Capítulo 1

Introducción

Este capítulo sirve de introducción a la presente Memoria de Proyecto Fin de Carrera. En él se describen los objetivos generales del proyecto, así como el contexto donde se enmarca. Por último, se describe como se estructura el presente documento.

Índice

1.1. Introducción **1**

1.2. Antecedentes: Ingeniería de Lenguajes Dirigida por Modelos **4**

1.3. Planificación del proyecto **8**

1.4. Estructura del Documento **10**

1.1. Introducción

El principal objetivo de este Proyecto de Fin de Carrera es extender la herramienta *Hydra* [] para que soporte la especificación y validación de restricciones que contengan características con cardinalidad. Dicho objetivo resultará, como es lógico, confuso para el lector no familiarizado con las líneas de productos software [] en general, y con los árboles de características con cardinalidad [], en particular. Por tanto, intentaremos introducir de forma breve al lector en estos conceptos.

El objetivo de una *Línea de Productos Software* [] es crear la infraestructura adecuada para una rápida y fácil producción de sistemas software similares, destinados a un mismo segmento de mercado. Las líneas de productos software se pueden ver como análogas a las líneas de producción industriales, donde productos similares o idénticos se ensamblan y configuran a partir de piezas prefabricadas bien definidas. Un ejemplo clásico de línea de producción industrial es la fabricación de automóviles, donde se pueden crear decenas de variaciones de un único modelo de coche con un so-

lo grupo de piezas cuidadosamente diseñadas mediante una línea de montaje específicamente diseñada para configurar y ensamblar dichas piezas.

Ya dentro del mundo del software, el desarrollo de software, por ejemplo, para teléfonos móviles implica la creación de productos con características muy parecidas, pero diferenciados entre ellos. Por ejemplo, una aplicación de agenda personal podrá ofrecer diferentes funcionalidades en función de si el terminal móvil posee GPS (*Global Positioning System*), acceso a mapas o *bluetooh*. Por tanto, el objetivo de una línea de productos software es crear una especie de línea de montaje donde una aplicación de agenda personal como la mencionada se pueda construir de la forma más eficiente posible de acuerdo a las características concretas de cada terminal específico.

Para construir una línea productos software, el primer paso es analizar qué características comunes y variables poseen cada uno de los productos que tratamos de producir. Para realizar dicho análisis de la variabilidad de una familia de productos se utilizan diversas técnicas. Las más utilizadas actualmente son la creación de árboles de características [] y los lenguajes específicos de dominio []. En este proyecto nos centraremos en la primera opción.

Un árbol de características [] es un tipo de modelo que especifica, tal como su nombre indica en forma de árbol, las características que puede poseer un producto concrete perteneciente a una familia de productos, indicando qué características son comunes a todos los productos, cuáles son variables, así como las razones por las cuales son variables.

Por ejemplo, en una línea de productos de agendas personales para teléfonos móviles, toda agenda personal debe permitir anotar eventos a los que debemos asistir en un futuro cercano. Por tanto, esta característica sería una característica obligatoria para todas las agendas personales. Sin embargo, ciertas agendas, dependiendo del precio que el usuario final esté dispuesto a pagar y las características técnicas de cada terminal, podrían ofrecer la función de geolocalizar el lugar del evento al que debemos asistir, y calcular la ruta óptima desde el lugar que le indiquemos a dicho lugar de destino. Esta última característica sería opcional, y podría no estar incluida en ciertas agendas personales instalados en terminales concretos.

Para obtener un producto específico dentro de una línea de productos software, el cliente debe especificar qué características concretas desea que posea el producto que va a adquirir. Es decir, en términos técnicos, debe crear una *configuración* del árbol características. Obviamente, no toda selección de características da lugar a una configuración válida. Por ejemplo, toda configuración debe contener al menos el conjunto de características que son obligatorias para todos los productos. De igual forma, puede ser obligatorio escoger al menos una características de entre una serie de alternativas. Por ejemplo, la agenda personal podría estar disponible en castellano, inglés y francés. En este caso sería posible seleccionar cualquiera de las tres alternativas, pero al menos una debería incluirse en nuestro producto. También

sería posible indicar que podemos seleccionar un único idioma, es decir, que no podemos instalar una agenda personal que soporte de forma simultánea dos idiomas distintos.

La mayoría de estas restricciones se pueden especificar usando la sintaxis propia de los árboles de características. No obstante, existen una serie de restricciones que no se pueden modelar con la sintaxis propia de los árboles de características. Un ejemplo de tal tipo de restricción son las relaciones de dependencias entre características. Por ejemplo, la selección de una característica de cálculo de rutas óptimas podría necesitar para funcionar que estuviesen instalados los servicios de mapas y geolocalización. Dichas tres características podrían no aparecer relacionadas en el árbol de características, por lo que tendríamos que definir dicha restricción como una restricción externa.

Estas restricciones externas se suelen especificar utilizando fórmulas de lógica proposicional []. Los átomos de dichas fórmulas son las características del sistema. Dichos átomos se evalúan a verdadero si las características correspondientes están seleccionadas, y a falso en caso contrario. Por ejemplo, la restricción anteriormente expuesta podría especificarse como $CalculoRutasOptimas \Rightarrow (Mapas \wedge Geolocalizacion)$.

Para que estas restricciones sean utilidad, además de especificarlas, debemos comprobar que satisfacen para las diferentes configuraciones creadas. En los últimos años se han ido creando diversas técnicas y herramientas para el análisis y validación de dichas restricciones [].

Paralelamente al problema de la especificación y validación de las restricciones externas, se han ido incorporando diversas modificaciones y novedades a los modelos de árboles de características en los últimos años. Por ejemplo, se han introducido conceptos como las *referencias entre características* [] y *atributos* [] para las características. Uno de estos conceptos, simple pero importante, ha sido el de característica clonable []. Una característica clonable es una características que pueden aparecer un número variable de veces dentro de un producto.

Por ejemplo, supongamos que tenemos una red de sensores para la monitorización y regulación del nivel de humedad de un determinado recinto, por ejemplo, de un invernadero. Dependiendo de donde fuésemos a instalar dicha red, podríamos necesitar un número diferente de sensores. Además, dependiendo de donde instalásemos cada sensor, podríamos configurar cada sensor de forma diferente. Por ejemplo, ciertos sensores podrían necesitar tener capacidades de enrutamiento, se tolerantes a fallo o poseer modos de hibernación para disminuir el consumo de energía. Por tanto, en dicho sistema sería interesante modelar *Sensor* como una característica que se puede clonar, es decir, crear un número variable de instancias de la misma, y donde cada clon fuese a su vez configurable con ciertas características.

La incorporación de las características clonables a los árboles de características hace que los mecanismos utilizados hasta ahora para especificar

y evaluar restricciones externas hayan quedado obsoletos. Dado que las características clonables no se seleccionan sino que se clonan, ya no podemos evaluar una característica clonable a verdadero o falso dependiendo de si está o no seleccionada. El concepto de *estar seleccionada* desaparece en el caso de las características clonables.

Para solventar dicho problema, el profesor Pablo Sánchez, dentro del Departamento de Matemáticas, Estadística y Computación, ha desarrollado un nuevo lenguaje para la especificación y validación de restricciones externas a los árboles de características donde dichas restricciones pueden contener *características clonables*. Dicho lenguaje se denomina *HCL (Hydra Constraint Language*).

El objetivo de este Proyecto Fin de Carrera es implementar un editor que permita especificar y validar restricciones especificadas en HCL, es decir restricciones sobre árboles de características que puedan incluir características clonables. Dicho editor se debe integrar en una herramienta para el modelado y configuración de árboles de características denominada *Hydra*, desarrollada también por el profesor Pablo Sánchez, en colaboración con un antiguo alumno suyo de la Universidad de Málaga, José Ramón Salazar. Con esto esperamos haber aclarado el primer párrafo de esta sección al lector no familiarizado con las líneas de productos software y/o los árboles de características.

Hydra se distribuye actualmente como un plugin para Eclipse, y ha sido desarrollada utilizando modernas técnicas de *Ingeniería de Lenguajes Dirigida por Modelos* []. Dichas técnicas permiten una rápida y cómoda creación de entornos de edición y evaluación de lenguajes tanto visuales como textuales mediante la especificación de una serie de elementos básicos a partir de los cuales se genera una gran cantidad de artefactos, reduciendo los tiempos de desarrollo y costo asociado al desarrollo de dichos entornos. El editor desarrollado en este Proyecto Fin de Carrera deberá distribuirse también como un plugin para Eclipse, instalable sobre *Hydra*. Para su desarrollo se usará también un enfoque de *Ingeniería de Lenguajes Dirigida por Modelos* [].

Tras esta introducción, el resto del presente capítulo se estructura como sigue: La Sección 1.2 proporciona unas nociones básicas sobre la *Ingeniería de Lenguajes Dirigida por Modelos*, nociones que son necesarias para poder entender la planificación del presente proyecto, la cual se describe en la Sección 1.3. Por último, la Sección 1.4 describe la estructura general del presente documento.

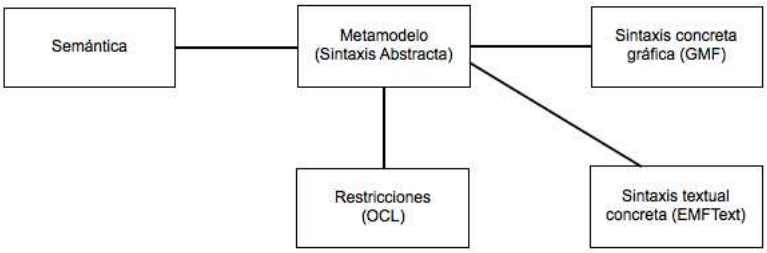

Figura 1.1: Componentes de la ingeniería de lenguajes dirigida por modelos

1.2. Antecedentes: Ingeniería de Lenguajes Dirigida por Modelos

Este proyecto ha sido desarrollado siguiendo un enfoque de *Ingeniería de Lenguajes Dirigida por Modelos*, la cual establece unas etapas claras para el proceso de desarrollo de un nuevo lenguaje software. Por tanto, antes de proceder a explicar la planificación del presente proyecto se hace necesario adquirir unas nociones básicas sobre la Ingeniería de Lenguajes Dirigida por Modelos, de forma que se pueda entender por qué el presente proyecto se organiza tal como se organiza.

La *Ingeniería de Lenguajes Dirigida por Modelos* [] no es más que un caso concreto de la más genérica *Ingeniería Software Dirigida por Modelos* [] aplicado desde el punto de vista de la *Teoría de Lenguajes Formales*. El proceso de de ingeniería de un nuevo lenguaje de modelado está compuesto de diversas fases, las cuales se explican con ayuda de la figura 1.1.

1. El primer paso es crear las reglas de construcción o sintaxis de nuestro lenguaje de modelado. Esto se realiza mediante la creación de un metamodelo, o modelo de nuestro lenguaje que describe usando conceptos parecidos a los de los diagramas de clases, la *sintaxis abstracta* o reglas para la construcción de modelos de nuestro lenguaje. Dicho metamodelo se construye usando un lenguaje de metamodelado. Existen diversos lenguajes de metamodelado, tales como KM3 [] o MOF [], aunque Ecore [] está considerado como el estándar *de facto* dentro de la comunidad de modelado. Normalmente no es posible especificar cualquier tipo de restricción entre los elementos de un lenguaje de modelado usando exclusivamente los elementos del lenguaje de metamodelado. Por ello, tales lenguajes de metamodelado se acompañan con un lenguaje de especificación de restricciones. OCL [] es el lenguaje de restricciones más usado para desarrollar esta tarea.

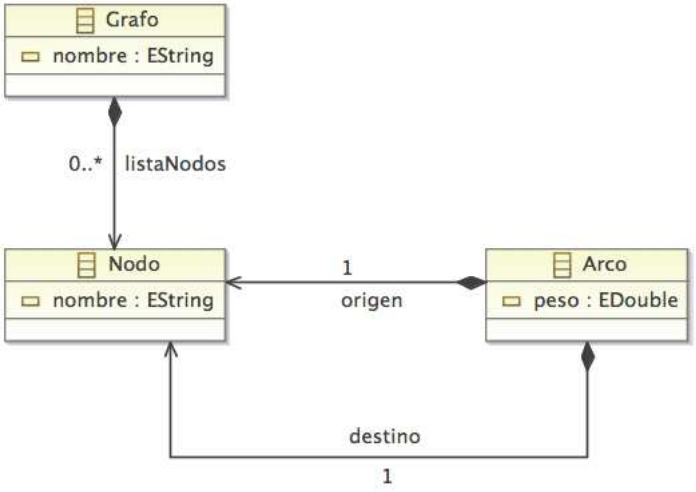

Figura 1.2: Metamodelo (sintaxis abstracta) de un lenguaje para modelar grafos pesados dirigidos

2. Como se ha comentado en el punto anterior, un metamodelo establece la sintaxis abstracta de nuestro lenguaje de modelado, pero no especifica la *sintaxis concreta* o notación de nuestro lenguaje de modelado. Por tanto el siguiente paso en la ingeniería de un nuevo lenguaje de modelado, es la definición de una notación o sintaxis concreta para nuestro lenguaje. Esta notación puede ser tanto textual como gráfica. Para la definición de notaciones gráficas, GMF es la herramienta más popular cuando se trabaja con Ecore. Para notaciones textuales, TCS [], xText, TEF o EMFText pueden ser usadas, no existiendo aún un estándar *de facto* para la definición de notaciones textuales.

3. Por último, nos quedaría definir la semántica del lenguaje de modelado. Existen un amplio rango de técnicas para desarrollar esta tarea, tales como: (1) definir la semántica de cada elemento de manera informal; (2) especificar la semántica de cada elemento usando un lenguaje formal, tales como máquinas de estado abstractas, o (3) crear generadores que transformen los elementos de modelado en elementos de otro lenguaje distinto bien definido.

Ilustramos los conceptos anteriormente expuestos mediante la creación de un lenguaje simple para el modelado de grafos dirigidos y pesados. La Figura 1.2 muestra el metamodelo o sintaxis abstracta para dicho lenguaje. La clase *Grafo* actúa como contenedor para el resto de los elementos. Representa un como conjunto de *Nodos* y *Arcos*. Los nodos poseen nombre, que permite distinguirlos uno de otros. Dichos nodos pueden estar conectados por arcos dirigidos, por lo que cada arco tiene un nodo origen y un nodo

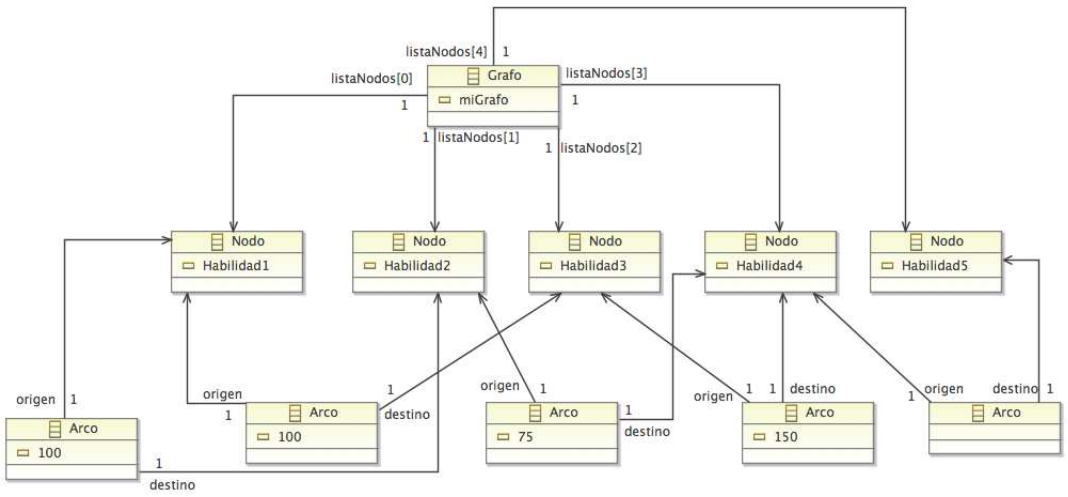

Figura 1.3: Modelo abstracto de un grafo pesado dirigido concreto

destino. Dado que el grafo es pesado, cada noto tiene además un peso.

Utilizando dicho metamodelo, que no es más que un diagrama de clases, podemos crear instancias del mismo. Una instancia de un metamodelo es un modelo. En nuestro caso, cada instancia concreta del metamodelo de la Figura 1.2 representaría un determinado grafo pesado dirigido, con un número determinado de nodos, pesos y arcos. Por ejemplo, la Figura 1.3 muestra una instancia del metamodelo de la Figura 1.2 que representa un grafo dirigido y pesado cuyos nodos representan las habilidades de una determinada disciplina, mientras que el peso de los arcos representa el tiempo necesario para adquirir esa habilidad. La habilidad 1 es el nodo inicial, es decir, aquella que todos poseen. A partir de ahí y siguiendo los arcos se pueden desarrollar unas habilidades u otras.

Como se ha ilustrado con los ejemplos anteriores, un metamodelo define un conjunto de reglas que debe obedecer todo modelo que pertenezca al lenguaje definido por dicho metamodelo. Por tanto, usando términos más técnicos, un metamodelo define la sintaxis abstracta de un lenguaje. Dicha sintaxis abstracta será similar a los árboles sintacticos de los lenguajes de programación tradicionales. No obstante, para que el lenguaje definido por un metamodelo sea fácilmente utilizable, debemos definir una sintaxis concreta, ya sea textual o gráfica, para dicho lenguaje. Dicha sintaxis concreta indicaría como podemos construir modelos que sean conformes al metamodelo, pero usando una notación que le sea familiar al usuario, en lugar de la notación genérica y abstracta mostrada en la Figura 1.3.

Por ejemplo, para nuestro ejemplo anterior, el del lenguaje para la especificación de grafos dirigidos pesados, podríamos elaborar una sintaxis visual donde los grafos se representasen utilizando la notación visual por todos co-

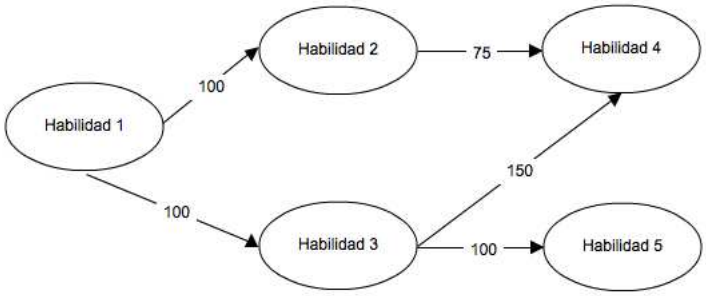

Figura 1.4: Modelo concreto de un grafo pesado dirigido concreto

nocida, tal como se ilustra en la Figura 1.4. Dicha figura muestra el mismo modelo que la Figura 1.3, pero utilizando una sintaxis visual concreta. En este caso, las instancias de la clase *Nodo* se representan como elipses. El nombre de cada nodo se muestra dentro de cada elipse. Las instancias de la clase *Arco* se dibujan como flechas. La punta de la flecha indica cuál es el nodo destino, mientras que la cola especifica cuál es el nodo origen. El peso de cada arco se muestra como texto de forma adjunta a cada flecha.

Los modernos entornos para el desarrollo de lenguajes dirigido por modelos proporcionan facilidades para, una vez definido un metamodelo, asociar a cada clase perteneciente a dicho metamodelo, un símbolo gráfico. Utilizando las facilidades proporcionadas por dichos entornos es posible generar, de manera automática, un editor visual que permita la creación de modelos conformes al metamodelo origen, utilizando para ello la sintaxis visual definida. Ejemplos de tales entornos son MetaEdit+ [], Microsoft DSL tools [] o la conocida combinación de herramientas Eclipse+Ecore+GMF (*Graphical Modelling Framework*) [].

De igual forma que hemos definido una sintaxis visual concreta hubiésemos podido definir una sintaxis textual concreta, donde los modelos conformes al metamodelo en cuestión pudiesen especificarse usando texto, en lugar símbolos gráficos. La Figura **??** muestra el ejemplo de la Figura 1.3 pero utilizando una sintaxis textual para nuestro lenguaje de grafos. En este caso, primero hay que definir un nombre para el grafo que estamos creando, y a continuación definir nodos y después arcos. De los nodos indicaremos su nombre, y de los arcos su peso y sus nodos de origen y destino, usando para ello las palabras reservadas pertinentes. De este modo podemos observar una de las claras ventajas de usar metamodelos para especificar lenguajes específicos de modelo, y es que permiten de forma relativamente sencilla poder expresar las sintaxis concretas de más de un modo, escogiendo para cada caso el que sea más conveniente.

Al igual que en el caso de las sintaxis visuales concretas, los entornos

```
Grafo : miGrafo;
Nodo : Habilidad1;
Nodo : Habilidad2;
Nodo : Habilidad3;
Nodo : Habilidad4;
Nodo : Habilidad5;
Arco : 100 from Habilidad1 to Habilidad2;
Arco : 100 from Habilidad1 to Habilidad3;
Arco : 75 from Habilidad2 to Habilidad4;
Arco : 150 from Habilidad3 to Habilidad4;
Arco : 100 from Habilidad3 to Habilidad5;
```

Figura 1.5: Sintaxis concreta textual de un grafo dirigido pesado

de desarrollo de lenguajes dirigidos por modelos proporcionan facilidades para ligar los elementos de una gramática tipo BNF (*Backus-Naus Form*) [] con las clases de un metamodelo. Una vez establecida dicha relación, dichos entornos son capaces de generar un editor textual más un analizador sintáctico que permita la especificación de modelos conformes a nuestro modelo y su posterior análisis para su conversión en un modelo abstracto, como el mostrado en la Figura 1.3.

Una vez que hemos completado todos estos pasos somos capaces de elaborar modelos conformes a un determinado lenguaje, unívocamente especificado por un metamodelo. El último paso para definir de forma completa un lenguaje sería definir su semántica. Dependiendo del lenguaje que estemos creado, dicha semántica podría ser de diferentes tipos. Por ejemplo, para el caso de un lenguaje basado en Redes de Petri, la semántica podría ser una semántica dinámica que especifique como deben ejecutarse los modelos basados en Redes de Petri. Dicha semántica dinámica debería permitir construir sin ambigüedades un simulador o máquina virtual para dicho lenguaje.

En otros casos, la semántica podría definirse de forma *translacional*, mediante la transformación del modelo en otro modelo con semántica bien definida. Este sería el caso, por ejemplo, de los lenguajes compilados, donde un programa escrito en un cierto lenguaje se transforma en un código ensamblador. En este caso la semántica quedaría implementada por medio de un compilador generador de código.

Por tanto, a modo de resumen, un proceso de desarrollo de un lenguaje software utilizando un enfoque dirigido por modelos está formado por los siguientes pasos:

1. Definición del metamodelo que especifica la sintaxis abstracta de nuestro lenguaje de modelado.

2. Definición de las restricciones adicionales que no pueden ser recogidas mediante la sintaxis propia del lenguaje de metamodelado utilizado.

3. Definición de una sintaxis concreta, visual o textual, para el metamodelo definido.

4. Generación (automática) del correspondiente editor, textual o gráfico.

5. Definición de la semántica del lenguaje.

6. Implementación de dicha semántica mediante la técnica que se considere más adecuada para ella (simulador, máquina virtual, generación de código).

Una vez explicado cómo funciona la Ingeniería de Lenguajes Dirigida por Modelos estamos preparados para poder definir como se ha estructurado y desarrollado el presente Proyecto Fin de Carrera. Dicha planificación se presenta en la siguiente sección.

1.3. Planificación del proyecto

Como se ha comentado con anterioridad, el objetivo de este Proyecto Fin de Carrera es el desarrollo de un editor para un novedoso lenguaje de especificación y validación de restricciones para árboles de características donde dichas restricciones puedan incluir características clonables. Dicho editor se desarrollará utilizando un moderno enfoque de *Ingeniería de Lenguajes Dirigido por Modelos*. Por tanto, el proceso de desarrollo del presente proyecto queda prácticamente determinado por dicho enfoque, el cual posee un proceso de desarrollo bien definido, el cual se describió en la sección anterior. La Figura 1.6 muestra como dicho proceso de desarrollo se ha instanciado para nuestro caso particular.

Obviamente, la primera tarea (Figura 1.6-*T1*) en este proceso de desarrollo fue la de adquirir los conocimientos necesarios para la realización de todas las tareas posteriores. Ello implicaba adquirir los conocimientos relacionados con las *Líneas de Producto Software* [] en general y con los árboles de características [] en particular, más concretamente, con la vesión de los árboles de características que soportan la definición de características clonables []. Dado que el proyecto se debía integrar con una herramienta para la especificación y configuración de árboles de características concreta, denominada *Hydra* [], el siguiente paso fue el de familiarizarse con dicha herramienta y adquirir ciertos conocimientos sobre su arquitectura interna.

A continuación, se tuvo que adquirir los conceptos necesarios para entender el funcionamiento de de la *Ingeniería de Lenguajes Dirigida por Modelos* []. La familiarización con las tecnologías concretas relacionadas con la *Ingeniería de Lenguajes Dirigida por Modelos*, como la utilización de EMF (*Eclipse Modelling Framework*) [] para la definición de metamodelos, se realizó dentro de cada fase concreto del proyecto, a medida que se iba necesitando aprender a utilizar dichas tecnologías.

Tras esta tarea inicial de adquisición de conocimientos previos, el resto del proyecto se estructura como un proyecto de desarrollo de un lenguaje software siguiendo un enfoque dirigido por modelos. Consecuentemente, la primera tarea tras la fase inicial de documentación (Figura 1.6-*T2*) fue la definición de la sintaxis abstracta, por medio de un metamodelo más un conjunto de restricciones externas, para el lenguaje que debía soportar nuestro editor. Para ello tuvimos que capturar los requisitos que debía satisfacer dicho lenguaje. Tras recoger dichos requisitos, se procedió al diseño del metamodelo y a la relización de las pruebas pertinentes con vistas a comprobar su correcto funcionamiento. Para crear dicho metamodelo se utilizó el lenguaje de metamodelado Ecore, integrado dentro de la herramienta EMF (Eclipse Modelling Framework) [].

A continuación, de acuerdo con los expuesto en la sección anterior, procedimos a definir la las restricciones externas que no podían ser definidas en Ecore (Figura 1.6-*T3*). Dichas restricciones se implementaron utilizando la facilidad de EMF denominada EMF Validation Framework [].

A continuación, procedimos a definir la sintaxis concreta, en nuestro caso textual, para nuestro lenguaje de modelado. Optamos por una sintaxis textual ya que las restricciones a especificar son una especie de fórmulas lógicas, las cuales resultan más cómodas de especificar mediante notaciones textuales que mediante notaciones gráficas (Figura 1.6-*T4*). Para el desarrollo de dicha sintaxis textual hubo que hacer un nuevo análisis de los requisitos que dicha notación textual debía satisfacer. A continuación se especificó la gramática de nuestra sintaxis textual, ligando sus elementos con los del metamodelo producido en la fase anterior y se ejecutaron los casos de prueba necesarios para comprobar su correcto funcionamiento. Para crear dicha sintaxis textual se utilizó la herramienta EMFText [].

Las etapas anteriores permitían disponer de un editor que soportaba la especificación de restricciones de acuerdo al lenguaje HCL. Por tanto, sólo restaba poder comprobar, para una configuración dada de un árbol de características, que dichas restricciones se satisfacían. Ello implicaba dotar de semántica al lenguaje y, a partir de dicha semántica, implementar los mecanismos necesarios para la comprobación de la validez de dichas restricciones. La semántica del lenguaje ya estaba definida por el profesor Pablo Sánchez, por lo que sólo hubo que implementar el código necesario para procesar un modelo de restricciones y comprobar que dichas restricciones se satisfacían. Dicho código se implementó en Java, utilizando las facilidades que el entorno EMF proporciona para la manipulación del modelo (Figura 1.6-*T5*). Para poder implementar la semántica, fue necesario crear una interfaz de comunicación con la herramienta *Hydra* que permitiese conocer el estado en el cual se hallaba cada característica. Tras la implementación, se ejecutó un exhaustivo conjunto de pruebas para comprobar el correcto funcionamiento del código creado.

En este punto del proceso de desarrollo teníamos implementado el editor

requerido, por lo que sólo restaba proceder a su despliegue (Figura 1.6-*T6*). Este despliegue implicaba su integración dentro de la arquitectura de plugins de Eclipse y, más concretamente, de la herramienta *Hydra*. Tras dicha integración, se procedió a realizar una serie de pruebas de aceptación, destinadas a comprobar que el trabajo realizado satisfacía las necesidades de los usuarios finales que iban a utilizar el editor creado.

1.4. Estructura del Documento

≪TODO: Arreglar esto cuando el proyecto esté terminado≫

Tras este capítulo de introducción, la presente memoria de Proyecto Fin de Carrera se estructura tal y como se describe a continuación: El Capítulo 2 describe en términos generales todos los conceptos y tecnologías que son necesarios para comprender el contenido de este documento. El Capítulo **??** explica la planificación del proyecto desde el punto de vista de las tareas involucradas. El capítulo 4 describe en profundidad la parte correspondiente a la creación de las sintaxis concreta y abstracta. El capítulo 5 describe el resto de tareas que se han llevado a cabo en el proyecto, y el capítulo 6 describe mis conclusiones personales y la situación de la herramienta de cara al futuro.

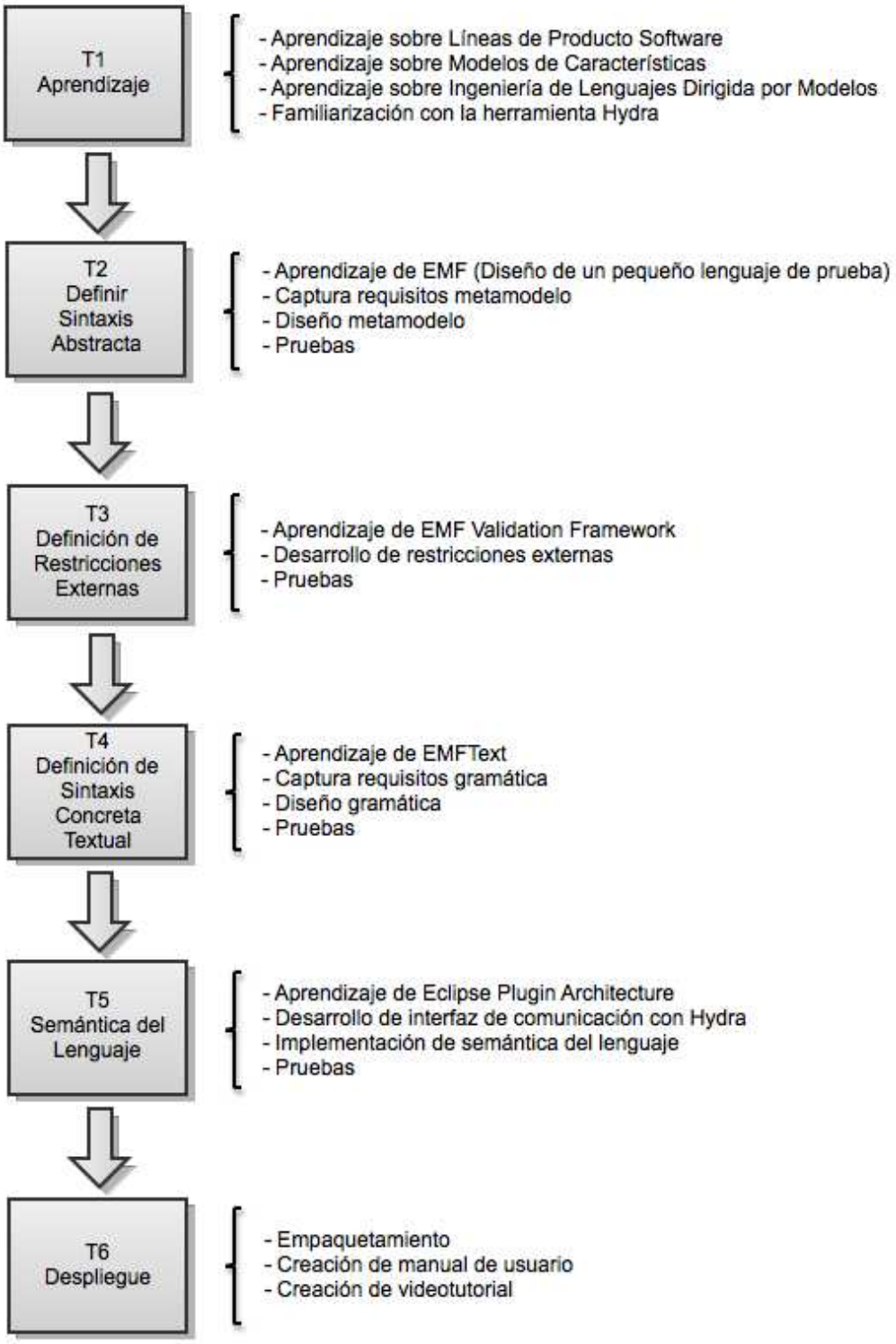

Figura 1.6: Proceso de desarrollo del Proyecto Fin de Carrera

Capítulo 2

Antecedentes

Este capítulo trata de describir a grandes rasgos las técnicas, tecnologías y herramientas utilizadas para el desarrollo del presente Proyecto Fin de Carrera. En primer lugar, se introducirá el caso de estudio que se utilizará de forma recurrente a lo largo del proyecto, que es una línea de productos software para software de control para hogares inteligentes. Para ello se describen en primer lugar diversos conceptos relacionados el dominio del proyecto, como son las líneas de productos software y los árboles de características. A continuación, se describen dos principales herramientas de Ingeniería de Lenguajes Dirigida por Modelos utilizadas: *Ecore* y *EMFText*. Por último, se describe brevemente la arquitectura de plugins de Eclipse, dado que nuestro proyecto debía integrarse en dicho entorno.

Índice

2.1. Captura de requisitos 11
2.2. Diseño de la gramatica 12
2.3. Pruebas 15

2.1. Caso de Estudio: Software para Hogares Inteligentes

El objetivo último del presente proyecto es la construcción de una línea de productos software sobre la plataforma .NET para hogares automatizados y/o inteligentes.

El objetivo de estos hogares es aumentar la comodidad y seguridad de sus habitantes, así como hacer un uso más eficiente de la energía consumida. Los ejemplos más comunes de tareas automatizadas dentro de un hogar inteligente son el control de las luces, ventanas, puertas, persianas, aparatos de frío/calor, así como otros dispositivos, que forman parte de un hogar. Un hogar inteligente también busca incrementar la seguridad de sus habitantes

mediante sistemas automatizados de vigilancia y alerta de potenciales situaciones de riesgo. Por ejemplo, el sistema debería encargarse de detección de humos o de la existencia de ventanas abiertas cuando se abandona el hogar.

El funcionamiento de un hogar inteligente se basa en el siguiente esquema: (1) el sistema lee datos o recibe datos de una serie de sensores; (2) se procesan dichos datos; y (3) se activan los actuadores para realizar las acciones que correspondan en función de los datos recibidos de los sensores.

Todos los sensores y actuadores se comunican a través de un dispositivo especial denominado puerta de enlace (*Gateway*, en inglés). Dicho dispositivo se encarga de coordinar de forma adecuada los diferentes dispositivos existentes en el hogar, de acuerdo a los parámetros y preferencias especificados por los habitantes del mismo. Los habitantes del hogar se comunicarán con la puerta de enlace a través de una interfaz gráfica. Este proyecto tiene como objetivo el desarrollo de un hogar inteligente como una línea de productos software, con un número variable de plantas y habitaciones. El número de habitaciones por planta es también variable. La línea de productos deberá ofrecer varios servicios, que podrán ser opcionalmente incluidos en la instalación del software para un un hogar determinado. Dichos servicios se clasifican en funciones básicas y complejas, las cuales describimos a continuación.

Funciones básicas

1. *Control automático de luces:* Los habitantes del hogar deben ser capaces de encender, apagar y ajustar la intensidad de las diferentes luces de la casa. El número de luces por habitación es variable. El ajuste debe realizarse especificando un valor de intensidad.

2. *Control automático de ventanas:* Los residentes tienen que ser capaces de controlar las ventanas automáticamente. De tal modo que puedan indicar la apertura de una ventana desde las interfaces de usuario disponibles.

3. *Control automático de persianas:* Los habitantes podrán subir y bajar las persianas de las ventanas de manera automática.

4. *Control automático de temperatura:* El usuario será capaz de ajustar la temperatura de la casa. La temperatura se medirá siempre en grados celsius.

Funciones complejas

1. *Control inteligente de energía:* Esta funcionalidad trata de coordinar el uso de ventanas y aparatos de frío/calor para regular la temperatura interna de la casa de manera que se haga un uso más eficiente de la energía. Por ejemplo, si se recibe la orden de calentar la casa, a la vez que se activan los radiadores se cerrarán las ventanas para evitar las pérdidas de calor.

2. *Presencia simulada:* Para evitar posibles robos, cuando los habitantes abandonen la casa por un periodo largo de tiempo, se deberá poder simular la presencia de personas en las casas. Hay dos opciones de simulación (no exclusivas):

   a) *Simulación de las luces:* Las luces se deberán apagar y encender para simular la presencia de habitantes en la casa.

   b) *Simulación de persianas:* Las persianas se deberán subir y bajar automática para simular la presencia de individuos dentro de la casa.

Todas estas funciones son opcionales. Las personas interesadas en adquirir el sistema podrán incluir en una instalación concreta de este software el número de funciones que ellos deseen. La siguiente sección describe la planificación general realizada para desarrollar este caso de estudio.

2.2. Líneas de producto software

El objetivo de una *línea de productos software* [**?**, **?**] es crear una infraestructura adecuada a partir de la cual se puedan derivar, tan automáticamente como sea posible, productos concretos pertenecientes a una familia de productos software. Una familia de productos software es un conjunto de aplicaciones software similares, que por tanto comparten una serie de características comunes, pero que también presentan variaciones entre ellos.

Un ejemplo clásico de familia de productos software es el software que se encuentra instalado por defecto en un teléfono móvil. Dicho software contiene una serie de facilidades comunes, tales como agenda, recepción de llamadas, envío de mensajes de texto, etc. No obstante, dependiendo de las capacidades y la gama del producto, éste puede presentar diversas funcionalidades opcionales, tales como envío de correos electrónicos, posibilidad de conectarse a Internet mediante red inalámbrica, radio, etc.

La idea de una línea de productos software es proporcionar una forma automatizada y sistemática de construir productos concretos dentro de una familia de productos software mediante la simple especificación de qué características deseamos incluir dentro de dicho producto. Esto representa una alternativa al enfoque tradicional de desarrollo software, el cual se basaba

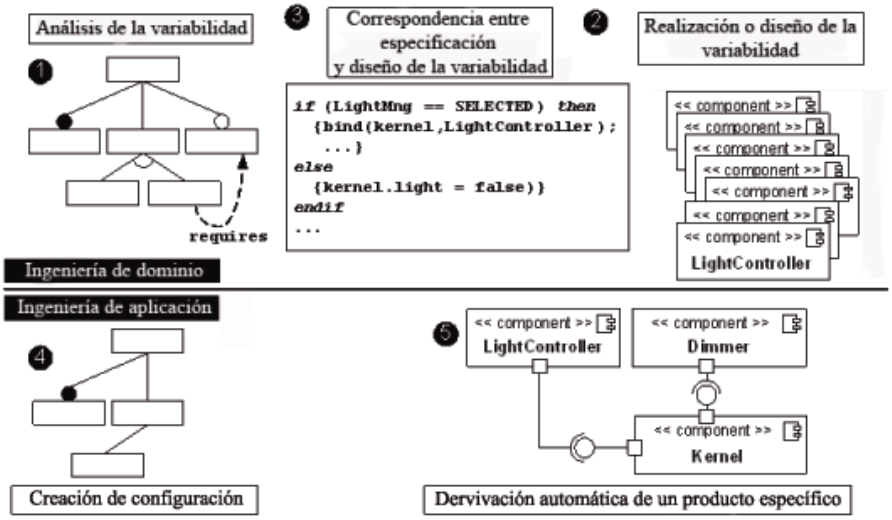

Figura 2.1: Proceso de desarrollo de una línea de productos software

simplemente en seleccionar el producto más parecido dentro de la familia al que queremos construir y adaptarlo manualmente.

El proceso de creación de líneas de producto software conlleva dos fases: *ingeniería del dominio* (en inglés, *Domain Engineering*) e *ingeniería de aplicación* (en inglés, *Application Engineering*) (laFfigura 2.1 ilustra el proceso para ambas fases). La *ingeniería del dominio* tiene como objetivo la creación de la infraestructura o arquitectura de la línea de productos, la cual permitirá la rápida, o incluso automática, construcción de sistemas software específicos pertenecientes a la familia de productos. La *ingeniería de aplicación* utiliza la infraestructura creada anteriormente para crear aplicaciones específicas adaptadas a las necesidades de cada usuario en concreto.

En la fase de ingeniería del dominio, el primer paso a realizar sería un análisis de qué características de la familia de productos son variables y por qué y cómo son variables. Esta parte es la que se conoce como *análisis o especificación de la variabilidad* (Figura 2.1, punto 1). A continuación, se ha de diseñar una arquitectura o marco de trabajo para la familia de productos software que permita soportar dichas variaciones. Esta actividad se conoce como *realización o diseño de la variabilidad* (Figura 2.1, punto 2). El siguiente paso es establecer una serie de reglas que especifiquen como hay que instanciar la arquitectura previamente creada de acuerdo con las características seleccionadas por cada cliente. Esta fase es la que se conoce como *correspondencia entre especificación y diseño de la variabilidad* (Figura 2.1, punto 3).

En la fase de ingeniería de aplicación, se crearía una *configuración*, que no es más que una selección de características que un usuario desea incluir en su producto (Figura 2.1, punto 4). En el caso ideal, usando esta con-

Figura 2.2: Árbol de características simple para nuestro caso de estudio

figuración, deberíamos poder ejecutar las reglas de correspondencia entre especificación y diseño de la variabilidad para que la arquitectura creada en la fase de ingeniería del dominio se adaptase automáticamente generando un producto concreto específico acorde a las necesidades concretas del usuario (Figura 2.1, punto 5). En caso no ideal, dichas reglas de correspondencia deberán ejecutarse a mano, lo cual suele ser un proceso tedioso, largo, repetitivo y propenso a errores.

Este proyecto se centra en la primera etapa de este proceso de desarrollo, es decir en el análisis de la variabilidad de una familia de productos software mediante árboles de características. La siguiente sección proporciona una breve pero completa descripción acerca del funcionamiento de los árboles de características.

2.3. Árboles de características

Como se ha comentado en la sección anterior, una de las tarea claves para el éxito de una línea de productos software consiste en analizar la variabilidad existente en la familia de productos software que dicha línea de productos software pretende cubrir. Aquí es donde entran en juego los *Árboles de características* []. Una *característica* se define como "*un incremento en la funcionalidad del producto*", o más formalmente, "*una característica es una propiedad de un sistema que es relevante a algunos* stakeholders *y que es utilizada para capturar propiedades comunes o diferenciar entre sistemas de una misma familia*-[]. De este modo un producto queda representado por las características que posee.

Para poder capturar las divergencias y aspectos comunes entre los distintos productos de una misma familia, los árboles de características organizan de forma jerárquica el conjunto de características que posee una familia de productos. Cada característica se representa como un nodo ene la árbol de características. La raíz de dicho árbol es siempre el sistema o producto software cuya variabilidad estamos analizando. Cada característica se puede descomponer en varias subcaracterísticas, siendo estás últimas nodos *hijos* de la primera características, que actuaría como *padre*. Dependiendo de sis dichas subcaracterísticas son obligatorias, alternativas u opcionales, existen diversos tipos de relaciones padre-hijo.

La Figura 2.2 muestra com ejemplo un árbol de características que especifica la variabilidad inherente a nuestro caso de estudio, sin considerar que las plantas y habitaciones puedan configurarse de manera individual.

Una vez creado un árbol de características para una línea de productos software, podemos indicar las características que podemos incluir en un

Figura 2.3: Árbol de características completo para nuestro caso de estudio

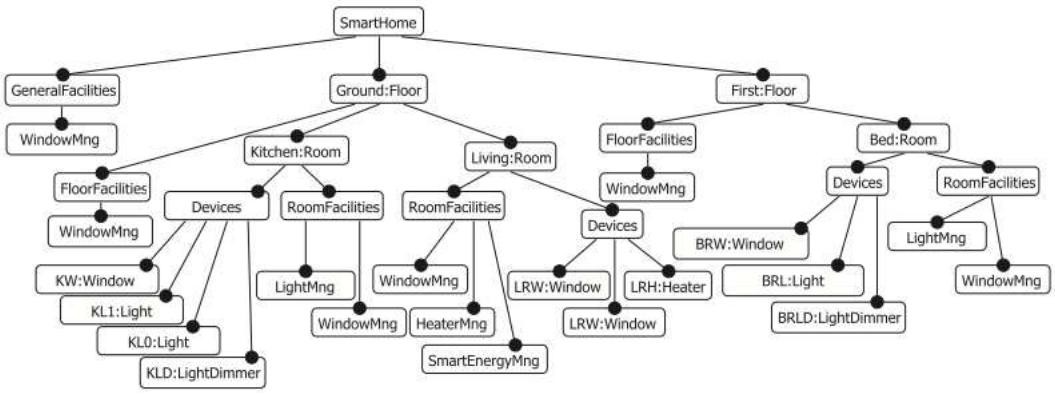

Figura 2.4: Posible modelo de configuración de una casa inteligente concreta

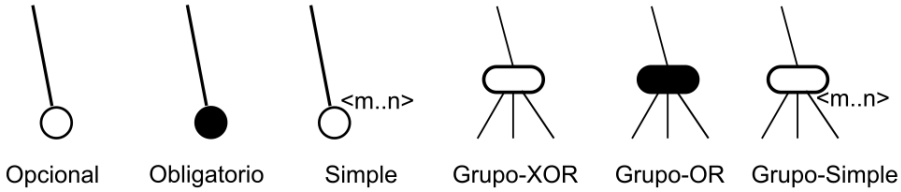

Figura 2.5: Tipos de relaciones entre características

producto software concreto mediante la creación de configuraciones. Una *configuración* no es más que una selección válida de características. La Figura **??** muestra en un ejemplo de configuración para el modelo de la Figura 2.2 donde se indica que el producto que deseamos construir debe incluir ≪TODO: COMPLETAR≫. Obviamente, dicho modelo debe satisfacer las restricciones externas declaradas.

Los árboles de características como los anteriormente expuestos no permiten modelar que pueda existir un número variable de ciertas características, como, en nuestro caso, de plantas y habitaciones, y que, además, cada instancia particular de una características pueda configurarse de forma distinta. Por ejemplo, podríamos decidir que el salón de la casa tenga control inteligente de temperatura, mientras que la cocina, que está sometida a mayores variaciones de temperatura, no contenga dicha características. Para solventar esta carencia, se introdujeron en los árboles de características el concepto de característica clonable.

La Figura 2.3 muestra el árbol de características para nuestro caso de estudio incluyendo *características clonables*. ≪TODO: Explicarla≫

A modo de resumen, la Figura 2.5 muestra las posibles relaciones que

pueden entre características, así como su representación gráfica. Dichas relaciones se describen a continuación:

Opcional La característica hija puede estar o no estar seleccionada

Obligatoria La característica siempre debe estar seleccionada.

Clonable La característica tendrá una cardinalidad $< m, n >$, siendo m y n números enteros que denotan el mínimo y el máximo respectivamente de características que podemos seleccionar.

Grupo-xor Sólo una de las características pertenecientes al grupo puede ser seleccionada.

Grupo-or Debemos seleccionar como mínimo una de las subcaracterísticas, pudiendo seleccionar más si lo deseamos.

Grupo con cardinalidad El número mínimo y máximo de características a seleccionar dentro del grupo vendrá determinado por su cardinalidad $< m, n >$.

Tras esta sección se han proporcionado al lector lo necesario para comprender el contexto del problema que este Proyecto Fin de Carrera pretende resolver. Las siguientes secciones están destinadas a explicar las tecnologías concretas que se han utilizado para implementar el lenguaje que da solución a los problemas planteados.

2.4. *Eclipse Modelling Framework*

EMF *Eclipse Modeling Framework* [] es un *plug-in* para Eclipse [] que permite elaborar metamodelos. Pera ello proporciona un lenguaje de metamodelo denominado Ecore, el cual se ha convertido en el estándar de factor para la realización de metamodelos. Utilizando Ecore se pueden crear metamodelos de forma gráfica usando una notación muy similar a la los diagramas de clases de UML. La Figura 1.2 muestra un sencillo ejemplo de metamodelo en Ecore (ver Sección 1.2 para más detalles). EMF también incorpora una herramienta para la validación reglas adicionales que no puedan ser especificadas a nivel de del metamodelo.

EMF permite que, a partir de un metamodelo especificado en Ecore, podamos, utilizando diversos generadores de código, crear automáticamente un conjunto de clases que nos permiten manipular dichos modelos a nivel de código. Dichas clases se pueden además distribuir como *plug-in* para el entorno Eclipse.

Además, al haberse convertido en estándar *de facto* para el desarrollo de metamodelos, Ecore es compatible con multitud de herramientas para Ingeniería de Lenguajes Dirigida por Modelos, como EMFText, la cual se

describe en la siguiente sección, o diversos generadores de código o herramientas de transformación de modelos.

2.5. EMFText

EMFText es una herramienta específicamente diseñada para diseñar las gramáticas de los lenguajes que hayan sido diseñados previamente con un metamodelo de Ecore. Está especializado para la creación de Lenguajes Específicos de Dominio, aunque también se pueden crear lenguajes de propósito general.

Pero, como en casi todos los casos de este tipo de herramientas, su mayor virtud es la enorme cantidad de código autogenerado que produce, y que elimina al programador de tareas tediosas que además en muchos casos podrían resultar complicadas. Todo el código generado es completamente independiente de EMFText, es decir, podrá ser ejecutado en plataformas que no tengan la herramienta instalada.

Todo el código generado por EMFText está diseñado de tal modo que sea fácil de modificar en caso de que queramos poner en práctica algunas funcionalidades poco habituales. Se facilita mucho la labor a la hora de modificar estructuras como el postprocesador de nuestra gramática. Todas las gramáticas construidas tendrán que ser LL por defecto, a no ser que queramos modificar los parsers generados, posibilidad también disponible.

Otro tipo de funcionalidades implementadas, quizás no tan importantes pero también de gran utilidad, son el coloreado de código (por defecto o personalizable), función de completar código, generación del árbol parseado en el la vista de eclipse Outline o generación de código para crear un depurador para nuestro lenguaje.

EMFText permite la definición de gramáticas utilizando un lenguaje estándar para la definición de expresiones regulares, además de incorporar algunas particularidades propias que facilitan ciertas tareas. En la figura 2.6 se muestra una pequeña captura que contiene una porción de la gramática construida para nuestro editor de especificación y validación de restricciones.

2.6. Arquitectura de plugins de Eclipse

Un plug-in en Eclipse es un componente que provee un cierto tipo de servicio dentro del contexto del espacio de trabajo de eclipse, es decir, una herramienta que se puede integrar en el entorno Eclipse junto con sus otras funcionalidades. Dado que la herramienta Hydra fue diseñada como un plug-in para Eclipse, y nuestro editor es una parte de la misma, ha sido necesario aprender el manejo de algunas de las funcionalidades de la arquitectura de plug-ins de Eclipse.

```
START Model

OPTIONS {
    usePredefinedTokens="true";
}

TOKENS {
    DEFINE DIGIT $('0'..'9')+$;
    DEFINE DIRECCION $('A'..'Z'|'a'..'z'|'0'..'9'|'_'|'-'|'/'|'.')+$;
}

RULES {

Model::= "import" featureList[DIRECCION] ";" (constraints";")*;
Constraint::= operators | "(" operators ")";
BoolPriorityOperand1::= "(" boolPriorityOp1 ")" | boolPriorityOp1:BoolOperandChoices;
BoolPriorityOperand2::= "(" boolPriorityOp2 ")" | boolPriorityOp2;
NumPriorityOperand1::= numPriorityOp1:NumOperandChoices | "(" numPriorityOp1 ")";
NumPriorityOperand2::= numPriorityOp2 | "(" numPriorityOp2 ")";

    // Operaciones logicas
    And ::= binaryOp1 "and" binaryOp2;
    Or ::= binaryOp1 "or" binaryOp2;
    Xor ::= binaryOp1 "xor" binaryOp2;
    Implies ::= binaryOp1 "implies" binaryOp2;
    Neg ::= "!" unaryOp;
```

Figura 2.6: Trozo de la gramática de nuestro editor de especificación y validación de restricciones

En particular, se han utilizado mucho los puntos de extensión. Un punto de extensión en un plug-in indica la posibilidad de que ese plug-in sea a su vez parte de otro, o que haya otros plug-ins que sean parte de él. Esta particularidad permite no sólo la integración de nuestro editor con Hydra, sino también la personalización de menús y botones para él gracias a la creación de puntos de extensión con plug-ins de creación de menús y barras de herramientas.