



Informe Técnico

DLSI-01/05/2005

Integración entre la Base de Datos *G* y el Editor de Ontologías *Protégé 2000 (V2.1.2)*

Ernesto Jiménez, Rafael Berlanga, Ismael Sanz
Temporal Knowledge Bases Group
Correo electrónico: {ejimenez, berlanga, isanz}@uji.es

Departamento de Lenguajes y Sistemas Informáticos

Universidad Jaume I de Castellón

Campus de Riu Sec

E-12080 Castellón, España

RESUMEN

El objetivo de este proyecto ha sido extender la funcionalidad de Protégé 2000¹, una herramienta para editar ontologías, y permitir el almacenamiento de éstas en G, y su posterior recuperación. G es una base de datos semiestructurada desarrollada por la empresa Helide S.A.². De esta forma podemos dar una visión ontológica de los diferentes objetos almacenados en la base de datos G.

Para llevar a cabo el diseño de las diferentes clases se ha utilizado como base un 'storage plugin' ya implementado. Concretamente se trabajará con el RDF Backend³. Con este paquete podremos reutilizar gran parte del trabajo realizado en lo que respecta a la estructura de las clases y el intercambio de información entre ellas.

La implementación se ha realizado en el lenguaje Java, ya que Protégé proporciona una interfaz para extensiones (plugins) en este lenguaje.

PALABRAS CLAVE

Ontología, Protege, Base de Datos Semiestructurada, Backend, Web Semántico, Lenguajes para Definición de Vistas

LISTA DE ABREVIATURAS

| | |
|-----|-------------------------------------|
| RDF | Resource Description Framework |
| OWL | Ontology Web Language |
| XML | eXtensible Markup Language |
| OID | Object Identifier |
| G | La Base de Datos Semiestructurada G |

¹ Editor de Ontologías: <http://protege.stanford.edu>

² Helide: <http://www.helide.com/>

³ RDF Backend Plugin: <http://protege.stanford.edu/plugins/rdf/>

TABLA DE CONTENIDO

| | |
|--|-----------|
| RESUMEN | 2 |
| PALABRAS CLAVE | 2 |
| LISTA DE ABREVIATURAS | 2 |
| TABLA DE CONTENIDO | 3 |
| 1. INTRODUCCIÓN | 5 |
| 1.1 Desarrollo de ontologías..... | 5 |
| 1.1.1 Definición formal de Ontología..... | 5 |
| 1.1.2 Alcance de las Ontologías | 6 |
| 1.1.3 El Web Semántico | 6 |
| 1.1.4 Ontología utilizada en el Proyecto | 6 |
| 1.2 Protege-2000, un entorno de desarrollo de ontologías extensible..... | 7 |
| 1.2.1 Objetivo de Protégé | 7 |
| 1.2.2 Descripción General de Protégé | 8 |
| 1.2.3 Interfaz para el desarrollo de Plug-ins..... | 8 |
| 1.3 El Sistema de Base de Datos G | 9 |
| 1.3.1 Base de Datos | 9 |
| 1.3.2 Arquitectura | 9 |
| 1.3.3 Modelo de datos | 9 |
| 2. FUNCIONALIDAD AUXILIAR DEL PLUGIN | 11 |
| 2.1 G Client Library for Java..... | 11 |
| 2.2 Definición de un Esquema de Datos | 12 |
| 2.2.1 Creación de Slots del Sistema Auxiliares..... | 13 |
| 2.3 Project Plug-in | 15 |
| 2.4 Eliminación de Ontologías | 15 |
| 2.5 Fichero Import.log..... | 15 |
| 2.6 Historial de servidores y bases de datos G | 16 |
| 2.7 Espacio de nombres | 16 |
| 2.8 Restricciones de una Clase | 17 |
| 2.9 Nombres válidos para los tipos (D.y) de G | 17 |
| 2.10 Tipos y claves de los campos en minúsculas..... | 18 |
| 3. PAQUETES BACKEND - IMPORT – EXPORT | 19 |
| 4. PLUG-INS IMPORT Y EXPORT | 20 |
| 4.1 Algoritmo de ordenación topológica | 20 |
| 4.2 Eliminación de Objetos | 23 |
| 4.3 Modificación de objetos (problemas con version-ID)..... | 23 |
| 5. FUNCIONALIDAD COMO BACKEND PLUG-IN | 24 |
| 5.1 Proceso <i>Save</i> | 24 |
| 5.2 Proceso <i>Load</i> | 24 |

| | | |
|-----------|---|-----------|
| 6. | PLUG-IN PARA LA REALIZACIÓN DE VISTAS | 25 |
| 6.1 | Gramática del Lenguaje OntoPathView | 25 |
| 6.2 | Implementación Parser: Java Compiler Compiler..... | 26 |
| 6.3 | Implementación Plugin..... | 27 |
| 6.3.1 | Extracción Objetos de la Consulta..... | 27 |
| 6.3.2 | Extracción concepto de partida: | 28 |
| 6.3.3 | Extracción de superclases y subclases:..... | 28 |
| 6.3.4 | Extracción de propiedades..... | 28 |
| 6.3.4.1 | Funciones auxiliares: | 29 |
| 6.3.4.2 | Nuevos slots del sistema:..... | 29 |
| 6.3.5 | Extracción de Instancias | 30 |
| 6.3.5.1 | Comparación de Valores según Operador y Tipo. | 30 |
| 6.3.6 | Unión de los Objetos de las Consultas | 31 |
| 6.3.7 | Tratamiento de las Relaciones de Asociación | 31 |
| | Caso 1: Relaciones directas, están clases dominio y destino | 31 |
| 6.3.7.1 | Implementación | 32 |
| 6.3.8 | Inferencia de la nueva Jerarquía de Clases..... | 33 |
| 6.3.9 | Tratamiento de las propiedades transitivas..... | 34 |
| 6.3.10 | Actualización de las Instancias..... | 36 |
| 7. | G SCHEMA IMPORT | 37 |
| 7.1 | Modelo de Datos de los Esquemas | 37 |
| 7.2 | Notas sobre la Implementación | 37 |
| 7.3 | Tratamiento del espacio de nombres | 39 |
| 8. | TRABAJO FUTURO | 40 |
| 8.1 | Adaptación a Protégé 3.0..... | 40 |
| 8.2 | Compatibilidad con OWL | 40 |
| 8.3 | Desarrollo de un Entorno Colaborativo..... | 40 |
| 9. | LISTA DE DOCUMENTOS DE REFERENCIA | 41 |
| | ANEXOS | 42 |
| A. | FORMAL DEFINITION OF AN ONTOLOGY AND A VIEW | 42 |

1. INTRODUCCIÓN

1.1 Desarrollo de ontologías

1.1.1 Definición formal de Ontología

El sentido filosófico del término ontología hace referencia a la esencia misma del ser, a su existencia (onto=ser). Para los sistemas de Inteligencia Artificial, lo que existe es lo que puede representarse.

En nuestro ámbito, una ontología es una descripción formal y explícita de los conceptos de un dominio, y las relaciones entre ellos. También describirá las propiedades y atributos de los conceptos, así como las restricciones de estos atributos. Una ontología junto con un conjunto de instancias individuales de clases dará lugar a una Base de Conocimiento.

El desarrollo de ontologías se ha estado desplazando, en los últimos años, desde el ámbito de los laboratorios de Inteligencia Artificial a los escritorios de los expertos en el dominio. A continuación se exponen las razones del interés de las ontologías [1][2]:

1. Describir un vocabulario común para investigadores que necesitan compartir información en un mismo dominio.
2. Compartir una comprensión común de la estructura de la información entre personas o agentes software.
3. Permitir la reutilización del conocimiento del dominio.
4. Hacer explícitos supuestos sobre el dominio.
5. Separar el conocimiento del dominio del conocimiento operacional.
6. Analizar el conocimiento del dominio.

Destaquemos que el desarrollo de ontologías es diferente del diseño de clases y relaciones en la programación orientada a objetos. La programación orientada a objetos se basa principalmente en los métodos de las clases—un programador toma sus decisiones de diseño en base a las propiedades *operacionales* de una clase, mientras que un diseñador de ontologías toma dichas decisiones basándose en las propiedades *estructurales* de una clase. Como resultado, la estructura de una clase y las relaciones entre las clases en una ontología son diferentes de la estructura para un dominio similar, en un programa orientado a objetos.

1.1.2 Alcance de las Ontologías

Las ontologías no deberán contener toda la información posible sobre el dominio: no es necesario especializar (o generalizar) más de lo que la aplicación requiera (como mucho un nivel extra en cada sentido).

Por ejemplo [3], para el desarrollo de una ontología que describa qué vinos acompañan mejor a una cierta comida, no necesitamos saber qué papel es utilizado para las etiquetas de los vinos o cómo cocinar platos de gambas. Además, sólo representaremos las propiedades más notables de las clases de nuestra ontología. Para la ontología acerca de vinos y comidas, no sería necesario incluir todas las propiedades que un vino o comida podría tener.

1.1.3 El Web Semántico

Las Ontologías se han convertido en algo común en la World-Wide Web, abarcando desde largas taxonomías categorizando Web Sites (como en Yahoo!) a categorizaciones de productos en venta y sus características (como en Amazon.com). Además una aplicación reciente que está ganando importancia es el Web Semántico, cuyo objetivo es enriquecer la Web actual, creando objetos de conocimiento que permitan tanto a los usuarios como a los programadores mejorar la explotación de los recursos Web.

El WWW Consortium (W3C)⁴ ha desarrollado el formato RDF (Resource Description Framework), un lenguaje para codificar el conocimiento en las páginas Web y hacerlo comprensible a los agentes electrónicos buscadores de información. Junto a RDF existen otros lenguajes para el desarrollo del Web Semántico como puedan ser: OWL, SHOE, etc. Estos lenguajes deberán conseguir expresar la información de forma precisa y que pueda ser interpretada por un computador, es decir, deberán permitir describir el contenido de la Web como una gran colección de recursos con etiquetas que sean significativas.

1.1.4 Ontología utilizada en el Proyecto

En la figura 1 se puede apreciar la ontología que se ha utilizado como ejemplo para la realización de las pruebas. Es una sencilla ontología que describe las relaciones entre obras, artistas y museos que exhiben las obras.

⁴ WWW Consortium: <http://www.w3.org/>

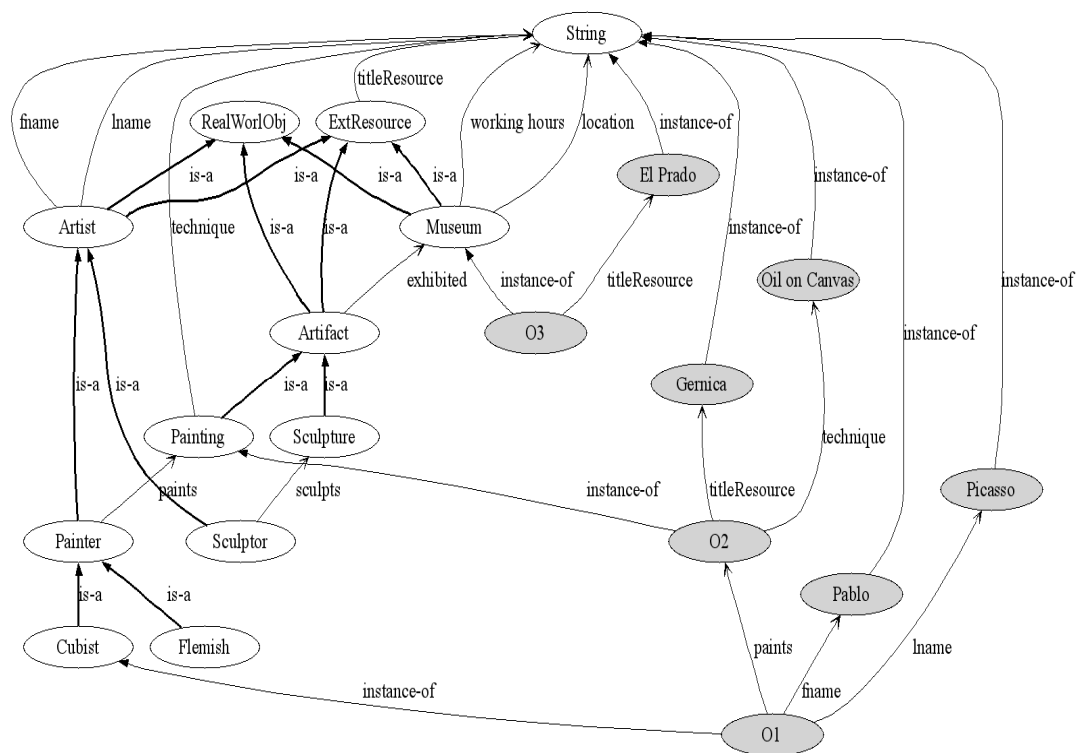


Figura 1: Ontología de ejemplo

1.2 Protege-2000, un entorno de desarrollo de ontologías extensible

1.2.1 Objetivo de Protégé

El conocimiento acerca del dominio de aplicación es uno de los más importantes problemas a la hora de desarrollar exitosamente un proyecto de software. Antes de empezar la codificación de nuestro software debemos conocer bien el funcionamiento de la empresa o sección de la misma, y así saber qué procesos de ella se podrán automatizar.

El objetivo será adquirir el conocimiento necesario a través de los expertos del dominio y reflejarlo en algún tipo de modelo de dominio. UML (Unified Modeling Language) podría ser una buena solución para representar los modelos del dominio con diagramas de clases y casos de uso, describiendo los diferentes procesos. Pero el problema es que estamos básicamente ante un lenguaje para programación orientada a objetos, que solo unos pocos expertos del dominio podrán entender. Además solo podremos trabajar con un conjunto fijo de constructores, poco flexibles para entornos específicos.

Si se quiere implicar a los expertos del dominio y a los clientes más de cerca en el proceso de desarrollo, deberemos usar algo más expresivo que UML. Protégé-2000 es una herramienta simple de usar pero muy potente, optimizada para construir

modelos de dominio. Esto nos permitirá que tanto desarrolladores como expertos de dominio puedan construir de forma sencilla Bases de Conocimiento.

1.2.2 Descripción General de Protégé

Protégé-2000 es una herramienta software usada por los desarrolladores de sistemas y por los expertos de dominio para desarrollar sistemas basados en el conocimiento. Las aplicaciones desarrolladas con Protégé-2000 son usadas para resolver problemas y tomar decisiones en un dominio particular.

El sistema Protégé inicial creado para Windows (Protégé/Win), se comportaba como un sistema de bases de datos clásico, ya que definía de forma separada clases de información (esquemas) y almacenaba instancias de esas clases. Protégé-2000 hace más fácil el trabajar de forma simultánea con clases e instancias. Así, una instancia singular puede ser usada en el mismo nivel de la definición de una clase, y una clase puede ser almacenada como una instancia. Por su parte, los atributos, que inicialmente fueron empleados solo dentro de las clases, ahora son elevados al mismo nivel que las clases, y pueden tener significado sin pertenecer a alguna de ellas.

La herramienta Protégé-2000 proporciona una interfaz gráfica (GUI) para acceder a las diferentes partes de la información del dominio. Esta interfaz permitirá la integración de [4]:

1. El modelado de una ontología describiendo el modelo de un campo de conocimiento en particular.
2. La creación de una herramienta de adquisición de conocimiento (KA tool) para recopilar todo el conocimiento. Mediante formularios los expertos del dominio podrán introducir de forma sencilla su conocimiento.
3. La introducción de instancias específicas y la creación de una base de conocimiento (KB).
4. La ejecución de aplicaciones. Sistemas expertos o métodos de soporte a la decisión, utilizarán la información de la base de conocimiento.

Protégé-2000 fue desarrollado por el grupo *Stanford Medical Informatics* en la *Stanford University School of Medicine* con la ayuda de *National Library of Medicine*, *National Science Foundation*, y *Defense Advanced Research Projects Agency*.

1.2.3 Interfaz para el desarrollo de Plug-ins

El núcleo del sistema Protégé es una plataforma flexible en la que módulos externos pueden ser *conectados* y proporcionar una funcionalidad adicional [5]. Este mecanismo asegura que el sistema pueda ser adaptado según las necesidades.

Actualmente, Protégé 2.1.1. soporta cinco tipos de plug-ins: *slot widgets*, *tab widgets*, *storage backends*, *import-export plugins* y *project plugins*. En los siguientes apartados comentaremos cómo desarrollar cada uno de ellos, y se presentará un ejemplo de los dos primeros.

1.3 El Sistema de Base de Datos G

1.3.1 Base de Datos

G [4] es una plataforma basada en estándares abiertos especialmente diseñada para aplicaciones web. La principal estructura subyacente en esta framework es una base de datos orientada a grafos, que soporta XML y permite una flexibilidad de tipos y campos que no es posible en el modelo relacional clásico ni en el modelo orientado a objetos. Destaquemos que los nodos del grafo representarán descriptores y los arcos relaciones entre los mismos.

Una de las ventajas fundamentales de ser una base de datos orientada a grafos es que puede atender repositorios en XML, ya que un fichero XML no es más que un árbol y un árbol es un tipo especial de grafo.

1.3.2 Arquitectura

G parte de una arquitectura cliente-servidor sobre TCP/IP, utilizando como software de cliente el navegador y como servidor de aplicaciones un servidor web. Además G proporciona un software de Front End capaz de atender distintas aplicaciones web, wap o cualquier otra tecnología programable por plantillas, de manera distribuida y sobre una base de datos orientada a XML.

A nivel de procesos, G está integrado por una serie de CGIs que pueden operar sin necesidad de ningún servidor, aparte del propio servidor web.

1.3.3 Modelo de datos

Los datos en G se organizan en registros de longitud variable; cada registro es un conjunto de pares clave-valor, de manera que cada registro contiene toda la información que le concierne: relaciones, descripción y valores de los campos. Esto nos permite evitar la utilización de tablas o caracterizaciones globales de un determinado conjunto de registros.

Los datos en G se agrupan por tipos, todos los tipos tienen un valor común en un determinado campo, el D.y. De esta forma todos los registros cuyo D.y sea cliente serán interpretados como clientes, aunque sus atributos no sean exactamente los mismos.

Los campos de un registro son de dos tipos:

- **Campos de usuario:** serán las propiedades definidas para un objeto. Para el ejemplo del cliente podrían ser: nombre, DNI, dirección, etc. En el proyecto a desarrollar, estos campos ganarán importancia en la inserción de instancias de la base de conocimiento de Protégé.
- **Campos de metainformación o campos de código:** estos campos están directamente involucrados en el modelo de dato de G. Todos los campos que G interpreta para desarrollar la aplicación comienzan por la letra 'D', de esta forma reservamos un espacio de nombres y evitamos colisiones con otras variables. Algunos de ellos se comentan a continuación:
 - *D.y*: indentificador de tipo y supertipo, o nombre de la clase. Entre los diferentes valores que puede tener este campo, destacamos una serie de tipos predefinidos que permiten el almacenamiento y consulta de ontologías en G [6]:
 - *gOntology*: esta categoría no tiene instancias, y gracias a ella podremos insertar una ontología dentro de otra dando la referencia a un concepto raíz de ésta.
 - *gConcept*: esta categoría representa todos los conceptos de la ontología.
 - *gEnumeration*: nos permitirá definir listas de valores sin necesitar instancias.
 - *GProperty*: representa todas las propiedades de la ontología.
 - Para el caso de las instancias, el *D.y* será el nombre de la clase que se está instanciando.
 - *D.u*: identifica el propietario de un registro, que será el único con privilegios para modificar o borrar contenidos.
 - *D.gu*: identifica el grupo al que pertenece el registro.
 - *D.k*: índice de referencia autonumérico en al base de datos, es único.
 - *D.gid*: concatenación de *D.y* y *D.k* que sirve como identificador persistente en movimientos entre distintas bases de datos.
 - *D.time* (año/mes/día): marca temporal con la fecha de creación del registro.
 - *D.time1* (año/mes/día): marca temporal con la fecha de modificación o visualización del registro.

2. FUNCIONALIDAD AUXILIAR DEL PLUGIN

En este apartado se describen las principales funcionalidades del plugin⁵ desarrollado: conexión con servidor G, exportación/importación de ontologías, save/load, etc.

2.1 G Client Library for Java

Esta librería nos proporciona la interfaz necesaria para realizar diferentes operaciones sobre la base de datos G (conexión, consultas, inserciones, etc.) [7]. El servidor G posee un demonio esperando peticiones en el puerto 8910, con el que nos conectaremos por medio del protocolo TCP/IP.

En la figura 2 podemos apreciar el diagrama UML con las clases presentes en la librería.

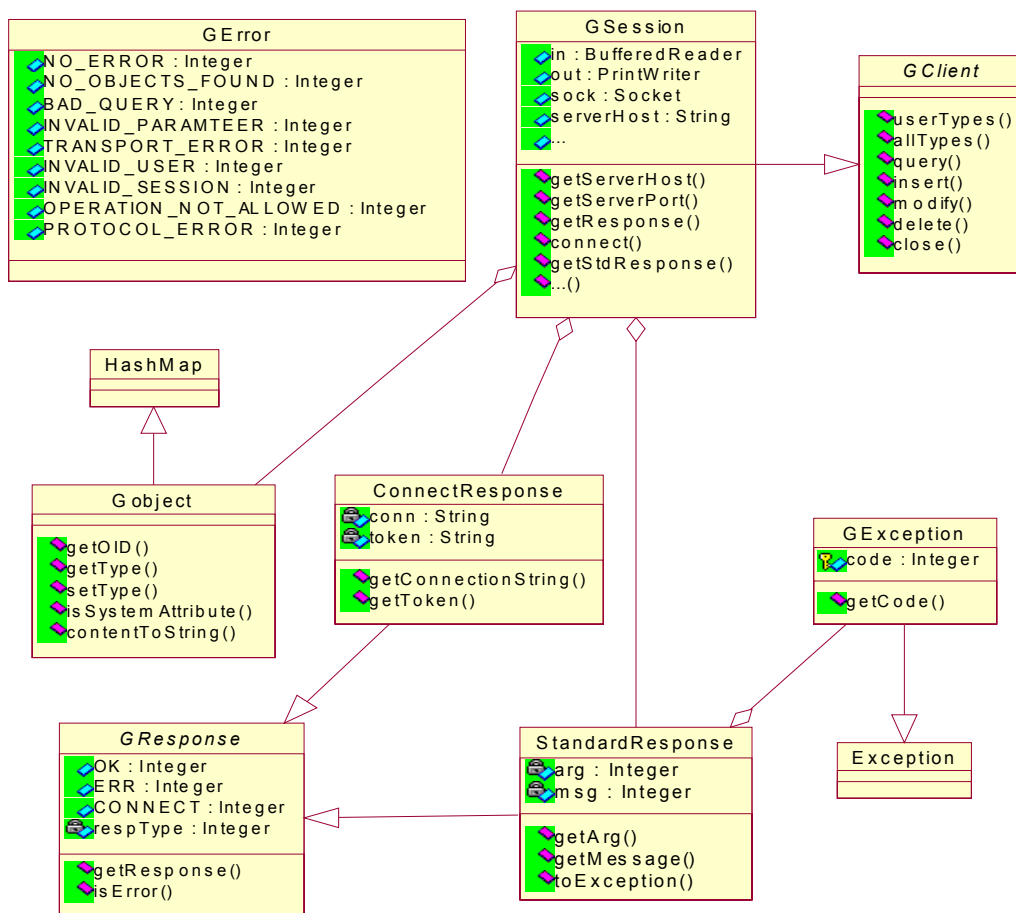


Figura 2. Diagrama UML de la librería para el cliente G.

⁵ La implementación del G Backend plug-in para Protégé está disponible en el url: <http://www3.uji.es/~ejimenez/GBackendPlugin.html>

2.2 Definición de un Esquema de Datos

La falta de esquema permite al sistema G almacenar cualquier tipo de datos irregulares, que aparecen frecuentemente en aplicaciones basadas en la Web (por ejemplo descripciones RDF, datos generados en formato XML, etc.). Sin embargo, un esquema puede ser útil para ayudar a los usuarios a expresar sus consultas, y conocer el contenido de la base de datos. Por esta razón, G provee un modulo que permite obtener un esquema actualizado de la base de datos [9].

Además para soportar la definición del esquema conceptual de una ontología en G se propusieron en [8] unas categorías que darán lugar a un meta-esquema. Estas categorías han sido extendidas en este trabajo para facilitar la recuperación de los diferentes objetos en las vistas (ver Tabla 1).

| D.y | Atributos | Comentario |
|------------------------|--|---|
| gOntology | name | Nombre de la ontología |
| | gConcept-OID | Referencia a un concepto raíz en el grafo |
| | nameSpace | Identificativo del espacio de nombres de la ontología |
| gConcept | name | Nombre del concepto |
| | Is-a | Super-concepto(s) del concepto |
| | role | Clase abstracta o concreta |
| | parentOID | OIDs de los super-concepto(s) del concepto |
| | types | Tipos en G asociados al concepto |
| gProperty | name | Nombre de la propiedad |
| | domain | Nombre del concepto fuente |
| | domainOID | OID del concepto fuente |
| | range | Nombre del concepto objetivo |
| | rangeOID | OID del concepto objetivo |
| | Is-a | Super-propiedad de la propiedad |
| | minOccur | Cardinalidad minimo de la propiedad |
| | maxOccur | Cardinalidad maximo de la propiedad |
| | maxValue | Valor máximo para propiedades de tipo numérico |
| | minValue | Valor mínimo para propiedades de tipo numérico |
| | transitive | Booleano indicando si la propiedad es transitiva |
| | nameTransitive | Conjunto de sinónimos para la propiedad transitiva |
| | enumerationOID | OID del objeto gEnumeration |
| attributes | Atributos utilizados en G asociados a la propiedad | |
| gEnumeration | name | Nombre de una lista de valores |
| | list-of-values | Lista de valores |
| Types/Instances | Instance-of | OID del concepto del que es instancia |
| | nameClass | Nombre original de la clase de la que es instancia |

Tabla 1. Representación de una ontología en G

En la figura 3 se puede apreciar con mayor claridad las diferentes relaciones entre las categorías del meta-esquema:

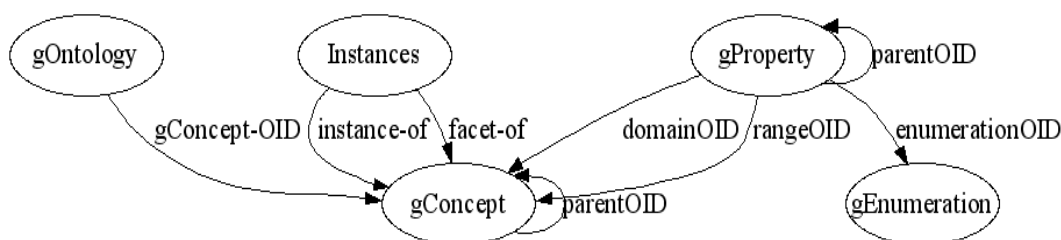


Figura 3: Meta-esquema de una ontología en G.

2.2.1 Creación de Slots del Sistema Auxiliares

Para cargar en Protégé información propia de G (OIDs, transitividad, types, attributes, etc.) es necesaria la creación de una serie de slots del sistema. Los slots del sistema son los que Protégé utiliza para la creación de ontologías (Name, Documentation, Role, etc.).

Estos slots se implementarán como *Standard Slots* de Protégé, mediante los siguientes pasos:

- **Creación de la clase *GSystemFrames*.** Esta clase nos creará los slots del sistema (son clases), como subclases de *StandardClsMetaCls* y *StandardSlotMetaCls* respectivamente. Además, para permitir el acceso al slot debemos almacenarlo en una estructura de datos (*Hashtable* o *HashMap*). En esta clase también se implementa la interface ID (para seguir la misma notación que Protégé), en la que se asigna un identificativo numérico (mediante *FrameID*) a estos slots, que en Protégé serán tratados como *frames*:

```
createSlot(FrameID id, String name, String parent, ValueType tipo, int card)
```

El primer argumento es el ID del *frame*, es necesario reservar un espacio para el nuevo slot a insertar, el segundo argumento es el nombre del slot, el tercero nos indicará si hereda de *StandardClsMetaCls* o *StandardSlotMetaCls* (según sea un slot para clases o para propiedades). El tercer argumento indicará el tipo, destaquemos que los slots que almacenen OIDs serán del tipo *any* y se les asociará el widget *GOIDTextFieldWidget* que no podrá ser editado y tendrá un botón de información. El último argumento indica la *cardinalidad*, que en el caso del slot para los nombres transitivos será tres (múltiple), y tendrá un *list widget* que soportará varios valores (Protégé se lo asigna automáticamente). Destaquemos que la recuperación de información y la actualización de este widget variará.

Los métodos *getXXXSlot* de *GSystemFrames* (ej.: *getAttributesSlot()*, *getTypesSlot()*) nos permitirán obtener el slot (frame) que queremos actualizar o recuperar.

- **Cambios necesarios otras clases.** El añadir nuevos slots implicará cambios en las siguientes clases:

- o *GResource*: deberemos añadir los nuevos atributos
- o *ProtegeFrameCreator*: en *CreateCls* y *Createslot* deberemos actualizar el valor de los nuevos campos.

Ejemplo slot cardinalidad simple:

```
((DefaultKnowledgeBase)_kb).
setDirectOwnSlotValue(slot, GSystemFrames.getOIDsSlot(), oid);
```

Ejemplo slot cardinalidad múltiple:

```
if (nameTransitive!=null) {
    List values = new
    ArrayList(((DefaultKnowledgeBase)_kb).getDirectOwnSlotValues(slot,
    GSystemFrames.getNameTransitiveSlot()));
    for (Iterator nameIt = nameTransitive.iterator(); nameIt.hasNext();){
        values.add(nameIt.next());
    }
    ((DefaultKnowledgeBase)_kb).setDirectOwnSlotValues(slot,
    GSystemFrames.getNameTransitiveSlot(), values);
}
```

- o *ProtegeFrameWalkerGraph*: en *walkClasses* y *WalkSlot* deberemos recuperar el valor de los nuevos campos.

Ejemplo slot cardinalidad simple:

```
String types=(String)
kb.getDirectOwnSlotValue(cls,GSystemFrames.getTypesSlot());
```

Ejemplo slot cardinalidad múltiple:

```
Collection nameTransitive = _kb.getDirectOwnSlotValues(slot,
GSystemFrames.getNameTransitiveSlot());
```

- o *GCreatorGraph*: deberemos introducir nuevos campos y su valor en G (si nos interesa)
- o *GOntologyFrameWalker*: deberemos recuperar valor de nuevos campos al construir el recurso en cuestión.
- o *GSchemaFrameWalker*: deberemos recuperar valor de nuevos campos (*getGmodel*).
- o *GIVFrameWalker*: deberemos recuperar valor de nuevos campos.

2.3 Project Plug-in

Los plug-ins de este tipo poseen métodos que pueden ser llamados a lo largo del ciclo de vida de Protégé y permiten modificar los proyectos y la interfaz de usuario. Para nuestro plug-in se ha utilizado los siguientes casos:

- Para crear un botón en la barra de herramientas que llamará al plugin *GExport*. Para ello es necesaria la construcción de una clase que extienda *AbstractAction*, y como acción construya un objeto *GExport* y llame a su método *'handleExportRequest'*.
- Para controlar la creación de los *GSystemFrames* para proyectos que no sean *G Backend*, *G Import* o *G Export*.
- Para crear una opción una nueva opción en el menú *Project* que nos permitirá acceder al formulario de eliminación de ontologías.
- Para crear una opción en el menú *Edit*, en el caso de importar un *G Schema*, que modificará os nombres de las clases y propiedades importadas.

2.4 Eliminación de Ontologías

Para la eliminación de ontologías deberemos ir a la opción del menú *Project*, *Delete G Ontology* que tras conectarnos a un servidor de ontologías nos mostrará las diferentes ontologías disponibles.

Para eliminar las ontologías se parte de un *gontology* y se va recorriendo todo el grafo de la ontología. Destaquemos que el procedimiento es análogo al utilizado en el plug-in *GImport* para cargar una ontología.

2.5 Fichero Import.log

Este fichero será simplemente un fichero auxiliar, para controlar qué OIDs se han cargado en Protégé, permitiendo controlar si algún objeto se eliminó. En la clase *'GOntologyFrameWalker'* construiremos el fichero y en la clase *GExport* leeremos del fichero si se realizó una importación previa, es decir, si slot *'OID'* de clase *:THING* contiene información. Además también almacenaremos información acerca del servidor G, del usuario y de la base de datos, y así controlar dónde se deberá realizar la exportación. También será útil saber si los datos importados corresponden a una vista, y si la exportación está permitida (no se realizaron cambios irreversibles en la vista). El fichero tendrá el siguiente formato:

```
# Ontology:MuesoReduced
# server:tempus.dlsi.uji.es
# data base:Without a Specific Data Base
# user:guest
# vista:false
# allow Export:true
# date:18/04/2005
486
485
487
489
502
505
...
```

2.6 Historial de servidores y bases de datos G

El almacenar en ‘GDataBases.log’ y en ‘GServers.log’ el historial de las diferentes bases de datos y servidores G, respectivamente, nos permitirá poder recuperarlos en el diálogo de conexión y facilitar la entrada de datos al usuario. El formato de los ficheros es bastante sencillo, como se aprecia a continuación:

```
# History Data Bases
Without a Specific Data Base
gprotege

# History Servers
tempus.dlsi.uji.es
yoshimi.act.uji.es
tictac.dlsi.uji.es
deva.act.uji.es
```

2.7 Espacio de nombres

Mediante la definición de un espacio de nombres (*namespace*) evitaremos conflictos de nombres al cargar varias ontologías en Protégé. Los objetos *gontology* de G tendrán un *item* con clave *namespace*, en el se definirá el espacio de nombres de la ontología. Al salvar y exportar podremos definirlo, por defecto es “http://protegeG.tkgb.uji.es/” + nombre_ontología + “#”. Al importar/cargar estableceremos el *namespace* de la ontología mediante el método *setNamespaces(String namespace)* de la clase *ProtegeFramecreator*.

Destaquemos que el espacio de nombres para clases y slots del sistema es: "http://protege.stanford.edu/system#"

2.8 Restricciones de una Clase

Las clases pueden tener asociadas instancias de la clase del sistema *:PAL_CONSTRAINT*, como restricciones. Cada clase almacenará los nombres de las restricciones (ítem *facets*), las restricciones almacenarán el campo *facets-of* el conjunto de OIDs de las clases de la que son restricción. Esto implicará que al insertar una clase con restricciones se mantenga un diccionario (tabla hash) con información acerca de la restricción en cuestión y el OID de la clase insertada (este diccionario se actualizará en el método *instGConcept* de *GCreatorGraph*). Pasos para la implementación:

- Protégé → G:

- Método *ProtegFWalkerGraph.walkClasses*: sacamos colección de instancias *constraint* y se lo pasamos a *createCls* de *GCreatorGraph*.
`Collection Constraints = _kb.getDirectOwnSlotValues(cls, GSystemFrames.sysFrame.getConstraintsSlot());`
- Método *GCreatorGraph.instGConcept*:
 - Insertamos en objeto *gconcept* los diferentes nombres de las instancias restricción.
 - Creamos o modificamos estructura con pares (*) (nombre_facet, setOIDs). Debemos almacenar el conjunto de OIDs de clases que hacen referencia a la restricción.
- Método *GCreatorGraph.createInst*: si nombre de instancia está en estructura (*) entonces almacenamos en campo *facet-of* los oids de las clases

- G → Protégé:

- Método *GOntologyFrameWalker.contructClass*: crea colección de recursos instance para las restricciones (hay que modificar la clase *GResporuce* para los conceptos, para que también almacene restricciones).
- *GOntologyFrameWalker.extracClasses*: saca objetos de tipo *:Pal_Constraint* con el facet-of que incluya el OID de la clase, y construye la instancia asociada.

2.9 Nombres válidos para los tipos (D.y) de G

Debido a que el campo D.y de G tiene un conjunto restringido de caracteres permitidos, debemos controlar los diferentes nombres de las clases, ya que si alguna clase posee un carácter no permitido, el D.y de sus instancias dará un error. Solo se permiten caracteres alfanuméricos, y sólo alfabéticos al comienzo de la cadena

Para controlarlo creamos la clase *GValidNames*, que implementará el método estático: 'Protege2G'. Este método transforma los D.y de las instancias al correspondiente formato permitido en G.

Los caracteres conflictivos serán sustituidos por una cadena con el siguiente formato: "G" + código UniCode del carácter + "G". Por ejemplo, la cadena

“:PAL_CONSTRAINT” será sustituida por “58gpalg95gconstraint”. Destaquemos que la cadena también pasa a minúsculas, debido a cuestiones internas de G, como veremos en el siguiente punto.

Para que las instancias, cuyo D.y haya sufrido algún cambio, no pierdan referencia con la clase de la que son instancia deberemos almacenar un campo auxiliar con el nombre original de la clase, el de Protégé. Este campo será ‘nameClass’.

Métodos que usan GValidNames (Protégé → G):

- GDeleteOntology.deleteInstance
- GCreatorGraph.createInstance
- GOntologyFrameWalker
 - .extracClasses
 - .extractInstances
- GIVFrameWalker
 - .extracClassFacets
 - .extracInstances

Por su parte, los métodos GOntologyFrameWalker.constructInstance y GIVFrameWalker.constructInstance (G → Protégé), obtendrán el valor original de la clase mediante el campo *nameClass*.

2.10 Tipos y claves de los campos en minúsculas

Debido a cuestiones internas de G los nombres de los diferentes tipos (D.y) y de las claves de los campos de cada objeto pasan a minúsculas al ser insertados. Esto plantea un ligero problema al trabajar con Protégé debido a que éste sí distingue entre minúsculas y mayúsculas.

Si no se realiza ningún cambio respecto a la primera versión del plug-in, las claves de las instancias pueden perder la relación con los nombres de las propiedades originales, ya que las primeras habrán pasado a minúsculas y las segundas podrán contener mayúsculas. Para evitar cualquier tipo de problema se decidió pasar todos los slots de protégé, de forma automática, a minúsculas (*GExport.exportProject*). De esta forma al insertar tanto claves de instancia, como nombres de propiedad estarán en minúsculas, y no se perderá la relación entre ellos.

El único problema que puede surgir es si se produce una coincidencia de nombres entre clases y propiedades, al pasar estas últimas a minúsculas, en tal caso se emitirá un mensaje de error y se interrumpirá la exportación.

3. PAQUETES BACKEND - IMPORT – EXPORT

| Packages | |
|---|--|
| <u>es.uji.tkgb.Protege G</u> | Contendrá las classes base para permitir la importación y exportación de ontologías |
| <u>es.uji.tkgb.Protege G.backend</u> | Importara y Exportará ontologías creando un proyecto de Protégé y un fichero auxiliar. |
| <u>es.uji.tkgb.Protege G.deleteOntology</u> | Paquete para borrar ontologías. |
| <u>es.uji.tkgb.Protege G.helide.gclient</u> | Este paquete forma parte de la librería G Client desarrollada por el profesor Ismael Sanz Blasco (TKBG at Jaume I University). Provee definiciones genéricas (G Objects, G Exceptions, etc) útiles para los metodos que accedan a la Base de Dato G. |
| <u>es.uji.tkgb.Protege G.helide.gclient.sockets</u> | Este paquete también forma parte de la librería G Client. Implementa el acceso a G via sockets |
| <u>es.uji.tkgb.Protege G.ontologyImportExport</u> | Nos permitirá importar y exportar ontologias de/a la base de datos semiestructurada G |
| <u>es.uji.tkgb.Protege G.ontologyViewImport</u> | Este paquete nos permitirá importar y exportar porciones de ontologias de/a G mediante el lenguaje OntoPathView |
| <u>es.uji.tkgb.Protege G.ontoPathParser</u> | Contiene las clases necesarias para implementar un sencillo parser para el lenguaje de definición de vistas OntoPathView |
| <u>es.uji.tkgb.Protege G.schemaImport</u> | Paquete para importar Esquemas de bases de datos en G |
| <u>es.uji.tkgb.Protege G.walker</u> | Interfaces útiles para cargar y salvar ontologias desde Protege. By "Stanford Medical Informatics". |
| <u>es.uji.tkgb.Protege G.walker.protege</u> | Este paquete también ha sido desarrollado por el grupo "Stanford Medical Informatics"; pero ha sido ligeramente modificado para las necesidades de este plugin. La diferentes clases de este paquete proporcionan un eficiente acceso a Protégé, tanto para cargar datos de la nueva ontología como para consultarlos. |

4. PLUG-INS IMPORT Y EXPORT

Con este nuevo plug-in insertaremos ontologías en G y las recuperaremos sin tener un control total de los D.k de la misma. La principal motivación para el cambio respecto a la metodología propuesta en [10] fue la necesidad de realizar vistas o proyecciones, según las necesidades, de una ontología compleja en la que no interese trabajar con toda la jerarquía de clases. En este caso no interesa tener un control de todos los OIDs de la ontología si no poder tener una visión de grafo de la misma, partiendo siempre de un nodo raíz.

Para hacer factible este plug-in fueron necesarias varias modificaciones en las estructuras de datos de protégé (creación de GSystemFrames) y en las de G. Debido a que el espacio de nombres de G no es único, identificar objetos por nombre (como ocurre en Protégé) puede resultar ambiguo, si tenemos varios conceptos o propiedades con el mismo nombre. La identificación debe realizarse mediante los OID con lo que fue necesario introducir los nuevos campos, como ya se vió en la sección anterior).

- Para **gProperties**: domain-OID, enum-OID, parent-OID y rangeOID
- Para **gConcepts**: parent-OID
- Para **instancias/tipos**: instante-of
- Para **gOntology**, el campo 'name' debe contener el nombre de la ontología más el OID del nodo raíz, así poder identificar ontologías con el mismo nombre.

4.1 Algoritmo de ordenación topológica

Una vez creadas las estructuras de datos fue necesario implementar un algoritmo que permitiese insertar en G los conceptos, slots e instancias en orden, ya que hay referencias a sus OIDs. Se utilizó el algoritmo de ordenación topológica para la inserción de la jerarquía de clases y también para la de slots.

El algoritmo de ordenación topológica, *Topological Sort*, genera una ordenación lineal de todos los vértices de un grafo **dirigido** y **acíclico** tal que si $(u,v) \in E$, entonces u aparece antes que v en la ordenación.

Ejemplo : Para el grafo que se presenta en la siguiente figura

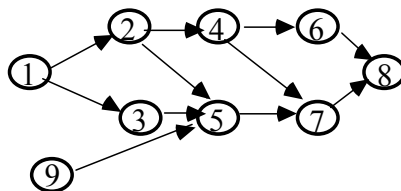


Figura 4: Grafo de ejemplo.

- Una ordenación es $O_1 = \{ 1, 2, 4, 3, 9, 5, 7, 6, 8 \}$
- Otra es $O_2 = \{ 9, 1, 3, 2, 5, 4, 6, 7, 8 \}$.

Algoritmo: Un vértice u sólo puede aparecer en la secuencia cuando todos sus predecesores ya están en la secuencia \Rightarrow Insertar el vértice en algún momento posterior a la entrada del último de sus predecesores en la secuencia.

Implementación hábil: Llevar un contador para cada vértice que indique cuántos de sus predecesores faltan por aparecer en la secuencia. En el momento en que ese contador sea cero, el vértice en cuestión ya está en condiciones de aparecer.

```

función ORD-TOPO ( g es grafo )
    dev ( s es secuencia ( vértices ) )
{ Pre : g es un grafo dirigido y sin ciclos }
    c := conj_vacio ;
    Para cada v ∈ V hacer
        NP[v] := 0;    c := añadir( c, v );
    fpara

    Para cada v ∈ V hacer
        Para cada w ∈ suc(g,v) hacer
            NP [w] := NP [w] + 1;
            c := eliminar( c, w );
        fpara
    fpara

/* para cada v ∈ V, NP[v] contiene el número predecesores que tiene v en g y c es el
conjunto que contiene los vértices que tienen cero predecesores. El recorrido puede
comenzar por cualquiera de ellos */
    s := sec_vacia;
    Mientras ¬vacio ( c ) hacer
        v := obtener_elem( c );
        c := eliminar( c, v );
        s := concatenar( s, v );
        Para cada w ∈ suc(g,v) hacer
            NP[w] := NP[w] -1;
            Si NP[w] = 0 entonces
                c := añadir( c, w );
        fpara
    fmientras
{ Post : s = orden_topológico( g ) }
    dev ( s )
ffunción

```

Implementación Java:

```

/* Devuelve un vector con las clases ordenadas topologicamente.
 * @param classes Grafo con la jerarquía de clases
 * @return Vector ordenado */
Object[] topologicalOrder_Cls(Collection classes){

```

```

Cls cls;      Cls subCls;
Collection subClasses;
Integer i;
int pred;
Object[] sequence = new Object[classes.size()];
//Inicializamos tablahash de predecesores. Cada componente contendrá
//el numero de predecesores asociados a clase. Pares clase - n_predecesores
Hashtable predecessors = new Hashtable();
//Mantenemos conjunto con las diferentes clases con num. de predecesores=0
HashSet vertex = new HashSet();
for (Iterator classIterator = classes.iterator(); classIterator.hasNext();) {
    cls = (Cls) classIterator.next();
    predecessors.put(cls.getName(), new Integer(0));
    vertex.add(cls);
}
for (Iterator classIterator = classes.iterator(); classIterator.hasNext();) {
    cls = (Cls) classIterator.next();
    subClasses = cls.getDirectSubclasses();

    for (Iterator subClassIterator = subClasses.iterator();
         subClassIterator.hasNext();) {
        subCls = (Cls)subClassIterator.next();
        i = (Integer)predecessors.get(subCls.getName());
        predecessors.put(subCls.getName(), new
            Integer(i.intValue()+1));
        vertex.remove(subCls);
    }
}
int j=0;
while (!vertex.isEmpty()){
    cls = (Cls) vertex.iterator().next();
    vertex.remove(cls);
    sequence[j]=cls; //Si ta en cjto es u eno tiene predecesores
    j++;
    subClasses = cls.getDirectSubclasses();
    for (Iterator subClassIterator = subClasses.iterator();
         subClassIterator.hasNext();) {
        subCls = (Cls)subClassIterator.next();

        pred=((Integer)predecessors.get(subCls.getName())).intValue()-1;
        predecessors.put(subCls.getName(), new Integer(pred));
        if (pred == 0) //Añadimos a conjutnto
            vertex.add(subCls);
    }
}
return sequence;
}

```

4.2 Eliminación de Objetos

Al realizar la exportación leemos del fichero *import.log* y creamos vector con oids de la ontología inicial (o de la vista inicial). También crearemos un vector con los oids de los objetos de la ontología (o vista) actual, según se vayan insertando.

Una vez insertada la nueva ontología compararemos ambos vectores y borraremos los objetos no usados, en el método *GCreatorGraph.deleteOID*.

4.3 Modificación de objetos (problemas con version-ID)

Al modificar un objeto se almacena una copia de seguridad, la idea es preguntar al usuario si se quiere guardar una copia de toda la ontología almacenada o no. Solicitaremos si se desea almacenar copia de seguridad, antes de mostrar barra de progreso, y si estamos ante la modificación de una ontología. Mostraremos un diálogo que se creará en *ProtegeFrameWalker*, método *SafetyCopyDialog*, y será llamado en el método 'walk'.

En *GCreatorGraph* implementaremos el método *safetyCopyManagement* al que le pasaremos el OID del objeto modificado, y según la opción elegida por el usuario, la copia de seguridad será eliminada o se insertará en algún fichero auxiliar, el cual nos permitirá la recuperación de la copia, y tratar versiones de una ontología

Búsqueda de elementos copia por campo 'version-ID', el cual tiene la siguiente estructura: D.y + D.k (objeto modificado) + "-" + versionObjetoModificado-1; por ejemplo: [gConcept6825-3](#). El problema es que no encuentra objetos buscando por este campo. Al realizar consultas, sólo obtenemos objetos de las versiones más actuales.

5. FUNCIONALIDAD COMO BACKEND PLUG-IN

Finalmente el plugin G – Protégé ha dejado de ser dependiente de un fichero de OIDs, y el plugin de Backend se ha convertido en una extensión de los Plugins Import-Export. Este plugin nos creará un nuevo proyecto permitiendo una total integración entre G y Protégé. Junto al fichero .pprj se almacenará un fichero auxiliar .log (no confundir con ‘import.log’, que es general), en el que almacenaremos datos acerca de la ontología almacenada en G (servidor, base de datos, usuario, OID ontología). El permitir la creación de un proyecto G – Protégé nos proporcionará comodidad (carga de ontología más rápida: open) y compatibilidad con otros plugins (ej. PROMPT) que requieren el fichero .pprj.

5.1 Proceso *Save*

Se comportará de forma semejante al GExport plugin, de hecho es básicamente una extensión del mismo. La única diferencia es que se creará el fichero auxiliar .log con el siguiente formato: (MuseoLibroGTempus2.log)

```
# Ontology:MuseoLibroGTempus2
# server:tempus.dlsi.uji.es
# data base:testDB
# user:guest
# oidOntology:7482
```

Destaquemos que este fichero lo creará el método estático ‘createLogProject’ de ‘GExport’, el cual será llamado desde el método ‘instGOntology’ de ‘GCreatorGraph’, tras obtener *oid* de la ontología (del objeto *gOntology*).

5.2 Proceso *Load*

Al igual que el proceso ‘Save’, el proceso ‘Load’ extiende la funcionalidad del plugin GImport permitiendo cargar directamente una ontología a partir del fichero de proyecto .pprj y del auxiliar .log, sin tener que seleccionarla.

El método ‘loadWalk’ de la clase ‘GBackend’ leerá del fichero .log auxiliar, el servidor, la base de datos, el usuario y el oid del *gOntology*, tras ello llamará al método ‘handleLoadRequest’ de la clase ‘GImport’, que se encargará de realizar la conexión con G, crear los frames del sistema necesarios y crear un objeto de la clase ‘GOntologyFrameWalker’. A partir de este momento el plugin se comportará exactamente igual que el GImport.

6. PLUG-IN PARA LA REALIZACIÓN DE VISTAS

Este plugin es una extensión del Export-Import, en el que podremos realizar consultas de una ontología y traernos la parte de ella que nos interese. Para la realización de las consultas nos hemos basado en el lenguaje *OntoPath* [15], con lo que ha sido necesario especificar una gramática para poder parsear las consultas. Las consultas en *OntoPath* se definen como caminos sobre el grafo de la ontología. Una consulta siempre parte de un nodo raíz, y es una secuencia alternada de conceptos y propiedades.

6.1 Gramática del Lenguaje *OntoPathView*

Debido a los problemas, con los caracteres no alfanuméricos (“ ”, “\t”, “+”, etc.) en los nombres de los conceptos y de las propiedades, es necesario que estos identificativos y los posibles valores de los slots se introduzcan entre comillas al escribir la consulta.

La sintaxis del lenguaje *OntoPathView* se define mediante la siguiente gramática BNF, donde *IdentOrValue* and *Number* están expresados por medio de las siguientes expresiones regulares: `" ([^\n "])+ "` y `[1-9] ([0-9])*`, respectivamente:

| | | |
|-------------------------------|---|--|
| <code><View></code> | → | <code>(<Query>)⁺</code> |
| <code><Query></code> | → | <code><Concept> (<Modifier>)? (<Slots>)? (eof '\n')</code> |
| <code><Concept></code> | → | <code>([' IdentOrValue ']) IdentOrValue</code> |
| <code><Modifier></code> | → | <code>('+' '*' '-') (Number) ?</code> |
| <code><Slots></code> | → | <code>' / ' { ' ((<Slot> (',' <Slot>)[*]) '?') ' }</code> |
| <code><Slot></code> | → | <code>IdentOrValue (<Depth> <Filter> <Slots>)?</code> |
| <code><Depth></code> | → | <code>' + ' ' (? Number ')'</code> |
| <code><Filter></code> | → | <code>(<OpsSome> <OpsAll>) (IdentOrValue)</code> |
| <code><OpsSome></code> | → | <code>' = ' '>=' '<=' '>' '<' '<>'</code> |
| <code><OpsAll></code> | → | <code>' ? = ' '?>=' '?<=' '?>' '?<' '?<>'</code> |

Las expresiones construidas con esta gramática parten de un concepto para definir la porción de la ontología que constituye una vista. A continuación, explicamos las expresiones más usuales que se pueden construir:

- **“concept”**: La consulta extrae el concepto partida y sus slots directos de tipo literal (clausura con reducción). Este tipo de consulta no extra instancias, solo clases y propiedades.
- **[“concept”]**: Los corchetes indican que las instancias también serán extraídas
- **“concept”+n**: La consulta también extrae subclases (el parámetro opcional *n* indica el número de niveles a extraer). Con ‘-’ en lugar de ‘+’ se extraen las superclases. Y con ‘*’ se extraen tanto subclases como superclases.
- **“concept”/{"slot₁", ..., "slot_n"}**: Se extrae el concepto partida con los slot especificados. Si alguno de los slots relaciona el concepto partida con otro concepto, también se extrae este concepto junto con sus slots directos de tipo literal.

- [{"concept"}]/{"slot₁"="value", ...}: Para filtrar las instancias del resultado, se asocia un valor y un operador a las propiedades.
- "concept"/{"slot₁"+, ...}: Extrae recursivamente los conceptos relacionados con *concept* a través de la propiedad *slot₁*.
- "concept"/{"slot₁"(n),...}: Igual que la anterior pero limitando la profundidad a *n* niveles.
- "concept"/{"slot₁", "slot₂"/{"slot₂₁", "slot₂₂"}, "slot₃"}: En este tipo de consulta el *slot₂* expande el grafo de la consulta. El rango de *slot₂* será el nuevo nodo de partida.
- "concept"/{"slot₁", "slot₂"/{?} "slot₃"}: Esta consulta extrae el subgrafo completo con el rango de *slot₂* como nodo partida.

6.2 Implementación Parser: Java Compiler Compiler

Java Compiler Compiler [tm] (JavaCC [tm]) es un generador de *parsers* para usar en aplicaciones Java. Un generador de *parsers* es una herramienta que lee una especificación de una gramática y la convierte en un programa Java que funcionará como un *parser*.

En el fichero "OntoPathParser.jj" se encuentra la especificación léxica, la gramática y las acciones necesarias, que deberán ser compiladas con el 'javacc'. Tras la compilación se generarán los ficheros .java que darán lugar a nuestro *parser*. Destaquemos que estos ficheros deberán ser añadidos a nuestro proyecto para poder ser utilizados.

- **Parser:**
 - o OntoPathParser.java
 - o OntoPathParserConstants.java
 - o OntoPathParserTokenManager.java
- **Auxiliares:**
 - o TokenMgrError.java: tratará los errores léxicos
 - o Token.java
 - o SimpleCharStream.java
 - o ParseException.java: tratará los errores sintácticos

Un ejemplo de consulta sería el siguiente:

```
[{"Painter"}]*/{"lname"="Picasso", "paints"/*{"exhibited", "title"}, "fname"}
```

Partimos del concepto 'Painter' y recuperamos los atributos 'lname', 'fname' y 'paints', el atributo 'paints' al hacer referencia a otra clase se puede expandir extrayendo la clase destino ('Painting') y sus atributos 'tittle' y 'exhibited'. El atributo 'exhibited' hace referencia al concepto 'Museum' que también será extraído junto a sus slots atómicos. Como se han indicado los corchetes las instancias de las clases también serán extraídas. Para las instancias de 'Painter' (o de alguna subclase) se ha añadido un filtro de modo que su atributo 'lname' coincida con 'Picasso'. Destaquemos que las superclases y subclases del concepto de partida.

Datos del parseado:

```

Concepto: Painter, with Instances?: true
With Superclasses: true, Numero Sup.: -1
With Subclasses: true, Numero Sub.: -1
Slot: fname, level: 1
Slot: lname, level: 1, value: Picasso, operator: 1
Slot: paints, level: 1
Slot: exhibited, level: 2, origin: paints
Slot: title, level: 2, origin: paints

```

6.3 Implementación Plugin

El funcionamiento básico del plugin para la extracción de vistas se describe a continuación.

Operaciones para cada consulta (método getGQuery()):

- Clase destino de una propiedad siempre se extrae → no problemas con clases de frontera.
- Clase propietaria de una propiedad siempre se extrae → clase dominio siempre estará presente.
- Toda clase extraída tendrá también sus slots directos atómicos (tipo literal).
- Los filtros de una clase serán heredados por las clases hija.

Operaciones tras unión (método camina()):

- Si extraemos dos clases que deben estar relacionadas, también sacamos propiedad que las une. De forma directa o indirecta (heredada), si es indirecta también traeremos clases domain y/o range.
- Con la jerarquía de clases se realiza una reestructuración para mantener la consistencia.
- Para propiedades transitivas se realiza reestructuración, semejante a la realizada con la jerarquía de clases.
- Las instancias resultantes de la unión, contemplarán todas las propiedades disponibles.

6.3.1 Extracción Objetos de la Consulta

La clase principal del import plugin en GImportView a partir de su clase 'handleImportRequest', que se encargará de abrir la conexión con G, sacar las ontologías disponibles y mostrar el diálogo para seleccionar ontología e introducir las consultas. Cada consulta debe introducirse en una única línea, es decir, cada línea del cuadro de texto del interfaz se considerará una nueva consulta.

Una vez introducidas las consultas se procederá al parseado de las mismas, si se encuentra algún error léxico o sintáctico se mostrará el correspondiente mensaje de error y se permitirá la corrección. Si todo es correcto se continuará la ejecución del método 'handleImportRequest' el cual llamará al método 'importProject' que será el

encargado de crear un nuevo proyecto, de inicializar la base de conocimiento, de crear los frames del sistema para G, y finalmente creará un objeto de la clase GIVFrameWalker, y se llamará a su método 'walk'.

Los métodos de la clase GIVFrameWalker trabajarán con las diferentes consultas de forma independiente, sacando conceptos, slots e instancias que se almacenarán en un vector de objetos de la clase 'resultsView', una vez extraído todo se realizará una unión de todos los objetos y se analizará la inconsistencia y los posibles errores.

6.3.2 Extracción concepto de partida:

El primer paso de la consulta es extraer el concepto raíz de la misma. Para ello extraeremos todos los gConcepts cuyo nombre coincida con el de la consulta ('extractQueryConcept(String)'); y nos quedaremos con el que pertenezca a la ontología en cuestión.

Metodología: Para cada uno de los gConcepts cuyo nombre coincida con el de la consulta comprobaremos si pertenece a la ontología analizando recursivamente su padre hasta llegar al nodo raíz, que deberá ser el de la ontología seleccionada ('extractFather(String oids)').

6.3.3 Extracción de superclases y subclases:

Extraeremos de forma recursiva las clases padre y clases hija. La recursividad vendrá limitada por los conceptos 'hoja', por el concepto raíz y por el número de niveles a extraer. Además también extraeremos los slots directos y atómicos (extractConceptSlots(GObject)).

6.3.4 Extracción de propiedades

Si no se indicó ninguna propiedad para la clase de partida, se extraerán sus slots directos (atómicos) por defecto (extractConceptSlots(GObject)). Si se indicó el signo de interrogación "concept"/{?} se extraerá todo el grafo subyacente al concepto de partida. El método que se encargará de ello será "extractAllGraphClasses(String oid)".

Partimos de una clase origen (inicialmente será el concepto de partida de la consulta) y sacamos los slots correspondientes a su nivel (tabla hash 'slots') comprobando que son correctos para la clase (con 'checkSlot'). Si el slot pertenece a la clase:

1. Se sacarán posibles enumeraciones y se construirá el correspondiente recurso propiedad.
2. Si el slot tenía algún filtro asociado lo añadiremos a una tabla hash con pares (slot, valor), esta tabla a su vez irá asociada al nombre de la clase en cuestión en una nueva tabla hash con pares (clase, hash_slot_valor).
3. Si el slot tiene como rango una clase destino, la construiremos junto a sus slots directos de tipo literal:
 - a. Sacaremos clases agregadas si se indicó una profundidad (en 'slotsDeep'), y los slots directos de estas clases.

- b. Si el slot expande el grafo en la consulta pasaremos a analizar el siguiente nivel (llamada recursiva).
- c. Si expande grafo de la consulta pero se introduce el signo de interrogación: “concept”/{"slot"/{?}, ...}, se extraerá todo el grafo subyacente al concepto destino.
- d. En otro caso solo se extraerán slots directos de la clase asociada ya construida.

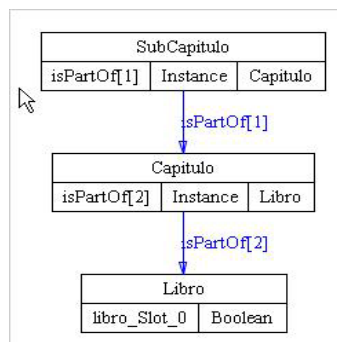
6.3.4.1 Funciones auxiliares:

- **checkSlot:** Comprueba si el slot de la consulta pertenece a su clase origen (slot directo) o a alguna de su padres (hereda slot). Destaquemos que si pertenece a alguna de las clases, seguro que es de nuestra ontología ya que el concepto de partida lo es.
- **extractAggregationClasses:** este método es semejante al extractSlots, pero solo tiene en cuenta la propiedad transitiva para extraer las clases agregadas/asociadas. Destaquemos que para cada clase construida también sacaremos sus slots directos.
- **extractAllGraphClasses:** A partir de un oid de una clase sacaremos de forma recursiva todas las clases sucesoras. Además también se extraerán los atributos (‘extractAllGraphSlots’). Destaquemos que este método se utilizará para consultas del tipo "concept"/{?} ó "concept"/{"slot"/{?}, ...} Si la clase ya fue tratado con anterioridad, no se hará nada.
- **extractAllGraphSlots:** Sacaa de forma recursiva los slots padre y los hijos. Si el slot ya fue tratado con anterioridad, no se hará nada

6.3.4.2 Nuevos slots del sistema:

- **Transitive:** será un checkbox asociado a las propiedades en el que indicaremos si la propiedad en cuestión es de tipo transitiva.
- **NameTransitive:** si la propiedad es transitiva indicaremos un nombre, que no tendrá que coincidir con el nombre del slot en Protégé. Al realizar la consulta podemos indicar indistintamente el nombre de la propiedad o el nombre de la propiedad transitiva que representa; esto será muy útil a la hora de realizar consultas extrayendo clases agregadas. El slot podrá llamarse ‘slot1’ y la propiedad transitiva aceptar los nombres ‘PartOf’, ‘isPartOf’ o ‘ContainedIn’. Ejemplos que generarán el mismo resultado:

- “SubCapitulo”/{"isPartOf"+}
- “SubCapitulo”/{"PartOf"+}
- “SubCapitulo”/{"ContainedIn"+}



6.3.5 Extracción de Instancias

Extrae las instancias de las clases contenidas en la estructura `_filtros`. Esta estructura estará formada por pares (`GObjectClass`, `ClassFilter`), donde `ClassFilter` será una nueva tabla hash en la que se almacenarán pares (`atributoG`, `valor_operador`), donde `valor_operador` es un vector de dos componentes: `valor_operador[0]` es el valor, y `valor_operador[1]` es el identificador de operador (ver tabla 2). La estructura `_filtros` se construirá a medida que vayamos extrayendo las clases de la consulta con los respectivos filtros. Destaquemos que como mínimo tendrán el filtro `instance-of=oid_clase`.

| Operadores algún valor de... | | Operadores para todo valor de... | |
|------------------------------|-----------------------|----------------------------------|-----------------------|
| <i>Símbolo</i> | <i>Identificativo</i> | <i>Símbolo</i> | <i>Identificativo</i> |
| '=' | 1 | '?=' | 7 |
| '<' | 2 | '?<' | 8 |
| '>' | 3 | '?>' | 9 |
| '<=' | 4 | '?<=' | 10 |
| '>=' | 5 | '?>=' | 11 |
| '<>' | 6 | '?<>' | 12 |

Tabla 2: Identificativos de operador

Filtros en clases hijas: en la consulta se permitirá añadir filtros a cualquier clase, abstracta o concreta, con instancias o sin ellas, y éstos serán heredados por las clases hija que, en general, serán las que tengan instancias asociadas. La herencia de filtros se realizará en el método `'addFilterToSubClasses()'`.

6.3.5.1 Comparación de Valores según Operador y Tipo.

En `extraInstances` extraemos de `G` todas las instancias de las clases de la consulta (vista), y de ellas solo nos quedaremos con las que cumplan el filtro. Par ellos se seguirán los siguientes pasos:

1. Miramos en vector de `Gobjects instances` si el atributo en cuestión si tiene múltiples valores.
 - a. Si valor único operadores 'para todo' y operadores 'existe' tiene el mismo comportamiento.
 - b. Si hay valores múltiples operadores 1-6 serán distintos a los 7-12.
 - i. El operador 'para todo' debe pasar el filtro a todos los valores de la lista.
 - ii. El operador 'existe' solo pasará el filtro hasta que algún elemento de la lista lo cumpla.
2. Tenemos que construir solo instancias que cumplan la restricción.

Utilizaremos método `compareValues` para tratar los diferentes operadores y tipos. Este método recorrerá la lista de valores (si hay múltiples valores), y dependiendo del tipo de datos realizará un tipo distinto de comparación, entre valor individual y filtro:

- Para tipos 'String', 'Any', 'Instance', 'Symbol' y 'Class', realizará una comparación de cadenas (método `compareStrings`)

- Para tipos 'Integer' y 'Float' realizará una comparación numérica (métodos *compareIntegers* y *compareFloats*).
- Finalmente para el tipo 'Boolean', solo utilizará operadores '=', '?=', '<>' y '?<>', ya que solo se compararán valores 'True' o 'False'.

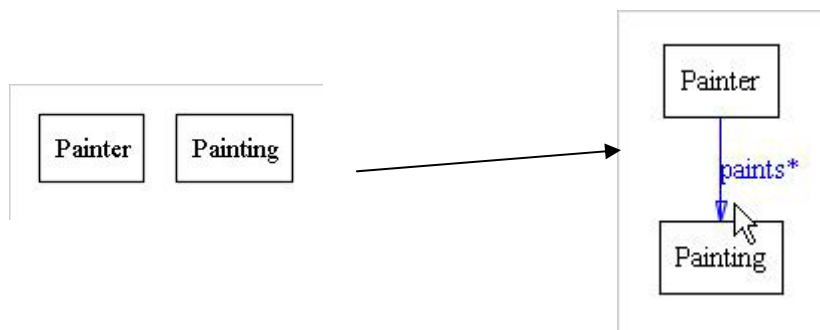
6.3.6 Unión de los Objetos de las Consultas

Deberemos recorrer el vector 'resultView', con los datos de cada consulta. La unión de los objetos será sencilla para 'classes' y 'slots' ya que simplemente deberemos ir añadiendo a las estructuras *_classes* y *_properties* respectivamente, los diferentes objetos de las diferentes consultas, y si alguno ya se insertó no volverlo a hacer. Para instancias será ligeramente más complicado ya que puede que en una consulta una instancia sea más completa que en otra, es decir, que tenga más pares (slot-valor). Si la instancia es nueva se insertará sin más; si ya se insertó se añadirán los nuevos pares (slot-valor) de la nueva instancia, si los hay.

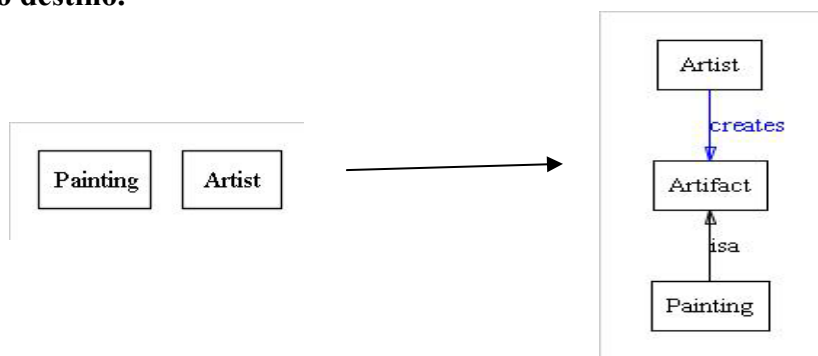
6.3.7 Tratamiento de las Relaciones de Asociación

Al realizar varias vistas puede que hayamos traído clases de forma independiente que estén relacionadas, de forma directa o indirecta, y la propiedad que las une no esté cargada. En este caso es interesante traer ésta propiedad, y las clases propietaria y/o destino si la relación no es directa (heredada). Casos posibles:

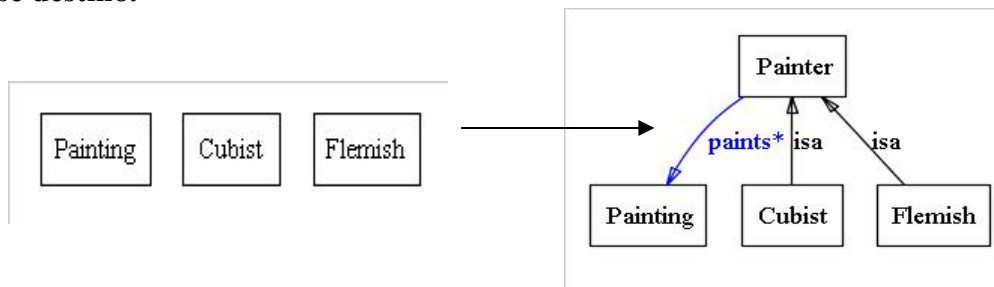
Caso 1: Relaciones directas, están clases dominio y destino



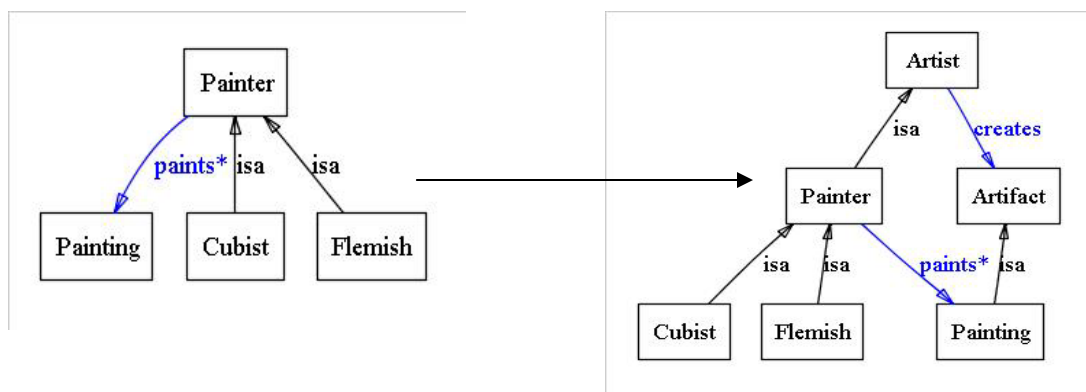
Caso 2: Relación indirecta 1, está clase dominio y también están hijas del concepto destino.



Caso 3: Relación indirecta 2, no está clase dominio, pero si conceptos hijo, y está clase destino.



Caso 4: Relación indirecta 3, no está ni clase dominio ni clase destino, pero sí que hay hijas de ambas.



6.3.7.1 Implementación

El método principal es **'inferredAllAssociationRelations'**, es un método recursivo que será llamado para cada una de las clases que hayamos insertado, y se encargará de sacar propiedades de tipo Instante de la clase en cuestión y de las superclases. Para cada propiedad extraída que no haya sido ya insertada, se buscarán los conceptos dominio y destino, y según estén ya insertados en **'_classes'** o no estaremos ante un caso u otro, de los vistos en el punto anterior. Utilizaremos una tabla hash para almacenar los pares (objetos propiedad – caso de asociación).

Una vez extraídos todos las posibles relaciones, se utilizará el método **'treatPossibleAssociationRelations()'**. Este método recorrerá las posibles relaciones extraídas y realizará las acciones pertinentes según el caso:

- **Caso 1:** construirá la propiedad sin más.
- **Caso 2:** comprobará si la clase destino de la propiedad tiene conceptos hijo insertados en **'_classes'** (uso de método **'haveSubClasses'**), en caso positivo insertará la propiedad y la case destino.
- **Caso 3:** comprobará si la clase dominio de la propiedad tiene conceptos hijo insertados en **'_classes'**, en caso positivo insertará la propiedad y la case dominio.

- **Caso 4:** Realizará comprobación de subclases tanto para clase dominio como para clase destino, si ambas tienen, se construirán estas clases y también se construirá la propiedad.

6.3.8 Inferencia de la nueva Jerarquía de Clases

El concepto 'range' siempre va estar, ya que nos traemos siempre la clase destino y de ella solo sus propiedades directas de tipo literal. El concepto domain o propietario de un slot también nos lo traeremos. El problema vendrá con los conceptos is-a de las clases, que pueden faltar. Será necesario reestructurar la jerarquía de clases.

Slots auxiliares:

- Para las clases, será necesario almacenar las clases padre originales en la ontología. Para ello se creará un slot del sistema, el cual no se podrá modificar. Si la clase tiene nuevos padres almacenaremos en slot los oids de los padres verdaderos, si los mantiene no se añadirá nada. Almacenar en los GResource los OIDS de los padres (parentOID).

Casos posibles:

1. Sin padres → buscar abuelos de todos los padres
2. Con algún padre → buscar abuelos del padre que no tiene
3. Con todos los padres → do nothing

Funcionamiento:

- **extracGrandFatherOIDs:** extraerá los OIDs de los abuelos de la clase a la que le falta algún padre
- **lookForGrandFathers:** a partir de los OIDs de los abuelos comprueba si éstos están insertados en `_classes`, en caso afirmativo se añadirán a la lista de nuevos padres. Este método será recursivo para el caso en el que el abuelo no esté insertado y se busque en niveles superiores hasta llegar al concepto `:THING`. Destaquemos que el concepto raíz `:Thing` será válido como padre si no se tiene ninguno.
- Si no se tienen todos los padres, tras buscar los nuevos, deberemos actualizar la clase en cuestión, modificando el recurso 'clase' de la estructura '`_classes`', con la nueva lista de superclases.

Ejemplo de una vista en OntoPathView. Esta vista está definida por medio de dos consultas en OntoPath:

- OntoPath Query 1: `["Cubist"]/{ "Iname"="Picasso", "fname" }`
- OntoPath Query 2: `["Museum"]/{ "titleresource"="El Prado" }`

Como resultado devuelve el subgrafo representado en la figura 5. Destaquemos que la jerarquía de clases ha sido reestructurada debido a que los conceptos *Cubist*, *Artist* and *Museum* tienen un grupo diferente de antecesores.

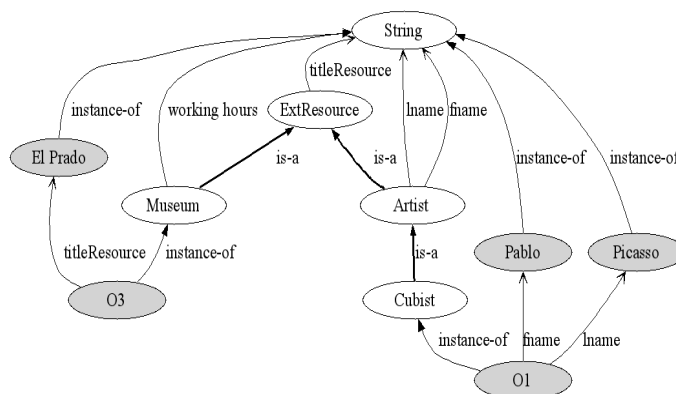


Figura 5: Ejemplo del grafo de la vista para el ejemplo anterior

6.3.9 Tratamiento de las propiedades transitivas

Las relaciones transitivas también será necesario reestructurarlas. Parte de la reestructuración, y quizá el caso más común, ya se realizó al completar las relaciones de asociación. Pero puede ser interesante completar relaciones transitivas que no sean directas, algo muy parecido a lo realizado con las jerarquía de clases. El método encargado de realizar la reestructuración será *'inferredTransitivePropertyRealations()'*

Funcionamiento:

- Será necesario recorrer las clases y extraer sus slots directos de tipo Instante y que sean transitivos
- Para cada spot extraído, que no esté ya insertado, sacaremos su clases destino:
 - o Si clase destino está insertada, el caso ya fue tratado en *'inferredClassRelations'*
 - o Si clase destino no insertada, buscaremos nuevas clases *'range'* (muy parecido a lo realizado con la jerarquía de clases, pero ahora en vez de cambiar el *is-a*, modificaremos la lista de clases destino (*range*))
- Método *'transitiveProperties'*: como parámetros tendrá del oid de una clase y el nombre de la propiedad transitiva (o conjunto de nombres).
 - o Extraerá los slots transitivos de la clase con el nombre de la propiedad en cuestión.
 - o Para cada propiedad buscará clases destino que estén cargadas.
 - Si clase cargada → la añadirá a lista de nuevas clases *'range'*.
 - Si no cargada → Recursividad con oid de la nueva clase.
 - o Destaquemos que pueden haber varias propiedades en una misma clase que representen a la misma propiedad transitiva (Ej: *hasPart1*, *hasPart2*, *hasPart3*, etc..., que serian las inversas de *'isPartOf'* más frecuente).
- Si se encontró alguna clase destino, construiremos la propiedad transitiva con el nuevo rango (que podrá ser múltiple). Si la lista de clases destino está vacía no se realizará ninguna acción.

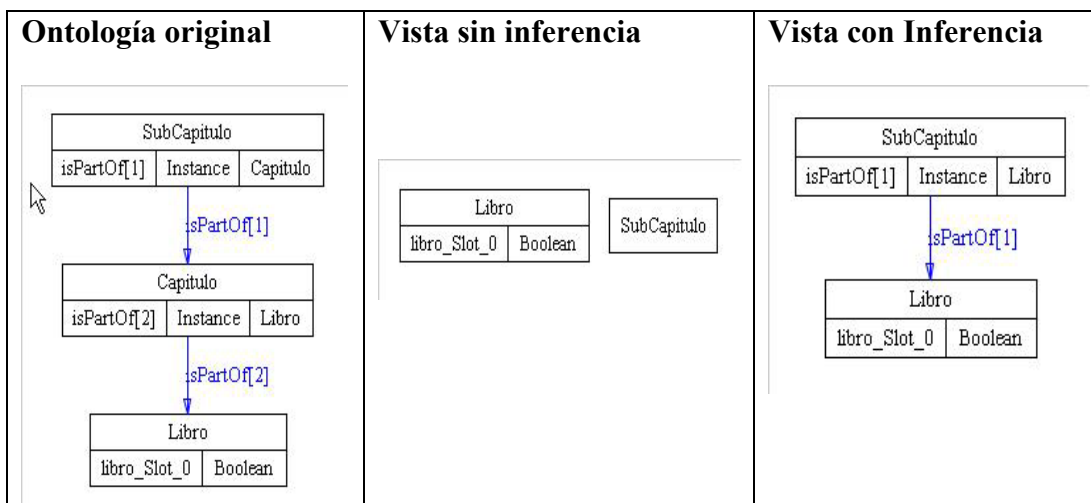
- Necesario nuevo slot del sistema 'originalRange' para no perder relación con el rango original, este slot almacenará el 'rangeOID'. Para los casos en los que el rango no se modifique este slot no almacenará nada.

Problemas con el volcado de la ontología, debido a que hemos transformado una propiedad, y podremos crear instancias de la clase propietaria y de las hijas, que no se adapten a la ontología inicial. En este caso puede interesar no permitir el volcado de la vista realizada y obligar a que sea una nueva ontología. Esto se reflejará en el fichero 'import.log', donde indicaremos si el *Export* sobre la misma ontología está permitido, o se debe crear una nueva (si se desea exportar/salvar).

Ejemplo:

“SubCapítulo”
“Libro”

Con esta vista traeremos clase 'Libro' y clase 'SubCapítulo', y detectaremos que hay una relación de transitividad entre ellos. Lo que habrá que tener en cuenta es que el rango del slot isPartOf[0] cambiará y pasará a ser 'Libro' en lugar de 'Capítulo'.



6.3.10 Actualización de las Instancias

Una vista está formada por varias subvistas que consultarán la ontología de forma independiente. Esto implicará que las instancias de una subvista solo tendrán en cuenta las propiedades de la misma. Cada subvista podrá traer nuevas propiedades que repercutirán a las instancias extraídas en otras subvistas, con lo que será necesario realizar una actualización de las instancias extraídas para que contemplen los valores de las propiedades disponibles.

También deberemos tener en cuenta las nuevas propiedades extraídas al inferir las posibles relaciones de asociación entre clases (no contempladas) y al inferir las relaciones de transitividad.

Funcionamiento (método `updateInstances()`):

- Recorreremos los recursos instancia de ‘_instances’ y extraeremos el objeto instancia correspondiente de G.
- Para cada instancia recorreremos la tabla hash ‘_slots_permitidos’ (pares slot-tipo).
 - o Si propiedad pertenece a objeto instancia de G y no esta insertada en la tabla hash (pares slot-valor) del recurso instancia → modificamos recurso instancia para que contemple el valor de una propiedad disponible.
 - o En caso contrario no se realiza ninguna acción.

También comprobamos que no se haga referencia a ninguna instancia que no esté cargada. Si una instancia hace referencia a otra que no está cargada (su nombre no está en estructura `_instances`), debemos cargarla, para ello:

- Tenemos que sacar a qué clase pertenece a partir del range del slot de la instancia que la referencia. Se accede a estructura `_properties` para sacar ‘oid’ del slot, consultamos G, y a partir del objeto de la propiedad sacamos ‘oid’ y nombre de la clase.
- Con nombre de la clase y *oid* podemos extraer de G la instancia requerida. Destaquemos que se construirá según la estructura `slots_permitidos`.
- Si la nueva instancia hace referencia a nuevas instancias, se aplicará recursividad

7. G SCHEMA IMPORT

Este plug-in extrae de G el esquema ya inferido de una base de datos cualquiera de G, proporcionando una visión ontológica del mismo. Los tipos específicos de los esquemas inferidos son los *gsComplexType*(conceptos) y *gsElement* (propiedades).

7.1 Modelo de Datos de los Esquemas

| D.y | Atributos | Comentario |
|---------------|-------------------|--|
| gsComplexType | name | Nombre del concepto |
| | created | Fecha de creación , de la inferencia |
| | cardinality | Numero de instancias de la clase (incica un máximo de 100) |
| | elements | OIDs de las propiedades (gsElements) |
| gsElements | name | Nombre de la propiedad |
| | parent | Nombre del concepto fuente (gsComplexType) |
| | type | Tipo lieteral |
| | dateCreated | Fecha de creación, de la inferencia |
| | gsComplexType-OID | OID del concepto fuente (gsComplexType) |
| | minOccurs | Cardinalidad minimo de la propiedad |
| | maxOccurs | Cardinalidad maximo de la propiedad |

Tabla 3. Modelo de datos de los Esquemas

7.2 Notas sobre la Implementación

La extracción del esquema y la correspondiente carga en Protégé posee las siguientes peculiaridades:

- Si el nombre de un gsElement termina en “-OID”, entonces hace referencia a una clase, es decir el rango de la propiedad de Protégé deberá ser el nombre de la clase, y como ‘ValueType’ será ‘instance’. También deberemos dar valor a ‘AllowedClasses’.
- El dominio de las propiedades se sacará a partir del campo ‘parent’.
- Si gsComplexType no tienen role → ‘concret’; si no tienen superclase (is-a) → ‘:THING’
- **Recuperación de los objetos por fecha o por gsSchema (no implementado).** Debido a que el esquema inferido no tiene un objeto raíz (gsSchema) una forma eficiente de recuperara el esquema es la fecha de creación de los gsElements (campo datecreated) y de los gsComplexType (created).
 - o Campos de los gsComplexType → name, created, elements (no usado), cardinality (valor no exacto, 100 es su máximo)

- Campos gsElements → datecreated, name, perent, minOccurs, type (string en todos, pero se tiene en cuenta si acaba el nombre en “-OID”).
 - Campos de protégé como superslots, allowedParents, inverseSlot quedan de momento con valor ‘null’.
- La **recuperación de las instancias** se realizará a partir del nombre de la clase, el problema puede surgir si tenemos varios gsComplexType en G con el mismo nombre, y con lo cual también pueden haber instancias con el mismo nombre y no ser de nuestra clase. Solución:
- Construiremos una tabla hash en ‘getGModel’, cuyas claves serán el nombre de las clases, y el valor asociado a cada clave será el conjunto de propiedades de las claves.
 - Esta tabla hash nos servirá para comparar las propiedades con los diferentes campos de las instancias (borrando previamente los campos del sistema: D.*, time, time1, gkeywords-OID). Si las propiedades de la instancia están incluidas en las de la clase (son subconjunto) entonces consideraremos la instancia como buena (es de nuestra clase). Aun así destaquemos que puede ser que carguemos en Protégé instancias que no sean de nuestro esquema, pero al menos serán ‘compatibles’.
- **Relaciones entre instancias:** en el esquema de G la relación entre instancias se realiza mediante el OID. En Protégé debemos indicar el nombre de la instancia. Solución:
- Al insertar las instancias tenemos que darles un nombre, y elegiremos uno que sea fácil de referenciar si es necesario. El nombre será pues, nombre_clase + “-” + OID instancia. Destaquemos que las clases en Protégé tendrán el sufijo Ej.: Grupo#~Class-1544.
 - Al referenciar la instancia el proceso será bastante sencillo. El nombre de la clase nos lo da el propio campo (*NombreClase*-OID), el OID es lo almacenado en el campo. Solo deberemos añadir la subcadena ‘#~Class’.
 - Par hacer referencia a propiedades será necesario introducir primero las propiedades antes que las instancias.
- **Problemas en las consultas:** Por defecto, si no se indica el número máximo de objetos de G con ‘limit’, solo se recuperan 100. Ejemplo de Consulta → “D.y=tipo limit=1000 key1=valor1 key2=valor2 ...”
- **Límite de almacenamiento en Protégé.** Si se tienen excesivas instancias, y varios campos por instancia, podemos ocupar excesiva memoria y saturar Protégé. La solución será traer vistas/imágenes de G, sin traernos todas las instancias ni todo el esquema. Destaquemos que la vista que nos traigamos deberá ser consistente.
- **Creación de Slots de Protégé:** Al traernos el esquema a Protégé podemos perder la referencia a él si cambiamos el nombre de las clases (que en nuestro caso se

cambia añadiendo sufijo Class) o de las propiedades. Para evitarlo será necesario crear unos campos auxiliares que asociaremos a las clases (campo types → slot de clase :STANDARD-CLASS) y las propiedades (campo attributes → slot de :STANDARD-SLOT).

7.3 Tratamiento del espacio de nombres

En G el espacio de nombres no es único pudiendo haber varias propiedades y clases con el mismo nombre, sin embargo en Protégé el espacio de nombres si que es único. El método *GSchema2ProtegeNames.GSchema2Protege* nos permitirá tratar los nombres entre Protégé y G. Para evitar problemas, si un nombre está ya en Protégé como clase o slot, deberemos transformarlo en *GSchemaFrameWalker*:

- Método *constructClass*: a cada clase se le asocia el sufijo ‘#~Class’
- Método *constructProperty*: a cada nombre de slot se le asociará el sufijo ‘#~’ + nombre clase. También deberemos transformar nombre clase dominio y clase rango si la hay.
- Método *constructInstance*: Para las instancias también deberemos cambiar el nombre de su tipo (clase de la que es instancia) según la transformación hecha en *constructClass*. Además las diferentes claves de los ítems también deberán ser transformadas para contemplar los nuevos nombres de la propiedades.

Vuelta a los nombres originales: La clase *GSchemaChangeNames* nos permitirá cambiar los nombres del esquema. Al cargar esquema inferido se ha utilizado una notación para poder mantener la relación entre los objetos, pero ahora ya en Protégé interesa volver al nombre inicial o semejante. Para ello recorreremos las diferentes clases, slots e instancias (ignorando los *frames* del sistema), y les eliminaremos el sufijo comentado en el punto anterior.

- Para las clases la eliminación del sufijo no tendrá ninguna repercusión.
- Para la instancias tampoco habrá un problema añadido.
- Para los slots si que pueden surgir problemas debido a coincidencia de nombres con clases y otros slots.
 - Se almacenará una estructura con los diferentes nombres de clase existente. Si nombre de slot sin sufijo coincide con nombre de clase, se le añadirá el sufijo ‘Slot’.
 - También se mantendrá una estructura de datos con los nombres de slot y el número de diferentes repeticiones (número de veces que han aparecido). De esta forma si el nombre del slot es único no se modificará, pero si hay otros con el mismo nombre, se le asociará un sufijo numérico, dependiendo del número de veces que un nombre de slot aparezca.

Destaquemos que los cambios se realizarán directamente sobre la base de conocimiento de Protégé con lo que el cambio en el nombre de una clase también repercutirá a los slots de dicha clase, los cuales actualizarán su dominio.

Nueva opción menú *Edit*: Si se desean realizar los cambios anteriores se deberá seleccionar la opción "Change Schema Names" del menú edit de Protégé.

8. TRABAJO FUTURO

8.1 Adaptación a Protégé 3.0

El plug-in implementado funciona correctamente con Protégé 2.1.2, pero actualmente ya está en funcionamiento las versiones 3.0, y la 3.1 Beta. Debido a cambios internos y en el entorno gráfico de Protégé el plug-in ha dejado de funcionar con total corrección, con lo que será necesaria una adaptación al nuevo núcleo de Protégé.

8.2 Compatibilidad con OWL

OWL es un lenguaje de marcado para la publicación de ontologías en la WWW y tiene como objetivo facilitar un modelo de marcado, construido sobre RDF y codificado en XML, que permita representar ontologías a partir de un vocabulario más amplio y una sintaxis más fuerte que la que permite RDF.

Nuestro plug-in soporta una expresividad semejante a la que permite RDF, por este motivo interesa en gran medida extender nuestro plug-in para que sea compatible con las nuevas características que ofrece OWL.

8.3 Desarrollo de un Entorno Colaborativo

La principal motivación seguida en el diseño de un lenguaje para definición de vistas fue el diseño de un entorno colaborativo. El sistema de Base de Datos G nos proporcionará el soporte necesario para el almacén de ontologías complejas que involucren miles de conceptos. Protégé 2000 por su parte, nos permitirá editar las diferentes secciones de una ontología de forma distribuida.

Como trabajo futuro se deberá tener en cuenta aspectos relativos a la gestión de conflictos en el acceso y a la notificación de cambios sobre una ontología al resto de usuarios que estén trabajando sobre ella. Se deberá estudiar la flexibilidad que se permitirá a la hora de realizar cambios que involucren varios objetos.

En la figura 6 se propone una arquitectura preliminar:

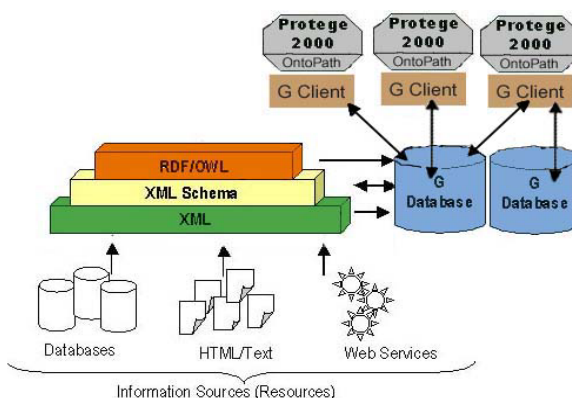


Figura 6: Arquitectura para un Entorno Colaborativo

9. LISTA DE DOCUMENTOS DE REFERENCIA

- [1] Ray W. Fegerson, Natalya F. Noy; Standford University; “Creating Semantic Web Contents with Protégé-2000”
- [2] Pisanelli D.M., Gangemi A., Steve G.; “Ontologies and Information Systems: the Marriage of the Century? Proceedings of Lyee Workshop, Paris, 2002
- [3] Natalya F. Noy and Deborah L. McGuinness, Stanford University, “Ontology Development”
- [4] W. E. Grosso, H. Eriksson, R. W. Fergerson, J. H. Gennari, S. W. Tu, & M. A. Musen. “Knowledge Modeling at the Millennium: The Design and Evolution of Protégé-2000”. 1999.
- [5] Holger Knublauch, 2003, “An AI tool for the real world”
- [6] A. Scheppler, “Almacenamiento y Consulta de Ontologías en la Base de Datos Semiestructurada G”, Proyecto Final de Carrera, Universitat Jaume I, 2004.
- [7] I. Sanz, “G Client Library Manual”, “G Client Protocol Specification”, Universitat Jaume I, 2004.
- [8] I. Sanz, J. M. Pérez, R. Berlanga, M. J. Aramburu ; “XML Schemata Inference and Evolution” 14th International Conference on Database and Expert Systems Applications, DEXA 2003.
- [9] Rafael Berlanga Llavori, Alex Scheppler, María José Aramburu Cabo, Ismael Sanz Blasco, Roxana Danger. OntoPath: A Query Language for Ontologies. IX Jornadas de Ingeniería del Software y Bases de Datos. Málaga 2004
- [10] Memoria Técnica del Proyecto (Proyectos Informáticos E80). “Almacenamiento y Recuperación de Ontologías en una Base de Datos Semiestructurada”, Ernesto Jiménez Ruiz, dirigido por Ismael Sanz Blasco. Universidad Jaume I , Septiembre de 2004.
- [11] Rafael Berlanga Llavori, Alex Scheppler, María José Aramburu Cabo, Ismael Sanz Blasco, Roxana Danger. OntoPath: A Query Language for Ontologies. IX Jornadas de Ingeniería del Software y Bases de Datos. Málaga 2004

ANEXOS

A. FORMAL DEFINITION OF AN ONTOLOGY AND A VIEW

In this section we present the formal conceptualization of our approach. We start from the definition of ontology and instance given in CRISOL⁶. Throughout this section we use the operator Π_i to project the i -th component of a tuple. Notice that the examples of the formal concepts are based on the ontology of Figure 1.

Definition 1 (Terminological Knowledge). A terminological knowledge (T-Box) is a structure $T\text{-Box} = (C, \leq_C, R, \sigma, \text{card}, \leq_R, IR)$ consisting of:

- Two disjoint sets C and R whose elements are called concepts and relations, respectively. Relations in R are represented as *concept.relation* where $\text{concept} \in C$ (e.g.: Artist.technique, Artifact.exhibited, Museum.location, ect.)
- A partial order \leq_C on C , called concept hierarchy or taxonomy, defined by the transitive relation *is-a* (e.g.: $\leq_C = \{(\text{Painter}, \text{Artist}), (\text{Sculptor}, \text{Artist}), (\text{Cubist}, \text{Painter}), \dots\}$).
- A function $\sigma: R \rightarrow C \times 2^{|C|}$, called signature (e.g.: $\sigma(\text{Artifact.exhibited}) = (\text{Artifact}, \{\text{Museum}\})$, $\sigma(\text{Artist.fname}) = (\text{Artist}, \{\text{String}\})$, $\sigma(\text{nut.partOf}) = (\text{nut}, \{\text{rim}, \text{wheel}\})$). Additionally, we define the function $\text{dom}: R \rightarrow C$ with $\text{dom}(r) = \Pi_1(\sigma(r))$ that gives the domain of r , and the function $\text{range}: R \rightarrow 2^{|C|}$ with $\text{range}(r) = \Pi_2(\sigma(r))$ that gives its range.
- A function $\text{card}: R \rightarrow \mathbb{N}^* \times \mathbb{N}$, that represents de minimal and maximal cardinality of each relation, where \mathbb{N}^* and \mathbb{N} are the sets of natural numbers including or not zero, respectively.
- A partial order \leq_R on R where $r_1 \leq_R r_2$, for $r_1, r_2 \in R$, defined by transitive relations. This order can imply two cases:
 - a. Property hierarchy with $\text{dom}(r_1) \leq_C \text{dom}(r_2)$ and $\text{range}(r_1) \leq_C \text{range}(r_2)$ (e.g.: $\text{Painter.paints} \leq_R \text{Artist.create}$, $\text{Sculptor.sculpts} \leq_R \text{Artist.create}$).
 - b. Order in other transitive relations with $\text{dom}(r_2) = \text{range}(r_1)$ (e.g.: $\text{nut.partOf} \leq_R \text{rim.ispartOf}$, $\text{rim.ispartOf} \leq_R \text{wheel.containedIn}$).
- A set IR of inference rules expressed in a logical language.

Definition 2 (Lexicon). Lexical and semantic information is needed in order to provide a bridge between the conceptualization and the natural language, for that reason we must define a lexicon. A lexicon for a terminological knowledge $T\text{-Box} = (C, \leq_C, R, \sigma, \text{card}, \leq_R, IR)$ is a structure $\text{Lex} = (S_C, S_R, \text{Ref}_C, \text{Ref}_R, IRL)$ consisting of:

- Two sets S_C and S_R whose elements are called signs (lexical entities with specific semantic) for concepts and relations, respectively. (e.g.: $S_C = \{\text{"Painting"}, \text{"Artist"}, \text{"Artiste"}, \text{"Peinture"}, \text{"O}_2", \text{"2.3"}, \dots\}$)
- Two relations $\text{Ref}_C \subseteq S_C \times C$, $\text{Ref}_R \subseteq S_R \times R$ called lexical reference assignment for concepts and relations respectively. (e.g.: $\text{Ref}_C = \{(\text{"Painting"}, \text{Painting}), (\text{"Peinture"}, \text{Painting}), (\text{"Artist"}, \text{Artist}), (\text{"Artiste"}, \text{Artist}), (\text{"O}_2", \text{Cubist}), (\text{"2.3"}, \text{Real}), \dots\}$)
- The inference rules IRL that must be used to assign unique names to instances.

⁶ Danger, R.; Berlanga, R.; Ruíz-Shulcloper, J.: "CRISOL: An approach for automatically populating a Semantic Web from Unstructured Text Collections". Database and Expert Systems. Ed. Springer-Verlag, 2004.

Definition 3 (Assertional Knowledge). In order to define the Assertional Knowledge (A-Box) we need the following auxiliary definitions:

- The set of relations associated to a class c will be denoted by $R|_c = \{r / r \in R, \sigma(r) = (c^*, \{c'\}), c^* \leq_c c, (\neg \exists r', c^* \leq_c \text{dom}(r') \leq_c c, r \leq_r r')\}$
- Based on Ref_c we define the domain of definition of a concept as follows: $\text{dom}_c(c) = \{s \in S_c / (s, c) \in \text{Ref}_c\}$
- We define an instance related to the object o of the class c , or simply an instance, to the set: $I|_o^c = \{(o, r, o') / \exists r \in R|_c, \sigma(r) = (c^*, c'), o \in \text{dom}_c(c^*), o' \in \text{dom}_c(c')\}$
For example: $I|_{o_1}^{\text{CUBIST}} = \{(o_1, \text{Artist.lname}, \text{"Picasso"}), (o_1, \text{Painter.paints}, o_2), \dots\}$
- The set of instances of a class c is denoted by: $I|_c = \bigcup_{\forall o_i \in \text{dom}_c(c)} \{I|_{o_i}^c\}$.

Finally, the A-Box is defined by means of the set composed by the union of all instances for all classes in the corresponding T-Box: $\text{A-Box} = \bigcup_{\forall c_i \in C} I|_{c_i}$

Definition 4 (Terminological Knowledge View). We define a Terminological Knowledge View or T-BView, over the T-Box $(\text{T-Box} = (C, \leq_c, R, \sigma, \text{card}, \leq_r, IR))$, as a structure

$\text{T-BView} = (C_v, \leq_c^v, R_v, \sigma, \text{card}, \leq_r^v)$ where $C_v \subseteq C$, $R_v \subseteq R$, $\leq_r^v \subseteq \leq_r$, and $\leq_c^v \subseteq \leq_c$.

Notice that the partial orders \leq_c^v and \leq_r^v are subsets of \leq_c and \leq_r , respectively. For this reason the application of the transitive closure to these partial orders will restructure the concept and property taxonomies, and the hierarchies defined by others transitive relations. (See examples in Section 3.2).

We also define a set of inference rules that will infer the necessary association relations in the above ontology view:

- if $\text{range}(r), \text{dom}(r) \in C_v \rightarrow r \in R_v$ (e.g.: $\text{Painter}, \text{Painting} \in C_v \rightarrow \text{Painter.paints} \in R_v$)
- if $\text{dom}(r) \in C_v, (\exists c' \in C_v, \text{range}(r) \leq_c^v c') \rightarrow r \in R_v, \text{range}(r) \in C_v$
- if $\text{range}(r) \in C_v, (\exists c' \in C_v, \text{dom}(r) \leq_c^v c') \rightarrow r \in R_v, \text{dom}(r) \in C_v$
(e.g.: $\text{Cubist}, \text{Painting} \in C_v \rightarrow \text{Painter.paints} \in R_v, \text{Painter} \in C_v$)
- if $(\exists c' \in C_v, \text{range}(r) \leq_c^v c'), (\exists c'' \in C_v, \text{dom}(r) \leq_c^v c'') \rightarrow r \in R_v, \text{dom}(r) \in C_v, \text{range}(r) \in C_v$

Definition 5 (Assertional Knowledge View). Similarly to the above A-Box definition we also need auxiliary definitions to denote the Assertional Knowledge View or A-BView: the set of valid predicates for a class, and the set of instances that satisfy these predicates.

– Let \wp_c be the set of valid predicates for class c , which is defined as follows:

$$\wp_c = \{(r, op, t, v) / r \in R_v, c = \text{dom}(r), v \in \text{dom}_c(\text{range}(r)), op \in \{=, \geq, \leq, >, <, \neq\}, t \in \{\forall, \exists\}\}$$

Where v is the value to match with the relation r in the instances, and op is the equality operator.

Additionally, we will denote with $P_{\forall}^c = \{(r, op, t, v) \in \wp_c / t = \forall\}$ the set of *for all values* predicates

and with $P_{\exists}^c = \{(r, op, t, v) \in \wp_c / t = \exists\}$ the set of *some values from* predicates.

– We define the set of instances that satisfy the predicates for the class ‘c’ by means of the set:

$$I|_{P_{\forall}, P_{\exists}}^c = \{I|_o^c \in I|_c^c / \forall (r, op, v, t) \in P_{\forall}^c, \exists (o, r, o') \in I|_o^c, o' op v\} \cup \{I|_o^c \in I|_c^c / \forall (r, op, v, t) \in P_{\exists}^c,$$

$$\forall (o, r, o') \in I|_o^c, o' op v\}$$

For example, the next query retrieves an instance of the class *Painting* whose *titleResource* is “Gernica”:

$$I|_{(\text{titleResource}, \neq, \exists, \text{Gernica})}^{\text{Painting}} = \{(o_2, \text{ExtResource.titleResource}, \text{"Gernica"}), (o_2, \text{Painting.technique}, \text{"Oil on Canvas"})\}$$

The A-BView will be the union of all instances for all classes in the corresponding T-BView:

$$\text{A-BView} = \bigcup_{\forall c_i \in C_v} I|_{P_{\forall}, P_{\exists}}^{c_i}$$

We also define the closure of instances by means of the following rule:

if $(\exists I|_o^c \in A\text{-BView}, (o, r, o') \in I|_o^c) \rightarrow I|_o^c \in A\text{-BView}$

For example, in the next case the instance $I|_{o_1}^{\text{Cubist}}$, which is in the view, references the instance $I|_{o_2}^{\text{Painting}}$, then both instances must be in the view:

if $\{(o_1, \text{Artist.lname}, "Picasso"), (o_1, \text{Painter.paints}, o_2), \dots\} \in A\text{-BView} \rightarrow$
 $\{(o_2, \text{ExtResource.titleResource}, "Gernica"), (o_2, \text{Painting.technique}, "Oil on Canvas"), \dots\} \in A\text{-BView} .$

Definition 6 (Ontology). An ontology is a triple (T-Box, A-Box, Lex) where T-Box is the Terminological Knowledge (abstract ontology), A-Box is the Assertional Knowledge and Lex is a lexicon for T-Box.

Definition 7 (View). A view is a triple (T-BView, A-BView, Lex). Notice that under this definition a view can be also considered as an ontology.