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

Índice general

Índice de figuras

Índice de cuadros

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.

Contents

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 solo 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 $CalculoRutasptimas \Rightarrow (Mapas \wedge Geolocalizacin)$.

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.

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

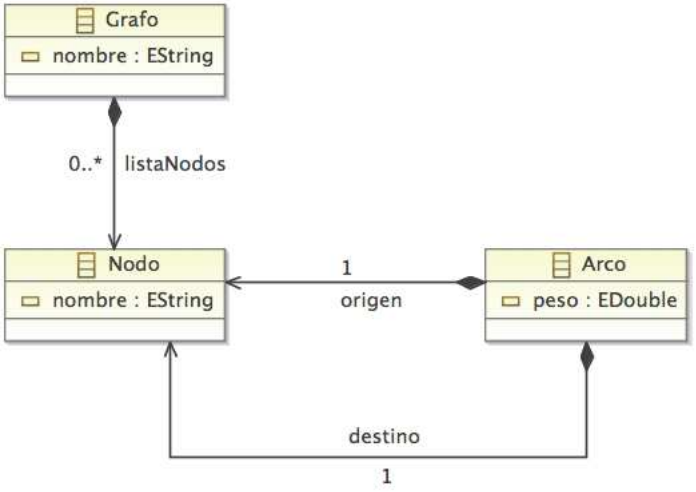

Figura 1.1: Metamodelo (sintaxis abstracta) de un lenguaje para modelar grafos pesados dirigidos

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*.

Ilustramos los conceptos anteriormente expuestos mediante la creación de un lenguaje simple para el modelado de grafos dirigidos y pesados. La Figura 1.1 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 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 1.1 representaría un determinado grafo pesado dirigido, con un número determinado de nodos, pesos y arcos. Por ejemplo, la Figura 1.2 muestra una instancia del metamodelo de la Figura 1.1 que representa un grafo dirigido y pesado que modela las distancias ≪TODO: Cambiar Ejemplo≫entre las ciudades de Santander, Barcelona y La Coruña. Cada ciudad se representa como un nodo, y

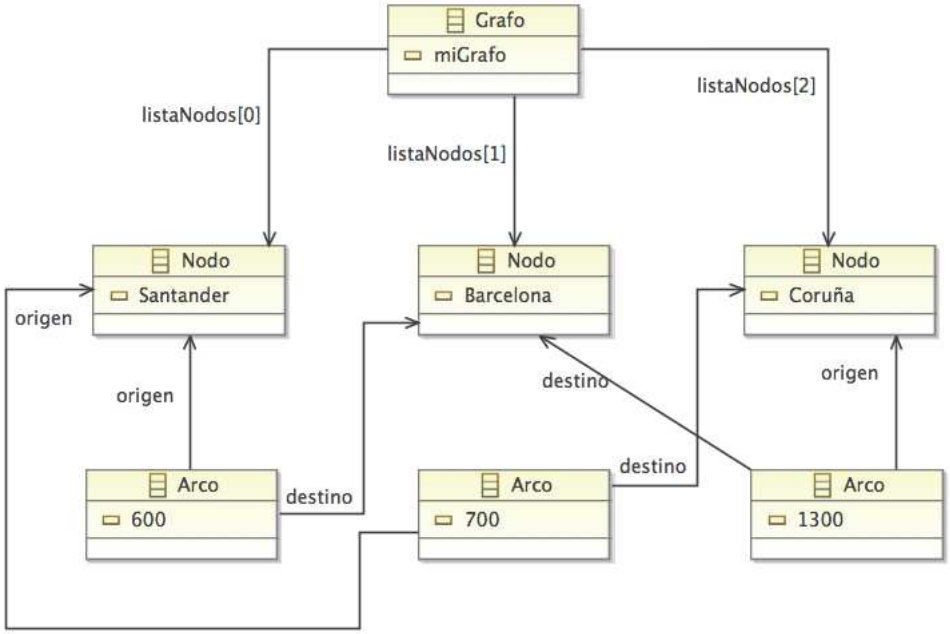

Figura 1.2: Modelo abstracto de un grafo pesado dirigido concreto

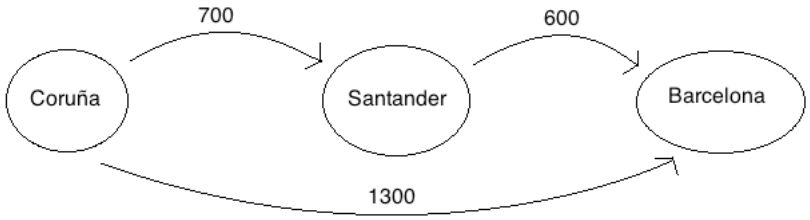

Figura 1.3: Modelo concreto de un grafo pesado dirigido concreto

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 1.2.

Por ejemplo, para nuestro ejemplo anterior, el del lenguaje para la espe-

cificación de grafos dirigidos pesados, podríamos elaborar una sintaxis visual donde los grafos se representasen utilizando la notación visual por todos conocida, tal como se ilustra en la Figura 1.3. Dicha figura muestra el mismo modelo que la Figura 1.2, 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 Tools*) [].

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 1.2 pero utilizando una sintaxis textual para nuestro lenguaje de grafos. En este caso, ≪TODO: Describir≫

Al igual que en el caso de las sintaxis visuales concretas, los entornos 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 1.2.

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 1.4 muestra como dicho proceso de desarrollo se ha instanciado para nuestro caso particular.

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.

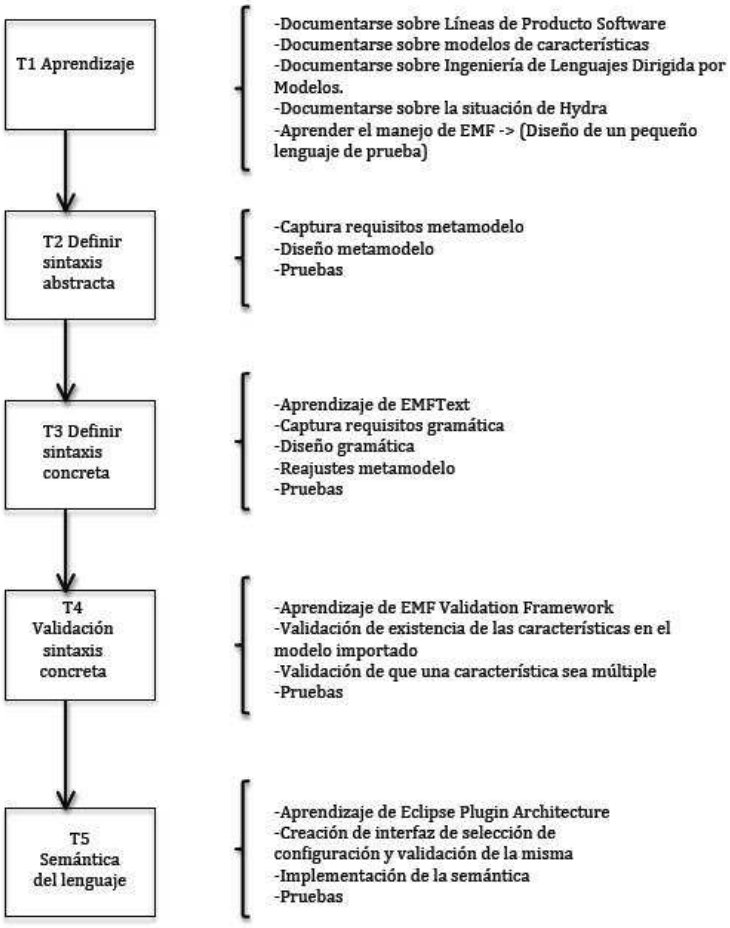

Figura 1.4: Proceso de desarrollo del Proyecto Fin de Carrera

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.

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 técnica 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.

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 **??**.

1.4. 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

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

2.1. 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.

2.2. 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 2.1 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 2.1: 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 2.2: Implementación del inicio de la gramática con EMFText. Con la figura 2.1 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 **??**. 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 2.2.

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 2.3: 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 2.1 mostrada anteriormente, y ahí terminaría.

2.3. 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 2.3.

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 **??** para comprobar que el árbol era el mismo que se creó en ese momento.

Capítulo 3

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

3.1. 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.

3.2. 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 4

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

4.1. 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 4.1 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.

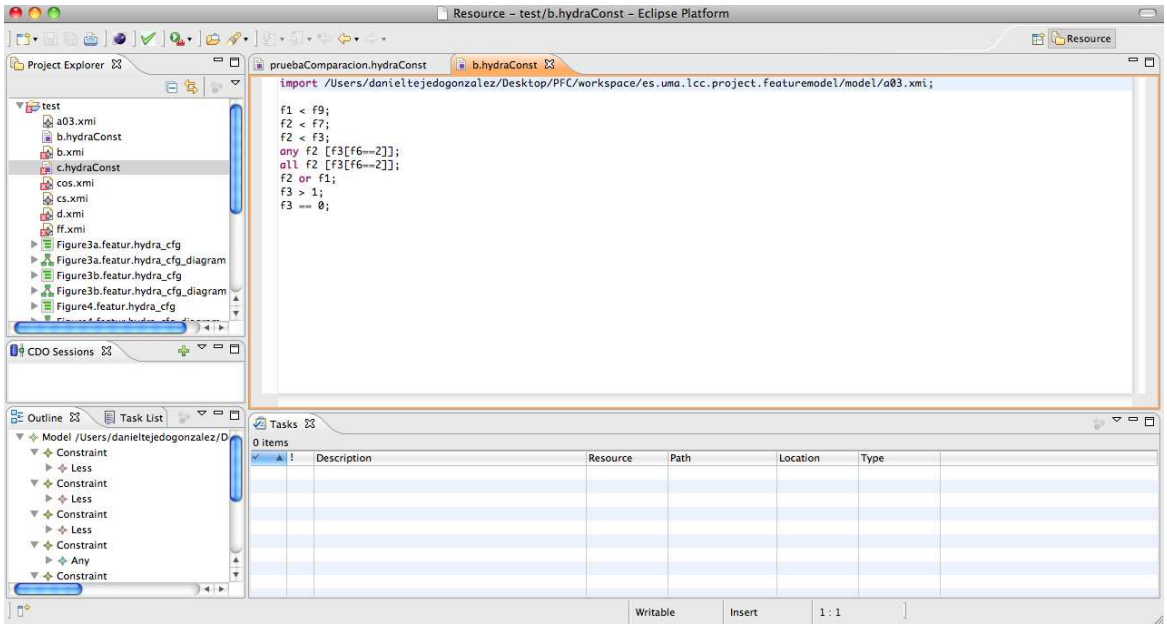

Figura 4.1: Captura de pantalla del editor en funcionamiento

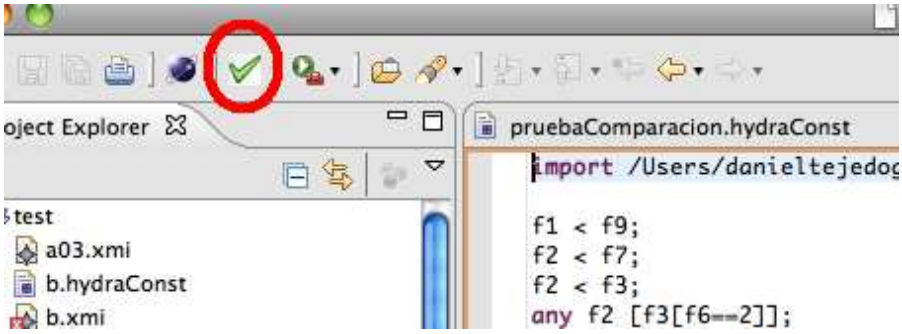

Figura 4.2: Botón que ejecutará la validación de las restricciones

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.

4.2. 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≫