ahora el nombre *control* es más significativo, ya que este subcomponente de la terna necesita controlar sus partes así como también mediar en la interacción entre la aplicación y la presentación.

En nuestro trabajo nos basamos en los patrones de sistema MVC y PAC para diseñar la arquitectura del plugin. Describimos como se organizan los objetos de software que componen el modelo, es decir su localización dentro de la arquitectura de la aplicación. Teniendo en cuenta la separación de funcionalidad en los componentes de la arquitectura, se ubican los objetos de software de acuerdo al rol que desempeñan. El diagrama en la Figura 4.1 muestra la arquitectura definida. El componente *Transacciones* (o *Control* y *Aplicación* en MVC) contiene los objetos que implementan la lógica del negocio y el control. Este componente, también es responsable de mediar entre los componentes *Modelo del Dominio* y *Presentación*, para evitar dependencias directas entre ellos. En el componente *Modelo del Dominio* (*Abstracción*) tenemos los objetos que implementan los conceptos del dominio o dominio del negocio. Finalmente, el componente *Interfase de Usuario* (*Presentación*) esta estructurado en clases de objetos que proveen funcionalidad de ventanas, menues y diálogos, manejan la entrada, y traducen eventos a requerimientos de servicios.

La comunicación entre objetos dentro de la arquitectura sigue las reglas siguientes:

- Los objetos colaboran dentro de cada componente.
- Cuando residen en distintos componentes, son los objetos en el componente *Transacciones* los que colaboran con los objetos en los componentes *Interfase de Usuario* y *Modelo del Dominio*.
- Sólo el componente *Interfase de Usuario* es expuesto al mundo exterior.
- Los componentes *Modelo del Dominio* e *Interfase de Usuario* no colaboran entre si.

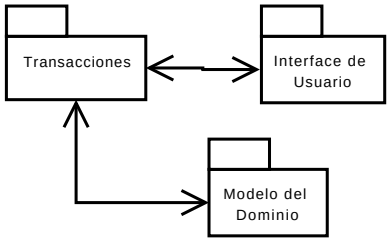

Figura 4.1: Arquitectura de software del plugin

4.2. Clases que componen el diseño

Una de las principales tareas en un diseño orientado a objetos es encontrar las clases que lo componen. Partimos de la especificación del método de búsqueda de similitudes, comportamiento que deberá proveer nuestra herramienta (descripta en la Sección 3.3.1), de las ontologías definidas en lenguaje OWL y de la especificación de Protégé (Sección A.1). No representamos todos los conceptos del dominio, sino sólo aquellos que son necesarios para dar soporte a la aplicación.

Objetos del dominio y objetos específicos de la aplicación

Los objetos del dominio representan conceptos que son familiares a usuarios y desarrolladores en un campo de interés especifico, en nuestro caso las ontologías y sus elementos, las correspondencias entre ellas, y el método de búsqueda de similitudes.

La Figura 4.2 muestra el diagrama de clases que modela el Modelo del Dominio de nuestra aplicación, es decir el modelo de conocimiento subyacente del método de búsqueda de similitudes (Sección 3.3.1). Las clases en el diagrama modelan los principales elementos que componen una ontología definida en el lenguaje OWL [4], clases, propiedades y restricciones. Recordemos también que, como se describió en el Capítulo 3, el método de búsqueda de similitudes clasifica a las clases y propiedades de una ontología de la siguiente manera: clases atributos *(AtributteClass)*, clases comunes *(CommonClass)*, propiedades tipo de dato *(DatatypeProperty)* y propiedades especiales *(SpecialProperty)*.

La clase *Ontology* modela con la composición *has_class* hacia la clase *Class* la relación todo-parte existente entre una ontología y sus elementos de tipo clase. Esto indica que cualquier instancia de la clase *Class* pertenecerá a una única ontología.

Como se ve en el diagrama, la clase *AtributteClass* y *CommonClass* se modelaron como una especialización de la clase *Class* ya que poseen ciertas diferencias tanto en su representación como en el análisis que el método de búsqueda aplica a cada una de ellas. Si bien ambas son definidas en la ontología para representar cosas sobre el mundo, la diferencia existe en el rol que cumplen dentro de la ontología. Las *clases comunes* tienen el rol de representar cosas sobre el dominio y las *clases atributo* tienen el rol de representar información sobre una clase común. Por otro lado, la asociación *has_superclass* representa la relación subclase/superclase que existe entre las clases de una ontología.

La clase *Property* tiene dos subclases que son las clases *DatatypeProperty* y

SpecialProperty. Una propiedad es un conjunto de tuplas que representa una relación entre objetos del dominio. De acuerdo a nuestra clasificación, las *propiedades tipo de dato* son propiedades que relacionan una clase o un conjunto de clases con un tipo de dato; es decir, las clases forman parte del dominio y el tipo de dato constituye el rango. Las *propiedades especiales* son propiedades que relacionan clases. Una *clase común* puede poseer ambos tipos de propiedades, tipos de dato y especiales, y las *clases atributo* no poseen ninguna. Por consiguiente, la asociación *has_property* representa la relación entre la subclase *CommonClass* y la superclase *Property*.

También, como los rangos de ambos tipos de propiedades son diferentes, las propiedades especiales poseen instancias de comparación diferentes a las propiedades tipo de dato. Para las clases *SpecialProperty* el rango es una o mas clases de la ontología, y esta representado por la asociación *has_classrange*. Por otro lado, el rango para la clase *DatatypeProperty* es sólo un tipo de dato XML; lo cual es modelado por la clase *XMLDatatype* y la asociación *has_XMLDatatyperange*. A su vez, cada tipo de dato puede ser compatible con cero o más tipos de datos. Esta relación esta modelada por la asociación *has_compatibility*.

Además, las propiedades especiales tienen una relación con la clase restricción denotando las restricciones definidas para cada propiedad. Por ejemplo, restricciones sobre propiedades (máxima cardinalidad, todos los valores de, etc.) y axiomas que definen características (funcional, inversa de, dominio, rango, etc). La asociación *has_compatibilty* indica las compatibilidades entre las restricciones asociadas a propiedades.

Siguiendo, la clase *Mapping* modela las correspondencias que se encuentran a medida que el método obtiene los valores de similitud entre los conceptos. Distinguimos entre las correspondencias de propiedades y las correspondencias de clases, dividiendo en dos tipos la clase *Mapping*: *PropertyMapping* y *ClassMapping*. Las asociaciones *has_classes* y *has_properties* representan la relación entre dos clases o dos propiedades que participan en una correspondencia. A su vez, cada correspondencia entre dos conceptos tiene un estado, ya que cuando se encuentra un valor de similitud que supera un umbral definido, se incorpora la correspondencia como *candidata*. En la instancia de comparación con el usuario, el estado puede cambiar a *confirmada* o *rechazada*. Las correspondencias del tipo propiedad que sean confirmadas serán utilizadas luego en el nivel de comparación semántico de las *clases comunes*.

En el Modelo de Transacciones se deben definir los objetos coordinadores para llevar a cabo las funciones de nuestra herramienta. La Figura 4.3 muestra la jerarquía de clases de este modelo. En la misma, se encuentra la clase *Similarity_Searcher* representando una abstracción del método de búsqueda de similitudes (Sección 3.3.1). La clase *ConceptAnalysis*, es la interfase que define las operaciones del análisis sobre los distintos tipos de conceptos (clases y propiedades). Luego, las clases *SpecialPropertyAnalysis*, *DatatypePropertyAnalysis* y *ClassAnalysis* implementan dichas operaciones definidas en la interfase. La clase *OWLSimMappingProcess* recibe las acciones introducidas por el usuario (a través de los objetos en el componente interfase de usuario) y coordina el inicio del proceso de búsqueda de similitudes. La clase *OntologyLoader* modela la funcionalidad necesaria para controlar la carga de las ontologías. Las clases *SemanticFunctions* y

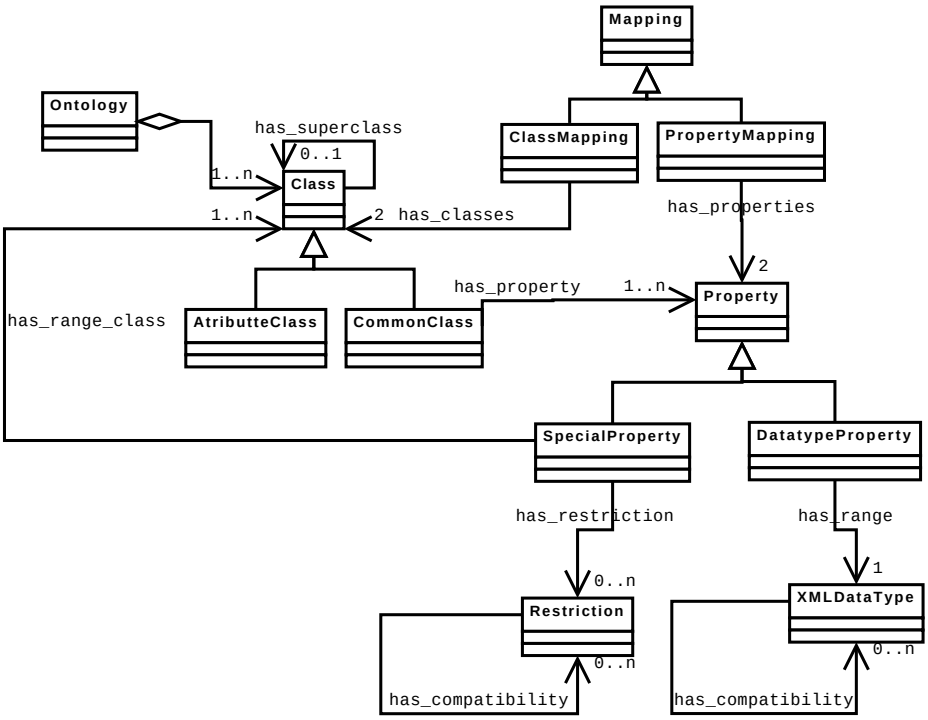

Figura 4.2: Diagrama de clases que modelan parte de las clases del dominio del plugin

SyntacticFunctions representan las funciones de similitud explicadas en la Sección 3.3.1.

La Figura 4.4 modela la jerarquía de las interfaces para la interacción con el usuario. La clase *UserInterface* es una superclase abstracta creada para modelar la categoría de formularios, menues y otros componentes gráficos que conforman las interfaces con el usuario.

Para integrar nuestra herramienta de software como un plugin de Protégé, usamos como base el plugin de tipo solapa ("Tab widget"). Luego, para proveer un componente de interfase de usuario de tipo solapa, que aparezca en la ventana principal de Protégé junto con otras solapas principales del sistema tales como la solapa de clases ("Classes Tab"), se debe extender la clase *AbstractTabWidget* (clase de Protégé [1]). Luego, el componente principal de interfase de usuario de nuestro plugin se representa por la clase *OWLSimTab*, la cual es una subclase de *AbstractTabWidget*.

Las dos pantallas principales del plugin están dadas por dos formularios representados por las clases *InitializationForm* y *MappingLayoutForm*. La primera permite al usuario la selección de las ontologías a analizar. La segunda muestra la jerarquía de clases de las ontologías a analizar, permite el inicio del proceso de análisis y muestra los resultados obtenidos. Estos formularios contienen componentes gráficos los cuales extienden componentes de la interfase gráfica de Protégé.

El formulario *MappingLayoutForm* consta de dos secciones principales. La parte derecha del formulario donde se muestran las jerarquías de clases de las ontologías que se eligen para analizar. Ésta, contiene los componentes del tipo *MapButton*, *SourcesPane* (contenedor de dos objetos del tipo *SourceClassPane*, uno

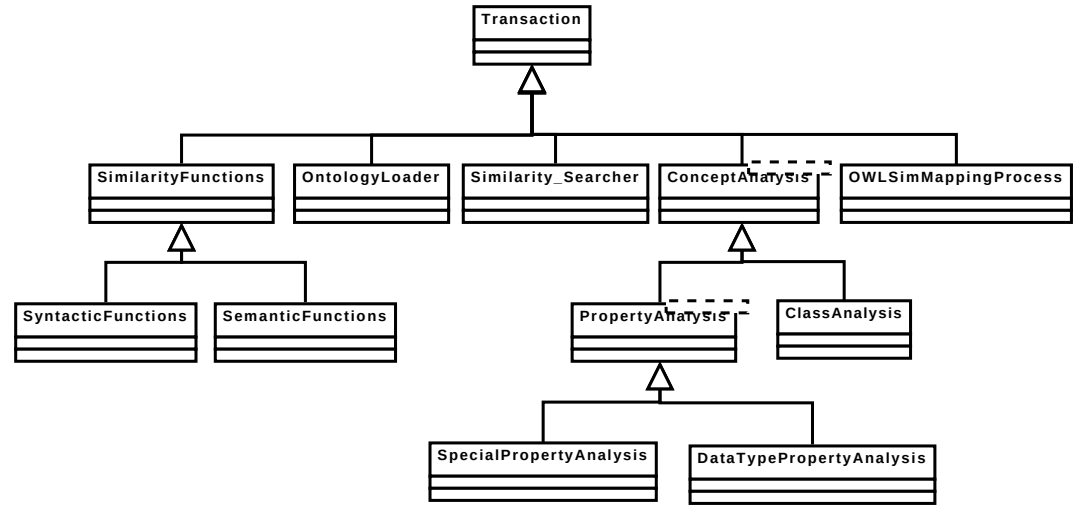

Figura 4.3: La jerarquía de transacciones.

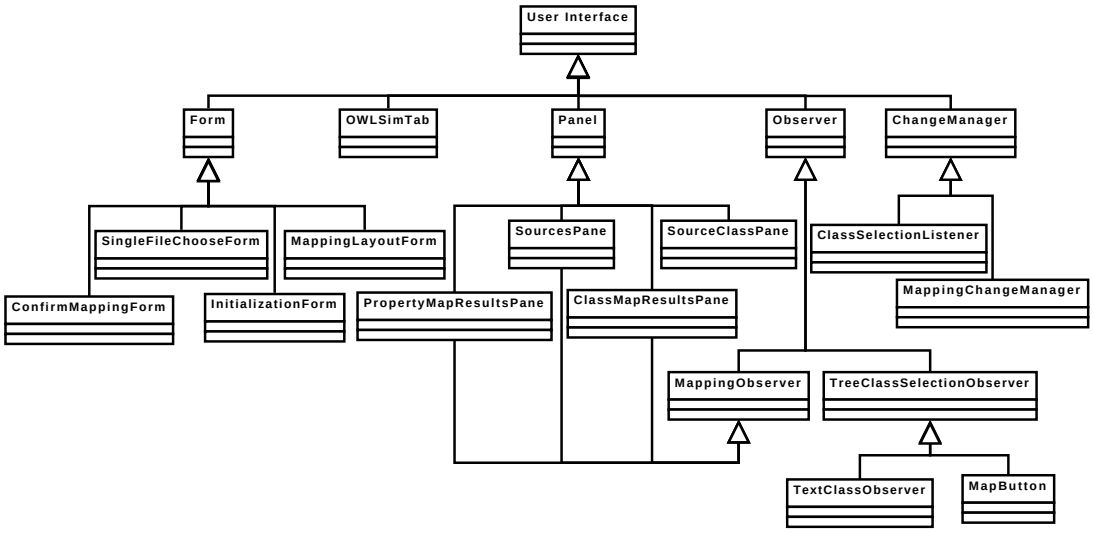

Figura 4.4: La jerarquía interfaces de usuario.

por cada jerarquía de clases). Los objetos del tipo *TextClassObserver* muestran los nombres de las clases seleccionadas. La parte izquierda, donde se muestran los resultados del proceso de análisis de correspondencias, consta de las listas de correspondencias analizadas para clases *ClassMapResultsPane* y propiedades *PropertyMapResultsPane*.

Por último, el formulario *SingleFileChooseForm* muestra una pantalla donde se pueden seleccionar los archivos de ontologías definidas en el lenguaje OWL. El formulario *ConfirmMappingForm* es la interfase utilizada para mostrar al usuario las correspondencias candidatas encontradas durante el proceso de búsqueda de similitudes.

La interfase de usuario se actualiza siguiendo el modelo del patrón *Observer* [32], tanto en la selección que hace el usuario de clases a analizar como en la actualización de las vistas de los resultados a medida que procede la búsqueda de similitudes. En las Figuras 4.5 y 4.6 se muestran estos modelos de actualización.

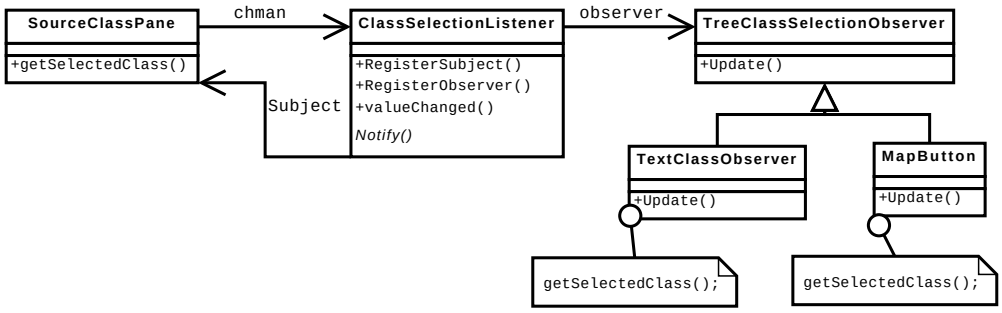

Figura 4.5: Elección de clases a analizar.

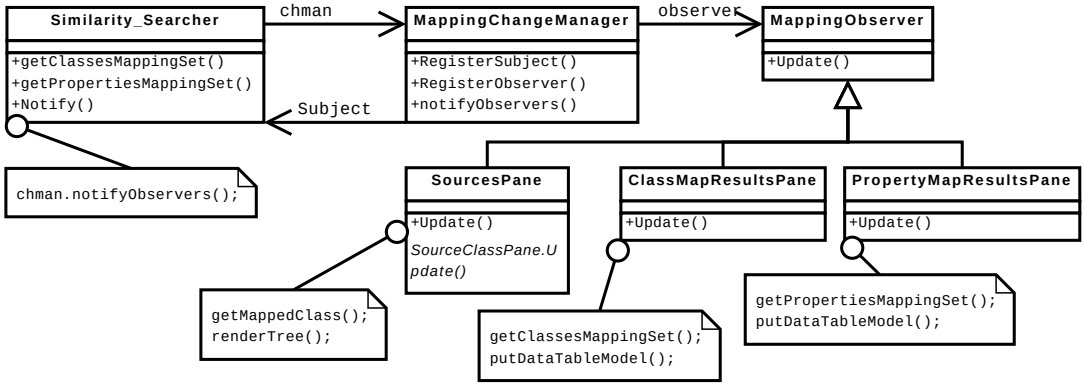

Figura 4.6: Actualización de vistas al cambiar el conjunto de correspondencias.

En ambos casos de la implementación del patrón observer las relaciones de dependencia entre sujetos y observadores es bastante compleja. Por ello, se incorpora un objeto que mantenga estas relaciones, llamado administrador de cambios, correspondiendo a la clase abstracta *ChangeManager* (Figura 4.4).

Al cambiar la clase seleccionada en el árbol de jerarquía de clases *SourceClassPane* se debe actualizar el cuadro de texto *TextClassObserver* que muestra el nombre de la clase seleccionada junto con el botón *MapButton* que recibe la orden del usuario para iniciar el análisis de correspondencias y que siempre conoce las clases a analizar. Llamamos a estos objetos observadores. Luego, cuando un cambio en la selección ocurre, esta clase debe informar a los observadores. Como la clase posee un objeto de tipo evento asociado *ClassSelectionListener* que automáticamente conoce cuando el cambio sucede, actúa como el administrador de cambios. Por eso en este modelo del patrón observer el administrador de cambios se notifica automáticamente por metodos internos heredados de la clase *TreeSelectionListener* de java.

Existen dos vistas de las correspondencias encontradas durante el proceso de búsqueda de similitudes. Por un lado, las clases *ClassMapResultsPane* y *PropertyMapResultsPane*, visualizan las correspondencias mostrando los valores obtenidos. Por otra parte, las jerarquías de clases de *SourcesPane*, marcan las clases de la jerarquía que ya fueron analizadas. La clase *MappingChangeManager* conoce a los sujetos y observadores, y es notificada por la clase *Similarity_ Searcher* cuando una correspondencia es hallada. Esta luego se encargará de notificar a los observadores, quienes procederán a actualizarse.

4.3. Responsabilidades asignadas a objetos

En la sección anterior mostramos las clases que modelan la herramienta, en esta sección continuamos con una declaración más precisa del propósito de las clases y el rol que las mismas juegan en el modelo de la aplicación. Para identificar y asignar responsabilidades buscamos las acciones que un objeto debe realizar, el conocimiento que debe mantener y las decisiones que un objeto toma afectando a otros. Para esta tarea tenemos en cuenta la especificación constituida por el método de búsqueda, el modelo de conocimiento subyacente (ontologías representadas en el lenguaje ontológico OWL) y Protégé. El hecho de que se haya identificado una clase indica que se vio la necesidad de abarcar una responsabilidad por lo menos. Las clases juegan ciertos roles lo cual implica la existencia de responsabilidades para cumplir dichos roles.

A continuación describimos las responsabilidades de cada clase contenida en los distintos componentes de la arquitectura descripta en la sección 4.1.2.

Como vimos, dentro del componente *Modelo del dominio*, o aplicación, se encuentran las clases de objetos que modelan los objetos específicos del dominio de la aplicación. Los roles característicos de los objetos en este componentes son contenedores de información, proveedores de servicios y estructuradores.

La clase *Ontology* conoce e informa sobre las clases que la componen, el rol que esta juega en la aplicación es de contenedora de información y estructuradora. De su relación con la clase *Class*, se identifica y se le asigna la responsabilidad de conocer e informar las clases que componen la ontología; es decir, es la responsable de responder las consultas sobre cuáles son las clases que componen la ontología. Su responsabilidad es:

- Mantener e informar las clases que componen el modelo en una ontología.

La clase *Class* se caracteriza como contenedora de información. De la asociación *has_ superclass* representando la relación subclase/superclase entre las clases de una ontología, se desprende la responsabilidad de conocer y proveer información acerca de la clase que es superclase de ella en el modelo de la ontología. Su responsabilidad es:

- Mantener e informar su superclase.

Las clases *AtributteClass* y *CommonClass* siendo subclases de *Class* comparten las responsabilidades asignadas a la superclase. Y en particular, la clase *CommonClass* es, además, responsable de conocer e informar las propiedades que posee la clase *CommonClass*, de esta forma se caracteriza dentro del estereotipo de rol de contenedora de información. La responsabilidad de *CommonClass* es:

- Mantener e informar las propiedades de tipo de dato y especiales.

Así mismo, la clase *Property* es la responsable de mantener y proveer información sobre el rango asociado caracterizándose bajo el rol de contenedora de información. Define esta responsabilidad para ambas subclases, *SpecialProperty* y *DatatypeProperty*, dado que los rangos de ambos tipos de propiedades son diferentes. Luego, cada una de ellas conoce e informa distintos tipos de objetos. La

clase *SpecialProperty* es también responsable de conocer e informar las restricciones definidas sobre las propiedades especiales. La clase *Restriction* es responsable de conocer e informar las compatibilidades entre las distintas restricciones. La clase *XMLDatatype* es responsable de conocer e informar las compatibilidades entre tipos de datos.

Por otro lado, a medida que el método de búsqueda avanza se van descubriendo correspondencias entre conceptos de las ontologías en comparación. La clase *Mapping* representa esta relación de correspondencia entre *clases* y entre *propiedades*. A la clase mapping se le asigna la responsabilidad de mantener e informar la relación entre dos clases *ClassMapping* o entre dos propiedades *PropertyMapping*. Así, se caracteriza por desempeñar el rol de un estructurador.

La clase *OWLSimTab* es subclase de la clase *AbstractTabWidget* del modelo de clases de Protégé [1], la cual es la superclase para todos los plugins del tipo *Tab Widgets* (Apéndice A). Esta clase, aunque pertenece a la jerarquía de clases *Interfase de Usuario*, es la clase principal del plugin. Por lo tanto, es responsable de:

- Crear la pantalla principal del plugin para el ingreso de las ontologías a comparar. → *InitializationForm*.
- Crear e inicializar los componentes para la interfase con el recurso lingüístico *WordNet*. → *JWNL* [28].
- Cargar de ontologías verificando que sean en el subjenguaje OWL-Lite. → *OntologyLoader*.
- Iniciar proceso de análisis de correspondencias entre ontologías. → *OWLSimMappingProcess*.

Nótese aquí, que para cada responsabilidad se indicó con qué clase se colabora, es decir, "→" indica una colaboración con las clases especificadas seguidamente de ésta. También, debemos mencionar, que la clase *JWNL* es una clase perteneciente al componente [28] que brinda el servicio de acceso al recurso lingüístico *WordNet*.

El componente *Transacciones*, es responsable de aceptar las entradas de usuarios como eventos, traducir estos eventos en la solicitud de servicios adecuadas, mostrar resultados, y llevar a cabo las transacciones que modelan la lógica del negocio. Los estereotipos de roles de objetos que se localizan en este componente son: coordinadores, controladores y proveedores de servicios.

La clase *OntologyLoader*, es la responsable de crear la estructura de objetos que representa una instanciación del modelo en la ontología. Si hubiese algún error al generar las estructuras a partir de los archivos en lenguaje OWL, éste informará al usuario. Su responsabilidad es:

- Crear e iniciar las estructuras de objetos que mantienen los modelos en las ontologías. → *Ontology*

La clase *Similarity_Searcher* es una abstracción que representa el método de búsqueda de similitudes entre conceptos de diferentes ontologías (Sección 3.3.1). Trabaja en colaboración con la clase *Class* para obtener las clases y propiedades. Y

también, con las clases *SemanticFunctions* y *SyntacticFunctions* que cumplen el rol de proveer un servicio, que es calcular y proveer ciertos valores de similitud. Éstas implementan las funciones descriptas también en la Sección 3.3.1. Las responsabilidades de esta clase son complejas para llevarlas a cabo por sí mismas. Luego, con el fin de hacer más claro el diseño y su implementación, las principales responsabilidades de *Similarity_ Searcher* se dividieron en sub-responsabilidades reasignadas a clases colaboradoras. Así, la clase *Similarity_ Searcher* modifica su rol para encargarse de coordinar un trabajo cooperativo entre otras clases. Para llevar adelante su tarea colabora con tres clases *SpecialPropertyAnalysis*, *DatatypePropertyAnalysis* y *ClassAnalysis*. La responsabilidad de la clase *Similarity_ Searcher* es la búsqueda de similitudes entre conceptos de dos ontologías:

- Analizar correspondencias de propiedades de tipo de datos. → *DatatypePropertyAnalysis*.
- Analizar correspondencias de propiedades especiales. → *SpecialPropertyAnalysis*.
- Analizar correspondencias de clases. → *ClassAnalysis*.

La clase *DatatypePropertyAnalysis* es la responsable de la comparación desde la perspectiva de las propiedades tipo de datos, es decir participa en la comparación de dos clases haciéndose responsable de la comparación de propiedades tipo de dato que estas clases poseen. Las funciones que utiliza son *Thesaurus* para la busqueda de sinónimos, y las funciones de similitud *Edit Distance*, *Trigram* y *Data Type*. Si el valor de similitud hallado para cada par de propiedades comparadas excede el umbral determinado, se agrega una correspondencia candidata. Después de la comparación de todas las propiedades tipo de datos, se mostrarán al usuario las correspondencias candidatas para que confirme los resultados obtenidos. Luego esta clase maneja el requerimiento de la clase *Similarity_ Searcher*: buscar correspondencias entre propiedades tipo de dato de distintas clases. Las responsabilidades de *DatatypePropertyAnalysis* son:

- Obtener las propiedades de tipo de dato de ambas clases. → *Class*
- Obtener valores de similitud aplicando las distintas funciones de similitud. → *SemanticFunctions*, *SyntacticFunctions*
- Crear correspondencias candidatas. → *PropertyMapping*
- Mostrar todas las correspondencias temporales al usuario para solicitar confirmación. → *ConfirmMappingForm*

La clase *SpecialPropertyAnalysis* es la responsable de la comparación desde la perspectiva de las propiedades especiales, es decir participa en la comparación de dos clases haciéndose responsable de la comparación de las propiedades especiales. La comparación es similar a la descripta para la clase *DatatypePropertyAnalysis*, salvo que la función de compatibilidad *Data Type* no es utilizada. Se agrega en cambio, el valor obtenido de la función que verifica las compatibilidades entre las

restricciones que pueden tener las propiedades especiales. Para hallar el valor de similitud final, se tiene en cuenta el valor de similitud de los rangos de las propiedades, y como el rango de las propiedades especiales son a su vez clases, se solicita a la clase *Similarity_Searcher* el valor de similitud de dichas clases que conforman los rangos. Por esto es un método recursivo, que finalizará cuando las clases que integran el rango sean, por lo menos una, del tipo *AtributteClass*. Después de comparar todas las propiedades especiales, se mostrarán al usuario las correspondencias candidatas para que confirme los resultados obtenidos. Esta clase, desempeña el rol de un proveedor de servicios atendiendo los requerimientos de la clase *Similarity_Searcher*: buscar correspondencias entre propiedades especiales de distintas clases. Las responsabilidades de *SpecialPropertyAnalysis* son:

- Obtener las propiedades especiales de ambas clases. → *Class*.
- Obtener valores de similitud aplicando las distintas funciones de similitud. → *SemanticFunctions*, *SyntacticFunctions*
- Crear correspondencias candidatas. → *PropertyMapping*
- Mostrar todas las correspondencias temporales al usuario para solicitar confirmación. → *ConfirmMappingForm*

La clase *ClassAnalysis* es responsable de la comparación desde la perspectiva de clases, es decir participa en la comparación de dos clases haciéndose responsable de la comparación de las clases mismas. La comparación aquí utiliza las funciones sintácticas para los dos tipos de clases y la función semántica para las clase del tipo *CommonClass*. La función semántica utiliza las correspondencias de propiedades ya confirmadas por el usuario. Luego de calcular el valor de similitud final para las clases, si dicho valor excede el umbral definido se agrega una correspondencia candidata. Se muestran luego al usuario las correspondencias candidatas para que confirme los resultados obtenidos. Las responsabilidades de *ClassAnalysis* son:

- Obtener valores de similitud para ambas clases aplicando las distintas funciones de similitud. → *SemanticFunctions*, *SyntacticFunctions*
- Crear una correspondencia candidata. → *ClassMapping*
- Mostrar las correspondencias candidatas al usuario para solicitar confirmación. → *ConfirmMappingForm*

4.4. Modelo de colaboraciones

Las colaboraciones son solicitudes desde un objeto a otro. Un objeto colabora con otro cuando necesita un servicio. Las responsabilidades de una aplicación son alcanzadas por grupos de objetos que trabajan en conjunto. En esta sección describimos las clases y responsabilidades que muestran cómo interactúan los objetos para lograr sus responsabilidades, es decir describimos el modelo de colaboraciones.

La arquitectura del plugin (Sección 4.1) establece ciertos patrones de comunicación que tienen implicancia en el diseño de las interacciones entre objetos y formas de comunicación, es decir patrones de colaboración. Tendremos entonces en cuenta, la posición que ocupa cada objeto en la arquitectura.

4.4.1. Inicialización del plugin

En primer lugar, recordemos que el plugin tiene dos responsabilidades principales que son la carga de las ontologías y la busqueda de similitudes entre ambas. Por lo tanto, los primeros pasos a realizar son solicitar al usuario las ontologías y proceder a su carga. Luego, el usuario indica la primer correspondencia entre dos conceptos de las distintas ontologías. A partir de aquí se da inicio al proceso de busqueda de similitudes. Para lograr su tarea el objeto *OWLSimTab* colaborará con otros objetos.

Como se muestra en la Figura 4.7, la solicitud de las ontologías en lenguaje OWL se realizará con la colaboración de *SingleFileChooseForm*, el cual permite indicar cuales son los archivos que contienen las ontologías. Se encarga de crear e inicializar el objeto de la clase *JWNL* para acceder al recurso lingüístico *WordNet*. Luego, la creación de los objetos que conforman la instanciación del modelo de la ontología es delegada por completo al objeto *OntologyLoader*. Finalmente, colabora con el objeto *OWLSimMappingProcess* para dar inicio al proceso de busqueda de similitudes. A través de este objeto colabora con el objeto de la clase *MappingLayoutForm* para mostrar las clases de las distintas ontologías a comparar y para mostrar los resultados del proceso de análisis.

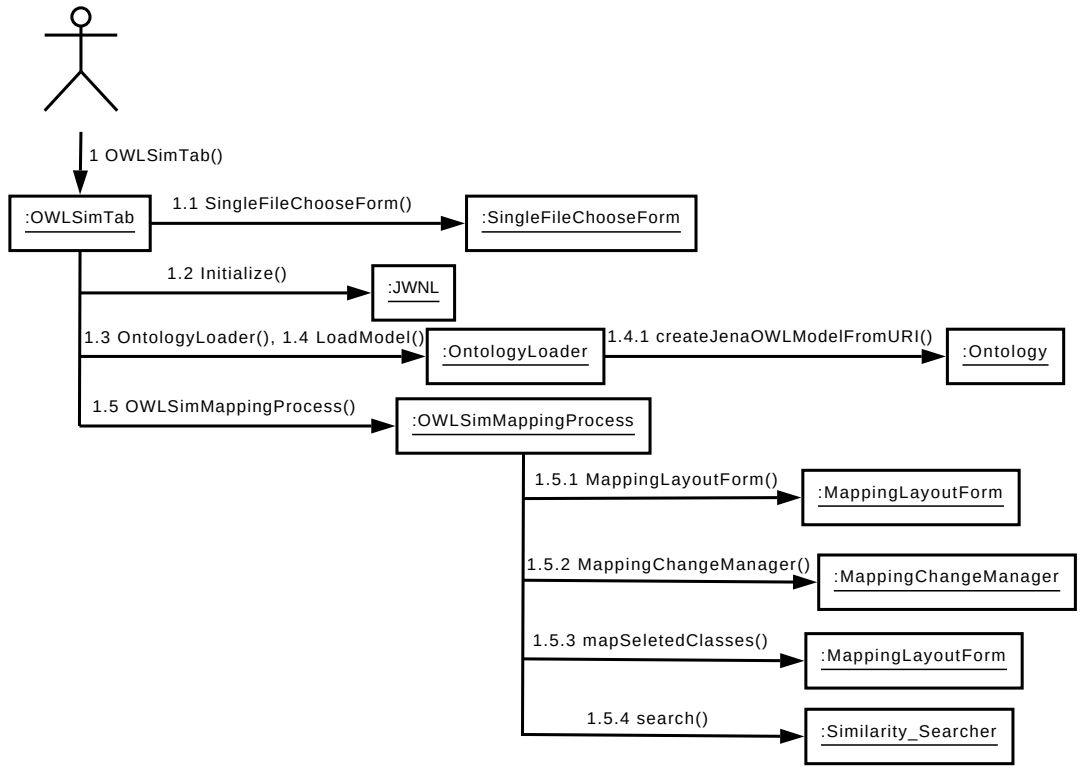

Figura 4.7: Diagrama de colaboraciones de la inicialización del plugin.

Este último formulario, *MappingLayoutForm*, colabora con el objeto *Ontology* para obtener las clases contenidas en cada ontología. Cabe aclarar aquí, que nuestra arquitectura no permite colaboración entre objetos de los componentes *Interfase de Usuario* y *Modelo del Dominio*. Sin embargo, como utilizamos clases de la *interfase de usuario* de Protégé, por su diseño éstas sí requieren de los objetos instanciación de la ontología. Siguiendo, una vez que se obtienen las clases, el formulario informará cuándo el usuario inicia el proceso de búsqueda de similitudes; esto indica una colaboración con el objeto *Similarity_Searcher*.

4.4.2. Búsqueda de similitudes

En la sección anterior vimos que la responsabilidad de búsqueda de similitudes en la clase *Similarity_Searcher* era extensa, por lo tanto se profundizó en su diseño y se particionaron las responsabilidades en distintos objetos colaboradores. Su responsabilidad se transformo en responsabilidades subordinadas, y su nueva responsabilidad es la de coordinar estar colaboraciones.

Distribuimos la responsabilidad de búsqueda de similitudes entre objetos especializados de búsqueda, cada uno responsable de la búsqueda por tipo de elemento a comparar (propiedad tipo de dato, propiedad especial y clase). Estos objetos colaboradores son coordinados por el objeto *Similarity_Searcher*, Figura 4.8. Si las clases a comparar son del tipo *CommonClass*, se comparan primero las propiedades de tipo de dato de ambas clases, y luego se comparan las propiedades especiales. Por último, se comparan las clases.

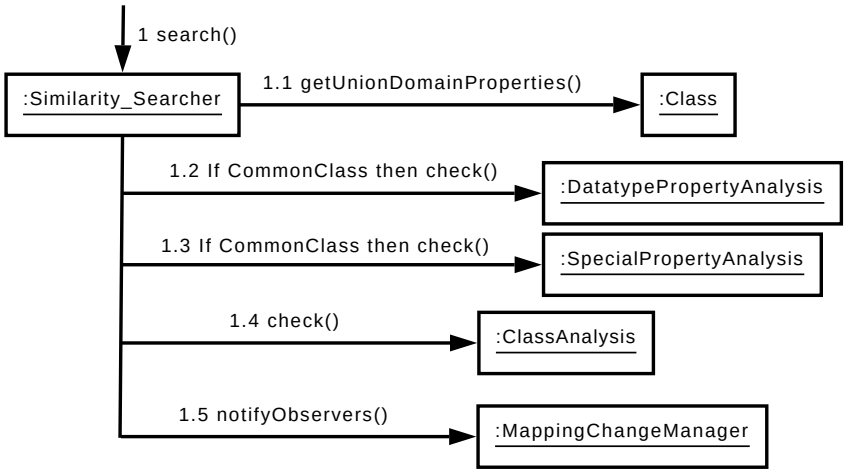

Figura 4.8: Diagrama de colaboraciones búsqueda de similitudes.

Cuando se encuentran nuevas correspondencias entre clases, se actualizan las vistas de las mismas. Para esto, el objeto *Similarity_Searcher* colabora con el objeto de la clase *MappingChangeManager* que forma parte del patrón de colaboraciones *observer*, descripto en la sección anterior. Este último objeto, a su vez, colabora con el objeto de la clase *ClassMapping*, que actúa como sujeto dentro del modelo del patrón, y los objetos de las clases *ClassMapResultsPane* y *SourcesPane*, que actúan como observadores. De estas clases observadoras, la primera muestra los resultados obtenidos, y la última indica las clases ya analizadas.

Los objetos especializados a quienes *SimilaritySearcher* delega las responsabilidades y coordina son: *DatatypePropertyAnalysis*, *SpecialPropertyAnalysis*, *ClassAnalysis*.

Para la búsqueda de similitudes entre propiedades tipo de dato, (Figura 4.9), el objeto *DatatypePropertyAnalysis* colabora con el objeto *CommonClass* para reunir la información que necesita, es decir las propiedades tipo de dato. En la comparación de las propiedades necesitará del servicio *Thesauruses* para la búsqueda de sinónimos, provisto por el objeto de la clase *SemanticFunctions*. Luego se solicita colaboración al objeto *SintacticFunctions* para la obtención de distintos valores resultantes de aplicar las funciones sintácticas que provee como servicios este objeto. Por último, si el valor calculado excede el umbral definido, se agrega una correspondencia candidata. Esto indica una colaboración con el objeto *PropertyMapping* el cual tiene la responsabilidad de dar de alta una nueva correspondencia. Las correspondencias encontradas se muestran al usuario para su confirmación. Esto requiere colaboración con el objeto *ConfirmMappingForm* quien será responsable de mostrar al usuario la correspondencia encontrada, y luego, saber y comunicar de la elección del usuario a la transacción. Luego dependiendo de la confirmación se debe actualizar el estado de la correspondencia consultada. Para esto se colabora nuevamente con la clase *PropertyMapping* pero esta vez para efectuar el cambio de estado.

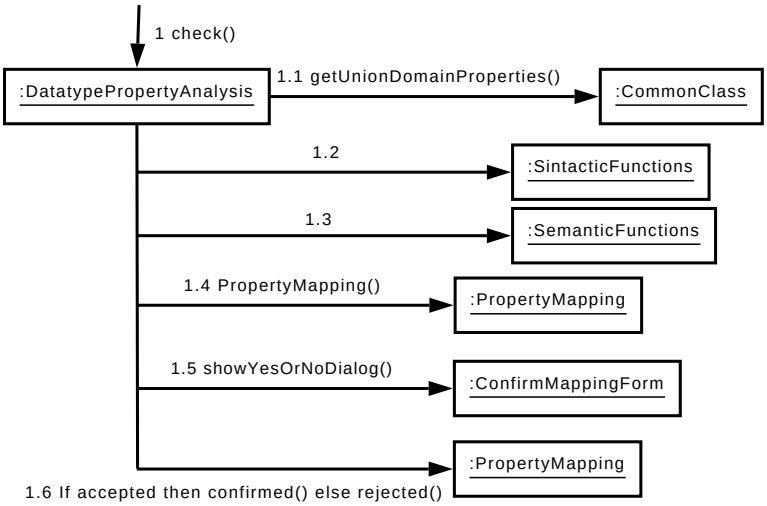

Figura 4.9: Diagrama de colaboraciones búsqueda de similitudes entre propiedades tipo de dato.

La comparación de las propiedades especiales (figura 4.10) es similar al caso anterior, pero no se calcula la compatibilidad de tipo de dato. En su lugar, para la comparación de los rangos, se inicia el proceso de búsqueda lo cual indica una recursión que finalizará cuando las clases de los rangos sean clases atributo, es decir *AtributteClass*. Para verificar las restricciones de las propiedades especiales, se colabora con el proveedor del servicio en este caso el objeto *SemanticFunctions*.

Finalmente para la comparación de las clases (Figura 4.11), tanto para clases atributo como clases comunes, se necesita la ayuda del objeto *SintacticFunctions* con sus responsabilidades o servicios de funciones de análisis sintáctico. También

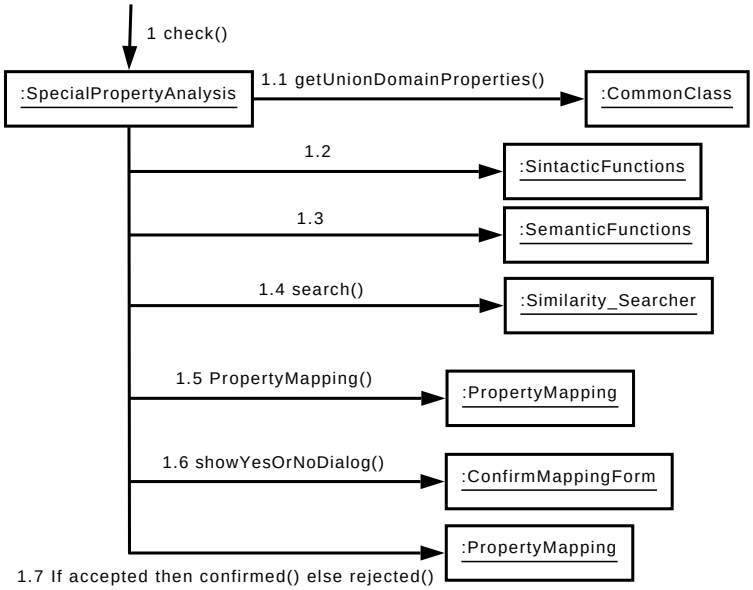

Figura 4.10: Diagrama de colaboraciones búsqueda de similitudes entre propiedades especiales.

se colabora con el objeto *SemanticFunctions* para la búsqueda de sinónimos, y en el caso de clases comunes para la función de comparación semántica basada en atributos (función (3.5) de la Sección 3.3.1). Se colabora con el objeto *ClassMapping* cuando se ha encontrado una correspondencia, y con el objeto *ConfirmMappingForm* para solicitar confirmación al usuario.

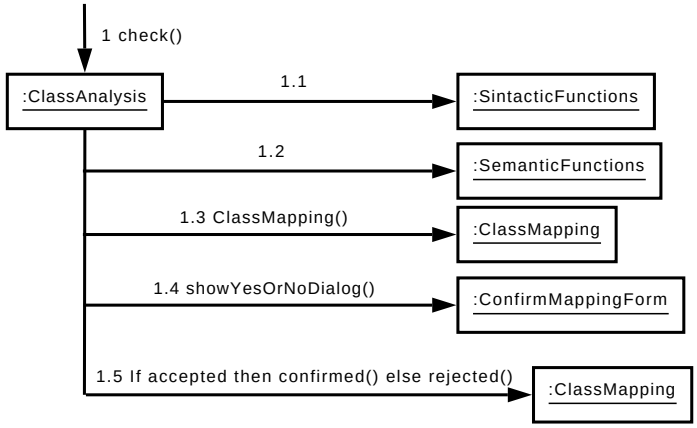

Figura 4.11: Diagrama de colaboraciones búsqueda de similitudes entre clases.

4.5. Resumen

En este capítulo se mostraron los roles o participantes del diseño, la asignación de responsabilidades que distribuye el trabajo y las colaboraciones que conectan los objetos que realizan los trabajos. A partir de aquí comenzamos con la implementación de la herramienta como un plugin de Protége mediante la traducción

de nuestro diseño.

Capítulo 5

Casos de estudio

En este capítulo mostramos el funcionamiento del plugin que implementa el método de búsqueda de similitudes extendido a través de distintos casos de estudio. Los casos de estudio señalan distintos aspectos del modelado de las ontologías y los resultados que el método genera para éstos. En todos los casos, los pares de ontologías elegidas cubren dominios que se solapan comparando así las partes en donde ambas tienen puntos en común.

Antes de comenzar con los casos de estudio, es importante mencionar dos aspectos que se tuvieron en cuenta en todos ellos. Primero, a pesar de que el método muestra al usuario sólo las correspondencias encontradas que sobrepasaron un umbral, a fines descriptivos, el plugin genera un "registro" con los todos los valores hallados, incluyendo resultados intermedios de funciones en los distintos niveles de analisis. Segundo, el umbral que hemos definido tiene un valor de 0,45, y por lo tanto, debemos tener en cuenta que la herramienta sólo mostrará aquellas correspondencias que resultaron en un valor mas alto a este.

El valor del umbral de 0,45 fue calculado empíricamente en base a las pruebas del sistema con ontologías reales y a las funciones del método. Este valor resultó el conveniente para encontrar similitudes cuando existen conceptos representados en forma diferente, pero nombrados con la misma terminología. Obviamente, si dos conceptos son representados en forma muy diferente, el método no encontrará similitud. Ya veremos este tipo de casos y otros en este capítulo.

5.1. Caso de estudio 1

Cómo primer caso de estudio seleccionamos un caso sencillo para describir el funcionamiento del método al comparar dos clases, concentrándonos en el análisis de propiedades tipo de dato. También queremos ilustrar el comportamiento del método cuando dos ontologías de un mismo dominio modelan lo mismo de manera muy diferente. Analizaremos las ontologías *Travel* y *Airport* que presentamos en la Sección 3.4; recordemos la definición de ambas:

- La ontología denominada *"Travel Ontolgy"*[1] modela vuelos, agencias de viajes, alquiler de autos, hoteles, etc. y posee alrededor de 40 clases junto con

[1] `www.ilby.net/travel.owl`

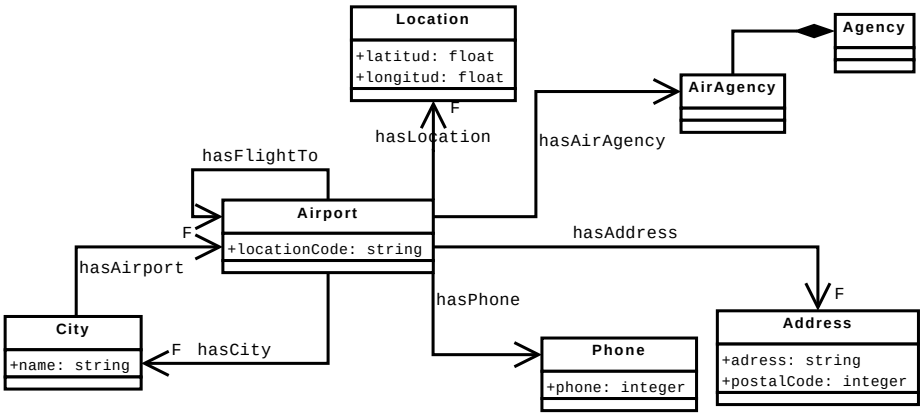

Figura 5.1: Parte de la ontología *"Travel"*

gran cantidad de propiedades para describirlas, ya sea propiedades de tipo de dato o especiales.

- La ontología *"Airport Ontology"*[2] es muy sencilla y sólo modela la clase *Airport*.

La Figura 5.1 muestra gráficamente la parte de la ontología *"Travel"* que modela la clase *Airport* (Aeropuerto) junto con las otras clases con las cuales esta relacionada. Como vemos, la clase *Airport* posee una propiedad tipo de dato, *locationCode* que es de tipo *String*, y seis propiedades especiales, *hasPhone*, *hasAirAgency*, *hasAddress*, *hasCity*, *hasLocation*, y *hasFlightTo* que se relacionan con las clases *Phone*, *AirAgency*, *Address*, *City*, *Location* y *Airport* respectivamente. Las cuatro últimas clases son funcionales y se denotan con la letra *"F"* en el diagrama. Recordemos que una propiedad funcional significa que un elemento del dominio se relaciona con un único del rango. Por ejemplo, un aeropuerto tendrá una única dirección para el caso de la propiedad *hasAddress*. Luego la clase *Location* posee dos propiedades tipo de dato, *latitude* y *longitude*, donde ambas se relacionan con el tipo de dato *float*. La clase *City* posee una propiedad tipo de dato, *name* que es *String*, y una propiedad especial, *hasAirport*, que se relaciona con la clase *Airport* y es funcional. Las clases *Phone* y *Address* poseen sólo propiedades tipo de dato. Para el primer caso, sólo la propiedad *phone* es representada y se relaciona con el tipo *integer*. En el segundo caso, *Address*, se representan las propiedades, *address* y *postalCode* de tipos *String* e *integer* respectivamente. La clase *AirAgency* no posee ninguna propiedad pero es una subclase de *Agency*. En el gráfico esto esta representado por un línea con un rombo en la punta.

La Figura 5.2 muestra gráficamente la segunda ontología. Aquí sólo se representa la clase *Airport* con siete propiedades tipo de dato.

Como podemos ver, ambas ontologías representan a la clase *Airport* en forma muy diferente. Nuestra herramienta, que implementa el método descripto en **??**, realiza un análisis estructural pero sólo la parte de comparaciones a nivel léxico del término "Airport" asociado a la clase *Airport* (tesauros en el nivel semántico y comparación de cadenas en el nivel sintáctico) genera un valor de similitud alto

[2] `www.daml.org/2001/10/html/airport-ont`

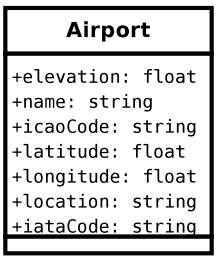

Figura 5.2: La ontología *"Airport"*

entre los conceptos de ambas ontologías. Por el contrario, la comparación de las propiedades retornará valores bajos. Nótese que la clase *Airport* de la segunda ontología no posee ninguna propiedad especial y esta descripta sólo con propiedades tipo de dato. En cambio la primera ontología posee una propiedad tipo de dato y seis propiedades especiales. Recordemos que las propiedades y clases se comparan según su tipo, es decir, no se compararán propiedades tipo de dato de una ontología con propiedades especiales de otra.

El formulario principal del plugin para la selección de las ontologías a analizar es el que podemos ver en la Figura 5.3. Si alguna de las ontologías seleccionadas no pertenece al lenguaje OWL-Lite no se cargan las mismas y se muestra un mensaje como en la Figura 5.4.

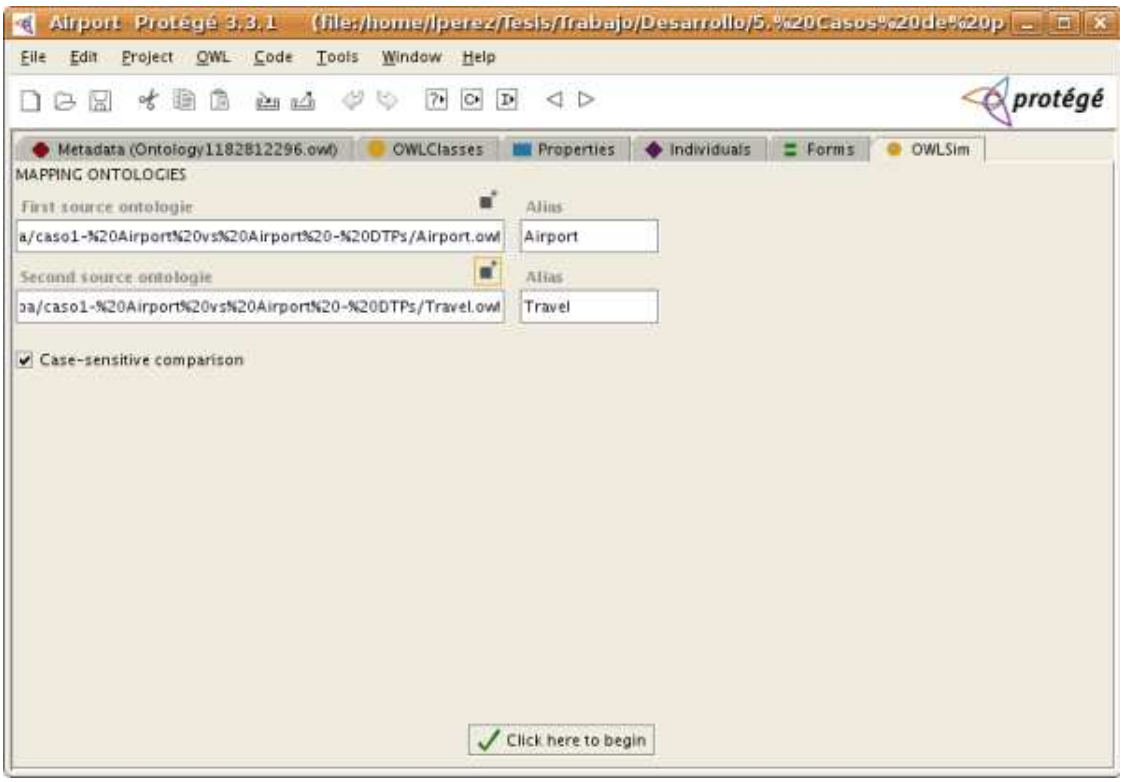

Figura 5.3: Formulario de selección de ontologías del plugin.

Los resultados obtenidos se muestran en la Figura 5.5. Según nuestro umbral, el plugin sólo encuentra y sugiere correspondencias candidatas en tres de las comparaciones realizadas entre las propiedades tipo de dato. Igualmente el plugin

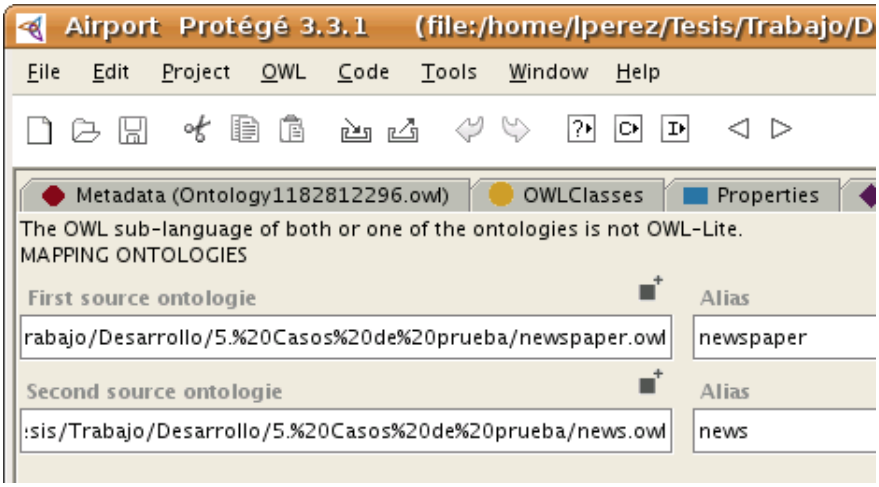

Figura 5.4: Cuando alguna o ambas de las ontologías seleccionadas no pertenece al sublenguaje OWL-Lite.

imprime todos los resultados en la consola para mostrar los valores obtenidos y su funcionamiento en todos los casos.

```
1. Visited node [Airport vs Airport ]
dtp) sim value: iataCode-locationCode==>0.45256412
                  (_simEd: 0.33333334,_simTri:0.07692308,_simDtc:1.0,_simThesaurus:0.5)
dtp) sim value: location-locationCode==>0.6354167
                  (_simEd: 0.375,_simTri:0.16666667,_simDtc:1.0,_simThesaurus:1.0)
dtp) sim value: latitude-locationCode==>0.046875
                  (_simEd: 0.125,_simTri:0.0625,_simDtc:0.0,_simThesaurus:0.0)
dtp) sim value: longitude-locationCode==>0.040935673
                  (_simEd: 0.11111111,_simTri:0.05263158,_simDtc:0.0,_simThesaurus:0.0)
dtp) sim value: elevation-locationCode==>0.047008548
                  (_simEd: 0.11111111,_simTri:0.07692308,_simDtc:0.0,_simThesaurus:0.0)
dtp) sim value: name-locationCode==>0.21785715
                  (_simEd: 0.0,_simTri:0.071428575,_simDtc:1.0,_simThesaurus:0.0)
dtp) sim value: icaoCode-locationCode==>0.48034188
                  (_simEd: 0.44444445,_simTri:0.07692308,_simDtc:1.0,_simThesaurus:0.5)
_simThesaurus: 1.0
_simSint: 1.0
_simAttDatatypeProperties: 0.0875
_simAttSpecialProperties: 0.0
ca-common) sim value: Airport-Airport===>0.38750002
```

Figura 5.5: Valores de similitud obtenidos en la comparación de las clases *Airport* de ambas ontologías.

Primero, la herramienta compara la propiedad tipo de dato *locationCode* de la clase *Airport* de la primera ontología con todas las propiedades tipo de dato de la clase *Airport* de la segunda. Esto se debe a que la clase *Airport* posee una sola propiedad tipo de dato y por lo tanto el sistema debe compararla con todas las propiedades del mismo tipo en la otra ontología.

De todas las comparaciones entre propiedades tipo de dato, el usuario debería confirmar *location-locationCode* que son las propiedades que se solapan de ambas ontologías, y para las que por consiguiente, el método encuentra el valor mas alto:

```
dtp) sim value: location-locationCode==>0.6354167
```

Este valor se debe a que, como podemos ver en la Figura 5.5 los valores de similitud en todas las funciones (`_simEd`, `_simTri`, `_simDtc` y `_simThesaurus`) son los mas altos. Esto coincide con nuestra comparación intuitiva ya que las dos propiedades tienen una superposición importante en su significado.

Para el caso de:

```
dtp) sim value: iataCode-locationCode==>0.4775641 y
dtp) sim value: icaoCode-locationCode==>0.5053419
```

las palabras o términos asociados a las propiedades especiales tienen en común la subcadena "Code". Por ello, aunque las subcadenas "iata" e "icao" no son sinónimos, el resultado de la función de búsqueda en Tesauro (`_simThesaurus`) igualmente devuelve un valor distinto de cero. También, podemos observar, que el resultado de la función *edit distance* (`_simEd`) es relativamente alto. Esto se debe a que tanto la subcadena de caracteres "iata" como "icao" no están tan lejos de ser transformadas a la cadena "location". Por último, estas propiedades poseen el mismo tipo de dato *Float* como rango.

Luego, según el método se comparan las clases mismas, teniendo en cuenta la correspondencia confirmada por el usuario. El valor retornando es el siguiente:

```
ca-common) sim value: Airport-Airport===>0.38750002
```

Como vemos, el valor de similitud final es muy bajo, si consideramos que ambas clases son similares e intentan representar las mismas entidades en ambas ontologías. El valor `0.38750002` se obtuvo principalmente del nivel sintáctico, ya que son sintácticamente iguales; y de la búsqueda en el *Tesauro*, ya que son sinónimos. Pero las funciones de comparación estructural de propiedades (`_simAttDatatypeProperties` y `_simAttSpecialProperties`) resultan en un valor bajo.

Así, por mas de que en algunos niveles el resultado fue positivo, en otros fue muy bajo y produjo un decremento del valor final, que como vemos no supera el umbral. Según la clasificación de incompatibilidades de *heterogeneidad ontológica* de Visser en [59], podríamos decir que este es un caso de *incompatibilidad de conceptualización*, en particular, incompatibilidad de relación en cuanto a la asignación de atributos a clases. Las dos conceptualizaciones definen las mismas clases pero difieren en la forma de representación de las mismas.

5.2. Caso de estudio 2

En este caso de estudio, pretendemos mostrar el funcionamiento del plugin cuando se analizan ontologías cuyas definiciones causan ciclos en el recorrido. Para ésto tomamos parte de la ontología *Location* y la parte de la ontología *Travel* que presentamos en la Sección 3.4. Suponemos que el usuario indica la clase *City* como la primer correspondencia de ambas ontologías .

En la ontología *Location* (Figura 5.6), la clase *Continent* no posee propiedades especiales. La clase *Country* posee dos propiedades especiales: *is_part_of* con la clase *Continent* de rango, y la propiedad *has_parts* que tienen a las clases *State* y *City* en el rango. Luego la clase *City* tiene la propiedad *is_part_of* que

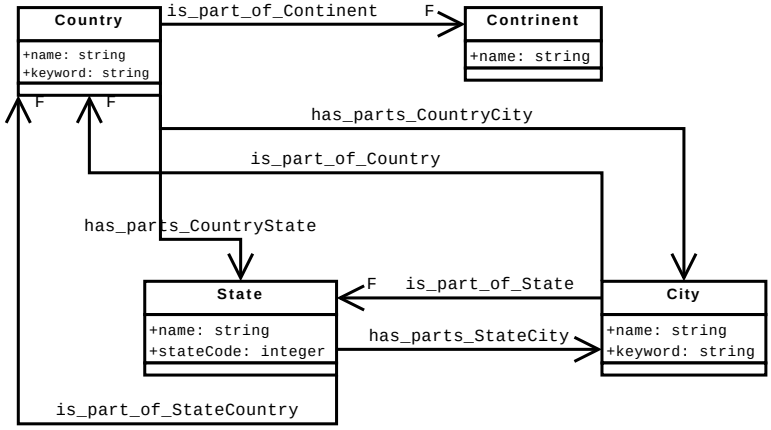

Figura 5.6: La ontología *"Location"*

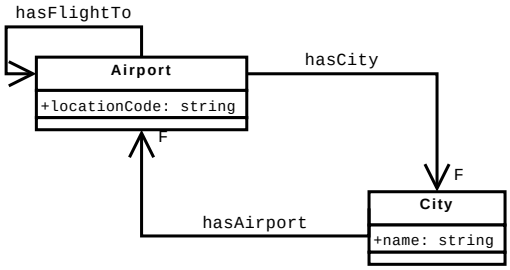

Figura 5.7: Parte de la ontología *"Travel"*

la relaciona con las clases *Country* y *State*. Por ultimo, la clase *State* posee la propiedad *is_part_of* cuyo rango es la clase *Country* y la propiedad *has_parts* con la clase rango *City*.

En la ontología *Travel* (Figura 5.7), la clase *Airport* tiene dos propiedades especiales: la propiedad *hasFlightTo* que tiene como clase rango a *Airport* y la propiedad *hasCity* que tiene como clase rango a *City*. Esta última, a su vez, tiene una propiedad especial que es la propiedad *hasAirport* con la clase *Airport* como rango.

Primero, se recuperan las propiedades tipo de dato de ambas y se comparan los términos asociados a las mismas, a nivel sintáctico y semántico, además de los tipos de datos de los rangos. Como podemos ver en el Figura 5.8 se comparan las propiedades tipo de dato de la clase *Location.City*, *cityName* y *cityKeyWords*, y de la de la clase *Travel.City*, *name*. El método sólo sugiere al usuario la correspondencia candidata *Location.City.cityName-Travel.City.name* ya que es la única que supera el umbral. La otra correspondencia analizada obtiene un valor muy bajo pues las propiedades son muy diferentes, tanto a nivel sintáctico como semántico.

```
1. Visited node [Location.City vs Travel.City ]
dtp) sim value: Location.City.cityName-Travel.City.name==>0.53125
         (_simEd: 0.0,_simTri:0.125,_simDtc:1.0,_simThesaurus:1.0)
dtp) sim value: Location.City.cityKeyWords-Travel.City.name==>0.21666667
         (_simEd: 0.0,_simTri:0.06666667,_simDtc:1.0,_simThesaurus:0.0)
```

Figura 5.8: Resultados del método para la comparación de las clases *City*.

En segundo lugar, el método procede con el análisis de las correspondencias entre propiedades especiales, continuando con las mismas clases, *Location.City*, con las propiedades *is_part_of_State* y *is_part_of_Country*, y *Travel.City* con la propiedad especial *has_Airport*.

```
2. Visited node [Location.State vs Travel.Airport ]
dtp) sim value: Location.State.SateCode-Travel.Airport.locationCode==>0.2525641
        (_simEd: 0.33333334,_simTri:0.07692308,_simDtc:0.0,_simThesaurus:0.5)
dtp) sim value: Location.State.stateName-Travel.Airport.locationCode==>0.23750001
        (_simEd: 0.1,_simTri:0.05,_simDtc:1.0,_simThesaurus:0.0)

Go on with the next node from:Location.State.has_parts_StateCity - Travel.Airport.hasCity
3. Visited node [Location.City vs Travel.City ]
3. CYCLE IN: City - City
Permanent Datatype mappings (already calculated) for: City and City are: cityName and name
(_simThesaurus: 1.0, _simEd:1.0, _simTri:1.0)
_simLexico: 1.0
_simAttDatatypeProperties: 0.23333333
_simAttSpecialProperties: 0.0
ca-common) sim value: City-City===>0.53333336
_simRest: 0.0
spa) sim value: Location.State.has_parts_StateCity
-Travel.Airport.hasCity==>0.31527779
```

Figura 5.9: Resultados del método para la comparación de las clases *City*.

Para el primer par de propiedades que compara, *has_parts_StateCity* de *Location.State* y *hasCity* de *Travel.Airport*, se evalúan los términos a nivel sintáctico y semántico, y luego, la evaluación estructural, es decir, se analizan las clases rango. En la Figura 5.9, podemos ver que el punto 2 corresponde al análisis de las clases rangos obtenidas de este par de propiedades especiales. Comparar las clases rango, significa iniciar el método para este nuevo par de clases, *Location.State* y *Travel.Airport*. Siguiendo, se encuentran los resultados del análisis de las propiedades tipo de dato de estas clases, los cuales son valores bajos ya que son diferentes. El método continua con el análisis de las propiedades especiales de estas clases, comparando *Location.State.is_part_of_StateCountry* y *Location.State.has_parts_StateCity* de una ontología con *Travel.Airport.hasFlightTo* y *Travel.Airport.hasCity* de la otra.

El primer par elegido es *Location.State.has_parts_StateCity* y *Travel.Airport.hasCity*. Cuando se analizan sus clases rango (punto 3 en Figura 5.9) el método compara a la clase *Location.City* con la clase *Travel.City*. Nuevamente, comparar las clases rango, significa iniciar el método para este par de clases. Pero aquí, cuando el método detecta que estas clases ya fueron analizadas y todavía no fueron resueltas, detecta la presencia de un ciclo (`3. CYCLE IN: City - City`). Luego, como explicamos en el Capítulo 3, para estas clases se sigue el algoritmo de busqueda de similitudes con excepción del análisis de las propiedades especiales. Igualmente, el valor parcial devuelto es un valor alto ya que del análisis sintáctico y semántico de los términos asociados a las clases, se obtiene el valor máximo (`_simThesaurus: 1.0`, `_simEd:1.0`, `_simTri:1.0`). Aquí, en el formulario de confirmación de correspondencias, se aclara que el valor hallado es un valor parcial (ver Figura 5.10).

Esta correspondencia candidata se muestra al usuario para su confirmación ya que supera el umbral:

```
ca-common) sim value: City-City===>0.53333336.
```

Finalmente, se analizan las restricciones obteniendo un valor igual a cero ya que una de ellas es funcional y la otra no. Así, se completa el análisis y se obtienen todos los valores para las propiedades *Location.State.has_parts_StateCity* y *Travel.Airport.hasCity* hallando el valor final `0.31527779`. Éste valor no supera el umbral, por consiguiente no se consultará ni mostrará al usuario.

```
spa) sim value:
Location.State.has_parts_StateCity-Travel.Airport.hasCity=>0.31527779
```

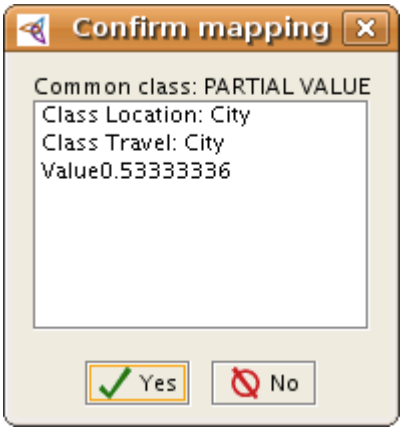

Figura 5.10: Solicitud de confirmación de correspondencia entre dos clases, en presencia de un ciclo.

Podemos observar también, algunos valores de comparaciones intermedias en el proceso de evaluación. Por ejemplo, a través del análisis de las clases rangos de las propiedades especiales se llega a analizar la similitud entre las clases *Location.Continent* y *Travel.City* (Figura 5.11). La clase *Location.Continent* sólo tiene una propiedad tipo de dato en la que si bien el término asociado a la propiedad es similar, y por ello el valor de similitud supera el umbral, el usuario debería rechazar esta correspondencia ya que las clases en comparación claramente no modelan el mismo concepto. Esto se ve reflejado en el valor de similitud `_simLexico` (análisis sintáctico y semántico de los términos) que es muy bajo.

```
145. Visited node [Location.Continent vs Travel.City ]
dtp) sim value: Location.Continent.continentName-Travel.City.name==>0.5192308
        (_simEd: 0.0,_simTri:0.07692308,_simDtc:1.0,_simThesaurus:1.0)
(_simThesaurus: 0.0, _simEd:0.0, _simTri:0.1)
_simLexico: 0.030000001
_simAttDatatypeProperties: 0.0
_simAttSpecialProperties: 0.0
ca-common) sim value: Continent-City===>0.009000001
```

Figura 5.11: Valor de similitud hallado en la comparación de la clase *Location.Continent* y *Travel.City*.

El método seguirá analizando todas las clases con sus propiedades especiales, armando el recorrido como grafo de las ontologías. Una vez hallados los valores de

similitud para las propiedades especiales de las clases *Location.City* y *Travel.City* (Figura 5.12) se calcula el valor de similitud final para las clases.

```
spa) sim value: Location.City.is_part_of_State-Travel.City.hasAirport==>0.13095239
spa) sim value: Location.City.is_part_of_Country-Travel.City.hasAirport==>0.13043478
```

Figura 5.12: Valores de similitud hallados para las propiedades especiales de las clases *Location.City* y *Travel.City*.

```
(_simThesaurus: 1.0, _simEd:1.0, _simTri:1.0)
_simLexico: 1.0
_simAttDatatypeProperties: 0.23333333
_simAttSpecialProperties: 0.0
ca-common) sim value: City-City===>0.53333336
```

Figura 5.13: Valor obtenido para las clases *Location.City* y *Travel.City*

El valor de similitud final hallado para estas clases (Figura 5.13) es el mismo que el valor parcial, ya que del análisis de sus propiedades especiales surge que no tienen propiedades en común. Así, los valores de similitud encontrados son muy bajos (ver Figura 5.12). Sin embargo, explicamos anteriormente los términos usados para nombrar las clases son similares dando así como valor de similitud entre las clases un valor alto.

Si bien las clases *City* de ambas ontologías modelan la misma entidad las ontologías en sí se solapan solo en esta clase. El modelado del resto de los conceptos en ambas ontologías son diferentes ya que representan dominios distintos.

5.3. Caso de estudio 3

En este caso hemos utilizado dos ontologías extraídas de un tutorial [3] de otra herramienta para la alineación de ontologías PROMPT [51].

La primera ontología llamada *"Air_ Reservation"* posee las clases y relaciones necesarias para efectuar reservaciones en vuelos. La Figura 5.14 muestra gráficamente esta ontología que posee doce clases y varias propiedades especiales y tipo de dato. Por ejemplo, la clase *Reservation_ record* (que es una especialización de la clase *Record*) posee los datos de reservación de los clientes con sus vuelos. Esta posee la propiedad tipo de dato *record_ location* de tipo *String* y dos propiedades especiales, *traveler* que se relaciona con la clase *Individual* e *itinerary* que se relaciona con la clase *Itinerary*. Esta última a su vez es funcional.

La clase *Customer* posee una propiedad tipo de dato que indica el nombre (propiedad *name*) y una propiedad especial *payment* que relaciona la misma con la clase *Payment_ record* y denota el registro de pago de un cliente. Como vemos esta propiedad no es funcional ya que un mismo cliente puede tener varios registros de pago. La clase *Individual* es una especialización de la clase *Customer* y se relaciona con la clase *Flight* la cual posee información relacionada con un vuelo. Esta última posee cinco propiedades tipo de dato que almacenan datos como el

[3] `http://protege.stanford.edu/plugins/prompt/prompt.html`

nombre de la aerolínea, número de vuelo, origen, destino. Además, se relaciona con la clase *Aircraft*, mediante la propiedad especial *aircraft* para indicar el/los aviones que se utilizan en el vuelo. La clase *Payment_ record*, que también es una especialización de la clase *Record*, posee información del pago. A su vez, posee dos propiedades tipo de dato que indican la cantidad a pagar y si pagó o no, y dos clases como especializaciones, *Check* y *Credit_ Card*. Luego, esta última posee dos propiedades tipo de dato que almacenan información de la tarjeta de crédito, el día de expiración y el número.

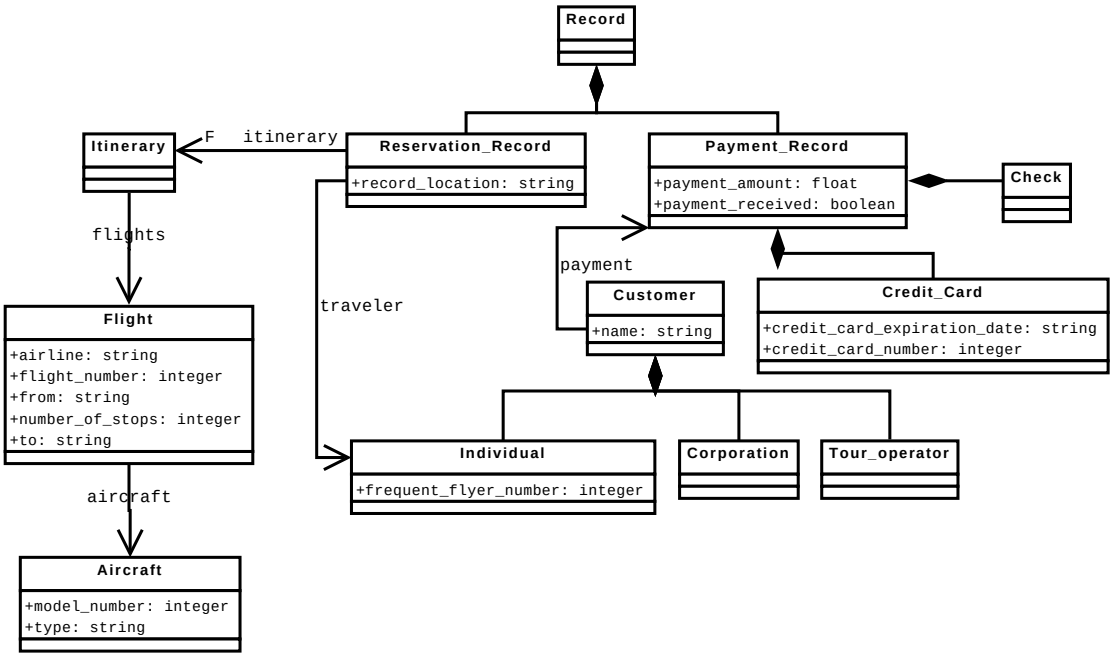

Figura 5.14: La ontología *"Air_ Reservation"*

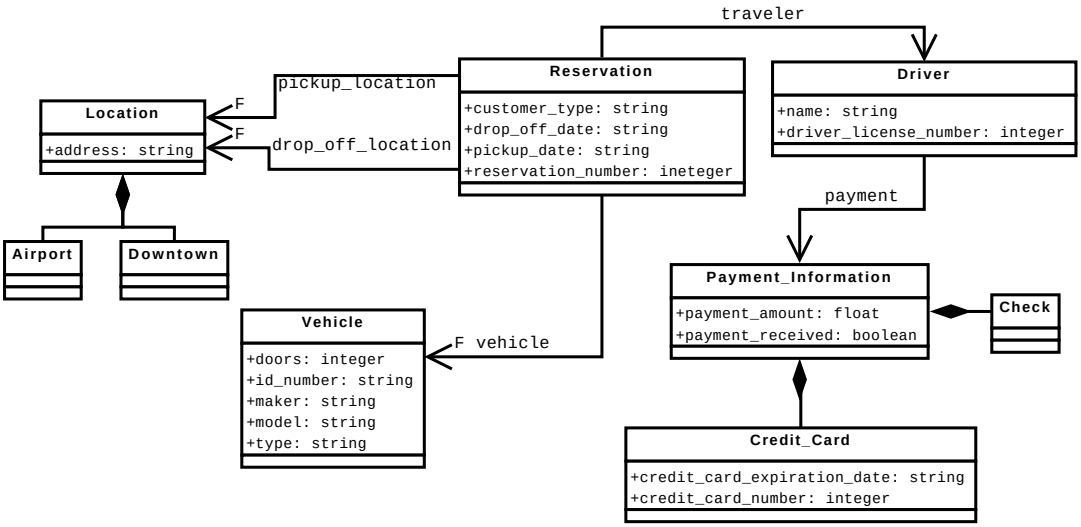

Figura 5.15: La ontología *"Car_ Rental"*

La segunda ontología, llamada *"Car_ Rental"* (Figura 5.15), posee la información necesaria para efectuar reservaciones de vehículos. Esta ontología posee ciertas

similitudes con la anterior, como por ejemplo, el registro de pago, aquí llamado *Payment_ Information*. Sin emabrgo, posee algunas representaciones diferentes como el registro de reservaciones ya que posee cuatro propiedades tipo de dato que guardan información acerca de la fecha de retiro del vehículo, el número de reservación, etc. Además, posee cuatro propiedades especiales, dos que se relacionan con la clase *Location* para indicar donde se retira y devuelve el vehículo, otra con la clase *Driver* para indicar el conductor del vehículo, y la última con el vehículo para indicar cuál es el reservado.

Haremos la primera comparación entre las clases *Payment_ record* de la primera ontología y *Payment_ Information* de la segunda. Como ambas clases poseen propiedades tipo de dato, el método las compara primero generando los resultados que muestra la Figura 5.16.

```
1.Visited node[Air_Reservation.Payment_record vs Car_Rental.Payment_Information]
dtp) sim value: Air_Reservation.Payment_record.payment_amount
-Car_Rental.Payment_Information.payment_received==>0.27380952
       (_simEd: 0.42857143,_simTri:0.06666667,_simDtc:0.0,_simThesaurus:0.5)
dtp) sim value: Air_Reservation.Payment_record.payment_amount
-Car_Rental.Payment_Information.payment_amount==>1.0
       (_simEd: 1.0,_simTri:1.0,_simDtc:1.0,_simThesaurus:1.0)
dtp) sim value: Air_Reservation.Payment_record.payment_received
-Car_Rental.Payment_Information.payment_received==>1.0
       (_simEd: 1.0,_simTri:1.0,_simDtc:1.0,_simThesaurus:1.0)
dtp) sim value: Air_Reservation.Payment_record.payment_received
-Car_Rental.Payment_Information.payment_amount==>0.27380952
       (_simEd: 0.42857143,_simTri:0.06666667,_simDtc:0.0,_simThesaurus:0.5)
(_simThesaurus: 0.5, _simEd:0.2857143, _simTri:0.05263158)
_simLexico: 0.30150378
_simAttDatatypeProperties: 0.35
_simAttSpecialProperties: 0.35
ca-common) sim value: Payment_record-Payment_Information===>0.79045117
```

Figura 5.16: Valores hallados en la comparación de las clases *Payment_ record* y *Payment_ Information*.

Como vemos aquí, el método encuentra como correspondencias candidatas sólo dos, las propiedades *payment_ amount* y las propiedades *payment_ received*, con un valor perfecto de 1. Por lo tanto, las seleccionamos como correspondencias definitivas.

Luego, el sistema realiza la comparación entre las clases comunes que seleccionamos al principio. No se efectúa ninguna comparación entre propiedades especiales ya que las clases no poseen este tipo de propiedades. Por lo tanto, el método retorna el valor `1`. Finalmente, por las ponderaciones o pesos aplicados al resultado de esta comparación, se obtiene el valor `_simAttSpecialProperties: 0.35`. Así, la única correspondencia que el método muestra es:

```
ca-common) sim value: Payment_record-Payment_Information===>0.79045117
```

Este valor es alto (`0.79045117`) ya que aunque sintácticamente no sean exactamente iguales, el nivel semántico da perfecto porque poseen dos propiedades tipo de dato, seleccionadas como similares, y no poseen ninguna propiedad especial. Por lo tanto estructuralmente son muy parecidas.

Continuando con la comparación de otras clases, por ejemplo *Credit_ Card* de ambas ontologías, obtenemos los resultados de la Figura 5.17. Aquí, sólo las

correspondencias entre las propiedades tipo de dato de ambas clases que tienen el mismo nombre *credit_ card_ expiration_ date* y *credit_ card_ number* son las que sugiere el sistema. Luego el valor final para la comparación de las clases es 1.

```
2.  Visited node [Air_Reservation.Credit_Card vs Car_Rental.Credit_Card ]
dtp) sim value: Air_Reservation.Credit_Card.credit_card_expiration_date
-Car_Rental.Credit_Card.credit_card_number==>0.30691922
        (_simEd: 0.22222222,_simTri:0.045454547,_simDtc:0.6,_simThesaurus:0.4)
dtp) sim value: Air_Reservation.Credit_Card.credit_card_expiration_date
-Car_Rental.Credit_Card.credit_card_expiration_date==>0.96000004
        (_simEd: 1.0,_simTri:1.0,_simDtc:0.8,_simThesaurus:1.0)
dtp) sim value: Air_Reservation.Credit_Card.credit_card_number
-Car_Rental.Credit_Card.credit_card_number==>1.0
        (_simEd: 1.0,_simTri:1.0,_simDtc:1.0,_simThesaurus:1.0)
dtp) sim value: Air_Reservation.Credit_Card.credit_card_number
-Car_Rental.Credit_Card.credit_card_expiration_date==>0.1869192
        (_simEd: 0.22222222,_simTri:0.045454547,_simDtc:0.0,_simThesaurus:0.4)
(_simThesaurus: 1.0, _simEd:1.0, _simTri:1.0)
_simLexico: 1.0
_simAttDatatypeProperties: 0.35
_simAttSpecialProperties: 0.35
ca-common) sim value: Credit_Card-Credit_Card===>1.0
```

Figura 5.17: Resultados comparación clases *Credit_ Card* de las ontologías *Car_ Rental* y *Air_ Reservation*.

```
3. Visited node [Air_Reservation.Check vs Car_Rental.Check ]
(_simThesaurus: 1.0, _simEd:1.0, _simTri:1.0)
ca-atributte) sim value: Check-Check===>1.0
```

Figura 5.18: Resultados comparación clases *Check*.

Para la clase *Check* como es una clase atributo, es decir no posee propiedades de ninguna clase, el método sólo aplica la comparación a nivel del término asociado a la clase. Dado que es exactamente el mismo, "Check", el resultado que se obtiene es igual a *1* (Figura 5.18).

Luego, en la comparación de las clases *Air_ Reservation.Customer* y *Car_ Rental.Driver* el método primero analiza las propiedades tipo de dato (Figura 5.19). La correspondencia que el método sugiere es de la propiedad *name* de ambas clases, ya que son idénticas sintáctica y semánticamente. Entre *name* y *driver_ licence_ number* a pesar de tener una cierta compatibilidad entre sus tipos de dato, no es suficiente para que el sistema la sugiera, ya que tanto la búsqueda en el tesauro como el análisis de las cadenas de caracteres son prácticamente nulos.

```
4. Visited node [Air_Reservation.Customer vs Car_Rental.Driver ]
dtp) sim value: Air_Reservation.Customer.name-Car_Rental.Driver.name==>1.0
                (_simEd: 1.0,_simTri:1.0,_simDtc:1.0,_simThesaurus:1.0)
dtp) sim value:
Air_Reservation.Customer.name-Car_Rental.Driver.driver_licence_number==>0.1313636
                (_simEd: 0.0,_simTri:0.045454547,_simDtc:0.6,_simThesaurus:0.0)
```

Figura 5.19: Resultado de propiedades tipo de dato para *Air_ Reservation.Customer* y *Car_ Rental.Driver*.

Siguiendo, al analizar las propiedades especiales de ambas clases *Air_ Reservation.Customer.payment* y *Car_ Rental.Driver.payment*, el método analiza los rangos, es decir compara las clases *Air_ Reservation.Payment_ record* y *Car_ Rental.Payment_ Information*. Sin embargo, estas clases fueron previamente analizadas y confirmadas, ya que fueron el primer par de clases que comparamos. Así, podemos ver en la Figura 5.20 que no se volvió a calcular la similitud de estas clases. El método, después de recuperar las correspondencias almacenadas, calculó directamente el valor de similitud entre las propiedades especiales *payment*.

```
Go on with the next node from:
   Air_Reservation.Customer.payment - Car_Rental.Driver.payment
spa) sim value:
   Air_Reservation.Customer.payment-Car_Rental.Driver.payment==>0.7702256
(_simThesaurus: 0.0, _simEd:0.0, _simTri:0.09090909)
_simLexico: 0.02727273
_simAttDatatypeProperties: 0.23333333
_simAttSpecialProperties: 0.35
ca-common) sim value: Customer-Driver===>0.5915151
```

Figura 5.20: Resultados hallados para las clases *Air_ Reservation.Customer* y *Car_ Rental.Driver*.

Finalmente, luego de encontrar los valores de similitud para las propiedades tipo de dato y las propiedades especiales, se comparan las clases comunes (*Customer* y *Driver*). Podemos observar que el valor de similitud encontrado entre las clases supera el umbral (`0.5915151`), pero sin embargo es sólo un valor medio. Esto se debe a que, si bien conceptualmente poseen una representación similar, los términos asociados a ambas clases *Customer* y *Driver* son muy distintos. Por ello, el valor del análisis sintáctico y semántico (*_simLexico*) es bajo, ya que por ejemplo en el Tesauro no se pudieron encontrar como sinónimos.

Por último, en las Figuras 5.21 y 5.22 vemos cómo el sistema presenta las correspondencias que quedaron como definitivas. La Figura 5.21 muestra las correspondencias entre las clases de ambas ontologías junto con el valor de similitud obtenido y la decisión del usuario para cada correspondencia (aceptación ó rechazo); en este caso vemos que todas son permanentes, es decir, todas fueron aceptadas por el usuario. Por otro lado, la Figura 5.22 muestra la misma información pero para las propiedades tipo de dato y especiales.

5.4. Resumen

En este capítulo, presentamos distintos casos de estudio con el objetivo de mostrar el funcionamiento del plugin y la interfase de usuario, y explicar los resultados obtenidos en cada uno de ellos.

En el primer caso, vimos que cuando los conceptos de las ontologías son modelados de manera muy diferente el método no sugiere correspondencias. Esto sucede ya que, aunque el resultado del análisis de similitudes a nivel sintáctico y semántico sea alto, la incompatibilidad de conceptualización causa valores bajos, o nulos, en el análisis a nivel estructural. Como consecuencia el resultado final no supera el umbral.

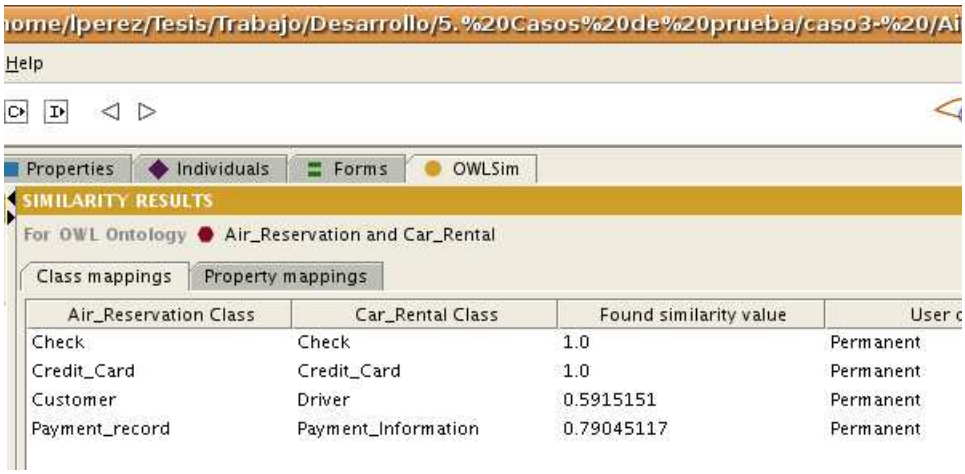

Figura 5.21: Formulario de visualización de las correspondencias de clases definitivas.

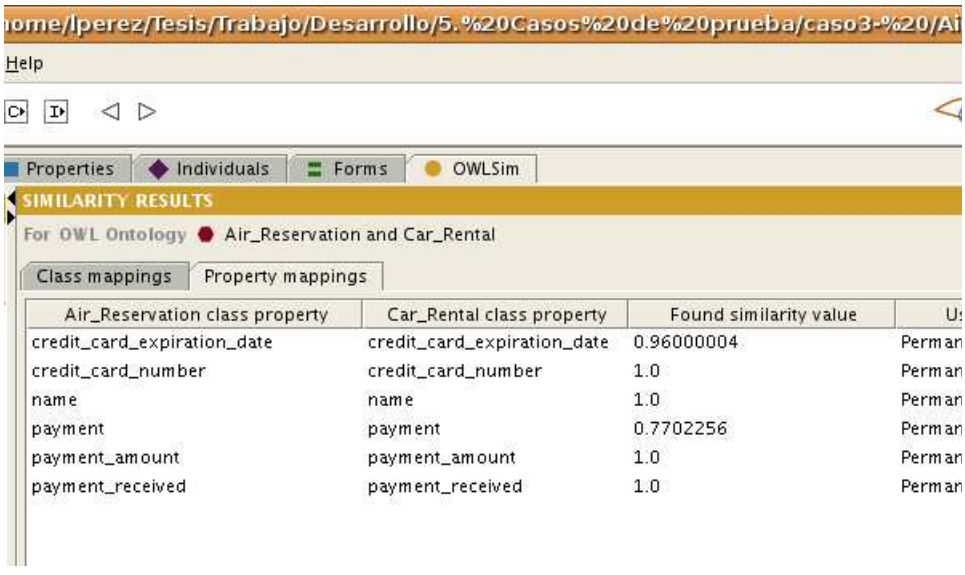

Figura 5.22: Formulario de visualización de las correspondencias de propiedades definitivas.

En segundo lugar, presentamos un caso con definiciones cíclicas en las ontologías. Aquí, mostramos como funciona la detección de ciclos que explicamos en la Sección 3.4. El método detecta que los conceptos ya se analizaron y que además ese análisis todavía esta pendiente de resolución. Seguidamente, si el calculo "parcial" sin el análisis de las propiedades especiales supera igualmente el umbral, entonces se solicita confirmación al usuario.

En el último caso se describe el análisis de dos ontologías más completas en cuanto a clases y propiedades. El método encuentra correspondencias en los conceptos sugeridos ya que éstos tienen un grado de similitud tanto a nivel semántico como sintáctico. Además, mostramos como se reutilizan las correspondencias encontradas y confirmadas con anterioridad, sin volver a realizar los mismos calculos.

En el próximo capitulo concluimos este trabajo de tesis, mencionando los aportes realizados y los objetivos que cumplimos además de plantear algunas propuestas de trabajo futuro.

Capítulo 6

Conclusión

En este trabajo de tesis, hemos desarrollado un plugin para el editor de ontologías Protégé [5], que implementa el método de búsqueda de similitudes entre ontologías propuesto en el trabajo de Tesis [12]. Éste método surge como una estrategia para la tarea de definir correspondencias semánticas entre ontologías, dentro del contexto de los sistemas de información federados.

Brevemente, la capa de federación es el centro de la arquitectura federada propuesta, la cual contiene los componentes para el proceso de integración. Los tres componentes principales (*ontologías fuentes*, *OM* y *vocabulario compartido*) de esta capa son los necesarios para contener la estructura de ontologías y realizar las tareas de búsqueda de correspondencias semánticas necesarias para la construcción de un sistema federado.

Nuestro trabajo parte del método semiautomático de tres niveles (sintáctico, semántico y de usuario) para la búsqueda de similitudes [12]. En cada uno de los niveles se analizan las posibles incompatibilidades ontológicas que surgen al integrar dos o mas ontologías. Por ejemplo, se brindan soluciones para problemas relacionados con términos asociados a conceptos que son sinónimos mediante la utilización de un Tesauro (es este caso *WordNet*).

En el presente capítulo, describimos los objetivos alcanzados al desarrollar este trabajo de tesis y las conclusiones que derivan de los resultados obtenidos. También mencionamos los aportes de este trabajo y las posibles modificaciones que se podrían realizar tanto al método de búsqueda de similitudes, base de esta tesis, como al plugin OWLSim para el editor de ontologías Protégé.

6.1. Resultados del desarrollo del plugin OWLSim

A lo largo del desarrollo de este trabajo se analizó, diseñó e implementó una herramienta de software, plugin OWLSim, para la búsqueda de similitudes entre ontologías. A continuación señalamos las concluciones mas importantes de cada etapa en el desarrolo.

En una primera etapa y a partir del análisis del contexto, sistemas federados, se profundizó en el estudio del método diseñado para realizar una de las principales tareas de la capa de federación, la búsqueda de correspondencias. En primer lugar, vimos que las definiciones de conceptos en las ontologías podrían generar ciclos en

el recorrido que hace el método sobre las mismas. Así, en este trabajo propusimos una extensión del método y diseñamos una *estrategia para el manejo de ciclos.*

En segundo lugar, se considero que agregar la *verificación de valores ya calculados* era una mejora para la *eficiencia* en la implementación del método. Esto se debe a que en el proceso de búsqueda de similitudes es posible volver a analizar un mismo par de conceptos, y que el análisis de un par de conceptos puede involucrar una gran cantidad de cálculos.

En una segunda etapa, se comenzó con el diseño del plugin. Para ello utilizamos la metodología *diseño guiado por responsabilidades.* Ésta nos permitió focalizarnos en la responsabilidad principal de nuestra herramienta a desarrollar y a partir de allí identificar las clases candidatas de nuestro diseño. Cada clase que se identificó cumple roles específicos y ocupa una posición bien definida en la arquitectura diseñada para el plugin (componentes *Interfase de Usuario*, *Modelo del Dominio* y *Transacciones*). Por otro lado, en el diseño de la interfase de usuario, específicamente en la actualización de las vistas de correspondencias, se aplicó el modelo de colaboraciones del patrón *Observer.* Si bien, en este caso los observadores eran objetos conocidos, el uso de este patrón nos facilitó la *reutilización* de un modelo de colaboraciones.

En cuanto a la etapa de implementación de la herramienta como plugin de Protégé (tercer etapa) se tradujo el diseño utilizando el lenguaje de programación *Java.* Éste es el lenguaje en el cual se encuentra implementado Protégé y el lenguaje por defecto de acuerdo a la manera en que se realizan las extensiones. Para nuestro desarrollo resultó adecuado ya que es un lenguaje orientado a objetos que facilito nuestra tarea de codificación.

Uno de los beneficios de implementar el método como plugin para Protégé fue su arquitectura abierta. Como el código fuente esta disponible, existen diversos componentes que se pueden utilizar. En nuestro trabajo utilizamos el componente de interfase gráfica y el componente de modelo de conocimiento (plugin OWL o OWL-API). La utilización del primero, clases especializadas que provee el paquete de interfase gráfica de Protégé, nos permitió mantener el mismo estilo de pantallas. En cuanto al uso del plugin OWL, éste nos facilitó la implementación del componente *Modelo del Dominio* de la arquitectura de nuestro plugin. Precisamente, no fue necesario el diseño e implementación de un componente responsable de leer, analizar y traducir a una estructura de objetos los archivos en lenguaje OWL.

Finalmente, en la etapa de verificación de nuestro plugin buscamos distintos casos de estudio que mostraran cómo trabaja la implementación del método de búsqueda de similitudes. La extracción de ontologías para las pruebas del plugin se hizo desde la Web. La mayor dificultad que enfrentamos fue la de encontrar diversas ontologías sobre *dominios solapados.* Además, la mayoría de las ontologías que hay en la Web describen simples jerarquías y están escritas en los sublenguajes OWL-DL y OWL-Full. Éstas no son posibles de analizar ya que el método de búsqueda de similitudes, y por lo tanto nuestra herramienta de software, fueron diseñados para ontologías definidas en el sublenguaje OWL-Lite. Igualmente, se tomaron las partes solapadas de las ontologías encontradas y se adaptaron para que las definiciones estén dentro del subconjunto de axiomas permitidos en el sublenguaje OWL-Lite.

En cuanto a los resultados que se observan del método de búsqueda de simi-

litudes con los casos de estudio, pudimos observar que no se obtienen resultados positivos cuando las ontologías que se comparan poseen representaciones muy diferentes. Esto sucede porque el método tiene en cuenta la estructura de las clases para determinar similitudes. De aquí se desprende un tipo de incompatibilidad ontológica la cual el método no da solución, *la incompatibilidad de conceptualización para las propiedades*. Este tipo de incompatibilidad es muy complejo de descubrir ya que sólo podemos utilizar la semántica del término asociado al concepto y no sus relaciones o representación. Por otro lado, si tenemos en cuenta sólo la semántica del concepto, podemos encontrarnos con mayores probabilidades de encontrar homónimos como sinónimos ya que la búsqueda de semántica se realiza dentro de un Tesauro. En el caso que ambos términos asociados a los conceptos están denotando diferente semántica y además son homónimos, el Tesauro devuelve un valor de similitud positivo, lo cual sería incorrecto. Sin embargo, el método solo encontrará homónimos como sinónimos, en los casos en que las representaciones de los conceptos sea muy similar. Igualmente, consideramos que las probabilidades de que esto suceda son menores, ya que si una ontología es completa dos conceptos homónimos no deberían tener las mismas propiedades.

6.2. Contribución

El principal aporte del plugin OWLSim fue proveer una herramienta de software que implementa el método semiautomático de búsqueda de similitudes entre ontologías [12], extendido para el manejo de ciclos. Los beneficios de contar con tal herramienta de software son, en primer lugar, que el método pueda ser probado con mayor cantidad de ontologías; y en segundo lugar, que su funcionamiento pueda ser evaluado y comparado con otras herramientas propuestas en la literatura. Además, al ser un plugin de Protégé estará disponible para ser accedido y probado por cualquier persona.

El proceso manual de alineación y mezcla de ontologías es una tarea difícil, laboriosa y propensa a la introducción de errores en el producto final. Hasta ahora, sólo algunas propuestas semiautomáticas han sido implementadas. Nuestro plugin es una herramienta más que proveerá al diseñador de ontologías facilidades para el *reuso* e *integración* de ontologías asistiendo en la búsqueda de correspondencias.

6.3. Trabajos futuros

Del trabajo realizado en esta tesis vemos dos perspectivas de trabajo futuro. Una de ellas sobre el método de búsqueda de similitudes y la otra sobre la implementación de la herramienta de software.

- El método de búsqueda de similitudes puede ser extendido en dos sentidos principales. Por un lado, a nivel de análisis conceptual, una extensión importante sería ampliarlo para que se puedan analizar algunas construcciones de OWL-DL (por ejemplo, `owl:oneOf` o jerarquías). Por otra parte, a nivel de análisis léxico, se podrían incorporar disitntas aplicaciones o herramientas

de procesamiento de lenguaje natural para el análisis de los términos asociados a los conceptos (clases y propiedades) en las ontologías. Por ejemplo, resolución de ambigüedad semántica de las palabras -"*word sense disambiguation (WSD)*", lematización o diccionarios morfológicos. Estas herramientas pueden contribuir en el aumento de la precisión en la comparación de los términos asociados a los conceptos. En cuanto al análisis semántico en particular, se podrían utilizar otras relaciones semánticas de WordNet, como por ejemplo, antónimos, parte de o hiperonimia-hiponimia.

- La funcionalidad del plugin puede ser ampliada para permitir el ajuste del método a distintos requerimientos de análisis de similitudes permitiendo la configuración de ciertos parámetros antes de iniciar el proceso de búsqueda. Por ejemplo, se podría permitir al usuario la modificación de los pesos [1] dados a las diferentes funciones de similitud, en los niveles sintáctico o semántico, que contribuyen al valor de similitud final. Otra forma de ajuste al método, sería permitir la configuración del umbral.

Por último, sería interesante combinar este proceso de búsqueda de similitudes con una estrategía de mezcla, y así, poder generar la ontologia global o vocabuario compartido.

[1] Por ejemplo, $w_{propiedades\_especiales}$ en $w_{propiedades\_especiales} * sim_{propiedades\_especiales}(c_1, c_2)$.

Apéndice A

Detalles de implementación

En este capítulo describimos brevemente el editor de ontologías Protégé, con respecto a su origen y motivación, modelo de conocimiento subyacente y a su arquitectura basada en plugins.

A.1. El editor de ontologías Protégé-OWL

Protégé es un editor de ontologías y un entorno basado en conocimiento, gratuito y de código abierto. Provee un conjunto de herramientas para construir modelos de dominio y aplicaciones basadas en conocimiento con ontologías. Básicamente, implementa un conjunto rico de estructuras de modelado de conocimiento y acciones que soportan la creación, visualización, y manipulación de ontologías en varios formatos de representación. Puede ser extendido, para construir herramientas basadas en conocimiento y aplicaciones, por medio de su arquitectura basada en plugins y de una Interfase de Programación de Aplicaciones (Application Programming Interfase - API) escrita en el lenguaje de programación Java.

La plataforma de Protégé soporta dos modos de modelado de ontologías a través de los editores *Protégé-Frames* y *Protégé-OWL*.

Protégé-Frames [49]: El editor Protégé-Frames permite a los usuarios construir ontologías basadas en marcos ("frame-based"), en concordancia con el protocolo Open Knowledge Base Connectivity protocol (OKBC) [25] para acceder a bases de conocimiento almacenadas en sistemas de representación de conocimiento. Los *marcos* ("frames") son las unidades principales para la construcción de una base de conocimiento. En este modelo, una ontología consiste de un conjunto de *clases*, *propiedades*, *restricciones* y *axiomas*. Las *clases* ("classes") representan los conceptos destacados del dominio, las *propiedades* ("slots") son asociados a clases para describir sus propiedades o relaciones, las *restricciones* ("facets") describen propiedades de las relaciones, y los *axiomas* especifican restricciones adicionales. Una base de conocimiento en este modelo, contiene también un conjunto de *instancias* de esas clases (ejemplares individuales de los conceptos que mantienen valores específicos para sus propiedades).

Protégé-OWL [50]: El editor Protégé-OWL permite a los usuarios construir ontologías para la *Web Semántica*, en particular, en el lenguaje ontológico

OWL [4]. Una ontología OWL puede incluir descripciones de clases, propiedades e instancias. La semántica formal de OWL especifica como derivar sus consecuencias lógicas, es decir, hechos que no están explícita o literalmente presentes en una ontología, sino implicados por la semántica. Estas implicaciones pueden estar basadas en un sólo documento o en múltiples documentos distribuidos que han sido combinados usando mecanismos definidos dentro de OWL [56].

La API central ("core API") [1], que mencionamos anteriormente, se utiliza para acceder a la funcionalidad básica de Protégé y bases de conocimiento basadas en marcos (aquellas creadas con Protégé-Frames). También posee una API OWL [2], que extiende la API central para proveer acceso a ontologías OWL (aquellas creadas con Protégé-OWL). Es decir, podemos ver a Protégé-OWL como una extensión de Protégé. Las APIs de Protégé pueden usarse directamente por aplicaciones externas para acceder a bases de conocimiento creadas con Protégé y hacer uso de los formularios e interfaces de usuarios sin ejecutar Protégé.

A.1.1. Uso de OWLPlugin como API

La API de Protégé OWL [2] es un librería Java de código abierto para OWL y RDF. Provee clases y métodos para cargar y guardar archivos OWL, para consultar y manipular modelos de datos OWL, y para realizar razonamiento basado en motores de Lógica Descriptiva (Description Logic). Aun mas, la API esta optimizada para la implementación de interfaces de usuario y diseñada para utilizarse en dos contextos:

- Desarrollo de componentes que son ejecutados dentro de Protégé.
- Desarrollo de aplicaciones independientes (por ejemplo, Servelts o plugins para Eclipse)

La API de Protégé-OWL tiene poderosas clases que permiten navegar dentro de una ontología de manera sencilla. Esta centrada en una colección de interfaces del paquete modelo ("model") que proveen acceso al modelo OWL en la ontología y sus elementos como clases, propiedades e individuos. La interfase mas importante del modelo es *OWLModel*, la cual provee acceso al contenedor de mayor nivel de los recursos de una ontología. Se puede utilizar OWLModel para crear, consultar y borrar recursos de distintos tipos y luego usar los objetos retornados por *OWLModel* para hacer operaciones especificas.

Para la conveniencia de los desarrolladores en OWL, la OWL API oculta todos los detalles de la API central o base.

A.2. Desarrollo de plugins

Existen seis tipos de plugins para Protégé:

- *Tab widget*: una solapa de interfase de usuario que aparece en la ventana principal de Protégé junto con otras solapas de sistema tales como la solapa de clases ("Classes Tab").

- *Slot widget*: un componente que se muestra en los formularios y se utiliza para ver y adquirir valores de propiedades para las instancias.
- *Back-end*: especifica el mecanismo que Protégé usará para almacenamiento (ya sea como texto o en una base de datos).
- *Createproject*: toma un archivo en un formato fuente producido por otro programa y crea una base de conocimiento Protégé lo mas cercana posible.
- *Export*: provee un mecanismo extensible para exportar bases de conocimiento Protégé en una gran variedad de formatos, tanto de archivos como de base de datos.
- *Project*: permite la manipulación de los proyectos de Protégé y de la interfase de usuario de los proyectos.

Para el desarrollo de plugins del tipo solapa ("Tab Widget") hay dos requerimientos:

- Subclasificar la clase *AbstractTabWidget*.
- Implementar el método *initialize()*.

A.3. Implementación del plugin OWLSim

Como se describió en la sección A.1.1, la API posee interfaces, clases y métodos que permiten cargar un archivo OWL en una estructura de objetos, proveyendo así acceso al modelo OWL en la ontología y sus elementos. En consecuencia, se elimina la necesidad de construir un componente que interprete el archivo OWL y cree la estructura de objetos a partir de éste.

En el diseño del plugin se modelaron las clases que forman el modelo del dominio representando las enologías y sus recursos como clases, propiedades y restricciones. Para su implementación utilizamos las clases e interfaces que provee la OWL-API de Protégé, específicamente, del paquete "model". Las mismas poseen numerosas operaciones, pero en este trabajo solo utilizamos aquellas necesarias para la implementación de las responsabilidades y colaboraciones que se modelaron en nuestro diseño (Capítulo 4).

En la siguiente subsección, describimos brevemente el recurso lingüístico y la interfase que utilizamos para acceder al mismo en la implementación de nuestra función de búsqueda en Tesauro.

A.3.1. Recurso lingüístico e interfase de acceso

Recurso Lingüístico: una breve reseña de *WordNet*

WorNet [6] es una base de datos lexicográfica electrónica para el idioma Ingles, y es considerado uno de los recursos mas importantes disponibles para los investigadores en lingüística computacional y análisis de textos, entre otras áreas

relacionadas. Su diseño esta inspirado en teorías actuales de psicolingüística y computacional sobre la memoria lexicográfica humana.

WorNet consiste de archivos lexicográficos, código para convertir estos archivos en base de datos, y rutinas de búsqueda e interfaces que muestran información desde la base de datos. Los archivos lexicográficos organizan sustantivos, verbos, adjetivos y adverbios en grupos de sinónimos, y describe las relaciones entre grupos de sinónimos.

La información en *WordNet* esta organizada en grupos lógicos llamados grupos de sinónimos ("synsets"). Cada grupo consiste de una lista de *palabras sinónimos* y punteros que describen relaciones entre estos grupos. Una palabra (o conjunto de palabras, palabras compuestas) puede aparecer en más de un grupo y en distintas partes. Las *palabras* en un grupo de sinónimos pueden acomodarse de manera que sean intercambiables en algunos contextos. Estos grupos se conectan por diversas relaciones semánticas como antónimos, sinónimos, hiperonimia-hiponimia, entre otras.

Interfase de acceso: JWNL Java WordNet Library API

La API utilizada como interfase para el recurso lingüístico en nuestra implementación es JWNL (Java WorNet Library). JWNL es una API escrita en Java para acceder a diccionarios relacionales que posean el mismo formato que *WordNet*. Esta provee otras funcionalidades mas allá del acceso a datos, tales como descubrimiento de relaciones y procesamiento morfológico. JWNL es una implementación en java y es de código abierto.

Para utilizar JWNL hay que, primero, llamar al método *JWNL.initialize()* al inicializar la aplicación donde se utiliza (Capítulo 4). Luego, se llama al método *Dictionary.getInstance()* para obtener una instancia del diccionario instalado en el sistema. Los métodos que utilizan de este diccionario son *lookupIndexWord()*, *lookupAllIndexWords()*, y *getIndexWordIterator()*.

También, hay otros metodos que podrían ser útiles como *Relationship.findRelationships()* que permite encontrar relaciones de un tipo dado entre dos palabras (por ejemplo, ancestro).

JWNL provee acceso a la base de datos WordNet a través de tres estructuras diferentes: la distribución estándar de archivos, una base de datos, o una estructura en memoria ("in-memory map"). Se proveen utilidades para convertir del formato de archivos a los otros dos formatos.

El archivo de configuración de JWNL, es un archivo de propiedades para la librería JWNL en formato XML que permite especificar tres propiedades:

Clase de diccionario ("Dictionary class"): esta define la clase usada para interfase con el diccionario. JWNL posee tres clases de diccionario -*MapBackedDictionary, FileBackedDictionary, y DatabaseBackedDictionary.* Sólo un elemento de este tipo se requiere en el archivo de propiedades; si hubiere mas de uno se utilizará el primero.

```
<dictionary class="[dictionary class name]">
    ..parameters
</dictionary>
```

Versión ("Version") brinda información sobre la versión de WordNet con la que se trabaja. Nuevamente solo un elemento de versión es requerido.

```
<version publisher="[publisher]" number="[version number]"
                    language="[language]" country="[country]"/>
```

Recursos ("Resources") especifica un archivo que contiene correspondencias entre claves y textos usados en el programa. Típicamente este archivo contiene los textos de los mensajes de error o de estado que permiten la configuración en distintos idiomas.

```
<resource class="[resource file path]"/>
```

Apéndice B

El Lenguaje Web Ontológico - OWL

Se eligió el lenguaje web ontológico OWL ya que es un estándar y por lo tanto su uso dentro de la comunidad de la Web es cada vez más amplio. En este apéndice describimos con más detalle el lenguaje OWL que presentamos en el Capítulo 2, junto con la estructura de los documentos OWL y las construcciones principales de OWL-Lite. En [4, 56] se presenta una descripción detallada de todo el lenguaje.

B.1. Características de OWL

Los lenguajes ontológicos permiten que los usuarios escriban conceptualizaciones formales y explícitas de modelos de dominio. Los requerimientos principales para esto son [7]:

1. Una sintaxis bien definida
2. Una semántica bien definida
3. Un soporte de razonamiento eficiente
4. Suficiente poder expresivo
5. Conveniencia de expresión

La importancia de una *sintaxis bien definida* es un concepto ya conocido en el área de los lenguajes de programación, ya que se necesita para que la máquina pueda procesar la información (*machine-processable*).

La *semántica formal* describe en forma precisa el significado del conocimiento. El término "en forma precisa" significa que la semántica no posee intuiciones subjetivas ni esta abierta a diferentes interpretaciones por parte de distintas personas (o máquinas). Un uso posible de la semántica formal es permitir a los humanos razonar sobre el conocimiento. En el caso de razonamiento ontológico podemos razonar sobre:

- *Miembro de clases*: Si x es una instancia de la clase C, y C es una subclase de D, entonces podemos inferir que x es una instancia de D.

- *Equivalencia de clases*: Si una clase A es equivalente a una clase B, y la clase B es equivalente a la clase C, luego A es equivalente a C.

- *Consistencia*: Por ejemplo, supongamos que declaramos a x como una instancia de la clase A. Si suponemos que A es una subclase de $B \cap C$, A es una subclase de D, y B y D son disjuntos; luego tenemos una inconsistencia porque A debería ser vacío, pero tiene una instancia (x). Esto indica un error en la ontología.

- *Clasificación*: Si declaramos que ciertos pares valor-propiedad son condiciones suficientes para ser miembro de la clase A, y luego un individuo x satisface esas condiciones, podemos concluir que x debe ser una instancia de A.

La semántica es un prerrequisito para el *soporte de razonamiento* ya que derivaciones como las antes explicadas pueden realizarse en forma automática. De esta manera, permite verificar grandes ontologías donde están involucrados varios autores, e integrar y compartir ontologías de varias fuentes. La semántica formal y el soporte de razonamiento se proveen mediante una correspondencia entre un lenguaje ontológico y un formalismo lógico conocido, y usando razonadores automáticos ya existentes para dichos formalismos.

El *poder expresivo* se ve fuertemente influenciado por el soporte de razonamiento. El lenguaje ontológico debe poseer características de modelado suficientes para poder expresar todo un modelo de dominio. Pero hay un problema con esa expresividad ya que cuanto más expresivo es un lenguaje más ineficiente es el soporte de razonamiento. Así, es necesario un compromiso, el lenguaje debe poder ser soportado por razonadores de eficiencia y a su vez poseer la suficiente expresividad para expresar grandes clases de ontologías y conocimiento. Es aquí donde surge el concepto de *conveniencia de expresión*.

Teniendo en cuenta estos cinco puntos, surgió OWL (Web Ontology Language) [56] junto con sus tres sublenguajes (Full, DL y Lite):

- *OWL-Lite*: Es un sublenguaje de OWL DL ya que provee un subconjunto reducido de construcciones. Por ejemplo, si bien provee restricciones de cardinalidad sólo permite valores de cardinalidad de 0 o 1. La ventaja es que es un lenguaje fácil de entender (desde el punto de vista de los usuarios) y fácil de implementar (desde el punto de vista de desarrollador). La desventaja es que posee una expresividad restringida.

- *OWL-DL*: Es un sublenguaje de OWL Full que restringe la forma en la cual pueden usarse los constructores de OWL y RDF. Por ejemplo, posee separación de tipos, es decir, una clase no puede ser también un individuo o una propiedad. La ventaja es que permite soporte de razonamiento eficiente. La desventaja es que se pierde la compatibilidad total con RDF. Un documento RDF tendrá que ser modificado (extendido en ciertos aspectos y restringido en otros) antes de ser un documento OWL legal, pero cada documento OWL DL legal es todavía un documento RDF legal.

- *OWL-Full*: Usa todas las primitivas del lenguaje OWL permitiendo combinarlas con RDF y RDFS de manera arbitraria. Por ejemplo, una clase puede ser tratada como una colección de individuos y como un individuo en sí simultáneamente. La ventaja de OWL Full es que es totalmente compatible tanto semántica como sintácticamente con RDF. Cualquier documento RDF legal es un documento OWL también legal y cualquier conclusión válida en RDF/RDFS es una conclusión válida también en OWL Full. La desventaja es que el lenguaje es tan poderoso que se vuelve indecidible.

Cada uno de estos sublenguajes es una extensión de su predecesor cumpliendo con el siguiente conjunto de relaciones (la inversa no se cumple):

- Cada ontología OWL Lite legal es un ontología OWL DL legal.
- Cada ontología OWL DL legal es un ontología OWL Full legal.
- Cada conclusión OWL Lite válida es una conclusión OWL DL válida.
- Cada conclusión OWL DL válida es una conclusión OWL Full válida.

Los desarrolladores de ontologías que usen OWL deben decidir qué sublenguaje se ajusta mejor a sus necesidades.

Otra característica importante de OWL es que hace una *suposición de mundo abierto* (como DL), contrario a la suposición de mundo cerrado de los sistemas de base de datos. Esto significa que la ausencia de información en una instancia de una base de datos es interpretada como información negativa (si no está, es falso), en cambio en una instancia de una ontología sólo indica falta de conocimiento. Las descripciones de los recursos no están confinadas a un simple archivo o alcance. Originalmente una clase puede definirse en una ontología pero puede ser extendida en otras ontologías. La nueva información no puede retractar informaciones previas, aunque la misma puede ser contradictoria. Los hechos pueden ser sólo agregados, nunca eliminados.

B.2. Primitivas de modelado del lenguaje OWL-Lite

En esta sección describimos las construcciones del lenguaje OWL, en particular las del sublenguaje OWL-Lite y las cuales forman parte de las ontologías que analizamos con nuestro método de búsqueda de similitudes 3.3.1.

B.2.1. Documento OWL

B.2.1.1. Contenido

Un documento OWL consiste de un encabezado de ontología opcional (generalmente se incluye por lo menos uno) más cualquier número de *axiomas de clase*, *axiomas de propiedades* y *hechos acerca de individuos*. Estos "axiomas" pueden denominarse también, de manera mas informal, definiciones. OWL no impone ningún orden en estos componentes.

Como en la mayoría de los documentos RDF, el código OWL debe ser un subelemento del elemento `rdf:RDF`. Este elemento encierra generalmente declaraciones de espacios de nombres ("namespaces") y declaraciones base.

B.2.1.2. Vocabulario OWL predefinido ("built-in")

El vocabulario predefinido para OWL proviene del espacio de nombres OWL `http://www.w3.org/2002/07/owl#`, por convención asociado al nombre "owl". Se recomienda que las ontologías no utilicen nombres de este espacio de nombres mas que para el mismo uso del vocabulario compartido.

B.2.2. Clases

Las clases proveen un mecanismo de abstracción para agrupar recursos con características similares. Cada clase OWL se asocia con un conjunto de individuos, llamado la *extensión* de la clase. Los individuos dentro de la extensión de la clase se denominan instancias de la clase. Una clase tiene un significado (el concepto subyacente) el cual esta relacionado pero no es igual a la extensión de la clase. Por ello, dos clases pueden tener la misma extensión pero sin embargo, ser clases diferentes.

Nótese que, en OWL-Lite y OWL-DL un individuo nunca puede ser al mismo tiempo una clase ya que clases e individuos son conjuntos disjuntos.

Las clases en OWL se describen a través de "descripciones de clase", los cuales pueden ser combinados en "axiomas de clase".

B.2.2.1. Descripciones de clases

Una descripción de clase es la mínima unidad constructora de los axiomas de clase. Una descripción de clase describe una clase OWL por un nombre o especificando la extensión de una clase anónima (sin nombre). OWL distingue seis tipos de descripciones de clase:

1. un identificador de clase (Uniform Resource Identifier, URI[1]), clases con nombre.
2. una enumeración de individuos que juntos forman las instancias de la clase.
3. una restricción sobre una propiedad.
4. la intersección de dos o mas descripciones de clase.
5. la unión de dos o mas descripciones de clase.
6. el complemento de una descripción de clase.

El primer tipo (clase con nombre) es especial en el sentido que describe a la clase a través de un nombre (sintácticamente representado por una URI). Los

[1]Una URI (Uniform Resource Identifier, identificador de recurso uniforme) es una cadena de caracteres compacta usada para nombrar o identificar un nombre o recurso en Internet

otros cinco tipos de descripciones de clase, describen una clase anónima colocando restricciones sobre la extensión de la clase.

Las descripciones de clases de los tipos 2 al 6, describen respectivamente, una clase que contiene exactamente los individuos enumerados (del tipo 2), la clase de individuos que satisfacen una restricción sobre una propiedad (del tipo 3), o una clase que satisface combinaciones lógicas entre descripciones de clases (tipos 4, 5 y 6). La intersección, unión y complemento pueden ser vistos como los operadores lógicos AND, OR y NOT. Los últimos cuatro tipos de descripciones de clase permiten descripciones anidadas, dando la posibilidad de crear descripciones de clases complejas.

Una descripción de clase del tipo 1 se representa sintácticamente como una instancia de la clase `owl:Class`:

```
<owl:Class rdf:ID="Humano"/>
```

Dos identificadores de clases OWL predefinidos son los llamados `owl:Thing` y `owl:Nothing`. La extensión de la clase *Thing* es el conjunto de todos los individuos, y la extensión de la clase *Nothing* es el conjunto vacío. En consecuencia, todas las clases son subclases de *Thing*, y *Nothing* es subclase de todas las clases.

Como las descripciones del tipo 2, 5 y 6 no pertenecen al sublenguaje OWL-Lite, no las describiremos en este apéndice. Los tipos de descripciones 3 y 4 son permitidos pero con ciertas limitaciones.

Desripción del tipo 3: Restricciones de propiedades

Una restricción sobre una propiedad es un tipo especial de descripción de clase, que describe una clase anónima, es decir, la clase de todos los individuos que satisfacen la restricción. OWL distingue dos tipos de restricciones de propiedad: *restricciones de valor* y *restricciones de cardinalidad*.

Una *restricción de valor* aplica restricciones sobre el rango de la propiedad cuando se aplica a esta descripción particular. Por ejemplo, podríamos referirnos a los individuos para los cuales el valor de la propiedad *adyacenteA* debe ser alguna *Region*. Nótese que esto es diferente a `rdfs:range` el cual se aplica en todas las situaciones en las que la propiedad sea usada.

Una *restricción de cardinalidad* aplica restricciones en el número de valores que una propiedad puede tener en el contexto de esta descripción. Por ejemplo, podríamos decir que para un equipo de fútbol la propiedad *tieneJugadores* debe tener 11 valores.

OWL posee un limitado conjunto de constructores para definir la cardinalidad global de una propiedad, ellos son `owl:FunctionalProperty` y `owl:InverseFunctionalProperty` que describiremos más adelante.

Las restricciones de propiedades tienen el siguiente formato general:

```
< owl:Restriction>
        <owl:onProperty rdf:resource = "(alguna propiedad)" />
        (precisamente una restricción de valor o de cardinalidad)
</owl:Restriction>
```

La clase `owl:Restriction` se define como una subclase de la clase `owl:Class`. Las restricciones de propiedades pueden aplicarse tanto a propiedades tipo de dato ("datatype properties"), propiedades para las cuales el valor es un literal, como a propiedades objeto ("object properties"), propiedades para las cuales el valor es un individuo.

Restricciones de valor

owl:allValuesFrom La restricción de valor `owl:allValuesFrom` es una propiedad predefinida de OWL que asocia una clase del tipo restricción a una descripción de clase o a un rango de datos. Una restricción que contiene la propiedad `owl:allValuesFrom` es usada para describir la clase de todos los individuos para los cuales todos los valores de la propiedad bajo consideración son, o bien, miembros de la extensión de la descripción de clase o son valores de datos dentro de un rango de datos especificado. En otras palabras, se identifica la clase de individuos x para los cuales se mantiene que si el par (x, y) es una instancia de P (la propiedad en cuestión), luego y debería ser una instancia de la descripción de clase o un valor en el rango de datos respectivamente. Por ejemplo,

```
< owl:Restriction>
        <owl:onProperty rdf:resource="#tienePadre" />
        <owl:allValuesFrom rdf:resource="#Humano" />
</owl:Restriction>
```

Esta restricción describe una clase OWL anónima de todos los individuos para los cuales la propiedad "`tienePadre`" sólo tiene valores de la clase `Humano`. Nótese aquí, que esta descripción de clase no indica que la propiedad siempre tiene valores de esta clase, sólo que es verdadero para individuos que pertenecen a la extensión de la clase anónima.

> En OWL-Lite el único tipo de descripción de clase permitido para `owl:allValuesFrom` es una clase con nombre, es decir descripciones de clase del tipo 1.

owl:someValuesFrom La restricción de valor `owl:someValuesFrom` es una propiedad OWL predefinida que relaciona una clase restricción a una descripción de clase o a un rango de datos. Una restricción que contiene `owl:someValuesFrom` describe la clase de los individuos para los cuales al menos un valor de la propiedad involucrada es una instancia de la descripción de clase o un valor en el rango de datos. Es decir, define la clase de los individuos x para los cuales existe por lo menos un y (una instancia de la descripción de clase o un valor en el rango de datos) tal que el par (x, y) es una instancia de P. Esto no excluye que existan otras instancias (x, y') de P para las cuales y' no pertenece a la descripción de clase o rango de datos.

El ejemplo siguiente define la clase de los individuos que tienen al menos un padre que es físico:

```
< owl:Restriction>
        <owl:onProperty rdf:resource="#tienePadre" />
        <owl:someValuesFrom rdf:resource="#Físico" />
</owl:Restriction>
```

> En OWL-Lite el único tipo de descripción de clase permitido para `owl:someValuesFrom` es una clase con nombre.

Restricciones de cardinalidad

En OWL se asume que una clase puede tener un número arbitrario (cero o más) de valores para una propiedad dada. De esta forma se puede restringir a la propiedad como obligatoria o requerida (por lo menos uno), restringir un número especifico de valores en la propiedad, o restringir que la propiedad no debe ocurrir. OWL provee tres construcciones para restringir la cardinalidad de las propiedades de manera local en el contexto de una definición de clase.

> OWL-Lite incluye los tres tipos de limitaciones de cardinalidad pero sólo pueden usarse con los valores "0" o "1".

owl:maxCardinality La restricción de cardinalidad `owl:maxCardinality` es una propiedad OWL predefinida que relaciona una clase restricción a un valor de datos perteneciente a los valores del tipo de datos XML Schema `nonNegativeInteger`. Una restricción conteniendo la limitación `owl:maxCardinality` describe la clase de todos los individuos que tiene como máximo N valores semánticamente diferentes (individuos o valores de datos) para la propiedad en cuestión, donde N es el valor de la limitación de cardinalidad.

El ejemplo siguiente describe la clase de individuos que tienen como máximo dos padres:

```
< owl:Restriction>
        <owl:onProperty rdf:resource= "#tienePadre" />
        <owl:maxCardinality rdf:datatype= "&xsd;nonNEgativeInteger">2
        <owl:maxCardinality/>
</owl:Restriction>
```

owl:minCardinality La restricción de cardinalidad `owl:minCardinality` es una propiedad OWL predefinida que relaciona una clase restricción a un valor de datos perteneciente a los valores del tipo de datos XML Schema `nonNegativeInteger`. Una restricción conteniendo la limitación `owl:minCardinality` describe la clase de todos los individuos que tiene al menos N valores semánticamente diferentes (individuos o valores de datos) para la propiedad en cuestión, donde N es el valor de la limitación de cardinalidad.

El ejemplo siguiente describe la clase de individuos que tienen como mínimo dos padres:

```
< owl:Restriction>
        <owl:onProperty rdf:resource= "#tienePadre" />
        <owl:minCardinality rdf:datatype= "&xsd;nonNEgativeInteger">2
        <owl:minCardinality/>
</owl:Restriction>
```

owl:Cardinality La restricción de cardinalidad `owl:Cardinality` es una propiedad OWL predefinida que relaciona una clase restricción a un valor de datos perteneciente a los valores del tipo de datos XML Schema `nonNegativeInteger`. Una restricción conteniendo la limitación `owl:Cardinality` describe la clase de todos los individuos que tiene exactamente N valores semánticamente diferentes (individuos o valores de datos) para la propiedad en cuestión, donde N es el valor de la limitación de cardinalidad.

Esta construcción es redundante ya que puede ser reemplazada por un par de restricciones `owl:minCardinality` y `owl:maxCardinality` con igual valor, se incluye sólo para proveer una facilidad de abreviatura. El ejemplo siguiente describe la clase de individuos que tienen *exactamente* dos padres:

```
< owl:Restriction>
        <owl:onProperty rdf:resource= "#tienePadre" />
        <owl:Cardinality rdf:datatype= "&xsd;nonNEgativeInteger">2<owl:Cardinality/>
</owl:Restriction>
```

Desripción del tipo 4: intersección de clases

owl:intersectionOf La propiedad `owl:intersectionOf` asocia una clase con una lista de descripciones de clase. Ésta describe una clase cuya extensión contiene precisamente aquellos individuos que son miembros de las extensiones de todas las descripciones de clase en la lista. En el ejemplo siguiente, el valor de `owl:intersectionOf` es una lista de dos descripciones de clases, precisamente enumeraciones de ciudades, ambas describiendo una clase con dos individuos. La intersección resultante es una clase con un sólo individuo, llamado "Tosca", ya que es el único individuo que es común a ambas clases.

```
<owl:Class>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <owl:Thing rdf:about="#Tosca" />
        <owl:Thing rdf:about="#Salome" />
      </owl:oneOf>
    </owl:Class>
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <owl:Thing rdf:about="#Turandot" />
        <owl:Thing rdf:about="#Tosca" />
      </owl:oneOf>
    </owl:Class>
  </owl:intersectionOf>
</owl:Class>
```

OWL-Lite esta restringido en el uso de de `owl:intersectionOf`.

B.2.2.2. Axiomas de clase

Las descripciones de clases son los bloques constructores para la definición de clases a través de axiomas. La forma más simple de un axioma de clase es una

descripción de clase del tipo 1 (clases con nombre o identificador). Sólo especifica la existencia de una clase, usando `owl:Class` con un identificador de clase.

Por ejemplo, el axioma de clase siguiente define que la referencia "#Humano" como el nombre de la clase OWL:

```
<owl:Class rdf:ID="Human"/>
```

Esta es una construcción OWL válida pero no nos dice mucho acerca de la clase "Humano". Los axiomas de clases contienen componentes adicionales que especifican características necesarias y/o suficientes de la clase. OWL contiene tres constructores para combinar descripciones de clases en axiomas:

- `rdfs:subClassOf` nos permite decir que la extensión de clase de una descripción de clase es un subconjunto de la extensión de clase de otra descripción de clase.
- `owl:equivalentClass` nos permite decir que la descripción de clase tiene exactamente la misma extensión que otra descripción de clase.
- `owl:disjointWith` nos permite decir que la extensión de clase de una descripción de clase no tiene miembros en común con la extensión de clase de otra descripción de clase.

OWL-Lite no permite los axiomas `owl:disjointWith`, si permite `rdfs:subClassOf` y `owl:equivalentClass` pero con ciertas limitaciones.

rdfs:subClassOf La construcción `rdfs:subClassOf` se definide como parte de RDF Schema. Si significado es: si la descripción de clase $C1$ se define como una subclase de la descripción de clase $C2$, el conjunto de individuos en la extensión de clase de $C1$ debe ser un subconjunto del conjunto de individuos en la extensión de la clase $C2$. En el ejemplo siguiente, se declara una relación de subclase entre dos clases OWL que se describen a través de sus nombres ("Opera" y "MusicalWork"). Las relaciones de subclase proveen condiciones necesarias de pertenencia a una clase. En este caso, para ser una opera el individuo debe ser un trabajo musical.

```
<owl:Class rdf:ID="Opera">
      <rdfs:subClassOf rdf:resource="#TrabajoMusical" />
</owl:Class>
```

> En OWL-Lite el sujeto de una declaración `rdfs:subClassOf` debe ser un identificador de clase, y el objeto debe ser o un identificador de clase o un restricción de propiedad.

Una clase dada puede tener cualquier numero de axiomas de clase.

owl:equivalentClass `owl:equivalentClass` es una propiedad predefinida en OWL que asocia dos descripción de clase. El significado de este axioma es que las dos descripciones de clases involucradas tienen la misma extensión de clase, es decir, ambas extensiones contienen los mismos individuos.

El ejemplo siguiente es la forma mas simple de axioma `owl:equivalentClass`, asocia dos clases con nombre:

```
<owl:Class rdf:about="#PresidenteDeArgentina">
       <equivalentClass rdf:resource="#ResidentePrincipalDeLaCasaRosada"/>
</owl:Class>
```

Otro ejemplo de `owl:equivalentClass` donde se asocia una descripción de clase del tipo 1 (identificador de clase) con una descripción de clase del tipo 2 (enumeración). En el ejemplo siguiente define las clases de operas que juntas representan las "operas Da Ponte de Mozart".

```
<owl:Class rdf:ID="DaPonteOperaDeMozart">
  <owl:equivalentClass>
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <Opera rdf:about="#Nozze_di_Figaro"/>
        <Opera rdf:about="#Don_Giovanni"/>
        <Opera rdf:about="#Cosi_fan_tutte"/>
      </owl:oneOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

Usando la construcción `equivalentClass` podemos especificar condiciones necesarias y suficientes para la pertenencia a una clase.

> En OWL-Lite el sujeto de una sentencia o axioma debe ser una clase nombrada y el objeto debe ser o bien una clase nombrada o un restricción de propiedad.

B.2.3. Propiedades

OWL distingue entre dos categorías principales de propiedades:

- Propiedades objeto ("Object properties") asocian individuos con individuos.
- Propiedades tipo de dato ("Datatype properties").

Una propiedad objeto se define como una instancia de la clase `owl:ObjectProperty` predefinida en OWL, y una propiedad tipo de dato como una instancia de la clase `owl:DatatypeProperty`. Ambas son subclases de la clase RDF `rdf:Property`.

Un axioma de propiedad define características de una propiedad y en su forma más simple sólo especifica la existencia de la misma:

```
<owl:ObjectProperty rdf:ID="hasParent"/>
```

Ésta sentencia define una propiedad con la restricción de que sus valores deben ser individuos. Frecuentemente, los axiomas definen características adicionales de las propiedades. OWL contiene los siguientes constructores de axiomas de propiedades:

- construcciones de RDF Schema `rdfs:subPropertyOf`, `rdfs:domain` y `rdfs:range`

- relaciones con otras propiedades `owl:equivalentProperty` y `owl:inverseOf`
- restricciones globales de cardinalidad `owl:FunctionalProperty` y `owl:InverseFunctionalProperty`
- características lógicas de las propiedades `owl:SymmetricProperty` y `owl:TransitiveProperty`

Nótese que se utiliza aquí también el término "extensión de propiedad" de igual manera que para las clases. La extensión de la propiedad es el conjunto instancias que es asociado con la propiedad. Las instancias de las propiedades son pares sujeto-objeto. También se pueden llamar "tuplas" de una relación binaria.

B.2.3.1. Construcciones de RDF Schema

rdfs:subPropertyOf Un axioma `rdfs:subPropertyOf` define que la propiedad es subpropiedad de alguna otra. Formalmente esto significa que si $P1$ es una subpropiedad de $P2$, luego la extensión de la propiedad $P1$ (conjunto de pares) debe ser un subconjunto de la extensión de la propiedad $P2$ (también un conjunto de pares). El ejemplo siguiente, especifica que todas las instancias (pares) contenidos en la extensión "tieneMadre" también son miembros de la extensión de propiedad "tienePadres".

```
<owl:ObjectProperty rdf:ID="tieneMadre">
  <rdfs:subPropertyOf rdf:resource="#tienePadres"/>
</owl:ObjectProperty>
```

Este constructor puede aplicarse tanto a propiedades especiales como propiedades tipo de dato.

rdfs:domain es una propiedad predefinida que asocia una propiedad a una descripción de clase. Establece que los sujetos de una definición de propiedad deben pertenecer a la extensión de clase de la descripción de clase indicada.

Se permiten múltiples axiomas `rdfs:domain` y deben ser interpretados como conjunciones: se restringe el dominio de la propiedad a aquellos individuos que pertenecen a la intersección de las descripciones de clases. Si quisiéramos decir que múltiples clases actúan como dominio se debería usar el constructor `owl:unionOf`. En el ejemplo siguiente, el axioma especifica que los sujetos en el dominio de la propiedad "tieneCuentaBancaria" deben pertenecer a la clase "Persona" o "Corporación":

```
<owl:ObjectProperty rdf:ID="tieneCuentaBancaria">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Persona"/>
        <owl:Class rdf:about="#Corporación"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
</owl:ObjectProperty>
```

> En OWL-Lite el valor de `rdfs:domain` debe ser un identificador, es decir, no se permiten uniones.

rdfs:range es una propiedad predefinida que asocia una propiedad a una descripción de clase o un rango de datos. `rdfs:range` establece que los valores de esta propiedad deben pertenecer a la extensión de la clase de la descripción de clase o valores de datos especificado en el rango de datos.

Igual que en la construcción anterior se pueden definir múltiples axiomas de rango interpretados como la unión o intersección. Nótese, que a diferencia de las restricciones de valor descriptas en la Sección B.2.2.1, de descripciones de clase, la restricción `rdfs:range` es global.

> En OWL-Lite el único tipo de descripciones de clase que se permiten como objetos de `rdfs:range` son clases nombradas.

B.2.3.2. Relaciones con otras propiedades

owl:equivalentProperty La construcción `owl:equivalentProperty` se utiliza para especificar que dos propiedades tienen las mismas extensiones. Ésta es una propiedad predefinida con `rdf:Property` como rango y dominio.

owl:inverseOf Las propiedades tienen una dirección, de domino a rango. En la practica es útil definir relaciones en dos direcciones. Por ejemplo, "las personas poseen autos" y "los autos son poseidos por personas". Éste axioma puede ser usado para definir tales relaciones entre propiedades. `owl:inverseOf` es una propiedad predefinida con `owl:ObjectProperty` como dominio y rango. Un axioma de la forma `P1 owl:inverseOf P2` establece que para cada par (x, y) en la extensión de la propiedad de $P1$, existe un par (y, x) en la extensión de la propiedad $P2$, y vice versa. Luego, `owl:inverseOf` es una propiedad simétrica. El ejemplo siguient muestra dos propiedades donde cada una es inversa de la otra:

```
<owl:ObjectProperty rdf:ID="hasChild">
      <owl:inverseOf rdf:resource="#hasParent"/>
</owl:ObjectProperty>
```

B.2.3.3. Restricciones de cardinalidad globales

owl:FuncionalProperty Una propiedad funcional es aquella que tiene un único valor y para cada instancia x. Tanto las propiedades objeto como las propiedades tipo de dato pueden declararse como "funcionales". La clase predefinida OWL `owl:FuncionalProperty` es una subclase de la clase RDF `rdf:Property`.

Por ejemplo, el axioma siguiente especifica que la propiedad "esposo" es funcional:

```
<owl:ObjectProperty rdf:ID="esposo">
     <rdf:type    rdf:resource="&owl;FunctionalProperty" />
     <rdfs:domain rdf:resource="#Mujer" />
     <rdfs:range  rdf:resource="#Hombre" />
</owl:ObjectProperty>
```

owl:InverseFuncionalProperty Si una propiedad es declarada como inversamente funcional, luego el objeto de la sentencia de definición de la propiedad determina univocamente el sujeto. Formalmente, si decimos que P es `owl:InverseFuncionalProperty`, esto establece que un valor y solo puede ser el valor de P para una sola instancia x. Un axioma de propiedad inversa-funcional es

especificado declarando la propiedad como una instancia de la clase OWL predefinida `owl:InverseFuncionalProperty`, la cual es subclase de `owl:ObjectProperty`.

En el ejemplo siguiente, la propiedad "MadreBiologica" define que para los individuos del rango (clase "Humano") existe sólo un individuo asociado en la clase dominio ("Mujer"):

```
<owl:InverseFunctionalProperty rdf:ID="MadreBiologica">
     <rdfs:domain rdf:resource="#Mujer"/>
     <rdfs:range rdf:resource="#Humano"/>
</owl:InverseFunctionalProperty>
```

Una diferencia con las propiedades funcionales es que para las propiedades inversa-funcional no se requiere ningún axioma adicional, estas son por definición propiedades objeto.

B.2.3.4. Características lógicas de las propiedades

owl:TransitiveProperty Cuando se define una propiedad P como transitiva, significa que si el par (x, y) es una instancia de P, y el par (y, z) también es una instancia de P. Luego podemos decir que el par (x, z) es una instancia de P. Una propiedad se define como transitiva creandola como instancia de la clase OWL predefinida `owl:TransitiveProperty`, la cual es, a su vez, definida como subclase de `owl:ObjectProperty`.

Un ejemplo típico, son las propiedades representando relaciones parte-todo, donde una instancia de clase es parte de otra instacia de la misma clase:

```
<owl:TransitiveProperty rdf:ID="subRegionDe">
     <rdfs:domain rdf:resource="#Region"/>
     <rdfs:range  rdf:resource="#Region"/>
</owl:TransitiveProperty>
```

owl:SymmetricProperty Una propiedad simétrica es una propiedad para la cual se mantiene que si el par (x, y) es una instancia de P, luego el par (y, x) es también instancia de P. Una propiedad se define como simétrica creandola como instancia de la clase OWL predefinida `owl:SymmetricProperty`, la cual es, a su vez, subclase de `owl:ObjectProperty`.

Un ejemplo clásico es la relación "amigo de":

```
<owl:SymmetricProperty rdf:ID="amigoDe">
     <rdfs:domain rdf:resource="#Human"/>
     <rdfs:range  rdf:resource="#Human"/>
</owl:SymmetricProperty>
```

Bibliografía

[1] Protégé-frames -javadoc. http://protege.stanford.edu/doc/pdk/api/index.html.

[2] Protégé-owl -javadoc. http://protege.stanford.edu/download/release-javadoc-owl/index.html.

[3] *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware* (New York, NY, USA, 2004), Springer-Verlag New York, Inc. Conference Chair-Hans-Arno Jacobsen.

[4] Web ontology language. http://www.w3.org, 2004.

[5] Protégé. http://protege.stanford.edu, 2007.

[6] Wordnet. http://wordnet.princeton.edu/, 2007.

[7] Antoniou, G., and van Harmelen, F. Web ontology language: Owl. In *Handbook on Ontologies in Information Systems* (2003), S. Staab and R. Studer, Eds., Springer-Verlag.

[8] Arens, Y., Hsu, C., and Knoblock, C. Retrieving and integrating data from multiple information sources. *International Journal in Intelligent and Cooperative Information Systems 2*, 2 (1993), 127–158.

[9] Barsalou, T., Siambela, N., Keller, A. M., and Wiederhold, G. Updating relational databases through object-based views. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1991), ACM Press, pp. 248–257.

[10] Baru, C. K., Ludäscher, B., Papakonstantinou, Y., Velikhov, P., and Vianu, V. Features and requirements for an xml view definition language: Lessons from xml information mediation. In *QL* (1998).

[11] Biron, P., and Malhotra, A. Xml schema part 2: Datatypes. W3C Recommendation, May 2001. Available at http://www.w3.org/TR/xmlschema-2/.

[12] Buccella, A. Integración de datos en base a ontologías e información contextual. Master's thesis, Universidad Nacional del Sur, 2005.

[13] Buccella, A., Cechich, A., and Brisaboa, N. Ontology-based data integration methods: A framework for comparison. *Revista Colombiana de Computación 6*, 1 (2005).

[14] Buccella, A., Cechich, A., and Brisaboa, N. R. A context-based ontology approach to solve explanation mismatches. In *JCC'03 Jornadas Chilenas de Computación* (Chillan, Chile, November 2003).

[15] Buccella, A., Cechich, A., and Brisaboa, N. R. An ontological approach to federated data integration. In *Proceedings of the CACIC'03 9 Congreso Argentino en Ciencias de la Computación* (La Plata, Argentina, 2003), pp. 905–916.

[16] Buccella, A., Cechich, A., and Brisaboa, N. R. An ontology approach to data integration. *Journal of Computer Science and Technology 6*, 1 (2003), 62–68.

[17] Buccella, A., Cechich, A., and Brisaboa, N. R. A federated layer to integrate heterogeneous knowledge. In *VODCA'04 First International Workshop on Views on Designing Complex Architectures* (Bertinoro, Italia, September 2004), Electronic Notes in Theoretical Computer Science, Elsevier Science B.V, pp. 101–118.

[18] Buccella, A., Cechich, A., and Brisaboa, N. R. Taking advantages of ontology and contexts to determine similarity of data. In *Proceedings of the CACIC'04 10 Congreso Argentino en Ciencias de la Computación* (La Matanza, Buenos Aires, 2004), pp. 30–41.

[19] Buccella, A., Cechich, A., and Brisaboa, N. R. A three-level approach to determine ontological similarity. In *JCC'04 Jornadas Chilenas de Computación* (Arica, Chile, November 2004).

[20] Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., and Stal, M. *Pattern Oriented Software Arquitecture - A system of Patterns.* Wiley and Sons Ltd., West Sussex, England, 1996.

[21] Busse, S., Kutsche, R., Leser, U., and Weber, H. Federated information systems: Concepts, terminology and architectures. Tech. Rep. Nr. 99-9, Technical University of Berlin, 1999.

[22] Busse, S., Kutsche, R.-D., and Leser, U. Strategies for the conceptual design of federated information systems. In *Engineering Federated Information (Database) Systems (EFIS)* (2000), pp. 23–32.

[23] Candan, K. S., Liu, H., and Suvarna, R. Resource description framework: metadata and its applications. *SIGKDD Explor. Newsl. 3*, 1 (2001), 6–19.

[24] Chandrasekaran, B., Josephson, J. R., and Benjamins, V. R. What are ontologies, and why do we need them? *IEEE Intelligent Systems 14*, 1 (1999), 20–26.

[25] Chaudri, V., Farquhar, A., Fikes, R., Karp, P., and Rice, J. OKBC: A Programmatic Foundation for Knowledge Base Interoperability. In *Proceedings 15th National Conference on Artificial Intelligence (AAAI-98)* (1998), AAAI Press, pp. 600–607.

[26] Connolly, D., Harmelen, F. V., Horrocks, I., McGuinness, D., Patel-Schneider, P., and Stein, L. Daml+oil reference description. W3C, December 2001. Available at http://www.w3.org/TR/daml+oil-reference.

[27] Cui, Z., and O'Brien, P. Domain Ontology Management Environment. In *Proceedings of the 33rd Hawaii International Conference on System Sciences* (2000), IEEE.

[28] Didion, J. The Java WordNet Library, 2004.

[29] Farquhar, A., Fikes, R., and Rice, J. The ontolingua server: a tool for collaborative ontology construction. In *International Journal of Human-Computer Studies* (1996).

[30] Fensel, D. *Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce*, 2nd edition ed. Springer-Verlag, Berlin 2003.

[31] Fowler, M., and Scott, K. *UML distilled (2nd ed.): a brief guide to the standard object modeling language.* Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 2000.

[32] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. *Design Patterns: Elements of Reusable Object-Oriented Software.* Adison-Wesley Professional Computing Series, 1995.

[33] Goh, C. H. *Representing and reasoning about semantic conflicts in heterogeneous information systems.* PhD thesis, 1997. Supervisor-Stuart E. Madnick.

[34] Gruber, T. Ontolingua: A mechanism to support portable ontologies. Tech. Rep. KSL 91-66, Knwoledge Systems Laboratory, Stanford University, Stanford, 1992.

[35] Gruber, T. A translation approach to portable ontology specifications. *Knowledge Acquisition 5*, 2 (1993), 199–220.

[36] Haarslev, V., and Moller, R. Information system integration. In *Proceedings of the CEUR-WS International Workshop on Description Logics* (Linkoeping, Sweden, August 1999), P. Lambrix, A. Borgida, M. Lenzerini, R. Moller, and P. Patel-Schneider, Eds., no. 22.

[37] Hasselbring, W. Information system integration. *Communications of the ACM 43*, 6 (2000), 32–38.

[38] Hasselbring, W. Information system integration. *Communications of the ACM 43*, 6 (2000), 32–38.

[39] Horrocks, I. The fact system. In *TABLEAUX '98: Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods* (London, UK, 1998), vol. 1397, Springer-Verlag, pp. 307–312.

[40] Horrocks, I., Fensel, D., Harmelen, F., Decker, S., Erdmann, M., and Klein, M. Oil in a nutshell. In *Knowledge Acquisition, Modeling and Management* (2000), pp. 1–16.

[41] Horrocks, I., Sattler, U., and Tobies, S. Practical reasoning for expressive description logics. In *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)* (September 1999), H. Ganzinger, D. McAllester, and A. Voronkov, Eds., no. 1705 in Lecture Notes in Artificial Intelligence, Springer-Verlag, pp. 161–180.

[42] Lassila, O., and Swick, R. Resource description framework (rdf) model and syntax specification. W3C Recommendation, February 1999.

[43] Lee, B. *Weaving the Web*. Texere Publishing Ltd., June 2001.

[44] Len Bass, P. C., and Kazman, R. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Masachusetts, USA, 1998.

[45] Levenshtein, I. V. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory* (1966).

[46] Lin, D. An information-theoretic definition of similarity. In *Proceedings of the Fifteenth International Conference on Machine Learning* (July 1998), pp. 296–304.

[47] Maedche, A., and Staab, S. Measuring similarity between ontologies. In *Proceedings of the EKAW'02 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web* (London, UK, 2002), Springer-Verlag, pp. 251–263.

[48] Mcguiness, D., Fikes, R., Rice, J., and Wilder, S. An Environment for Merging and Testing Large Ontologies. In *Proc. 17th Intl. Conf. on Principles of Knowledge Representation and Reasoning (KR'2000)* (Colorado,USA, April 2000), pp. 483–493.

[49] Noy, N., Fergerson, R., and Musen, M. The knowledge model of protégé-2000: combining interoperability and flexibility. In *2nd International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000)* (Juan-les-Pins, France, 2000).

[50] Noy, N., Sintek, M., Decker, S., Crubézy, M., Fergerson, R., and Musen, M. Creating semantic web contents with protégé-2000. In *IEEE Intelligent Systems* (2001), no. 2(16), pp. 60–71.

[51] Noy, N. F., and Musen, M. A. Prompt: Algorithm and tool for automated ontology merging and alignment. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence* (2000), AAAI Press / The MIT Press, pp. 450–455.

[52] Ozsu, M. T., and Valduriez, P. *Distributed Databases: Principles and Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[53] Richardson, R., and Smeaton, A. Using wordnet in a knowledge-based approach to information retrieval. Tech. Rep. CA-0395, Dublin City Univ., School of Computer Applications, Dublin, Ireland, 1995.

[54] Rodríguez, M. A., and Egenhofer, M. J. Putting similarity assessments into context: Matching functions with the user's intended operations. In *CONTEXT '99: Proceedings of the Second International and Interdisciplinary Conference on Modeling and Using Context* (London, UK, 1999), Springer-Verlag, pp. 310–323.

[55] Rodríguez, M. A., and Egenhofer, M. J. Determining semantic similarity among entity classes from different ontologies. *IEEE Transactions on Knowledge and Data Engineering 15*, 2 (2003), 442–456.

[56] Smith, M. K., Welty, C., and McGuinness, D. Owl web ontology language guide. W3C, February 2004.

[57] Stumme, G., and Maedche, A. Fca-merge: Bottom-up merging of ontologies. pp. 225–230.

[58] Tversky, A. Features of similarity. *Psychological Review 84*, 4 (1977), 327–352.

[59] Visser, P., Jones, D., Bench-Capon, T., and Shave, M. An analysis of ontology mismatches; heterogeneity versus interoperability. In *AAAI 1997 Spring Symposium on Ontological Engineering* (1997).

[60] Visser, U., and Schlieder, C. Modelling with ontologies. In *Proceedings of the Ontology and Modeling of Real Estate Transactions in European Juristictions* (Ashgate, 2002).

[61] Wache, H., Vele, T., Visser, U., Stuckenschmidt, H., Schuster, G., Neumann, H., and Hbner, S. Ontology-based integration of information - a survey of existing approaches. In *Proceedings of the IJCAI-01 Workshop: Ontologies and Information Sharing* (Seattle, WA, 2001), pp. 108–117.

[62] Weibel, S., Gridby, J., and Miller, E. Oclc/ncsa metadata. Tech. rep., Dublin, EUA, 1995. http://www.oclc.org:5046/oclc/research/conferences/metadata/dublin_core_report.html.

[63] Wirfs-Brock, R., and McKean, A. *Object Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley 2003, 2003.

[64] Wirfs-Brock, R., Wilkerson, B., and Wiener, L. *Designing Object-Oriented Software*. Prentice Hall 1990, 1990.