FACULTAD DE CIENCIAS

UNIVESIDAD DE CANTABRIA

Proyecto Fin de Carrera

Desarrollo de un entorno para la especificación y validación de restricciones en árboles de características con cardinalidad

(Development of enviroment for the specification and validation of constraints for cardinality-based feature model)

Para acceder al Título de
INGENIERO EN INFORMÁTICA

Autor: Daniel Tejedo González
Julio 2011

FACULTAD DE CIENCIAS

INGENIERÍA EN INFORMÁTICA

CALIFICACIÓN DEL PROYECTO FIN DE CARRERA

Realizado por:	Daniel Tejedo González
Director del PFC:	Pablo Sánchez Barreiro
Título:	Desarrollo de un entorno para la especificación y validación de restricciones en árboles de características con cardinalidad
Title:	Development of enviroment for the specification and validation of constraints for cardinality-based feature model

Presentado a examen el día:

para acceder al Título de
INGENIERO EN INFORMÁTICA

Composición del Tribunal:

Presidente (Apellidos, Nombre):
Secretario (Apellidos, Nombre):
Vocal (Apellidos, Nombre):
Vocal (Apellidos, Nombre):
Vocal (Apellidos, Nombre):

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: Vocal

Fdo.: Vocal

Fdo.: Vocal

Fdo.: El Director del PFC

II

Agradecimientos

TODO: Aquí se suelen poner agradecimientos si uno quiere y dedicatorias.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Esta memoria de Proyecto Fin de Carrera presenta un entorno para la especificación y validación de restricciones en árboles de características con cardinalidad. El entorno ha sido desarrollado dentro del contexto de la aplicación Hydra, una herramienta para el modelado de características. Este capítulo introduce la situación anterior del proyecto Hydra y los conceptos implicados en el mismo, así como los nuevos conceptos que esta ampliación incorpora. Se describirán también los objetivos que este trabajo pretende alcanzar y la motivación que subyace tras ellos. Por último, se indicará la estructura que presentará este documento.

Contents

1.1. Introducción

El principal objetivo de este Proyecto de Fin de Carrera es continuar el desarrollo de la herramienta Hydra allá dónde se dejó. Pero para entender un poco mejor las características del entorno que aquí se ha desarrollado conviene explicar un poco las razones que motivaron la creación del proyecto Hydra en primera instancia. Desde su nacimiento, Hydra ha pretendido convertirse en la aplicación más completa para trabajar con Líneas de Producto Software, dado que no existe ninguna que ofrezca una serie de características de manera conjunta. Trabajar con Líneas de productos software conlleva a su vez trabajar con una nada desdeñable cantidad de conceptos íntimamente vinculados a ellas. En los próximos párrafos se describirá lo que es una línea de productos software y los conceptos subyacentes que nos permiten trabajar con ellas.

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, que son reutilizadas para la construcción de productos similares. Un ejemplo clásico es la fabricación de automóviles, donde se pueden crear decenas de variaciones de un único modelo de coche con un solo grupo de piezas cuidadosamente diseñadas y una fábrica específicamente concebida para configurar y ensamblar dichas piezas.

Una pieza clave en la creación y desarrollo de una línea de productos es el análisis y especificación de qué elementos son comunes y qué elementos son variables dentro del conjunto de productos similares producidos por la línea de productos software. Para realizar dicha tarea se suelen construir modelos de características **??** **??** **??**. Dichos modelos definen una descomposición jerárquica, en forma de árbol, de las características de una serie de productos similares. Un característica se define, de forma genérica, como un elemento visible del sistema, de interés para alguna persona que interactúa con el sistema, ya sea un usuario final o un desarrollador software **??**. Por ejemplo, una característica de un coche sería su color. Una característica puede ser descompuesta en varias subcaracterísticas que pueden ser obligatorias, opcionales o alternativas. Continuando con el mismo ejemplo, podemos entender que un utilitario convencional tiene obligatoriamente cuatro ruedas, puede opcionalmente tener GPS y se puede adquirir en una serie de colores alternativos. Para obtener un producto específico, el usuario debe crear una configuración de este modelo de características, es decir, una selección de características que quiere que estén presentes en su producto.

Una configuración o selección de características debe de obedecer las reglas del modelo de características. Por ejemplo, dentro de un conjunto de alternativas mutuamente excluyentes, como el color de un automóvil, sólo una alternativa de entre todas las posibles debe ser seleccionada. Existen una serie de restricciones que no se pueden modelar con la sintaxis básicas de los modelos de características. Un ejemplo de tales tipos de restricciones son las restricciones de implicación, que especifican que la selección de una característica obliga a la selección de otra característica diferente. Por ejemplo, la selección de GPS para un automóvil podría obligar a la inclusión en dicho automóvil de mandos integrados en el volante para el manejo tanto de la radio como del GPS. Estas restricciones se suelen modelar de forma externa al modelo de característica, usando algún tipo de formalismo adicional, como lógica proposicional. En el ejemplo planteado, la anterior restricción se modelaría como una simple implicación entre características. Surge por tanto un nuevo reto dentro del modelado de características, que es la validación de las configuraciones respecto a tales restricciones externas.

Por último, cabe destacar que en los últimos años se ha añadido a los

modelos de características un simple pero importante concepto, como son las características clonables **??**, que son características que pueden aparecer con un diferente número o cardinalidad dentro de un producto. Por ejemplo, el modelo de características de una casa puede tener como característica clonable planta, dado que una casa posee un número variable de plantas, es decir, usando nuestra terminología, diferentes clones de la característica planta. Esta ligera modificación hace que se pueda modelar variabilidad estructural, tal como que una casa tenga un número variable de plantas, en los modelos de características, acercando su potencia expresiva a la de los lenguajes de dominio específico para líneas de producto software **??**.

En la actualidad existen diversas herramientas para crear modelos de configuraciones, en su mayoría académicas, tales como RequiLine **??** o fmp **??**, siendo ésta última posiblemente la más conocida. No obstante, no existe ninguna herramienta en este momento que posea las siguientes características de forma conjunta:

1- Una interfaz gráfica, amigable y que, en la medida de lo posible, asista al usuario en la creación de configuraciones.

2- Soporte el modelado y la configuración de características clonables.

3- Permita la especificación de restricciones externas entre características, incluyendo características clonables.

4- Sea capaz de determinar si una determinada configuración es válida, es decir, no sólo obedece las reglas sintácticas del modelo de características, sino que también se satisfacen las restricciones externas. Estas restricciones externas pueden estar definidas sobre características clonables.

Para resolver tales limitaciones de las herramientas actuales para el modelado de características, hemos creado Hydra. Hydra es una herramienta para el modelado de características que pretende proporcionar las siguientes funcionas:

1- Un editor completamente gráfico y amigable al usuario para la construcción de modelos de características.

2- un editor textual y una sintaxis propia para la especificación de restricciones entre características.

3- Un editor gráfico, asistido y amigable al usuario para la creación de configuraciones de modelos de características.

4- Un validador que comprueba que las configuraciones creadas satisfacen las restricciones definidas para el modelo de características.

En el momento de empezar este proyecto de fin de carrera Hydra ya tenía implementadas las funciones 1 y 3, pero aún faltaban por implementar las funciones 2 y 4, labor de la que nos hemos hecho cargo en este proyecto.

Para crear el lenguaje de dominio específico que permita la especificación de restricciones externas entre características se ha utilizado la técnica de Ingeniería de Lenguajes Dirigido Por Modelos. Esta técnica esa una aproximación de la Ingeniería Dirigida Por Modelos desde el punto de vista de la Teoría de Lenguajes Formales. La Ingeniería Dirigida Por Modelos es una metodología de desarrollo de software que se basa en la construcción de la aplicación final a partir de uno o más modelos abstractos que representen el comportamiento y la funcionalidad de la misma. Mediante la modificación de los distintos parámetros configurables dentro de los modelos será posible la construcción de herramientas diversas para un problema específico de manera relativamente sencilla, bastando simplemente con crear una serie de instancias válidas de los modelos representativos de nuestra aplicación.

La Ingeniería Dirigida Por Modelos se puede usar para crear nuevos lenguajes de programación, especialmente DSLs o Lenguajes Específicos de Dominio. Basta con imaginar uno o varios metamodelos cuyas instancias válidas representen una línea o estructura correcta de código. Estos metamodelos forman la llamada sintaxis abstracta de nuestro lenguaje, pues representan de manera abstracta todas las posibles representaciones gráficas o textuales que podemos hacer dentro de ese lenguaje.

A partir del metamodelo que representa la sintaxis abstracta podremos construir una serie de modelos que sean instancias del mismo. Esto se denomina la sintaxis concreta, es decir, la representación concreta de una de las múltiples posibilidades de instanciación del modelo abstracto. De todos modos, si queremos que las palabras admitidas por nuestro lenguaje sean expresadas de otro modo aparte de mediante modelos, no nos veremos liberados de la tarea de tener que expresar una gramática formal con producciones, pero hace que la construcción de la misma esté acotada dentro de unos términos delimitados por el modelo construído, lo cual favorece la sencillez de la gramática y su comprensión. Esta gramática servirá para identificar si la creación de expresión concreta a la que puede ser transformada es viable mediante el metamodelo abstracto.

Una vez hemos construido los medios necesarios para comprobar que las expresiones que fabriquemos son correctas, necesitamos idear el modo de que las órdenes que esas líneas producen sean ejecutadas. Para ello entra en juego la semántica del lenguaje, es decir, la encargada de aportar un significado real a todas las expresiones que hayamos construido. Dicho de otro modo, la semántica es la encargada de implementar las funciones derivadas de las órdenes descritas por cada una de las sintaxis concretas posibles. Poniendo un ejemplo, el metamodelo que represente la sintaxis abstracta de java puede generar una infinidad de sintaxis concretas, entre ellas System.out.println("Hola Mundo"). Y para que esa instrucción escriba el mensaje Hola Mundo por pantalla es necesaria una semántica que así lo indique. Para implementar la semántica del lenguaje no existe otro método que la programación directa en un lenguaje determinado, en nuestro caso Java.

Por último y para terminar esta introducción, conviene contextualizar un poco el trabajo que hemos hecho mediante un ejemplo concreto de lo que se quiere implementar. Nuestro editor para la especificación y validación de restricciones en árboles de características con características clonables tiene que implementar la siguiente funcionalidad:

1 - Obligar a que todas los ficheros de restricciones empiecen con una línea de import que servirá para importar el modelo de características sobre el cual se analizarán las restricciones.

2 - Escribir restricciones válidas para ese modelo (ejemplo: "(a or b) implies (c and d);", más adelante se hablará en detalle del lenguaje y de las operaciones que implementa).

3 - Detectar que las características a las que estamos aplicando esas restricciones en efecto se hallan en el modelo que ha sido importado.

4 - Cargar una instancia de ese modelo (lo que llamamos modelo de especialización) y mirar si para ella las restricciones que han sido especificadas se cumplen.

1.2. Objetivos

Como ya se ha comentado en la sección de introducción, no existe ninguna herramienta que posea de forma conjunta una serie de elementos de interés para el modelado de Líneas de Productos Software y Árboles de Características. Más concretamente, no existe ninguna herramienta que contemple el modelado, configuración y validación de características clonables. Estas características son imprescindibles para el modelado de la variabilidad estructural. Por lo tanto, el objetivo de Hydra siempre fue suplir esas carencias, en la medida de lo posible.

Concretando más en concreto, los objetivos de Hydra se pueden clasificar en los 4 que se enumeran a continuación:

1. Desarrollar un editor completamente gráfico y amigable al usuario para la construcción de modelos de características, incluyendo soporte para el modelado de características clonables.

2. Desarrollar un editor textual y una sintaxis propia para la especificación de restricciones entre características, incluyendo restricciones que involucren características clonables.

3. Desarrollar Un editor gráfico, asistido y amigable al usuario para la creación de configuraciones de modelos de características, incluyendo soporte para la configuración de características clonables.

4. Crear un validador que compruebe que las configuraciones creadas satisfacen las restricciones definidas para el modelo de características, incluso cuando estas restricciones contengan características clonables.

La labor a desarrollar dentro del marco concreto de este proyecto de fin carrera fue continuar el proyecto Hydra donde se había dejado anteriormente, es decir, una vez los objetivos 1 y 3 habían sido cumplimentados, pasar a implementar la funcionalidad correspondiente a los objetivos 2 y 4. Para satisfacer dichos objetivos, se realizaron las tareas que se describen a continuación:

1. Estudio del estado del arte. El objetivo de esta fase es adquirir los conceptos necesarios para comprender el contexto del proyecto Hydra, así como los necesarios para continuar desarrollando la aplicación en el punto en que fue visitada por última vez. Más concretamente, ha sido fundamental familiarizarse con los conceptos de Línea de Producto Software, Árbol de Características (con y sin características clonables) y de Ingeniería Dirigida por Modelos en general, y de Ingeniería de Lenguajes Dirigida por Modelos en particular.

2. Estudio de las herramientas utilizadas. El objetivo de esta fase comprende la familiarización con todas las herramientas y tecnologías necesarias para desarrollar la parte estipulada de la aplicación. En concreto, con EMF, Ecore, EMFText, Eclipse Validation Framework, Eclipse Plugin Development y Subversion.

3. Desarrollo de un editor de restricciones externas entre características. El objetivo de este editor es soportar la especificación de restricciones externas ante un modelo de características proporcionado por el usuario. Tales restricciones son expresiones similares a fórmulas lógicas, salvo por alguna peculiaridad específica. Es por eso que se optó por el uso de un editor textual en lugar de uno gráfico, ya que es el método más habitual de representar este tipo de operaciones. Para crear el metamodelo del lenguaje se ha utilizado la herramienta Ecore, mientras que para definir la gramática se ha utilizado EMFText.

4. Desarrollo de un validador de configuraciones. Una vez se finalizó de crear el editor para las restricciones, el siguiente paso lógico era aportarle una semántica que permitiera comprobar si las restricciones creadas satisfacen la configuración proporcionada por el usuario. Para implementar la semántica se utilizaron las herramientas EMF, Eclipse Validation Framework y Eclipse Plugin Development.

5. Validación y pruebas. Con objeto de evaluar, probar y verificar el correcto funcionamiento de nuestra herramienta se han sometido algunas configuraciones del árbol de caracterísitcas Smarthome a una serie de pruebas de caja negra, tratando de probar todas las operaciones de restricciones posibles en todos los contextos problemáticos y habituales.

1.3. Estructura del Documento

Tras este capítulo de introducción, la memoria se estructura tal y como se describe a continuación: El capítulo 2 introduce un poco más en profundidad los conceptos implicados en la herramienta Hydra, así como las herramientas más determinantes a la hora de llevar a cabo su implementación. El capítulo 3 describe 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.

Capítulo 2

Antecedentes

Este capítulo trata de describir a grandes rasgos las técnicas, tecnologías y herramientas utilizadas para la creación de nuestro entorno de especificación y validación de restricciones. El capítulo comienza describiendo más en detalle lo que es la Ingeniería de Lenguajes Dirigida Por Modelos, para a continuación presentar las dos principales herramientas de modelación que han sido utilizadas: Ecore y EMFText. Más adelante se hablará también en detalle de los árboles de características, y por último se describirá brevemente el entorno de desarrollos de plugins de Eclipse que se utilizó para implementar diversas funciones de nuestro editor.

Contents

2.1. Ingeniería de Lenguajes Dirigida por Modelos

La Ingeniería de Lenguajes Dirigida por Modelos no es más que un caso concreto de la más genérica Ingeniería Dirigida por Modelos (o Model-Driven Architecture o MDA) aplicado desde el punto de vista de la Teoría de Lenguajes Formales, con lo cual es conveniente explicar en qué consiste MDA y comentar los añadidos que introduce el enfoque gramático.

La Inginiería Dirigida por Modelos intenta definir la funcionalidad de el sistema que pretendemos crear a través de la creación de uno o varios metamodelos que representen todas las características de nuestro sistema y todas las operaciones que puede llevar a cabo. El principal objetivo de la

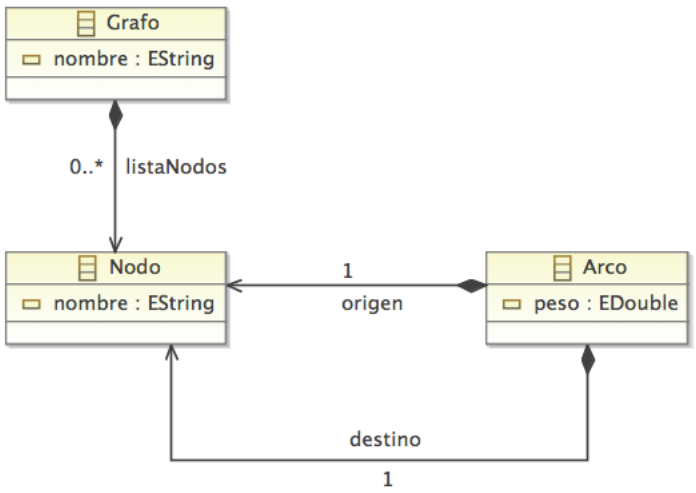

Figura 2.1: Metamodelo (sintaxis abstracta) de un creador de grafos

Ingeniería Dirigida por Modelos es elevar el nivel de abstracción aún más, situándolo por encima del límite establecido por los Lenguajes de Alto Nivel.

El nivel de abstracción y complejidad de nuestros metamodelos variará dependiendo de la cantidad de ellos que incorporemos al sistema. De este modo, un metamodelo que represente todo el sistema directamente será más difícil de entender a primera vista que varios metamodelos que implementen cada uno un tipo de operación o funcionalidad del sistema.

La transformación de esos modelos a código permite la automatización de tareas que pueden resultar triviales y/o repetitivas al programador, en las cuales de otro modo se invierte mucho tiempo de programación y de detección y depuración de errores.

Una vez que tengamos los metamodelos necesarios para definir el comportamiento de nuestro sistema, podremos instanciarlos para crear modelos que representen sistemas concretos. Una instancia de un metamodelo es un modelo que cumple todos los requisitos marcados por su metamodelo, y que da valor a los parámetros del mismo (por ejemplo a sus atributos).

La figura 2.1 es un ejemplo de un metamodelo sencillo que represente un constructor de grafos unidireccionales con pesos. O dicho de otro modo, representa la sintaxis abstracta de nuestro sistema. Se denomina modelo de sintaxis abstracta a cualquier metamodelo que represente los conceptos y el comportamiento del sistema, y cuyas instancias representen elementos reconocibles dentro de ese sistema.

En el ejemplo presentado, el metamodelo presenta la sintaxis abstracta de nuestro sisstema porque cualquier instanciación válida del mismo constituye un grafo completo. Dentro del contexto particular de la Ingeniería de

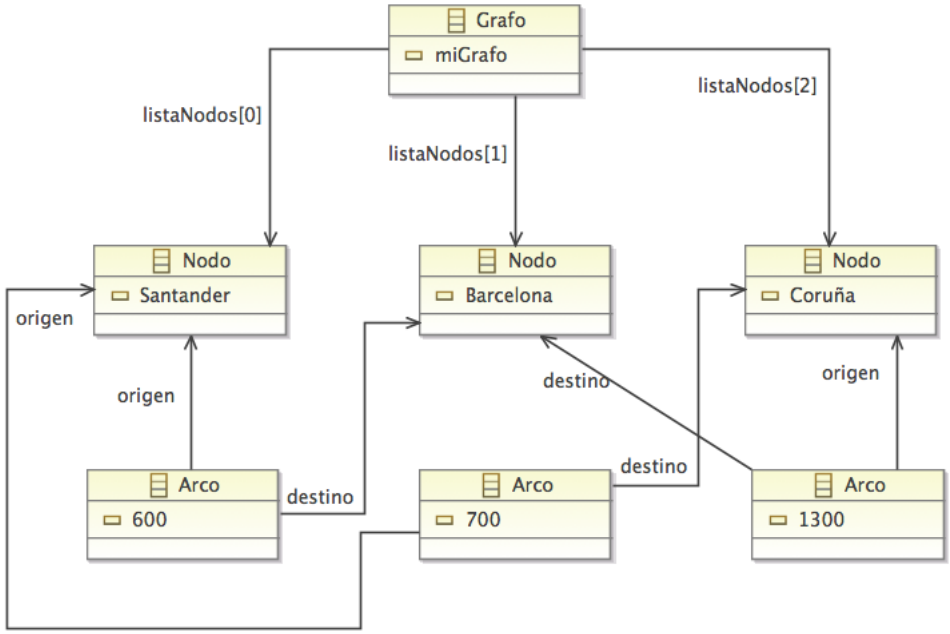

Figura 2.2: Instancia del metamodelo (sintaxis concreta) que representa un grafo específico

Lenguajes Dirigida por Modelos, la sintaxis abstracta representa cualquier conjunto de líneas de código válidas que pueden construirse a partir de ella, y posteriormente ser ejecutadas.

Explicando el metamodelo de nuestro ejemplo un poco más en detalle, podemos decir que la clase Grafo representa el conjunto final de un grafo construido, y puede tener varios Nodos. Esos nodos, a su vez, pueden estar conectados por Arcos. Cada arco tiene un origen, un destino y un peso. Cualquier instanciación de este metamodelo representa un grafo completo.

La figura 2.2 muestra un ejemplo de instanciación del metamodelo anterior. Esta instancia representa un grafo sencillo que muestra las distancias entre algunas ciudades. O dicho de otro modo, es una de las múltiples sintaxis concretas que podemos construir a partir de nuestra sintaxis abstracta.

Dentro del marco de la Ingeniería de Lenguajes Dirigida por Modelos, una sintaxis concreta es cualquier conjunto de expresiones válidas que hayan sido generadas con nuestra sintaxis abstracta. Pero no sólo eso, el hecho de que estemos trabajando con lenguajes conlleva irrevocablemente el tener que desarrollar una gramática de producciones para poder construir nuestras líneas de código en el orden apropiado y con los símbolos apropiados. Eso sí, la gramática será más sencilla y comprensible que la que habría que construir de no estar usando este tipo de metodología. Esta tarea también queda englobada en la sintaxis concreta. De este modo, la sintaxis concreta se suele clasificar en sintaxis concreta visual (el modelo, expresado mediante

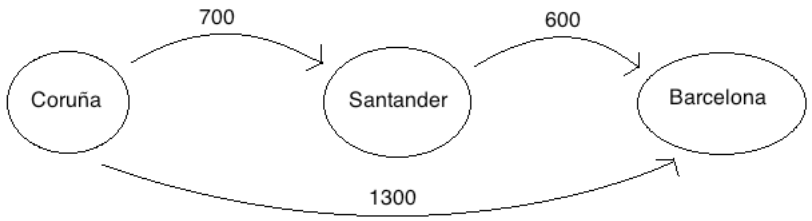

Figura 2.3: Sintaxis visual del grafo de la figura 2.2

líneas de código en el caso de lenguajes, o un grafo pintado en el caso del ejemplo mostrado) y sintaxis concreta textual (la gramática del lenguaje).

La figura 2.3 muestra una sintaxis concreta visual del grafo de la figura 2.2.

Una vez que hemos completado estos pasos, sólamente queda dotar al sistema de una semántica, es decir, de un comportamiento en ejecución. En el ejemplo de nuestro generador de grafos podríamos crear una semántica para calcular distancias mínimas entre caminos. En el caso concreto de la Ingeniería de Lenguajes Dirigida por Modelos esta semántica representa el comportamiento de las líneas de código cuando son ejecutadas. Por poner un ejemplo sencillo, es la encargada de que la instrucción "4 <5çompruebe si efectivamente el 4 es menor que el 5.

Una vez se han explicado las bases de la Ingeniería de Lenguajes Dirigida por Modelos vamos a proceder a mostrar y explicar muy brevemente el metamodelo usado para generar las estructuras válidas de código para el lenguaje de especificación y validación de restricciones que hemos creado en este PFC. Ese metamodelo corresponde a la figura 2.4

La clase que engloba todo el conjunto resultante es Model. Un modelo puede tener varias restricciones, y estas a su vez pueden tener varias operaciones booleanas (ya que una restricción siempre tiene que evaluarse a true o false). Esas operaciones booleanas se dividen en varios tipos: unarias (negación) , binarias (and, or, etc.), de comparación (mayor que, menor que, etc.). o de selección (all, any) Los operandos pueden ser otras operaciones o características (en inglés Features). Más adelante se explicará en detalle la sintaxis del lenguaje creado.

2.2. EMF, Ecore y EMF Validation Framework

EMF o Eclipse Modeling Framework es un plug-in para eclipse que permite trabajar con la metodolía de Ingeniería Dirigida por Modelos. Tiene incorporado varios generadores de código que permiten, entre otra multitud de funciones, crear automáticamente la implementación de los modelos que creemos, así como su integración inmediata como plug-in con el entorno

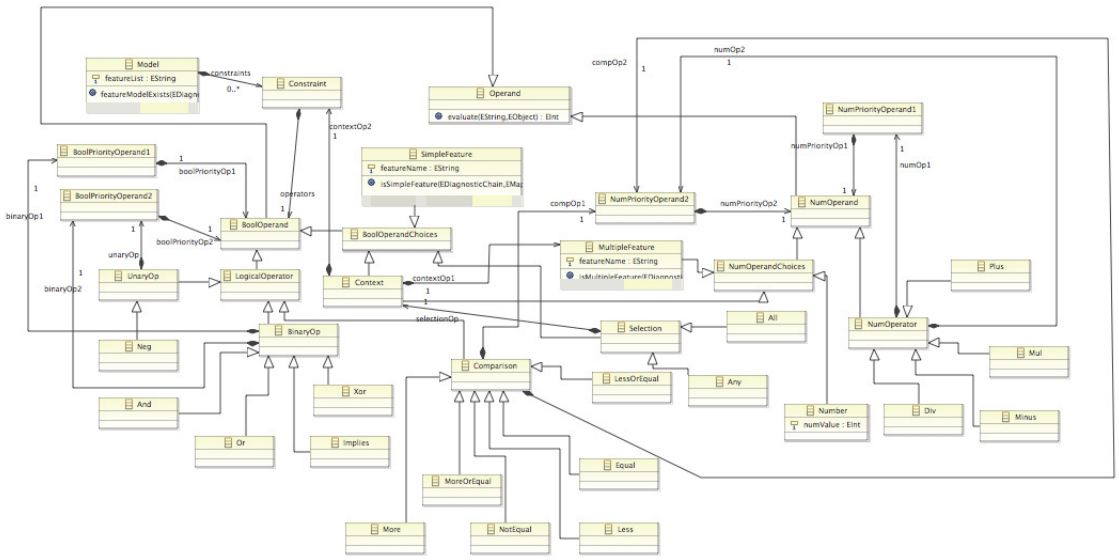

Figura 2.4: Metamodelo utilizado para la creación de nuestro lenguaje de especificación y validación de restricciones

eclipse. Además, es capaz de integrarse con multitud de herramientas orientadas a tareas mucho más específicas dentro de la Ingeniería Dirigida por Modelos, de las cuales cabe destacar Ecore.

Ecore es la herramienta que permite la creación y edición de metamodelos, gracias a un entorno visual bastante atractivo que facilita enormemente el proceso. El fichero resultante de nuestra creación presentará automáticamente un formato estándar XMI, que es el utilizado para todo tipo de modelos y sus instancias. Ecore posee otro tipo de funcionalidades menos utilizadas que se engloban dentro del paquete Ecore Tools, enfocadas todas ellas al uso de los metamodelos y su validación.

Por último, EMF también incorpora una herramienta integrada con Ecore para la validación de las múltiples sintaxis concretas que podamos construir. EMF Validation Framework sirve, en otras palabras, como soporte en caso de que nuestro lenguaje pueda contener reglas adicionales que no puedan ser satisfechas únicamente con la descripción del metamodelo y la gramática. En el caso particular de nuestro editor para escificación y validación de restricciones hemos utilizado EMF Validation Framework para comprobar si las características escritas por el usuario existen en el modelo importado, y también para determinar si tienen cardinalidad o no.

2.3. EMFText

EMFText es una herramienta específicamente diseñada para diseñar las gramáticas de los lenguajes que hayan sido diseñados previamente con un

```
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 2.5: Trozo de la gramática de nuestro editor de especificación y validación de restricciones

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 2.5 se muestra una pequeña captura que contiene una porción de la gramática construida para nuestro editor de especificación y validación de restricciones.

2.4. Líneas de producto software y Árboles de características

El objetivo de las líneas de productos software es crear la infraestructura para la rápida producción de sistemas software para un segmento de mercado específico, donde estos sistemas software son similares, y aunque comparten un subconjunto de características comunes, también presentan variaciones entre ellos **?? ?? ?? ??**.

El principal logro en las líneas de productos software es, construir productos específicos lo más automáticamente posible a partir de un conjunto de elecciones y decisiones adoptadas sobre un modelo común, conocido como modelo de referencia, que representa la familia completa de productos que la línea de productos software cubre.

El desarrollo de líneas de producto software se compone de dos procesos de desarrollo software diferentes pero íntimamente relacionados, conocidos como ingeniería del dominio e ingeniería de la aplicación.

En el nivel de ingeniería del dominio, comenzamos por los documentos de requisitos que describen una familia de productos similares para un segmento de mercado específico. Entonces, diseñamos una arquitectura e implementación de referencia para esta familia de productos. Esta arquitectura de referencia contiene los elementos que son comunes para todos los productos de la familia.

En el nivel de ingeniería de la aplicación, comenzamos un documento de requisitos de un producto específico. Este documento estable las variaciones específicas que deben ser incluidas en este producto concreto. Con esta información, introducimos los cambios en la arquitectura y en la implementación de referencia, y se debería obtener como resultado un producto software único.

Para ser capaces de completar con éxito la tarea correspondiente a ingeniería del dominio, una de las cuestiones clave es establecer una forma de especificar los productos software que una línea de productos es capaz de productor, y aquí es donde entran en juego los Árboles de características o Modelos de Características. Los productos de una línea de productos software se diferencian por sus características, siendo una característica un incremento en la funcionalidad del producto, o más formalmente, ünа característica es una propiedad de un sistema que es relevante a algunos stakeholders y es usada 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 características comunes entre los distintos productos, los modelos de características organizan el conjunto de características jerárquicamente mediante las siguientes relaciones entre ellos: (1) relación entre una o varias características padre y un conjunto de carac-

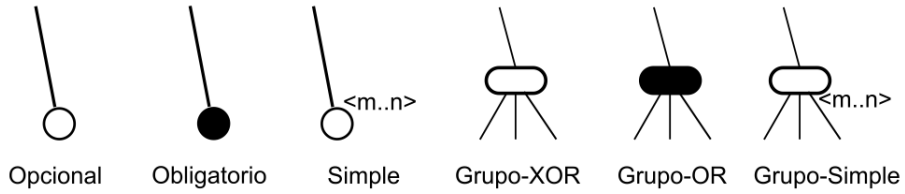

Figura 2.6: Distintos tipos de relaciones en un modelo de características

terísticas hijas o subcaracterísticas y (2) relaciones no jerárquicas del tipo "si la característica A aparece, entonces la B se debe excluir".

Por otro lado, un modelo de características debe poder representar la cardinalidad de las características, por motivos tanto de comprensión (es mucho mejor contar con un árbol de 8 nodos que con uno de 100, teniendo ambos un significado equivalente), como de funcionalidad, ya que permite expresar ciertas restricciones que de no contar con la cardinalidad no podrían expresarse.

Las posibles relaciones que pueden darse en un árbol de características (mostradas gráficamente en la figura 2.6) son las siguientes:

1 - Opcional: La característica hija puede estar o no estar seleccionada

2 - Obligatoria: La característica es requerida.

3 - Simple: 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.

4 - Grupo-xor: Sólo una de las características pertenecientes al grupo será seleccionada.

5 - Grupo-or: Podremos seleccionar como mínimo una de las subcaracterísticas, y como máximo todas.

6 - Grupo simple: El número de características seleccionadas del grupo vendrá dado por su cardinalidad <m,n>.

Además se podrán disponer de restricciones de usuario más complejas, que son las que se han implementado en el editor de especificación y validación de restricciones desarrollado en este proyecto.

La figura 2.7 muestra un ejemplo de modelo de características. En este caso se trata de un modelo de una casa inteligente o SmartHome, a través del cual, seleccionando ciertas características u otras podremos construir qué tipo de casa queremos. Cada una de las múltiples casas diferentes que podamos construir es lo que se denonima una especialización o configuración de nuestro modelo de características.

El proceso de crear una configuración a partir de un modelo de características se conoce como proceso de configuración o proceso de especialización.

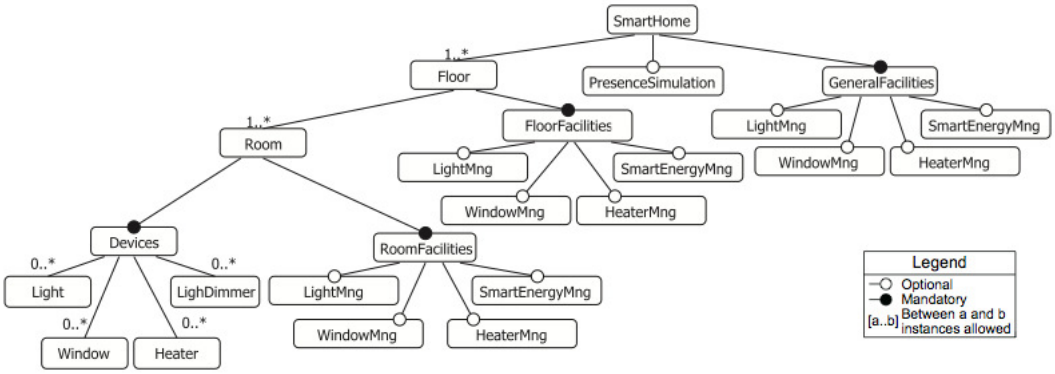

Figura 2.7: Árbol de características para crear una SmartHome

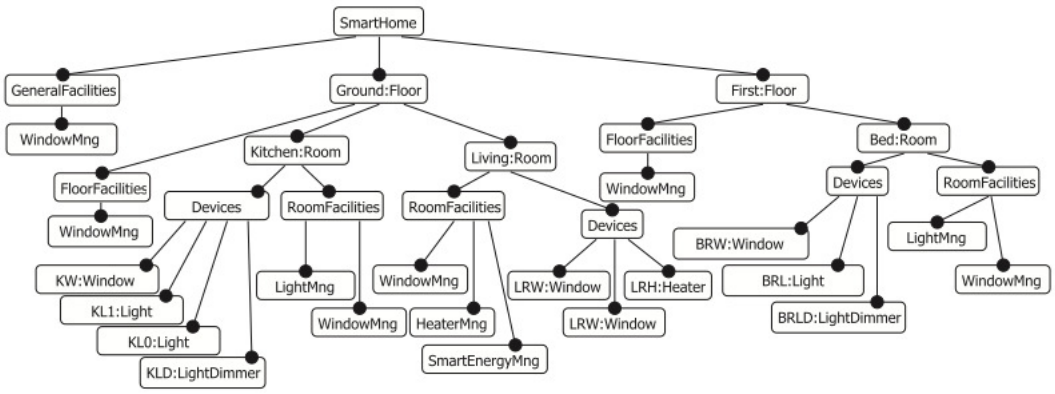

Figura 2.8: Especialización del modelo de la figura 2.7, que representa una de las posibles casas que se pueden construir

Consiste en transformar un modelo de características de tal forma que el modelo resultante sea un subconjunto de las posibles configuraciones denotadas por el primer modelo. La figura 2.8 muestra una posible configuración para el modelo de características de la figura 2.7.

La relación entre un diagrama de características y una configuración es análoga a la existente entre una clase y una de sus instancias en programación orientada a objetos.

2.5. 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.

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.

Capítulo 3

Planificación

Este capítulo trata de describir la planificación que se siguió a la hora de abordar este proyecto: metodología utilizada, enumeración de las diversas tareas implicadas así como el orden en que fueron realizadas y el tiempo que fue necesario invertir para llevarlas a buen puerto.

Contents

3.1. Planificación del proyecto

Como en todo proyecto de este tipo, existe un primer paso de documentación y aprendizaje de los diversos conceptos implicados en el mismo. En mi caso, esta tarea conllevó la familiarización con las Líneas de Producto Software, los Árboles de Características, la Ingeniería de Lenguajes Dirigida por Modelos y la situación en que se hallaba en ese momento la herramienta Hydra. Una vez culminado este proceso de adquisición de información, que duró aproximadamente 3 meses, pudimos iniciar la planificación de las tareas a realizar en el proyecto, tal y como se detalla en la figura 3.1.

La tarea 2, definición de la sintaxis abstracta, comprende la captura de requisitos del lenguaje que hemos de desarrollar (para así poder crear el metamodelo de la manera adecuada), diseño del metamodelo y pruebas de que funciona correctamente. El desarrollo de esta tarea se prolongó durante aproximadamente 3 meses.

La tarea 3, definición de la sintaxis concreta, comprende un nuevo aprendizaje, en este caso el de la herramienta para creación de gramáticas para metamodelos llamada EMFText. Después hubo que hacer una nueva captura de requisitos, menos profunda que la anterior, para poder construir adecuadamente la gramática. La construcción de la misma tuvo como consecuencia sucesivas pruebas y cambios en el metamodelo hasta dejarlo terminado. Esta tarea tuvo una duración aproximada de 5 meses.

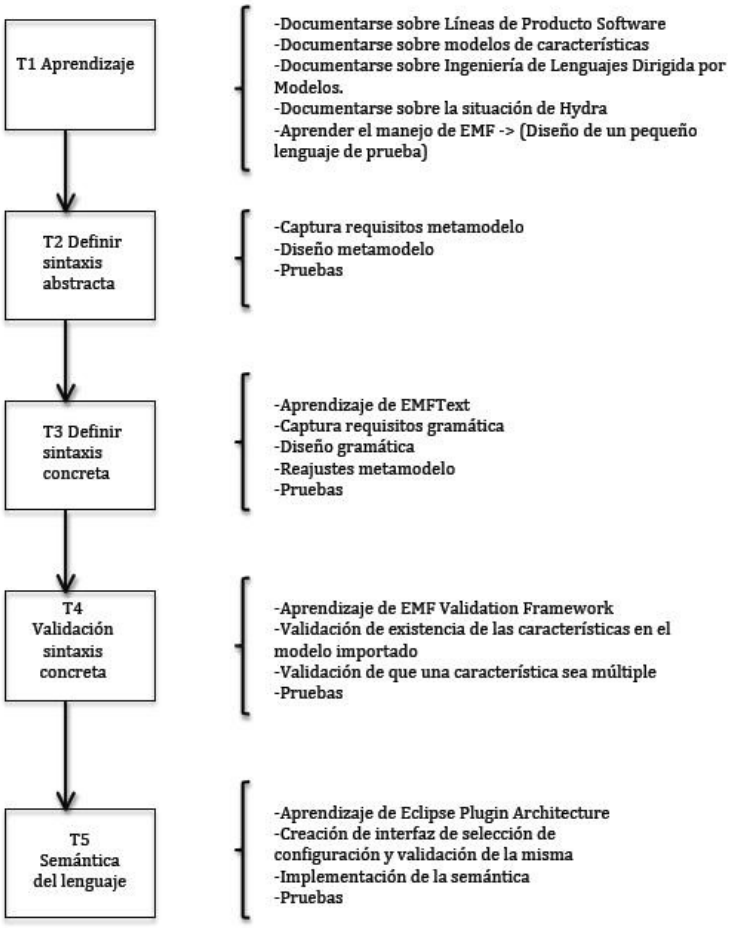

Figura 3.1: Tareas realizadas en este proyecto de fin de carrera

La tarea 4, validación de sintaxis abstracta, comienza con el aprendizaje de una nueva herramienta, el EMF Validation Framework. Tras ello, se construyen los mecanismos necesarios para poder validar que las características a las que queremos aplicar las restricciones existen en el modelo importado, y también validar que una característica parseada como múltiple (con cardinalidad mayor que 1) sea en efecto múltiple en el modelo importado. Esta tarea tuvo una duración aproximada de 2 meses.

La tarea 5, creación de la semántica del lenguaje, comprende la creación de los mecanismos para que las restricciones puedan ser validadas. Es decir, implementar el código que evalua si son ciertas o no, e implementar la interfaz que permite cargar una configuración del modelo. Esta tarea tuvo una duración aproximada de 2 meses.

Todas estas tareas serán explicadas más en detalle en capítulos sucesivos.

La metodología de desarrollo de este proyecto vino impuesta por la téc-

nica de Ingeniería de Lenguajes Dirigida por Modelos. Es decir, no se pudo aplicar ninguna de las técnicas clásicas como "metodología incremental", pues las peculiares características de la ingeniería de modelos impiden que eso sea viable.

Capítulo 4

Creación de la sintaxis abstracta

A partir de aquí, los siguientes capítulos tratan de describir en detalle cada una de las tareas expuestas en la planificación del proyecto. Evidentemente, se ha obviado la tarea de documentación, pues no requiere explicación alguna. Este capítulo en concreto versa sobre la creación de la sintaxis abstracta del lenguaje, así como cada una de las subtareas subyacentes.

Contents

4.1. Captura de requisitos

El diseño de la sintaxis abstracta es el primer paso en cualquier creación de un lenguaje mediante la técnica de Ingeniería de Lenguajes Dirigida por Modelos. Y por lo tanto, la tarea de captura de requisitos en este punto pasa por comprender qué es lo que tiene que hacer exactamante el lenguaje que se pretende crear.

Nuestro lenguaje tiene que permitir, (1), importar un modelo de características sobre el cual se aplicarán las consiguientes restricciones. (2), tiene que permitir definir un numero indeterminado de restricciones sobre ese modelo, y aplicarlas a cualquier configuración del mismo que cargue el usuario. (3), tiene que permitir que una serie de operaciones sean definidas dentro de las restricciones. Y (4), como es lógico, tiene que valorar si las restricciones definidas se cumplen dentro del modelo.

De entre todos esos requisitos básicos, es necesario entrar en detalle en el número 3 y enumerar la lista de operaciones que pueden ser definidas por

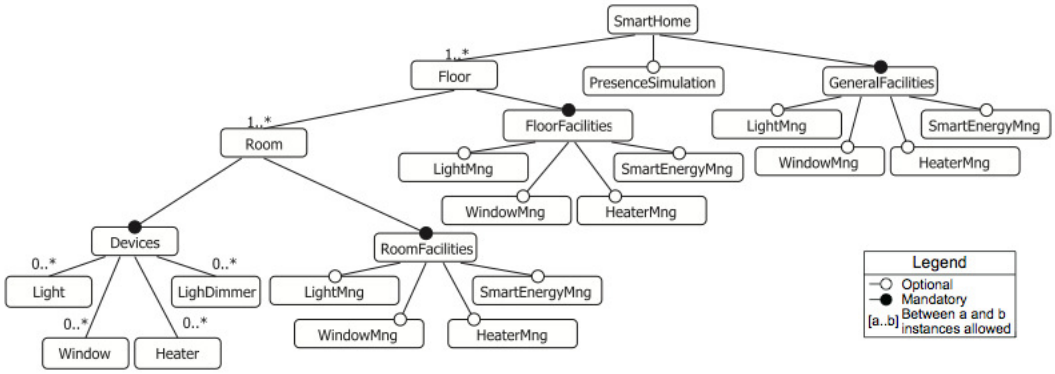

Figura 4.1: Modelo de características de una casa inteligente o SmartHome

nuestro lenguaje. Se pueden clasificar en los siguientes tipos:

- Lógicas: Son operaciones cuyos operandos han de ser características sin cardinalidad (también llamadas características simples), y que se evaluan a verdadero o falso. Entre las operaciones lógicas encontramos las clásicas not, and, or, xor e implica.

- Numéricas: Sus operandos han de ser características con cardinalidad (también llamadas características múltiples) o simplemente números. Su resultado se evalua con un valor numérico. Las operaciones numéricas a implementar son la suma, resta, multiplicación y división.

- Comparativas: Sus operandos han de ser características múltiples o simplemente números, pero su resultado se evalua con un valor booleano. Las operaciones de comparación a implementar son igual que, mayor que, menor que, distinto que, mayor o igual que y menor o igual que.

- Operación de contexto: Operación que permite hacer referencia a una característica hija de otra característica. Esta operación tiene sentido para seleccionar características cuyo nombre pueda estar repetido pero que tengan contextos diferentes. Por ejemplo, en el modelo de características SmartHome de la figura 4.1 podemos observar que la característica HeaterMng está presente en muchos contextos diferentes. Esta operación es necesaria para poder saber con seguridad a cual de esos contextos estamos aplicando la restricción.

- Operación de selección: Operación que corresponde a los operadores lógicos clásicos "para todo.º .existe", y que tiene la misma funcionalidad. Evalua si una restricción se cumple para todos los casos en que puede existir o si se cumple en alguno de los casos. Por ejemplo, en el modelo de la figura 4.1 se podría evaluar una restricción para cada una de las habitaciones que hayan sido definidas, y saber si se cumple en todas, en alguna o en ninguna.

4.2. Creación del metamodelo

Una vez hemos definido cuales son los requisitos del lenguaje, tenemos que construir nuestro metamodelo de tal modo que se ajuste a ellos y sea capaz de cumplirlos. El requisito más complejo y que requerirá más desafíos de diseño es, evidentemente, el de implementar todas las operaciones. Así que empezaremos primero por las partes más fáciles.

Todas las restricciones que sean definidas en cada sintaxis concreta han de ser aplicadas al mismo modelo de características (aunque a varias configuraciones si así lo desea el usuario). Por lo tanto, la información relativa al modelo de características sobre el que queremos aplicar esas restricciones es susceptible de ser almacenada en el metamodelo de nuestro lenguaje. Se entiende que el modelo de características que hay que guardar habrá sido previamente creado usando la herramienta Hydra, pues este editor es una extensión de la misma. Eso reduce mucho el número de factores de los que hay preocuparse, viéndose reducidos en este punto a tener que almacenar únicamente la localización del fichero dentro del sistema, para luego poder cargarlo en partes de validación posteriores.

Así pues, nuestra clase inicial Model, que representa el modelo sobre el que aplicaremos las restricciones, tiene un atributo featureList en el que se guardará la dirección del fichero del modelo de características Hydra.

Para permitir que sobre ese modelo que ya hemos representado se puedan definir un número indeterminado de restricciones, es necesario crear una clase Constraint, y relacionar la clase Model con ella. La relación resultante se podría expresar como ün modelo tiene de 0 a x restricciones", donde x es cualqueir número entero positivo.

El requisito correspondiente a la validación de las restricciones que hayamos definido no tiene modo de ser resuelto a estas alturas del desarrollo de nuestra aplicación, por lo que únicamente queda la implementación de las operaciones. Lógicamente, esta es la tarea de mayor complejidad de nuestro sistema.

El primer paso es definir toda la estructura necesaria para la implementación de las operaciones, haciendo que cada una de ellas esté representada en nuestro metamodelo mediante una clase, pero sin preocuparnos todavía por las relaciones entre ellas. La clase raíz de toda esta estructura es Operand. Es una clase abstracta, es decir, en los modelos que luego instanciemos de este metamodelo no podrá haber ninguna instancia de Operand, sólo de los hijos no abstractos que tenga. A medida que vayamos definiendo clases hijas de Operand estaremos especificando cada vez con más exactitud a qué tipo de operación estamos haciendo referencia.

En el segundo nivel de la estructura de implementación de las operaciones hacemos una ramificación según el tipo del valor de retorno o de evaluación de las posibilidades. Es decir, a la clase Operand le añadiremos dos hijos: BoolOperand para operaciones que se evalúan a booleano y NumOperand

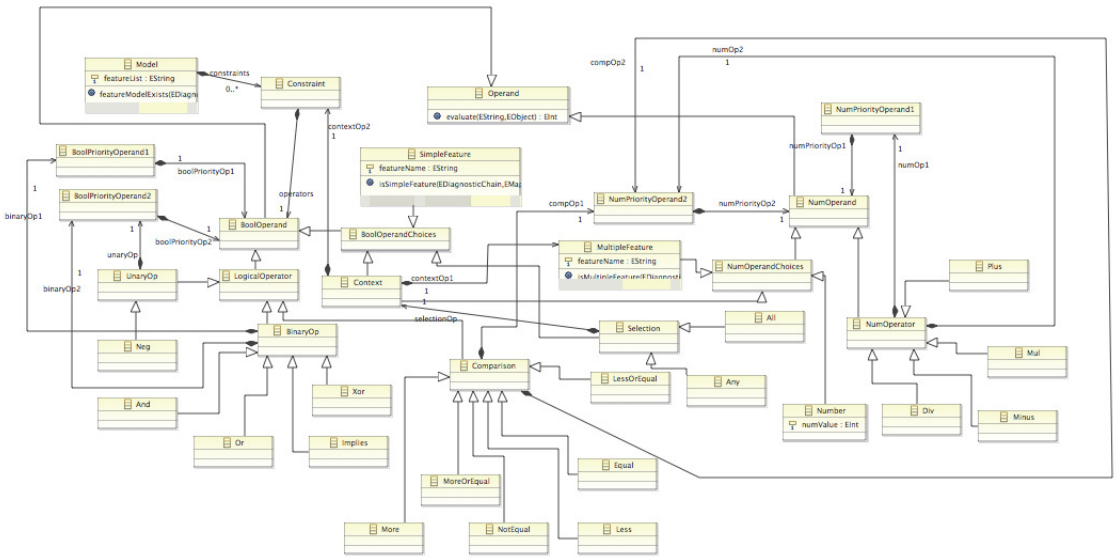

Figura 4.2: Metamodelo utilizado para la creación de nuestro lenguaje de especificación y validación de restricciones

para operaciones que se evalúan a numérico. Estas clases también serán abstractas.

El proceso de división a partir de aquí es más o menos análogo para todas las operaciones, así que vamos a centrarnos únicamente en la rama que da lugar a las operaciones binarias lógicas, para comentar después los casos y situaciones especiales. Una vez tenemos la clase BoolOperand, podemos especializarla un poco más a LogicalOperator, que a su vez se dividirá en operaciones unarias, binarias, o de comparación. Todas ellas son clases abstractas. Por fin, la clase BinaryOp heredará las clases de las operaciones propiamente dichas, en este caso And, Or, Implies y Xor. Estas ya podrán ser instanciadas en las sintaxis concretas que creemos.

Cabe hacer mención también a las clases SimpleFeature, MultipleFeature y Number, que representan a las características simples, múltiples y números respectivamente. En cualquier árbol resultante de parsear nuestro lenguaje, estas clases representarán las hojas. En última instancia todas las operaciones tendrán como operandos características o números. Podemos observar que SimpleFeature es un operando booleano (está en la parte estructural de las operaciones booleanas) ya que su evaluación será verdadero o falso, dependiendo si esa característica ha sido seleccionada en la configuración correspondiente o no. MultipleFeature sin embargo se evalúa a número entero. Su valor será el número de apariciones de esa característica dentro de la configuración correspondiente.

Muchas de las clases que ahora se pueden contemplar en el metamodelo de la figura 4.2 aún no estaban presentes en esta etapa temprana del diseño, y

su inclusión fue necesaria a raíz de la creación de la gramática y los problemas que se observaron en ese punto. En particular, las terminadas en Choices y en PriorityOperand. Las operaciones All, Any y Context en este momento eran simples herencias de BoolOperand. El motivo de estas modificaciones será explicado en el capítulo siguiente.

Para terminar este apartado, vamos a hablar de las relaciones entre las diferentes clases de nuestro metamodelo. En este punto del diseño no eran las mismas que las de la figura 4.2 por los motivos explicados anteriormente. Simplemente buscábamos una forma de relacionar cada operación con los tipos de sus operandos (que también pueden ser operaciones, como es lógico). Las operaciones lógicas binarias tendrán dos operandos que también serán binarios. En este momento del diseño binaryOp1 y binaryOp2 iban relacionados a BoolOperand, al igual que unaryOp. Del mismo modo, compOp1, compOp2, numOp1 y numOp2 (es decir, los operandos de operaciones de comparación y numéricas respectivamente) estaban relacionados con la clase NumOperand.

La relación de toda estructura de operaciones con los dos elementos anteriores, Model y Constraint, se realiza entre Constraint y BoolOperand. Toda restricción en última ha de ser evaluada a verdadero o falso, es por eso que la relación no va con Operand, como podría pensarse en primera instancia. De este modo estamos forzando que la operación con menos profundidad del árbol parseado de nuestra restricción sea booleana, y que por lo tanto el resultado final de validar la restricción sea un dato booleano.

Quizás a alguien le pueda sorprender el hecho de que la relación "operators" entre Constrain y BoolOperand sea 1..1 y no 1..*. El motivo es que como los operadores de esa primera operación booleana que estamos forzando pueden ser a su vez operaciones, la complejidad en la restricción que podemos definir se propaga por ahí en lugar de por la relación creada.

4.3. Pruebas del metamodelo

Las pruebas en este punto del diseño no fueron especialmente fructíferas, ya que la creación de la gramática para la sintaxis textual de nuestro lenguaje motivó numerosos cambios en un metamodelo que en este punto parecía totalmente correcto. Las pruebas consistieron en la creación de varias instancias del metamodelo y observar su árbol de creación, comprobando si este se correspondía con el de diversas instrucciones que fuimos creando intentando tener en cuenta todos los casos.

No fueron especialmente rigurosas dado que a estas alturas del desarrollo de la aplicación aún falta muchos elementos implicados por crear que sin duda tendrían repercusión en nuestra sintaxis abstracta, como de hecho ocurrió más adelante.

Las instrucciones que fueron puestas a prueba, y que se comprobó que

```
C01 GeneralFacilities[SmartEnergyMng implies (HeaterMng and WindowMng)])
C02 PresenceSimulation implies (Room >= 5) or (Floor > 2)
C03 !PresenceSimulation
C04 any Room[SmartEnergy implies (HeaterMng and WindowMng)]
C05 all Room[LightMng implies (Light >= 0)]
C06 any Room[WindowMng implies (Window < 0)]
C07 all Room[HeaterMng implies (Heater != 0)]
C08 PresenceSimulation implies ((Room[Light] / Room) * 100 >= 25)
```

Figura 4.3: Conjunto de instrucciones que puso a prueba el funcionamiento

en efecto eran parseadas de un modo apropiado, fueron las que se ven en la figura 4.3. Este conjunto de instrucciones servirán como pruebas también en momentos más avanzados del desarrollo.

Capítulo 5

Creación de la gramática

Una vez ha sido definido el metamodelo, el siguiente paso es definir la sintaxis concreta textual de nuestro lenguaje, es decir, los medios que permitan expresarnos en él de modo escrito. De no hacerlo, solo podríamos usar este lenguaje creando instancias del metamodelo, lo cual lógicamente no es ni cómodo ni conveniente. Este capítulo versa sobre la creación de la gramática que permite definir esa sintaxis textual, así como de las repercusiones que su diseño tuvo en la sintaxis abstracta.

Contents

5.1. Captura de requisitos

La captura de requisitos de la gramática pasa por informarse sobre qué tipo de sintaxis textual queremos que tengan nuestras operaciones, lo cual en este caso ya estaba especificado previamente por documentos creados en iteraciones previas del proyecto Hydra. Lo más lógico e inmediato era mantenerse fiel a esa sintaxis, según la cual las operaciones deberían ser expresadas textualmente tal como sigue:

Suma: operando1 + operando2
Resta: operando1 - operando2
Multiplicación: operando1 * operando2
División: operando1 / operando2
And: operando1 and operando2
Or: operando1 or operando2
Xor: operando1 xor operando2
Implica: operando1 implies operando2

Mayor que: operando1 >operando2
Menor que: operando1 <operando2
Igual que: operando1 == operando2
Mayor o igual que: operando1 >= operando2
Menor o igual que: operando1 <= operando2
Distinto que: operando1 != operando2
Contexto: operando1 [operando2]
Para todo: all operando1 [operando2]
Existe: any operando1 [operando2]

Otros aspectos concernientes a la gramática que ha habido que tener en cuenta dentro de la fase de captura de requisitos son los siguientes:

- Ha de permitirse la posibilidad de especificar prioridad en las operaciones, es decir, de poder delimitar las operaciones con paréntesis que denoten el orden de realización de las mismas.
- Todas las representaciones textuales de nuestro lenguaje han de empezar con una línea de carga del modelo de características al que han de aplicarse las restricciones. La sintaxis de este aspecto será ïmport operando", donde operando es la dirección del fichero hydra del modelo dentro del disco duro del sistema.
- Todas las restricciones definidas han de separarse entre ellas mediante el carácter " ; ".

Por supuesto, además de los requisitos aquí expuestos también habrá que tener en cuenta todo lo comentado en el apartado de requisitos del capítulo anterior, ya que también influirán a la hora de tomar decisiones de diseño en la gramática.

5.2. Diseño de la gramatica

Una vez han sido definidas las características que queremos que nuestra sintaxis textual posea, el siguiente paso es diseñar una gramática que se ajuste a ellas.

La parte más trivial e inmediata del diseño de la gramática es la concerniente a la implementación de las operaciones, pues las producciones necesarias simplemente requieren la inclusión de los operandos involucrados y los caracteres que deseemos que definan la operación. La figura 5.1 muestra la implementación de estas operaciones.

Sí que cabe comentar con respecto a las operaciones las últimas líneas, que muestran la asignación de valor a las hojas de nuestros árboles parseados. En esas líneas estamos indicando que los atributos de las instancias de clase

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

// Operaciones numericas
Plus ::= numOp1 "+" numOp2;
Minus ::= numOp1 "-" numOp2;
Mul ::= numOp1 "*" numOp2;
Div ::= numOp1 "/" numOp2;

// Operaciones de contexto
Context ::= contextOp1 "[" contextOp2 "]";

// Operaciones de seleccion
All ::= "all" selectionOp;
Any ::= "any" selectionOp;

// Operaciones de comparacion
More ::= compOp1 ">" compOp2;
MoreOrEqual ::= compOp1 ">=" compOp2;
Less ::= compOp1 "<" compOp2;
LessOrEqual ::= compOp1 "<=" compOp2;
NotEqual ::= compOp1 "!=" compOp2;
Equal ::=  compOp1 "==" compOp2;

// Operaciones primitivas
SimpleFeature ::= featureName[TEXT];
MultipleFeature ::= featureName[TEXT];
Number ::= numValue[DIGIT];
```

Figura 5.1: Implementación de las operaciones de nuestro editor com EMFText

Number van a ser números, y que los atributos de las instancias de las clases SimpleFeature y MultipleFeature van a ser palabras.

La parte más complicada corresponde a la implementación del inicio de la gramática y de las producciones que conducen a la misma. Pero antes de mostrar la figura con esta parte de la gramática conviene explicar el problema que llevó a realizar los cambios en el metamodelo mencionados en el capítulo anterior. Este problema surgió a la hora de implementar las operaciones con prioridad, es decir, la inclusión de los paréntesis.

El inconveniente es que el tipo de gramática LL que implementa EMFText hacía imposible tomar una decisión sobre hacia qué elemento seguir parseando en caso de encontrarnos con un paréntesis. La mejor solución que se nos ocurrió para evitar este problema fue la adición de diversas clases y relaciones auxiliares en el metamodelo, cuya única función es estructural y de apoyo a la gramática. Gracias a ellas y a una mejor definición de las producciones conseguimos evitar esos problemas de parsing y podemos llevar a

```
SYNTAXDEF hydraConst
FOR <http://www.unican.es/personales/sanchezbp/spl/hydra/constraints>
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
```

Figura 5.2: Implementación del inicio de la gramática con EMFText. Con la figura 5.1 se completa la gramática

cabo las operaciones de prioridad con paréntesis.

Las clases añadidas para solventar esta situación fueron las siguientes: BoolPriorityOperand1, BoolPriorityOperand2, NumPriorityOperand1, NumPriorityOperand2, BoolOperandChoices y NumOperandChoices. Las relaciones añadidas fueron boolPriorityOp1, boolPriorityOp2, numPriorityOp1 y numPriorityOp2.

Una situación similar fue la que propició que las operaciones Context, All y Any hayan sido diseñadas tal y como presenta el metamodelo, ya que que la particular sintaxis de estas (diferente a las demás que siguen el mismo esquema de op + char + op) también mostraba ciertos problemas de parsing. En este caso no fue necesario añadir elementos auxiliares, sino simplemente recolocarlos para evitar estos problemas. Con esto ya se han hecho todos los cambios en el metamodelo, que alcanza en este punto su versión final tal como muestra la figura 4.2. Con respecto al metamodelo solamente quedan por comentar los métodos que muestran algunas clases, que serán explicados en los próximos capítulos ya que se usan en el proceso de validación y semántica.

Una vez comentados estos detalles es momento de explicar el inicio de la gramática, que se muestra en la figura 5.2.

En la primera línea y mediante la claúsula SYNTAXDEF indicamos la extensión que queremos que tengan los ficheros escritos en nuestro lenguaje. En nuestro caso nos hemos decantado por la terminación .hydraConst. En la segunda línea y mediante la claúsula FOR se indica la URI del metamodelo.

```
C00 GeneralFacilities[SmartEnergyMng implies (HeaterMng and WindowMng)])
C01 GeneralFacilities[LightMng] implies (all FloorFacilities[LightMng])
C02 GeneralFacilities[WindowMng] implies (all FloorFacilities[WindowMng])
C03 GeneralFacilities[HeaterMng] implies (all FloorFacilities[HeaterMng])
C04 GeneralFacilities[SmartEnergy] implies (all Floor[FloorFacilities[SmartEnergy]])
C05 all FloorFacilities[SmartEnergy implies (HeaterMng and WindowMng)])
C06 all Floor[FloorFacilities[LightMng] implies (all Room[LightMng])]
C07 all Floor[FloorFacilities[WindowMng] implies (all Room[WindowMng])]
C08 all Floor[FloorFacilities[HeaterMng] implies (all Room[HeaterMng])]
C09 all Floor[FloorFacilities[SmartEnergy] implies (all Room[SmartEnergy])]
C10 all Room[SmartEnergy implies (HeaterMng and WindowMng)]
C11 all Room[LightMng implies (Light > 0)]
C12 all Room[WindowMng implies (Window > 0)]
C13 all Room[HeaterMng implies (Heater > 0)]
C14 PresenceSimulation implies ((Room[Light] / Room) * 100 >= 25)
```

Figura 5.3: Batería de instrucciones para probar el funcionamiento de la gramática

Una URI es un formato de dirección interno de Eclipse, que se usa para localizar otros ficheros en el workspace. En la tercera línea, delimitada por la claúsula START, indicamos a la gramática que la clase inicial de nuestro metamodelo (y la que será la raiz en todos los árboles parseados) es Model.

El bloque OPTIONS permite activar algunas opciones de configuración que incluye EMFText. En nuestro caso la única que tiene utilidad es usePredifinedTokens, que permite ahorrarnos la definición del token text. El bloque TOKENS sirve para definir los tokens de nuestra gramática. En nuestro caso usaremos 3: DIGIT para asignar al valor numérico, TEXT para asignar a las características y DIRECCION para asignar la dirección física del modelo de características.

Por último, el bloque RULES permite crear las producciones. Como inicial, tal y como se especificó en los requisitos, exigimos un import y una dirección, que será almacenada en el atributo featureList de la clase Model. En la línea inicial también se indica, mediante una expresión regular, que el número de restricciones a definir puede ser tan grande como se desee y que estas deben acabar con el carácter ";" .

La línea de producción de Constraint diferencia entre operaciones con prioridad y sin ella. Sin el problema comentado de EMFText la gramática podría quedar así, pero para solucionarlo nos vemos obligado a incluir las cuatro líneas siguientes, cuya única función es solventar esa situación. El resto de la gramática continuaría en la figura 5.1 mostrada anteriormente, y ahí terminaría.

5.3. Pruebas

Para hacer las pruebas correspondientes a la gramática fue de mucha utilidad la vista Outline de Eclipse, que permite obsevar el árbol de parsing

de todos los ficheros de código que creemos en nuestro lenguaje.

La batería de pruebas simplemente consistió en comprobar una serie de instrucciones y observar dentro de la vista Outline si se parseaban de modo correcto. En este caso se utilizaron unas restricciones definidas en un documento previo de Hydra que contenían todos los aspectos problemáticos de la gramática, es decir, operaciones largas con prioridad y multitud de contextos. Estas instrucciones son las que se muestran en la figura 5.3.

El resto de operaciones fueron puestas a prueba con restricciones más sencillitas y, como en el caso anterior, fueron exitosas. También se usaron las instrucciones de las pruebas del capítulo anterior, que se pueden observar en la figura 4.3 para comprobar que el árbol era el mismo que se creó en ese momento.

Capítulo 6

Validación de las sintaxis concretas

Este capítulo trata de describir el desarrollo de la tarea de implementación del proceso de validación de las sintaxis concretas. La validación consiste en tratar de averiguar si ciertos aspectos de la sintaxis concreta construida que no se pueden comprobar mediante la gramática y el metamodelo son correctos. Un ejemplo de uno de esos aspectos es si la dirección introducida en el import para el modelo de características es correcta.

Contents

6.1. Captura de requisitos

La captura de requisitos del proceso de validación pasa por detectar qué aspectos de las sintaxis concretas que construyamos son susceptibles de convertirla en errónea, y que además no pueden ser detectados por los mecanismos restrictivos proporcionados por la gramática y el metamodelo. Los requisitos encontrados han sido fundamentalmente los siguientes:

- Comprobación de que la dirección introducida en el import para cargar el modelo de características al que se aplicarán las restricciones es correcta. Ser correcta no significa tanto que esté en un formato de dirección válido (esto se detecta por la gramática, salvo quizás alguna trampa específica puesta a propósito), sino que la dirección especificada contenga un fichero xmi con un modelo de características.

- Comprobación de que las características escritas existan en el modelo importado. Lógicamente, no tendría sentido permitir escribir características que no estén presentes en ese modelo, pues luego a la hora de evaluar las

restricciones en las que estuvieran presentes nunca iban a estar seleccionadas.

- Comprobación de que una característica parseada como simple realmente lo sea. Cuando en una restricción tiene una operación lógica, se fuerza a que sus operandos sean parseados como características simples, ya que son las únicas que pueden evaluarse a 1 ó 0. En caso de poner una característica que realmente es múltiple en una operación lógica provocaría un error en la ejecución, pues al evaluarse podría tener un valor mayor que 1. En las operaciones en que se fuerza que las características sean parseadas a múltiples (como las de comparación) esto no es un problema, ya que las simples se pueden ver como un caso concreto de las múltiples. Por eso, aunque sea parseada a múltiple y en realidad sea simple no tiene importancia y la operación seguirá teniendo sentido.

6.2. Implementación de la validación

Para implementar todo lo relativo a la validación se ha utilizado una herramienta específica para esta funcionalidad que está englobada dentro de la instalación de EMF, cuyo nombre es EMF Validation Framework. Otra opción podría haber sido modificar el postprocesador de nuestro lenguaje ayudándonos de las opciones de modificación que nos proporciona EMFText, pero sería mucho más laborioso y a fin de cuentas el resultado final sería el mismo. Desde la propia documentación de EMFText se recomienda no modificar el postprocesador si la funcionalidad que se quiere añadir se puede implementar directamente desde Validation Framework.

El primer paso para implementar la validación de las sintaxis concretas mediante Validation Framework nos obliga a crear un método en cada clase del metamodelo cuya validación sea necesaria. Ese método tiene que tener una definición concreta: ha recibir dos parámetros (uno llamado "diagnostics"del tipo EDiagnosticChain y otro llamado çontext"que es un mapa) y retornar un valor booleano.

El parámetro "diagnostics.[es] el que usa Validation Framework para determinar el resultado de la validación. En este parámetro se puede guardar información tal como el tipo de error producido o el mensaje de error que queremos que se produzca cuando el error se produzca. En caso de que la validación haya sido satisfactoria no se ha de realizar ninguna tarea adicional con este parámetro. La labor de implementación por nuestra parte consistirá en resumidas cuentas en controlar qué información se mete en este parámetro y en qué casos hay que darle uso.

Como se mencionó en el apartado anterior, había 3 aspectos a validar dentro del marco de Validation Framework:

- Validar que la dirección indicada en el ïmport"sea válida y contenga un modelo de características. Esta validación se lleva a cabo en la clase Model. Para llevar a cabo esta validación simplemente se carga la dirección

indicada y se manejan las posibles excepciones que una dirección errónea pueda generar. Además, se comprueba que el contenido de esa dirección sea un modelo de características mediante el uso de determinados métodos definidos para los modelos. Se aprovecha también para generar una variable global que contenga el modelo leido, para aprovechar así la lectura realizada y que no haya que buscar el modelo cada vez que se quiera hacer uso de él.

- Validar que las características escritas en nuestro fichero de restricciones existan en el modelo de características proporcionado. Esta validación se realiza en la clase MultipleFeature y también en SimpleFeature. Simplemente consiste en buscar el nombre de la característica en concreto (haciendo uso del parámetro featureName) en el modelo cargado anteriormente. Si se encuentra una coincidencia es que la validación ha sido exitosa y por lo tanto no habrá que mostrar ningún mensaje de error.

- Validar que las características parseadas como simples realmente sean simples. Esta validación se lleva a cabo en SimpleFeature, no teniendo sentido su implementación en el caso múltiple. Para llevarla a cabo tenemos que mirar que, una vez comprobada la existencia de la característica, esta no pueda ser instanciada en más de una ocasión. Para ello tenemos que mirar en el modelo importado las relaciones entre las características, y buscar si el límite de cardinalidad es mayor que uno para cualquier relación que implique a la característica que estamos intentando validar. Además, tendremos que comprobar que la característica no sea hija de una característica múltiple. Para ello miramos también las relaciones en que estén implicadas las características padre de la que tenemos intención de validar.

Las pruebas de EMF Validation Framework se hicieron simultáneamente a las pruebas de la semántica, por motivos principalmente de comodidad, así que serán expuestas más adelante.

Capítulo 7

Semántica del lenguaje

Este es el último capítulo que describe tareas implicadas en el desarrollo de la aplicación. En concreto se hablará sobre la creación de la interfaz de nuestra aplicación, el desarrollo de la semántica del lenguaje construido y las pruebas que han servido para poner a prueba el conjunto final de la aplicación.

Contents

7.1. Creación de la interfaz del programa

EMF y EMFText proporcionan una interfaz por defecto para la creación de editores y su posterior uso. Se incluyen algunos aspectos como coloreado de palabras clave y detección instantánea de errores de sintaxis. Es por eso que ya contábamos con gran parte del trabajo hecho, y esta ha sido una de las razones por las que EMFText pareció más conveniente en su momento.

La interfaz del editor desarrollado es parte de la herramienta Eclipse y no puede ejecutarse fuera de su entorno. No tendría sentido hacerlo de otro modo, ya que no solo necesitamos las diversas funciones que EMF y EMFText nos proporcionan, sino que la propia herramienta Hydra es también un plug-in de Eclipse. La figura 7.1 muestra la interfaz del editor en funcionamiento.

Las características del editor creado por defecto por EMF no son suficientes para satisfacer toda la funcionalidad que debe llevar a cabo, por lo que ha sido necesaria una ligera modificación para poder cumplir con los objetivos de nuestra aplicación. Para poder implementar la semántica es necesario que nuestro editor cargue previamente dos modelos: el modelo de características (lo cual ya ha sido implementado) y una configuración de ese modelo, que es sobre el que se validarán las restricciones que definamos.

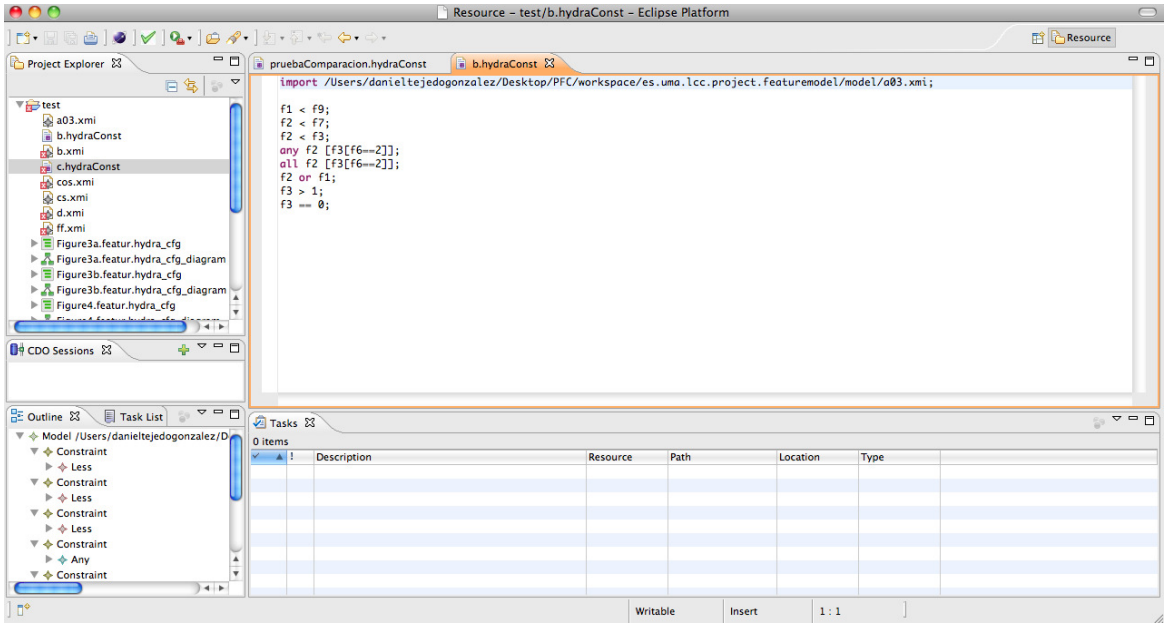

Figura 7.1: Captura de pantalla del editor en funcionamiento

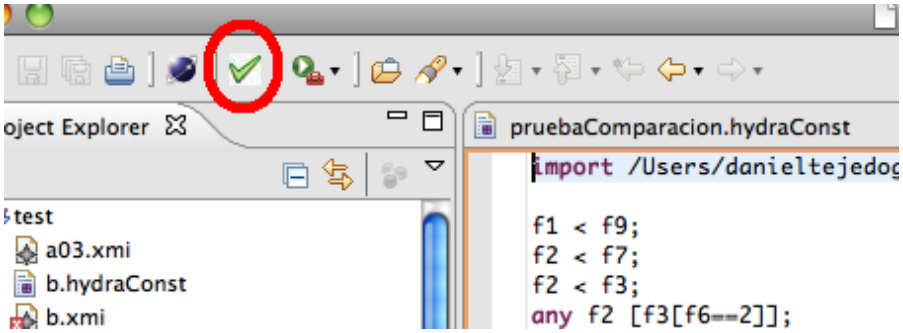

Figura 7.2: Botón que ejecutará la validación de las restricciones

Para permitir que se pueda cargar esa configuración hemos añadido un botón llamado "Validate"que abre una ventana de carga en la que se pide al usuario escoger un fichero de extensión .xmi exclusivamente. Una vez cargado, el manejador del botón ejecutará la validación y mostrará el resultado de la misma en un cuadro de diálogo. Pero esto es tarea de la semántica y será explicado en el apartado siguiente.

Dado que nuestra aplicación no es sino un plug-in de Eclipse, añadir el botón ha requerido informarse de la Arquitectura de Plug-ins de Eclipse. Sin entrar en demasiados detalles, el proceso consiste en añadir lo que se conoce como "punto de extensión.ªl plug-in previo. Un punto de extensión sirve para dotar a un plug-in de la funcionalidad de otros plug-ins e incorporarla a ellos.

En este caso nuestro punto de extensión corresponde a uno proporcionado por la propia arquitectura, cuya utilidad es precisamente la de añadir botones al menú de herramientas de nuestra aplicación.

7.2. Implementación de la semántica del lenguaje

Implementar la semántica del lenguaje es el último paso (sin contar las pruebas) para dar por concluido el desarrollo de nuestro editor. La semántica es la que permite que las acciones que se definen en las líneas de código escritas en el editor sean ejecutadas, luego es una parte bastante importante de todo el proceso.

En el apartado anterior indicamos que el botón de Validate era el que, además de cargar la configuración pertinente, iniciaba la validación de las restricciones definidas. Por lo tanto, el inicio de la implementación de la semántica estará contenido en el marco del manejador del botón Validate.

Desde este manejador cargamos todas las restricciones definidas, e iniciamos el proceso de validación evaluando cada restricción una a una. Usaremos el resultado obtenido para constuir un cuadro de diálogo en el que mostraremos el resultado de validar cada una de las restricciones definidas en la configuración previamente seleccionada.

Para realizar la validación se utiliza el método .evaluate"de la clase Operand. Este método recibe la configuración sobre la que hay que realizar la evaluación, y una característica que se usará como contexto. Al estar en la clase abstracta Operand se puede observar que es heredado a su vez por todas las posibles operaciones que pueden expresarse, de modo que cada operación pueda redefinir el método evaluate de acuerdo con la funcionalidad que ha de implementar.

Por ejemplo, el método evaluate de la clase Plus simplemente retornará el valor numérico de la suma de su primer operando y su segundo operando. Teniendo en cuenta de que sus operandos a su vez pueden ser operaciones, la función ha de retonar en realidad el valor de la evaluación del primero de sus operandos sumado al valor de la evaluación del segundo de sus operandos.

De este modo llegará un momento en que haya que evaluar directamente objetos de clase SimpleFeature, MultipleFeature o Number, que serán las hojas del árbol resultante de parsear la restricción. Evaluar un número simplemente consiste en retornar su valor, y evaluar una característica consiste en comprobar si está seleccionada (en caso de ser simple) o mirar en cuántas ocasiones ha sido seleccionada (en caso de que sea múltiple).

Así pues, inicar la tarea de validación en el manejador requiere acceder a la operación más significativa de la restricción, ya que la clase Constraint no contiene un método evaluate. Conviene recordar que en el metamodelo habíamos indicado que una restricción solo se relaciona con una operación, y las demás las consigue mediante las relaciones de sus operandos. Gracias

a eso podemos concluir que evaluar la restricción es lo mismo que evaluar su operación booleana más significativa, a la que se accede directamente desde esa relación, que nos facilita mucho la tarea en este punto.

En general, la implementación del método .evaluate"para la mayoría de las operaciones es trivial y no conlleva más de un par de líneas de código (como el caso de la suma anteriormente definido). No obstante, algunas operaciones son algo más complicadas y requieren algo más de reflexión. Hablamos de las operaciones que requieren un contexto. Hay que tener en cuenta que, tal como ha sido explicado en capítulos anteriores, las operaciones con contexto solo miran una parte muy concreta de la configuración, e implementar fue complicado ya que las evaluaciones se van encadenando y ese contexto puede modificar el comportamiento de otras operaciones.

Para solucionar este problema se añadió el parámetro çontext", que simplemente es una Feature que indica a partir de qué punto en la configuración hay que tener en cuenta la evaluación. De este modo tenemos que modificar la implementación de la evaluación de SimpleFeature y MultipleFeature para tener en cuenta esto, y contar para el resultado de la evaluación únicamente las características que sean hijas de la Feature que se pasa como parámetro.

Una vez se ha concluido de implementar el método evaluate en todas las operaciones y se lanza el proceso desde el manejador del botón "Validate"se puede dar por finalizada la fase relativa a la semántica.

≪TODO: Capítulo 7: Discusión, Conclusiones y Trabajos Futuros≫