Proyecto Informático

Facultad de Informática
Universidad de Murcia

Representación de Información Clínica para La Web Semántica haciendo uso de Ingeniería de Modelos

Catalina Martínez Costa

Directores
Jesualdo Tomás Fernández-Breis
Jesús Sánchez Cuadrado

10 de Septiembre 2007

Índice general

Índice de figuras

Parte I

Introducción

Capítulo I

Introducción

1.1 Contexto

Una de las necesidades básicas de cualquier profesional sanitario es tener acceso a los historiales clínicos de sus pacientes. Estos historiales suelen encontrarse con frecuencia distribuidos en diferentes clínicas u hospitales, y almacenados en distintos formatos. La información clínica de una persona almacenada en un medio electrónico constituye lo que llamamos su Historia Clínica Electrónica (HCE). Los historiales clínicos electrónicos permiten desarrollar sistemas independientes de la representación de los datos y de la tecnología utilizada posibilitando así la compartición de la información a través de la interoperabilidad a nivel de datos y de conocimiento con otros sistemas.

Actualmente existen diferentes estándares para la representación y comunicación de la información contenida en una HCE como HL7, OpenEHR y CEN ENV13606. Cada estándar realiza un tratamiento diferente de dicha información y define su propio modelo sobre esta, por lo que HCE de diferentes organizaciones pueden nos ser representados de la misma forma. Los estándares actuales sobre HCE siguen una arquitectura denominada de modelo dual. Esta aproximación define dos niveles conceptuales: i) modelo de referencia y ii) modelo de arquetipos. En el proyecto nos centraremos especialmente en el segundo nivel conceptual, modelo de arquetipos.

La información clínica de una persona se representa en términos de arquetipos donde un arquetipo es un modelo formal de un concepto clínico en uso, es decir, un conjunto de restricciones sobre un dominio de información, que definen un concepto clínico. Para expresar dichos arquetipos se utiliza el lenguaje ADL, este lenguaje describe los arquetipos independientemente del estándar elegido, es un lenguaje genérico y flexible que puede ser utilizado en distintos ámbitos de aplicación. Sin embargo, al ser genérico presenta ciertos inconvenientes como el no garantizar la consistencia de la información clínica que representa, garantizándola solo a nivel de la sintaxis del lenguaje. Permite representar la información pero sin dotarla de significado.

Por otro lado la Web Semántica constituye el futuro de la Web, en el que la información se representa como información con significado o semántica facilitando su procesamiento e integración de forma automática. De ahí que las tecnologías de la Web semántica, concretamente las ontologías, sean apropiadas para administrar la información clínica, arquetipos, y proporcionar interoperabilidad entre los sistemas donde se almacenan los HCE. Una ontología puede ser vista como un esquema conceptual dentro de un dominio de aplicación dado que intenta facilitar la comunicación y la compartición de la información entre diferentes sistemas. Siguen un modelo de referencia y se caracterizan por ser tecnológicamente independientes. En el proyecto ofrecemos la posibilidad de representar los arquetipos expresados en ADL de forma alternativa usando el lenguaje de ontologías OWL. La representación de la información clínica en OWL, como ya se puede deducir y veremos más adelante, presenta ciertas ventajas respecto a su representación en ADL, tales como su representación formal, idoneidad para ser compartida y reutilizada, para deducir conocimiento .etc.

1.2 Objetivos

En este proyecto hemos tenido como objetivo el lograr obtener una metodología que permitiera lograr obtener la información clínica actualmente representada a través de un lenguaje de arquetipos, ADL, para la arquitectura propuesta por el estándar CEN ENV 13606, en un lenguaje de la Web Semántica, OWL, debido a las posibilidades que ofrece la representación de esta segunda forma y que se irán describiendo a lo largo del proyecto. Es importante destacar el hecho de que actualmente la información clínica se encuentra representada en los diferentes sistemas principalmente a través del lenguaje de arquetipos ADL.

El proyecto se ha centrado por tanto en obtener una metodología para lograr obtener la información clínica en OWL a partir de la misma información expresada en ADL y sin perder su significado semántico. Es importante destacar también que la información en OWL la hemos obtenido a través de una interpretación semántica y no sintáctica como ya se había hecho en otras ocasiones.

Como metodología nos planteamos como objetivo aplicar técnicas basadas en la Ingeniería de Modelos para cumplir nuestro objetivo. Ambos lenguajes, ADL y OWL, pertenecen a un espacio tecnológico específico, el espacio tecnológico de las gramáticas y el de la Web Semántica respectivamente por lo que en el proyecto se trata de establecer una conexión

entre ambos espacios tecnológicos usando un espacio tecnológico pivote, MDE[1]. En la Figura 1.1 podemos ver esta idea de forma gráfica.

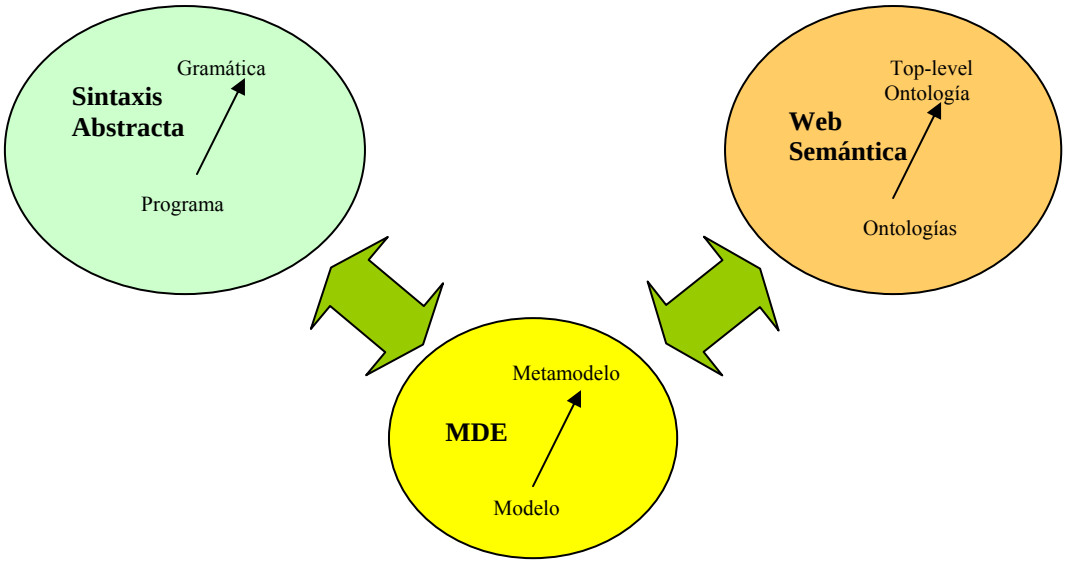


Figura 1.1: Tres Espacios Tecnológicos implicados

El desarrollo del proyecto se ha centrado por tanto en cubrir los siguientes objetivos:

1. Alineamiento del espacio tecnológico del grammarware correspondiente a ADL y el espacio tecnológico de MDE.
   1.1. Estudio del lenguaje ADL para la representación de información clínica.
   1.2. Obtención de la gramática que describa la sintaxis del lenguaje ADL.
   1.3. Evaluación y selección de herramientas y tecnologías para la implementación de la gramática ADL.
   1.4. Generación del metamodelo Ecore a partir de la gramática.

2. Obtención de modelos ADL conforme al metamodelo anterior.
   2.1. Obtención de un editor de ejemplos ADL.
   2.2. Procesamiento de ficheros ADL a nivel sintáctico.
   2.3. Generación de modelos ADL.

3. Alineamiento del espacio tecnológico de la Web Semántica correspondiente a CEN-AR y el espacio tecnológico de MDE.
   3.1. Estudio de la ontología CEN-AR en OWL.

[1] El acrónimo MDE es usado actualmente por la comunidad investigadora internacional cuando se refieren a ideas relacionadas con la ingeniería de modelos sin centrarse exclusivamente en los estándares OMG.

3.2. Evaluación y selección de las herramientas y tecnologías para la obtención de las interfaces EMF.
3.3. Obtención de las interfaces Java anotadas de la ontología CEN-AR.
3.4. Obtención del metamodelo Ecore correspondiente a las interfaces Java anotadas.

4. Transformación de arquetipos ADL en arquetipos OWL en el espacio tecnológico de MDE.
   4.1. Definición de las correspondencias a nivel conceptual entre los metamodelos ADL y OWL generados anteriormente.
   4.2. Evaluación y Selección de las herramientas y tecnologías a emplear para implementar las correspondencias.
   4.3. Implementación de un transformador de modelos ADL a modelos CEN-AR.

5. Obtención de instancias de arquetipos OWL a partir del metamodelo CEN-AR.
   5.1. Definición de las reglas basadas en plantillas para la generación textual de OWL conforme a la ontología CEN-AR .
   5.2. Evaluación y selección de las herramientas y tecnologías a emplear.
   5.3. Implementación de un generador de instancias de arquetipos OWL .

1.3 Organización del documento

A continuación en el capítulo dos se describe con mayor nivel de detalle lo expuesto en esta introducción, desarrollándose cada uno de los conceptos mencionados, HCE, ADL, CEN, OWL, DSDM, para que pueda se entendida la motivación del proyecto. El capítulo tres presenta las herramientas empleadas centrándose especialmente en las características que nos han sido útiles. En el capítulo cuatro se muestran cada uno de los pasos dados en el desarrollo de la solución y por último en el capítulo cinco se presentan las conclusiones extraídas y las posibles vías futuras que ofrece el proyecto.

Parte II

Fundamentos Tecnológicos

Capítulo II

Fundamentos tecnológicos

2 Historia Clínica Electrónica (HCE)

Para que un profesional sanitario pueda desarrollar su labor debe disponer de toda la información clínica existente de sus pacientes. Cuando dicha información se encuentra en soporte informático hablamos de Historia Clínica Electrónica (HCE) [2].

La HCE, como elemento integrador de los datos sanitarios y fuente de información para los profesionales, es el elemento esencial de un sistema sanitario eficiente y de calidad. Tener reunidos todos los datos de un paciente, y actualizados con toda la nueva información que se obtiene, es un objetivo estratégico desde el punto de vista asistencial, de gestión y de investigación.

Ningún profesional del mundo de la sanidad duda ya de la prioridad de dirigirse hacia una HCE en la que estén informatizados e integrados los documentos que contienen datos, valoraciones, e informaciones de cualquier índole sobre la situación y la evolución clínica de un paciente a lo largo del proceso asistencial.

Las ventajas de utilizar una HCE son claras. Según un informe de la Sociedad Española de Informática de la Salud, las historias clínicas tradicionales plantean dificultades como el desorden y la falta de uniformidad en los documentos, así como la inclusión de información muchas veces ilegible e inalterable y con una disponibilidad cuestionable. Por otro lado, no es poco frecuente que en estas historias tradicionales se produzcan errores de archivado de modo que su garantía de confidencialidad es dudosa. Además, hay que sumar el deterioro del soporte documental que experimentan, así como la dificultad para separar los datos de filiación de los clínicos.

2.1 Arquitectura de la HCE

La información clínica contenida en una HCE debe tener alguna estructura de forma que pueda ser manipulada o procesada por un sistema informático. La estructura debe ser adecuada tanto para el proceso de atención sanitaria como para otros posibles usos (investigación, formación, etc.). Por ello uno de los aspectos más importantes a la hora de desarrollar sistemas de HCE es como organizar dicha información clínica. Una arquitectura de historia clínica electrónica (AHCE) modela las características genéricas aplicables a cualquier anotación en una HCE.

Actualmente existen diferentes estándares que llevan a cabo esta labor, destacamos entre todos ello el desarrollado por el Comité Europeo de Normalización (CEN) el cual se ha utilizado en el proyecto [24].

2.2 Estándares aplicables en HCE

Los temas de estandarización han cobrado mayor importancia y se están volviendo más complejos con la globalización de la economía y la liberalización de los mercados. Los productos se tienen que diseñar para ser aceptados por usuarios de múltiples países con diferentes lenguas, sistemas de valores y condiciones de trabajo.

Existen diferentes organizaciones oficiales reconocidas en el ámbito internacional relacionadas con las tareas de normalización de informática médica y por lo tanto también con aspectos de los sistemas de la HCE.

ISO es la organización de alcance mundial en la que opera el Comité ISO TC215. En Europa la autoridad es CEN (Comité Europeo de Normalización) en el que participan los organismos nacionales como es el caso de AENOR en España.

ANSI (American Nacional Standard Institute) es el organismo oficial de EEUU que coordina las actividades nacionales de normalización en informática para la salud mediante el HISPP (Healthcare Informatics Standard Planning Panel). Este comité canaliza la participación de los grupos de normalización de varias organizaciones independientes como son HL7, DICOM, ASTM, IEEE y SNOMED. Otros organismos internacionales son el comité IT 14 de Estándares Australia, o el MEDIS-DC del MITI en Japón.

Merece también ser destacada la contribución de otras organizaciones que aportan su esfuerzo al desarrollo de los estándares de HCE. Entre ellas cabe destacar OpenEHR Foundation [28], Open Source Health Care Alliance, EUROREC-European Health Records Institute y los Centros Nacionales PROREC en Europa.

2.2.1 El Comité Europeo de Normalización (CEN)

Las actividades de estandarización en informática y telemática para la salud en Europa se remontan a 1990 con el establecimiento del Comité Técnico TC251 del CEN. El campo de competencias del CENT251 es la Informática y Telemática para la Salud y su ámbito de actuación incluye la organización, coordinación y monitorización del desarrollo de los estándares incluyendo los estándares de pruebas, así como la promulgación de dichos estándares. Actualmente el CEN TC251 está estructurado en cuatro grupos de trabajo (WGs), cada uno de ellos se encarga de desarrollar determinados aspectos:

1. WG1: Modelos de información
2. WG2: Terminología y bases de conocimiento
3. WG3: Protección, seguridad y calidad
4. WG4: Tecnología para la interoperabilidad

Las actividades de estandarización relacionadas con arquitectura y modelos de información de HCE, que son las que nos interesan, se desarrollan dentro del grupo de trabajo WG1.

En 1995, CEN publicó el pre-estándar ENV 12265 “Arquitectura de historia clínica electrónica”, que fue un estándar de referencia histórica definiendo los principios básicos en los que se deben basar las historias clínicas electrónicas. A este documento le sucedió el pre-estándar ENV 13606 “Comunicación con la historia clínica electrónica”, que se publicó en 1999 y finalmente se desarrolló en el 2003 el estándar oficial denominado ENV 13606 y futuro ISO basado en un paradigma del modelo dual (2.3). La norma CEN/TC251 ENV13606 está orientada al desarrollo de un marco estándar para la comunicación de la HCE. Esta norma se encuentra en la actualidad en la fase de análisis y votación para convertirse en un estándar ISO siendo una referencia a nivel mundial. Este nuevo estándar consta de cinco partes:

1. *Parte 1: Modelo de Referencia*: un modelo de información genérico para comunicar con la historia clínica electrónica de cualquier paciente, que es un refinamiento de la Parte 1 de ENV 13606.
2. *Parte 2: Especificación de intercambio de arquetipos*: un modelo de información genérico y un lenguaje para representar y comunicar la definición de instancias individuales de arquetipos.

3. *Parte 3: Arquetipos de referencia y listas de términos*: un rango de arquetipos reflejando una diversidad de requisitos clínicos y condiciones, como un "conjunto de arranque" para ilustrar cómo otros dominios clínicos podrían representarse de forma similar (por ejemplo por grupos de profesionales sanitarios), y más listas (normativas o informativas) para soporte de otras partes de este estándar. Esto se correlaciona con ENV 13606 Parte 2.

4. *Parte 4: Características de seguridad*: define los conceptos del modelo de información que se necesitan reflejar dentro de instancias de HCE individuales para permitir una interacción apropiada con los componentes de seguridad que pudieran ser requeridos en cualquier implantación futura de HCE. Se construye sobre la ENV 13606 Parte 3.

5. *Parte 5: Modelos de Intercambio*: contiene un conjunto de modelos que se construyen sobre las partes anteriores de la norma y pueden formar el soporte de comunicaciones basadas en mensajes o en servicios, cumpliendo el mismo papel que la ENV 13606 Parte 4.

2.3 Modelo Dual

La mayoría de los sistemas de información siguen una metodología basada en un modelo de un sólo nivel, Figura 2.1. En este modelo los conceptos del dominio, o también llamadas entidades de negocio, se modelan directamente en software a través de modelado de casos de uso u otras técnicas de desarrollo del software.

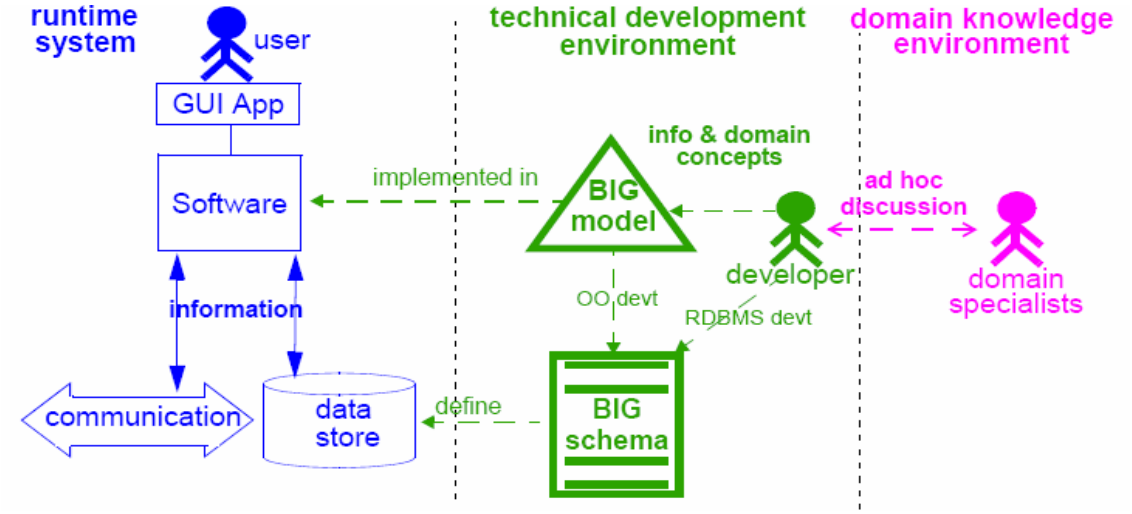


Figura 2.1: Modelo de un nivel

El sistema de la izquierda de la figura crea la información como instancias de las entidades de negocio, la almacena, transmite y transforma para ser entendible por las personas u otros sistemas de información. La base de datos, el software y la interfaz de usuario, en el centro de la figura, se desarrollan siguiendo un enfoque orientado a objetos (OO) o un enfoque relacional (ER) que describe de manera formal la semántica del sistema de información. Con este enfoque la mayoría de los sistemas relacionales u orientados a objetos traducen directamente los conceptos del dominio a código. Por ejemplo el concepto PERSONA se modela con una serie de atributos explícitamente como nombre, dirección, etc. Cuando la complejidad del dominio es baja y exige pocos cambios, sabemos que una entidad PERSONA va a tener esos atributos y es raro que tenga otros, esta aproximación es buena. Sin embargo en dominios más complejos como el de la medicina los cambios son frecuentes por lo que sería más apropiado usar lo que denominamos un modelo dual.

La arquitectura de modelo dual [1] define una separación clara entre información y conocimiento donde conocimiento refleja los posibles cambios. Para ver la diferencia entre ambos conceptos, un ejemplo claro sería decir que información es “Pedro Martínez tiene una presión sanguínea (BP) de 110/80” y conocimiento sería “La presión sanguinea consta de dos medidas: sistólica (la alta) y diastólica (la baja), cada una se mide en mmHg, y el protocolo para medirla: se utiliza un instrumento determinado indicando la posición etc”.

No todo el conocimiento es volátil, la anatomía humana o la psicología describen conceptos que ya son bastante estables. Además del problema de los cambios nos encontramos también con que el número de entidades de conocimiento en el campo de la medicina es muy extenso, existen millones y millones de términos por lo que los cambios son costosos.

Por lo tanto la alternativa a la aproximación de un solo nivel es, como hemos dicho, separar información y conocimiento. La información se estructura a través de un Modelo de Referencia (MR) y el conocimiento se representa utilizando un Modelo de Arquetipos (MA).

El modelo de referencia es un modelo orientado a objetos que se utiliza para representar las propiedades genéricas y estables de la información contenida en cualquier HCE. Es el encargado de codificar la información pero no determina como debe presentarse o almacenarse. Pueden existir diferentes MR adaptados a diferentes objetivos o diseñados bajo distintas perspectivas. El modelo adoptado en el proyecto es el MR del CEN y está orientado a la representación de documentos clínicos. Existen otros como el propuesto por openEHR que se centra más en la práctica clínica, definiendo entidades tales como observaciones, evaluaciones e instrucciones. En este nivel ningún concepto es volátil y debe ser de tamaño pequeño para asegurar su fácil mantenimiento.

El modelo de arquetipos representa el nivel de conocimiento, está formado por un conjunto de definiciones, arquetipos, que representan formalmente diversos conceptos clínicos construidos en base a entidades de un MR concreto. Un arquetipo define formalmente un concepto clínico, como puede ser un informe de alta, un análisis de glucosa o la historia familiar. Debido al escaso número de conceptos genéricos en el modelo de referencia es necesario usar los arquetipos para expresar el significado semántico de los conceptos. Nosotros utilizaremos el lenguaje ADL para representar los arquetipos (3).

En el centro de la Figura 2.2 se encuentra el sistema encargado de almacenar y comunicar la información. El modelo del software y de bases de datos no ocupan mucho espacio y no son volátiles, se encuentran a la derecha. Los conceptos que representan la semántica del sistema se encuentra a la izquierda de la figura en lo que denominamos “concept library”. Los modelos formales o lenguajes en los que se expresan dichos conceptos se encuentran incluidos en la parte derecha de la figura en lo que denominamos “technical development environment”.

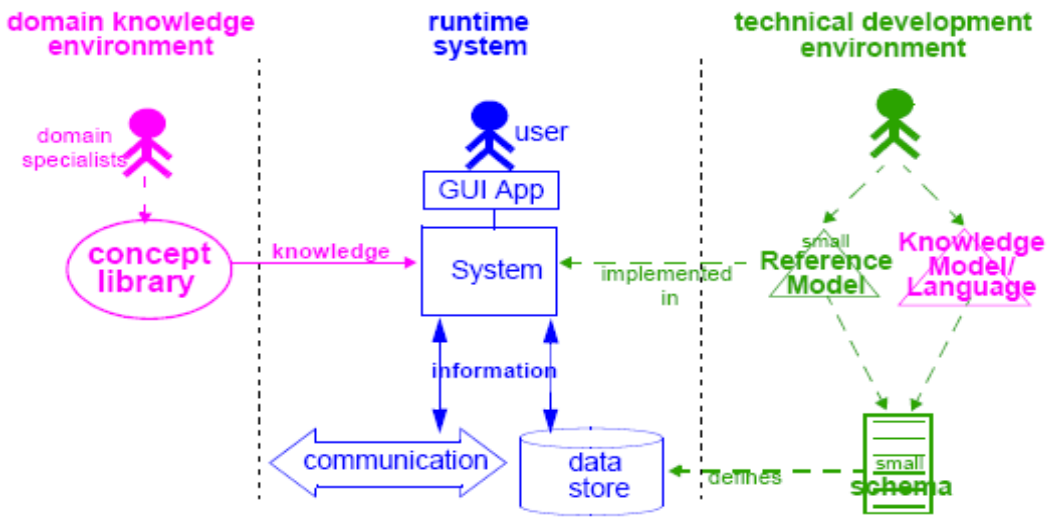


Figura 2.2: Modelo Dual

Las arquitecturas de HCE basadas en un modelo dual fueron pensadas en un principio para la implantación de nuevos sistemas de información sanitarios pero como podemos deducir también pueden utilizarse como base de un sistema para la integración y estandarización de información.

3 Arquetipos

Los profesionales sanitarios para la realización de sus actividades suelen manejar un conjunto de estructuras de información, que representan conceptos médicos, por ejemplo, informe de alta, resultados bioquímicos, diagnóstico, etc. Estos conceptos clínicos pueden definirse formalmente por medio de arquetipos tomando como modelo de referencia una AHCE.

De manera más precisa un arquetipo [1] se define como un modelo formal de un concepto clínico en uso donde la definición de ese concepto puede ser volátil. Los arquetipos definen configuraciones válidas de los datos, es decir, instancias de un modelo de referencia siguiendo cierta tecnología de un dominio determinado y han sido creados con los siguientes propósitos entre otros:

- Para permitir modelar de manera formal los conceptos de un dominio determinado por expertos de dicho dominio.
- Para validar la entrada de datos durante la creación o modificación de cierta información, garantizando que toda la información “instancias” se corresponden con los requerimientos del dominio.

- Para permitir interoperabilidad a nivel de conocimiento y no sólo a nivel de datos.
- Para proporcionar una base de definición para permitir búsquedas especializadas y eficientes de datos.

Los datos se representan como arquetipos a través del uso de plantillas definidas localmente para un uso determinado, y que especifican el manejo de los mismos mediante cierta estructura jerárquica así como algunos valores por defecto. La construcción de arquetipos en el ámbito clínico plantea la necesidad de dar soporte a ciertos aspectos como el mantenimiento de diferentes versiones, o contemplar la especialización y la composición de éstos. El ámbito clínico es un ámbito cambiante, que recibe constantemente resultados de nuevas investigaciones clínicas. Es por ello que la construcción de estos tiene que tener en cuenta este aspecto y permitir manejar y definir diferentes versiones. Permitir la reutilización de arquetipos es también algo obviamente positivo para preservar el esfuerzo y el tiempo empleados e incrementar la productividad. De esta manera algunos pueden ser definidos como extensiones o especializaciones de otros existentes, por ejemplo la definición de cierta condición genética puede ser vista como la especialización de un arquetipo que representa problemas genéricos. Este es otro aspecto que debe considerarse en su manejo. Finalmente pueden ser partes estructurales de otros, en este caso los mecanismos para manejar esta estructura deben de proporcionarse también.

3.1 Lenguaje de definición de arquetipos (ADL)

El lenguaje de definición de arquetipos (ADL) es un lenguaje creado para su expresión formal. Un arquetipo, como se ha descrito, define un modelo formal de un concepto clínico en uso donde la definición de este puede ser volátil.

Un arquetipo ADL está compuesto de tres partes: cabecera o descripción, definición y ontología, lo vemos en la siguiente figura:

```
archetype
identificador_arquetipo
specialise
identificador_arquetipo_padre
concept
código_nombre_concepto
description
```

sección_metadatos(dADL)
definition
sección_estructura_arquetipo(cADL)
ontology
sección_definiciones(dADL)

Figura 3.1: Estructura de un arquetipo representado en ADL

Los arquetipos expresados en ADL se asemejan a ficheros de lenguaje de programación y tienen una sintaxis definida. ADL utiliza dos sintaxis bien definidas: el formato de definición de restricciones (cADL) y el formato de definición de datos (dADL). La sintaxis cADL se utiliza para expresar la definición del arquetipo y la sintaxis dADL para el resto de secciones.

La cabecera está compuesta a su vez por varias secciones: i) sección arquetipo que introduce este e incluye un identificador, ii) sección especialización que indica que este es especialización de otro padre identificándolo, iii) sección concepto que indica un código que lo representa, un concepto del mundo real, asegurando que puede ser expuesto en cualquier lenguaje en que este haya sido traducido y iv) sección descripción que contiene información descriptiva (metadatos), p.ej Autor, Fecha de creación, Estado, Revisión, Propósito, etc.

La parte de definición contiene la definición formal principal del arquetipo escrita en cADL. Está compuesta por una serie de restricciones en forma de árbol creadas a partir del modelo de referencia.

La sección ontología se expresa en dADL y la podemos dividir a su vez en varias secciones: i) una cabecera en la que se indica el lenguaje en el que el arquetipo fue escrito y los lenguajes disponibles, es decir para los que existe traducción y terminologías para las que existe unión disponible, ii) sección de definición de términos en donde son definidos todos los términos locales del arquetipo, es decir términos de la forma [atNNNN], iii) sección de definición de restricciones con el mismo formato que la anterior, proporciona las definiciones, que son de la forma [acNNNN]. Las definiciones de restricción no proporcionan las restricciones propiamente dichas, sino que definen el significado de tales restricciones, de una manera comprensible para el humano, iv) .sección de bindings en la que se describen las equivalencias entre términos locales del arquetipo y términos encontrados en terminologías externas. El propósito de esto es sólo permitir software de consulta para poder buscar una instancia de un término externo o determinar que equivalencia usar en el mismpo y v) sección de restricción de bindings en la que se describen formalmente las restricciones sobre ítems de texto en el cuerpo principal del arquetipo.

ADL no es un lenguaje definido exclusivamente para el ámbito clínico, puede usarse para definir cualquier tipo de arquetipo. Es bastante flexible ya que la misma estructura puede utilizarse para especificar arquetipos basados en distintos modelos de referencia, sin embargo estamos hablando de la misma estructura sintáctica pero no semántica.

Un arquetipo ADL es necesario que sea definido para un modelo de información en particular, siendo posible definir arquetipos basados en diferentes modelos de referencia, CEN, OpenEHR, HL7, etc. Por otro lado un parser para ADL entiende ADL pero no sabe nada acerca del modelo de referencia en el que se basa, el procesamiento de un fichero ADL devuelve una colección de objetos sintácticos siguiendo un modelo abstracto de objetos de arquetipos (AOM. Archetype Object Model) que no pueden usarse para desarrollar ninguna actividad semántica. Por otro lado es un lenguaje genérico por lo que no presenta ninguna garantía sobre la consistencia de la información clínica que representa. Sólo ofrece consistencia a nivel de arquetipos, esto es consistencia en base a los principios en los que se basa ADL/AOM. Para procesar por tanto el contenido de un arquetipo descrito con ADL existe la necesidad de utilizar dos elementos: un parser ADL para capturar los objetos ADL y un parser basado en un determinado modelo de información para garantizar la corrección clínica de la información.

Con el objetivo de realizar el procesamiento semántico de un arquetipo descrito en ADL, el documento debe identificar el modelo de referencia en el que se basa dicho arquetipo, esto se realiza en la parte de identificación de la cabecera. En el caso que se muestra se hace referencia al modelo de referencia (CEN-EHR), tipo de la estructura clínica (ENTRY), nombre de la estructura clínica (Colesterol) y versión (v1) del arquetipo.

```
Archetype          CEN-EHR-ENTRY.Coagulacion.v1
```

Consideremos a continuación un fragmento de la parte de definición y de ontología del arquetipo Coagulacion basado en el modelo de referencia del CEN.

```
…
ENTRY[at0000] matches { -- Coagulación de la sangre
            items cardinality matches {0..1; ordered} matches {
                  ELEMENT[at0001] occurrences matches {0..1} matches {

                        value matches {
                            BL matches {
                                value matches {True, False}
                            }
                        }
                  }
            }
}
ontology
    primary_language = <"es">
    languages_available = <"es", ...>
    term_definitions("es") = <
      items("at0000") = <
          text = <"Coagulación de la sangre">
          description = <"*">
      >
      items("at0001") = <
          text = <"Actividad de Protombina">
          description = <"*">
      >
     >
…
```

Aquí un *ENTRY* definido por *at0000* está formado por un elemento identificado como *at0001*, si observamos el primero hace referencia a la coagulación de la sangre y el segundo a la actividad de protombina y se dice que dicha actividad puede tomar dos valores: verdadera o falsa, es decir puede o no existir.

Como hemos mencionado los arquetipos se modelan siguiendo un modelo de objetos. En él un arquetipo contiene translations, audit details, descriptions, archetype ontology y *C_Complex_Objects*. Los dos últimos componentes son los más importantes para nuestro proyecto ya que contienen la información sobre el concepto clínico. Podemos ver las diferentes clases de las que se compone un arquetipo en la Figura 3.2, donde se muestra una parte del modelo de objetos de ADL.

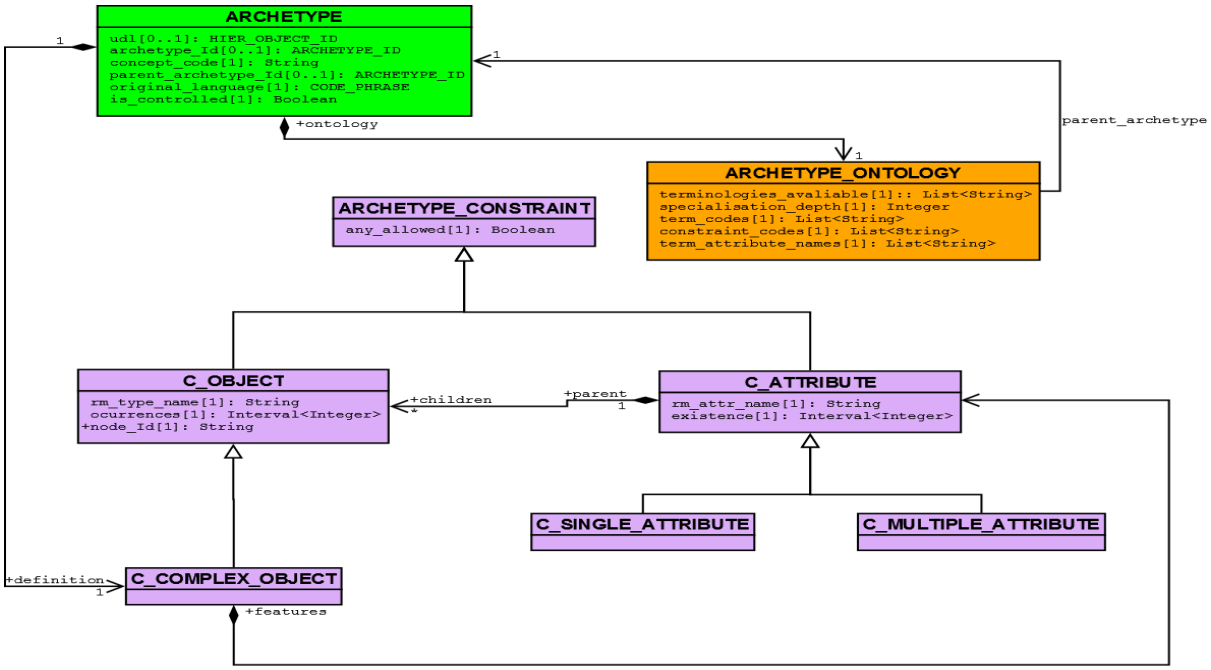


Figura 3.2: Fragmento del modelo de objetos (AOM) de ADL

Este modelo no representa la información semánticamente. Un parser sobre el ejemplo anterior devolvería un objeto Archetype con una propiedad para cada parte de la estructura ADL. Para la parte de definición devolvería un ENTRY como un C_COMPLEX_OBJECT con las propiedades: i) rm_type_name:String y ii) node_id:String. La primera propiedad establece el nombre del tipo en el modelo de referencia, en este caso ENTRY, y la segunda sería el identificador at0000, de forma parecida para ELEMENT [at0001] y para BL. Como se puede observar el parser devuelve un conjunto de objetos sin ninguna relación semántica entre ellos siendo muy limitadas las posibilidades de manejo de arquetipos ADL a un nivel mayor que el sintáctico, ya que obliga al programador a crear un parser específico para cada modelo de referencia que consiga dotar al arquetipo de significado semántico.

Para obtener más información sobre el lenguaje ADL es posible consultar la especificación del lenguaje: “http://svn.openehr.org/specification/TAGS/Release-1.0.1/publishing/architecture/am/adl2.pdf”

4 Web Semántica y su aplicación en el proyecto

La Web Semántica fue una propuesta visionaria de Tim Berners-Lee, inventor de la web tradicional, que pretende cubrir ciertas deficiencias presentes en la web actual. En la actualidad la World Wide Web está basada principalmente en documentos escritos en HTML. Este lenguaje es válido para adecuar el aspecto visual del documento e incluir objetos multimedia en el texto (imágenes, esquemas de diálogo, etc.) pero da pocas posibilidades para categorizar los elementos que configuran el texto más allá de las típicas funciones estructurales. En un trozo de código HTML no es posible relacionar la información que aparece en él, simplemente nos la muestra de la manera que le hemos indicado.

Estas deficiencias serían resueltas con la Web Semántica. Para ello dispone de tecnologías de descripción de contenidos, como RDF y OWL, además de XML para describir los datos. Estas tecnologías se combinan haciendo posible a las aplicaciones interpretar los documentos y realizar procesos inteligentes de captura y tratamiento de información.

Los principales componentes de la Web Semántica son las ontologías, la anotación semántica y los metalenguajes y estándares de representación XML, XML Schema, RDF, RDF Schema y OWL. A continuación describimos estos lenguajes:

- XML aporta la sintaxis superficial para los documentos estructurados, pero sin dotarlos de ninguna restricción sobre el significado.

- XML Schema es un lenguaje para definir la estructura de los documentos XML.

- RDF es un modelo de datos para los recursos y las relaciones que se puedan establecer entre ellos. Aporta una semántica básica para este modelo de datos que puede representarse mediante XML.

- RDF Schema es un vocabulario para describir las propiedades y las clases de los recursos RDF, con una semántica para establecer jerarquías de generalización entre dichas propiedades y clases.

- OWL añade más vocabulario para describir propiedades y clases tales como relaciones entre clases como la disyunción, cardinalidad, igualdad, tipologías de propiedades más complejas, caracterización de propiedades, por ejemplo simetría, o clases enumeradas.

Todas estas tecnologías son objeto de estudio por parte de grupos de trabajo del World Wide Web Consortium (W3C) dedicado a mejorar, extender y estandarizar las tecnologías para la Web.

En la actualidad se recomienda el uso de tecnologías semánticas con varios fines, entre los que se encuentra la búsqueda de la interoperabilidad entre sistemas. La tecnología semántica base para la Web Semántica es la ontología, que representa una visión común, compartible y reutilizable del conocimiento de un dominio de aplicación. Estas características se ajustan a las necesidades de compartición y reusabilidad de arquetipos entre sistemas de información sanitarios.

La interoperabilidad entre sistemas impone un entorno de trabajo en el que cada sistema emplea el significado de los datos en diferentes contextos. El uso de tecnologías semánticas posibilita la descripción de la naturaleza y contexto lógico de la información a intercambiar, mientras que permite que cada sistema mantenga su máxima independencia. Con este enfoque no se requiere sustituir las tecnologías actuales de integración, bases de datos y aplicaciones ya existentes.

Una ontología es esencialmente un modelo conceptual de información formal y estructurada. Podemos ver una ontología como un modelo semántico que contiene conceptos, sus propiedades, sus atributos, las relaciones entre conceptos, y los axiomas relacionados con estos elementos, y que soporta un modelo de referencia estándar para la integración de información conocido como compartición de conocimiento. En la práctica, uno de los motivos básicos por los que la ontología ha cobrado tanta importancia y se ha extendido su uso es por las ventajas que posee: i) reusabilidad, esto es, una misma ontología se puede reutilizar en diversas aplicaciones de forma individual o combinada con otras y ii) compartición, esto es, el conocimiento que incluye permite que sea compartido por una determinada comunidad. Sin embargo, la creación de una ontología no es tarea sencilla y lleva bastante tiempo conseguir una ontología que refleje adecuadamente el conocimiento de un dominio. A la hora de afrontar la integración de datos, las ontologías pueden ayudar simplificando la comprensión del dominio mediante la introducción de generalizaciones, ignorando los detalles para resaltar ideas generales.

En el proyecto se ha hecho uso de una arquitectura ontológica desarrollada en la Universidad de Murcia que permite la construcción de arquetipos haciendo uso del Ontology Web Language (OWL). El uso de este lenguaje también nos permite emplear herramientas de la

comunidad de la web semántica para poder gestionar y explotar la información clínica de manera más eficiente [16].

4.1 Ontology Web Language (OWL)

El lenguaje OWL se ha diseñado para representar información semántica que pueda ser procesada por programas o aplicaciones. A diferencia de otros lenguajes que sólo presentan el contenido a los seres humanos OWL puede usarse para representar explícitamente el significado de términos de un vocabulario y las relaciones entre ellos. A la suma de de dichos términos, relaciones, propiedades y axiomas se le denomina ontología.

En realidad, OWL está construido sobre el lenguaje RDF, y posee un mayor poder expresivo. Posee más funcionalidades para expresar el significado y semántica que XML, RDF, y RDFS, ofreciendo la posibilidad de representar contenido de la Web de forma que sea interpretable por una máquina. OWL es un lenguaje de etiquetado semántico para publicar y compartir ontologías en la World Wide Web.

Actualmente OWL posee tres variantes, cada una de ellas enfocada a una comunidad determinada de usuarios:

OWL Lite apoya a usuarios que necesiten sobre todo una jerarquía de clasificación y restricciones sencillas. Por ejemplo permite definir restricciones de cardinalidad pero restringe su valor a 0 o 1. Proporciona una forma fácil de migración para otras taxonomías. Su complejidad formal es la más baja de las tres variantes.

OWL DL proporciona una expresividad máxima con garantías de cómputo incluyendo todas las construcciones de su lenguaje con el inconveniente de que estas construcciones sólo pueden ser utilizadas bajo ciertas restricciones. Su nombre proviene de la correspondencia que mantiene con la lógica descriptiva y es la más usada en la actualidad, por el hecho de asegurar la completitud y finitud de los razonamientos.

OWL Full está destinado a usuarios que desean una expresividad máxima y la libertad sintáctica ofrecida por RDF aunque sin garantías de cómputo. Permite que una ontología aumente el significado, RDF o OWL, del vocabulario predefinido.

En general, *OWL Lite* es más sencillo que *OWL DL*, y *OWL DL* es más sencillo que *OWL Full*. En el proyecto cuando hablemos de OWL nos referiremos siempre a la versión OWL DL.

4.2 Representación de arquetipos en OWL

La representación de los arquetipos en el lenguaje OWL puede garantizar la consistencia de la información clínica a diferencia de lo que ocurría con ADL ofreciendo además otros beneficios. El intercambio de estos es una tarea común en arquitecturas basadas en arquetipos. Un sistema particular puede recibir un nuevo arquetipo que tiene que clasificar. Esta clasificación no puede ser realizada utilizando ADL, pero sí a través de OWL. Por otro lado la terminología es muy importante en el dominio biomédico y en el modelado de arquetipos. Cualquier concepto clínico incluido en un arquetipo puede estar relacionado con diferentes terminologías. La terminología más importante actualmente, SNOMED, está adaptando su representación a la de la Web Semántica. Tener por tanto la representación de la información clínica y la información terminológica en un mismo formalismo facilitaría el manejo de dicha información.

El lenguaje OWL presenta la suficiente expresividad para modelar arquetipos clínicos. Las ontologías OWL se encuentran estructuradas mediante primitivas de lenguaje (subclass of) y propiedades definidas por el usuario. Las restricciones sobre los arquetipos pueden ser establecidas usando restricciones OWL o definiendo los elementos apropiados. El modelado de arquetipos implica ofrecer facilidades de versiones, especialización composición. Esto también puede ser expresado en OWL.

Pero además de la correspondencia sintáctica [23] es necesaria una correspondencia semántica entre OWL y los arquetipos expresados en ADL. Para ello se requiere primero la creación de una ontología basada en la interpretación semántica del modelo de referencia y de arquetipos del CEN ENV 13606. Esta ontología ha sido desarrollada como parte del proyecto de investigación POSEACLE[2] y como resultado se ha obtenido una ontología OWL del modelo de arquetipos cuya estructura describimos a continuación.

En dicha ontología se pueden distinguir dos entidades principales: la entidad *Archetype* y la entidad *Archetype_term*.

[2] Proyecto de investigación de la Universidad de Murcia: http://klt.inf.um.es/~poseacle/

Archetype: cada arquetipo representa un concepto clínico (presión sanguinea,…) que se encuentra definido en la parte de definición conceptual del arquetipo. Este concepto clínico será de un tipo clínico específico del modelo de información subyacente, en nuestro caso del modelo de información del CEN. El concepto puede ser una especialización de un arquetipo ya existente. Además un arquetipo contiene información general:

Auditory_details y *Archetype_description*: información general sobre quién y por qué creó el arquetipo, por qué fue revisado, información de copyright, propósito, uso, etc…

Assertions: Los asertos son axiomas aplicados sobre los arquetipos.

Translations to other languages: Un objetivo importante en el modelado de arquetipos es que las instituciones de la salud se encuentran geográficamente dispersas a través de diferentes países por lo que los arquetipos deberían estar disponibles en varios lenguajes. Por lo tanto deben existir mecanismos para gestionar la internacionalización de los arquetipos. Las traducciones son realizadas a nivel de términos, cada archetype term tendrá una serie de traducciones asociadas.

Terminologies: Las terminologías presentan las diferentes formas de codificar, representar y clasificar los términos médicos y son básicas en el ámbito biomédico. Manejar múltiples terminologías en los arquetipos permite a los creadores de estos usar sus propios términos sin disminuir la estandarización del contenido.

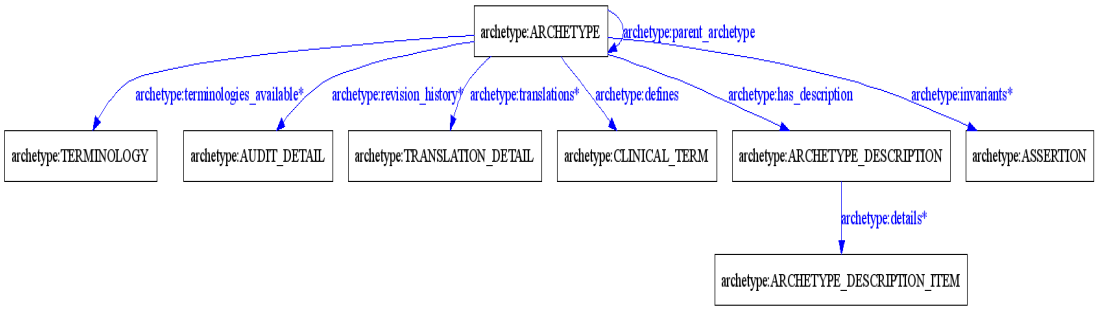


Figura 4.1: El concepto *Archetype* y su contexto

Archetype terms: un archetype term puede hacer referencia a restricciones o entidades conceptuales, es decir, constraint terms o clinical terms. La Figura 2.6 muestra el contexto de este concepto en el modelo obtenido. Este contexto incluye term definitions, term translations, terminological bindings y el tipo de un item clínico: Cluster, Element, Section, Entry, Folder, Item o Composition. Existen también otro tipo de restricciones referidas a la cardinalidad de los términos, existencia y número de ocurrencias.

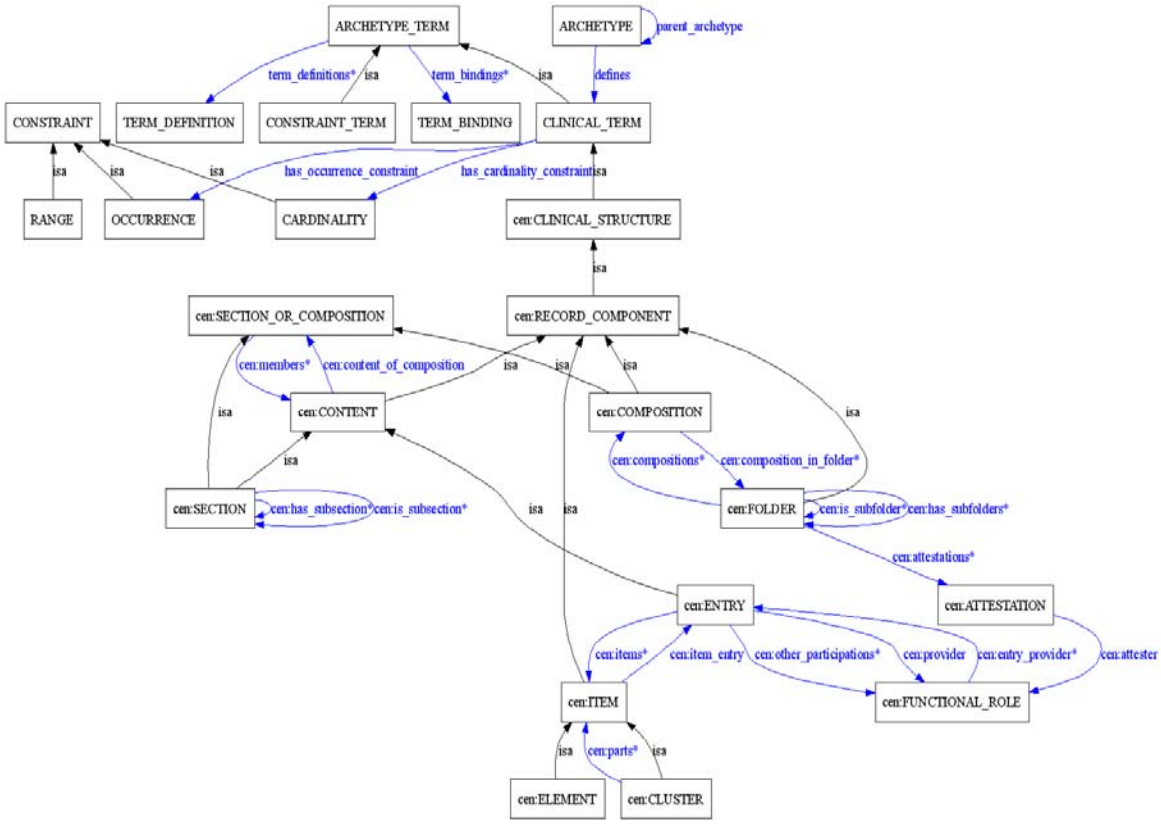


Figura 4.2: El concepto *Archetype term* y su contexto

Finalmente mostramos el aspecto de un fragmento de código OWL En él se describe la clase ARCHETYPE_DESCRIPTION la cual posee una serie de propiedades y valores definidos en su interior. Para obtener más información acerca de este lenguaje es posible consultar el sitio http://www.w3.org/TR/owl-features/.

```
<cen-archetype:ARCHETYPE_DESCRIPTION rdf:ID="CEN-EHR-ENTRY-Cholesterol-Description">
  <cen-archetype:details>
    <cen-archetype:ARCHETYPE_DESCRIPTION_ITEM rdf:ID="CEN-EHR-ENTRY-Cholesterol-Details">
      <cen-archetype:language rdf:resource="#en"/>
      <cen-archetype:purpose rdf:datatype="http://www.w3.org/2001/XMLSchema#string">CEN test</cen-archetype:purpose>
    </cen-archetype:ARCHETYPE_DESCRIPTION_ITEM>
  </cen-archetype:details>
  <cen-archetype:original_author rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
   unknown</cen-archetype:original_author>
  <cen-archetype:lifecycle_state rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
   draft</cen-archetype:lifecycle_state>
</cen-archetype:ARCHETYPE_DESCRIPTION>
```

5 Desarrollo de Software dirigido por Modelos

El Desarrollo de Software Dirigido por Modelos (DSDM) es una propuesta para el desarrollo de software en la que los modelos juegan el papel principal de todo el proceso, frente a las propuestas tradicionales basadas en lenguajes de programación y plataformas de objetos y componentes software.

En la Ingeniería de Modelos, cualquier artefacto software puede ser tratado o representado como un modelo: ontologías, esquemas relacionales, esquemas XML, etc. Un modelo describe una realidad física, abstracta o hipotética, recogiendo únicamente la información necesaria que permite conseguir unos objetivos específicos como pueden ser la generación de código, integración de aplicaciones, interoperabilidad entre aplicaciones, etc. Trabajar directamente sobre modelos aumenta el nivel de abstracción de estas tareas y permite automatizarlas.

5.1 Metamodelado y lenguajes de metamodelado

El metamodelado nace debido a la necesidad de definir la sintaxis abstracta de un lenguaje de modelado de manera "formal". Esta tendencia genera importantes ahorros en tiempo y recursos destinados al mantenimiento de software, desarrollo de nuevas aplicaciones y en el desarrollo de nuevas características sobre el software existente [10].

En sentido amplio definimos un metamodelo como un modelo de un lenguaje de modelado. La palabra "meta" significa por encima de, haciendo referencia al hecho de que un metamodelo describe un lenguaje de modelado con un nivel de abstracción mayor que el que posee el lenguaje en si mismo. Para entender que es un metamodelo, es útil entender la diferencia entre este y un modelo. Mientras que un metamodelo es también un modelo, tiene dos características principales que lo diferencian de este. La primera es que captura las propiedades y características esenciales del lenguaje que modela. La segunda es que forma parte de una arquitectura de metamodelado.

Una arquitectura de metamodelado posibilita que un metamodelo sea visto como un modelo, el cual es descrito a su vez por otro, permitiendo que todos los metamodelos sean descritos por uno único. Este último, es a veces conocido como meta-metamodelo y es la clave del metamodelado ya que permite que todos los lenguajes de modelado sean descritos de forma

unificada, es decir, que seamos capaces de describir cualquier lenguaje utilizando el mismo lenguaje de metamodelado. Entre los lenguajes de metamodelado más extendidos se encuentran MOF de OMG y Ecore para Eclipse.

Como hemos mencionado un metamodelo forma parte de una arquitectura de metamodelado. OMG define una arquitectura basada en cuatro niveles de abstracción que permiten distinguir entre los distintos niveles conceptuales que intervienen en el modelado de un sistema. Los niveles son, de más a menos concreto, los siguientes:

- M0: Las instancias. Modela el sistema real y sus elementos son las instancias que componen dicho sistema. Ejemplos de sus elementos son datos como por ejemplo "Pedro Martínez" que vive en "Av de la Constitución".

- M1: El modelo del sistema. En este nivel se realiza el modelado de la aplicación. Sus elementos son modelos de datos de los sistemas concretos, por ejemplo en el se definen entidades como "Persona" o "Coche", atributos como "nombre" o relaciones entre esas entidades. Existe una relación muy estrecha entre los niveles M0 y M1: los conceptos del nivel M1 definen las clasificaciones de los elementos del nivel M0, mientras que los elementos del nivel M0 son las instancias de los elementos del nivel M1.

- M2: El modelo del modelo, el metamodelo. Sus elementos son los lenguajes de modelado, por ejemplo UML. Los conceptos a este nivel podrían ser Clase, Atributo, Asociación. A este nivel trabajan las herramientas. Al igual que pasaba entre los niveles M0 y M1, aquí también existe una gran relación entre los conceptos de los niveles M1 y M2: los elementos del nivel superior definen las clases de elementos válidos en un determinado modelo de nivel M1, mientras que los elementos del nivel M1 pueden ser considerados como instancias de los elementos del nivel M2.

- M3: El meta-metamodelo, el modelo de M2. Finalmente el nivel M3 define los elementos que constituyen los distintos lenguajes de modelado. Es el nivel más abstracto donde se encuentran los lenguajes como MOF o ECORE que proporcionan los constructores y mecanismos necesarios para describir meta-modelos de lenguajes de modelado, es decir, los elementos que van a constituir un lenguaje dado, y las relaciones entre tales elementos, como el del lenguaje UML o en nuestro caso el lenguaje OWL y el lenguaje ADL.

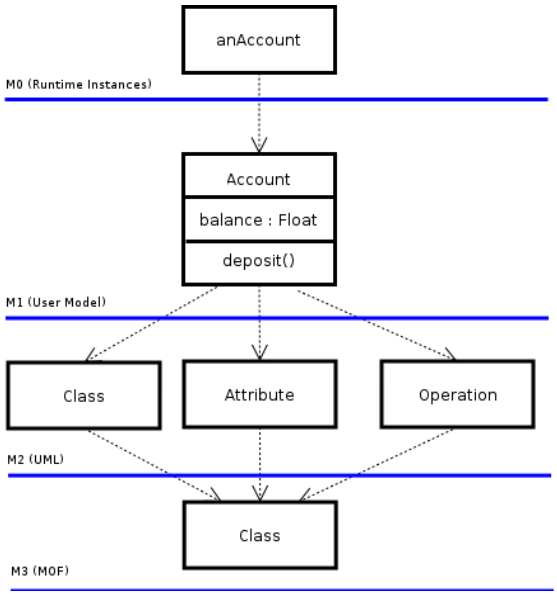


Figura 5.1: Ejemplo de la Arquitectura de 4 capas OMG

5.2 Espacios Tecnológicos

En Ingeniería del Software es muy frecuente que un problema pueda ser resuelto de múltiples formas, utilizando diferentes tecnologías para ello. Ante esto nos preguntamos ¿cuál es la forma más adecuada de resolver el problema?

Podemos definir un espacio tecnológico (TS) [14] como un área de trabajo que asocia varios conceptos, conocimientos, herramientas y posibilidades. Normalmente asociamos espacio tecnológico con una comunidad de usuarios que comparten conocimiento y apoyo a través de literatura, reuniones etc. Entre los TS conocidos se encuentran, el espacio tecnológico de las gramáticas, el espacio tecnológico de XML, el espacio tecnológico de los modelos (OMG/MDA), el espacio tecnológico de las ontologías, el espacio tecnólogo de las bases de datos (DBMS), etc. Cada uno de estos espacios tecnológicos está definido en base a un par de conceptos, por ejemplo sin nos fijamos en los anteriores tenemos, Programa/Gramática, Documento/Esquema, Modelo/Metamodelo, Ontología/Ontología de alto nivel y Datos/Esquema respectivamente. En la Figura 5.2 mostramos esta relación de forma gráfica.

Los espacios tecnológicos no son espacios aislados, se pueden comunicar entre ellos, unidireccionalmente e incluso algunos bidireccionalmente. Es posible realizar ciertas tareas en uno de ellos y utilizar el resultado en otro. Esto permite trabajar a varios niveles de abstracción. El Ingeniero del Software compara los espacios tecnológicos evaluando las posibilidades de

desarrollo e interoperabilidad que cada uno le ofrece a la hora de logra un determinado objetivo. En nuestro caso hemos utilizado el espacio tecnológico de las gramáticas a través del framework xTEXT, el espacio tecnológico de los modelos a través de EMF, RubyTL y MOFScript y el espacio tecnológico de las ontologías a través de Protégé, consiguiendo la interoperabilidad deseada entre ellos y que comentaremos en la parte de desarrollo del proyecto.

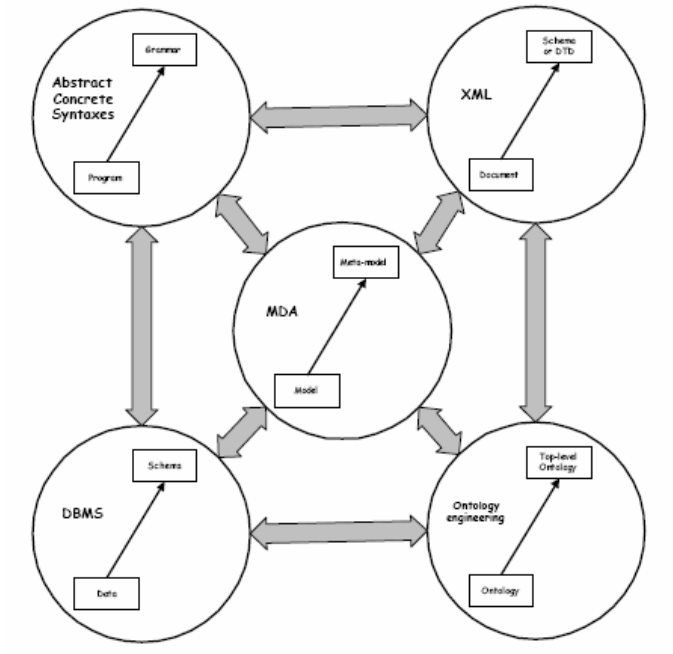


Figura 5.2: Interoperabilidad entre cinco TS

5.3 Transformación de modelos

La transformación de modelos constituye un objetivo importante en el DSDM. Su aplicación nos ofrece muchas posibilidades, por ejemplo, es posible realizar la traducción de un lenguaje a otro, si contamos con los modelos de ambos.

Dentro de la metodología del DSDM se pueden distinguir dos tipos de transformación entre modelos: i) transformación vertical de modelos y ii) transformación horizontal de modelos. El primer tipo consiste en transformar modelos de un nivel alto de abstracción en modelos menos abstractos, por ejemplo modelos independientes de una plataforma en concreto a modelos específicos de dicha plataforma, el segundo tipo es el que nos interesa y consiste en describir transformaciones de modelos del mismo nivel de abstracción.

Las transformaciones de modelos, Figura 5.3, requieren de lenguajes específicos para su definición. Estos lenguajes deben tener base formal, por ejemplo tener un metamodelo que los sustente y permitir un tratamiento automatizado. Actualmente existen numerosas

aproximaciones para diseñar estas transformaciones, la relacional, basada en grafos, estructurada, etc y numerosos lenguajes de transformación como ATL, Tefkat, MTF, QVT, muchos de ellos integrados en herramientas. En el proyecto se ha utilizado como lenguaje de transformación RubyTL del que hablaremos posteriormente

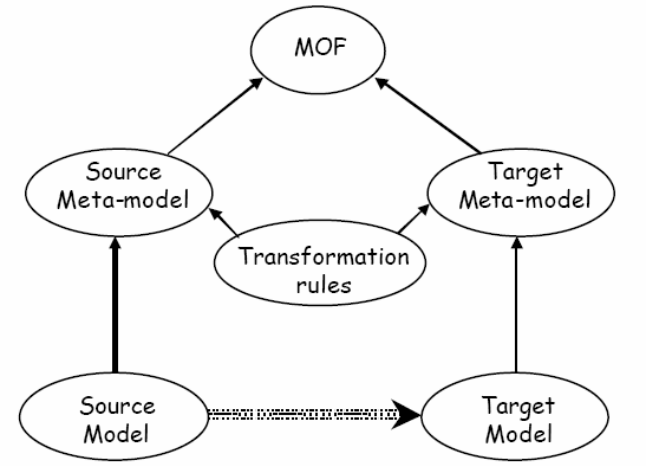


Figura 5.3: Transformación entre modelos

5.4 Transformación modelo código

Las transformaciones modelo-código son el último paso en DSDM. En el mundo de la arquitectura dirigida por modelos podemos distinguir tres etapas fundamentalmente: i) el diseño de un modelo independiente del dominio (PIM-Platform Independent Model), ii) el diseño de un modelo específico de un dominio (PSM-Platform Specific Model) y por último iii) la generación de código a partir del PSM como se muestra en la Figura 5.4. Podemos pasar de una etapa a otra de la arquitectura a través de transformaciones.

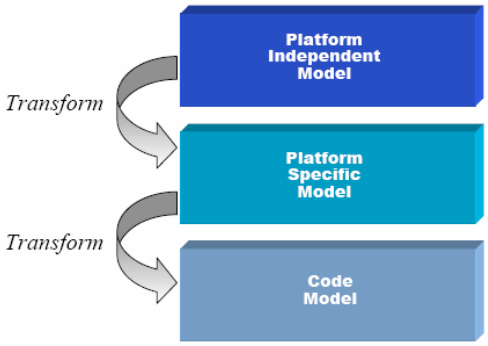


Figura 5.4: Tres etapas fundamentales de MDE

La transformación modelo-código es un tipo de transformación vertical que consiste en pasar los modelos de más bajo nivel a un lenguaje de propósito general ejecutable Existen diferentes alternativas a la hora de pasar de un modelo a código:

- Usar lenguajes de programación como Java o C#

- Usar lenguajes de tipo script como XSLT, JET (Eclipse Java Emitter Templates) o Velocity Template Language que no fueron creados para trabajar con metamodelos.

- Usar lenguajes de transformación como ATLAS Transformation Language (ATL), basado en conceptos de metamodelado pero desarrollado sin tener en mente la posibilidad de generar código.

- Usar herramientas tipo case que poseen lenguajes de script por ejemplo basados en UML con el inconveniente de estar ligados a una herramienta determinada y de que no están basados en estándares.

- Usar herramientas basadas en DSLs (Domain Specific Languages), que ofrecen las mismas ventajas a usar herramientas basadas en MOF, ambas generarán código a partir de un modelo concreto.

- Usar herramientas basadas en MOF, como MOFScript que ofrecen las mismas ventajas que las anteriores.

Parte III

Herramientas Empleadas

Capítulo III

Herramientas empleadas

6.1 Herramientas para el uso de ontologías

6.1.2 Protégé

Protégé ha sido desarrollada como una herramienta de código abierto [20] en la universidad de Informática Médica de Standford y actualmente cuenta con una comunidad de miles de usuarios. Su objetivo es facilitar el trabajo de creación de sistemas de adquisición de conocimientos, proporcionando interoperabilidad con otras herramientas. Inicialmente fue creada para trabajar en entornos de aplicación médicos pero hoy en día su uso en otras áreas ha resultado exitoso. En la figura 6.1 se puede observar el aspecto de dicha herramienta.

Al igual que ocurre con otras herramientas de modelado, la arquitectura de Protégé la podemos dividir en dos bloques, el modelo y la vista. El modelo es la representación interna que usa Protégé para definir ontologías y bases de conocimiento. La vista está formada por los componentes que proporcionan una interfaz de usuario para trabajar con el modelo.

La parte del modelo de Protégé está basada en un metamodelo que permite representar de forma flexible ontologías formadas por un conjunto de clases, propiedades (slots), restricciones e instancias. Además proporciona una API java para manipular modelos.

La parte de la vista permite a los usuarios diseñar ontologías y a partir de ellas permitir a través de una interfaz de usuario crear instancias de las mismas. Por cada clase de la ontología se crea un formulario para editar las propiedades de esta.

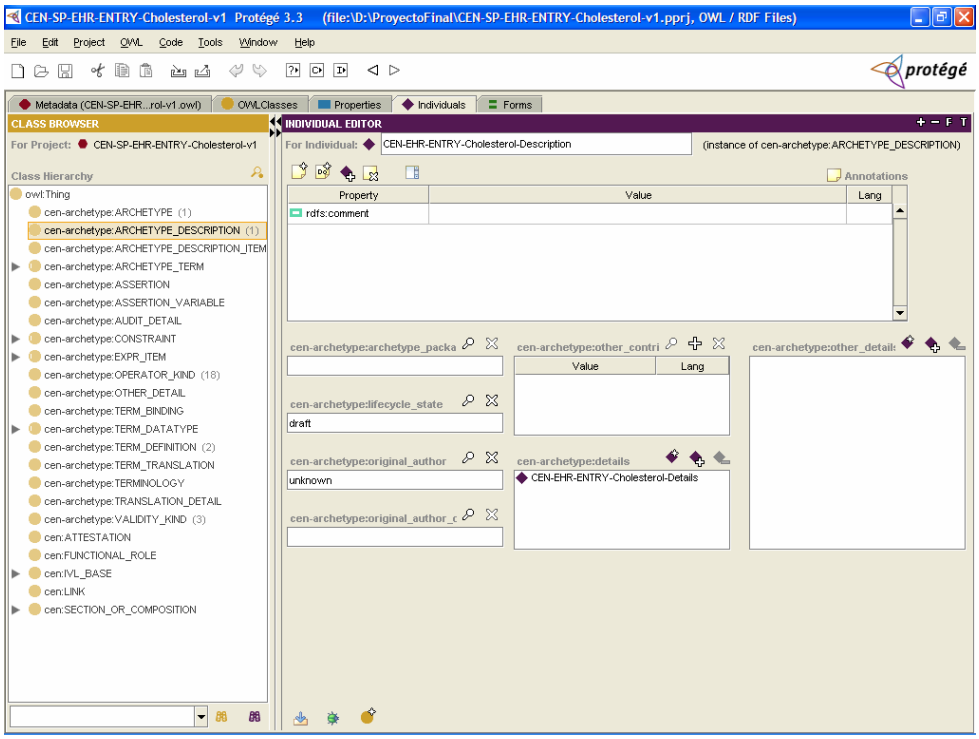


Figura 6.1: Aspecto de Protégé 3.3

Protégé puede cargar, editar y guardar ontologías en varios formatos como CLIPS, RDF, XML, UML o OWL. Este último formato fue añadido posteriormente, en el año 2003, debido a la importancia que actualmente está teniendo la Web Semántica y se ha desarrollado un plugin específico para trabajar con este tipo de ontología que será por otro lado la que hemos utilizado en el proyecto.

El plugin OWL [21] ha sido desarrollado como una extensión de Protege siendo necesario modificar la parte del modelo para adaptarla al metamodelo de OWL. Soporta los formatos RDF(S), OWL Lite, OWL DL y OWL Full prácticamente en su totalidad salvo algunas excepciones en el caso de OWL DL y OWL Full. Este plugin está constituido por una API OWL y una interfaz de usuario.

La API OWL, Figura 6.2, permite manipular las ontologías OWL extendiendo la API de Protégé. Inicialmente, tras cargar la ontología usando la librería Jena, se crea el modelo y la API de OWL se encarga de hacer la traducción de estos objetos a objetos Protégé.

La interfaz de usuario también se encuentra ligeramente personalizada a la hora de tratar con OWL.

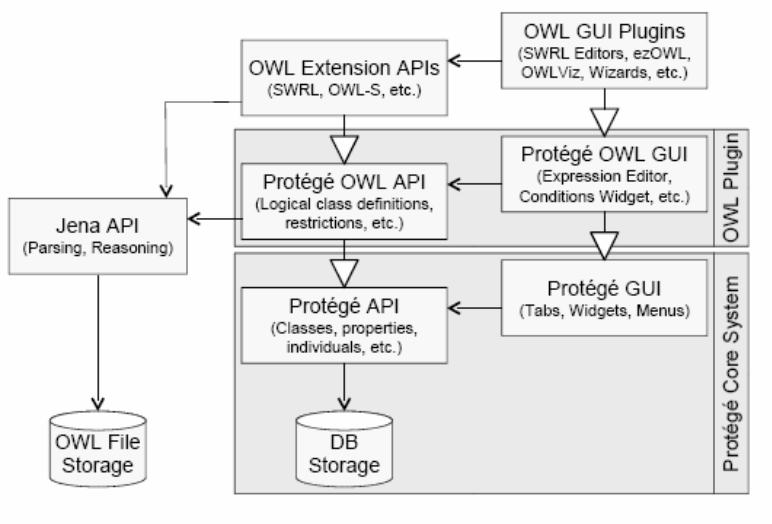


Figura 6.2: Plugin OWL como una extensión del núcleo de Protégé

Protégé permite exportar una ontología a otros formatos, entre las posibilidades que ofrece, la exportación a interfaces EMF en Java nos has resultado de utilidad en el proyecto. Por cada clase de la ontología Protégé genera una interfaz Java con un método get y set para cada propiedad y una serie de metadatos interpretables luego por EMF, que como mencionamos es capaz de generar un metamodelo Ecore a partir de dichas interfaces Java.

6.2 Herramientas para el DSDM

6.2.1 EMF

El Eclipse Modeling Framework (EMF) [25] es un framework desarrollado como plugin para Eclipse y cuyo objetivo es el de construir de forma rápida aplicaciones basadas en modelos. Unifica tres tecnologías importantes: Java, XML y UML, siendo capaz de generar un modelo EMF usando cualquiera de estas tecnologías y transformarlo a su vez al formato de cualquiera de ellas como se muestra en la Figura 6.3.

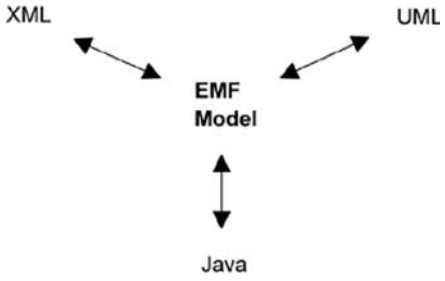


Figura 6.3: EMF unifica Java, XML y UML

Está compuesto fundamentalmente por dos frameworks, el framework del Core y el EMF.Edit. El primero proporciona soporte en tiempo de ejecución y generación básica de código para crear la implementación de clases Java de un modelo. EMF.Edit está construido sobre este como una extensión añadiéndole la posibilidad de crear clases adaptadoras que permitan la visión y edición de comandos de un modelo e incluso la generación de un editor básico para este.

Como hemos dicho podemos definir un modelo EMF usando diferentes tecnologías: a través de un diagrama UML, un fichero XML siguiendo un determinado XML Schema o interfaces Java anotadas. Para representar estos modelos EMF utiliza el metamodelo Ecore. El metamodelo Ecore es el lenguaje de metamodelado que permite definir metamodelos en el framework EMF. Este metamodelo actúa por tanto de meta-metamodelo de la herramienta EMF. El concepto no es tan complicado, un metamodelo es el modelo de un modelo y cuando uno de estos modelos es también un metamodelo entonces decimos que el metamodelo es un meta-metamodelo. Ecore es un subconjunto simplificado de MOF (Meta Object Facility) compatible con EMOF (Essential MOF). A continuación mostramos las clases básicas del metamodelo Ecore:

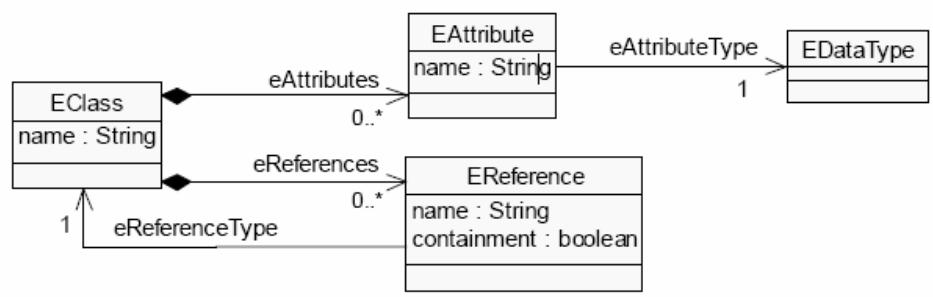


Figura 6.4: Modelo Ecore Simplificado

Como podemos observar en la figura anterior existen cuatro elementos fundamentales a la hora de representar un metamodelo.

- EClass representa las metaclases de un metamodelo. Se identifican a través de un nombre y pueden contener atributos y referencias.

- EAttribute representa los meta-atributos de las metaclases. Poseen un nombre y un tipo.

- EReference se usa para representar las asociaciones entre las metaclases. Se identifican a través de un nombre y puede además contener un booleano que indica que la

referencia es de un tipo más fuerte de asociación llamada contención y una referencia hacia otra clase.

- EDataType modela los tipos de los meta-atributos, pueden ser tipos primitivos o compuestos.

Para el proyecto nos interesa especialmente la generación de modelos a partir de interfaces java anotadas [11]. A través de la herramienta Protégé [6.1.2] podemos generar las interfaces Java de una ontología OWL, y con EMF crear un metamodelo Ecore a partir de ellas. Dichas interfaces están formadas por una serie de métodos get/set definidos por cada propiedad. EMF no interpreta estos métodos como atributos y referencias del modelo directamente, es necesario indicar explícitamente las partes de la interfaz que se corresponden con elementos del modelo que queremos que se generen. Esto se consigue a través de la anotación @model introducida en el comentario Javadoc. A continuación vemos un pequeño ejemplo.

```
/**
* @model
*/
public interface OrdenCompra {

	/**
	* @model
	*/
	String enviar();

	/**
	* @model
	*/
	String facturar();

	/**
	* @model type="Item" containment="true"
	*/
	List getarticulos();
}
```

La etiqueta *@model* identifica a OrdenCompra como una metaclase con dos meta-atributos enviar y facturar, ambos de tipo String. Con articulos se indica que es una referencia a objetos de tipo Item, y que son más de uno, *containment=true*. Los métodos set no son necesarios ya que EMF los asume por defecto.

EMF junto con la posibilidad de crear metamodelos de forma relativamente simple nos ofrece la posibilidad de generar código Java automáticamente a partir de ellos. Por cada clase Ecore (EClass) por ejemplo generará su interfaz correspondiente y la implementación de la misma.

También ofrece una API para la recuperación y el almacenamiento de los modelos. EMF por defecto, utiliza XMI (XML Metadata Interchange) como formato de persistencia para cualquier modelo. Sin embargo, si el modelo fue definido mediante un esquema XML, EMF permite guardar el modelo como una instancia XML del documento que sigue dicho esquema.

Además de la creación de modelos, de la generación automática de código y de la persistencia de los mismos, a través del componente EMF.Edit podemos construir visores y editores gráficos para los modelos EMF que permiten mostrar y editar (copiar, pegar, arrastrar y soltar, deshacer, rehacer, etc.) instancias del modelo utilizando vistas y hojas de propiedades estándares de Eclipse.

Podemos crear un modelo OWL haciendo uso del editor generado al crear el metamodelo de este lenguaje. Cuando creamos el metamodelo OWL, se crea un plugin que nos permite crear modelos, instancias concretas, en base a dicho metamodelo. Como vemos en la Figura 6.5, ahora existe un nuevo tipo de fichero llamado OWL Model que nos permitirá crear una instancia conforme al metamodelo creado.

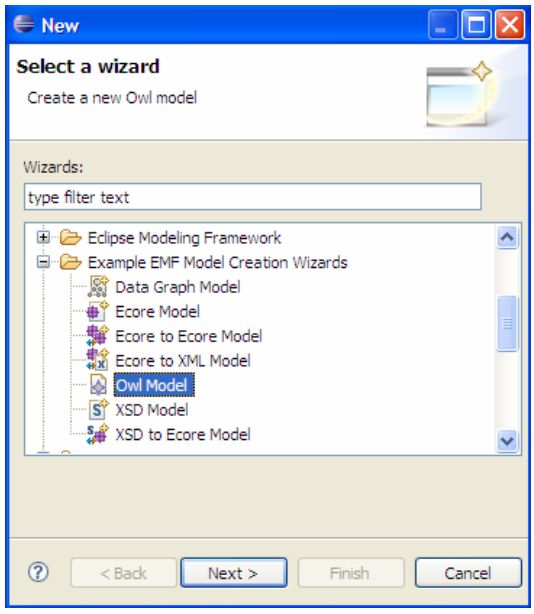


Figura 6.5: Creando un modelo OWL

Estos modelos no se crean de forma automática como veremos que ocurre en el caso de ADL, para instanciarlo debemos ir paso a paso creando cada uno de los nodos de la estructura que conforman el árbol del modelo como se observa en la Figura 6.6.

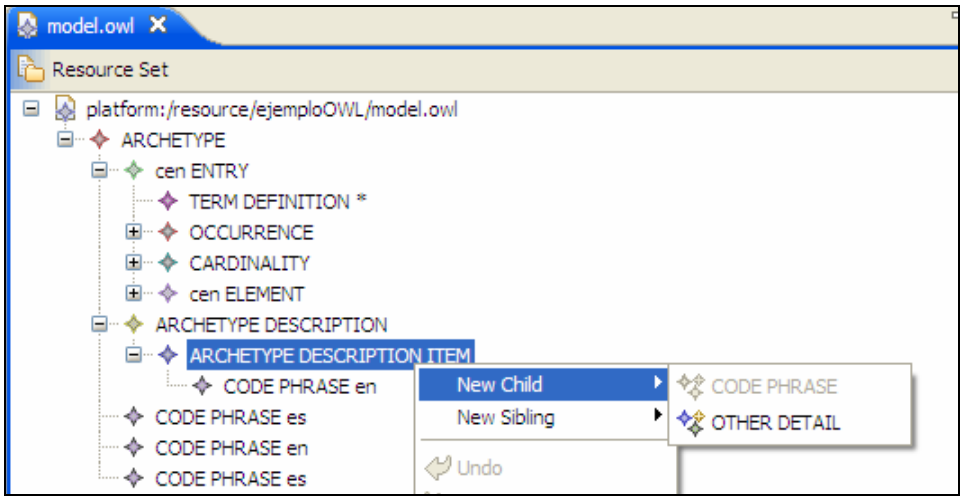


Figura 6.6: Creando un modelo Owl (II)

Cada uno de los nodos se corresponderá con una metaclase del metamodelo OWL y tendrá por lo tanto sus propiedades y referencias a otras metaclases del metamodelo que deberemos al igual generar manualmente. Como vemos a continuación en la Figura 6.7 rellenamos dos de las propiedades que posee la clase *ARCHETYPE DESCRIPTION*, las propiedades *Original author* y *Lifecycle state.*

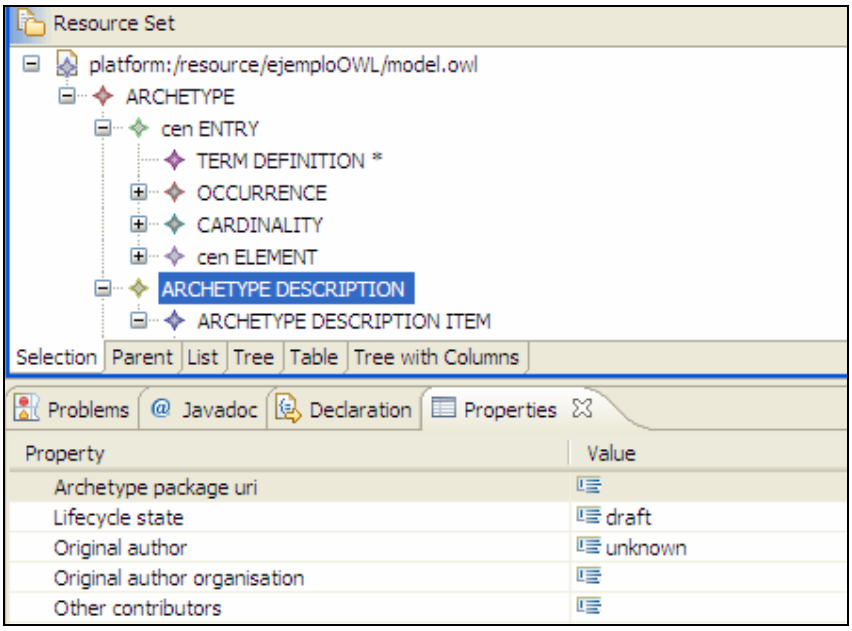


Figura 6.7: Propiedades de la metaclase Archetype description

6.2.2 XText

XText es un framework [6] para DSDM basado en openArchitectureWare, EMF y ANTLR. Permite describir un DSL (Lenguaje específico de un dominio) usando un lenguaje de notación similar a EBNF y a partir de él generar un metamodelo para dicho lenguaje, su parser correspondiente y un editor de este para Eclipse con resaltado de sintaxis, comprobación semántica y con la posibilidad de añadir tus propias restricciones. Esta aproximación permite que el desarrollo dirigido por modelos no parta originalmente de un modelo expresado de forma gráfica sino de un modelo generado a partir de una sintaxis concreta textual, lo que quizás pueda proporcionar una mayor exactitud en el desarrollo.

El lenguaje utilizado como hemos dicho es similar a EBNF [7]. A continuación mostramos un ejemplo de la gramática de un lenguaje muy simple expresada en xText, para más información sobre su sintaxis consultar la guía de referencia disponible en http://www.eclipse.org/gmt/oaw/doc/4.1/r80_xtextReference.pdf

```
Model :
   (entities+=Entity)*;

Entity :
   "entity" name=ID ("extends" extends=ID)? "{"
      (features+=Feature)*
   "}";

Abstract Feature :
   Attribute | Reference;

Attribute :
   "attr" type=ID name=ID;

Reference :
   "ref" type=ID (multiple?"[]")? name=ID;
```

Como vemos el lenguaje xText está basado en un conjunto de reglas y tokens. Cada regla se identifica a través de su nombre que será también el nombre de uno de los nodos del metamodelo y podrá contener propiedades y referencias a otras reglas.

En el caso del ejemplo, la regla *Entity* indicamos que debe comenzar con la palabra reservada o token *entity* seguida de un identificador (ID) que en el metamodelo será una propiedad de la metaclase *State*, a continuación la palabra reservada *extends* y un identificador indican que son opcionales mediante el operador "*?*", le sigue una llave abierta, una referencia a otra regla, *Feature,* y una llave cerrada. Con el operador "+=" indicamos que esta referencia es una lista de objetos de tipo *Feature*. Las reglas también pueden ser abstractas si van precedidas

de la palabra clave *Abstract*, como ocurre con la regla *Feature*. En este caso el cuerpo de la regla contendrá varias reglas como alternativa, un objeto Attribute o un objeto Reference. Mostramos en la Figura 6.8 como queda el metamodelo generado y observamos como las propiedades comunes de *Attribute* y *Reference* aparecen en la metaclase abstracta *Feature*:

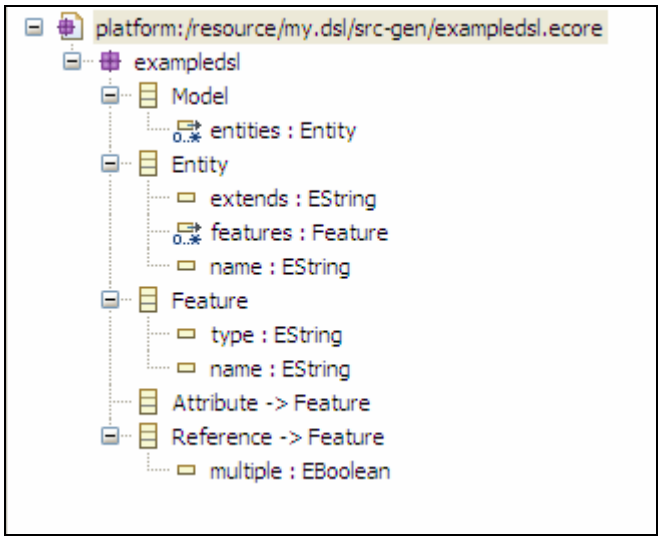


Figura 6.8: Metamodelo generado con xText

6.2.3 AGE

Agile Generative Enviroment (AGE) [19] es una herramienta para el desarrollo dirigido por modelos desarrollada en la Facultad de Informática de la Universidad de Murcia. Está basada en el lenguaje Ruby y se encuentra disponible como plug-in para la plataforma Eclipse.

AGE surge como entorno para un nuevo lenguaje de transformación de modelos, RubyTL. Actualmente la transformación de modelos constituye una tecnología esencial en los diversos enfoques de desarrollo de software dirigido por modelos.

RubyTL es un lenguaje de transformación híbrido definido como un lenguaje específico del dominio embebido en el lenguaje de programación Ruby, y diseñado como un lenguaje extensible en el que un mecanismo de plugins permite añadir nuevas características al núcleo de características básicas.

La versión actual de la herramienta, AGE 0.3.1, ofrece las siguientes características:

- Un lenguaje de transformación de modelos, RubyTL.

- Generación de código a través de un DSL, RubyTL, y plantillas.
- Un lenguaje de validación similar a OCL.
- Importación y exportación de EMOF y ECore (a través del proyecto RMOF).
- Editor para RubyTL, proporcionando resaltado de sintaxis, autocompletado y plantillas de código (a través del proyecto RDT).

6.2.4 El lenguaje de transformación RubyTL

RubyTL fue diseñado para satisfacer tres requisitos principales: i) debería ser un lenguaje híbrido ya que a veces la expresividad declarativa no es apropiada para transformaciones complejas en las que sería mejor utilizar un lenguaje imperativo. El estilo declarativo debería estar basado en una operación binding similar a la proporcionada por ATL, ii) debería facilitar la experimentación con características del lenguaje, y iii) debería permitir una rápida implementación. Estos requisitos se han satisfecho a través de dos decisiones de diseño: la definición del lenguaje como un DSL embebido en Ruby y la incorporación de un mecanismo de transformación.

En RubyTL una definición de transformación consiste en un conjunto de reglas de transformación. Cada regla tiene un nombre y cuatro partes: i) la parte *from* que especifica la metaclase del elemento origen, ii) la parte *to* que especifica la metaclase del elemento destino, iii) la parte *filter* que especifica la condición o filtro que deben satisfacer los elementos origen y iv) la parte *mapping* que especifica las relaciones entre los elementos de los modelos origen y destino, bien de forma declarativa a través de bindings, o bien de forma imperativa utilizando instrucciones Ruby. Un binding es una clase especial de asignación que permite declarar "qué necesita ser transformado en qué", en vez de "cómo debe ser algo transformado en algo".

Una regla de transformación podría ser por ejemplo aquella en la que queremos transformar los objetos de tipo clase en objetos de tipo tabla. Ambos se identifican a través de un nombre y cada atributo de una clase deberá transformarse en una columna de la tabla. La regla *klass2table* incluye dos ejemplos de *bindings*. En la parte *mapping* de esta regla se establecen las relaciones entre las propiedades de una instancia de Class y las de una instancia de Table. El primer *binding* establece que el nombre de la tabla creada será el de la clase, y el segundo *binding* especifica cómo se transforman los atributos (instancias de Attribute) en columnas (instancias de Column). Es importante notar que mientras el primer *binding* se resuelve directamente porque el elemento origen es de un tipo primitivo, el segundo disparará la

regla cuyas metaclases se correspondan con las de la asignación. Esta regla se ejecutará una vez por cada elemento cuya metaclase (o tipo) es *Class*, creándose un elemento cuya metaclase es *Table.*

```
rule 'klass2table' do
       from ClassM::Class
       to TableM::Table
  mapping do |klass, table|
       table.name = klass.name
       table.cols = klass.attrs
  end
end
```

Como hemos mencionado anteriormente RubyTL es un lenguaje extensible, su estructura permite definir puntos de extensión que pueden ser implementados para añadir cierta funcionalidad al lenguaje a través de plugins desarrollados en Ruby. A continuación se describen algunas de las características de lenguaje implementadas como plugins:

- Invocación explícita de reglas. Por defecto, el algoritmo de transformación está basado en la ejecución implícita de reglas a través de bindings. Se ha definido un plugin que permite invocar a las reglas por su nombre en un fragmento de código imperativo.

- Regla creadora. En RubyTL una regla nunca transforma un elemento origen dos veces, ya que este es el comportamiento esperado en la mayoría de las ocasiones. Sin embargo, en ciertas transformaciones es necesario que una regla sea ejecutada más de una vez para un mismo elemento origen. Esto se hace a través de la palabra *copy_rule*.

- Transformación en fases. Algunas transformaciones podrían ser simplificadas si pudieran ser expresadas en fases. Este concepto permite pensar en una transformación como un conjunto de pasos de refinamiento, donde cada fase confía en el trabajo realizado por las fases anteriores para cumplir su tarea. Usaremos para ello la palabra clave *phase*, posteriormente en la parte de desarrollo del proyecto veremos un ejemplo concreto de uso.

6.2.5 MOFSCRIPT

El lenguaje MOFScript surge tras la petición (RFP-Request for Proposal) del OMG de desarrollo de un lenguaje de transformación modelo-texto. Fue desarrollado por SINTEF ICT como parte del proyecto MODELWARE y del proceso de estandarización del OMG. Se encuentra integrado en un plug-in desarrollado para Eclipse. El OMG estableció una serie de

requisitos que debía cumplir el lenguaje creado. Entre los requisitos que estableció, los obligatorios fueron:

- La generación debía partir de modelos MOF 2.0.
- La reutilización de especificaciones anteriores del OMG si era posible, en concreto de QVT.
- Las transformaciones debían de definirse a nivel de metamodelo.
- El lenguaje debía ofrecer soporte para conversiones de string del modelo de datos y permitir su manipulación.
- Debía permitir combinar datos del modelo con código generado.
- Ofrecer soporte para transformaciones complejas.
- Permitir tener como entrada múltiples modelos MOF.

También estableció algunos requerimientos opcionales: la posibilidad de permitir la realización de ingeniería inversa y la detección de cambios en la regeneración, es decir permitir establecer relaciones de trazabilidad entre modelos y el código generado.

Finalmente como respuesta a esta propuesta MOFScript fue diseñado siguiendo la arquitectura de cuatro capas, Figura 6.9, propuesta por el OMG. En esta arquitectura MOF y por lo tanto también Ecore se sitúan en el meta-meta-nivel. El lenguaje MOFScript es por tanto un metamodelo y cada transformación específica una instancia de este que junto con el modelo fuente que a su vez conforma con su metamodelo genera como salida el código.

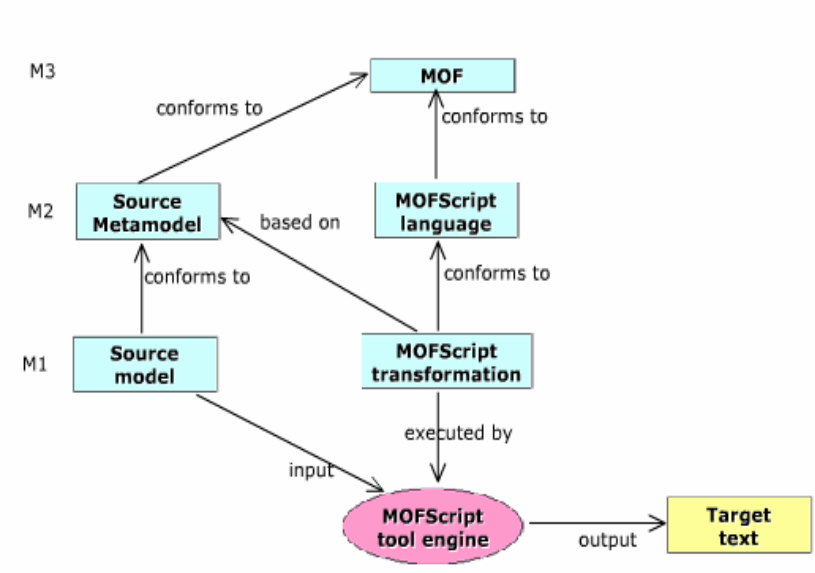


Figura 6.9: Arquitectura de 4 capas MOFScript

A continuación mostramos un ejemplo de transformación mediante el cual podemos observar de forma superficial la sintaxis y estructura del lenguaje MOFScript, para más detalle dirigirse a la guía de usuario disponible en http://modelbased.net/mofscript/docs/MOFScript-User-Guide.pdf. Con la palabra *texttransformation* indicamos cuales, si hay más de uno, son los metamodelos que utilizaremos para hacer la transformación. Para iniciar a definir la transformación debemos crear una regla *main* a partir de la cual comenzará la ejecución. MOFScript ofrece un lenguaje bastante intuitivo, similar a la que estamos acostumbrados a utilizar con ciertos lenguajes de programación orientados a objetos, existen llamadas a métodos con la notación objeto.metodo, expresiones de flujo de control, expresiones condicionales, operaciones para manipulación de cadenas, herencia, polimorfismo y también contamos con el uso de iteradores entre otros.

```
texttransformation MultipleMetaModels (in
       uml:"http://www.eclipse.org/uml2/1.0.0/UML", in
       ec:"http://www.eclipse.org/emf/2002/Ecore") {

main () {
       uml.objectsOfType (uml.Class)->forEach (cl) {
       cl.ecoreModelTest()
}

ec.objectsOfType (ec.EClass)->forEach (eccl) {
       eccl.umlModelTest()
}

}
ec.EClass::ecoreModelTest () {
       stdout.println ("Ecore class: " + self.name)
}

uml.Class::umlModelTest(){
       stdout.println ("Class: " + self.name)
}
```

Parte IV

Trabajo Desarrollado

Capítulo IV

Trabajo Desarrollado

En este capítulo se describe como se ha llevado a cabo el proceso de traducción de un arquetipo expresado sintácticamente en ADL a su representación semántica en OWL.

7.1 Diseño de la solución

El lenguaje ADL describe un arquetipo de manera sintáctica siguiendo las reglas de una gramática. El procesamiento de un fichero escrito en este lenguaje nos devuelve una lista de objetos no relacionados, no aportando significado semántico alguno. Por otro lado, OWL es un lenguaje formal para la construcción de ontologías que permite obtener significado, consistencia e interoperabilidad de la información.

Ambos lenguajes pertenece a un TS específico, el espacio tecnológico de las gramáticas y el de la Web Semántica respectivamente por lo que en el proyecto se trata de establecer una conexión entre ambos espacios tecnológicos usando un espacio tecnológico pivote, MDE[3]. El problema consiste en establecer las transformaciones entre el mundo de las gramáticas y MDE, entre MDE y las ontologías y definir mediante transformaciones de modelos la conexión entre ontología y gramática.

De manera más concreta el trabajo ha consistido en la realización de las siguientes tareas enumeradas del 1-5 y en orden de desarrollo:

1. Obtener el metamodelo Ecore de la ontología. Esta parte la hemos abordado con el uso de la herramienta Protégé que permite la exportación de una ontología en OWL a un conjunto de interfaces Java anotadas que representan el punto de entrada para la obtención de un metamodelo Ecore en EMF. Y el uso de EMF que ofrece la opción de generar un metamodelo EMF a partir de las interfaces generadas con Protégé.

[3] El acrónimo MDE es usado actualmente por la comunidad investigadora internacional cuando se refieren a ideas relacionadas con la ingeniería de modelos sin centrarse exclusivamente en los estándares OMG.

2. Obtener el metamodelo Ecore del lenguaje ADL. El entorno OpenArchitectureWare dispone de la herramienta xText que permite la transición entre el espacio tecnológico de las gramáticas (BNF) y el de los modelos (Ecore). Es decir permite expresar una gramática en una notación similar a EBNF y generar para ella el metamodelo Ecore que la representa.

3. Procesar un fichero ADL y obtener un modelo conforme al metamodelo del punto (2). xText además de generar un metamodelo Ecore a partir de una gramática implementa automáticamente un parser que procesa ficheros conforme a esa gramática para los que obtiene un modelo Ecore conforme al metamodelo anterior. También genera un editor que realiza resaltado de palabras clave, autocompletado, etc. para el lenguaje.

4. Determinar las correspondencias a nivel conceptual entre los metamodelos de ADL y OWL.

5. Implementación de las correspondencias entre los metamodelos en el sentido ADL->OWL en un lenguaje de transformaciones como RubyTL. Este lenguaje se encuentra integrado en el entorno de programación generativa AGE.

6. Generar la representación textual en OWL de un modelo Ecore conforme al metamodelo. Esta transformación de un modelo a su representación textual puede hacerse con distintas herramientas, MOFScript, JET, XPand o plantillas ERB del entorno AGE, entre ellas hemos elegido MOFScript.

A continuación, Figura 7.1 mostramos como quedaría gráficamente el diseño de la solución.

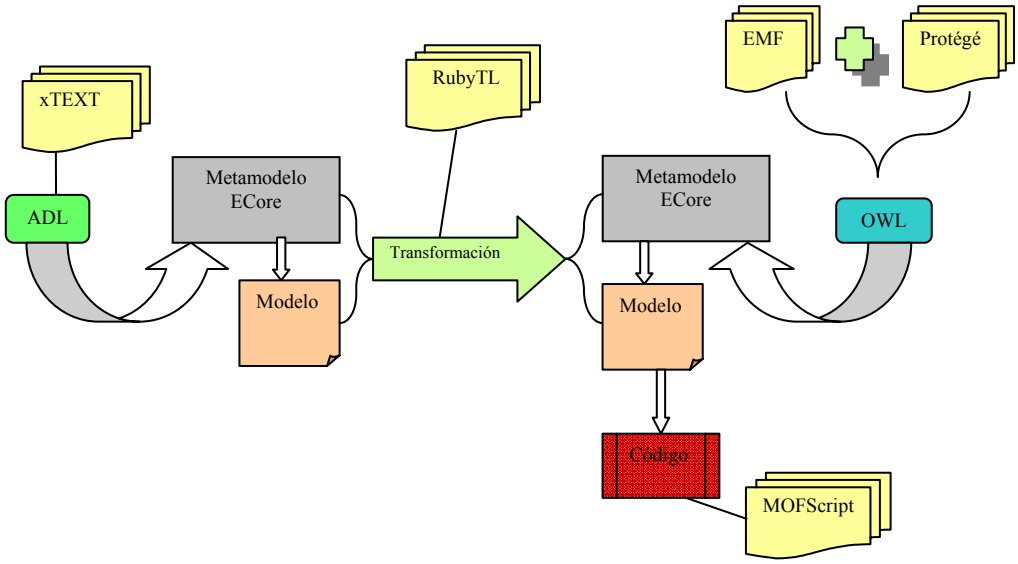


Figura 7.1: Diseño de la solución

7.2 Obtención del metamodelo OWL

Este fue el primer paso realizado en el proyecto. El objetivo consistía en obtener a partir de la ontología CEN-AR-v1.0.owl[4] [4.1], que consta de una serie de clases y atributos que establecen como estructurar la información para construir arquetipos basados en el modelo de referencia del CEN, un metamodelo Ecore. Inicialmente se comprobó si existía la posibilidad de utilizar la herramienta IODT (Integrated Ontology Development Toolkit) de IBM que permite crear directamente un metamodelo a partir de una ontología OWL y viceversa a través de EODM (EMF Ontology Definition Metamodel), sin embargo esto no fue posible debido a la complejidad que presenta la ontología. Por lo tanto para llevar a cabo este objetivo utilizamos las siguientes herramientas:

- Protégé 3.3 (http://protege.stanford.edu/)
- EMF 2.3 (http://www.eclipse.org/modeling/emf)
- Eclipse 3.3 (http://www.eclipse.org/)

La decisión de usar EMF fue tomada en base a la facilidad que ofrecía para generar un metamodelo Ecore a partir de interfaces Java anotadas (6.2.1). Por otro lado conseguir las interfaces java anotadas no tenía porque resultar una tarea complicada si se usaba la herramienta

[4] Ontología desarrollada en el marco del proyecto de investigación POSEACLE: http://klt.inf.um.es/~poseacle/ontologies.html

Protégé (6.1.2). Esta herramienta posee la opción de exportar una ontología a este tipo de interfaces [22].

```
package org.eclipse.owl;
import org.eclipse.emf.common.util.EList;

public interface ARCHETYPE {

    /**
     * Generated from property #uid
     * @model null
     */
    String getUid();

    /**
     * Generated from property #archetype_id
     * @model null
     */
    String getArchetype_id();

    /**
     * Generated from property #has_description
     * @model type="ARCHETYPE_DESCRIPTION"
     */
    ARCHETYPE_DESCRIPTION getHas_description();



    /**
     * Generated from property #translations
     * @model  type="TRANSLATION_DETAIL"
     */
    EList<TRANSLATION_DETAIL> getTranslations();
}
```

Figura 7.2: Ejemplo de interfaz Java generada con Protégé

Inicialmente se pensó que la obtención del metamodelo Ecore debía ser por tanto un paso sencillo y rápido pero nos encontramos con que la versión que usábamos de Protégé, la más actual en ese momento, Protégé 3.3 Beta, no generaba correctamente las interfaces. El problema consistía en que en ciertos métodos de cada interfaz, no se devolvía el tipo correcto, sino un tipo genérico que Protégé denominaba Instance, como se muestra en la Figura 7.3. Ante esto, se decidió resolver el problema, indicando manualmente los tipos correctos y dirigiéndonos al foro de Protégé-OWL para intentar que lo solucionaran y así quizás antes de la finalización del proyecto podríamos tener el problema resuelto ya que de lo que se trataba era de ver como una cuestión que normalmente se hubiera resuelto a través de la programación podía tratarse con un enfoque basado en modelos. Finalmente tuvimos suerte y arreglaron el problema, mientras tanto pudimos continuar con nuestro objetivo.

```
/**
* Generated from property #has_description
* @model type=null
*/
Instance getHas description();
```

```
/**
* Generated from property #has_description
* @model type="ARCHETYPE_DESCRIPTION"
*/
ARCHETYPE_DESCRIPTION getHas_description();
```

Figura 7.3: Problema de tipo Instance con Protégé

Una vez que se obtuvieron las interfaces Java, nos familiarizamos con EMF y nos dispusimos a generar el metamodelo Ecore. Comenzamos creando un proyecto EMF al que importamos el paquete conteniendo las interfaces Java previamente generadas con Protégé, Figura 7.4.

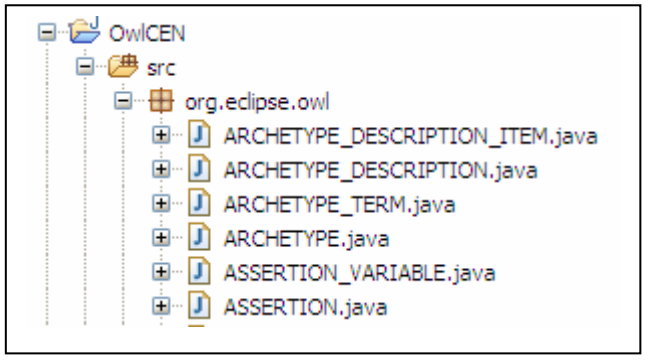


Figura 7.4: Importamos las interfaces Java anotadas

EMF como se ha mencionado permite obtener un metamodelo a partir de estas interfaces, como vemos en la Figura 7.5. Además de crear el metamodelo Ecore, también crea un modelo generador (*.genmodel) a partir del cual es posible generar el código y un editor de modelos de forma automática que nos permitirá crear instancias, es decir, ejemplos concretos con los que trabajar. Al generar el metamodelo con EMF también surgieron algunos conflictos con la ontología que solucionamos con pequeños cambios de la misma.

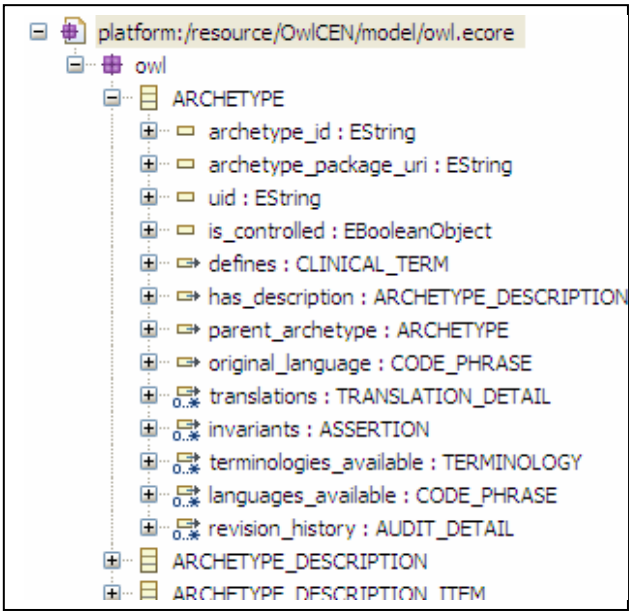


Figura 7.5: Fragmento del metamodelo Ecore

7.3 Obtención del metamodelo ADL

Para obtener el metamodelo Ecore del lenguaje ADL se decidió usar el lenguaje xText disponible en el framework openArchitectureWare (6.2.2). Este lenguaje es de sintaxis parecida a EBNF y puesto que la gramática de ADL es también de este tipo nos decantamos por esta opción. Para la realización de esta fase se emplearon las siguientes herramientas:

- OpenArchitectureWare 4.1.2 (http://www.eclipse.org/gmt/oaw/)
- Eclipse 3.2.2 (http://www.eclipse.org/)

Tras familiarizarnos con el nuevo entorno, comenzamos a escribir una gramática algo simplificada del lenguaje. Para ello creamos un proyecto de tipo xText. Entonces se generan tres proyectos, el proyecto principal que es el que contiene el fichero de definición de la gramática (*xtext), un proyecto editor y otro denominado generador.

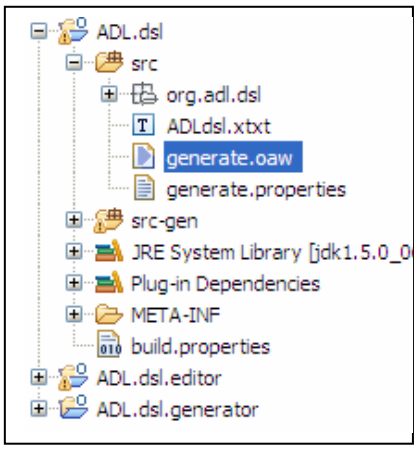


Figura 7.6: gramática xText Project

Este paso resultó de cierta complejidad dada la extensión de la gramática y algunos problemas que fueron surgiendo como la recursividad. El lenguaje xText utiliza el parser ANTLR y este no admite la recursividad por la izquierda. Por ello fue necesario transformar reglas del tipo de la Figura 7.7 a reglas del tipo que aparece en la Figura 7.8:

```
Attr_vals:
      attr_val
      |attr_vals attr_val
      | attr_vals ";" attr_val
;
```

Figura 7.7:Regla recursiva por la izquierda

```
Attr_vals:
      (attrval += Attr_val) (";")? (attrval += Attr_val))*
;
```

Figura 7.8:Misma regla eliminada recursividad

También surgieron ciertos problemas en la definición de cadenas, la sintaxis ADL posee una gran cantidad de cadenas y es difícil representarlas en xText.

```
Native V_LOCAL_TERM_CODE_REF:
  "'['('a'..'z'|'A'..'Z'|'0'..'9')('a'..'z'|'A'..'Z'|'0'..'9'|'.'|'-')*']'"
;
```

Figura 7.9:Ejemplo de cadena expresada en xText

La herramienta xText se encuentra todavía en desarrollo y funciona mejor para definiciones de gramáticas más sencillas. Tampoco existe apenas documentación acerca de cómo utilizar el lenguaje xText, sobre todo en lo que respecta al uso de cadenas. Las dificultades pudieron ser salvadas simplificando la gramática ADL y dirigiéndonos numerosas veces al foro de xText[5].

El lenguaje ADL (3.1) utiliza dos sintaxis diferentes, la parte de definición usa la sintaxis cADL y el resto usan la sintaxis dADL.

[5] Foro de xText: http://www.openarchitectureware.org/forum/index.php?forum=29

```
attr_1 = <
        attr_2 = <
                attr_3 = <leaf_value>
                attr_4 = <leaf_value>
        >
        attr_5 = <
                attr_3 = <
                        attr_6 = <leaf_value>
                >
                attr_7 = <leaf_value>
        >
>
attr_8 = <>
```

```
XXX matches {
        aaa matches {-
                YYY matches {0..3}
        }
        bbb matches {
                ZZZ matches {>1992-12-01}
        }
}
```

Figura 7.11: Estructura típica del lenguaje cADL

Figura 7.10: Estructura típica del lenguaje dADL

Tras simplificar algo la gramática eliminando sobre todo restricciones sobre cadenas, definimos las reglas que la componían, podemos consultar como quedó la gramática en el anexo A, en la Figura 7.12 sólo mostraremos un pequeño fragmento.

```
Model:
      (archetype=Archetype);

Archetype:
      "archetype"       archetypeId = V_ARCHETYPE_ID
      ("specialize"     specializeId = V_ARCHETYPE_ID)? "concept"
conceptCode = V_LOCAL_TERM_CODE_REF
      "description"   description = Attr_vals
      "definition"    definition = C_complex_object
      "ontology"      ontology = Attr_vals
;

Attr_vals:
      (attrval += Attr_val) (";")? (attrval += Attr_val))*
;

Attr_val:
      ((attrid = Attr_id) "=" (objectblock = Object_block))
;

Attr_id:
      (name = ID ) ("(" (value = STRING) ")")?
      | name = "description" ("(" (value = STRING) ")")?
;

C_complex_object:
      (ccomplexobjecthead = C_complex_object_head) "matches" "{"
(ccomplexobjectbody = C_complex_object_body) "}"
;
```

Figura 7.12: Ejemplo parcial gramática ADL

Si comenzamos a construir el árbol sintáctico de esta gramática, la regla *Model* representaría la raíz del árbol y tendría un hijo que se correspondería con la regla *Archetype*, esta a su vez tendría cinco hijos que se corresponden con cada una de las partes de un fichero ADL, y así sucesivamente, Figura 7.13.

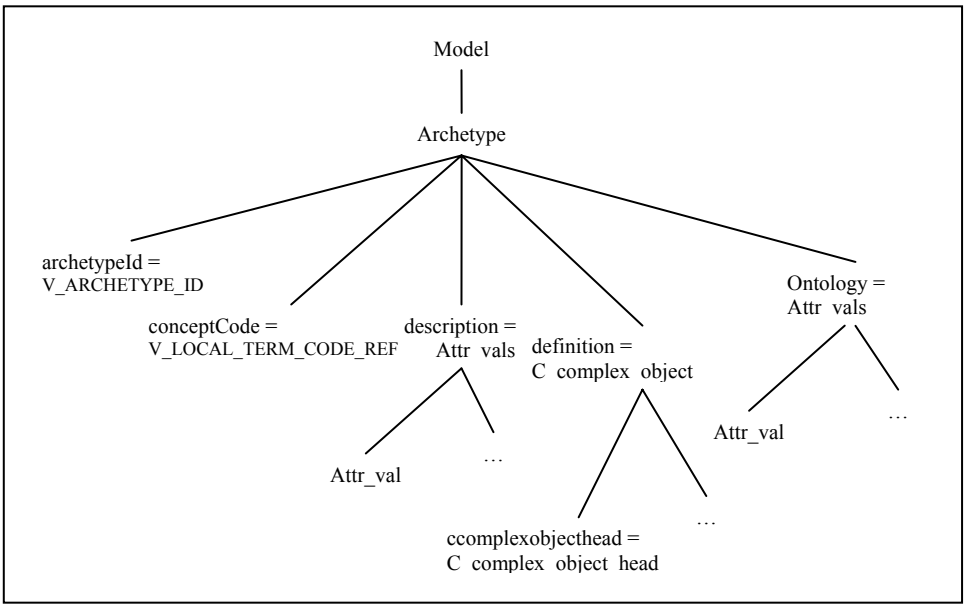


Figura 7.13: Fragmento de árbol sintáctico

Una vez que definimos correctamente la gramática es posible obtener el parser de esta y el metamodelo EMF correspondiente, podemos ver un fragmento de este metamodelo en la Figura 7.14. XText genera una clase por cada regla definida en la gramática así como los atributos y referencias correspondientes.

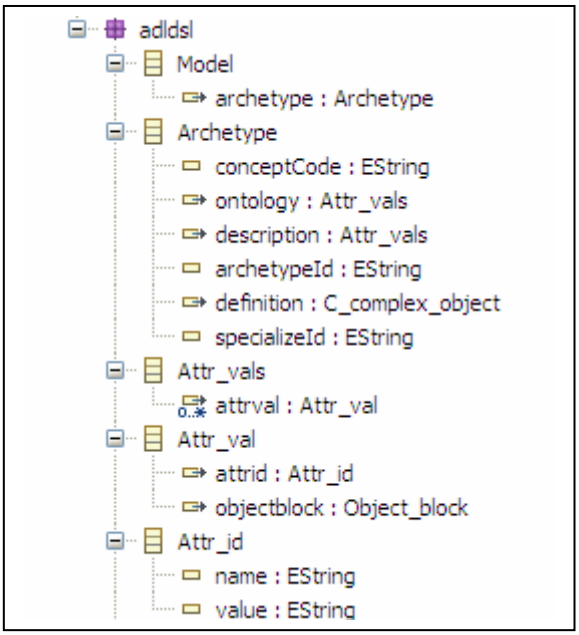


Figura 7.14: Extracto del metamodelo ADL

Además de generarse el metamodelo también se genera un editor de ADL para construir así ejemplos concretos. Este editor permite escribir de forma textual el arquetipo deseado aprovechando las facilidades que proporciona, resaltado de sintaxis, comprobación sintáctica, etc y a partir de él generar el modelo de este automáticamente, se puede ver con más detalle en el siguiente punto.

7.4 Obtención de un modelo ADL

Para proseguir con el proyecto era necesario definir al menos un ejemplo de un arquetipo concreto con el que poder iniciar la traducción del lenguaje ADL al lenguaje OWL. Al contar con el metamodelo y el editor ADL generado en el paso anterior esto no requiere excesiva complicación.

Al obtener el metamodelo ADL también se ha creado un plugin para la edición y creación de modelos. Este plugin permite describir de forma textual los ejemplos ADL como se muestra en la Figura 7.15, ofreciéndonos resaltado de palabras clave, comprobación de la sintaxis, etc. y a partir de ellos generar el modelo de ADL de forma automática.

```
archetype
    CEN-EHR-ENTRY.Colesterol.v1
concept
    [at0000]

description
    author = <"Unknown">
    status = <"draft">
    description("en") = <
        purpose = <"CEN test">
    >

definition

entry[at0000] occurrences matches {1..1} matches {
        items cardinality matches {0..1} matches {
           element[at0001] occurrences matches {0..1} matches {
                value matches {
                    pq occurrences matches {1..1} matches {
                        units matches {
                           cs matches {
                               codeValue matches {"mg/dl"}
                        }
```

Figura 7.15: Ejemplo de un arquetipo escrito en ADL usando el nuevo editor

Una vez escrito un ejemplo en el editor es posible automáticamente crear el modelo correspondiente, Figura 7.16, con la ayuda de la clase XmiWriter de EMF, a través de un fichero de flujo (*.oaw), este modelo se crea en formato XMI.

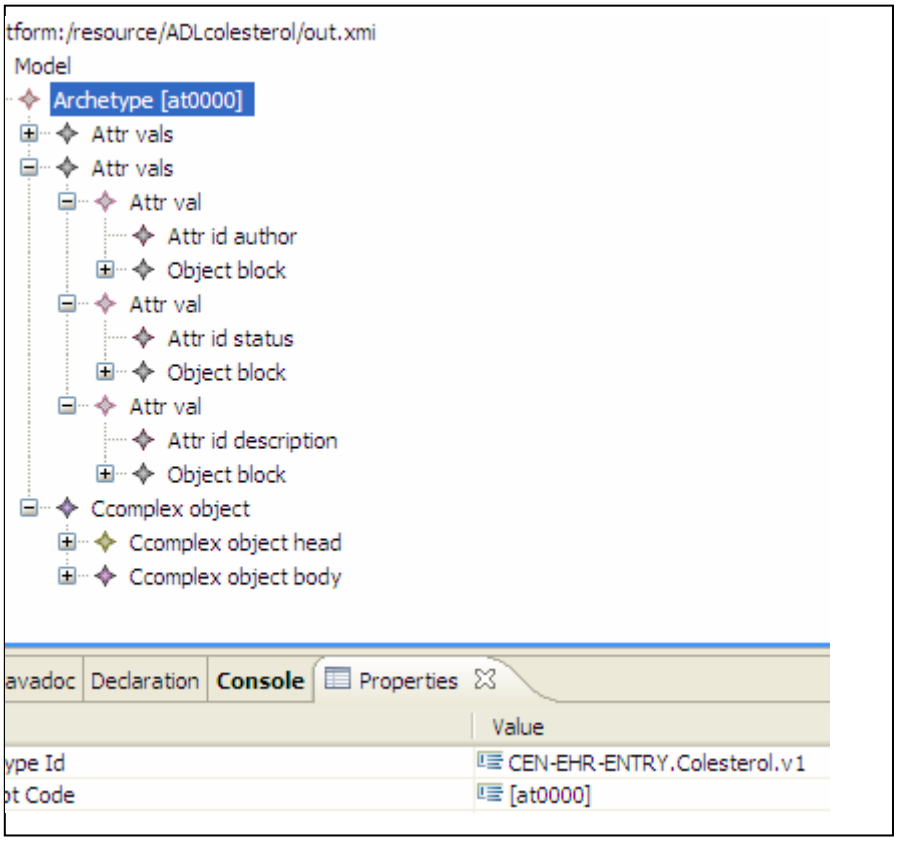


Figura 7.16: Extracto del modelo generado

7.5 Obtención de las reglas de transformación

En esta fase del proyecto ya nos encontramos en el espacio tecnológico de la Ingeniería de Modelos, hemos llevado el espacio de las gramáticas y el de las ontologías a un espacio común, el de MDE. Una vez creados ambos metamodelos, ADL y OWL y un modelo instancia del metamodelo origen de la transformación, ADL, es necesario describir las correspondencias entre ambos metamodelos y las reglas para realizar dicha transformación. Para realizarla hicimos uso del entorno AGE 0.3.1 (http://gts.inf.um.es/trac/age/) (6.2.3) y concretamente del lenguaje de transformación de modelos RubyTL integrado en la misma.

El objetivo de esta fase se puede dividir en dos subobjetivos: i) describir de forma conceptual las correspondencias entre los metamodelos ADL y OWL y ii) escribir una serie de

reglas que permitan traducir las instancias o modelos ADL en instancias o modelos OWL. En los siguientes apartados podemos ver distintas tablas con las correspondencias establecidas, y las reglas de transformación obtenidas.

7.5.1 Correspondencias entre los metamodelos ADL y OWL

Este apartadado intenta mostrar a través de una serie de tablas las relaciones establecidas entre el metamodelo generado de ADL, a la izquierda de cada una de las tablas, y el metamodelo de la ontología OWL.a la derecha de las tablas. A continuación describimos algunos aspectos de cada una de ellas.

Se ha establecido una relación entre la clase *Archetype* de ADL y la clase *ARCHETYPE* de OWL. Se pueden observar las equivalencias entre los distintos atributos de las mismas, destaca el hecho, por ejemplo, de que la propiedad *original_languaje* de OWL pertenece a la clase *CODE_PHRASE*, concretamente a su atributo *code_string* mientras que el ADL es un nodo descendiente de *ontology*.

Archetype	archetypeId	ARCHETYPE	archetype_id
	conceptCode		cen_ENTRY::cen_act_id
	ontology		defines
	description		has_description
	definition		defines
	ontology.attrval.objectblock.singleattrobjectblock.untypedsingleattrobjectblock.primitiveobjectvalue.simplevalue.stringvalue		original_language (CODE_PHRASE::code_string)
	ontology.attrval.objectblock.singleattrobjectblock.untypedsingleattrobjectblock.primitiveobjectvalue.simple_list_value.stringlistvalue.stringvalue		languages_available (CODE_PHRASE::code_string)

En la siguiente tabla dos clases en OWL, *ARCHETYPE_DESCRIPTION* y *ARCHETYPE_DESCRIPTION_ITEM* se corresponden con el mismo nodo ADL, de tipo *Attr_vals.* Este nodo a su vez se encontrará formado por diferentes objetos de tipo *Attr_val* que definirán cada una de las propiedades de las dos clases OWL.

Attr_vals	attrval. objectblock.singleattrobjectblock.untypedsingleattrobjectblock. primitiveobjectvalue.simplevalue.stringvalue	ARCHETYPE_ DESCRIPTION	original_author
	attrval. objectblock.singleattrobjectblock.untypedsingleattrobjectblock. primitiveobjectvalue.simplevalue.stringvalue		lifecycle_state
	attrval.objectblock.singleattrobjectblock.untypedsingleattrobjectblock.attrvals		details
	attrval. objectblock.singleattrobjectblock.untypedsingleattrobjectblock. primitiveobjectvalue.simplevalue.stringvalue		other_details(OTHER_DETAIL::name y value)
	attrval. objectblock.singleattrobjectblock.untypedsingleattrobjectblock. primitiveobjectvalue.simplevalue.stringvalue	ARCHETYPE_ DESCRIPTION _ITEM	purpose

Las clases *cen_ENTRY, cen_ELEMENT, cen_PQ, cen_CS_UNITS y cen_BL* de OWL se corresponden con un nodo ADL de tipo *C_complex_object*, el cual a su vez tiene dos hijos *ccomplexobjecthead* y *ccomplexobjectbody.*Este tipo es el nodo raíz de la parte de definición del arquetipo.

C_complex_ object	Archetype::conceptCode	cen_ENTRY	cen_act_id
	ccomplexobjecthead.coccurrences.occurrencespec		has_occurrence_constraint
	adl.ccomplexobjectbody.cattributes.cattribute.		has_cardinality_constraint
	ccomplexobjectbody.cattributes.cattribute. cattrvalue.cobject.ccomplexobject		cen_items
	Archetype.ontology.attrval. objectblock.singleattrobjectblock.untypedsingleattrobjectblock. attrvals.attrval. objectblock.singleattrobjectblock.untypedsingleattrobjectblock		term_definitions
	ccomplexobjecthead.coccurrences.occurrencespec	cen_ELEMENT	has_occurrence_constraint
	Archetype.ontology.attrval.objectblock.singleattrobjectblock.untypedsingleattr.		term_definitions
	ccomplexobjectbody.cattributes.cattribute.	cen_PQ	has_range_constraint

	cattrvalue.cobject.primitiveobject.cprimitiveobject.creal.crealspec.realintervalvalue		
	ccomplexobjectbody.cattributes.cattribute.cattrvalue.cobject.first.ccomplexobject		cen_units
	ccomplexobjectbody.cattributes.cattribute.cattrvalue.cobject.cprimitiveobject.cprimitiveobject.cstring.cstringconstraint.stringvalue	cen_CS_UNITS	cen_codeValue
	ccomplexobjectbody.cattributes.cattribute.cattrvalue.cobject.cprimitiveobject.cprimitiveobject.cboolean.cbooleanconstraint	cen_BL	has_constraint_datatype

A continuación se establecen las correspondencias entre la clase OWL *TERM_DEFINITION* con sus dos atributos *text* y description y el nodo ADL Untyped_single_attr_object_block, hijo del nodo *Attr_val*.

Untyped_single_attr_object_block	attrvals.attrval.objectblock.singleattrobjectblock.untypedsingleattrobjectblock.primitiveobjectvalue.simplevalue.stringvalue	TERM_DEFINITION	text
	attrvals.attrval.objectblock.singleattrobjectblock.untypedsingleattrobjectblock.primitiveobjectvalue.simplevalue.stringvalue		description

La clase *INTEGER_INTERVAL* de OWL se corresponde con la clase *Occurrence_spec de* ADL y se utiliza para establecer cardinalidades y ocurrencias de los objetos definidos en la parte de definición del arquetipo ADL

.

Occurrence_spec	lower	OCCURRENCE::INTEGER_INTERVAL	lower_bound
	upper		upper_bound

El nodo *C_attribute* en este caso se corresponde con la clase *CARDINALITY* de OWL y se utiliza para establecer la cardinalidad.

C_attribute	cattrhead.ccardinality.cardinalityspec.occurrencespec	CARDINALITY	interval

La clase *REAL_INTERVAL* de OWL describe los límites de un intervalo cuyos extremos son valores reales y se corresponde con los límites del intervalo del nodo ADL *Real_interval_value*.

Real_interval_value	lower	RANGE ::REAL_INTERVAL	lower_bound
	upper		upper_bound

El nodo ADL *C_boolean_constraint* se utiliza para establecer correspondencias de tipo booleano y se corresponde con la clase *C_STRING* en OWL

C_boolean_constraint	valuelow + valuehigh	C_STRING	list

7.5.2 Definición de la reglas de transformación entre los metamodelos ADL y OWL

Para realizar la transformación entre ambos modelos escogimos el lenguaje RubyTL [18] por ser un lenguaje desarrollado recientemente en la misma Universidad de Murcia, además de por las ventajas que ya hemos mencionado que ofrece (6.2.4).

Un proyecto Ruby se encuentra organizado en cuatro partes a través de cuatro carpetas, cada una de ellas con una serie de ficheros:

La carpeta ***helpers*** almacena los ficheros en los que se definen ciertas reglas llamadas decoradores que sirven de ayuda a la hora de definir transformaciones. Estos ficheros son de tipo (*.rb). El metamodelo generado por xText tiene un nivel de abstracción muy bajo, es un metamodelo de la sintaxis concreta de ADL, y por tanto la navegación en modelos conformes a este metamodelo presenta una sintaxis poco legible, los decoradores hacen más legible la transformación. En el proyecto por ejemplo hemos definido un fichero de este tipo para indicar abreviaturas que eviten alargar demasiado el cuerpo de algunos métodos, como el caso que indicamos en la siguiente figura en el que para la metaclase *Attr_val* de ADL podemos utilizar el "seudónimo" *untyped_object_single* para referirnos a una larga hilera de referencias:

```
decorator Adl::Attr_val do
  def untyped_object_single

self.objectblock.singleattrobjectblock.untypedsingleattrobjectblock
  end
end
```

O un decorador para definir un método en cierta metaclase, por ejemplo un método que nos indique si la propiedad value de la metaclase C_complex_object es o no vacío.

```
decorator Adl::C_complex_object do
 def attribute_exist(name)

not adl.ccomplexobjectbody.cattributes.cattribute.select { |attr|
attr.cattrhead.get('id') == "value"}.empty?

 end
end
```

La carpeta ***metamodels*** contiene los metamodelos implicados en la transformación, en nuestro caso los metamodelos Ecore ADL y OWL que hemos generado anteriormente.

La carpeta ***models*** contiene las instancias o modelos de los metamodelos que forman parte de la transformación. En nuestro caso contendrá inicialmente un modelo ADL y posteriormente tras la transformación se creará el correspondiente modelo OWL, ambos en formato XMI.

La carpeta ***transformations*** contiene las reglas de transformación, sus ficheros son de tipo (*.rb).

Por último la carpeta ***tasks*** contiene los ficheros de tipo (*.rake), estos son los ficheros que se encargan de lanzar la transformación. En este fichero se indica la localización de los metamodelos origen y destino de la transformación así como cual es el modelo origen y como se llamará y donde se guardará el modelo destino.

El fichero o ficheros, en nuestro caso uno, donde se definen las reglas de transformación puede consultarse en el anexo C, a continuación mostramos unos pequeños extractos de estas reglas, principalmente de la primera.

La regla principal de la transformación se declara a través de la palabra reservada "top_rule" en este caso se ejecuta entre las metaclases Archetype de ADL y ARCHETYPE de OWL, esto lo indicamos a través de las palabras reservadas "from" y "to". Cada una de las propiedades de esta clase en ADL se asigna a su correspondiente propiedad en el modelo destino OWL.

```
top_rule 'adl2owl' do
   from  Adl::Archetype
   to    Owl::ARCHETYPE
            owl.archetype_id = adl.archetypeId
            owl.has_description=adl.description
   mapping do |adl, owl|
```

El lenguaje RubyTL es un lenguaje de bindings, la parte izquierda de cada asignación representa el modelo destino y la derecha el modelo origen, de esta forma a la propiedad "has_description" del modelo OWL se le asigna la propiedad "description" del modelo ADL, al no ser estas propiedades de tipo primitivo se llamará a la regla que coincida con dichos tipos. Sin embargo si los tipos son primitivos la asignación se realiza directamente.

Es posible también llamar a una regla determinada explícitamente como si fuera una función, esto puede ser útil en ciertas situaciones donde se pueden dar ambigüedades sino lo hacemos de esta forma.

```
owl.defines=entry(adl.definition)
```

A veces las asignaciones son más complejas al encontrarnos con listas de objetos de entre los cuales tenemos que elegir uno determinado, para ello contamos con las funciones select y map. En este caso seleccionamos de una lista aquel atributo cuyo nombre coincide con "primary_language" y lo asignamos a la propiedad "original_language" de "ARCHETYPE" del modelo OWL.

```
owl.original_language=adl.ontology.attrval.select { |attr| attr.attrid.name == "primary_language" }.
              map {
                   |attr| attr.untyped_object_single.singleattrobjectprimitive.primitiveobjectvalue.simplevalue.vstring
               }
```

Otras veces es necesario crear el objeto y luego hacer la asignación. Creamos el objeto a través de la palabra reservada "new" indicándole el tipo "Owl::AUDIT_DETAIL.new", y rellenando sus propiedades (:revision=>…).

```
owl.revision_history = Owl::AUDIT_DETAIL.new(:revision => adl.description.attrval.select {
                |attr| attr.attrid.name == "revision" }.map {
                                                |attr| attr.untyped_object_single.singleattrobjectprimitive.
                                                        primitiveobjectvalue.simplevalue.vstring.value })
```

Un aspecto avanzado que a veces puede surgir a la hora de realizar una transformación es la necesidad de evitar que unas reglas entren en conflicto con otras. Para lograr este objetivo es posible usar fases identificadas de la siguiente forma:

```
phase 'nombre_de_la_fase' do
        --reglas incluidas en la fase
end
```

De esta manera se permite crear una especie de "espacio de reglas" donde todas las reglas incluidas en una fase no entren en conflicto con las reglas de otra fase. En nuestro caso en ADL se utiliza el mismo tipo de nodo, C_Complex_Object, para representar varios tipos de nodos, por ejemplo cen_ENTRY o cen_PQ.

En una regla también pueden establecerse filtros, un filtro es una condición que deben cumplir las instancias que ejecutan dicha regla Definimos por ejemplo un decorador que dice que un "entry" debe comenzar por la palabra "ENTRY".

```
def is_entry?
    self.obj_id.upcase =~ /^ENTRY/
  end
```

Incluimos esta condición en una regla, de manera que sólo ejecutarán esta regla las instancias del modelo ADL de tipo "C_complex_object" que comiencen por "ENTRY". Los filtros pueden usarse también dentro de una regla para establecer la condición de una asignación.

```
rule 'entry' do
    from Adl::C_complex_object
    to   Owl::Cen_ENTRY
    filter { |adl| adl.is_entry? }
    mapping do |adl, owl|
```

7.6 Generación de código OWL

Una vez obtenido el modelo de arquetipos CEN-OWL, podemos ver un ejemplo gráfico para el arquetipo denominado colesterol en la Figura 7.14, sólo nos queda convertirlo a código OWL, para ello existían diferentes herramientas que nos facilitaban este proceso y finalmente elegimos una herramienta basada en MOF, MOFscript [3.1] que ofrece las mismas ventajas que otras herramientas pero se caracteriza por ser más independiente al no encontrarse ligada a un framework concreto como es el caso de xpand de openArchitectureWare. Por lo tanto para el desarrollo de esta fase hemos usado:

- Eclipse 3.2.2 (http://www.eclipse.org/)
- MOFScript 1.2.2 (http://www.modelbased.net/mofscript/)
- ANTLR 4.1.1 (http://antlreclipse.sourceforge.net/updates/features/)
- EMF 2.2.0 (http://www.eclipse.org/modeling/emf)

Hemos usado esta versión de MOFScript que eliminaba el problema que poseía la versión anterior con la operación objectsOfType, y elegido las versiones de las herramientas que no presentaban problemas de incompatibilidades.

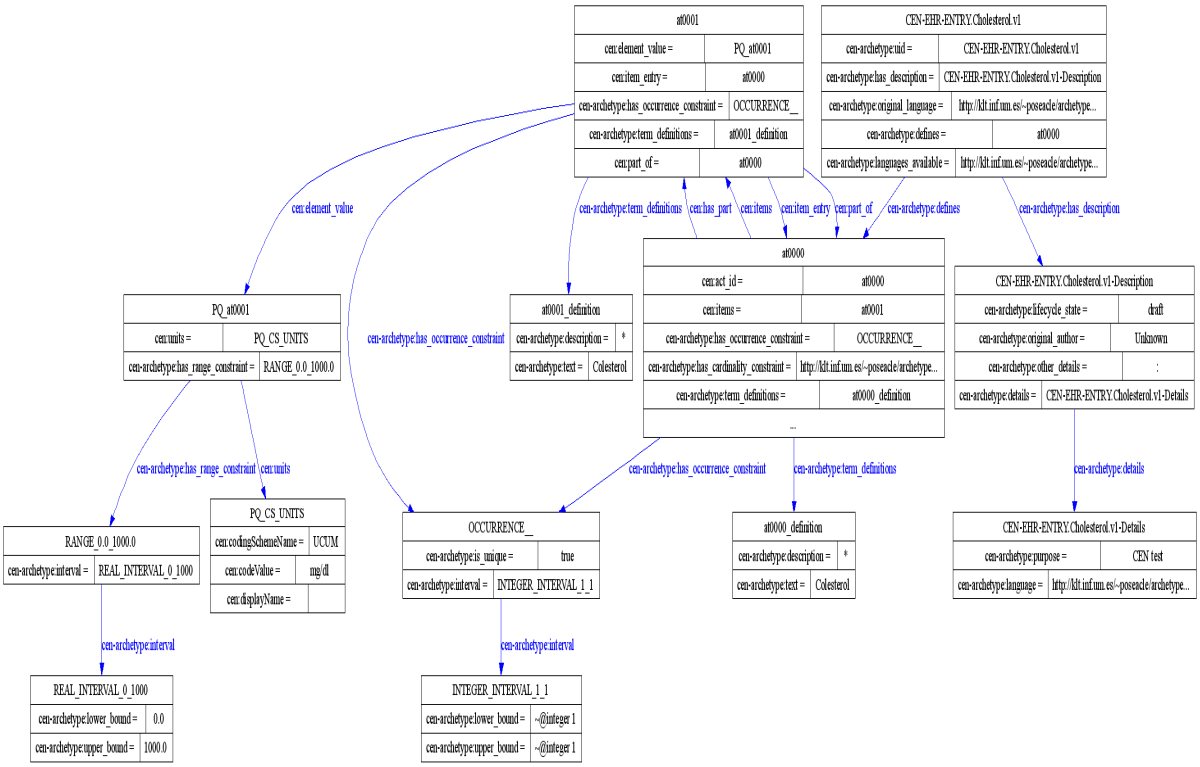


Figura 7.17: Ejemplo del modelo OWL generado para el arquetipo colesterol a partir de ADL

El lenguaje MOFScript nos permite de manera relativamente sencilla extraer la información de un modelo y generar el código correspondiente (6.2.5). Para comenzar a generar

código creamos un proyecto Java en Eclipse y una serie de carpetas para que el proyecto se encontrara organizado ya que los ficheros MOFscript no tienen que estar en una carpeta determinada. Creamos tres carpetas como se puede observar en la Figura 7.18: i) metamodelos donde se almacena el metamodelo del modelo a partir del cual se genera el código deseado, ii) modelos donde se almacenan los modelos de los que vamos a obtener el código y iii) transformaciones donde se almacenan el o los ficheros donde se definen las reglas para generar el código, estos ficheros tendrán la extensión (*.m2t)

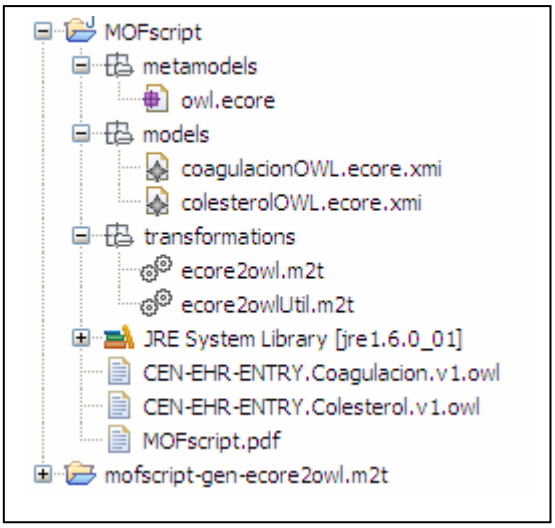


Figura 7.18 : Estructura del proyecto MOFScript

A continuación mostramos un fragmento del fichero de transformaciones donde aparece la regla principal main a partir de la cual se ejecutan el resto de reglas de la transformación. Como observamos la definición de una transformación comienza con la palabra reservada "texttransformation", seguida del nombre de la transformación. A continuación se declaran las URI de los metamodelos de los modelos de entrada, o una sola URI como en este caso ya que sólo necesitamos un metamodelo, el de OWL. A cada URI se le asocia un espacio de nombres que se utilizará para referirse a los elementos del metamodelo, en el ejemplo "ec". Es posible importar otras transformaciones, para ello se usa la palabra reservada "import".

```
import "ecore2owlUtil.m2t"

texttransformation ecore2owl (in ec:"http:///org/eclipse/owl.ecore") {
```

Es posible definir propiedades y variables globales o locales a un método, en este caso hemos definido una serie de propiedades y variables globales. Las propiedades "property" indican un valor constante, mientras que las variables se identifican a través de la palabra reservada "var".

```
property a_string = "\"http://www.w3.org/2001/XMLSchema#string\""
 property a_integer = "\"http://www.w3.org/2001/XMLSchema#integer\""
 property a_float = "\"http://www.w3.org/2001/XMLSchema#float\""
 var nivel:Integer = 2;
 var id_elem:String
 var cont = 0
```

La regla de entrada "main" posee lo que denominamos contexto, en este caso el contexto es la clase del metamodelo denominada "ARCHETYPE", este contexto define el elemento que será el punto de arranque de la transformación. Se ejecutará tantas veces como instancias de ese elemento existan en el modelo, en nuestro caso sólo puede existir una instancia por lo que se ejecutará una sola vez. Dentro de la regla usamos habitualmente la palabra reservada "self", lo hacemos para referirnos al elemento concreto del contexto, es decir a la instancia "ARCHETYPE".

```
ec.ARCHETYPE::main(){

  file (self.archetype_id +".owl")
  self.header()
  gen_tab(nivel)'<cen-archetype:'self.oclGetType()' rdf:ID="'self.archetype_id'">\n'
  gen_tab(nivel+1)'<cen-archetype:has_description rdf:resource="#'self.archetype_id+"-Description"""/>\n'
  gen_tab(nivel+1)'<cen-archetype:uid rdf:datatype='a_string'>'self.archetype_id'</cen-archetype:uid>\n'
  gen_tab(nivel+1)'<cen-archetype:defines
rdf:resource="""#"+self.defines.cen_act_id.substring(1,self.defines.cen_act_id.size()-1)'"/>\n'
          gen_tab(nivel+1)'<cen-archetype:original_language
rdf:resource="""#"+self.original_language.code_string'"/>\n'
  self.languages_available->forEach(c:ec.CODE_PHRASE){
          gen_tab(nivel+1)'<cen-archetype:languages_available rdf:resource="#'c.code_string'"/>\n'
  }
          gen_tab(nivel)'</cen-archetype:'self.oclGetType()'>\n'

          self.has_description.archetype_description(self.archetype_id)
          ec.objectsOfType(ec.CODE_PHRASE)->forEach(c){c.code_phrase(nivel)}
          ec.objectsOfType(ec.OCCURRENCE)->forEach(c){c.constraint(nivel)}
          ec.objectsOfType(ec.CARDINALITY)->forEach(c){c.constraint(nivel)}
          ec.objectsOfType(ec.INTEGER_INTERVAL)->forEach(c){c.interval(nivel)}
          ec.objectsOfType(ec.REAL_INTERVAL)->forEach(c){c.interval(nivel)}
          self.defines.cen_element()
          gen_tab(nivel)'</rdf:RDF>'
}
```

Para generar texto hemos utilizado la salida escape, todo el texto contenido entre los caracteres de escape ‘’ se generará en la salida.

```
gen_tab(nivel)'<cen-archetype:'self.oclGetType()' rdf:ID="'self.archetype_id'">\n'
  gen_tab(nivel+1)'<cen-archetype:has_description rdf:resource="#'self.archetype_id+"-Description"""/>\n'
  gen_tab(nivel+1)'<cen-archetype:uid rdf:datatype='a_string'>'self.archetype_id'</cen-archetype:uid>\n'
```

La función gen_tab(nivel) la importamos desde “ecore2owlUtil.m2t” y se encarga de imprimir las tabulaciones que le indicamos con la variable nivel.

```
module::gen_tab(nivel:Integer) {
    nivel->forEach(n){tab}
  }
```

Finalmente mostramos un fragmento de código generado para el modelo del colesterol.

```
…
<owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://klt.inf.um.es/~poseacle/CEN-AR-v1.0.owl"/>
</owl:Ontology>

<cen-archetype:ARCHETYPE rdf:ID="CEN-EHR-ENTRY.Colesterol.v1">
    <cen-archetype:has_description rdf:resource="#CEN-EHR-ENTRY.Colesterol.v1-Description"/>
    <cen-archetype:uid rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
                                        CEN-EHR-ENTRY.Colesterol.v1
    </cen-archetype:uid>
    <cen-archetype:defines rdf:resource="#at0000"/>
    <cen-archetype:original_language rdf:resource="#es"/>
    <cen-archetype:languages_available rdf:resource="#es"/>
</cen-archetype:ARCHETYPE>
…
```

Parte V

Conclusiones

Capítulo V

Conclusiones

8.1 Conclusiones y trabajo futuro

Tener acceso a la información clínica de los pacientes es una necesidad básica de cualquier profesional del mundo sanitario. Frecuentemente esta información se encuentra distribuida en diferentes sistemas y representada en cada uno de ellos de distinta forma. Los arquetipos, concretamente los arquetipos clínicos, surgieron con el objetivo de resolver esta situación. Un arquetipo describe un concepto clínico y se considera la unidad básica de intercambio de información clínica. ADL fue creado como un lenguaje para describir la sintaxis de dichos arquetipos. Al ser un lenguaje sintáctico se ha observado que presenta ciertos inconvenientes como el no garantizar la consistencia de la información clínica que representa, garantizándola solo a nivel de la sintaxis del lenguaje recayendo así la mayoría del esfuerzo a la hora de trabajar con ellos en los desarrolladores. Por ello se pensó que sería conveniente tener los arquetipos expresados en un lenguaje semántico que permitiera procesar la información de manera más inteligente. Es en este momento en el que entra en juego la Web Semántica y concretamente el lenguaje de ontologías OWL, como una forma de obtener mayores ventajas a la hora de representar los arquetipos. A lo largo del proyecto se ha descrito como se representan los arquetipos mediante ADL y OWL, observando los beneficios que se podían obtener mediante la segunda representación.

Actualmente la mayoría de los arquetipos se encuentran expresados mediante ADL por lo que en el proyecto hemos obtenido un mecanismo para traducir dichos arquetipos a otros expresados mediante el lenguaje de ontologías OWL usando como espacio tecnológico la Ingeniería de Modelos (MDE). La traducción no ha consistido en una traducción sintáctica como ya se había realizado en otros trabajos [23], sino en una traducción basada en la interpretación semántica. Ambos lenguajes, ADL y OWL, pertenecen a un TS específico, el espacio tecnológico de las gramáticas y el de la Web Semántica respectivamente por lo que en el proyecto se trata de establecer una conexión entre ambos espacios. El problema consiste en establecer las transformaciones entre el mundo de las gramáticas y MDE, entre MDE y las ontologías y definir mediante transformaciones de modelos la conexión entre ontología y gramática.

En el proyecto hemos usado xText para pasar del mundo de las gramáticas al mundo de MDE. Este lenguaje es de sintaxis parecida a EBNF y puesto que la gramática de ADL es también de este tipo nos decantamos por esta opción. Mediante xText ha sido posible describir una gramática de la sintaxis del lenguaje ADL, a pesar de algunas dificultades dada su complejidad y el hecho de utilizar una tecnología actualmente en desarrollo como hemos mencionado [7.4], y a partir de ella un metamodelo para el lenguaje y un editor textual dentro de la plataforma Eclipse con el cual es posible crear distintas instancias del metamodelo.

Para conseguir el paso del mundo de la Web Semántica, concretamente de las ontologías, al de MDE, se valoró inicialmente la posibilidad de utilizar la herramienta IODT (Integrated Ontology Development Toolkit) de IBM que permite crear directamente un metamodelo a partir de una ontología OWL y viceversa a través de EODM (EMF Ontology Definition Metamodel), que es una implementación del estándar ODM[6] construido sobre EMF, sin embargo esto no fue posible debido a la complejidad que presenta la ontología ya que la herramienta no se encuentra suficientemente madura. Esto nos llevó a buscar otras alternativas como usar el API de Jena, sin embargo esta solución requería un alto esfuerzo de implementación y un difícil mantenimiento comparada con una solución basada en la generación de plantillas. Finalmente optamos por utilizar EMF para obtener el metamodelo de la ontología OWL a partir de una serie de interfaces Java generadas con Protégé y utilizar el lenguaje de plantillas MOFscript para obtener el código OWL a partir de los modelos obtenidos.

Finalmente una vez tenemos ambos espacios tecnológicos expresados a través de un mismo espacio, MDE, la conexión entre ontología y gramática la llevamos a cabo a través del lenguaje de transformación de modelos RubyTL, estableciendo las correspondencias entre ambos metamodelos obtenidos.

El proyecto inicia los primeros pasos para conseguir la interoperabilidad semántica entre diferentes sistemas de información clínica. Utilizar tecnologías semánticas ofrece ciertas ventajas ya que no es necesario reemplazar la tecnología actual y es posible obtener mayores ventajas en el procesamiento de la información clínica. En este proyecto la interpretación semántica de los arquetipos expresados en OWL se ha hecho según el estándar de HCE del CEN pero es posible aplicar la misma metodología pero basándose en el estándar del OpenEHR, así como establecer las correspondencias entre ambos estándares para facilitar la transformación de

[6] ODM (Ontology Definition Metamodel) del OMG, define el metamodelo OWL y como obtener a partir de el otros metamodelos como el de UML.

los arquetipos basados en CEN en otros basados en OpenEHR facilitándose la interoperabilidad entre diferentes sistemas.

También es una posible vía futura realizar el paso de ADL a OWL en dos pasos: ADL→AOM[7]→OWL de forma que logremos que el paso ADL→AOM sea independiente del estándar de HCE utilizado y el dependiente sea el paso de AOM→OWL. De esta forma podríamos realizar el paso de ADL a OWL según el estándar del CEN y el del OpenEHR partiendo del mismo metamodelo, que habríamos generado a partir del AOM.

[7] AOM (Archetype Object Model) , Modelo de Arquetipos

Apéndice A

A.1 Grámatica simplificada de la sintaxis del lenguaje de arquetipos ADL escrita en xText

```
Model:
      (archetype=Archetype);

Archetype:
      "archetype"       archetypeId = V_ARCHETYPE_ID
      ("specialize"     specializeId = V_ARCHETYPE_ID)? "concept"
conceptCode = V_LOCAL_TERM_CODE_REF
      "description"   description = Attr_vals
      "definition"    definition = C_complex_object
      "ontology"      ontology = Attr_vals
;

Attr_vals:
      (attrval += Attr_val) (";")? (attrval += Attr_val))*
;

Attr_val:
      ((attrid = Attr_id) "=" (objectblock = Object_block))
;

Attr_id:
      (name = ID ) ("(" (value = STRING) ")")?
      | name = "description" ("(" (value = STRING) ")")?
;

Abstract Object_block:
      singleattrobjectblock = Single_attr_object_block
      |multipleattrobjectblock = Multiple_attr_object_block
;

Abstract Multiple_attr_object_block:
      untypedmultipleattrobjectblock =
Untyped_multiple_attr_object_block
      | name = ID  (untypedmultipleattrobjectblock =
Untyped_multiple_attr_object_block)
;

Abstract Untyped_multiple_attr_object_block:
      "<" (keyedobjects = Keyed_objects) ">"
;

Keyed_objects:
      (keyedobject += Keyed_object) (keyedobject += Keyed_object)*
;

Keyed_object: (objectkey = Object_key) "=" (objectblock =
Object_block)
;
```

```
Object_key: "[" (simplevalue = Simple_value) "]"
;

Abstract Single_attr_object_block:
      (untypedsingleattrobjectblock =
Untyped_single_attr_object_block)
      | name = ID (untypedsingleattrobjectblock =
Untyped_single_attr_object_block)
;

Abstract Untyped_single_attr_object_block:
      "<"">"
      |"<" (attrvals = Attr_vals) ">"
      |"<" (primitiveobjectvalue = Primitive_object_value) ">"
;

Primitive_object_value:
      (simplevalue = Simple_value)
      |(simple_list_value = Simple_list_value)
      |(simpleintervalvalue = Simple_interval_value)
      |(query = Query)
;

Query: "query" "(" name = STRING "," name = STRING ")"
;

Simple_value:
      (stringvalue = STRING)
      | (integervalue = INT)
      | (realvalue = V_REAL)
      | (booleanvalue = Boolean_value)
      | (charactervalue = V_CHAR)
;

Abstract Simple_list_value:
      (stringlistvalue = String_list_value)
      |(integerlistvalue = Integer_list_value)
      |(reallistvalue = Real_list_value)
      |(booleanlistvalue = Boolean_list_value)
      |(characterlistvalue = Character_list_value)
;

Abstract Simple_interval_value:
      (integerintervalvalue = Integer_interval_value)
      |(realintervalvalue = Real_interval_value)
;

String_list_value:
      (stringvalue = STRING) ',' ((stringvalue = STRING) | '...') (','
(stringvalue = STRING))*
;

Integer_list_value:
      (value = INT) ',' ((value = INT)|'...') (',' (value = INT))*
;

Integer_interval_value:
      "|"(lower = INT) ".." (upper = INT) "|"
      | "|" "<" (value = INT) "|"
      | "|" "<=" (value = INT) "|"
      | "|" ">" (value = INT) "|"
```

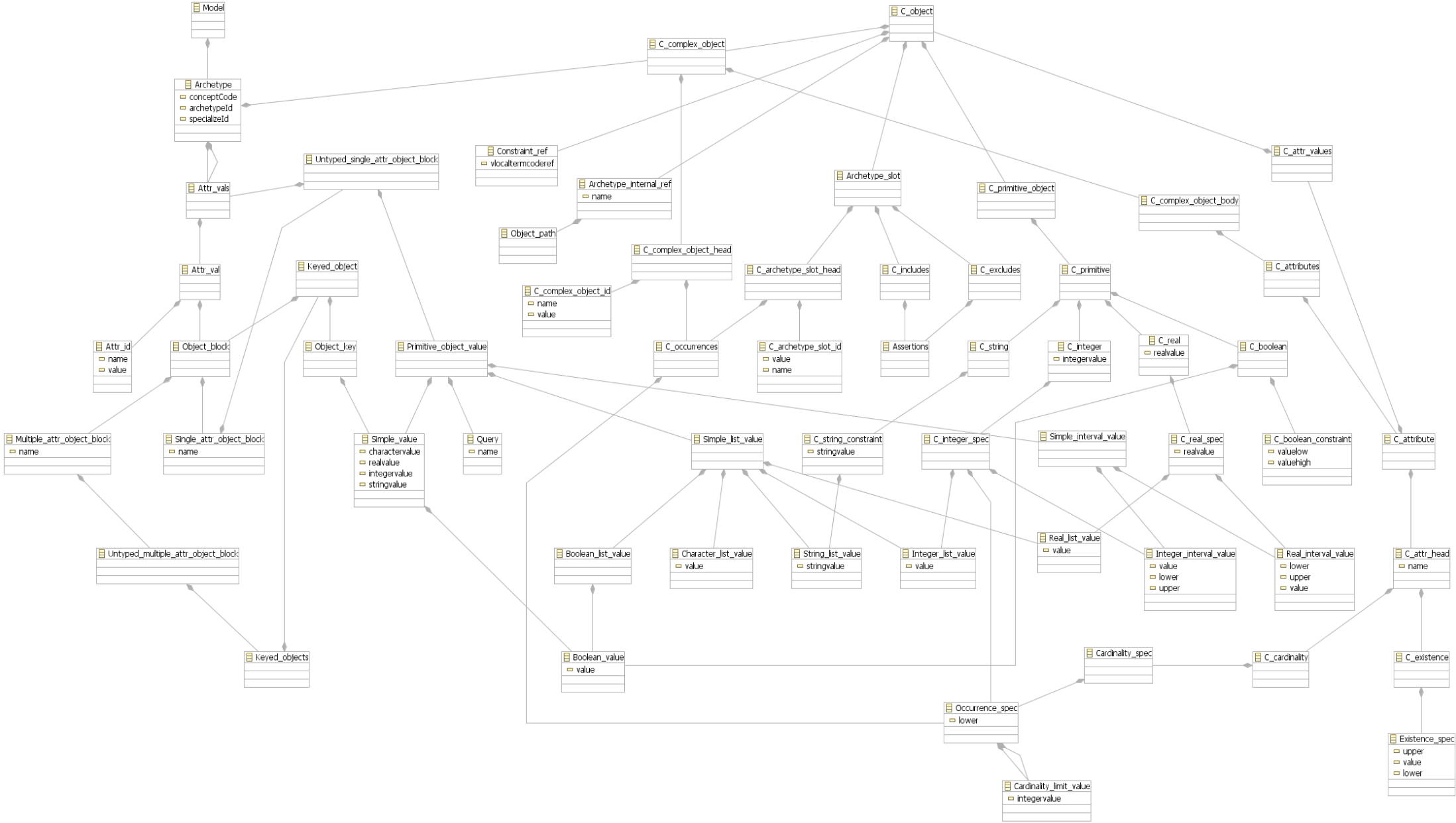
```
      | "|" ">=" (value = INT) "|"
;


Real_list_value:
      value = V_REAL ',' (value = V_REAL|'...') (',' value = V_REAL)*
;

Real_interval_value:
      "|" (lower=V_REAL) ".." (upper=V_REAL) "|"
      | "|" "<" (value = V_REAL) "|"
      | "|" "<=" (value = V_REAL) "|"
      | "|" ">" (value = V_REAL) "|"
      | "|" ">=" (value = V_REAL) "|"
;

Boolean_value:
      value = "True"
      | value = "False"
;

Boolean_list_value:
      value = Boolean_value ',' (value = Boolean_value|'...') (','
value = Boolean_value)*
;

Character_list_value:
      value = V_CHAR ',' (value = V_CHAR|'...') (',' value = V_CHAR)*
;
 /*############################CADL_SYNTAX############################*/

C_complex_object:
      (ccomplexobjecthead = C_complex_object_head) "matches" "{"
(ccomplexobjectbody = C_complex_object_body) "}"
;

C_complex_object_head:
      (ccomplexobjectid = C_complex_object_id) (coccurrences =
C_occurrences)?
;

C_complex_object_id:
      (name = ID) (value = V_LOCAL_TERM_CODE_REF)?
;

C_complex_object_body:
      "*"
      | (cattributes = C_attributes)
;

Abstract C_object:
      (ccomplexobject = C_complex_object)
      |(archetypeinternalref = Archetype_internal_ref)
      |(archetypeslot = Archetype_slot)
      |(constrainref = Constraint_ref)
      |(cprimitiveobject = C_primitive_object)

;

Archetype_internal_ref:
      "use_node" name = V_TYPE_ID (objectpath = Object_path)
```

```
;

Archetype_slot:
      (carchetypeslothead = C_archetype_slot_head) "matches" "{"
(cincludes = C_includes)? (cexcludes = C_excludes)? "}"
;

C_archetype_slot_head: (carchetypeslotid = C_archetype_slot_id)
(coccurrences = C_occurrences)?
;

C_archetype_slot_id:
      "allow_archetype" name = V_TYPE_ID (value =
V_LOCAL_TERM_CODE_REF)?
;

C_primitive_object:
      (cprimitiveobject = C_primitive)
;

Abstract C_primitive:
      (cinteger = C_integer)
      |(creal = C_real)
      |(cstring = C_string)
      |(cboolean = C_boolean)
;

C_attributes:
      (cattribute += C_attribute) (cattribute += C_attribute)*
;

C_attribute:
      (cattrhead = C_attr_head) "matches" "{" (cattrvalue =
C_attr_values) "}"
;

C_attr_head:
      (name =ID) (cexistence = C_existence)? (ccardinality =
C_cardinality)?
;

C_attr_values:
      (cobject += C_object) (cobject += C_object)*
;

C_includes:
       "include" (INCLUDES+=Assertions)*
;

C_excludes:
       "exclude" (EXCLUDES+=Assertions)*
;

C_existence:
       "existence" "matches" "{" (existencespec = Existence_spec) "}"
;

Existence_spec:
      value = INT
      | lower = INT ".." upper = INT
;
```

```
C_cardinality: "cardinality" "matches" "{" (cardinalityspec =
Cardinality_spec)"}"
;

Cardinality_spec:
      (occurrencespec = Occurrence_spec)
      | (occurrencespec = Occurrence_spec) ";" "ordered"
      | (occurrencespec = Occurrence_spec) ";" "unordered"
      | (occurrencespec = Occurrence_spec) ";" "unique"
      | (occurrencespec = Occurrence_spec) ";" "ordered" ";" "unique"
      | (occurrencespec = Occurrence_spec) ";" "unordered" ";"
"unique"
      | (occurrencespec = Occurrence_spec) ";" "unique" ";" "ordered"
      | (occurrencespec = Occurrence_spec) ";" "unique" ";"
"unordered"
;

Cardinality_limit_value:
      (integervalue = INT) | "*"
;

C_occurrences:
       "occurrences" "matches" "{" (occurrencespec = Occurrence_spec)
"}"
;

Occurrence_spec:
      (limitvalue = Cardinality_limit_value)
      | (lower = INT) ".." (upper = Cardinality_limit_value)
;

C_integer_spec:
      (integerlistvalue = Integer_list_value)
      | (integerintervalvalue = Integer_interval_value)
      | (occurrencespec = Occurrence_spec)
;

C_integer:
      (cintegerspec = C_integer_spec) (";" integervalue = INT)?
;

C_real_spec:
      (realvalue = V_REAL)
      | (reallistvalue = Real_list_value)
      | (realintervalvalue = Real_interval_value)
;

C_real:
      (crealspec = C_real_spec) (";" realvalue = V_REAL)?
;

C_string_constraint:
      (stringvalue = STRING)
      |(stringlistvalue = String_list_value)  ("..")?
;

C_string:
      (cstringconstraint = C_string_constraint) (';' cstringconstraint
= C_string_constraint)?
;
```

```
C_boolean_constraint:
      valuelow = "True"
      | valuelow = "False"
      | valuelow="True" "," valuehigh="False"
      | valuelow="False" "," valuehigh="True"
;

C_boolean:
      (cbooleanconstraint = C_boolean_constraint) (";" booleanvalue =
Boolean_value)?
;

Constraint_ref:
      (vlocaltermcoderef = V_LOCAL_TERM_CODE_REF)
;

Object_path:
      "path"
;

Assertions:
      "assertions"
;


Native V_ARCHETYPE_ID:
      "('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'-
')+'.'('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'-
')+'.'('a'..'z'|'A'..'Z'|'0'..'9')+"
;

Native V_TYPE_ID:
      "('A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*"
;

Native V_LOCAL_TERM_CODE_REF:
      "'['('a'..'z'|'A'..'Z'|'0'..'9')('a'..'z'|'A'..'Z'|'0'..'9'|'.'|
'-')*']'"
;

Native V_CHAR:
      "('a'..'z')"
;

Native V_REAL:
      "('0'..'9')+'.'('0'..'9')+"
;
```

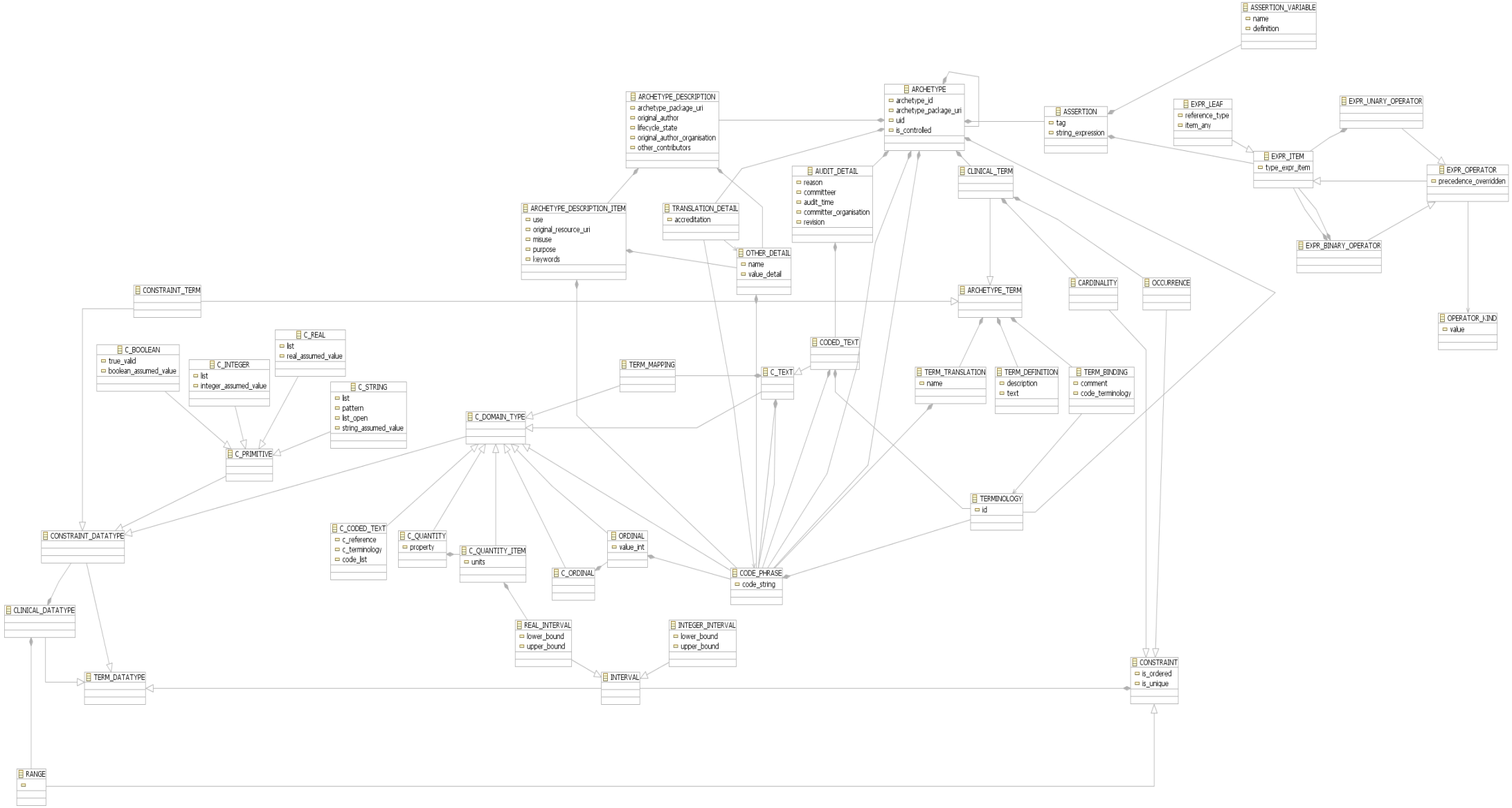
A.2. Metamodelo ADL


Model
C_object
C_complex_object
Archetype
conceptCode
archetypeId
specializeId
Constraint_ref
vlocaltermcoderef
Untyped_single_attr_object_block
C_attr_values
Attr_vals
Archetype_internal_ref
name
Archetype_slot
C_primitive_object
C_complex_object_body
Object_path
C_complex_object_head
C_archetype_slot_head
C_includes
C_excludes
C_primitive
C_attributes
Attr_val
Keyed_object
C_complex_object_id
name
value
Attr_id
name
value
Object_block
Object_key
Primitive_object_value
C_occurrences
C_archetype_slot_id
value
name
Assertions
C_string
C_integer
integervalue
C_real
realvalue
C_boolean
Multiple_attr_object_block
name
Single_attr_object_block
name
Simple_value
charactervalue
realvalue
integervalue
stringvalue
Query
name
Simple_list_value
C_string_constraint
stringvalue
C_integer_spec
Simple_interval_value
C_real_spec
realvalue
C_boolean_constraint
valuelow
valuehigh
C_attribute
Untyped_multiple_attr_object_block
Boolean_list_value
Character_list_value
value
String_list_value
stringvalue
Integer_list_value
value
Real_list_value
value
Integer_interval_value
value
lower
upper
Real_interval_value
lower
upper
value
C_attr_head
name
Keyed_objects
Boolean_value
value
Cardinality_spec
C_cardinality
C_existence
Occurrence_spec
lower
Existence_spec
upper
value
lower
Cardinality_limit_value
integervalue

Apéndice B

B.1 Metamodelo OWL

A continuación se muestra el metamodelo OWL, sólo se indican las clases principales del modelo, no encontrándose reflejadas los tipos de datos del CEN.


ASSERTION_VARIABLE
name
definition
ARCHETYPE_DESCRIPTION
archetype_package_uri
original_author
lifecycle_state
original_author_organisation
other_contributors
ARCHETYPE
archetype_id
archetype_package_uri
uid
is_controlled
ASSERTION
tag
string_expression
EXPR_LEAF
reference_type
item_any
EXPR_UNARY_OPERATOR
EXPR_ITEM
type_expr_item
EXPR_OPERATOR
precedence_overridden
EXPR_BINARY_OPERATOR
AUDIT_DETAIL
reason
committeer
audit_time
committer_organisation
revision
CLINICAL_TERM
ARCHETYPE_DESCRIPTION_ITEM
use
original_resource_uri
misuse
purpose
keywords
TRANSLATION_DETAIL
accreditation
OTHER_DETAIL
name
value_detail
CONSTRAINT_TERM
ARCHETYPE_TERM
CARDINALITY
OCCURRENCE
OPERATOR_KIND
value
C_BOOLEAN
true_valid
boolean_assumed_value
C_INTEGER
list
integer_assumed_value
C_REAL
list
real_assumed_value
C_STRING
list
pattern
list_open
string_assumed_value
CODED_TEXT
TERM_MAPPING
C_TEXT
TERM_TRANSLATION
name
TERM_DEFINITION
description
text
TERM_BINDING
comment
code_terminology
C_DOMAIN_TYPE
C_PRIMITIVE
TERMINOLOGY
id
CONSTRAINT_DATATYPE
C_CODED_TEXT
c_reference
c_terminology
code_list
C_QUANTITY
property
C_QUANTITY_ITEM
units
ORDINAL
value_int
C_ORDINAL
CODE_PHRASE
code_string
CLINICAL_DATATYPE
REAL_INTERVAL
lower_bound
upper_bound
INTEGER_INTERVAL
lower_bound
upper_bound
CONSTRAINT
is_ordered
is_unique
TERM_DATATYPE
INTERVAL
RANGE

Apéndice C

C.1. Reglas de Transformación RubyTL

```
# Write your rules here. Use template to ease your work.
load_helper 'helpers/helper.rb'

transformation 'adl2owl'

input 'Adl' => 'http://www.example.org/ADL/dsl'
output  'Owl' => 'http:///org/eclipse/owl.ecore'


decorator Adl::C_complex_object do

  def obj_id
    ccomplexobjecthead.ccomplexobjectid.name
  end

  def obj_id_value
    ccomplexobjecthead.ccomplexobjectid.value
  end

  def extract_name
    raise "Error #{self.obj_id_value.upcase}" if not self.obj_id_value.upcase
=~ /^\[[A-Z0-9]+\]/
    cadena=self.obj_id_value.split("[")[1]
    cadena.slice(0,cadena.length-1)
  end

  def is_entry?
    self.obj_id.upcase =~ /^ENTRY/
  end

  def is_element?
    self.obj_id.upcase == "ELEMENT"
  end

  def is_pq?
    self.obj_id.upcase == "PQ"
  end

  def is_bl?
    self.obj_id.upcase == "BL"
  end

  def is_cs?
    self.obj_id.upcase == "CS"
  end

end









phase 'header' do
```

```
top_rule 'adl2owl' do
  from  Adl::Archetype
  to    Owl::ARCHETYPE

  mapping do |adl, owl|
     owl.archetype_id = adl.archetypeId
     owl.has_description=adl.description
     owl.original_language= Owl::CODE_PHRASE.new
                 (:code_string => adl.ontology.attrval.
                 select { |attr| attr.attrid.name == "primary_language" }.
                  map { |attr| attr.untyped_object_single.
                 primitiveobjectvalue.simplevalue.stringvalue })

     owl.languages_available=Owl::CODE_PHRASE.new(:code_string =>
                 adl.ontology.attrval.select
                 { |attr| attr.attrid.name == "languages_available" }.
                  map { |attr| attr.untyped_object_single.
                 primitiveobjectvalue.simple_list_value.
                 stringlistvalue.stringvalue})

     owl.revision_history = Owl::AUDIT_DETAIL.new(:revision =>
                 adl.description.attrval.
                 select { |attr| attr.attrid.name == "revision" }.
                 map { |attr| attr.untyped_object_single.
                 primitiveobjectvalue.simplevalue.stringvalue })

     owl.defines=entry(adl.definition)
  end
end



rule 'Attr_Vals2ARCHETYPE_DESCRIPTION' do
  from    Adl::Attr_vals
  to      Owl::ARCHETYPE_DESCRIPTION
  mapping do |adl, owl|
    owl.original_author = adl.attrval.
                 select { |attr| attr.attrid.name == "author" }.
                  map { |attr| attr.untyped_object_single.
                 primitiveobjectvalue.simplevalue.stringvalue }.join(", ")

     owl.lifecycle_state = adl.attrval.
                 select { |attr| attr.attrid.name == "status" }.
                  map { |attr| attr.untyped_object_single.
                 primitiveobjectvalue.simplevalue.stringvalue }.join(", ")

    details_attr = adl.attrval.
                 select { |attr| attr.attrid.name == "description" }.first

    owl.details  = details_attr.objectblock.singleattrobjectblock.
                 untypedsingleattrobjectblock.attrvals

    owl.details.language = Owl::CODE_PHRASE.new(:code_string =>
                 details_attr.attrid.value)

    owl.other_details=Owl::OTHER_DETAIL.new(:name => adl.attrval.
                 select {|attr| attr.attrid.name == "adl_version" }.
                  map { |attr| attr.attrid.name } ,
                 :value_detail => adl.attrval.
                 select { |attr| attr.attrid.name == "adl_version" }.
                  map { |attr| attr.untyped_object_single.
                 primitiveobjectvalue.simplevalue.stringvalue })

  end
 end



rule 'Attr_Vals2ARCHETYPE_DESCRIPTION_ITEM' do
```

```
    from    Adl::Attr_vals
    to      Owl::ARCHETYPE_DESCRIPTION_ITEM
    mapping do |adl, owl|
        owl.purpose=adl.attrval.select {|attr| attr.attrid.name == "purpose"}.
                    map {|attr| attr.untyped_object_single.
                    primitiveobjectvalue.simplevalue.stringvalue}

    end
  end



  rule 'entry' do
    from Adl::C_complex_object
    to   Owl::Cen_ENTRY
    filter { |adl| adl.is_entry? }
    mapping do |adl, owl|
      archetype = Adl::Archetype.all_objects.first

       owl.cen_act_id = archetype.conceptCode
      if adl.ccomplexobjecthead.coccurrences
        owl.has_occurrence_constraint = adl.ccomplexobjecthead.coccurrences.
                                        occurrencespec
      end
      owl.has_cardinality_constraint = adl.ccomplexobjectbody.cattributes.
             cattribute.select { |attr| attr.cattrhead.name == "items"}.first

       owl.cen_items = adl.ccomplexobjectbody.cattributes.cattribute.
              select { |attr|attr.cattrhead.name == "items"}.
              first.cattrvalue.cobject.
               map { |cobj| cobj.ccomplexobject }

       name = adl.extract_name
      owl.term_definitions = Adl::Archetype.all_objects.first.
                    ontology.attrval.
                    select { |attr| attr.attrid.name == "term_definitions"
}.first.
                    untyped_object_single.attrvals.attrval.
                    select { |attr| attr.attrid.name == 'items' }.
                    select { |attr| attr.attrid.value == name }.
                    map { |attr| attr.untyped_object_single }
    end
  end



  rule 'Untyped_single_attr_object_block2TERM_DEFINITION' do
    from    Adl::Untyped_single_attr_object_block
    to      Owl::TERM_DEFINITION
    mapping do |adl, owl|
          owl.text = adl.attrvals.attrval.
                    select { |attr| attr.attrid.name == "text"}.first.
                    untyped_object_single.primitiveobjectvalue.
                    simplevalue.stringvalue

           owl.description = adl.attrvals.attrval.
                    select { |attr| attr.attrid.name == "description"}.first.
                    untyped_object_single.primitiveobjectvalue.
                    simplevalue.stringvalue
    end
  end



  rule 'C_complex_object2OCCURRENCE' do
    from    Adl::Occurrence_spec
    to      Owl::OCCURRENCE
    mapping do |adl, owl|
      owl.interval = adl
    end
```

```
  end

  rule 'Occurrence_spec2INTEGER_INTERVAL' do
    from    Adl::Occurrence_spec
    to      Owl::INTEGER_INTERVAL
    mapping do |adl, owl|
      owl.lower_bound = adl.lower
      owl.upper_bound= adl.upper
    end
  end


  rule 'C_attributes2CARDINALITY' do
    from    Adl::C_attribute
    to      Owl::CARDINALITY
    mapping do |adl, owl|
      owl.interval = adl.cattrhead.ccardinality.cardinalityspec.occurrencespec
    end
  end


  rule 'C_attribute2Cen_ELEMENT' do
    from    Adl::C_complex_object
    to      Owl::Cen_ELEMENT
    mapping do |adl, owl|
      owl.has_occurrence_constraint = adl.ccomplexobjecthead.coccurrences.
                                          occurrencespec
      name = adl.extract_name
      owl.term_definitions=Adl::Archetype.all_objects.first.ontology .attrval.
             select { |attr| attr.attrid.name == "term_definitions" }.first.
             untyped_object_single.attrvals.attrval.
             select { |attr| attr.attrid.name == 'items' }.
             select { |attr| attr.attrid.value == name }.
             map { |attr| attr.untyped_object_single }
    end
  end
end

phase 'pq' do
  top_rule 'pq' do
    from Adl::C_complex_object
    to   Owl::Cen_PQ
    filter { |adl| adl.is_pq? }
    mapping do |adl, owl|

      if not(adl.ccomplexobjectbody.cattributes.cattribute.
             select { |attr| attr.cattrhead.name == "value"}.empty?)

       owl.has_range_constraint = adl.ccomplexobjectbody.cattributes.
             cattribute.select { |attr| attr.cattrhead.name == "value"}.
             first.cattrvalue.cobject.first.
             cprimitiveobject.cprimitiveobject.creal.crealspec.
             realintervalvalue
      end
      owl.cen_units = adl.ccomplexobjectbody.cattributes.cattribute.
             select { |attr| attr.cattrhead.name == "units"}.
             first.cattrvalue.cobject.first.ccomplexobject
    end
  end



  rule 'Real_interval_value2RANGE' do
    from    Adl::Real_interval_value
    to      Owl::RANGE
    mapping do |adl, owl|
    owl.interval=adl
    end
  end
```

```
  rule 'Real_interval_value2INTERVAL' do
    from    Adl::Real_interval_value
    to      Owl::REAL_INTERVAL
    mapping do |adl, owl|
      owl.lower_bound=adl.lower
      owl.upper_bound=adl.upper
    end
  end



  rule 'cs' do
    from Adl::C_complex_object
    to   Owl::Cen_CS_UNITS
    filter { |adl| adl.is_cs? }
    mapping do |adl, owl|
      owl.cen_codeValue=adl.ccomplexobjectbody.cattributes.cattribute.
                        select {|attr| attr.cattrhead.name == "codeValue"}.
                        first.cattrvalue.cobject.first.
                       cprimitiveobject.cprimitiveobject.
                       cstring.cstringconstraint.stringvalue
    end
  end
end

phase 'bl' do
top_rule 'bl' do
    from Adl::C_complex_object
    to   Owl::Cen_BL
    filter { |adl| adl.is_bl? }
    mapping do |adl, owl|

      if not(adl.ccomplexobjectbody.cattributes.cattribute.
             select { |attr| attr.cattrhead.name == "value"}.empty?)

       owl.has_constraint_datatype = adl.ccomplexobjectbody.cattributes.
             cattribute.select { |attr| attr.cattrhead.name == "value"}.
             first.cattrvalue.cobject.first.
             cprimitiveobject.cprimitiveobject.
             cboolean.cbooleanconstraint
      end
    end
  end


  rule 'C_boolean_constraint2C_STRING' do
    from    Adl::C_boolean_constraint
    to      Owl::C_STRING
    mapping do |adl, owl|
      owl.list=adl.valuelow + " " + adl.valuehigh
    end
  end

end



phase 'merge' do



 refinement_rule 'C_attribute2Cen_ELEMENT' do
   from    Adl::C_complex_object
   to      Owl::Cen_ELEMENT
   mapping do |adl, owl|

     pqs = adl.ccomplexobjectbody.cattributes.cattribute.
```

```
              map    { |attr| attr.cattrvalue.cobject }.flatten.
              map    { |c| c.ccomplexobject }.
              select { |c| c.is_pq?}.
              map    { |pq| mapper(pq).take_one(Owl::Cen_PQ) }

    if pqs.empty?

    pqs = adl.ccomplexobjectbody.cattributes.cattribute.
              map    { |attr| attr.cattrvalue.cobject }.flatten.
              map    { |c| c.ccomplexobject }.
              select { |c| c.is_bl?}.
              map    { |bl| mapper(bl).take_one(Owl::Cen_BL) }


    end

    owl.cen_element_value = pqs
  end
 end



end
```

Apéndice D

D.1. Plantilla MOFScript de generación de código OWL

```
/**
 * transformation ecore2owl
 *
 *
 */
import "ecore2owlUtil.m2t"

texttransformation ecore2owl (in ec:"http:///org/eclipse/owl.ecore") {

  property a_string = "\"http://www.w3.org/2001/XMLSchema#string\""
  property a_integer = "\"http://www.w3.org/2001/XMLSchema#integer\""
  property a_float = "\"http://www.w3.org/2001/XMLSchema#float\""
  var nivel:Integer = 2;
  var id_elem:String
  var i=0

  ec.ARCHETYPE::main(){

    file (self.archetype_id +".owl")
    self.header()
    gen_tab(nivel)'<cen-archetype:'self.oclGetType()' rdf:ID="'self.archetype_id'">\n'
    gen_tab(nivel+1)'<cen-archetype:has_description rdf:resource="#'self.archetype_id+"-
Description"'"/>\n'
    gen_tab(nivel+1)'<cen-archetype:uid
rdf:datatype='a_string'>'self.archetype_id'</cen-archetype:uid>\n'
    gen_tab(nivel+1)'<cen-archetype:defines
rdf:resource="'"#"+self.defines.cen_act_id.substring(1,self.defines.cen_act_id.size()-
1)'"/>\n'
        gen_tab(nivel+1)'<cen-archetype:original_language
rdf:resource="'"#"+self.original_language.code_string'"/>\n'
    self.languages_available->forEach(c:ec.CODE_PHRASE){
        gen_tab(nivel+1)'<cen-archetype:languages_available
rdf:resource="#'c.code_string'"/>\n'
    }
        gen_tab(nivel)'</cen-archetype:'self.oclGetType()'>\n'

        self.has_description.archetype_description(self.archetype_id)
        ec.objectsOfType(ec.CODE_PHRASE)->forEach(c){c.code_phrase(nivel)}
        ec.objectsOfType(ec.OCCURRENCE)->forEach(c){c.constraint(nivel)}
        ec.objectsOfType(ec.CARDINALITY)->forEach(c){c.constraint(nivel)}
        ec.objectsOfType(ec.INTEGER_INTERVAL)->forEach(c){c.interval(nivel)}
        ec.objectsOfType(ec.REAL_INTERVAL)->forEach(c){c.interval(nivel)}
        self.defines.cen_element()
        gen_tab(nivel)'</rdf:RDF>'
  }


  /*-------------------------ARCHETYPE-DESCRIPTION-----------------------------------
----*/
  ec.ARCHETYPE_DESCRIPTION::archetype_description(nIndividual:String){
    gen_tab(nivel)'<cen-archetype:'self.oclGetType()' rdf:ID="'nIndividual+"-
Description"'">\n'
    gen_tab(nivel+1)'<cen-archetype:details>\n'
    self.details.archetype_description_item(nivel+2,nIndividual)
        gen_tab(nivel+1)'</cen-archetype:details>\n'
        gen_tab(nivel+1)'<cen-archetype:original_author
rdf:datatype='a_string'>'self.original_author'</cen-archetype:original_author>\n'
        gen_tab(nivel+1)'<cen-archetype:lifecycle_state
rdf:datatype='a_string'>'self.lifecycle_state'</cen-archetype:lifecycle_state>\n'
        if(!self.other_details.isEmpty()){
                gen_tab(nivel+1)'<cen-archetype:other_details>\n'
                self.other_details->forEach(c){c.other_details(nivel+2)}
                gen_tab(nivel+1)'</cen-archetype:other_details>\n'
        }
        gen_tab(nivel)'</cen-archetype:'self.oclGetType()'>\n'
  }
```

```
  /*--------------------------ARCHETYPE-DESCRIPTION-ITEM-------------------------------
----*/
  ec.ARCHETYPE_DESCRIPTION_ITEM::archetype_description_item(nivel:Integer,id:String){
    gen_tab(nivel)'<cen-archetype:'self.oclGetType()' rdf:ID="'id+"-Details"'">\n'
       gen_tab(nivel+1)'<cen-archetype:language
rdf:resource="'"#"+self.language.code_string'"/>\n'
       gen_tab(nivel+1)'<cen-archetype:purpose
rdf:datatype='a_string'>'self.purpose'</cen-archetype:purpose>\n'
       gen_tab(nivel)'</cen-archetype:'self.oclGetType()'>\n'
  }

  /*---------------------------OTHER_DETAIL-----------------------------------------
----*/
  ec.OTHER_DETAIL::other_details(nivel:Integer){
       gen_tab(nivel)'<cen-archetype:OTHER_DETAIL
rdf:ID="'self.name.firstToUpper()'">\n'
    gen_tab(nivel+1)'<cen-archetype:value_detail
rdf:datatype='a_string'>'self.value_detail'</cen-archetype:value_detail>\n'
    gen_tab(nivel+1)'<cen-archetype:name rdf:datatype='a_string'>'self.name'</cen-
archetype:name>\n'
    gen_tab(nivel)'</cen-archetype:OTHER_DETAIL>\n'
  }


  /*--------------------------CLINICAL_TERM-----------------------------------------
----*/

  abstract ec.CLINICAL_TERM::cen_element()

  ec.cen_ENTRY::cen_element(){
       var id_entry:String = self.cen_act_id.substring(1,self.cen_act_id.size()-1 )

    gen_tab(nivel+1)'<cen:ENTRY
rdf:ID="'self.cen_act_id.substring(1,self.cen_act_id.size()-1)'">\n'
    if (self.cen_items.size()>=1){
       self.cen_items->forEach(c:ec.cen_ELEMENT){
              gen_tab(nivel+2)'<cen:items>\n'
              gen_index()
              c.cen_element(nivel+3,id_elem)
              gen_tab(nivel+2)'</cen:items>\n'
       }
    }
    self.term_definitions->forEach(c:ec.TERM_DEFINITION){
       gen_tab(nivel+2)'<cen-archetype:term_definitions>\n'
              c.term_definition(nivel+3,id_entry)
              gen_tab(nivel+2)'</cen-archetype:term_definitions>\n'
    }
    if(self.has_occurrence_constraint!=null)gen_tab(nivel+2)'<cen-
archetype:has_occurrence_constraint
rdf:resource="#OCCURRENCE_'+self.has_occurrence_constraint.interval.lower_bound+"_"+self
.has_occurrence_constraint.interval.upper_bound '"/>\n'
    if(self.has_cardinality_constraint!=null)gen_tab(nivel+2)'<cen-
archetype:has_cardinality_constraint
rdf:resource="#CARDINALITY_'+self.has_cardinality_constraint.interval.lower_bound+"_"+se
lf.has_cardinality_constraint.interval.upper_bound '"/>\n'
    gen_tab(nivel+2)'<cen:act_id rdf:datatype='a_string'>'id_entry'</cen:act_id>\n'
    gen_tab(nivel+2)'</cen:ENTRY>\n'
  }

  ec.cen_ELEMENT::cen_element(nivel:Integer,id:String){
       gen_tab(nivel)'<cen:ELEMENT rdf:ID="'id'">\n'
       gen_tab(nivel+1)'<cen:element_value>\n'
       self.cen_element_value.value(nivel+2,id)
       gen_tab(nivel+1)'</cen:element_value>\n'
       self.term_definitions->forEach(c:ec.TERM_DEFINITION){
              gen_tab(nivel+1)'<cen-archetype:term_definitions>\n'
              c.term_definition(nivel+2,id)
              gen_tab(nivel+1)'</cen-archetype:term_definitions>\n'
       }
    gen_tab(nivel+1)'<cen:item_entry rdf:resource="#at0000"/>\n'
       gen_tab(nivel+1)'<cen-archetype:has_occurrence_constraint>\n'
       self.has_occurrence_constraint.constraint(nivel+2)
       gen_tab(nivel+1)'</cen-archetype:has_occurrence_constraint>\n'
       gen_tab(nivel)'</cen:ELEMENT>\n'
  }
```

```
  /*-----------------------------------PQ-----------------------------------------
---*/
  ec.cen_PQ::value(nivel:Integer,id:String){
    gen_tab(nivel)'<cen:PQ rdf:ID="PQ_'id'">\n'
    gen_tab(nivel+1)'<cen:units>\n'
    self.cen_units.cen_cs_units(nivel+2)
    gen_tab(nivel+1)'</cen:units>\n'
    gen_tab(nivel+1)'<cen-archetype:has_range_constraint>\n'
    if (self.has_range_constraint!=null){
    self.has_range_constraint.constraint(nivel+2)}
    gen_tab(nivel+1)'</cen-archetype:has_range_constraint>\n'
    gen_tab(nivel)'</cen:PQ>\n'
   }

   ec.cen_BL::value(nivel:Integer,id:String){
        gen_tab(nivel)'<cen:BL rdf:ID="'id'_BL"/>\n'
   }

 /*---------------------------------TERM-DATATYPE--------------------------------
----*/
  /* cen-archetype:TERM_DATATYPE/cen-archetype:CLINICAL_DATATYPE/cen:CLINICAL_DATATYPE
        cen:DATAVALUE/cen:CD/cen:CV/cen:CS/cen:CS_UNITS
        cen_cs_units */


        ec.cen_CS_UNITS::cen_cs_units(nivel:Integer){
              gen_tab(nivel)'<cen:CS_UNITS rdf:ID="PQ_CS_UNITS">\n'
        gen_tab(nivel+1)'<cen:displayName
rdf:datatype='a_string'>'self.cen_displayName'</cen:displayName>\n'
        gen_tab(nivel+1)'<cen:codeValue
rdf:datatype='a_string'>'self.cen_codeValue'</cen:codeValue>\n'
        gen_tab(nivel)'</cen:CS_UNITS>\n'
     }

  /*---------------------------------CONSTRAINT-----------------------------------
----*/
  abstract ec.CONSTRAINT::constraint(nivel:Integer)
        ec.CARDINALITY::constraint(nivel:Integer){
              gen_tab(nivel)'<cen-archetype:CARDINALITY
rdf:ID="'"CARDINALITY_"+self.interval.lower_bound+"_"+self.interval.upper_bound'">\n'
        gen_tab(nivel+1)'<cen-archetype:interval
rdf:ID="#'self.interval.oclGetType()+"_"+self.interval.lower_bound+"_"+self.interval.upp
er_bound'"/>\n'
        gen_tab(nivel)'</cen-archetype:CARDINALITY>\n'
        }

        ec.OCCURRENCE::constraint(nivel:Integer){
              gen_tab(nivel)'<cen-archetype:OCCURRENCE
rdf:ID="'"OCCURRENCE_"+self.interval.lower_bound+"_"+self.interval.upper_bound'">\n'
              gen_tab(nivel+1)'<cen-archetype:interval
rdf:ID="#'self.interval.oclGetType()+"_"+self.interval.lower_bound+"_"+self.interval.upp
er_bound'"/>\n'
              gen_tab(nivel)'</cen-archetype:OCCURRENCE>\n'
        }

        ec.RANGE::constraint(nivel:Integer){
              gen_tab(nivel)'<cen-archetype:RANGE
rdf:ID="'"RANGE_"+self.interval.lower_bound+"_"+self.interval.upper_bound'">\n'
        gen_tab(nivel+1)'<cen-archetype:interval
rdf:ID="#'self.interval.oclGetType()+"_"+self.interval.lower_bound+"_"+self.interval.upp
er_bound'"/>\n'
        gen_tab(nivel)'</cen-archetype:RANGE>\n'
        }
```

```
/*-----------------------------------INTERVAL-----------------------------------
--*/
  abstract ec.INTERVAL::interval(nivel:Integer)
        ec.INTEGER_INTERVAL::interval(nivel:Integer){
                gen_tab(nivel)'<cen-archetype:INTEGER_INTERVAL
rdf:ID="'"INTEGER_INTERVAL_"+self.lower_bound"_"+self.upper_bound'">\n'
        gen_tab(nivel+1)'<cen-archetype:upper_bound
rdf:datatype='a_integer'>'self.upper_bound'</cen-archetype:upper_bound>\n'
        gen_tab(nivel+1)'<cen-archetype:lower_bound
rdf:datatype='a_integer'>'self.lower_bound'</cen-archetype:lower_bound>\n'
                gen_tab(nivel)'</cen-archetype:INTEGER_INTERVAL>\n'
        }
        ec.REAL_INTERVAL::interval(nivel:Integer){
                gen_tab(nivel)'<cen-archetype:REAL_INTERVAL
rdf:ID="'"REAL_INTERVAL_"+self.lower_bound"_"+self.upper_bound'">\n'
         gen_tab(nivel+1)'<cen-archetype:upper_bound
rdf:datatype='a_float'>'self.upper_bound'</cen-archetype:upper_bound>\n'
          gen_tab(nivel+1)'<cen-archetype:lower_bound
rdf:datatype='a_float'>'self.lower_bound'</cen-archetype:lower_bound>\n'
          gen_tab(nivel)'</cen-archetype:REAL_INTERVAL>\n'
        }


  /*-------------------------------TERM-DEFINITION-------------------------------
----*/
  ec.TERM_DEFINITION::term_definition(nivel:Integer,id:String){
        gen_tab(nivel)'<cen-archetype:TERM_DEFINITION rdf:ID="'id+"_definition"'">\n'
     gen_tab(nivel+1)'<cen-archetype:text rdf:datatype='a_string'>'self.text'</cen-
archetype:text>\n'
     gen_tab(nivel+1)'<cen-archetype:description
rdf:datatype='a_string'>'self.description'</cen-archetype:description>\n'
        gen_tab(nivel)'</cen-archetype:TERM_DEFINITION>\n'
  }


  /*-------------------------------CODE-PHRASE-----------------------------------
----*/
    ec.CODE_PHRASE::code_phrase(nivel:Integer){
        gen_tab(nivel)'<cen-archetype:CODE_PHRASE rdf:ID="'self.code_string'">\n'
     gen_tab(nivel+1)'<cen-archetype:code_string
rdf:datatype='a_string'>'self.code_string'</cen-archetype:code_string>\n'
        gen_tab(nivel)'</cen-archetype:CODE_PHRASE>\n'
  }


  /*-------------------------------CABECERA-------------------------------------
----*/
  ec.ARCHETYPE::header(){
        <%<?xml version = "1.0"?>
        <rdf:RDF
                xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
        xmlns="http://klt.inf.um.es/~poseacle/archetypes/%>self.archetype_id<%.owl#"
        xmlns:cen="http://klt.inf.um.es/~poseacle/CEN-SP-v1.0.owl#"
        xmlns:owl="http://www.w3.org/2002/07/owl#"
        xmlns:cen-archetype="http://klt.inf.um.es/~poseacle/CEN-AR-v1.0.owl#"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"

        xml:base="http://klt.inf.um.es/~poseacle/archetypes/%>self.archetype_id<%.owl">
                <owl:Ontology rdf:about="">
                <owl:imports rdf:resource="http://klt.inf.um.es/~poseacle/CEN-AR-
v1.0.owl"/>
                </owl:Ontology>\n%>
  }


}
```

Apéndice E

E.1. Contenido del CD adjunto

En el cd adjunto se reunen las herramientas utilizadas para el desarrollo del proyecto, a continuación se describe cuál es la utilidad de cada una nombrándolas de la misma forma en la que se encuentran disponibles en el cd.

1. *EclipseEMF-OWL*: Esta versión de Eclipse, la 3.0, ha sido la utilizada para generar el metamodelo OWL a partir de interfaces Java anotadas.

2. *Eclipse_xText*: Se corresponde con la versión 3.2 de eclipse y dispone de los plugin necesarios para trabajar con xText. Su utilización nos ha permitido definir la sintaxis de la gramática ADL y generar un metamodelo, y diversos modelos de ejemplo.

3. *AGE 0.3.1*: En este entorno hemos trabajado para obtener las reglas de transformación a partir de los metamodelos de ADL y OWL. Concretamente hemos utilizado RubyTL, que se encuentra integrado en este framework.

4. *Protégé 3.3*: Herramienta para trabajar con ontologías. Con ella hemos podido visualizar y modificar la ontología utilizada en el proyecto, así como generar las interfaces Java que nos han permitido obtener el metamodelo OWL.

5. *Eclipse MOFScript*: Se corresponde con la versión 3.2.2 de Eclipse con los plugin necesarios para el uso de MOFScript. Con esta herramienta hemos creado las plantillas para la generación de código OWL.

Junto a la carpeta herramientas se encuentran cada uno de los proyectos Eclipse empleados en la realización del proyecto. En la parte de diseño de la solución de este documento se puede leer con más detalle de que constan y conocer detalles sobre la implementación y finalidad de los mismos. A continuación se enumeran y se indica, entre paréntesis, a qué parte del diseño de la solución, según este documento, se corresponde cada uno de ellos.

1. OwlEMF (7.2 Obtención del metamodelo OWL)

2. xText (7.3 Obtención del metamodelo ADL)
3. RubyTL (7.5 Obtención de las reglas de transformación)
4. MofScript (7.6 Generación de código OWL)

Bibliografía

[1] Beale, T. (2001). "Archetypes, Constraint-based Domain Models for Future-proof Information Systems". Available: http://www.deepthought.com.au/it/archetypes/archetypes.pdf

[2] Monserrat Robles, "Aspectos relacionados con la Historia Clínica Electrónica", http://www.medicamentos-innovadores.org/documentos/Workshop%20GC/Montserrat_Asp_relac_Hist_Clica_Elec2.pdf

[3] José Luis Monteagudo Peña, "Estándares para la Historia Clínica Electrónica". Available: http://www.conganat.org/Seis/informes/2003/PDF/CAPITULO7.pdf

[4] Carlos Hernández Salvador, Universidad Politécnica de Madrid Tesis "Modelo de Historia Clínica para Teleconsulta Médica". Available: http://oa.upm.es/231/01/09200417.pdf

[5] Jose Alberto Maldonado, Montserrat Robles, Pere Crespo, "Utilización de arquetipos para la integración de sistemas de información hospitalarios". Available: http://gim.upv.es/sih/articulos/caseib2002.pdf

[6] Sven Efftinge , "oAW xText: A framework for textual DSLs". Available: http://www.eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium12_xTextFramework.pdf

[7] xText Reference Documentation.
Disponible: http://gts.inf.um.es/trac/age/chrome/site/tutorial.pdf

[8] OMG (2006) Meta Object Facility (MOF) 2.0 Core Specification, OMG Document formal/2006-01-01, http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

[9] Ontology Metamodel Definition Specification (2006). OMG Documentformal/2006-05-01. http://www.omg.org/cgi-bin/doc?ad/2006-05-01.pdf

[10] Tony Clark, Andy Evans, Paul Sammut, James Willans, ”Applied Metamodelling”. Disponible:http://www.uio.no/studier/emner/matnat/ifi/INF5120/v06/undervisningsmateriale/AppliedMetamodelling.pdf

[11] “Tutorial Gemerating an EMF Model”. Available: http://dev.eclipse.org/viewcvs/indextools.cgi/org.eclipse.emf/doc/org.eclipse.emf.doc/tutorials/clibmod/clibmod.html

[12] Stephan Roser, “Ontology-based Model Transformations” Available: http://www.cs.colostate.edu/models05/DoctoralSymposium/doctoralSymposiumPapers/stephan.pdf

[13] Jean Bézivin, University of Nantes “From Object Composition to Model Transformation with the MDA”. Available: http://www.sciences.univ-nantes.fr/info/lrsg/Recherche/mda/TOOLS.USA.pdf

[14] Ivan Kurtev, Jean Bézivin, Mehmet Aksit ,“Technological Spaces: an Initial Appraisal”.Available:http://www.sciences.univnantes.fr/lina/atl/www/papers/PositionPaperKurtev.pdf

[15] Addison Wesley Eclipse Modelling Framework, chapter 2 “Introduciong EMF”

[16] Jesualdo Tomás Fernández-Breis, Rafael Valencia-García, Marcos Menarguez Tortosa, Pedro José Vivancos-Vicente. “Approaching Electronic Healthcare Records Management from a Semantic Web perspective”.

[17] Jesualdo Tomás Fernández-Breis, Marcos Menarguez Tortosa, Catalina Martínez Costa, rafael Valencia García, David Moner, Jesús Sánchez Cuadrado, José Alberto Maldonado.”A Model-driven Approach for Representing Clinical Archetypes for Semantic Web Environments”.

[18] Sánchez-Cuadrado, J., García-Molina, J, Menárguez-Tortosa, M. (2006) RubyTL: A Practical, Extensible Transformation Language. ECMDA-FA 2006. http://rubytl.rubyforge.org/

[19] Jesús Sánchez Cuadrado, “AGE Tutorial”, Marzo 2007. Available: http://gts.inf.um.es/trac/age/chrome/site/tutorial.pdf

[20] John H. Gennari, University of Standford “The Evolution of Protégé: An Envitonment for Knowledge-Based Systems “Development. Available: http://smi.stanford.edu/smi-web/reports/SMI-2002-0943.pdf

[21] Holger knublauch Ray, W. Fergerson, Natalya F. Noy and Mark A. Musen “The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications”.Disponible:http://protege.stanford.edu/plugins/owl/publications/ISWC2004-protege-owl.pdf

[22] Deepak K. Sharma, Thomas M. Johnson, Harold R. Solbrig, Dr. Christopher G. Chute MD, DrPH,“Transformation of Protégé Ontologies into the Eclipse Modeling

Framework".Disponible:http://protege.stanford.edu/conference/2005/submissions/abstracts/accepted-abstract-sharma.pdf

[23] Ozgur Kilic, Veli Bicer, Asuman Dogac,"Mapping Archetypes to OWL". Available: http://www.srdc.metu.edu.tr/webpage/publications/2005/MappingArchetypestoOWLTechnical.pdf

[24] Protégé: http://protege.stanford.edu/

[25] CENTC251: http://www.centc251.org

[26] EMF: http://www.eclipse.org/emf/

[27] EODM: http://www.eclipse.org/emft/projects/eodm/

[28] OMG: http://www.omg.org

[29] OpenEHR: http://www.openehr.org

[30] "Mofscript User Guide" Disponible: http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf