UNIVERSIDAD
POLITECNICA
DE VALENCIA

DSIC
DEPARTAMENTO DE SISTEMAS
INFORMÁTICOS Y COMPUTACIÓN

Un Framework para el Análisis Automático de Líneas de Producto Software

Clara Khachan Cano

Universidad Politécnica de Valencia
Departamento de Sistemas Informáticos y Computación
Cno. De Vera, s/n. 46022 Valencia.

Dirigida por:

Abel Gómez
Silvia Abrahão

Índice de contenidos

Índice de Figuras

Índice de Tablas

Índice de Listados

Parte I

Introducción

1. Introducción

1.1 Contexto

El uso de modelos de características para describir Líneas de Productos Software o *Software Product Lines* (SPL) en inglés, es una práctica muy extendida en la industria de hoy en día. Las líneas de productos software [14] se presentan como un paradigma tecnológico para construir productos software centrándose en el desarrollo de un conjunto de productos distintos que comparten un conjunto de características que satisfacen las necesidades de un dominio en particular. De esta manera, la ingeniería de líneas de producto, consiste en la producción de familias de sistemas similares más que en el desarrollo de diferentes productos uno por uno desde el principio. En este contexto, los modelos de características se utilizan para representar todos los posibles productos de una línea de productos software. Los modelos de características representan de manera jerárquica las características. Estas características podrán definirse como opcionales, comunes, etc. También podrán definirse restricciones entre ellas, requiriendo o excluyendo la presencia de ciertas características cuando se seleccionan algunas de ellas. Teniendo en cuenta esto, los posibles productos serán todas las combinaciones posibles de las características.

En ocasiones, la producción de los modelos de características se realiza de manera ambigua e incorrecta. Conforme el modelo va creciendo, estos errores se acumulan produciendo modelos imprecisos que no se ajustan a la realidad que pretenden representar.

Una forma de detectar la presencia de errores en nuestro modelo es analizarlo para determinar su corrección. La tarea de analizar el modelo de forma manual es muy tediosa, y si estamos trabajando con modelos de características extensos, es incluso imposible de realizar. El uso de una herramienta que nos permita analizar de manera automática nuestro modelo nos ayuda a obtener modelos de características de calidad y libres de errores. De esta manera, a medida que el modelo va creciendo en número de características, podemos llevar a cabo análisis del modelo para determinar su corrección en una etapa de desarrollo temprana, ahorrando así todo tipo de costes derivados de una detección tardía.

Además, si revisamos la literatura actual, uno de los aspectos que podemos resaltar es que de las propuestas existentes para analizar los modelos de características de manera automática se soportan sólo sobre una notación concreta. Es posible que tengamos que resignarnos a utilizar una herramienta que no cumpla nuestras expectativas o incluso que no haya herramienta alguna que soporte nuestra notación

Por otra parte, tenemos el Desarrollo de Software Dirigido por Modelos (DSDM), en el cual un modelo es una estructura de datos que puede ser definida mediante un lenguaje de modelado (generalmente llamado metamodelo). Un modelo permite definir la funcionalidad, estructura y/o comportamiento de un sistema [49] dependiendo del metamodelo utilizado. La utilización de modelos en un proceso de DSDM permite

automatizar el desarrollo de aplicaciones y su evolución mediante técnicas de programación generativa [16], como las transformaciones de modelos y la generación de código.

La definición de modelo que se maneja en la Ingeniería Dirigida por Modelos casa a la perfección con el modelado de características. En el desarrollo de modelos de características, existen diferentes aportaciones sobre las notaciones empleadas para su representación. Así, podemos definir un metamodelo que refleje la funcionalidad de nuestro modelo de características.

1.2 Motivación

En los párrafos introductorios se describe el contexto en el que nos hayamos a la hora de realizar el presente trabajo. Pero la verdadera motivación que subyace, se desencadenó a raíz de la colaboración, en materia de líneas de productos software, con la empresa británica Rolls Royce.

Rolls Royce es un grupo de compañías de automóviles y aeronáutica fundada por Henry Royce y Charles Stewart Rolls en 1906. En el sector de la aeronáutica, la empresa se encarga de motorizar algunos de los aviones diseñados por Airbus entre otros proyectos. Estamos hablando de artefactos de gran complejidad formados por numerosos componentes. Dichos componentes pueden configurarse de distintas maneras y cumplir infinidad de requisitos que aseguren su correcto funcionamiento.

Imaginemos una SPL como la de la empresa inglesa. Efectivamente, el modelo de características que la especifica contiene más de mil características. Elaborar una línea de productos en este tipo de industrias no es tarea fácil; hay que tener en cuenta muchos detalles, y el resultado final es un modelo muy grande. Además, teniendo en cuenta el tipo de artefactos que se están elaborando, si queremos utilizar la información contenida en esta SPL como entrada para cualquier proceso, es de vital importancia que no contenga errores. Para poder conseguir un modelo perfecto hay que ir refinándolo y modificándolo gradualmente, para poder realizar esta tarea, tenemos que ser capaces de analizar el modelo. De su análisis se desprenderán las conclusiones que permitirán corregirlo y también depurar las técnicas para su elaboración, de manera que cada vez se incorporen mejores prácticas.

De momento nos encontramos con un modelo de características muy grande, muy complejo y representado con una notación concreta. El análisis de SPL es un tema de «rabiosa» actualidad en la literatura de hoy en día. La tendencia actual se decanta por el análisis basado en los lenguajes lógicos, de manera que representando nuestro modelo de esta forma, podremos analizar sus contenidos. Ahora tenemos el modelo anteriormente nombrado y una serie de propuestas (algunas con sus correspondientes herramientas) para poder analizarlo y lanzar luz sobre su corrección. Pero nuestro modelo utiliza una notación concreta y cada herramienta de análisis automático acepta unos modelos en base a su notación.

Durante la concepción de este trabajo, también nos documentamos sobre cuáles eran las propuestas actuales para analizar automáticamente modelos de características.

En los últimos años, hay un nombre que aparece recurrentemente; FaMa. En el trabajo de Benavides [4], uno de los padres de la herramienta, se compara la funcionalidad que aporta esta herramienta frente a las propuestas existentes más importantes.

Esta tesis ha sido realizada en el grupo de investigación ISSI (Ingeniería del Software y Sistemas de Información) del Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia. En el grupo, se dispone de un framework llamado MULTIPLE que está formado por una serie de componentes. Una parte de este framework consiste en una herramienta para la explotación de Líneas de Producto Software (SPL). Esta herramienta nos permiten trabajar con los modelos de características que especifican estas SPLs, proporcionándonos un editor gráfico y la capacidad de realizar una serie de validaciones sobre los modelos. Al tratarse de un framework extensible por la plataforma en la que ha sido desarrollado, podría aprovecharse dicha herramienta para extenderla aportando la funcionalidad que hiciera posible el análisis de SPLs.

1.3 Objetivos

En este trabajo queremos demostrar como, mediante el desarrollo de un framework que integra el modelado de característica, la Ingeniería Dirigida por Modelos (MDE) y una herramienta de análisis automático de modelos de características, podemos llevar a cabo un análisis completo de un modelo de características industrial de gran escala. De esta manera, podremos deliberar sobre la corrección del mismo y manejar resultados que nos permitan extraer conclusiones acerca del origen y solución de los problemas extraídos.

Además, como comentábamos en el apartado anterior, nos encontramos con la problemática de que dependiendo de la notación en la que esté expresado nuestro modelo, tendremos que elegir una herramienta que soporte esa notación.

La novedad de este framework reside en el método que emplea, concebido de manera abstracta para que, partiendo de un modelo de características en el que la notación empleada no es relevante, pueda usarse una herramienta tan potente como FaMa para su análisis automático.

Nuestro framework nos permitirá tomar como entrada el modelo industrial (de más de mil características) y analizarlo sintáctica y semánticamente. Con este fin, nuestro framework implementa los componentes necesarios para la realización de las siguientes tareas:

- Obtención de un modelo analizable. Esta parte engloba el análisis y corrección sintáctica del modelo necesarios para obtener un modelo válido. Además, exponemos la clasificación y soluciones a los errores extraídos.

- Transformación del modelo sintácticamente correcto a un modelo entendible por la herramienta de análisis automático.

- Análisis semántico del modelo mediante la herramienta de análisis automático. En este último paso, realizaremos una clasificación de los errores y hablaremos de su origen y corrección.

Nuestra herramienta está construida sobre Eclipse [53] y EMF [53], lo que además nos permite seguir una estrategia basada en MDE ya que:

- El modelo válido obtenido tras el análisis sintáctico se ajusta a nuestro metamodelo propuesto que sirve de pivote entre tecnologías.

- Se realizan una serie de transformaciones de nuestro modelo al modelo aceptado por la herramienta de análisis automático, usando el lenguaje de transformaciones QVT-Relations. Usaremos QVT-Relations ya que es un lenguaje declarativo que nos proporciona trazabilidad y define reglas de transformación mediante patrones, lo que lo hace un lenguaje entendible para el usuario.

Para la realización de esta herramienta, reutilizaremos los componentes de MULTIPLE que nos permiten explotar líneas de producto software, añadiendo aquellos componentes que nos permitirán llevar a cabo los objetivos descritos en esta sección.

Para la implementación de las transformaciones, usaremos la interfaz de transformaciones implementada también en MULTIPLE. Esta interfaz utiliza el motor de transformaciones MediniQVT [48] para dar soporte al lenguaje de transformaciones QVT-Relations. Además, utilizaremos la herramienta de análisis automático FAMA [37] para poder realizar el análisis semántico del modelo.

Asimismo, repasaremos los beneficios, problemas y limitaciones encontradas del uso de una herramienta de análisis automático con un modelo de gran escala.

1.4 Descripción del documento

Este documento se organiza de la siguiente manera. En este primer capítulo, se detallan los motivos que nos han llevado al desarrollo del framework y los objetivos que se persiguen. La Parte II de este documento contiene los fundamentos en los que se basa nuestro desarrollo. Esta parte se compone de los capítulos 2, 3 y 4. En el capítulo 2, se describe el estado del arte de los modelos de características como especificaciones de las Líneas de Producto Software y las propuestas existentes para el análisis automático de los modelos de características. Además, se expone el framework para el desarrollo de herramientas de análisis automático FaMa y la herramienta que nos proporciona. El capítulo 3 repasa el estado del arte en Ingeniería Dirigida por Modelos y qué soporte tecnológico existe en este campo. El siguiente capítulo (capítulo 4) presenta la herramienta MULTIPLE, que integra el modelado de características y la Ingeniería Dirigida por Modelos. También se resaltan las carencias detectadas en esta herramienta. MULTIPLE será la base de nuestro desarrollo. En la Parte III, compuesta de los capítulos 5, 6 y 7, se presenta nuestro framework para el análisis de modelos de características. En el capítulo 5 se muestran nuestras contribuciones. El capítulo 6 contiene la estructura de nuestro Framework. En él se describirán cada uno de los

componentes que conforman nuestro framework y se describe en detalle cómo hemos implementado la propuesta. En el capítulo 7 se presenta el caso de estudio sobre el que hemos aplicado nuestra propuesta, describiendo cómo se ha empleado la herramienta desarrollada y elaborando un informe detallado de los resultados obtenidos. Este informe contiene la descripción, clasificación y solución aportada a los errores detectados. Además se comentan las limitaciones encontradas al analizar el modelo de características de gran escala. Finalmente, se presentan las conclusiones, trabajos futuros y agradecimientos. Adicionalmente, se proporcionan una serie de anexos con información detallada sobre el código de implementación de la herramienta.

Parte II

Fundamentos

2. Modelos de Características

De un tiempo a esta parte, el software ha ido invadiendo prácticamente todos los sectores. Hoy en día representa la base de muchas organizaciones que nunca se habían imaginado inmersas en el negocio software. La mayoría de estas organizaciones producen familias de sistemas similares, que se diferencian por sus características. Un ejemplo ilustrativo sería el los dispositivos móviles. En una empresa de desarrollo de estos dispositivos, nos encontramos con que toda la gama de móviles producidos cuentan con una serie de componentes y características comunes (pantalla, cámara, antena, etc.) y otras que varían (pantalla capacitiva o resistiva, cámara frontal y/o trasera, etc.).

Según Clements y Northrop [14]:

«Una línea de productos software es un conjunto de sistemas de software que comparten un conjunto común y gestionado de características que satisfacen las necesidades específicas de un segmento particular de mercado y que se desarrollan a partir de un conjunto común de recursos (core–assets) de una forma preestablecida».

Las líneas de productos software surgen como alternativa a aquellas empresas que se dedican al desarrollo de software y que lo hacen empezando sus proyectos desde cero cada vez. De esta manera, se reutiliza el trabajo realizado en proyectos previos, disminuyendo el esfuerzo al no tener que desarrollar siempre desde el principio.

En este contexto, se plantea la forma de especificar los productos que una SPL puede producir. En algunas industrias ya había surgido, de manera natural, la especificación de las SPL mediante características opcionales que ayudaban al cliente a configurar su producto dependiendo de sus necesidades.

En 1990, Kang [27] presenta, por primera vez, el concepto de modelo de características (FM). Los modelos de características tienen como función principal representar los aspectos comunes y las divergencias entre los distintos productos de una Línea de Productos Software o *Software Product Line* (SPL) en inglés. Para hacerlo definen una serie de relaciones entre las características o *features* que se clasifican en:

- Relaciones padre-hijo: Son las relaciones que se definen entre una característica padre y sus sub-características hijas.

- Relaciones conocidas como *coss-tree* que describen las restricciones entre características como la exclusión (la selección de una característica excluye a otra) y la inclusión (la selección de una característica implica la selección de otra).

De esta manera, los modelos de características representan a las características de manera jerárquica conformando un árbol.

En la sección 2.1 expondremos las diferentes propuestas existentes para representar estos modelo de características. En el apartado 2.2 se explica en qué consiste el análisis de estos modelos de características y en qué nos beneficia. También se introducirá la herramienta de análisis automático FaMa, ya que es la que elegiremos para la realización del caso práctico.

2.1 Notaciones

En esta primera propuesta de Kang, se especificó una primera notación para representar los modelos de características conocida como FODA [27]. Pero conforme el interés en esta tecnología ha ido creciendo, son muchos los que se han dedicado al estudio del modelado de características, desarrollando notaciones que extienden esta primera propuesta.

Las secciones 2.1.1 y 2.1.2 contienen los tipos de notaciones presentes en la actualidad, resumiendo las propuestas existentes que dividimos en: modelos básicos y modelos extendidos. Dentro de los modelos extendidos englobamos los modelos basados en cardinalidad y los modelos con atributos.

2.1.1 Modelos de características básicos

En la tesis doctoral de Benavides [4], se clasifican como modelos de características básicos las propuestas de FODA [27] y feature-RSEB [24].
Como antes hemos comentado, los modelos constan de relaciones padre-hijo y de restricciones *cross-tree.* En estos modelos de características básicos encontramos los siguientes tipos de relaciones:

- **Relaciones padre-hijo:**

  - Obligatoria (*Mandatory)*. Esta relación entre dos *features,* se presenta cuando la *feature* hija ha de incluirse en un producto si la *feature* padre está incluida. Por ejemplo, es *obligatorio* que un teléfono móvil tenga pantalla. La figura 2.1.a representa esta relación.

  - Opcional (*Optional*). Esta relación (figura 2.1.b) entre dos *features,* se presenta cuando la *feature* hija puede o no incluirse en un producto si la *feature* padre está incluida. Por ejemplo, es *opcional* que un teléfono móvil tenga cámara.

  - Alternativa (*Alternative*). Esta relación (figura 2.1.c) entre una *feature* padre y un grupo de *features* hijas*,* se presenta cuando sólo una de las *features* hijas puede incluirse en un producto si la *feature* padre está incluida. Por ejemplo, la pantalla de un móvil sólo puede ser táctil o normal, pero no ambas.

  Además, feature-RSEB introduce una nueva relación que no había en FODA:

  - Or. Esta relación (figura 2.1.d) entre una *feature* padre y un grupo de *features* hijas*,* se presenta cuando una o más *features* hijas puede incluirse en un

producto si la *feature* padre está incluida. Por ejemplo, un teléfono móvil puede tener una cámara frontal y otra trasera.

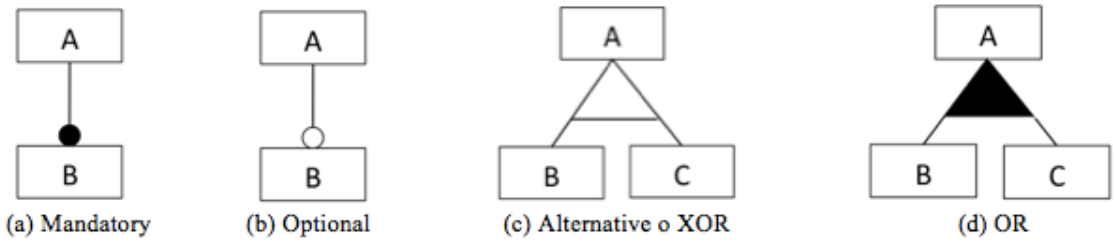


Figura 2.1 Representación tipos de relaciones padre-hijo

❖ **Restricciones *cross-tree*:**

- Exclusión (*Excludes*). Si una *feature* A excluye a una *feature* B, significa que si A está presente en el producto, B no puede estar, y viceversa. Por ejemplo, un teléfono móvil no puede tener una pantalla normal y tener reconocimiento de escritura manual.

- Inclusión (*Requires*). Si una *feature* A requiere a una *feature* B, significa que si A está presente en el producto, B también ha de estar, pero no al contrario. Por ejemplo, si un teléfono móvil tiene radio, ha de tener una entrada para auriculares, pero no al contrario.

Por ejemplo, el software embebido en un teléfono móvil viene determinado por las características del teléfono; cada conjunto de características puede definir un programa único en una línea de productos. La Figura 2.2 es un modelo de características con notación feature-RSEB que representa esta línea de productos. Un teléfono móvil consiste en una pantalla, y opcionalmente puede tener reconocimiento de escritura manual, cámara, radio o auriculares. La pantalla puede ser táctil o manual (sólo una) y la cámara puede ser frontal, trasera o ambas. Si la pantalla es normal no puede haber reconocimiento de escritura manual, y si el teléfono tiene radio tendrá que tener obligatoriamente auriculares ya que gracias a la antena de los auriculares que funciona la radio.

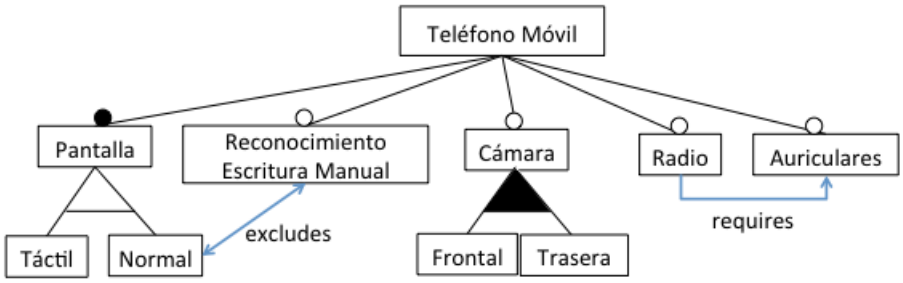


Figura 2.2 Ejemplo representación modelo de características Teléfono Móvil con notación feature-RSEB

2.1.1.1 Propuesta PLUSS (Product Line Use case modeling for Systems and Software engineering)

En esta popuesta de Eriksson y Börstler [21], el manejo de la variabilidad de los modelos de características se basa en feature-RSEB.

En PLUSS, los sistemas de características se organizan en árboles de nodos AND y OR, que representan las partes comunes y divergentes en una familia de sistemas relacionados. Las características generales se sitúan en la parte superior del árbol y las más refinadas más abajo.

La notación PLUSS describe los siguientes tipos de relaciones:

- *Mandatory*.
- *Optional*.
- *Single Adaptor*. Representan la relación de selección "exactamente-uno-de-muchos", lo que significa que sólo podrá seleccionarse una *feature* de entre las *features* hermanas de este tipo. Se corresponde con la relación XOR de las propuestas tradicionales.
- *Multiple Adaptor*. Representan la relación de selección "al-menos-uno-de-muchos", lo que significa que se podrán seleccionar de 1 a n *features* de entre las *features* hermanas de este tipo, siendo n el número de *features* posibles a seleccionar. Se corresponde con la relación *OR* de las propuestas tradicionales.
- *Requires.*
- *Excludes.*

La Figura 2.3 muestra un ejemplo de modelo de características representado en PLUSS.

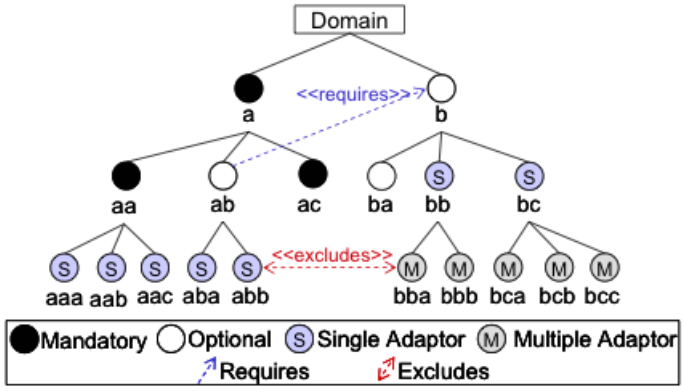


Figura 2.3 FM ejemplo en notación PLUSS

2.1.2 Modelos de características extendidos

Entendemos como modelos de características extendidos aquellos modelos que extienden la expresividad de la notación básica para poder representar modelos de características más complejos. Según esta definición, en la literatura actual, nos encontramos con los modelos basados en cardinalidades y los modelos que incorporan atributos a las características. En [4] podemos encontrar estos dos tipos de modelos

clasificados por separado como modelos basados en cardinalidades y modelos extendidos [2] [7]. Lo cierto es que ambos amplían la expresividad de los modelos aportando nuevos conceptos, por lo tanto pueden englobarse ambos tipos de modelos como modelos de características extendidos.

2.1.2.1 Modelos de características basados en cardinaliades

La principal diferencia que presentan estos tipos de modelos respecto a la propuesta inicial de FODA [27], es que cada *feature* tiene asociada una cardinalidad que especifica cuántos clones de esa *feature* podrán aparecer en una configuración específica.

El motivo principal por el cual se introduce el concepto de cardinalidad, es la imposibilidad de representar algunos casos sólo con las relaciones alternativa y *OR*. El clonado de *features* es útil para definir varias copias de una parte del sistema que puede configurarse de maneras diferentes.

El origen de los modelos basados en cardinalidades puede observarse en trabajos como [15] [30] [31] dónde se propone la extensión de los modelos básicos añadiendo multiplicidades. Así, las relaciones obligatoria y opcional conservan el mismo significado que en FODA, pero las relaciones alternativa y or se generalizan en lo que llamarían relación 'set', estableciendo las siguientes analogías:

- Alternative = Set con cardinalidad [1..1]
- Or = Set con cardinalidad [1…N], N = número de *features* en la relación.

Pero no fue hasta más adelante, cuando Czarnecki [17] [18] propuso los modelos basados en cardinalidades. Czarnecki definió así, lo que conocemos como cardinalidad de las *features,* que viene a decirnos que las veces que una feature está incluida en un producto viene determinada por su cardinalidad. De esta manera, las relaciones:

- *Mandatory* se define con cardinalidad [1..1]
- *Optional* se define con cardinalidad [0..1]

La Figura 2.4 muestra el ejemplo del modelo de características que representa la línea de productos de los teléfonos móviles con notación basada en cardinalidades.

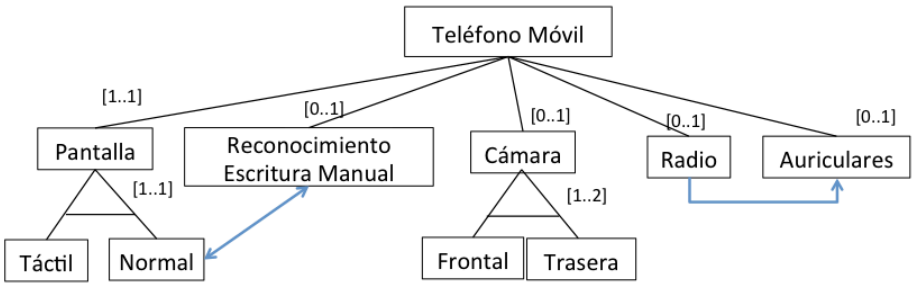


Figura 2.4 Ejemplo representación FM de Teléfono Móvil con notación basada en cardinalidad

2.1.2.2 Modelos de características con atributos

En ocasiones se precisa extender los modelos de características con más información acerca de las *features*.

En la propuesta inicial de FODA, ya se contemplaba la inclusión de información extra como las relaciones entre *features* y atributos de las *features*. Además, Czarnecki introdujo el concepto de parámetros, usados para representar valores simples de las *features*. En [13], se hacía distinción entre *features* funcionales y de calidad, señalando la necesidad de un método específico para incluir atributos en los modelos de características. Son muchos los autores que han propuesto la inclusión de atributos en los modelos de características, lo que refleja que existe consenso sobre su inclusión. Sin embargo, no existe una notación estándar ni un acuerdo sobre qué tipo de atributos pueden incluirse en los modelos de características.

En [4] recogen los siguientes conceptos, comúnmente utilizados cuando se trabaja con atributos de *features*:

- Atributo. Cualquier característica de una *feature* que puede medirse. Por ejemplo, en el caso de un teléfono móvil, un atributo podría ser la resolución de la cámara (Figura 2.5)

- Dominio del atributo. Especifica el espacio de posibles valores para un atributo. Todo atributo tiene asociado un dominio.

- Valor de un atributo. Es el valor de un atributo perteneciente a un dominio. Puede haber un valor por defecto en el caso de que no se seleccione la *feature*. Un valor puede ser:

  - Atributo Básico: directamente un valor en el dominio

  - Atributo Derivado: una expresión que combine otros atributos de la misma u otras *features*.

- Relación de atributos. Una relación entre uno o más atributos de una *feature* o una misma *feature*.

El ejemplo de la Figura 2.5 muestra un extracto del modelo de características 'Teléfono Móvil' en el que la característica cámara tiene dos atributos; resolución x y resolución y.

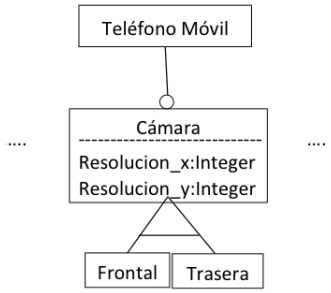


Figura 2.5 Ejemplo representación FM de Teléfono Móvil con atributos

2.2 Análisis de modelos de características

Los modelos de características nos permiten especificar los productos que una línea de productos puede producir. Estos modelos se han usado, principalmente, en muchas actividades relacionadas con la ingeniería de dominios, ingeniería de aplicaciones o para gestión de las línea de productos [4]. Sin embargo, contienen gran cantidad de información útil que podemos extraer si sabemos cómo hacerlo. El número de productos potenciales, las características comunes y las divergentes, son un ejemplo de la información contenida en dichos modelos. Si sólo consideramos los modelos de características como elementos gráficos para la especificación de las SPL's, estamos desaprovechando el potencial que estas estructuras pueden ofrecernos.
Pero para extraer esta información para que nos sea de utilidad, hemos de confiar en que los datos contenidos en nuestro modelo son exactos y no contienen errores o inconsistencias.

Desde el nacimiento del modelado de características con la propuesta de Kang [27], se señala la necesidad de analizar los modelos de características antes de emplear la información contenida en ellos para cualquier otro propósito. Desde un principio, los que trabajamos con SPL nos topamos con el desafío de encontrar la manera de analizar nuestros modelos de características con la mayor precisión posible e ir obteniendo cada vez modelos más perfectos.

Siendo el modelado de características una disciplina recién nacida, ya se cayó en la cuenta de que la tarea de analizar los modelos no es nada trivial. Primero hay que determinar en qué puntos pueden haber errores para poder detectarlos, luego hay que concebir técnicas para su detección y deliberar sobre qué solución es la más correcta para cada problema.

Pero aquí no acaban las complicaciones, y es que un modelo de características es una estructura en la que se disponen jerárquicamente las características. ¿Qué pasa si nuestro modelo contiene mil características? Analizar manualmente una estructura de estas dimensiones es tarea imposible. Como Kang muy bien señalaba, es de vital importancia contar con herramientas que posibiliten el análisis automático de los mismos.

Esta sentencia puso manos a la obra a los investigadores de la materia. Y como la historia nos dicta, las matemáticas, y en concreto la lógica, han sido el mayor surtidor de mecanismos de comprobación jamás inventado. Se puede decir que todo lo que

puede representarse matemáticamente, puede comprobarse. Así que, como señalan en [4], transformar nuestro modelo en una representación lógica, nos permitirá usar los correspondientes *solvers* o soluciondores para analizar la SPL de manera automática.

La Figura 2.6 refleja el proceso de automatización del análisis:

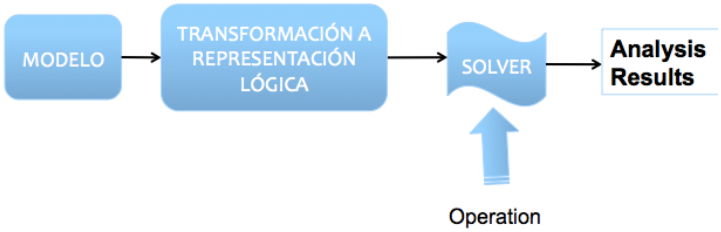


Figura 2.6 Proceso para analizar una Línea de Productos Software [4]

El uso de los *solvers* nos va a permitir operar sobre el modelo para extraer información útil. Existen diferentes propuestas en la literatura para analizar un modelo de características. En cada propuesta se presentan diferentes implementaciones que usan diferentes *solvers*. Algunos *solvers* realizan mejor que otros ciertas operaciones de análisis.

Pero para poder analizar nuestro modelo de características en contenido (semánticamente), es necesario que haya sido correctamente elaborado en forma (sintácticamente). Es una tarea que prácticamente se ha obviado en la literatura, pero lo cierto es que en la práctica, nos encontramos con modelos de características que contienen errores sintácticos, como el uso incorrecto de la variabilidad, la repetición de características a lo largo del modelo, etc… Estos errores, que pueden no ser detectados a priori, alteran el contenido de nuestro modelo, haciendo que no se corresponda con la realidad que quiere representar. En la sección 2.2.1 hablaremos de la posible solución a este problema. En las secciones 2.2.2 y 2.2.3 repasaremos las propuestas de análisis automático existentes y explicaremos las operaciones que usaremos para el análisis de nuestro caso práctico.

2.2.1 Análisis sintáctico de los modelos de características

Mucho se ha hablado del análisis del contenido de los modelos de características y de cómo automatizarlo. Pero se habla poco de si el modelo con el que estamos trabajando ha sido, en primer lugar, correctamente elaborado.

Lo cierto es que las líneas de productos desarrolladas en los diferentes campos de la industria, y en concreto los modelos de características para especificarlas, han sido elaborados manualmente, y todo aquello que se elabora manualmente puede contener errores. Si además tenemos modelos de características muy grandes, estas inconsistencias (aunque más fáciles de detectar que las de contenido) pueden ser pasadas por alto. Por ello, creemos que antes de pasar a un análisis semántico de nuestro modelo mediante la herramienta elegida, hemos de comprobar que efectivamente el modelo no contiene errores sintácticos. Esta tarea es mucho más fácil de llevar a cabo que el análisis de contenidos, ya que no es necesario representar de manera lógica

nuestro modelo para poder operar sobre él. Simplemente necesitamos un algoritmo que recorra nuestro modelo de características y compruebe que cada característica y cada relación (padre-hijo y *cross-tree*) ha sido correctamente especificada en forma. Por ejemplo, si calculamos el número de productos potenciales de nuestra SPL y el modelo contiene una característica repetida en distintas partes de la jerarquía, los resultados no serán los que debieran y este error pasaría desapercibido a ojos de la herramienta de análisis automático.

Además, recorriendo la definición de cada característica padre y sus hijos podemos comprobar si los mecanismos de variabilidad se han usado de manera adecuada.

La necesidad de analizar sintácticamente los modelos de características, surge como resultado del uso de herramientas que nos permiten definir modelos inconsistentes. Lo ideal es que usemos herramientas que controlen que no se cometen errores en la creación del modelo.

2.2.2 Un repaso a las propuestas de análisis automático de modelos de características

Si repasamos todas las publicaciones presentes en la literatura sobre el análisis automático de modelos de características, encontramos varias propuestas. Estas pueden clasificarse en :

1. Basadas en el cálculo proposicional [29] [34] [43] [1]
2. Basadas en lógica descriptiva [42]
3. Basadas en programación con restricciones [7] [5] [8] [9] [36]
4. Basadas en soluciones ad-hoc [40] [12] [45]

En el 2010, se publicó un artículo sumamente interesante titulado *Automated Analysis of Feature Models 20 Years Later: A Literature Review* [10]. En él se recopilan las propuestas más relevantes en el ámbito del análisis automático de modelos de características durante los veinte años desde su concepción. Este documento recoge todas las operaciones de análisis identificadas, las técnicas usadas para el análisis automático, los resultados obtenidos de probar las distinta técnicas y además, proporciona un debate sobre estos resultados señalando los desafíos a los que debemos enfrentarnos en un futuro.

Aunque repasar todas estas propuestas no es el objetivo de la presente tesis, si es interesante resaltar que lo que diferencia a estas tres primeras propuestas es el tipo de lógica que se utilizará para representar el modelo. Dependiendo del tipo de representación lógica que usemos, tendremos disponibles unos u otros *solvers*. Y cada *solver* pone a disposición del usuario una serie de operaciones a realizar sobre el modelo.

En el caso de las soluciones ad-hoc, se emplean otro tipo de lenguajes de descripción no lógicos para representar los modelos de características

2.2.3 Operaciones de análisis

Como se recoge en [10], son muchas las operaciones de análisis identificadas a lo largo de la literatura. En este apartado haremos mención de aquellas que utilizaremos para el análisis de nuestro modelo de características. Estas operaciones, toman como entrada un modelo de características proporcionando respuestas extraídas de consultar sus propiedades sin modificar el modelo.

2.2.3.1 Determinar si un modelo de características es inválido (void)

Entrada: Modelo de características
Salida: Valor que indica si el modelo es válido o no.

Un modelo de características es válido si representa al menos un producto. Un modelo de características básico sin restricciones *cross-tree* no puede ser inválido porque es imposible incluir contradicciones si no existen este tipo de restricciones. De esta forma, un modelo sólo puede ser inválido si contiene restricciones *cross-tree.*
Esta operación se hace necesaria especialmente si estamos trabajando con modelos de características muy extensos. Esto se debe a que en la localización manual de errores en los modelos de características existe una probabilidad elevada de fallar [27] [1].

En la Figura 2.7 se muestra un modelo de características *void* ya que no representa ningún producto.

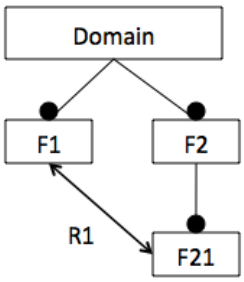


Figura 2.7 Modelo de características *void*

2.2.3.2 Determinar si un modelo es válido para un modelo de características

Entrada: Modelo de características, producto.
Salida: Valor que indica si el producto es válido o no.

Esta operación puede resultarle útil a un analista que quiera verificar si un producto está disponible en la SPL o no.
En el ejemplo siguiente, el producto Producto1, no pertenece al modelo de características descrito en la Figura 2.2 ya que un teléfono móvil no pude tener una pantalla táctil y otra normal al mismo tiempo. Sin embargo, el Producto2 sí que pertenece al modelo de características descrito en la Figura 2.2.

Producto1 = {Teléfono móvil, Pantalla, Táctil, Normal, Cámara, Frontal}

Producto2 = { Teléfono móvil, Pantalla, Táctil, Cámara, Frontal }

2.2.3.3 Obtener todos los productos

Entrada: Modelo de características.
Salida: Todos los productos válidos del modelo.

Esta operación es muy práctica cuando el número de productos es bajo, sin embargo, su ejecución puede resultar inviable en modelos de características con un gran número de productos potenciales.
Además, puede ser utilizada para saber si un modelo de características es *void*, ya que un modelo no es *void* si el conjunto de productos que devuelve esta operación no es vacío. Por el contrario, es *void* si el conjunto de productos que devuelve es vacío.
Esta operación no es muy utilizada para realizar un análisis del modelo, pero es útil para definir y llevar a cabo otras operaciones de análisis.

Por ejemplo, el modelo de la Figura 2.2 devuelve el siguiente conjunto de productos:

```
Product 1: Telefono_Movil Pantalla Tactil
Product 2: Telefono_Movil Pantalla Normal
Product 3: Telefono_Movil Pantalla Tactil Reconocimiento_Escritura_Manual
Product 4: Telefono_Movil Pantalla Tactil Camara Frontal
Product 5: Telefono_Movil Pantalla Normal Camara Frontal
Product 6: Telefono_Movil Pantalla Tactil Reconocimiento_Escritura_Manual Camara Frontal
Product 7: Telefono_Movil Pantalla Tactil Camara Trasera
Product 8: Telefono_Movil Pantalla Tactil Camara Frontal Trasera
Product 9: Telefono_Movil Pantalla Normal Camara Trasera
Product 10: Telefono_Movil Pantalla Normal Camara Frontal Trasera
Product 11: Telefono_Movil Pantalla Tactil Reconocimiento_Escritura_Manual Camara Trasera
Product 12: Telefono_Movil Pantalla Tactil Reconocimiento_Escritura_Manual Camara Frontal Trasera
Product 13: Telefono_Movil Pantalla Tactil Auriculares
Product 14: Telefono_Movil Pantalla Normal Auriculares
Product 15: Telefono_Movil Pantalla Tactil Reconocimiento_Escritura_Manual Auriculares
Product 16: Telefono_Movil Pantalla Tactil Camara Frontal Auriculares
Product 17: Telefono_Movil Pantalla Normal Camara Frontal Auriculares
Product 18: Telefono_Movil Pantalla Tactil Reconocimiento_Escritura_Manual Camara Frontal Auriculares
Product 19: Telefono_Movil Pantalla Tactil Camara Trasera Auriculares
Product 20: Telefono_Movil Pantalla Tactil Camara Frontal Trasera Auriculares
Product 21: Telefono_Movil Pantalla Normal Camara Trasera Auriculares
Product 22: Telefono_Movil Pantalla Normal Camara Frontal Trasera Auriculares
Product 23: Telefono_Movil Pantalla Tactil Reconocimiento_Escritura_Manual Camara Trasera Auriculares
Product 24: Telefono_Movil Pantalla Tactil Reconocimiento_Escritura_Manual Camara Frontal Trasera Auriculares
Product 25: Telefono_Movil Pantalla Tactil Radio Auriculares
Product 26: Telefono_Movil Pantalla Normal Radio Auriculares
Product 27: Telefono_Movil Pantalla Tactil Reconocimiento_Escritura_Manual Radio Auriculares
Product 28: Telefono_Movil Pantalla Tactil Camara Frontal Radio Auriculares
Product 29: Telefono_Movil Pantalla Normal Camara Frontal Radio Auriculares
Product 30: Telefono_Movil Pantalla Tactil Reconocimiento_Escritura_Manual Camara Frontal Radio Auriculares
Product 31: Telefono_Movil Pantalla Tactil Camara Trasera Radio Auriculares
Product 32: Telefono_Movil Pantalla Normal Camara Trasera Radio Auriculares
Product 33: Telefono_Movil Pantalla Tactil Reconocimiento_Escritura_Manual Camara Trasera Radio Auriculares
Product 34: Telefono_Movil Pantalla Tactil Camara Frontal Trasera Radio Auriculares
Product 35: Telefono_Movil Pantalla Normal Camara Frontal Trasera Radio Auriculares
Product 36: Telefono_Movil Pantalla Tactil Reconocimiento_Escritura_Manual Camara Frontal Trasera Radio Auriculares
```

2.2.3.4 Recuperar las características comunes (core)

Entrada: Modelo de características.
Salida: Conjunto de características que aparecen en todos los productos.

Esta operación es útil cuando se quieren determinar qué características son las más importantes de una SPL.

Por ejemplo, el conjunto de características comunes o *core* del modelo de la Figura 2.2 es : {Teléfono Móvil, Pantalla}

2.2.3.5 Recuperar las características divergentes

Entrada: Modelo de características.
Salida: Conjunto de características que no aparecen en todos los productos.

Por ejemplo, el conjunto de características divergentes del modelo de la figura 2.2 es :

{ Normal, Cámara, Frontal, Trasera, Radio, Auriculares, Táctil, Reconocimiento Escritura Manual}

2.2.3.6 Calcular el número de productos

Entrada: Modelo de características.
Salida: Número de productos del modelo de características.

Esta operación revela información sobre la flexibilidad y complejidad de la línea de productos software [7] [19] [40] [41]. Un gran número de productos potenciales una línea de productos más flexible y compleja. Por ejemplo, el número de productos en el modelo de características del modelo de la Figura 2.2 es 36.

2.2.3.7 Calcular la variabilidad

Entrada: Modelo de características.
Salida: Ratio entre el número de productos del modelo de características y el número de productos potenciales si todas las características pudieran combinarse entre sí sin restricciones.

El número de productos potenciales se define como $2^n - 1$, siendo n el número de características que se tienen en cuenta. La característica raíz no se suele tener en cuenta y en ocasiones sólo se consideran las características *hoja*. Esta operación puede ayudar en el análisis de los modelos de características ya que un factor alto de variabilidad indicará que la SPL es flexible, mientras que un factor bajo mostrará una SPL más compacta.

Por ejemplo, la variabilidad del modelo de características del modelo de la Figura 2.2 considerando todas las características salvo la raíz, es:

$$36/(2^9 - 1) \approx 0{,}07$$

2.2.3.8 Detección de características muertas (dead features)

Entrada: Modelo de características.
Salida: Conjunto de características muertas, si las hay.

Aunque estemos trabajando con un modelo de características no vacío (*void*), el modelo puede contener inconsistencias. Una de estas inconsistencias es la presencia de características muertas. Estas características son aquellas que no están presentes en ningún producto del modelo [19] [35] [36] [44]. Un modelo que contenga este tipo de características se considera que un modelo con errores. En la Figura 2.8 se muestran algunos casos de presencia de *dead features*.

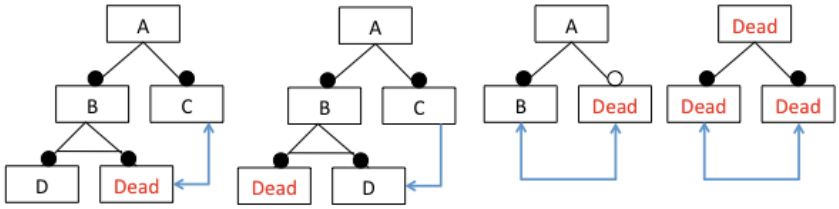


Figura 2.8 Casos de *dead features*

2.2.3.9 Proporcionar Explicaciones

Entrada: Modelo de características.
Salida: Explicación cuando el modelo contiene errores en general (es *void,* contiene *dead features…*)

Durante el análisis de un modelo de características, es posible que el modelo revele inconsistencias o errores. Las herramientas de análisis automático son capaces de detectar la presencia de este tipo de errores, pero la localización precisa del origen del error es un desafío importante y más aún si trabajamos con modelos extendidos. Es deseable que la herramienta nos proporcione el motivo del error con una explicación mínima.

Si observamos la Figura 2.9, nos encontramos con un modelo de características *void*. El motivo de esto es que ambas características F1 y F21 son *mandatory*, por lo cual ambas aparecerán en todos los productos, sin embargo, la relación r1 introduce una contradicción, excluyendo la aparición de ambas a la vez. Una posible explicación mínima sería decir que "El modelo de características está vacío debido a la relación r1".

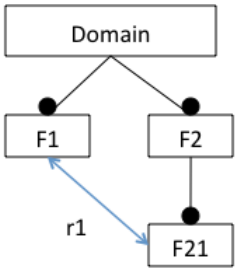


Figura 2.9 Casos de relación que causa *modelo* void

2.2.3.10 Proporcionar Explicaciones Correctivas

Entrada: Modelo de características.

Salida: Explicación correctiva cuando el modelo contiene errores en general (es *void,* contiene *dead features…*)

Cuando nos encontramos trabajando con un modelo de características con errores, es deseable que la herramienta de análisis automático, a parte de proporcionarnos una explicación acerca del origen del error, nos explique que medidad podemos tomar para la resolución del mismo.
Por ejemplo, en el caso de la Figura 2.9, una explicación correctiva sería "Borra la relación r1" o "Cambia las características F1 y F21 por opcionales".

Como comentábamos anteriormente, para poder operar de manera automática sobre nuestro modelo de características, necesitamos de una herramienta que implemente los *solvers* y las operaciones de análisis. En el siguiente apartado hablaremos de la herramienta FaMa.

2.2.4 El framework FaMa

FaMa (Feature Model Analyser) [4], es un framework abstracto que nos proporciona una base para el desarrollo de herramientas de análisis automático de líneas de producto. Ha sido desarrollado por un grupo de investigadores de la Universidad de Sevilla.

FaMa Tool [47], es la herramienta que hemos elegido para realizar el análisis semántico de nuestro caso práctico. En los siguientes apartados hablaremos de las razones que motivaron la creación de FaMa y cual es la funcionalidad que proporciona. De esta manera, sacaremos a la luz los motivos que nos llevaron a elegirla para nuestro análisis.

2.2.4.1 Características

Las razones que motivaron la creación de este framework, fueron que las soluciones existentes hasta el momento:

- Sólo se focalizaban en los modelos de características como posibilidad para modelar SPL (problema abstracción).
- No proporcionaban semánticas formales. Estas semánticas son necesarias para definir de manera rigurosa que son las SPL y los modelos de características y así evitar errores e inconsistencias.
- Ninguna permitía trabajar con modelos extendidos con atributos.
- No integraban distintos *solvers*. Algunos *solvers* realizan mejor que otros ciertas operaciones de análisis, por ello es deseable que se integre más de un tipo de *solver*.

FaMa se creó para solucionar todos estos problemas. En la Tabla 2.1 se muestra un resumen de las propuestas para el análisis automático de líneas de producto software. Esta tabla ha sido elaborada en base a la existente en [4], dónde se exponen todas las características de FaMa y cómo se creó.

Las propiedades estudiadas tienen el siguiente significado:

- Abstracción: se refiere al nivel de abstracción considerado para el análisis automático de SPL. El símbolo ✓ significa que el nivel de abstracción está a nivel de la SPL, mientras que ✗ significa que el nivel de abstracción está a nivel del modelo de características.
- Formalización: se refiere a las semánticas formales para los FM's. El símbolo ✓ significa que se proporcionan semánticas formales, mientras que ✗ significa que no se proporcionan.
- FM's extendidos: se refiere a la capacidad de analizar modelos de características extendidos. El símbolo ✓ significa que se permite trabajar con modelos extendidos, mientras que ✗ significa que no se puede trabajar con ellos.
- FM's básicos: se refiere a la capacidad de analizar modelos de características básicos. El símbolo ✓ significa que se permite trabajar con modelos básicos, mientras que ✗ significa que no se puede trabajar con ellos.
- Multi Solver: se refiere a la implementación de el análisis automático de FM's usando diferentes *solvers*. El símbolo ✓ significa que se propone más de un *solver*, mientras que ✗ significa que sólo se usa un *solver* o ninguno.

Tabla 2.1 Resumen de propuestas para el análisis de SPL's

	Batory [1]	Benavides et al. [7]	Schobbens et al [32]	Cao et al. [12]	Deursen y Klint [20]	Mannion [29]	Sun et al. [34]	Wang et al. [42]	Zhang et al. [44]	**FaMa [4]**
Abstracción	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Formalización	✗	✗	✓	✗	✗	✗	✗	✓	✗	✓
FM's	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓
FM's básicos	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Multi Solver	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

Además, en la Tabla 2.2 se resumen las operaciones que pueden efectuarse sobre modelos de características básicos por las distintas propuestas anteriormente mencionadas. Cada fila hace referencia a las operaciones estudiadas en el apartado 2.2.3. El símbolo ✓ significa que la operación ha sido propuesta, el símbolo ✎ representa que aunque la operación no ha sido propuesta, se considera que puede

realizarse, y el símbolo ✗ significa que la operación no se ha contemplado y que no se prevé ninguna forma de incluirla en la propuesta.

Tabla 2.2 Resumen de propuestas para el análisis de SPL's

	Batory [1]	Benavides et al. [7]	Schobbens et al [32]	Cao et al. [12]	Deursen y Klint [20]	Mannion [29]	Sun et al. [34]	Wang et al. [42]	Zhang et al. [44]	**FaMa [4]**
Producto válido	✓	✓	✓	✗	✗	✓	✓	✓	✎	✓
FM Void	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Todos los Productos	✓	✓	✎	✓	✓	✎	✓	✗	✎	✓
FM's equivalentes	✎	✎	✓	✎	✎	✎	✓	✎	✎	✓
Características comunes	✎	✎	✎	✗	✎	✎	✎	✗	✎	✓
Características divergentes	✎	✎	✎	✗	✎	✎	✎	✗	✎	✓
Núm. Productos	✎	✓	✎	✓	✓	✓	✎	✗	✎	✓
Variabilidad	✎	✓	✎	✎	✎	✎	✎	✗	✎	✓
Filtrado	✓	✓	✎	✗	✎	✎	✎	✎	✎	✓
Comunalidad	✎	✓	✎	✗	✎	✎	✎	✗	✎	✓
Optimización	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓
Características muertas	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Explicaciones	✓	✗	✗	✗	✗	✗	✓	✓	✗	✓
Explicaciones correctivas	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

Como podemos observar, la propuesta de FaMa es la más completa. Además, este framework puede extenderse para soportar otros modelos además de los modelos de características, para soportar más *solvers* y para implementar más operaciones de análisis.

2.2.4.2 Herramienta de análisis automático FaMa

La herramienta FaMa sirve para analizar líneas de productos software representadas como modelos de características. Asimismo, puede analizar cualquier sistema que pueda expresarse como un conjunto de características en una jerarquía (con o sin atributos). Además, FaMa integra las más prometedoras representaciones lógicas en el área del análisis automático de modelos de características; CSP [39], SAT [25], Choco y BDD [11]. Como comentábamos en el apartado anterior, más *solvers* pueden añadirse si es neceario ya que FaMa es extensible.

Puedes usar FaMa de cuatro formas diferentes:

- FaMa Shell: entorno para los usuarios finales. Consiste en una interfaz de línea de comandos dónde puedes cargar los modelos e invocar operaciones de análisis.
- FaMa Web Service: se proporciona un WSDL con la mayor parte de las operaciones FaMa. Puede integrarse en tu aplicación.
- FaMa OSGi: FaMa también está disponible como un conjunto de paquetes OSGi. OSGi, es una especificación java para la integración de servicios.
- FaMa Standalone: Además, FaMa está disponible como una librería extensible.

En nuestro caso, nos interesa usar FaMa en su versión OSGi. Nuestra herramienta extenderá a MULTIPLE, que es una herramienta extensible desarrollada con Eclipse. Eclipse dispone de un modulo de ejecución de configuraciones que permite a los desarrolladores implementar aplicaciones como un conjunto de paquetes (plugins) usando unos servicios e infraestructura comunes. Para proporcionar esta funcionalidad, Equinox implementa OSGi. Esta es la forma más dinámica y coherente de integrar FaMa en nuestro desarrollo. Como se integrará se explica en la Parte III de este documento.

En [47] se encuentra disponible un manual de instalación y utilización de FaMa para consumir FaMa en cualquiera de estas cuatro formas.

2.3 Conclusiones

En el presente capítulo, hemos hablado del estado del arte de los modelos de características. Repasando su estructura y las diferentes notaciones existentes.

La Tabla 2.3 resume las propuestas que dieron lugar a los elementos con los que actualmente contamos para la representación de los modelos de características. Aunque todos los conceptos fueron expuestos en la propuesta original de FODA, no se daba soporte para la notación gráfica de todos ellos. En primer lugar, nos encontramos con la propuesta original de FODA, que introdujo por primera vez una notación para las relaciones *mandatory*, *optional*, *alternative* y las restricciones *excludes* y *requires*. Posteriormente, fue en la propuesta feature-RSEB de Griss en la que se incluyó la nueva relación de tipo OR. Aunque es difícil concretar el momento exacto de la introducción del concepto de atributo en los modelos de características, propuestas como [13] fueron las primeras en la literatura en nombrar la necesidad de estos elementos para ampliar la

expresividad de los modelos. Lo mismo pasa con el concepto de cardinalidades. En el 2002, ya existen trabajos en los que se introduce la necesidad de incorporar multiplicidades a los modelos [15] [30] [31], aunque no es hasta el 2004 [17] [18], cuando se presentan los llamados modelos basados en cardinalidades.

En la siguiente tabla, la primera fila representa el año en el que se publicaron las propuestas que se especifican el la segunda línea. Mientras que en la tercera línea encontramos los elementos que estas propuestas introdujeron.

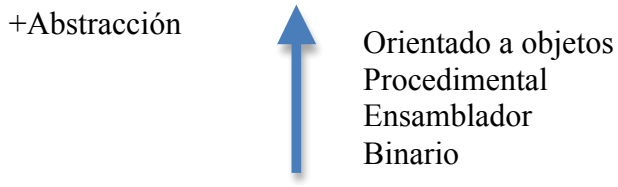
Tabla 2.3 Resumen de relaciones presentes en las diferentes notaciones para FM

Año	1990	1998	2001	2002-2004
Propuesta	FODA [27]	Feature-RSEB [24]	Modelos con atributos [13]	Modelos basados en cardinalidades [15] [30] [31] [17] [18]
Elementos introducidos	Mandatory Optional Alternative Requires Excludes	OR	Atributos	Cardinalidades

Además, hemos hablado de cómo se analizan estos modelos, aportando nuestra visión sobre la necesidad de analizarlos en forma (sintácticamente) cuando no se usan herramientas adecuadas y repasando las propuestas existentes para su análisis de contenidos (semántico). Por último, hemos hablado del nacimiento y funcionalidad ofrecida por la herramienta de análisis automático FaMa, la cual utilizaremos posteriormente.

3. Ingeniería Dirigida por modelos

El crecimiento de la complejidad del software, la no reutilización del conocimiento tecnológico ni del dominio y la necesidad de industrializar el desarrollo software, fueron la principal motivación de la aparición de esta tecnología. Nos encontramos con que el nivel de abstracción de los lenguajes va en aumento.



Ya en los años 70, empezaron a surgir proyectos que analizaban la relación existente entre los requisitos de un problema y las necesidades que éstos generaban. Aunque fue en 1984 cuando surge la primera herramienta CASE (Computer Aided Software Engineering). Las herramientas CASE tenían entre sus objetivos:

- Aumentar la calidad del software
- Mejorar la productividad en el desarrollo y mantenimiento del software
- Reducir el tiempo y coste de desarrollo y mantenimiento de los sistemas informáticos
- Mejorar la planificación de un producto
- Automatizar el desarrollo del software, la documentación, la generación de código, las pruebas de errores y la gestión del proyecto.
- Ayuda a la reutilización del software, portabilidad y estandarización de la documentación
- Gestión global en todas las fases de desarrollo de software con una misma herramienta.

Estas herramientas tocaron techo en los años 90, siendo los *mainframes* cada vez menos utilizados dando paso a una mayor utilización de herramientas más específicas para cada fase del ciclo de vida del software. El motivo de esta falta de éxito lo encontramos en las limitaciones de los procesos de traducción que trasladaban las representaciones gráficas de los sistemas (mediante lenguajes gráficos de propósito general) a una plataforma o tecnología específica. En [26], se repasan los motivos por los que estos tipos de herramientas dejaron de usarse.

El enfoque de la Ingeniería Dirigida por Modelos (MDE) tiene por objeto aumentar la productividad mediante la maximización de la compatibilidad entre los sistemas (a través de la reutilización de modelos estandarizados). Esta propuesta promueve el uso de modelos como los artefactos principales del desarrollo. De esta manera, el proceso de desarrollo software se convierte en un proceso de refinación y transformación entre modelos.

Este paradigma simplifica el proceso de diseño (a través de modelos de patrones de diseño que se repiten en el dominio de aplicación), y promover la comunicación entre

los individuos y equipos que trabajan en el sistema (a través de una estandarización de la terminología y las mejores prácticas utilizadas en el dominio de aplicación). MDA

Pero para poder aplicar las propuestas promovidas por MDE era necesario contar con lenguajes de modelado y una metodología de desarrollo además de los estándares necesarios que permitieran la interoperabilidad.

Con motivo de abordar los problemas en el contexto de MDE, el Object Management Group (OMG) [52] lanzó en el año 2000, la iniciativa Model Driven Arquitecture (MDA) [49].

MDA nació como una nueva forma de desarrollar aplicaciones y escribir especificaciones. Una especificación MDA completa consiste en un modelo base independiente de plataforma (PIM), uno o más modelos específicos de plataforma (PSM) y colecciones de definiciones de interfaz, cada una describiendo cómo está implementado el modelo base en las diferentes plataformas *middleware.*

Una aplicación MDA completa consta de un PIM definitivo, uno o más PSMs y de implementaciones completas, una en cada plataforma a la que el desarrollador de la aplicación decida dar soporte.

El desarrollo MDA se centra en primer lugar en la funcionalidad y comportamiento de una aplicación o sistema distribuido. De este modo, MDA separa los detalles de implementación de las funciones de negocio. En otras palabras, separa y relaciona PIMs y PSMs usando técnicas de transformación. Así no será necesario repetir el proceso de definir una aplicación o la funcionalidad de un sistema y su comportamiento cada vez que surja una nueva tecnología. Otras arquitecturas están ligadas a una tecnología en concreto. Con MDA, la funcionalidad y el comportamiento son modeladas una sola vez.

Transformar un PIM a uno o varios PSM en las plataformas soportadas por MDA se implementa mediante herramientas, facilitando la tarea de dar soporte a distintas o nuevas tecnologías. MDA define un gran número de estándares de OMG.

3.1 Estándares OMG

El OMG ha adoptado un número de tecnologías, que juntas hacen posible un enfoque dirigido por modelos. Estas incluyen MOF, UML, OCL, QVT, XMI, etc.

Los conceptos que promueve MDA pueden llevarse a cabo sin hacer uso de los estándares que promueve el OMG. Pero para que MDA se use de forma productiva es necesario tener una serie de estándares relacionados para el modelado. De esta forma, será más fácil para la industria desarrollar herramientas y posibilitar la interoperabilidad entre las soluciones MDA y las herramientas.

3.1.1 MOF

MOF (*Meta Object Facility*) define una serie de conceptos de modelado que el modelador usará para poder definir y modelar un conjunto de metamodelos interoperables. Además MOF está en si mismo definido en MOF y se encarga de capturar la estructura y semántica de los metamodelos, en particular de los modelos UML. Sí estás familiarizado con UML, los conceptos más importantes de MOF también te serán familiares.

MOF ofrece cinco conceptos importantes que podemos usar para definir un modelo: tipos (clases, tipos primitivos y enumeraciones), generalizaciones, atributos, asociaciones y operaciones.

Los metamodelos y modelos se suelen organizar en una estructura de cuatro capas (Figura 5.1) M3-M0 con la siguiente distribución:

- En la capa inferior, llamada M0 es dónde se encuentran las instancias reales. Estas instancias son, por ejemplo, un usuario llamado "Pepito" que vive en la calle "Colón 1".
- En la capa M1 están contenidos los modelos, por ejemplo un UML de un sistema software. En este nivel se definiría, siguiendo con el ejemplo anterior, el concepto de *Usuario* con las propiedades *nombre* y *calle.* Los elementos de esta capa especifican la apariencia de las instancias de M0.
- En la capa M2 residen los metamodelos, que definirán cómo serán los modelos de la capa M1.
- En la capa más superior (M3), define el lenguaje con el que se definirán los metamodelos. El tipo de relación que mantiene M2 con M3 es la misma que M0 con M1 y M1 con M2, cada elemento de M2 es una instancia de un elemento definido en M3.

La Figura 3.1 muestra una vista panorámica sobre el contenido de cada capa de esta arquitectura.

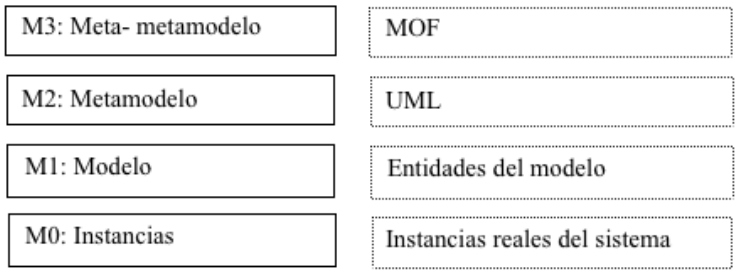


Figura 3.1 Contenido de las capas de MOF

3.1.2 OCL

OCL es un lenguaje formal que pueden utilizan los modeladores para expresar condiciones que deben cumplirse para el sistema que se está modelando. OCL juega un papel muy importante en las especificaciones detalladas de muchos elementos de UML.

Constituye un lenguaje de expresión puro, de manera que cuando una expresión OCL es evaluada, devuelve un valor puro que no causa ningún efecto colateral.

Tres conceptos de OCL constituyen lo que llamamos una regla bien formada:

- **Precondición:** Condición que debe cumplirse en el momento en que la ejecución de una operación está a punto de empezar.
- **Invariante:** Condición que debe cumplirse para todas las instancias de un elemento del modelo en particular.
- **Poscondición:** Condición que debe cumplirse en el momento en el que la ejecución d una operación ha terminado.

Las expresiones OCL normalmente aparecen en diagramas UML dentro de notas o de las definiciones de clases.

Existen muchas buenas razones para usar OCL en modelos relacionados con MDA:

- No son restricciones ambiguas, lo que es siempre bueno de cara a mejorar el entendimiento del modelo y la habilidad de los miembros del equipo para comunicarse.
- Las restricciones añaden:
    - precisión al modelo.
    - información sobre los elementos del modelo y sus relaciones; esta información nos sirve como documentación del sistema a modelar.

OCL en MDA

Cabe destacar el papel de OCL en la definición de transformaciones. Una transformación realiza la traza desde uno o más elementos del modelo fuente a uno o más elementos en el modelo objetivo. Una pregunta OCL especifica los elementos en el modelo fuente de una transformación, mientras que una expresión OCL específica los elementos en el modelo objetivo.

Muchas transformaciones pueden ser únicamente aplicadas bajo ciertas condiciones, Éstas, pueden expresarse también en OCL. Una condición se encuentra en los elementos fuente a ser trasformados y otra en los elementos objetivo. Todas las expresiones OCL usadas en una definición de transformación se especifican en el metamodelo de los lenguajes fuente y objetivo.

3.1.3 QVT

Query/View/Transformation, en la arquitectura dirigida por modelos, es un estándar para la transformación de modelos definido por el OMG.

3.1.3.1 Los lenguajes QVT

La especificación QVT (Figura 3.2) comprime actualmente tres lenguajes (Operational Mapping, Relations y Core) y tiene una naturaleza híbrida entre declarativa e imperativa:

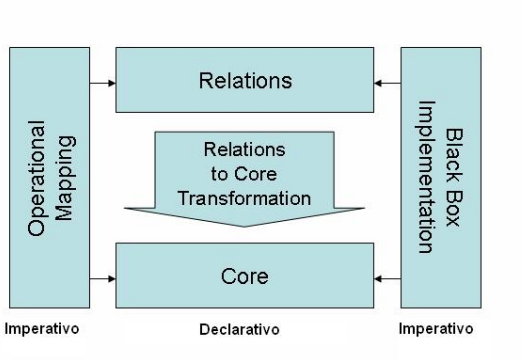


Figura 3.2 Arquitectura QVT

Las partes declarativas de está especificación están estructuradas en una arquitectura de dos capas:

1. Un metamodelo *Relations* conocido y un lenguaje que soporta *pattern-matching* de objetos complejos y creación de plantillas de objetos. Las trazas entre los elementos del modelo involucrados en la transformación se crean implícitamente.
2. Un metamodelo *Core* y un lenguaje definido usando extensiones mínimas de EMOF y OCL. Las clases de trazabilidad se definen como modelos MOF, y la creación y eliminación de una instancia de clase de trazabilidad se define como la creación y eliminación de otro objeto cualquiera.

El lenguaje *Relations*

Una especificación declarativa de la relación entre los modelos MOF. Soporta *pattern-matching* de objetos complejos y crea de manera implícita clases traza y sus instancia para registrar que ocurre durante la ejecución de la transformación.

El lenguaje *Core*

Lenguaje sencillo que sólo soporta el ajuste de patrones *(pattern-matching)* sobre un conjunto plano de variables, evaluando las condiciones sobre esas variables de un conjunto de modelos.

Core trata de manera simétrica todos los elementos del modelo de origen, destino y trazabilidad. Es igual de potente que Relations y dada su simplicidad, sus semánticas pueden definirse fácilmente. Por el contrario, la descripción de transformaciones es muy extensa. Además, los modelos de trazabilidad deben definirse explícitamente y no se deducen de la descripción de transformación como se hace en Relations.

Implementaciones Imperativas

Existen dos mecanismos para la invocación de implementaciones imperativas de transformaciones de Relations Core: *Operational Mappings* e implementaciones *Black-Box MOF Operation.* Cada relación define una clase que será instanciada para la traza entre los elementos del modelo a transformar.

El lenguaje *Operational Mapping*

Nos proporciona de forma estandarizada implementaciones imperativas. También se usa cuando es difícil que una o más relaciones nos proporcionen una especificación puramente declarativa.

Implementación Black-Box

Las operaciones MOF deben poder derivarse de Relations haciendo posible integrarlas con cualquier implementación de una operación MOF con la misma signatura. Esto resulta beneficioso por las siguientes razones:

- Permite codificar algoritmos complejos en cualquier lenguaje de programación con correspondencia a MOF.
- Permite el uso de librerías específicas de dominio para calcular ciertas propiedades del modelo. Por ejemplo, en matemáticas, en ingeniería y en otros muchos campos existen grandes librerías que almacenan algoritmos específicos de dominio, que serían muy difíciles de expresar usando OCL.
- Permite ocultar la implementación de algunas partes de una transformación.

No obstante, dicha integración puede ser peligrosa dado que la implementación asociada a la integración tiene acceso libre a las referencias de objetos en los modelos.

Cada Black-box debe implementar explícitamente una relación que sea responsable de conservar la trazabilidad entre los elementos del modelo relacionado con la implementación de la operación MOF.

Con motivo de extender esta analogía con la arquitectura Java, la habilidad para invocar implementaciones «Black-Box» y «Operational Mappings» puede considerarse equivalente con la llamada a la Java Native Interface (JNI).

3.2 Soporte Tecnológico a la Ingeniería Dirigida por Modelos

3.2.1 Eclipse

La plataforma Eclipse ha sido diseñada para construir entornos de desarrollo integrados (IDEs) [53] que pueden usarse para crear aplicaciones muy diversas; páginas web, programas Java, programas C++, Enterprise JavaBeans…

Aunque la plataforma Eclipse incorpora mucha funcionalidad, esta funcionalidad es muy genérica. Eclipse permite la incorporación de herramientas para extender la plataforma con nuevos contenido, hacer nuevas cosas con los contenidos existentes y focalizar dicha funcionalidad genérica en algo más específico.

Eclipse emplea un mecanismo para descubrir, integrar y ejecutar módulos que llamamos plug-ins. Podemos implementar estos componentes con funcionalidad que se ‘enchufarán’ a nuestra plataforma. La plataforma Eclipse misma, y las herramientas que

la extienden se componen de plug-ins. Los plug-in son componentes estructurados que se configuran utilizando un archivo de manifiesto (plugin.xml) en el sistema de ejecución de Eclipse. El diagrama siguiente (Figura 3.3) muestra los cinco componentes principales de la plataforma Eclipse:

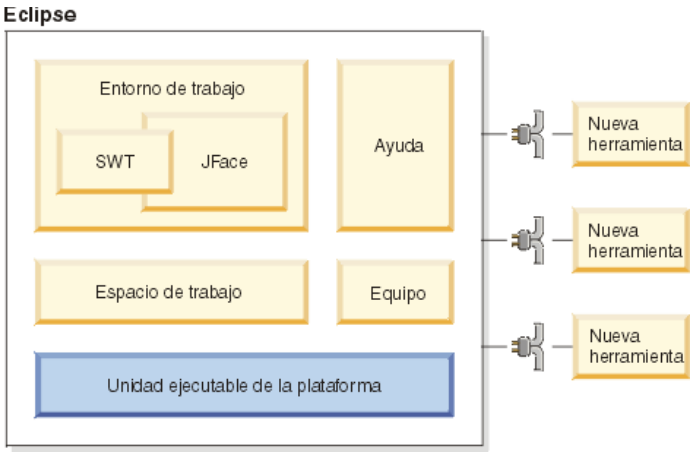


Figura 3.3 Arquitectura de la plataforma Eclipse

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

Eclipse es también una comunidad de usuarios, extendiendo constantemente las áreas de aplicación cubiertas. Un ejemplo es el recientemente creado Eclipse Modeling Project, cubriendo casi todas las áreas de Model Driven Engineering. Los principales frameworks que constituyen este proyecto (Eclipse Modeling Framework y Graphical Modeling Framework) se describen a continuación,

3.2.2 Eclipse Modeling Framework (EMF)

Eclipse Modeling Framework [54] es un Framework Java y una utilidad para la generación de código para el desarrollo de herramientas y otras aplicaciones basadas en modelos estructurados. EMF nos ayuda a convertir nuestros modelos en código Java eficiente, correcto y fácilmente configurable.

Generalmente, cuando hablamos de modelado, nos referimos a cosas como diagramas de clase, diagramas de colaboración, diagramas de estado, etc. UML define una notación estándar para esta clase de diagramas. Usando una combinación de diagramas UML, se puede especificar un modelo completo de una aplicación. Este modelo se usará únicamente como documentación o, si contamos con las herramientas adecuadas, podrá ser usado como entrada para generar parte de la aplicación.

Esta manera de modelar requiere herramientas de análisis y diseño orientado a objetos (OOA/D) muy caras. EMF nos proporciona un bajo coste de entrada. Esto es porque un modelo EMF sólo requiere un pequeño subconjunto del tipo de cosas que

puedes modelar en UML, en concreto, definiciones simples de las clases, sus atributos y sus relaciones, no siendo necesaria una herramienta de modelado gráfico completa.

A pesar de que EMF usa XMI (XML *Metadata Interchange*) como su forma canónica de definir modelos, podemos persistir nuestro modelo en este tipo de diferentes formas:

- Creando un documento XMI directamente, usando un editor de texto o de XML
- Exportando el documento XMI desde una herramienta de modelado como Rational Rose, por ejemplo
- Interfaces Java anotadas con propiedades de modelo
- Usando XML *Schema* para describir la forma del modelo serializable

Una vez hemos especificado nuestro modelo en EMF, el generador de EMF crea una serie de clases de implementación Java. Podemos editar estas clases para añadir métodos e instancias y seguir regenerando desde el modelo como se necesite: las adiciones realizadas se conservarán. Si el código añadido depende de información que hemos cambiado en el modelo, seguiremos necesitando actualizar el código para que los cambios se reflejen.

Además de incrementar la productividad, usar EMF para construir una aplicación, nos proporciona más ventajas , como:

- Notificación de cambios en el modelo
- Soporte a la persistencia incluyendo XMI y serialización XML
- Un Framework para la validación del modelo
- Una API muy eficiente para la manipulación de objetos EMF.
- Provee de fundamentos para la interoperabilidad con otras herramientas y aplicaciones EMF

EMF comenzó como una implementación de la especificación MOF y fue evolucionó con la experiencia derivada de su uso. EMF puede verse como una implementación Java muy eficiente de un subconjunto de la API MOF. A pesar de esto, para evitar confusiones, el metamodelo principal basado en MOF en EMF se llama Ecore,

Para definir un modelo deberemos disponer de una terminología común para describirlo. Y lo que es más importante, para implementar las herramientas de EMF y el generador se requiere un modelo para la información.

Ecore

El modelo empleado para representar modelos en EMF se denomina Ecore. Ecore es también a su vez un modelo EMF, esto implica que Ecore es su propio metamodelo (o expresado en otras palabras, Ecore es un meta-metamodelo).
Gracias a Ecore, es posible definir los vocabularios locales de dominio que permiten el trabajo con modelos en distintos contextos.

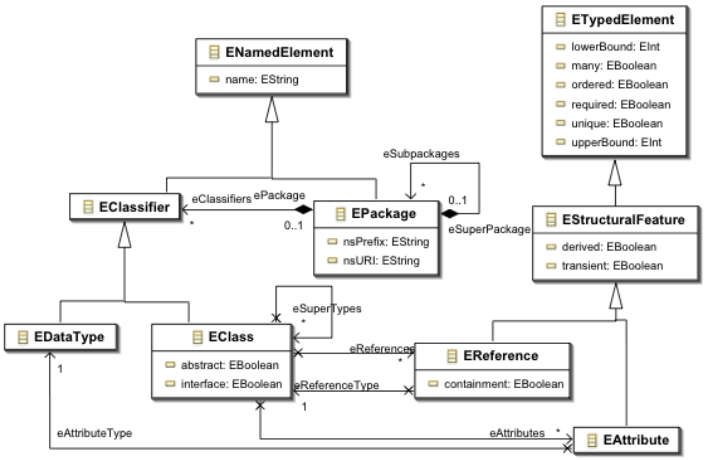


Figura 3.4 Metamodelo Ecore simplificado

Ecore es un vocabulario diseñado para permitir la definición de cualquier tipo de metamodelos. Para ello, proporciona elementos útiles para describir conceptos y las relaciones entre ellos. En la Figura 3.4 se muestra un subconjunto simplificado este modelo.

Las similitudes de EMF con UML son evidentes, y es que EMF es un subconjunto de MOF, el cual a su vez está basado en los elementos del diagrama de clases de UML.

La cuestión de porqué no se ha utilizado UML como lenguaje de modelado es sencilla: Ecore es un subconjunto pequeño y simplificado de UML. UML soporta un modelado mucho más ambicioso que el soporte básico que se proporciona en EMF. Por ejemplo, UML permite modelar el comportamiento de una aplicación, a parte de su estructura de clases.

3.2.3 Graphical Modeling Framework

GMF surge de la necesidad por parte de los desarrolladores de tener que usar frecuentemente los frameworks EMF y GEF de Eclipse para el desarrollo de sus herramientas basadas en modelos. GEF (Graphical Editing Framework) es el framework de Eclipse que permite a los desarrolladores crear editores gráficos ricos a partir de un modelo de aplicación. GEF está formado por dos plugins. El plugin org.eclipse.draw2d proporciona las herramientas para renderizar y ordenador los elementos al mostrar los gráficos. El framework GEF emplea una arquitectura de «modelo-vista-controlador».

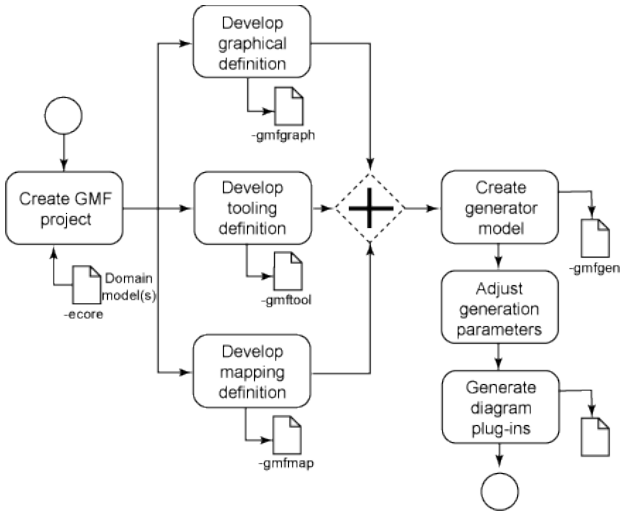


Figura 3.5 Proceso creación edito gráfico con GMF

La Figura 3.5 ilustra los principales componentes y modelos usados durante un desarrollo basado en GMF. En la parte central de la figura se observa que un proyecto GMF parte de 3 modelos: *Domain Model, graphical Definition y Tooling Definition*. El primero de ellos, se corresponde con el modelo EMF para el que deseamos crear el nuevo editor gráfico. El segundo describe cuáles serán las primitivas gráficas que se dibujarán en el entorno de ejecución basado en GEF pero sin definir ningún tipo de correspondencia con los elementos del modelo dominio para los que van a proporcionar capacidades de representación y edición. El tercer modelo permite definir las herramientas que se mostrarán en la paleta de dibujo del editor así como otros elementos de la interfaz gráfica (menús, barras de herramientas, etc.).

Generalmente, una definición gráfica puede ser igualmente válida para diferentes dominios. Por ejemplo, en el diagrama de clases de UML encontramos diferentes elementos que son extremadamente parecidos en su apariencia y estructura. Un objetivo de GMF es que una definición gráfica pueda ser reutilizada por distintos dominios. Esto se consigue mediante un modelo separado llamado *Mapping Model* que permite enlazar los elementos gráficos y las definiciones de herramienta con los elementos deseados del modelo dominio.

Una vez se han definido los enlaces apropiados, GMF proporciona un modelo generador para permitir afinar los últimos detalles de implementación para la fase de generación automática de código. La obtención del plugin de un editor basado en un modelo generador obtendrá un último modelo, llamado modelo notacional. El entorno de ejecución de GMF es el que enlaza este modelo notacional con el modelo dominio cuando el usuario está trabajando con un diagrama. A su vez, éste también proporciona las capacidades de persistencia y sincronización para ambos.

3.2.4 Medini QVT

Medini QVT es una herramienta para transformaciones de modelo a modelo desarrollada por la empresa *ikv++ technologies ag* [48]. El motor principal de esta herramienta implementa el estándar QVT-Relations definido por el OMG. Además, incluye herramientas para un desarrollo conveniente de las transformaciones, como un

depurador gráfico y un editor. Estos componentes están disponibles de manera libre para usos no comerciales.

3.2.4.1 Características

Las principales características de la herramienta completa son:

- Ejecución de transformaciones QVT expresadas en la sintaxis concreta textual del lenguaje QVT-Relations
- Integración en Eclipse
- Editor con asistente de código.
- Depurador para trazar la ejecución de transformaciones paso a paso a través de las reglas.
- Implementación del concepto de clave (*key*) del lenguaje QVT, permitiendo las transformaciones para actualizaciones incrementales.
- Transformaciones con n-dominios participantes.
- Transformaciones bidireccionales.

Toda esta funcionalidad está implementada en diversos plugins de Eclipse. En este sentido, debe aclararse que es únicamente el plugin que contiene el motor de ejecución de transformaciones el que es público y de código abierto. Todos los plugins que implementan funcionalidad adicional (editor textual, depuración, etc.) son de código cerrado.

4. Modelado de características basado en cardinalidades e Ingeniería Dirigida por Modelos

Este capítulo resume el trabajo realizado por Abel Gómez e Isidro Ramos expuesto en [22] [23]. En él, se integran los modelos de características en el contexto de la Ingeniería Dirigida por Modelos (MDE), propiciando un debate sobre las principales cuestiones que surgen cuando se intenta usar los modelos de características en procesos MDE y como superarlos. Además, se proporciona una herramienta de modelado y metamodelado llamada MULTIPLE, que permite a los desarrolladores de líneas de producto software definir, usar y explotar los modelos de características. En este capítulo, también se pretenden resaltar las oportunidades de extensión de la herramienta en materia de análisis de modelos (*model-checking*) de características.

En la sección 4.1 hablaremos de los motivos que llevaron al desarrollo de MULTIPLE, en el apartado 4.2 y 4.3 se habla de cómo se ha desarrollado esta herramienta y en el apartado 4.4 comentaremos que aspectos pueden extenderse sobre esta propuesta.

4.1 Motivación

El modelado de características consiste en el uso de diagramas que utilizan una notación específica para representar las partes variables y comunes de las líneas de producto software. En este contexto, la Arquitectura Dirigida por modelos (MDA) [49] propuesta por el OMG promueve una serie de estándares de metamodelado que podemos utilizar para la explotar el uso de modelos de características.

Estándares como MOF y QVT, explicados en el capítulo anterior, nos permiten definir modelos e instanciarlos. Estos modelos pueden ser modelos de características y sus instancias configuraciones del mismo.

Ya existen frameworks, como EMF, que proporcionan la base suficiente para explotar estos modelos de características en un proceso de Ingeniería Dirigida por Modelos. Sin embargo, estos frameworks presentan limitaciones cuando trabajan con artefactos software en las diferentes capas de abstracción.

Una de las limitaciones observadas se deriva del uso de modelos extendidos. Como explicábamos en el apartado 2.1.2, estos modelos de características extendidos asocian una cardinalidad a cada característica y grupo de características. También pueden especificar un atributo para éstas. Además, pueden definirse restricciones *cross-tree* entre estas características. En FODA, estas restricciones se expresaban mediante fórmulas proposicionales. Sin embargo, Czarnecki [19] expone que cuando estamos trabajando con modelos basados en cardinalidades, su interpretación mediante estas fórmulas proposicionales no es muy adecuada ya que podemos tener varias copias de

una misma característica. Así que es necesario el uso de lenguajes más expresivos para definir las restricciones entre las características y razonar sobre el modelo y estas restricciones. Además, también es necesaria una mayor expresividad para trabajar con variables tipadas cuyos valores pueden carecer de cardinalidad para representar más fácilmente los tipos de atributo.

Otro problema surge de la definición de configuraciones. Tradicionalmente una configuración es *un grupo de características seleccionadas de un modelo de características que no violan ninguna de las restricciones definidas en él*. De esta definición, se deduce que una configuración es una copia más que una instancia. Esta interpretación puede llevar a confusiones en el caso de los modelos basados en cardinalidades con atributos. Es más adecuado considerar una configuración como una instanciación y no como una copia.

4.2 Juntando el modelado de características y la ingeniería dirigida por modelos

El estándar MOF (Meta Object Facility), que proporciona soporte para el metamodelado, define una estricta clasificación de los artefactos software en una arquitectura de cuatro capas. Como veíamos en el apartado 3.1.1, el lenguaje MOF se encuentra en la capa M3 o capa del meta-metamodelo.

Como MOF proporciona soporte para el modelado y el metamodelado, podemos definir modelos de características definiendo su metamodelo. De esta forma, somos capaces de definir modelos que capturen toda la variabilidad del dominio, y así especificar configuraciones del modelo. Además, podemos usarlo para definir transformaciones basadas en modelos que permiten el uso de modelos de características y sus configuraciones en otros procesos complejos. La figura Figura 4.1 muestra en qué parte de la arquitectura de cuatro capas de MOF encajan los modelos y sus configuraciones.

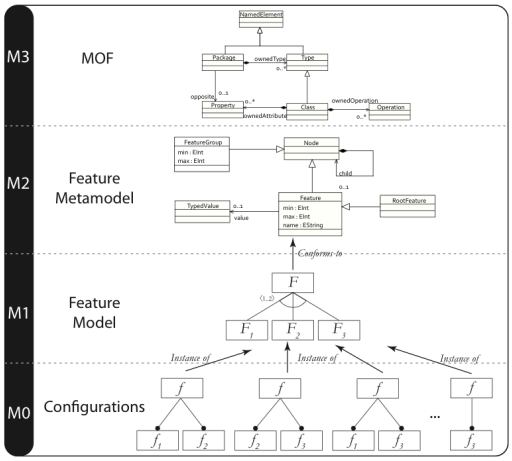


Figura 4.1. Definición y configuración de modelos de características en el contexto de MOF

4.3 Herramienta basada en un modelo de cardinalidades

La herramienta presentada en el trabajo de Gómez [23], desarrollada usando Eclipse Modeling Framework, permite automatizar varios pasos que nos permitirán ajustar un modelo de características que puede ser usado para desarrollar una SPL en el contexto de MDA. Esta herramienta proporciona:

- Soporte gráfico para definir una variante de modelos de características basado en cardinalidades con restricciones descritas usando un lenguaje de restricciones.
- Soporte para generar automáticamente Modelos de Variabilidad de Dominio (DVMs) a partir de FMs, que contienen toda la variabilidad del dominio de aplicación (incluyendo restricciones complejas). En [22] encontramos una explicación detallada.
- Soporte para transformar restricciones de modelo en expresiones OCL.
- Editor de configuraciones, con asistente para desarrolladores.
- Capacidades para comprobar la consistencia de una configuración sobre su modelo de características usando motores OCL preconstruidos.

Todas estas tareas están soportadas usando técnicas MDE (modelado, metamodelado, transformaciones de modelos y generación de código).

4.3.1 Metamodelo de características basado en cardinalidades

La base de este trabajo es el metamodelo de características basado en cardinalidades. Ha sido definido teniendo en cuenta que cada elemento tendrá una representación gráfica diferente. De esta manera, es posible generar automáticamente el editor gráfico para dibujar modelos de características basados en este metamodelo.

En la Figura 4.2, un modelo de características se representa mediante la clase *FeatureModel*. Un modelo de características puede verse como un grupo de características (*Features*), el conjunto de sus relaciones (*Relationships*) y una serie de restricciones (*modelConstraints*) que se aplican sobre él. Además, el modelo a de tener una característica raíz (*rootFeature*).

En la Figura 4.2 podemos ver como se representan de manera explícita las relaciones entre las características. Así, se representa de manera uniforme, las relaciones jerárquicas (clase *StructuralRelationship*) y las restricciones entre las características (*RestrictionRelationship*).

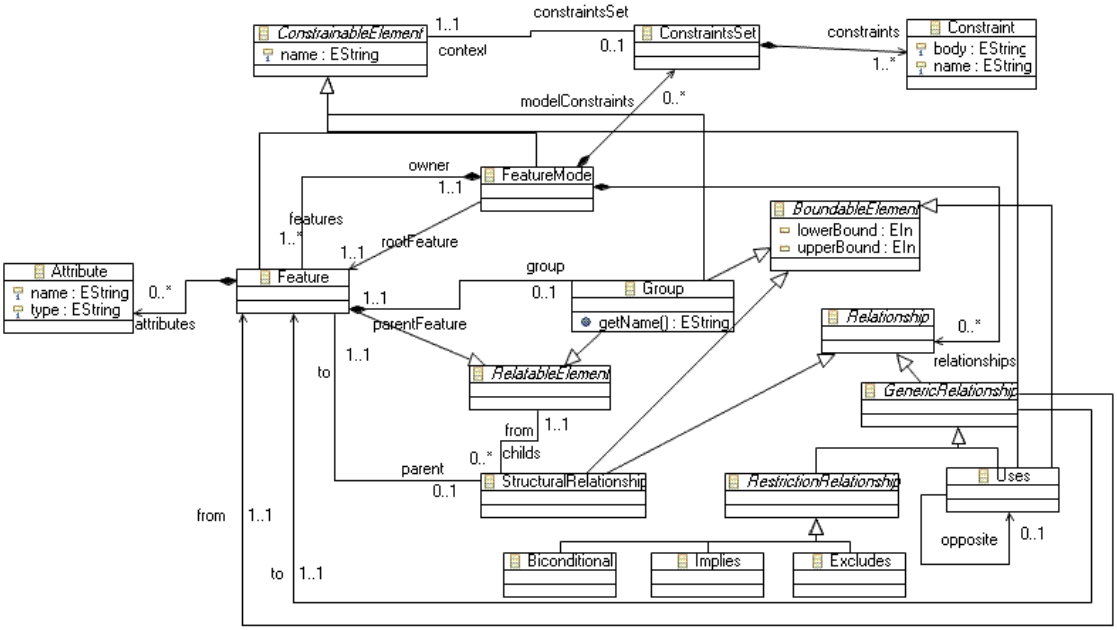


Figura 4.2 Metamodelo de características basado en cardinalidades

4.3.1.1 Restricciones del modelo de características

En el modelado de características, es común tener la posibilidad de definir restricciones para describir de manera precisa, qué configuraciones se considerarán válidas. Normalmente, estas restricciones estas restricciones se describen a través de las relaciones de implicación o exclusión. Este tipo de relaciones son las descritas en el metamodelo como relaciones *binarias y horizontales.*. Estas relaciones se definen entre dos características y expresan una restricción (coimplicaciones, implicaciones y exclusión) o dependencias (*use*).

- *Implication (A —> B)*. Si existe una instancia de la característica A, al menos una instancia de B existe también.
- *CoImplication (A <—> B)*. Si existe una instancia de la característica A, al menos una instancia de B existe también y viceversa.
- *Exclusion (A x—x B)*. Si existe una instancia de la característica A, no pueden existir instancias de B y viceversa.
- *Use (A - - ->B)*. Esta relación se definirá a nivel de configuraciones, especificando que una instancia en concreto de la característica A, estará relacionada con una (o más) instancias especificas de la característica B según el valor de su cota superior (n).

Además de estas relaciones que describen restricciones a groso modo, el metamodelo de esta propuesta proporciona capacidades para describir restricciones más precisas. Estas restricciones se almacenan en una instancia *ConstraintSet,* y pueden aplicarse a cualquier subclase de la clase abstracta *ConstrainableElement* (*FeatureModel, feature, Group* o *Uses*). Estas restricciones se expresan como texto (atributo *body* de la clase *Constraint*).

En el trabajo de Gómez, se describe también un lenguaje de definición de estas restricciones más precisas llamado *Feature Model Constraint Language* (FMCL), cuya

sintaxis se basa en OCL y cuya semántica se define por un conjunto de patrones que describen equivalencias entre expresiones FMCL y OCL.

4.3.1.2 Editor de modelos de características basados en cardinalidades

Dado que este trabajo utiliza una aproximación de desarrollo de software dirigida por modelos (MDSD), se puede generar un editor gráfico de manera automática a partir del metamodelo definido. Para la obtención del editor gráfico se ha usado Graphical Modeling Framework (GMF). La Figura 4.3 muestra la apariencia del editor.

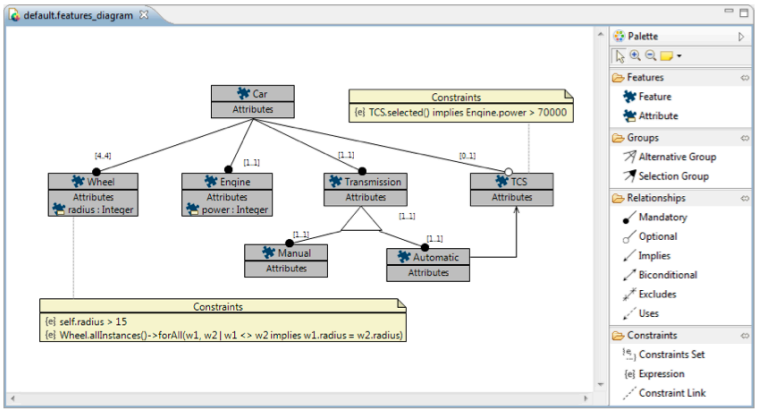


Figura 4.3 Ejemplo del editor de modelos de características basado en cardinalidades

4.4 Conclusiones y oportunidades de extensión de MULTIPLE

En el trabajo realizado por Gómez y Ramos en [22] [23], se presenta un framework para definir y usar modelos de características en un proceso MDE. Esta tecnología resuelve:

- La incapacidad actual de las herramientas de metamodelado de trabajar con artefactos ubicados en cualquiera de las capas MOF.

- La complejidad de definir restricciones en modelos de características dónde las características pueden clonarse y tener atributos.

La herramienta MULTIPLE nos ofrece un buen *background* para explotar nuestros modelos de características, proporcionándonos un editor gráfico y posibilidad de definir restricciones en modelos extendidos. Sin embargo, le queda una tarea pendiente; el análisis de los modelos de características. Esta herramienta comprueba la corrección de las configuraciones definidas en base a las restricciones descritas, pero no valida el contenido del modelo.

Sí que implementa un plugin, que contiene un proyector manual escrito en Java (código en Anexo A), que crea, a partir de los modelos definidos en MULTIPLE, un modelo analizable por la herramienta de análisis automático FaMa [47].

Con la funcionalidad proporcionada por MULTIPLE, para analizar el contenido de nuestro modelo de características, teníamos que seguir el siguiente proceso:

1. Invocar manualmente al proyector para crear el archivo legible por fama con la información de nuestro modelo. Esta tarea se realiza mediante el menú contextual definido por el plugin (Figura 4.4).

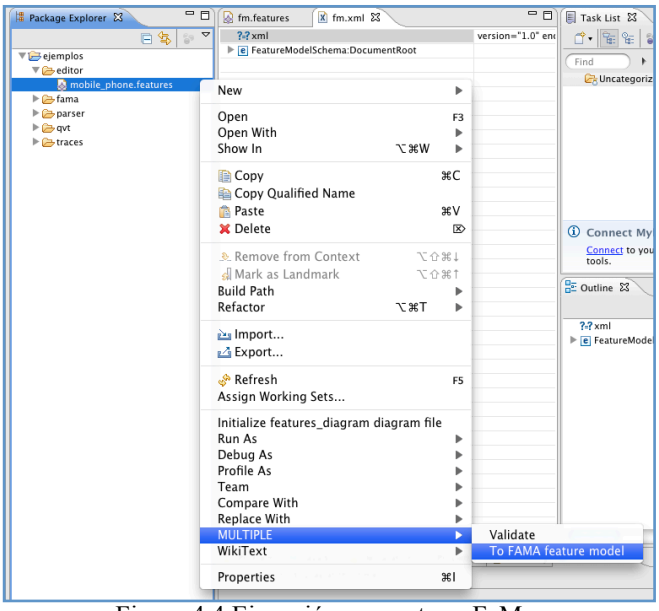


Figura 4.4 Ejecución proyector a FaMa

Obtenemos el archivo plano con extensión .fm (Figura 4.5), uno de los formatos aceptados por FaMa.

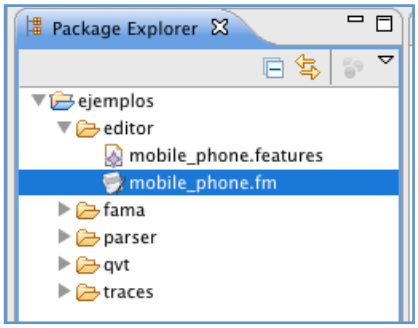


Figura 4.5 Obtención del archivo .fm

2. Descargar la herramienta FaMa [47] y llamarla desde la línea de comandos. Una vez descargada la herramienta, la ejecutamos mediante el comando java –jar famaSDK-v.jar como se observa en la Figura 4.6. La v corresponde con la versión de la herramienta que estemos utilizando.

3. Cargar el modelo (*load mobile_phone.fm* en la Figura 4.6) y ejecutar las operaciones de análisis como comandos del shell (comando *products* de la Figura 4.6).

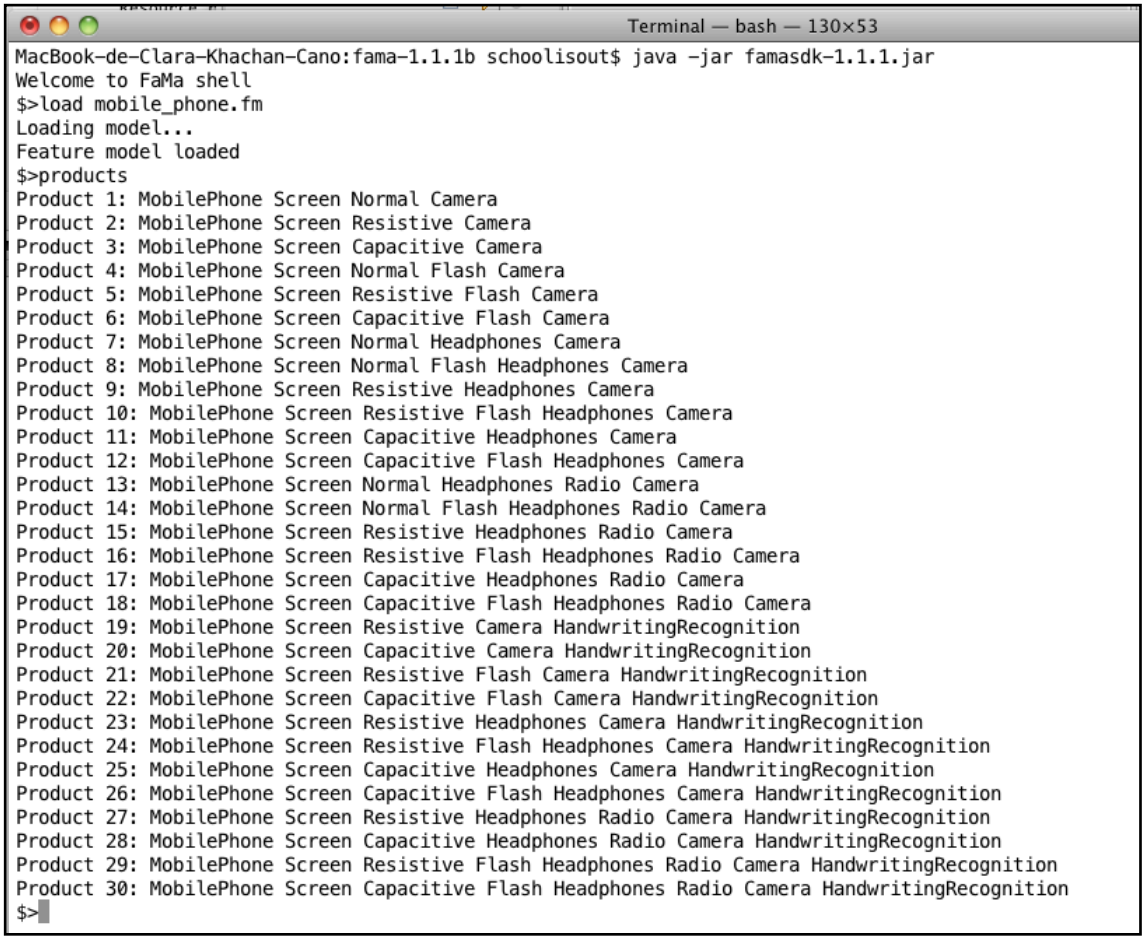


Figura 4.6 Uso de FaMa mediante línea de comandos

Esta solución presenta los siguientes inconvenientes:

- No proporciona trazabilidad, ya que se crea el archivo legible por FaMa mediante código. No tenemos información sobre qué elemento del modelo origen se corresponde con qué elemento del modelo destino. Así que comprobar manualmente que nuestro modelo en FaMa está bien construido manualmente, sería una tarea igual de tediosa que elaborarlo a mano desde cero para FaMa.

- No ofrece interoperabilidad con FaMa. Su uso no se integra en la herramienta, de manera que para poder analizar el modelo, hemos de bajarnos FaMa y utilizarla en su versión de interfaz de línea de comandos. Lo que requiere además aprender el uso de otra herramienta.

- Además, el modelo origen tiene que estar elaborado de manera que se conforme con el metamodelo definido por MULTIPLE.

Parte III

Un Framework para el Análisis de Líneas de Producto Software

5. Contribuciones

En este capítulo expondremos el contexto y los motivos que nos llevaron a la creación de este framework para el análisis de modelos de características. Además, listaremos las contribuciones que aporta nuestra propuesta.

5.1 Contexto y Motivaciones

En la actualidad, sigue representando un desafío el poder realizar un análisis completo sobre modelos de características de gran escala (con muchas características). Es deseable disponer de herramientas que faciliten la realización de esta tarea. Además, es necesario seguir alimentando el debate sobre los tipos de errores más comunes que se detectan en estos modelos de características industriales de gran escala y de cómo solucionarlos. Para poder establecer una discusión sobre lo anteriormente citado, hemos de disponer de resultados obtenidos tras haber analizado modelos de características industriales de gran escala reales.

En este contexto, nos encontramos con que existen gran variedad de notaciones con las que especificar nuestra SPL. Partimos de que las herramientas de análisis automático que existen hasta el momento, se basan en una representación (modelo) al que nuestro modelo de características ha de adaptarse para poder ser analizado. De esta manera, coger nuestro modelo de características y analizarlo mediante una herramienta de análisis automático no es una tarea inmediata, ya que probablemente éste no se adapte totalmente a la representación aceptada por la herramienta. Asimismo, si nuestro modelo es muy extenso, es posible que contenga errores de sintaxis que nos sean muy difíciles de localizar manualmente y que hagan del modelo un modelo no representable y por lo tanto no analizable por la herramienta.

Luego, para poder analizar nuestro modelo, necesitamos uno o varios pasos intermedios en los que:

- Se compruebe que el modelo es sintácticamente correcto
- Se adapte el modelo a la representación aceptada por la herramienta de análisis automático

Además, comprobar que el modelo es sintácticamente correcto es en sí mismo analizar la corrección del modelo; no en contenido, pero sí en forma.

En el capítulo anterior «Modelado de características basado en cardinalidades e Ingeniería Dirigida por Modelos», hablábamos de la herramienta MULTIPLE. Esta herramienta integra el modelado de características y la Ingeniería Dirigida por Modelos, implementando una serie de componentes que nos permiten explotar los modelos de características extendidos. Pero dicha herramienta no integra la funcionalidad necesaria para poder analizar los modelos.

La Figura 5.1 muestra la parte de la arquitectura descrita en MULTIPLE, en la que se da soporte al modelado de características. Como podemos observar, se compone de una serie de plugins que soportan el trabajo descrito en el capítulo 4. El plugin es.upv.dsic.issi.multiple.features contiene el metamodelo. Los plugins es.upv.dsic.issi.-multiple.features.edit, es.upv.dsic.issi.multiple.features.editor, es.upv.dsic.issi.multipl-e.features.diagram contienen código generado por EMF y GMF para la representación del modelo en Ecore y su representación gráfica y editable.

Por otro lado, el plugin es.upv.dsic.issi.multiple.famabridge contiene el código en Java que transforma de forma manual, un modelo de característica descrito en MULTIPLE a un archivo legible por FaMa (apartado 4.4)

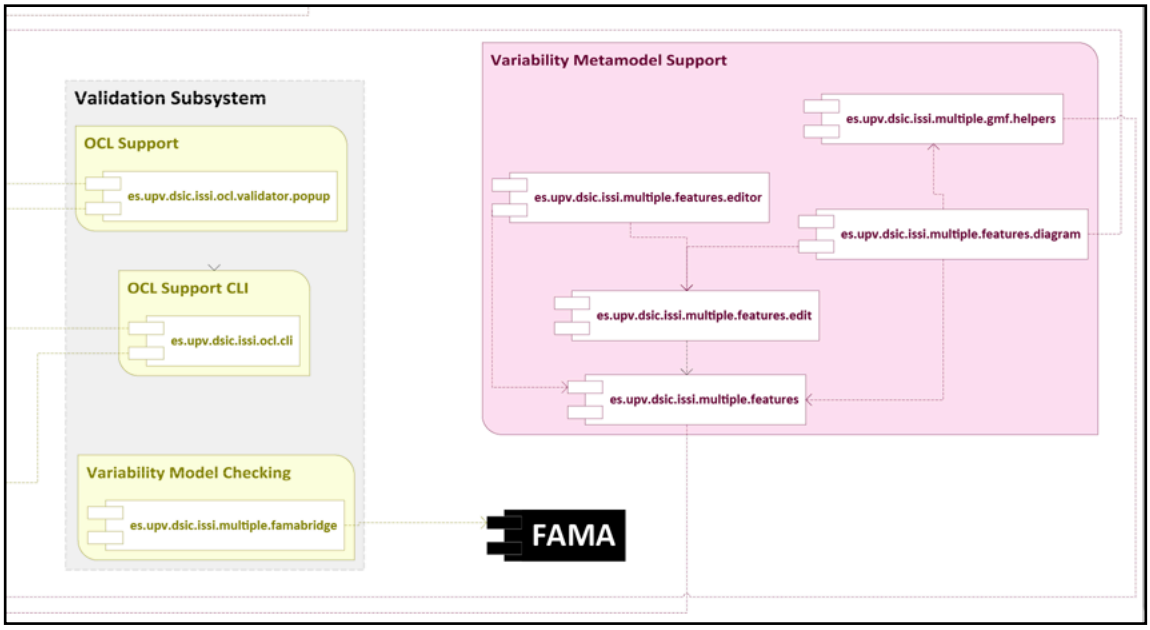


Figura 5.1 Arquitectura simplificada* de MULTIPLE

Como comentamos a lo largo de este documento, nuestro propósito es desarrollar un framework único que nos permita tomar nuestro modelo de características industrial (proporcionado por Rolls Royce) en notación PLUSS y poder analizarlo en profundidad. Para ello utilizaremos el *background* que nos proporciona MULTIPLE gracias a su extensibilidad.

5.2 Listado de aportaciones

Estudiado el contexto y encontrados los motivos, nuestro framework viene a salvar las dificultades comentadas en el apartado 5.1, proporcionando los componentes adecuados para acercar distancias entre un modelo de características cualquiera y su análisis semántico mediante el uso de una herramienta de análisis automático.

* La arquitectura que define MULTIPLE incorpora otros tantos componentes que omitimos ya que no son relevantes para este trabajo.

Nuestra propuesta contribuye con las siguientes aportaciones:

1. Se extiende el metamodelo de características definido en MULTIPLE, de manera que los elementos del modelo podrán almacenar información extra en forma de anotaciones.

2. Se implementa un *parser* que se encarga de procesar los datos del modelo origen y crea la instancia XMI conforme a nuestro metamodelo pivote. Se automatiza además, el análisis y corrección sintáctica del modelo.

3. Implementación de un componente para la validación del contenido de los modelos de características mediante la integración de FaMa a nuestra herramienta (es.upv.es.dsic.issi.multiple.fama.bridges de la Figura 5.2).

Se implementan los componentes que contienen el metamodelo de FaMa (es.upv.es.dsic.issi.multiple.fama, es.upv.es.dsic.issi.multiple.fama.edit, es.upv.es.dsic.-ssi.multiple.fama.editor de la Figura 5.2).

4. Aportaciones sobre la herramienta actual de MULTIPLE:

    a. Interfaz mejorada

    b. Se adapta el editor de modelos de características para que soporte el nuevo metamodelo.

    c. Se definen las reglas en QVT-Relations para la transformación del modelo a un modelo FaMa, aportando así trazabilidad ya que se sustituye el componente es.upv.dsic.issi.multiple.famabridge.

5. Por último, se aplica este framework a nuestro caso de estudio en el que analizamos el modelo de características de gran escala de Rolls Royce, aportando:

    a. Detección de errores.

    b. Clasificación de errores.

    c. Soluciones a estos errores.

    d. Porcentajes de corrección del modelo.

Lo novedoso en este trabajo es la concepción de un método en el que se especifican los pasos a seguir para resolver la problemática existente y su materialización en forma de prototipo para el usuario final. De manera que, partiendo de un modelo de características en el que la notación empleada no es relevante, pueda usarse una herramienta tan potente como FaMa para analizar su contenido automáticamente.

La Figura 5.2 muestra las aportaciones realizadas a la arquitectura de MULTIPLE.

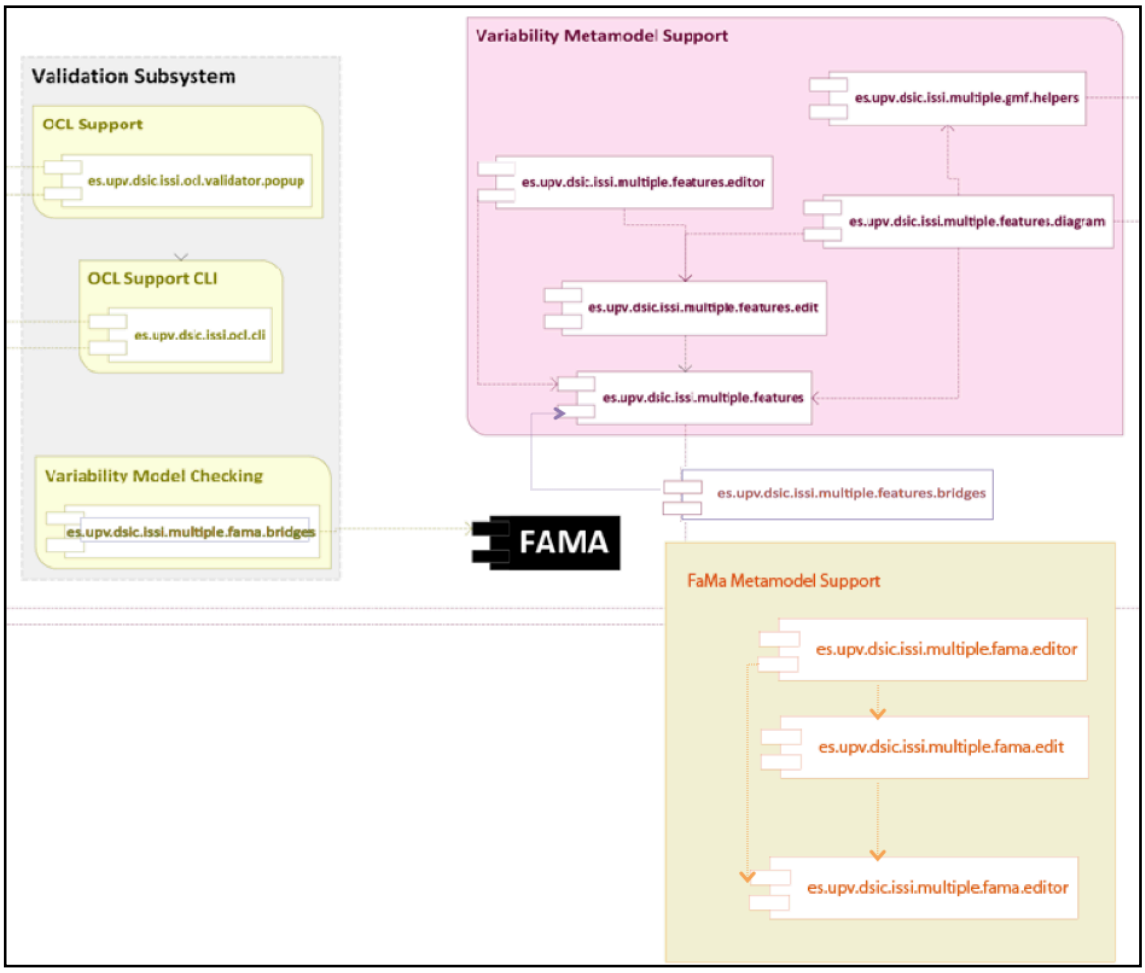


Figura 5.2 Aportaciones a la arquitectura de MULTIPLE

6. Framework de extensión de MULTIPLE

6.1 Estructura del framework

Nuestro framework, desarrollado en Eclipse y EMF mediante una serie de plugins (cuya implementación se relatará en el próximo capítulo), nos permitirá tomar como entrada el modelo de características y analizarlo sintáctica y semánticamente. Con este fin, nuestro framework implementa los componentes necesarios para la realización de las siguientes tareas:

I. Obtención de un modelo analizable.

- Análisis y corrección sintáctica del modelo necesarios para obtener un modelo válido.
- Clasificación y soluciones a los errores extraídos.

II. Transformación del modelo sintácticamente correcto a un modelo entendible por la herramienta de análisis automático.

- Definición mediante un lenguaje de transformaciones las reglas de transformación entre el dominio origen y el dominio destino.

III. Análisis semántico del modelo mediante la herramienta de análisis automático.

- Clasificación y soluciones a los errores extraídos

El esquema de la Figura 6.1 muestra la estructura de nuestro framework:

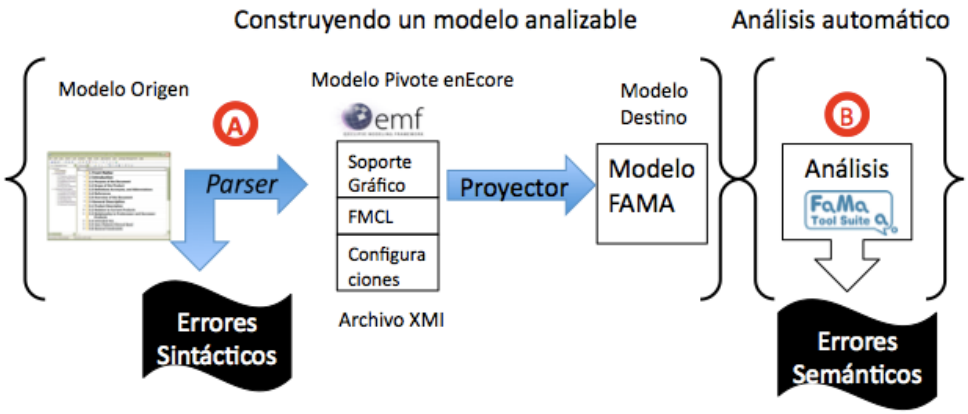


Figura 6.1 Esquema de nuestro framework

A continuación, expondremos con más detalle en qué consiste cada una de las partes de nuestro framework.

6.1.1 Obteniendo un modelo analizable

El propósito de esta primera parte del framework, es obtener un modelo analizable. Esto significa obtener un modelo apto para ser analizado por la herramienta de análisis automático que se vaya a usar. En nuestro caso, hemos decidido usar FaMa [47], por ello, tendremos que obtener un modelo de características que se ajuste a la representación usada por FaMa.

6.1.1.1 El Modelo Origen

Es el modelo que nuestra herramienta tomará como entrada. No existen restricciones sobre la notación en que se encuentre expresado este modelo, ya que antes de ser proyectado a FaMa, lo representaremos como un modelo de características que se ajuste a nuestro propio modelo (que usaremos como modelo pivote), el cual explicaremos en detalle a continuación.

6.1.1.2 El Modelo Pivote

Este modelo, es el que utilizaremos como puente entre el modelo origen y su representación en FaMa. El modelo en Ecore, se ajusta al metamodelo definido en el capítulo anterior y basado en [23], el cual contiene toda la funcionalidad de las propuestas de la literatura actual sobre el manejo de la variabilidad en los modelos de características. De esta forma, cualquiera que sea la notación en la que esté expresado el modelo de entrada, podremos realizar los *mappings* necesarios para transformarlo nuestro modelo pivote.

Es posible que el modelo origen contenga cierta información que no encaje en nuestro modelo pivote pero que el usuario quiera conservar para otros posibles procesos del modelo. Esta información no es necesaria para llevar a cabo el análisis del modelo, y además, es sensible de perderse a lo largo del proceso. Para poder reflejar esta información en nuestro modelo pivote de manera que esté disponible para otros usos posibles, modificamos el metamodelo de [23] expuesto en el capítulo anterior, incorporando las clases *AnnotationSet* y *Annotation*. De esta manera, podremos tener asociada a cada *feature* el conjunto de valores que deseemos. Un elemento del tipo *AnnotationSet* podrá contener una serie de anotaciones (*Annotation*), dónde el atributo *key* contendrá el tipo de información extra que se quiera almacenar y *value* el contenido de esta. Además, se ha modificado la clase *ConstrainableElement* del modelo Figura 4.2 sustituyéndola por la clase *ModelElement.* Esta clase contendrá los conjuntos de anotaciones y restricciones definidos por las clases *AnnotationSet* y *ConstraintSet*. Hemos considerado que esta era la mejor manera de representar nuestro metamodelo ya que ambas clases, *AnnotationSet* y *ConstraintSet,* definen estructuras semejantes.

La Figura 6.2 representa el metamodelo, dónde las clases coloreadas son las añadidas a la propuesta original.

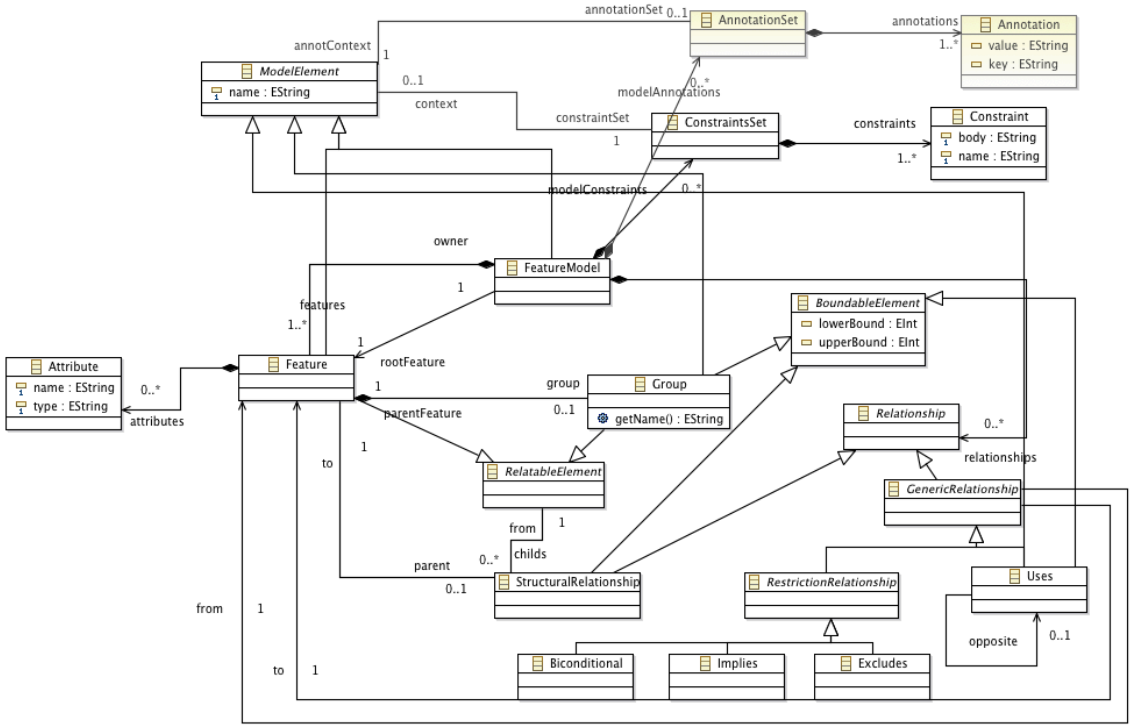


Figura 6.2 Metamodelo que representa nuestra propuesta

Este metamodelo, esta definido en Eclipse Modeling Framework (EMF), por lo que podemos hacer uso de la funcionalidad descrita e implementada en el trabajo de Gómez y Ramos que ha ocupado el capítulo 4 de este documento:

- Soporte gráfico para definir una variante de modelos de características basado en cardinalidades con restricciones descritas usando un lenguaje de restricciones.
- Soporte para generar automáticamente Modelos de Variabilidad de Dominio (DVMs) a partir de FMs, que contienen toda la variabilidad del dominio de aplicación (incluyendo restricciones complejas).
- Soporte para transformar restricciones de modelo en expresiones OCL.
- Editor de configuraciones, con asistente para desarrolladores.
- Capacidades para comprobar la consistencia de una configuración sobre su modelo de características usando motores OCL preconstruidos.

Soporte Gráfico

Como al metamodelo se le han añadido dos clases (*AnnotationSet* y *Annotation*) con las que no contaba de la propuesta inicial, el editor generado usando GMF ha sido modificado para poder definir este tipo de instancias del modelo. En la Figura 6.3 observamos como queda el editor tras la incorporación de estas clases al modelo. Podemos ver como en la paleta de la derecha, se incluyen tres nuevos elementos necesarios para su representación. El elemento *AnnotationSet* nos servirá para definir el conjunto de anotaciones (*Annotation*) que iremos incluyendo en él. Por otra parte, el elemento *Annotation Link* nos servirá para enlazar la anotación con el elemento del FM al que haga referencia.

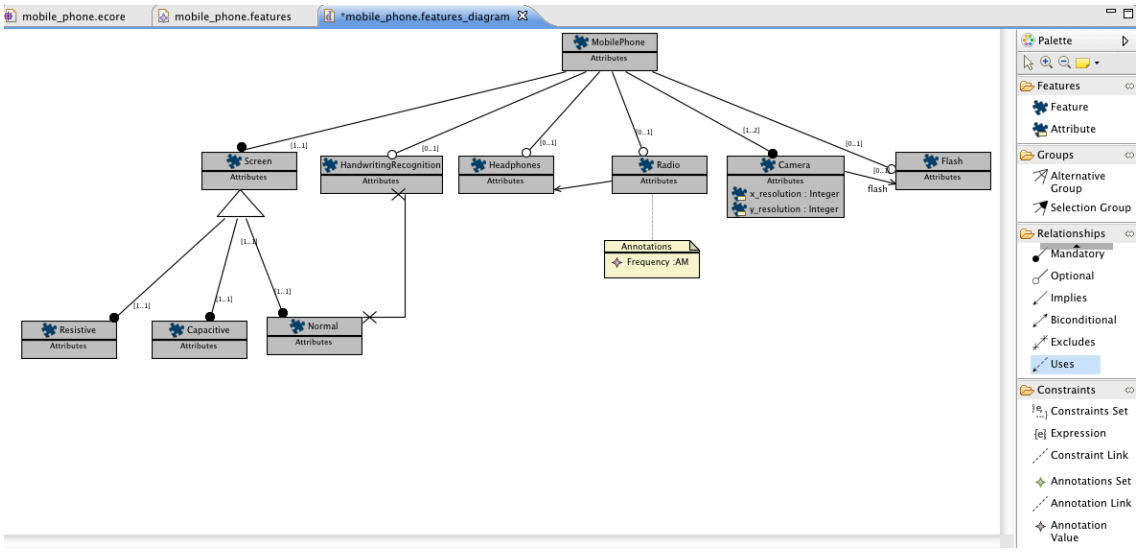


Figura 6.3 Ejemplo FM con anotaciones

6.1.1.3 Componente Parser (A)

Este componente *Parser* del framework, es el que se encarga de procesar los datos del modelo origen, el cual puede estar expresado en cualquier notación y persistido en diversos formatos. Este proceso consiste en la transformación del modelo de entrada, a una instancia XMI que se conforma al modelo pivote (basado en cardinalidades) descrito en el apartado 6.1.1.2.

En este desarrollo, se lee el modelo de características de entrada y se recorren todas las características, relaciones y restricciones presentes en él, creando la instancia equivalente en EMF conforme al modelo pivote. De esta manera, obtenemos un modelo en Ecore totalmente equivalente al de entrada con todos los beneficios que se derivan de trabajar en EMF.

Además, la elaboración de este código de procesado (Anexo B), incluye un análisis de la corrección sintáctica del modelo origen, corrigiendo posibles inconsistencias sintácticas para obtener un modelo representable. El resultado de este análisis y las medidas tomadas para la corrección de las posibles inconsistencias, se muestran al usuario por una consola propia. Así, el usuario tiene constancia de los posibles errores que contenga su modelo y que lo hacen no representable. De esta manera, el *parser* no sólo corrige el modelo, si no que proporciona al usuario un *feedback* que le permitirá deliberar más profundamente sobre los motivos de los errores contenidos.

En resumen, la funcionalidad del componente *Parser* es:

- Transformación del modelo entrada en modelo pivote (Ecore)
- Obtención por consola de un análisis sintáctico del modelo origen y de las medidas que se han tomado para su corrección y persistencia en Ecore.

Cabe comentar que este componente del Framework, varía dependiendo del tipo modelo de entrada que se vaya a tomar, ya que el procesado depende del aspecto y

contenido de éste. En nuestro caso, ha sido desarrollado para un modelo concreto de características utilizado en la industria, cuya estructura veremos en el capítulo siguiente.

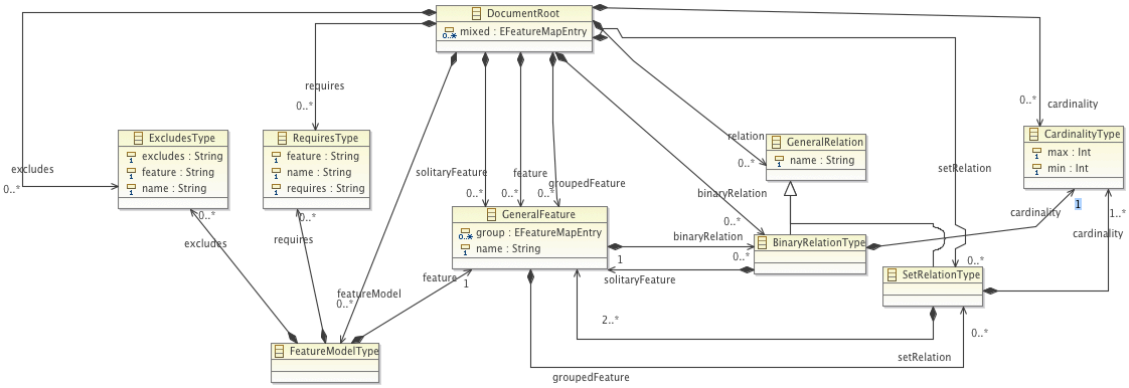
En nuestro caso práctico, el modelo industrial que tomamos como entrada utiliza la notación PLUSS (explicada en el apartado 2.1.1.1) para representar la variabilidad. La Tabla 6.1 representa las correspondencias entre las relaciones presentes en PLUSS y nuestro modelo pivote.

Tabla 6.1 Tabla de correspondencias PLUSS y nuestro modelo basado en cardinalidades

PLUSS	Modelo basado en cardinalidades
Mandatory o Common 	**Mandatory**  **[1..1]**
Optional 	**Optional**  **[0..1]**
Single 	**XOR**  **[1..1]**
Multiple 	**OR**  **[1..n]**
Requires 	**Implication** 
Excludes 	**Prohibited** 

6.1.1.4 El modelo de FaMa

La herramienta FaMa, toma como entrada modelos de características tanto simples como extendidos. La Figura 6.4 muestra el metamodelo de características FaMa. Este metamodelo se proporciona como un archivo XSD, por lo que podemos importarlo en EMF como se explicaba en la sección 3.2.2.



En este metamodelo, la clase *FeatureModelType* representa al modelo. *GeneralFeature* sirve para instanciar las características que se relacionan mediante las relaciones de jerarquía *BinaryRelationType* y *SetRelationType*. La relación *BinaryRelationType* sirve para representar las relaciones que en la literatura se conocen como Mandatory y Optional, mientras que *SetRelationType*, representará las relaciones de tipo OR y XOR. Para expresar las restricciones o relaciones de tipo *cross-tree*, el modelo define las clases *ExcludesType* y *RequiresType*.

Dicho modelo, esta disponible en XML Schema, así que podemos trabajar con él en EMF y definir las equivalencias entre nuestro modelo pivote y el de FaMa.

6.1.1.5 Proyector del modelo pivote a FaMa

Este es el último paso de esta primera parte del framework, que nos da como resultado un modelo analizable por FaMa.

Para ello, se han definido las reglas de transformación que permiten convertir los datos del dominio origen al dominio destino. Estas reglas expresan las correspondencias entre el modelo pivote y el modelo de FaMa.

El lenguaje con el que se expresan las relaciones entre ambos dominios es QVT-Relations, del que hablamos en la sección 3.1.3.1 y que resumimos brevemente a continuación.

En QVT-Relations, una transformación son un conjunto de relaciones establecidas entre los dominios participantes en la transformación que deben cumplirse para que ésta sea satisfactoria [50]. Un dominio es una variable tipada que puede corresponderse con algún elemento del modelo que va a transformarse. Éste puede tener un patrón, que puede considerarse como un conjunto de restricciones que deben cumplir los elementos del modelo candidato —modelo sobre el que se aplica la transformación— para que se trate de una correspondencia válida.
Los dominios además, pueden caracterizarse mediante el uso de las palabras clave *checkonly* y *enforce*. Para un dominio *checkonly*, la transformación comprobará que existe una correspondencia válida —que satisfaga el patrón del dominio— en el modelo candidato. En caso de que el dominio destino sea *enforce*, si al ejecutar la

transformación no existe ninguna correspondencia posible, se creará un elemento que cumpla con el patrón del dominio.

La relación *Model2Model* de la Figura 6.5, muestra un ejemplo de la sintaxis para la definición de relaciones y dominios. Esta relación, establece la correspondencia entre el elemento raíz del modelo de características basado en cardinalidades (pivote) y el del modelo FaMa. Encontramos dos dominios: *featuresDomain* se corresponde con el dominio origen y *famaDomain* con el dominio destino (FaMa). Para el dominio *featuresDomain*, se comprueba que exista un modelo *featureModel* con la relación *rootfeature*, que contiene la *Feature* raíz del modelo. Si esto se cumple, deberá existir (y si no, se creará) un *DocumentRoot* que contenga el *FeatureModelType* con la *GeneralFeature* raíz, cuyo atributo *name* se corresponde al nombre de la *Feature* raíz del dominio origen.

```
top relation Model2Model {

    checkonly domain featuresDomain fmodel : features::FeatureModel{
        rootFeature = root :features::Feature{}
        };
    enforce domain famaDomain fmodel2 : FeatureModelSchema::DocumentRoot{
        featureModel = model : FeatureModelSchema::FeatureModelType {
            feature = first:FeatureModelSchema::GeneralFeature{
                name=root.name
            }
                }
        };
    }
```

Figura 6.5 regla Model2Model

El proceso de transformación completo se realiza de arriba abajo, navegando el modelo origen a través de las relaciones de contención, definidas como asociaciones de composición. Esto es, se parte del elemento raíz del modelo origen (*FeatureModel*), y se navega hacia abajo (*FeatureModel* → *Structuralrelationship* → *Group* → *Excludes* → *Includes*) creando en el modelo destino los elementos correspondientes, tal y como expresa de forma resumida la Tabla 6.2.

Tabla 6.2 Correspondencias entre dominio origen y destino

Modelo basado en cardinalidades	**FaMa**
FeatureModel	FeatureModelType
Feature	GeneralFeature
StructuralRelationship	BinaryRelationType
Group	SetRelationType
Implies	RequiresType
Excludes	ExcludesType
BoundableElement	Cardinality

Todas las reglas de transformación especificadas, se explican en detalle en la sección 6.2.3.

6.1.2 Análisis automático del modelo de características

Esta segunda parte del Framework comprende el análisis semántico de nuestro modelo, el cual se realizará mediante la herramienta de análisis automático FaMa.

Como explicábamos en el capítulo 2.2.4, FaMa nos surte de una serie de operaciones para extraer información importante de nuestro modelo de características. Esta herramienta puede utilizarse de diversas maneras. Una de ellas es como implementación OSGi lo que significa que incorporando las librerías proporcionadas por la herramienta a nuestro entorno de desarrollo, podemos usarlas como mejor nos convenga. Dado que nuestra herramienta ha sido desarrollada íntegramente en Eclipse, este es el formato más conveniente para nosotros.

Este componente del framework, desarrollado también mediante un plugin, implementa las operaciones de análisis de FaMa, pudiéndolas ejecutar mediante un menú contextual sobre nuestro modelo ya representable para FaMa. De esta forma, desde un mismo entorno de trabajo, podremos gestionar cada una de las funcionalidades de la herramienta de manera sencilla e intuitiva.

Además, se implementa una consola en la que se van mostrando los resultados del análisis realizado.

6.2 Implementación

En esta sección describiremos como hemos implementado cada uno de los componentes que conforman nuestro framework. En el apartado 6.1, definíamos la arquitectura genérica y los artefactos que necesitábamos para llevar a acabo nuestros objetivos. En los siguientes apartados, materializamos el desarrollo de los componentes descritos en el capítulo anterior.

A continuación, detallaremos la definición e implementación de los metamodelos que participan del proceso. En segundo lugar, se especificará el contenido de las herramientas desarrolladas como componentes de la plataforma Eclipse (plugins). Posteriormente, procederemos a la definición de las transformaciones entre el dominio origen (basado en cardinalidades) y el dominio destino (FaMa) y la representación gráfica de las mismas. Por último, se presentarán los plugins realizados para la realización del análisis automático mediante el uso de las librerías proporcionadas por el Framework FaMa.

Estos plugins se incluían en la Figura 5.2, dónde exponíamos las aportaciones realizadas a la arquitectura de MULTIPLE.

6.2.1 Metamodelos

Dado que vamos a trabajar con dos dominios diferentes, necesitamos definir los metamodelos de ambos para poder tratar con las instancias de estos modelos Ecore directamente en Eclipse. A continuación, describiremos los plugins que hemos

desarrollado con este fin. El código contenido en estos plugins ha sido, en su mayor parte, generado de manera automática. Tan sólo detallaremos aquellas partes del código que hayamos modificado manualmente, mientras que aquellas partes obtenidas mediante técnicas de programación generativas no serán pormenorizadas.

Los plugins es.upv.dsic.issi.multiple.features, es.upv.dsic.issi.multiple.features.-edit y es.upv.dsic.issi.multiple.features.editor, que contienen al modelo pivote y los elementos para su representación, ya estaban implementados en MULTIPLE. Los cambios vienen dados por la modificación del metamodelo de características definido en es.upv.dsic.issi.multiple.features. El código de *edit* y *editor* se genera automáticamente a partir del metamodelo Ecore. Por este motivo, sólo se hará una breve descripción del contenido de estos plugins.

6.2.1.1 Plugin es.upv.dsic.issi.multiple.features

6.2.1.1.1 Descripción

Este plugin, define el metamodelo del modelo de características basado en cardinalidades, que será empleado en las transformaciones. Fundamentalmente, está formado por código generado automáticamente por EMF. Para ello, se ha definido mediante Ecore el metamodelo que se presentaba en la Figura 6.2 y se han empleado los mecanismos estándares de EMF.

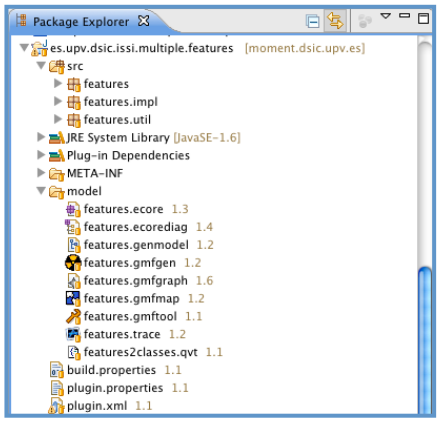


Figura 6.6 Proyecto es.upv.dsic.issi.multiple.features

En la Figura 6.6 podemos ver el contenido de este plugin desde el explorador de proyectos de Eclipse

6.2.1.2 Plugin es.upv.dsic.issi.multiple.features.edit

6.2.1.2.1 Descripción

El plugin es.upv.dsic.issi.multiple.features.edit proporciona la funcionalidad básica para representar los elementos del modelo basado en cardinalidades en un editor.

En Figura 6.7 la podemos ver el contenido de este plugin desde el explorador de proyectos de Eclipse

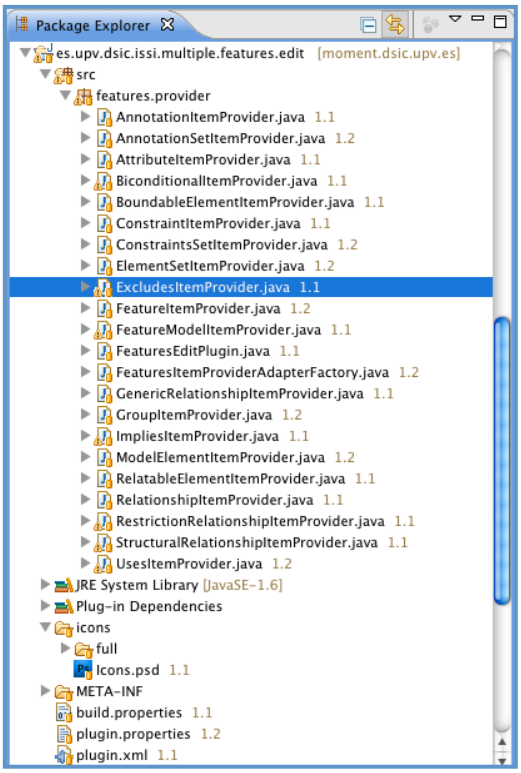


Figura 6.7 Proyecto es.upv.dsic.issi.multiple.features.edit

6.2.1.3 Plugin es.upv.dsic.issi.multiple.features.editor

6.2.1.3.1 Descripción

Este plugin implementa un editor en árbol típico de EMF. Se encuentra asociado por defecto con la extensión «*.features». Además, incluye el asistente de creación de un nuevo fichero «*. features » integrado con el resto de asistentes de creación de recursos de Eclipse.

En la Figura 6.8 podemos ver el contenido de este plugin desde el explorador de proyectos de Eclipse

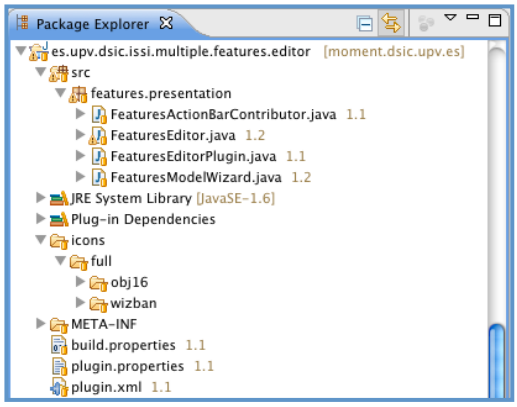


Figura 6.8 Proyecto es.upv.dsic.issi.multiple.features.editor

6.2.1.4 Plugin es.upv.dsic.issi.multiple.fama

6.2.1.4.1 Descripción

Este plugin, define el metamodelo que representa los modelos de características FaMa, que será empleado en las transformaciones. Fundamentalmente, está formado por código generado automáticamente por EMF. Para ello, se ha importado el metamodelo de FaMa en formato XML Schema (el proporcionado por la plataforma), representado en la Figura 6.4 y se han empleado los mecanismos estándares de EMF.

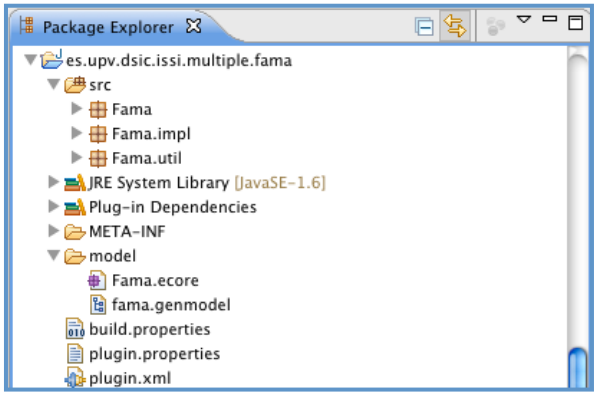


Figura 6.9 Proyecto es.upv.dsic.issi.multiple.fama

6.2.1.4.2 Dependencias

El código de este plugin está generado completamente por EMF, habiendo importado el modelo en formato XML Schema. Las dependencias son las siguientes:

- org.eclipse.core.runtime
- org.eclipse.emf.ecore
- org.eclipse.emf.ecore.xmi

6.2.1.4.3 Descripción de paquetes y clases

❖ **Paquete es.upv.dsic.issi.multiple.fama**

El paquete es.upv.dsic.issi.multiple.fama contiene las interfaces que implementan las clases que se encuentran en el paquete es.upv.dsic.issi.multiple.fama.impl. En EMF se implementa tanto una interfaz como una clase por cada una de las clases definidas en el modelo Ecore. De esta manera, se pueden simular en Java los mecanismos de herencia múltiple que permite Ecore. Las interfaces *FamaFactory* y *FamaPackage* no se generan a partir de las clases del modelo, sino que se corresponden con el modelo en

sí. En particular, FamaFactory es la interfaz que permite acceder a la factoría para crear nuevas instancias de los objetos del modelo, y FamaPackage se corresponde con la interfaz que contiene la información del paquete (nsPrefix, nsUri, etc.) así como el acceso a la instancia *singleton* del paquete.

Listado de clases

- BinaryRelationType.java
- CardinalityType.java
- DocumentRoot.java
- ExcludesType.java
- FamaFactory.java
- FamaPackage.java
- FeatureModelType.java
- GeneralFeature.java
- GeneralRelation.java
- RequiresType.java
- SetRelationType.java

❖ **Paquete es.upv.dsic.issi.multiple.fama.impl**

El paquete es.upv.dsic.issi.multiple.fama.impl contiene las clases que implementan el código del modelo. En este sentido, se crea una clase Java (con el sufijo *–Impl*) para cada una de las clases del modelo. Cada una de esta clases contiene los métodos (con código generado) para consultar, navegar y modificar instancias del modelo (métodos *getters* y *setters*). Estos métodos además, contienen los mecanismos pertinentes para mantener la coherencia entre los objetos del modelo cuando se encuentran relaciones bidireccionales entre las clases del modelo (mediante la interfaz *Notifier*).

Además de aquellas clases que se corresponden con las del modelo, encontramos las clases especiales FamaFactoryImpl y FamaPackageImpl. La primera de ellas se corresponde con la clase que implementa las factorías de objetos del modelo (en EMF se recomienda usar el patrón de factorías, en lugar de emplear el operador *new* de Java). La segunda clase, FamaPackageImpl, se corresponde con la clase que implementa el paquete del modelo. De ésta clase sólo habrá una única instancia en memoria (según el patrón *singleton*) y contiene los métodos que permiten recrear en memoria la estructura del modelo al inicializarse el plugin.

Listado de clases

- BinaryRelationTypeImpl.java
- CardinalityTypeImpl.java
- DocumentRootImpl.java
- ExcludesTypeImpl.java
- FamaFactoryImpl.java
- FeatureModelTypeImpl.java
- GeneralFeatureImpl.java
- GeneralRelationImpl.java
- RequiresTypeImpl.java
- SetRelationTypeImpl.java

- FamaPackageImpl.java

❖ **Paquete es.upv.dsic.issi.multiple.fama.util**

En este paquete se encuentran las dos clases de utilidades que realizan tareas auxiliares comunes en los plugins de EMF para la creación y navegación de los elementos del modelo y además tres clases más que se generan al haber importado el modelo como un archivo XML *Schema* .

Listado de clases

- FamaAdapterFactory.java
- FamaSwitch.java

Clases creadas automáticamente al haber importado el modelo en XML *Schema*:

- FamaResourceFactoryImpl.java
- FamaResourceImpl.java
- FamaXMLProcessor.java

6.2.1.5 Plugin es.upv.dsic.issi.multiple.fama.edit

6.2.1.5.1 Descripción

El plugin es.upv.dsic.issi.multiple.fama-.edit proporciona la funcionalidad básica para representar los elementos del modelo de FaMa en un editor.

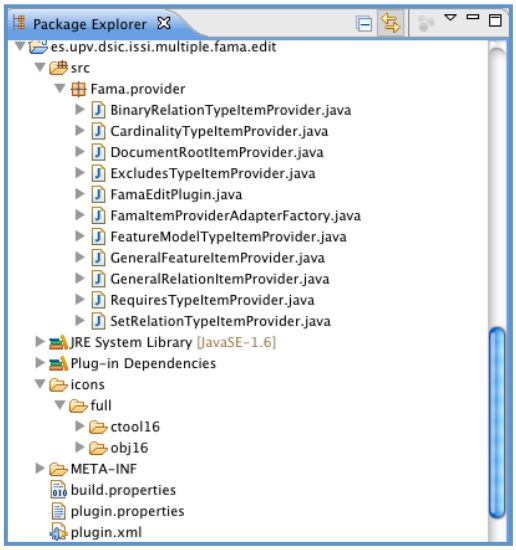


Figura 6.10 Proyecto es.upv.dsic.issi.multiple.fama.edit

6.2.1.5.2 Dependencias

Las dependencias de este plugin son:

- org.eclipse.core.runtime (por ser un plugin de Eclipse)
- org.eclipse.emf.edit (por ser un plugin de EMF Edit)
- El metamodelo de FaMa (es.upv.dsic.issi.multiple.fama) por ser el soporte para su edición.

6.2.1.5.3 Descripción de paquetes y clases

El contenido de este plugin se ha generado de forma automática, aunque se han realizado cambios menores en el código (principalmente en los métodos getText() de los diferentes *ItemProviders*).

❖ **Paquete es.upv.dsic.issi.multiple.fama.provider**

Este es el único paquete del plugin. Contiene la clase activadora del plugin (FamaEditPlugin), que se encarga de controlar el ciclo de vida del plugin mediante su instancia *singleton*. A su vez, contiene la clase FamaItemProviderAdapterFactory que es la clase de la factoría para acceder a cada uno de los *Items providers* contenidos en el paquete.

Por último, contiene una clase *provider* por cada clase del modelo (todas con el sufijo *–ItemProvider*). Estas clases sirven para consultar cómo se deben mostrar los elementos en los editores, esto es, qué icono representará a cada elemento, cómo se construirá el texto de sus etiquetas, qué elementos del modelo son sus hijos (de manera que se presentan los correspondientes menús de creación de nuevo hijo), qué propiedades deben de mostrarse en las hojas de propiedades para poder ser visualizadas/editadas, etc.

Listado de clases y métodos

A continuación se listan las clases del paquete. Los métodos proporcionados por un *ItemProvider* son los mismos que los listados en la tabla de la sección 7.1.2.3, así que no los volveremos a listar aquí.

- BinaryRelationTypeItemProvider.java
- CardinalityTypeItemProvider.java
- DocumentRootItemProvider.java
- ExcludesTypeItemProvider.java
- FamaFactoryItemProvider.java
- FamaPackageItemProvider.java
- FeatureModelTypeItemProvider.java
- GeneralFeatureItemProvider.java
- GeneralRelationItemProvider.java
- RequiresTypeItemProvider.java
- SetRelationTypeItemProvider.java

6.2.1.6 Plugin es.upv.dsic.issi.multiple.fama.editor

6.2.1.6.1 Descripción

Este plugin implementa un editor en árbol típico de EMF. Se encuentra asociado por defecto con la extensión «*.fama». Además, incluye el asistente de creación de un nuevo fichero «*. fama » integrado con el resto de asistentes de creación de recursos de Eclipse.

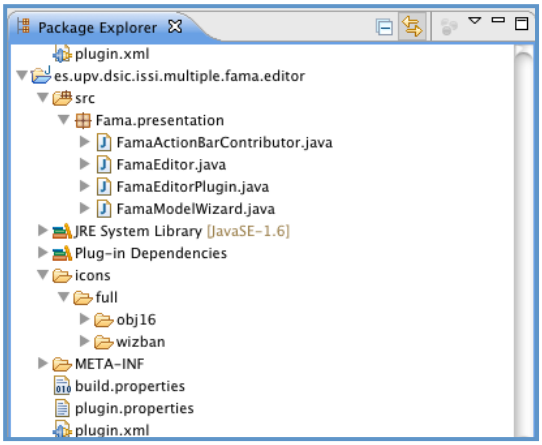


Figura 6.11 Proyecto es.upv.dsic.issi.multiple.fama.editor

6.2.1.6.2 Dependencias

- org.eclipse.core.runtime
- org.eclipse.core.resources
- org.eclipse.ui.ide
- org.eclipse.emf.ecore.xmi
- org.eclipse.emf.edit.ui
- es.upv.dsic.issi.multiple.fama.edit

6.2.1.6.3 Descripción de paquetes y clases

❖ **Paquete es.upv.dsic.issi.multiple.fama.presentation**

Este es el único paquete del plugin y su código es completamente generado de forma automática. Contiene a su vez las siguiente clases:

FamaActionBarContributor

La clase FamaActionBarContributor contiene las acciones que se contribuirán a la interfaz estándar de Eclipse. Como contribuciones entendemos los botones que se añadirán a la barra de herramientas de Eclipse, el menú que aparecerá en la barra de menús cuando el editor se encuentra abierto y activo, etc.

FamaEditor

La clase FamaEditor implementa el editor en árbol para las instancias del modelo de FaMa . En cuanto a funcionalidad es similar a cualquier otro editor en árbol genérico, siendo su única diferencia en cómo en la inicialización del editor (método initializeEditingDomain()) se añade el *ItemProviderAdapterFactory* (creado en el plugin es.upv.dsic.issi.multiple.fama.edit comentado anteriormente) al adapterFactory propio del editor.

FamaEditorPlugin

La clase FamaEditorPlugin es la clase activadora del plugin conteniendo la instancia *singleton* que permite acceder al estado de éste y controlar su ciclo de vida.

FamaModelWizard

Esta clase implementa el asistente de creación de una nueva instancia del modelo basado en cardinalidades. El asistente se integra en la categoría «Example EMF Model Creation Wizards»

6.2.2 Parser

6.2.2.1 Plugin es.upv.dsic.issi.multiple.features.bridges

6.2.2.1.1 Descripción

En este plugin, se implementa la lógica fundamental que permite reconstruir una instancia en XMI del metamodelo pivote a partir de un fichero CSV dónde se nos proporciona el modelo de características a analizar. Este plugin, además implementa una consola en la que se nos muestran los errores sintácticos del modelo original.

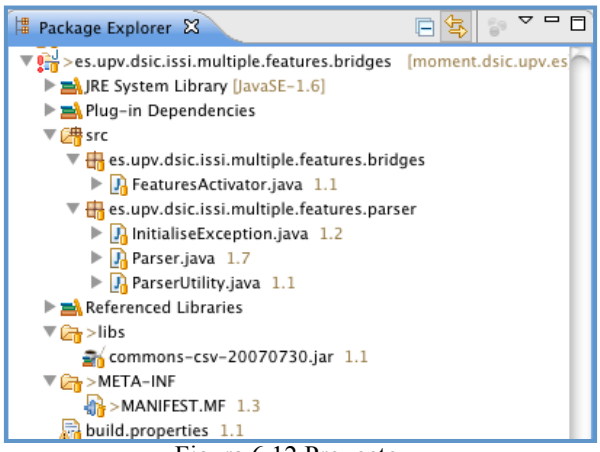


Figura 6.12 Proyecto es.upv.dsic.issi.multiple.features.bridges

6.2.2.1.2 Dependencias

Como plugin de Eclipse, este plugin depende de:

- org.eclipse.ui
- org.eclipse.core.runtime
- org.eclipse.core.resources – Permite acceder a los recursos del workspace. org.eclipse.ui.console – Permite volcar mensajes en la vista de consola de Eclipse.
- es.upv.dsic.issi.multiple.features – Permite tratar con instancias del metamodelo basado en cardinalidades (pivote) .

Además, importa e incluye las siguientes bibliotecas:

- commons-csv.jar. Esta biblioteca nos proporciona soporte para leer y escribir ficheros CSV.

6.2.2.1.3 Descripción de paquetes y clases

❖ **Paquete es.upv.dsic.issi.multiple.features.bridges**

Este paquete únicamente contiene la clase FeaturesActivator, que es la clase activadora del plugin. Ésta se encarga de controlar el ciclo de vida del plugin (métodos start() y stop()), así como proporcionar los métodos de acceso a la funcionalidad del plugin. La clase contiene una variable estáticas featuresParser (Listado 1), instancia de la clase Parser, que comentaremos más adelante.

```
private static Parser featuresParser = new Parser();
```

Listado 1 Declaración de la variable featuresParser

Los métodos que se han añadido a la implementación por defecto son:

createFeatureModel(IFile).

Este método (Listado 2) accede a los contenidos de un IFile (archivo del workspace) e invoca al método createFeatureModel(InputStream) para que realice la re- construcción del modelo de características ajustándose a nuestro metamodelo basado en cardinalidades.

```
public FeatureModel createFeatureModel(IFile file) throws InitialiseException, CoreException, IOException {
    return createFeatureModel(file.getContents());
}
```

Listado 2 Método createFeatureModel(IFile)

createFeatureModel(InputStream)

Este método (Listado 3) toma un stream de texto, e invoca al método createFeatureModel(InputStream) de la clase Parser. El valor devuelto es el modelo de características, instancia del metamodelo de EMF pivote.

```
public FeatureModel createFeatureModel(InputStream stream) throws InitialiseException, CoreException, IOException {
    return featuresParser.createFeatureModel(stream);
}
```

Listado 3 Método createFeatureModel(InputStream)

- **Paquete es.upv.dsic.issi.multiple.features.parser**

Este paquete contiene las clases InitialiseException, Parser y ParserUtility. La primera de ellas, es una excepción que será lanzada en caso de que ocurra algún error al intentar parsear el modelo original en CSV (el fichero CSV no esté accesible, sea un fichero CSV mal formado, etc.).

La clase Parser por su parte, implementa el parser de ficheros CSV. Hace uso de la librería CSV-Commons [46] para parsear el documento.

La mayor parte de la lógica de esta clase está implementada en el método createFeatureModel(InputStream). El código se encuentra en el Apéndice B. Este método se encargará de recorrer el modelo origen para reconstruirlo como una instancia del modelo pivote basado en cardinalidades. Esta tarea comprende además, la búsqueda de inconsistencias sintácticas en cada uno de los elementos que lo conforman y su corrección. También implementa una consola en la que se mostrarán todas las inconsistencias encontradas en el modelo y qué medidas hemos tomado para corregirlas, construyendo así un modelo representable o 'correcto' en forma.

6.2.2.2 Plugin es.upv.dsic.issi.multiple.features.parser.ui

6.2.2.2.1 Descripción

Este plugin implementa los elementos que se contribuyen a la interfaz gráfica de Eclipse para poder invocar de forma manual al parser que convertirá nuestro archivo CSV en una instancia del metamodelo basado en cardinalidades implementado en el plug-in es.upv.dsic.issi.multiple.features.bridg-es. Específicamente, este plugin implementa un menú contextual llamado «Multiple» que aparecerá en el «Explorador de proyectos» de Eclipse, y permitirá invocar diversas accionas sobre un determinado fichero que esté seleccionado. Para ello, se define una clase que implementa la interfaz *org.eclipse.ui.IObjectActionDelegate* por cada opción del menú. Posteriormente, en el archivo de manifiesto «plugin.xml» se define la conexión al punto de extensión para que la acción se contribuya a la interfaz.

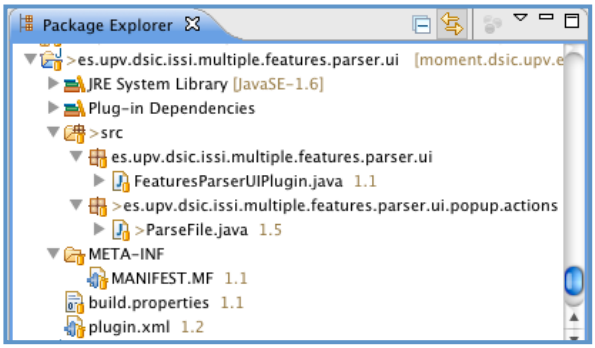


Figura 6.13 Proyecto es.upv.dsic.issi.multiple.ffeatures.parser.ui

En el Listado 4 se muestra un ejemplo de cómo la clase es.upv.dsic.issi.multiple.features.parser.ui.popup.actions.ParseFile se conecta al punto de extensión de los menús popup de Eclipse.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
   <extension
         point="org.eclipse.ui.popupMenus">
      <objectContribution

            id="es.upv.dsic.issi.multiple.features.parser.ui.contribution1"
            nameFilter="*.csv"
            objectClass="org.eclipse.core.resources.IFile">
         <menu
               id="es.upv.dsic.issi.multiple.features.parser.ui.menu1"
               label="MULTIPLE"
               path="additions">
            <separator
                  name="group1">
            </separator>
         </menu>
         <action
               class="es.upv.dsic.issi.multiple.features.parser.ui.popup.actions.ParseFile"
               enablesFor="1"
               id="es.upv.dsic.issi.multiple.features.parser.ui.popup.actions.ParseFile"
               label="Parse CSV file"
               menubarPath="es.upv.dsic.issi.multiple.features.parser.ui.menu1/group1">

         </action>
      </objectContribution>
   </extension>

</plugin>
```

Listado 4 Ejemplo de conexión a un punto de extensión de un menú popup

6.2.2.2.2 Dependencias

Como plugin de Eclipse, este plugin depende de:

- org.eclipse.ui – Proporciona el entorno de ejecución del entorno gráfico básico de Eclipse.
- org.eclipse.ui.ide – Proporciona acceso a algunas clases fundamentales del entorno, como cuadros de diálogo predefinidos.
- org.eclipse.core.runtime – Proporciona el soporte básico de la plataforma. org.eclipse.core.resources – Permite tratar con los recursos del workspace.
- es.upv.dsic.issi.multiple.features – Permite tratar con instancias del metamodelo pivote basado en cardinalidades .
- es.upv.dsic.issi.multiple.features.bridges – Proporciona la lógica para parsear y proyectar los documentos CSV nativos.

6.2.2.2.3 Descripción de paquetes y clases

❖ **Paquete es.upv.dsic.issi.multiple.features.parser.ui**

Este paquete únicamente contiene la clase activadora del plugin, FeaturesParserUIPlugin, que controla el ciclo del vida del plugin. En este caso, el código de esta clase ha sido generado de forma automática y contiene la implementación estándar.

❖ **Paquete es.upv.dsic.issi.multiple.features.parser.ui.popup.actions**

Este paquete contiene la clase ParseFile, que se ejecutará al seleccionar la opción correspondiente en el menú contextual que aparecerá al ejecutar el plugin. Todas implementan la interfaz *org.eclipse.ui.IObjectActionDelegate*, implementando de forma similar los métodos run() y selectionChanged() (hereadados de *org.eclipse.ui.IActionD-elegate).*

Método selectionChanged(. . .)

Este método se invoca cada vez que la selección de recursos del *workspace* varía y se invoca una determinada acción. Es en este método donde se deben recuperar los datos que interesen de la selección, y almacenarla en los atributos de clase para poder ser accedidos posteriormente en el método run().

Una implementación típica de este método (que es común a las tres clases del paquete) se muestra en el Listado 5. En esta implementación, se consulta el primer elemento de la selección (el menú contextual sólo estará disponible en caso de que se seleccione un único elemento), y se almacena en la variable file en caso de que la selección sea un fichero del *workspace* (instancia de *IFile*).

```
public void selectionChanged(IAction action, ISelection selection) {
    file = null;
    if(selection instanceof IStructuredSelection)
    {
        IStructuredSelection sel = (IStructuredSelection)selection;
        Object selElem = sel.getFirstElement();
        if(selElem instanceof IFile)
            file = (IFile)selElem;
    }
}
```

Listado 5 Método selectionChanged(…)

Método ParseFile.run(. . .)

Este es el método que invocará al parser del archivo CSV de forma manual. Fundamentalmente, llamará al parser pasando como argumento el fichero seleccionado, creará un nuevo recurso de EMF, al que le añadirá el resultado del proceso anterior, y salvará el recurso en disco, tal y como muestra el fragmento de código mostrado en el Listado 6.

```
public void run(IAction action) {
    Shell shell = new Shell();
    try {

        FeatureModel model = FeaturesActivator.getDefault().createFeatureModel(file);
        Resource resource = new ResourceSetImpl().createResource(URI.createPlatformResourceURI
                (file.getFullPath().removeFileExtension().addFileExtension("features").toPortableString(),false));
        resource.getContents().add(model);
        resource.save(Collections.EMPTY_MAP);


    } catch (InitialiseException e) {
        MessageDialog.openError(
                shell,
                "MULTIPLE CSV parser",
                "Error creating the model.");
        e.printStackTrace();
    } catch (CoreException e) {
        MessageDialog.openError(
            shell,
            "MULTIPLE CSV parser",
            "Error opening the source file.");
        e.printStackTrace();
    } catch (IOException e) {
        MessageDialog.openError(
                shell,
                "MULTIPLE CSV parser",
                "Error saving file.");
        e.printStackTrace();
    }

}
```

Listado 6 Método run(…)

6.2.3 Transformaciones en QVT-Relations

En este apartado se detallan las reglas de transformación empleadas entre el dominio origen (basado en cardinalidades) y el dominio destino (FaMa). Para la notación gráfica se han empleado la sintaxis gráfica definida en [51]. El código completo de las transformaciones se adjunta en el Anexo C.

6.2.3.1 Regla Model2Model

La regla inicial de transformación crea, tal como muestra la regla *Model2Model* (Figura 6.14), un elemento *DocumentRoot*, un elemento *FeatureModelType* y un elemento *GeneralFeature* para cada *FeatureModel* de la instancia origen. El elemento *DocumentRoot* es la raíz del modelo destino. Por su parte, el elemento *FeatureModelType* es el que contendrá todos los elementos del modelo de características. El elemento *GeneralFeature* representa la *feature* raíz del modelo, creada a partir de la raíz del modelo origen.

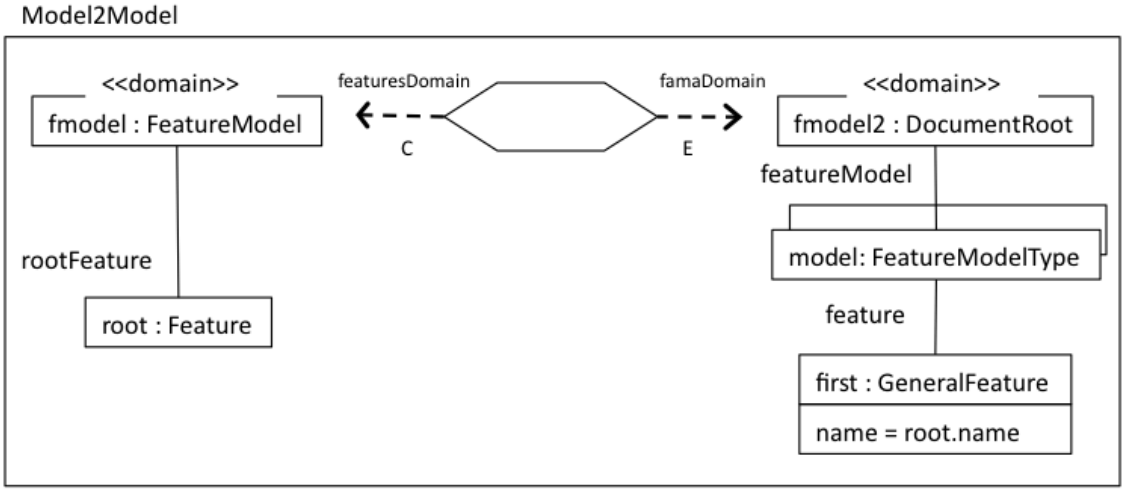


Figura 6.14 Regla Model2Model para Medini-QVT

6.2.3.2 Regla StructuralRelationship2BinaryRelation

La regla *StructuralRelationship2BinaryRelation* (Figura 6.15), especifica que, si existen en el modelo origen *features* relacionadas mediante elementos del tipo *StructuralRelationship*, se crearán en el modelo destino, las *features* y relaciones equivalentes. Esto significa que, para cada *feature* en origen que contenga relaciones del tipo *StructuralRelationship*, se creará una *GeneralFeature* con las *BinaryRelation* correspondientes en destino. Cada *BinaryRelation* contendrá una *solitaryFeature* que se corresponde al elemento *to* de la *relationship* origen y un elemento *Cardinality* que reflejará las cardinalidades máximas y mínimas establecidas por la relación estructural en origen (*lowerBound* y *upperBound*).

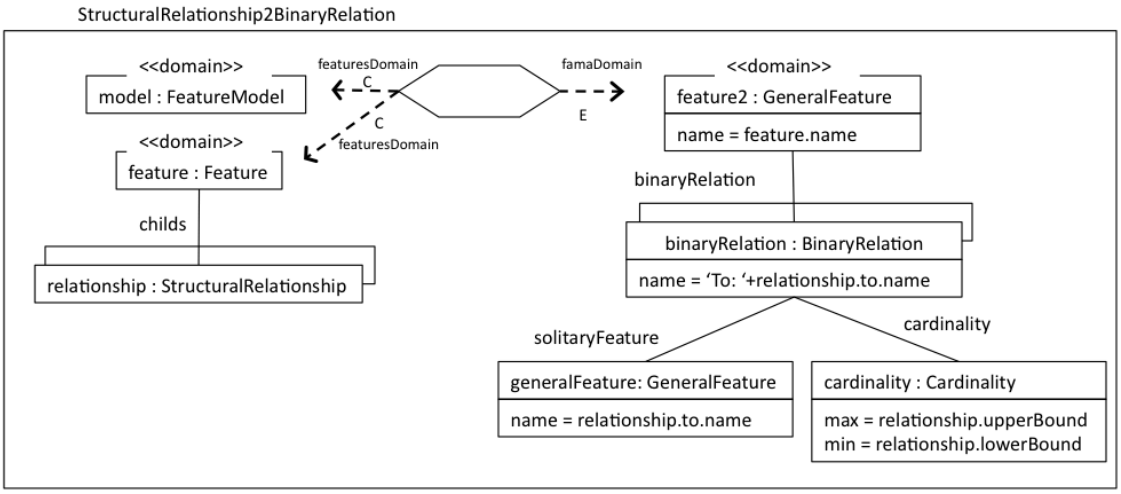
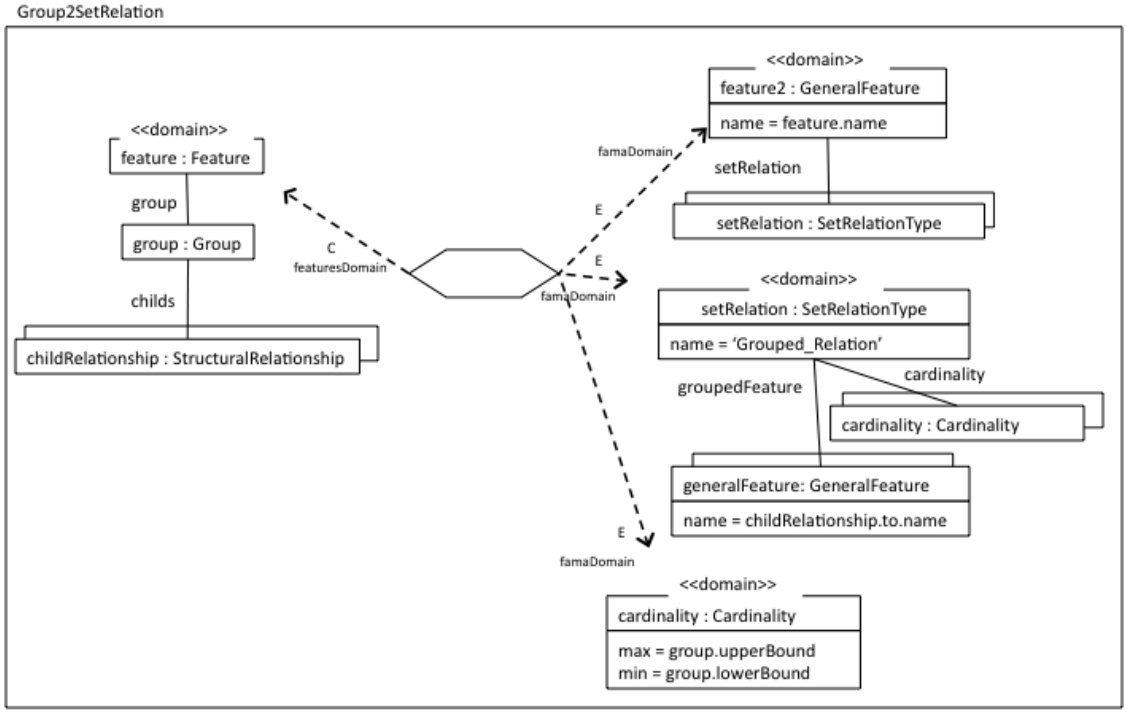


Figura 6.15 Regla StructuralRelationship2BinaryRelation en MediniQVT

6.2.3.3 Regla Group2SetRelation

La regla *Group2SetRelation* (Figura 6.16) crea en el dominio destino, para cada *feature* del dominio origen que contenga un elemento del tipo *Group* con relaciones del tipo *StructuralRelationship:*

- Una *GeneralFeature* cuyo nombre es el nombre de la *feature* origen
- Para cada *GeneralFeature,* las *SetRelationType* de nombre '*Grouped_Relation'*
- Para cada *SetRelationType*, los elementos *GeneralFeature* cuyo nombre será el nombre del elemento *to* de la relación origen (*childRelationship.to.name*)
- Cada *SetRelationType* tendrá un elemento *Cardinality*, cuyos valores para la cardinalidad máxima y mínima serán los establecidos por el elemento *Group* origen (*lowerBound* y *upperBound*).



6.2.3.4 Regla Excludesrelationship2ExcludesType

La regla *ExcludesRelationship2ExcludesType* (Figura 6.17) creará, para cada elemento del tipo *Excludes* (que contiene un elemento *from* y un elemento *to*) en el dominio origen, un elemento *ExcludesType* en destino, siendo el valor de su atributo excludes el nombre del elemento *from* (*from.name*) y el valor de *features* el nombre del elemento *to* de origen (*to.name*). Estas relaciones se crearán en el elemento *FeatureModelType*, por ello se comprobará primero, que este elemento ya existía.

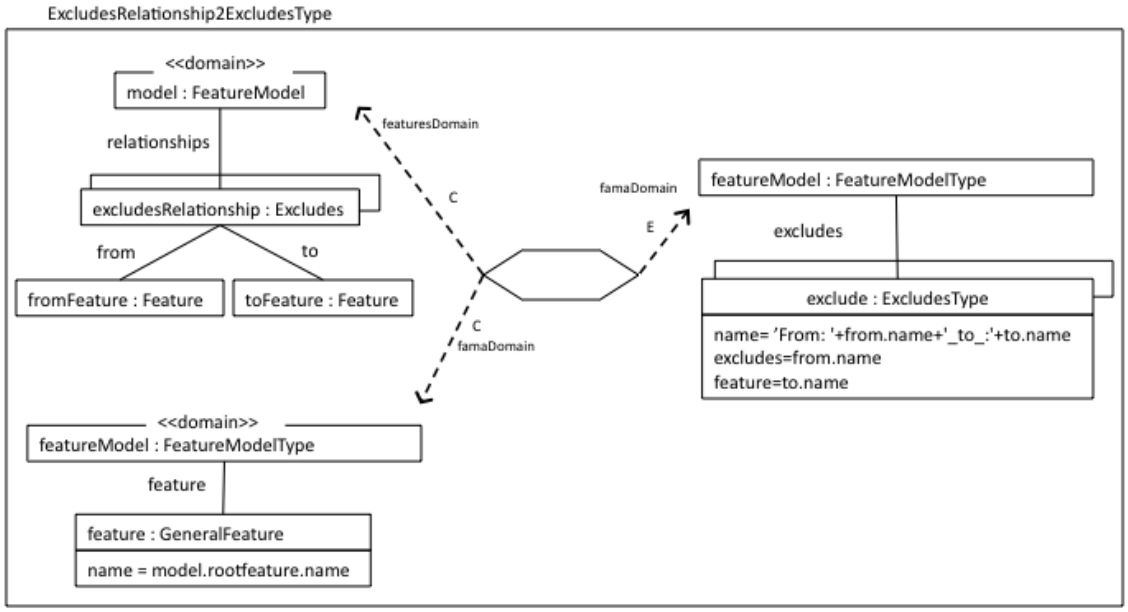


Figura 6.17 Regla ExcludesRelationship2ExcludesType en MediniQVT

6.2.3.5 Regla IncludesRelationship2RequiresType

La regla *IncludesRelationship2RequiresType* (Figura 6.18) creará, para cada elemento del tipo *Includes* (que contiene un elemento *from* y un elemento *to*) en el dominio origen, un elemento *RequiresType* en destino, siendo el valor de su atributo *requires* el nombre del elemento *from* (*from.name*) y el valor de *features* el nombre del elemento *to* de origen (*to.name*). Estas relaciones se crearán en el elemento *FeatureModelType*, por ello se comprobará primero, que este elemento ya existía.

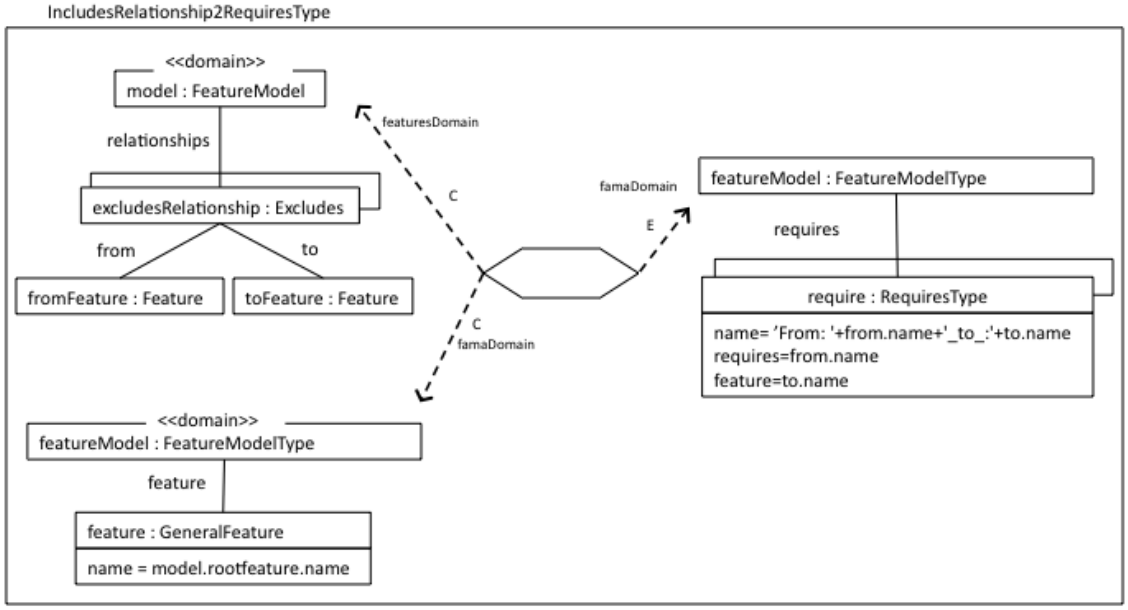


Figura 6.18 Regla IncludesRelationship2RequiresType en MediniQVT

6.2.4 Análisis automático del modelo de características

6.2.4.1 Plugin es.upv.dsic.issi.multiple.fama.bridges

6.2.4.1.1 Descripción

En este plugin se implementan los métodos que nos permitirán usar las operaciones proporcionadas por las librerías de FaMa para analizar nuestro modelo semánticamente.

6.2.4.1.2 Dependencias

Este plugin depende de:

- org.eclipse.ui
- org.eclipse.core.runtime
- org.eclipse.core.resources
- org.eclipse.ui.console
- es.us.isa.FaMaSDK (1.1.1) – Librería proporcionada por FaMa en la que se encuentran todos los *solvers* y operaciones para analizar el modelo

6.2.4.1.3 Descripción de paquetes y clases

- **Paquete es.upv.dsic.issi.multiple.fama.bridges**

Este paquete únicamente contiene la clase FamaBridgesActivator, que es la clase activadora del plugin. Nos permitirá iniciar el plugin y acceder a su funcionalidad. No contiene información extra a parte de la generada automáticamente por Eclipse.

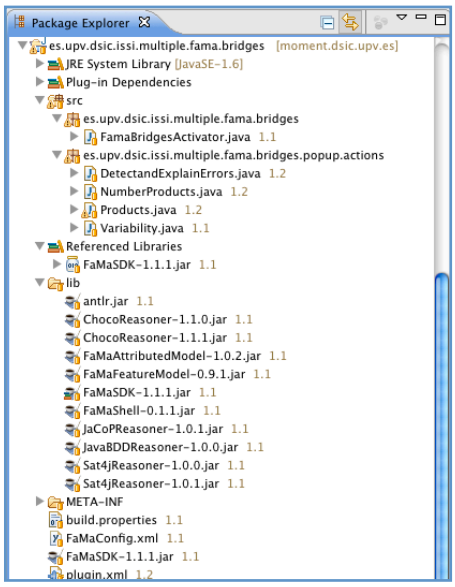


Figura 6.19 Proyecto es.upv.dsic.issi.multiple.fama.bridges

- **Paquete es.upv.dsic.issi.multiple.fama.bridges.popup.actions**

En este paquete encontramos la implementación de las acciones que se llevarán a cabo desde el menú contextual que aparecerá al ejecutar el plugin.

- **Paquete es.upv.dsic.issi.multiple.features.parser.ui.popup.actions**

Este paquete contiene cuatro clases clases que implementan las acciones que serán ejecutadas al seleccionar las distintas opciones. Todas implementan la interfaz *org.eclipse.ui.IObjectActionDelegate*, implementando de forma similar los métodos run() y selectionChanged() (hereadados de *org.eclipse.ui.IActionDelegate).*

Método selectionChanged(. . .)

El método selectionChanged() es el que se ejecuta cuando seleccionamos un elemento del *workspace* distinto. Se encarga de recuperar los datos del recurso seleccionado para poder acceder a ellos desde el método run() que ejecutará la funcionalidad seleccionada. Su implementación es la misma que en otros plugins anteriormente descritos (ver Listado 5)

Método NumberProducts.run(. . .)

Seleccionando, mediante el menú contextual esta operación de análisis sobre los archivos .xml que contendrán el modelo analizable por FaMa, obtendremos el número de productos contenidos en el modelo de características. Para ejecutar cualquier operación de análisis de extracción de datos, primero hay que comprobar que el modelo es válido.

En el Listado 7 se muestra la implementación de la lógica para usar las librerías FaMa para obtener el número de productos.

```
public void run(IAction action) {
    ConsolePlugin.getDefault().getConsoleManager().addConsoles(new IConsole[]{console});
    redStream.setColor(Display.getDefault().getSystemColor(SWT.COLOR_RED));
    blackStream.setColor(Display.getDefault().getSystemColor(SWT.COLOR_BLACK));

    blackStream.println("WELCOME TO THE FAMA ANALYSIS CONSOLE");

    // The main class is instantiated
    QuestionTrader qt = new QuestionTrader();

    String filename=file.getLocation().toString();
    blackStream.println(filename);
    // A feature model is loaded
    try{
        VariabilityModel fm = qt.openFile(filename);
        qt.setVariabilityModel(fm);

        // //////// VALID QUESTION + NUMBER PRODUCTS QUESTION ///////////
        ValidQuestion vq = (ValidQuestion) qt.createQuestion("Valid");
        qt.ask(vq);
        if (vq.isValid()) {
            NumberOfProductsQuestion npq = (NumberOfProductsQuestion) qt
                    .createQuestion("#Products");
            qt.ask(npq);
            blackStream.println("The number of products is: "
                    + npq.getNumberOfProducts());
        } else {
            redStream.println("Your feature model is not valid");
        }
    } catch (Exception e) {
        redStream.println("Error in processing the file");
        e.printStackTrace();
    }
}
```

Listado 7 Método NumberProducts.run(…)

Método Products.run(. . .)

Este es el método que invocará a la operación Products proporcionada por las librerías FaMa. El resultado serán todos los productos (combinaciones posibles de *features)* que contendrá nuestro modelo. Esta información se mostrará por la consola.

En el Listado 8 se muestra la implementación de la lógica para usar las librerías FaMa para obtener los productos.

```
public void run(IAction action) {
    ConsolePlugin.getDefault().getConsoleManager()
            .addConsoles(new IConsole[] { console });
    redStream.setColor(Display.getDefault().getSystemColor(SWT.COLOR_RED));
    blackStream.setColor(Display.getDefault().getSystemColor(
            SWT.COLOR_BLACK));

    blackStream
            .println("WELCOME TO THE FAMA ANALYSIS CONSOLE. It will maybe took some time to provide answers");

    // The main class is instantiated
    QuestionTrader qt = new QuestionTrader();

    // A feature model is loaded
    String filename = file.getLocation().toString();
    blackStream.println(filename);
    // A feature model is loaded
    try {
        VariabilityModel fm = qt.openFile(filename);
        qt.setVariabilityModel(fm);

        // //////// VALID QUESTION + PRODUCTS ///////////
        ValidQuestion vq = (ValidQuestion) qt.createQuestion("Valid");
        qt.ask(vq);
        if (vq.isValid()) {
            ProductsQuestion pq = (ProductsQuestion) qt
                    .createQuestion("Products");
            qt.ask(pq);
            Collection<? extends GenericProduct> products = pq
                    .getAllProducts();
            Iterator<? extends GenericProduct> iterator = products.iterator();
            blackStream.println("The products are: ");
            while (iterator.hasNext()) {
                GenericProduct f = (GenericProduct) iterator.next();
                blackStream.println(f.getElements().toString());
            }
        } else {
            redStream.println("Your feature model is not valid");
        }
    } catch (Exception e) {
        redStream.println("Error in processing the file");
        e.printStackTrace();
    }

}
```

Listado 8 Método Products.run(…)

Método DetectAndExplainErrors.run(. . .)

Este método implementa la funcionalidad que nos va a permitir usar las operaciones de búsqueda de errores y explicación de los mismos que pueda contener nuestro modelo de características. Seleccionando el archivo correspondiente desde el *workspace*, esta operación nos mostrará por consola los resultados.

En el Listado 9 se muestra el método que implementa la lógica para usar las librerías FaMa y obtener los posibles errores y su explicación.

```
public void run(IAction action) {
    ConsolePlugin.getDefault().getConsoleManager()
            .addConsoles(new IConsole[] { console });
    redStream.setColor(Display.getDefault().getSystemColor(SWT.COLOR_RED));
    blackStream.setColor(Display.getDefault().getSystemColor(
            SWT.COLOR_BLACK));

    blackStream
            .println("WELCOME TO THE FAMA ANALYSIS CONSOLE. It will maybe took some time to provide answers");

    // The main class is instantiated
    QuestionTrader qt = new QuestionTrader();

    String filename = file.getLocation().toString();
    blackStream.println(filename);
    // A feature model is loaded
    try {
        VariabilityModel fm = qt.openFile(filename);
        qt.setVariabilityModel(fm);
        DetectErrorsQuestion q = (DetectErrorsQuestion) qt
                .createQuestion("DetectErrors");
        q.setObservations(fm.getObservations());
        qt.ask(q);

        Collection<Error> errors = q.getErrors();
        ExplainErrorsQuestion qe = (ExplainErrorsQuestion) qt
                .createQuestion("Explanations");
        qe.setErrors(errors);
        qt.ask(qe);
        errors = qe.getErrors();

        Iterator<Error> it = errors.iterator();
        while (it.hasNext()) {
            Error e = it.next();
            Collection<Explanation> explanations = e.getExplanations();
            Iterator<Explanation> itExp = explanations.iterator();
            blackStream.println("Explanations for error " + e);
            while (itExp.hasNext()) {
                Explanation exp = itExp.next();
                Collection<GenericRelation> relations = exp.getRelations();
                Iterator<GenericRelation> itRel = relations.iterator();
                while (itRel.hasNext()) {
                    GenericRelation rel = itRel.next();
                    blackStream.println(rel.getName());
                }
            }
        }

    } catch (Exception e) {
        redStream.println("Error in processing the file");
        e.printStackTrace();
    }
}
```

Listado 9 Método DetectandExplainErrors.run(…)

Método Variability.run(. . .)

Este método implementa la funcionalidad que nos va a permitir usar las operaciones proporcionadas por FaMa para calcular el grado de variabilidad de nuestro modelo de características. Al igual que las otras acciones implementadas en este paquete, su ejecución se llevará a cabo manualmente desde el menú contextual y los resultados se mostrarán por consola.

En el Listado 10 se muestra el método que implementa esta lógica.

```
public void run(IAction action) {
    ConsolePlugin.getDefault().getConsoleManager()
            .addConsoles(new IConsole[] { console });
    redStream.setColor(Display.getDefault().getSystemColor(SWT.COLOR_RED));
    blackStream.setColor(Display.getDefault().getSystemColor(
            SWT.COLOR_BLACK));

    blackStream
            .println("WELCOME TO THE FAMA ANALYSIS CONSOLE. It will maybe took some time to provide answers");
    // The main class is instantiated
    QuestionTrader qt = new QuestionTrader();

    String filename = file.getLocation().toString();
    blackStream.println(filename);
    // A feature model is loaded
    try {

        // //////// VALID QUESTION + VARIABILITY ///////////
        ValidQuestion vq = (ValidQuestion) qt.createQuestion("Valid");
        qt.ask(vq);
        if (vq.isValid()) {
            VariabilityQuestion va = (VariabilityQuestion) qt
                    .createQuestion("Variability");
            qt.ask(va);
            System.out.println("The variability degree of the model is: "
                    + va.getVariability());
        } else {
            redStream.println("Your feature model is not valid");
        }
    } catch (Exception e) {
        redStream.println("Error in processing the file");
        e.printStackTrace();
    }
}
```

Listado 10 Método Variability.run(…)

6.3 Conclusiones

En este capítulo se muestra la estructura e implementación de nuestro framework, que extenderá a MULTIPLE. Los componentes que se añaden, desarrollados como plugins, nos permitirán llevar a cabo el análisis automatizado de un modelo de características. Estos plugins, al haber sido desarrollados con Eclipse, se integrarán con la funcionalidad ofrecida por MULTIPLE, siguiendo un enfoque dirigido por modelos. Además, se modifican algunos de los componentes ya existentes en MULTIPLE para añadirles funcionalidad. Este es el caso del metamodelo de características definido por la herramienta. En nuestro desarrollo, añadimos las clases necesarias al metamodelo para que los elementos de sus instancias (modelos de características) puedan almacenar información extra en forma de anotaciones. También se hacen los cambios necesarios para que el editor gráfico de líneas de producto existente, permita representar y editar instancias de este nuevo metamodelo.

Una de nuestras aportaciones a MULTIPLE es el componente *parser,* que creará una instancia del metamodelo pivote a partir del modelo de características origen. Además, el *parser* incluye una consola en la que se mostrarán los resultados del análisis sintáctico del modelo. En segundo lugar, se definen las transformaciones en QVT-Relations que crearán, a partir del modelo de características conforme a nuestro metamodelo, un modelo equivalente instancia de FaMa. Por último, se implementa un componente que integrará el motor de análisis de FaMa a nuestro desarrollo, permitiendo realizar operaciones de análisis sobre el modelo instancia de FaMa y obteniendo los resultados de este análisis por consola.

7. Aplicación al caso de estudio

En este capítulo detallaremos el proceso de análisis del modelo de características industrial proporcionado por Rolls Royce utilizando nuestra herramienta. En primer lugar, mostraremos paso a paso cómo se ejecuta la herramienta y los resultados que se obtienen. Además, clasificaremos los errores encontrados durante las etapas de análisis (sintáctico y semántico) y propondremos soluciones a estos.

7.1 El proceso de análisis

7.1.1 El Modelo Origen

El modelo de características de Rolls Royce, que tomaremos como entrada, utiliza la notación propuesta en PLUSS [21] por Eriksson y Börstler. Este modelo contiene 1195 características o *features*, por lo que consideramos que es un modelo de características de gran escala. La Figura 7.1 muestra un extracto de la apariencia del modelo.

Module Name	Module Num	Absolute Num	Parent Num	Object Num	Object Heading	Feature Node	Variation	Requires	Prohibits
Behavioural Fe	0000706b	178	166	01/01/0007	Perform Quick Windmill Relight	Feature	Optional		
Behavioural Fe	0000706b	188	166	01/01/0008	Recover from Flameout (Auto Relight)	Feature	Optional	0000706b,233,5.2.2.2	
Behavioural Fe	0000706b	183	166	01/01/0009	Perform Engine Cranking for Maintenan	Feature	Common		0000706b,500,3.2
Behavioural Fe	0000706b	190	166	01/01/2010	Perform Pilot Initiated Engine Shutdowr	Feature	Common		
Behavioural Fe	0000706b	135	364	01:02	Set Engine Power Demand	Feature Group	Common		
Behavioural Fe	0000706b	392	135	01/02/0001	Obtain Engine Power Demand	Feature Group	Common		
Behavioural Fe	0000706b	393	392	1211	Obtain Throttle Lever Position	Feature	Common		
Behavioural Fe	0000706b	394	392	1212	Obtain Power Demand from Aircraft Co	Feature Group	Optional		
Behavioural Fe	0000706b	155	394	12121	Provide Digital Power Setting (aka ëAut	Feature	Optional		

Figura 7.1 Extracto modelo origen

Podemos observar, que este modelo está codificado como texto plano (una tabla) en el que las características se definen como campos de la tabla. A continuación exponemos lo que significa cada campo de la tabla:

- Module Name. Contiene el nombre del módulo de la SPL al que pertenece la *feature*.
- Module Number. Contiene el número del módulo de la SPL al que pertenece la *feature*.
- Absolute Number. Contiene el número que representa a la *feature* en el conjunto total de la SPL.
- Parent Number. Contiene el número de la *feature* padre.
- Object Number. Contiene el número que representa a la *feature* en la jerarquía de características.
- Object Heading. Contiene el nombre de la *feature*.
- Feature Node. Puede contener los valores “Feature” o “Feature Group”, que indican si la *feature* tiene hijos o es una hoja del árbol.
- Variation. Contiene el tipo de variación que representa a la *feature.* Puede tomar los valores “Common”, ”Optional”, ”Single”o ”Múltiple” como especifica la notación PLUSS.

- Requires. Contiene la *feature* o *features* que esta característica necesita cuando se selecciona (que también se seleccionarán).
- Prohibits. Contiene la *feature* o *features* que no pueden ser seleccionadas cuando esta característica se selecciona.

7.1.2 Analizando el modelo origen con nuestra herramienta

A continuación, listaremos los pasos que hemos seguido para analizar el modelo de características de Rolls Royce con nuestra herramienta.

7.1.2.1 El *workspace*

Al abrir la herramienta, nos encontramos con que nuestro *workspace* contiene un archivo proporcionado por la herramienta. Este archivo, **features2Fama.qvt** (figura 7.2), contiene las reglas QVT-Relations para transformar del dominio de nuestro metamodelo pivote al dominio de FaMa.

El siguiente paso es incluir en el *workspace,* el modelo de características que queremos analizar, en nuestro caso el de Rolls Royce. Este archivo se llama **RollsRoyce.csv** (Figura 7.2) y contiene la línea de productos de Rolls Royce en un archivo CSV cuyo contenido se ha mostrado en el apartado 7.1.1.

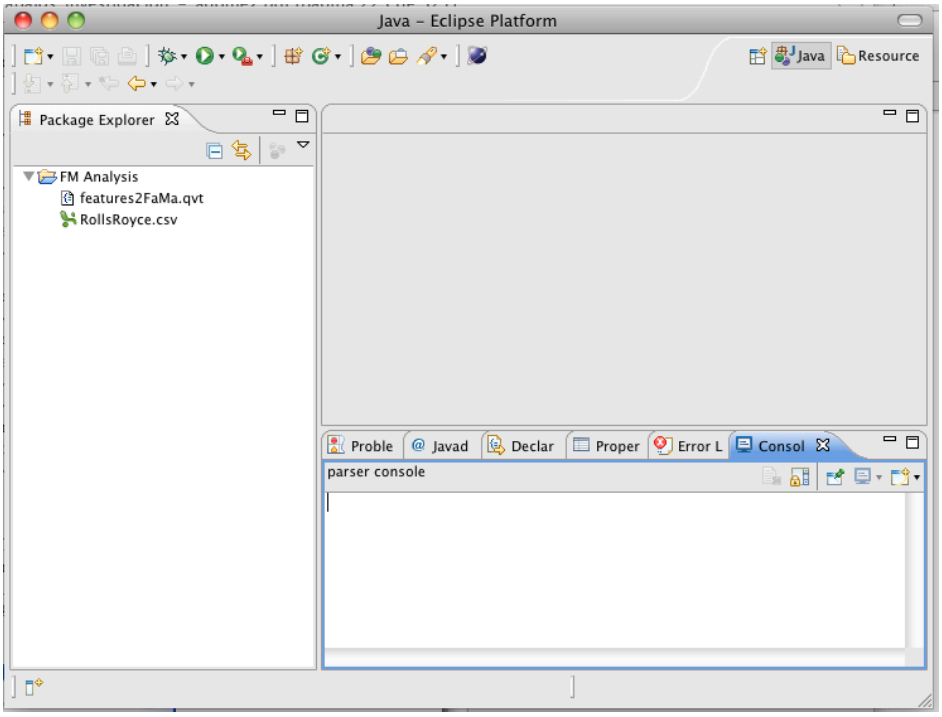


Figura 7.2 *Workspace* con archivos

7.1.2.2 Ejecución del parser y resultados del análisis sintáctico

Primero ejecutaremos el *parser*, que nos creará un nuevo modelo instancia del modelo pivote. Para ejecutar el *parser,* el usuario debe seleccionar esta opción desde el menú contextual (véase la Figura 7.3). Este modelo resultante, llamado **RollsRoyce.features,** es equivalente al modelo original. En la Figura 7.4 muestra una versión simplificada de este archivo (para ver cada tipo de relación) en el editor de modelos (Features Model Editor).

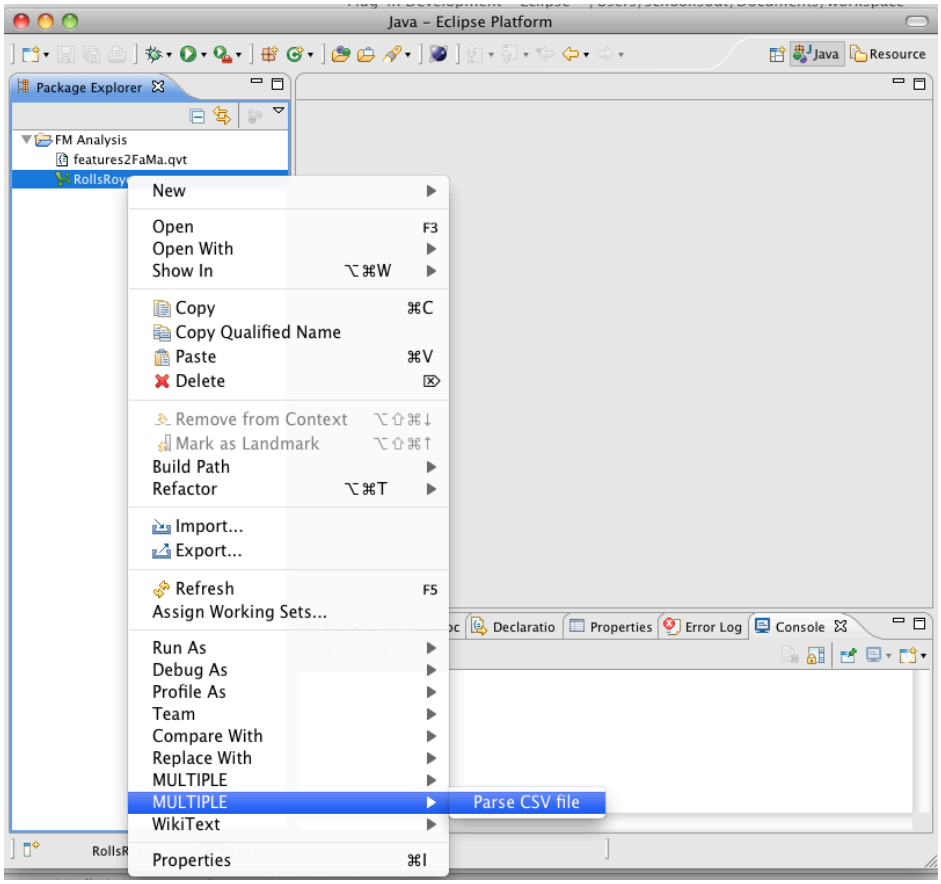


Figura 7.3 Editor de modelos basado en cardinalidades

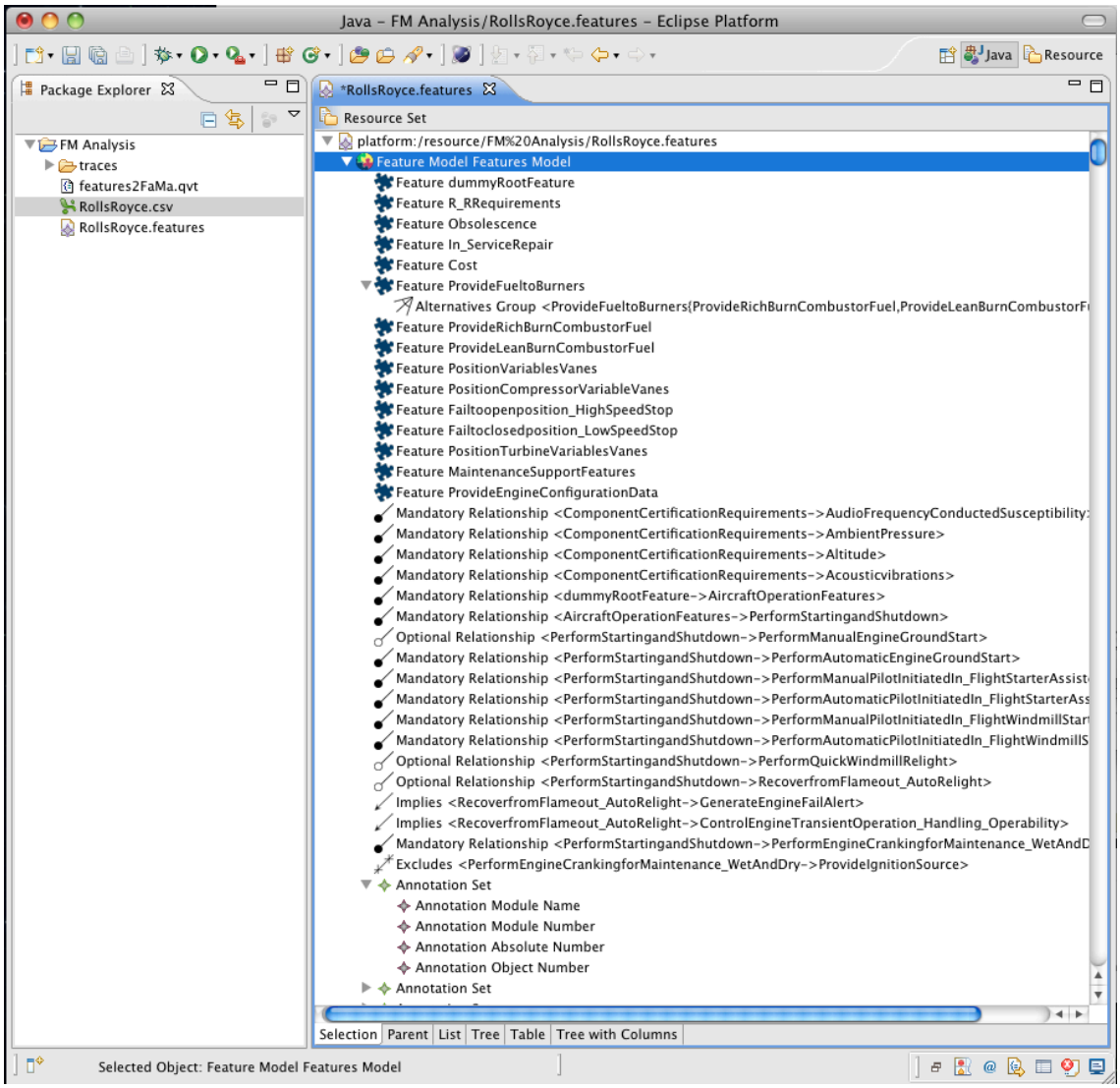


Figura 7.4 Editor de modelos Basado en cardinalidades

Además, se muestra por consola el resultado del análisis sintáctico del modelo. En él se listan todos los errores de forma detectados y qué medidas se han tomado para corregirlo. En la Figura 7.5 podemos ver como se lista el número de la línea del modelo en la que se ha encontrado el error, el motivo y la acción correctiva que se ha llevado a cabo.

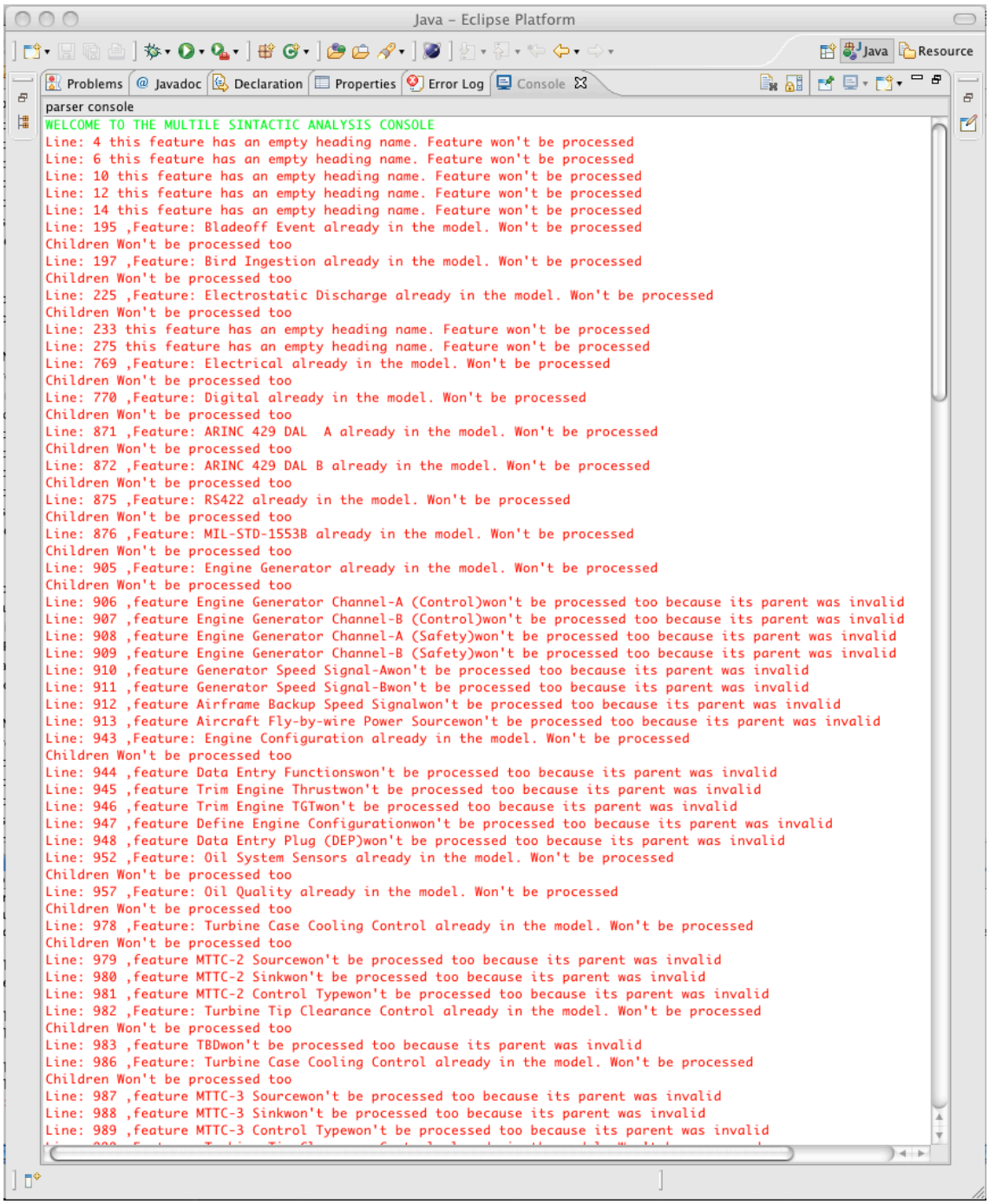


Figura 7.5 Resultados del análisis sintáctico mostrados por consola

7.1.2.3 Editor gráfico del modelo pivote basado en cardinalidades

Como nuestra herramienta implementaba un editor gráfico para la representación y edición de los modelo instancia de nuestro metamodelo pivote. Habiendo obtenido el archivo **RollsRoyce.features** podemos crear la representación gráfica del mismo. Para crear esta representación gráfica el usuario habrá de seleccionar la opción desde el menú contextual (véase la Figura 7.6). Esto nos va a permitir modificar el modelo desde el editor y guardarlo.

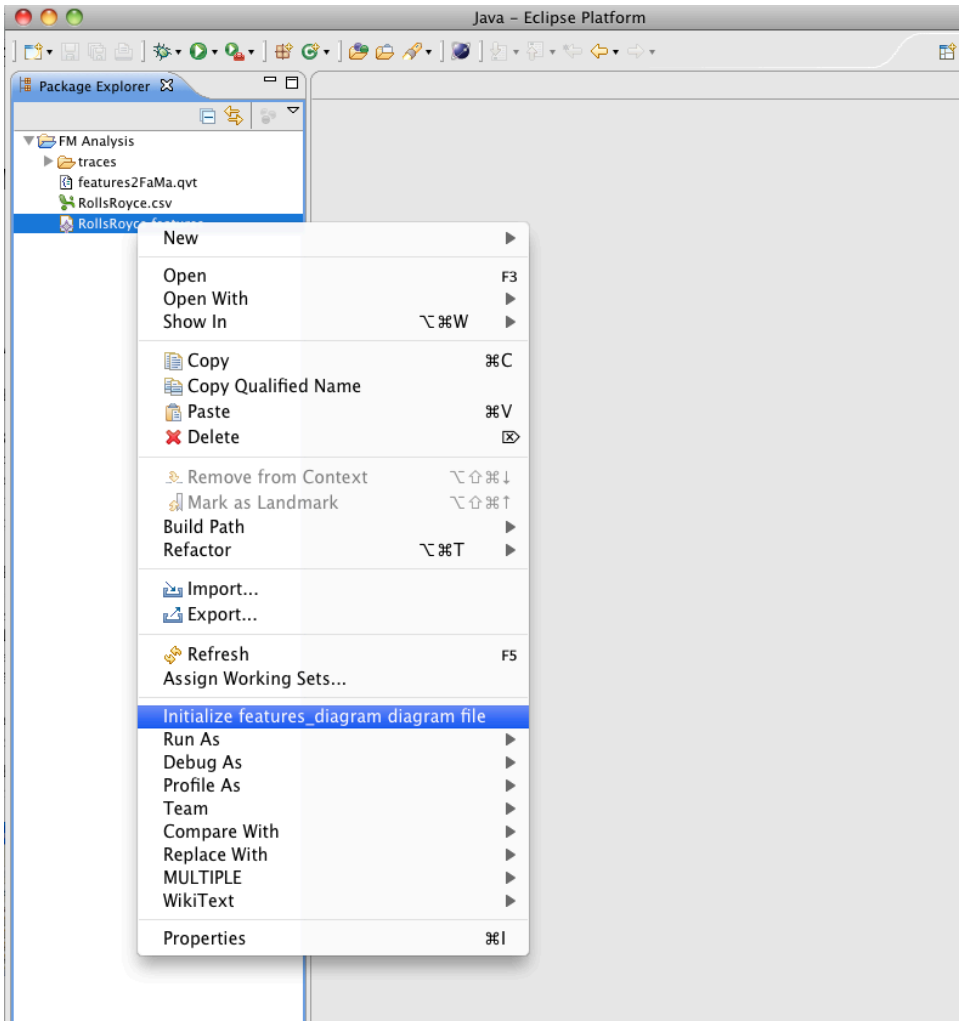


Figura 7.6 Creando la representación gráfica del modelo

Las figuras Figura 7.7 y Figura 7.8 se muestra el aspecto de nuestro modelo en el editor gráfico. Al ser un modelo con tantas características es imposible verlo de una sóla vez, hemos de navegar por el editor para ver las distintas partes del modelo. En la figura Figura 7.8 se aprecia mejor el tamaño del modelo de características.

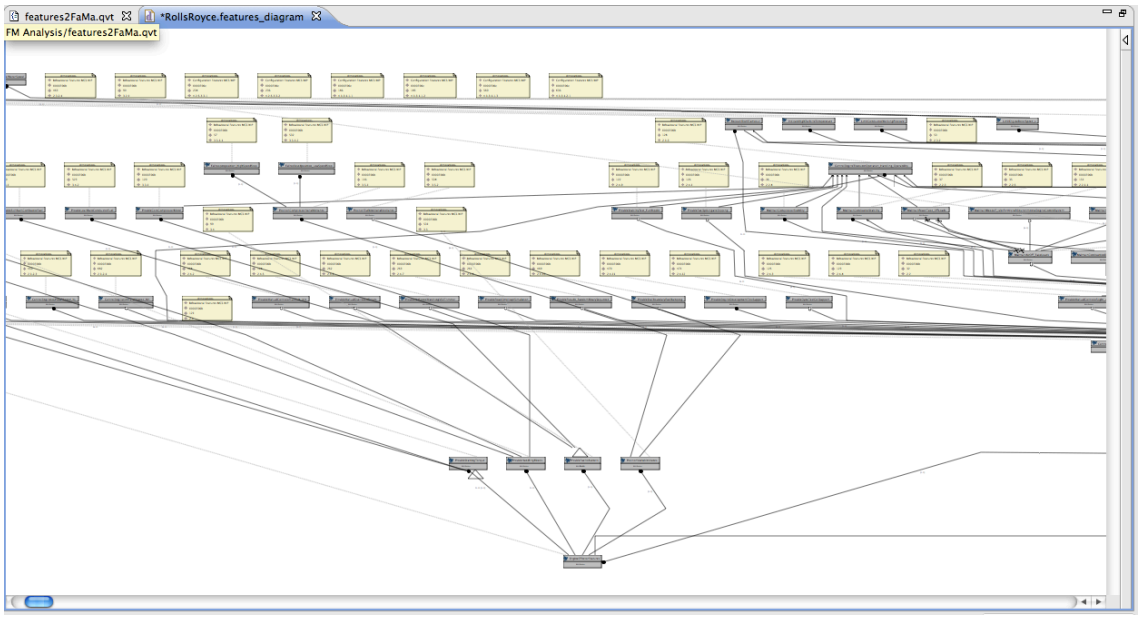


Figura 7.7 Modelo de características de Rolls Royce en el editor gráfico

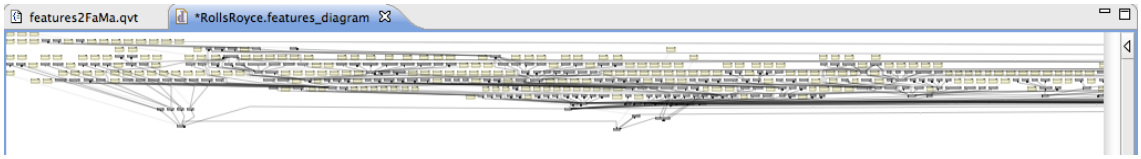


Figura 7.8 Modelo de características de Rolls Royce en el editor gráfico más completo

7.1.2.4 Ejecución de las transformaciones

Para poder ejecutar las transformaciones que crearán un modelo equivalente al nuestro como instancia de FaMa, el usuario ha de seleccionar Run as → 1 QVT Transformation como se muestra en la Figura 7.9. La interfaz usada para la ejecución de las transformaciones es la desarrolla en MULTIPLE, que utiliza el motor de transformaciones de Medini-QVT.

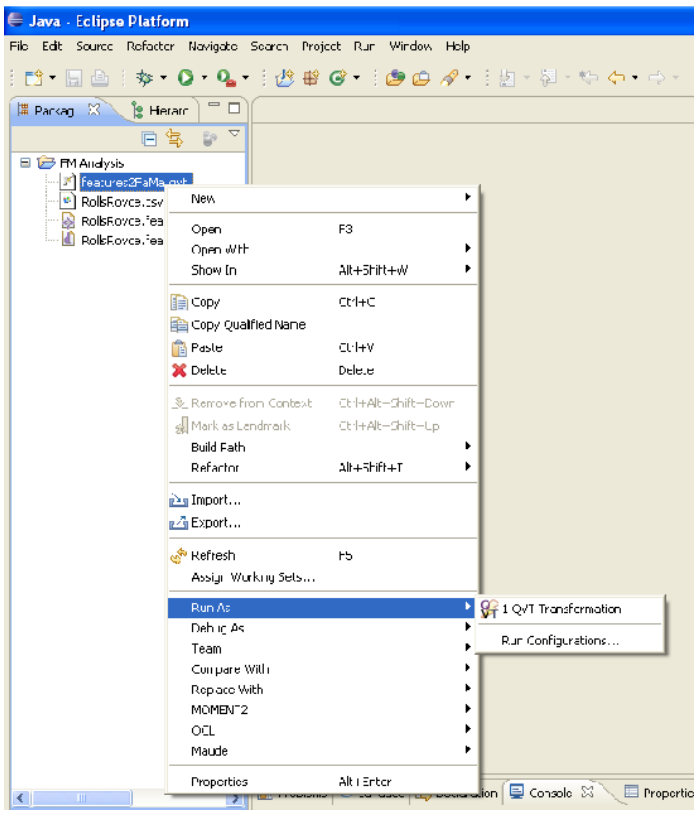


Figura 7.9 Ejecutando la transformación QVT

En la

Figura 7.10 se muestra el cuadro de diálogo empleado para configurar las transformaciones. El campo editable que se encuentra en su parte superior contiene el nombre que identifica la transformación seleccionada en el panel izquierdo. Este nombre se inicializa al nombre del la transformación seleccionada al abrir el cuadro de diálogo. El segundo campo, indica la ruta del archivo que contiene la transformación a ejecutar. Mediante el botón *Browse*, podemos seleccionar el archivo de la transformación a ejecutar (archivo «*.qvt»).

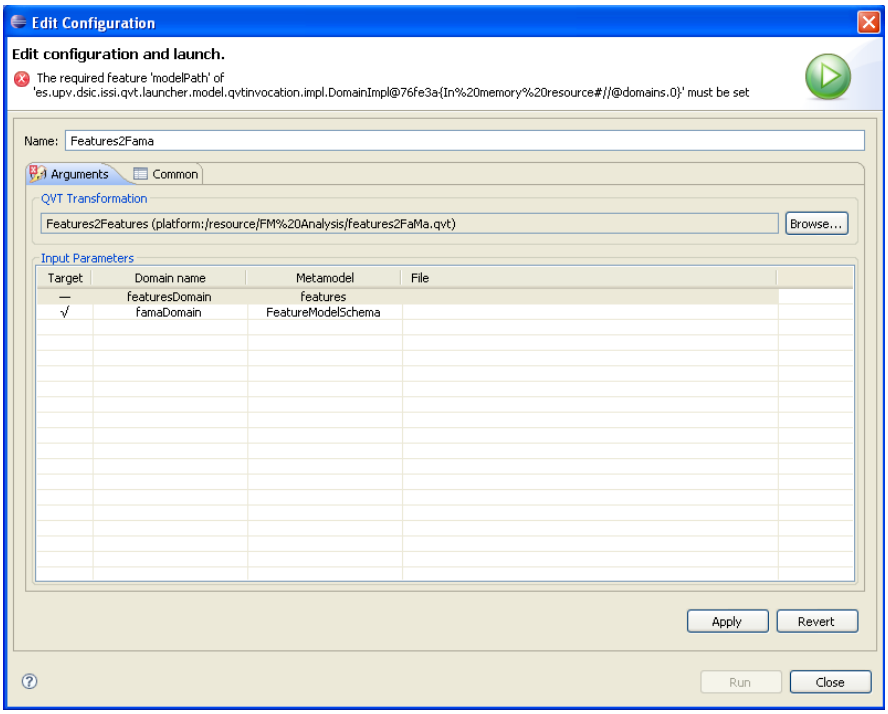
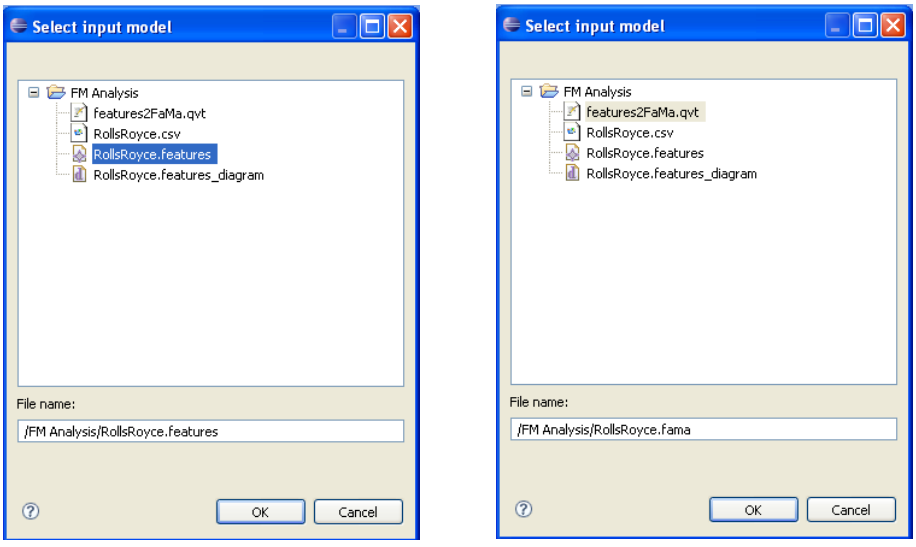


Figura 7.10 Configuración de la transformación

Una vez se ha seleccionado un archivo QVT, la tabla central puede emplearse para especificar qué archivos se corresponden con cada uno de los dominios de la transformación, tal y como la Figura 7.10 muestra. Las figuras Figura 7.11ª y Figura 7.11b muestran los cuadros de diálogo para especificar los dominios features y fama respectivamente.



(a) (b)

Figura 7.11 Archivo de entrada (a) y archivo de salida (b) de la transformación

En la Figura 7.12 se muestra el aspecto del cuadro de diálogo con toda la información que se necesita para ejecutar la transformación. La dirección de la transformación viene determinada por la marca en la celda correspondiente de la columna *target* del dominio destino. Por defecto, el dominio destino es el último

dominio definido en la transformación. Ya podemos darle al botón inferior **Run** para ejecutar la transformación.

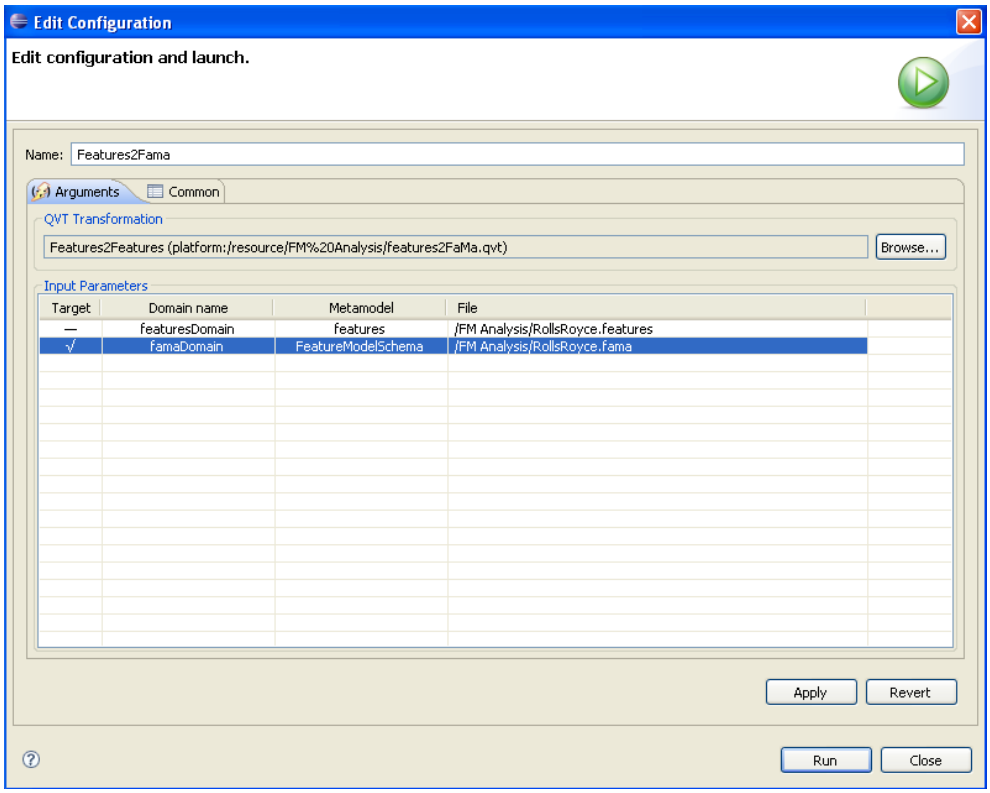


Figura 7.12 Configuración completa de la transformación

Como resultado obtenemos un archivo llamado **RollsRoyce.fama.** Como FaMa acepta archivos con extensión .fama como formato de los modelos de características de entrada, ya lo tenemos listo para poder ejecutar las operaciones de análisis proporcionadas por FaMa. En la Figura 7.13 se muestra el contenido del archivo FAMA obtenido.

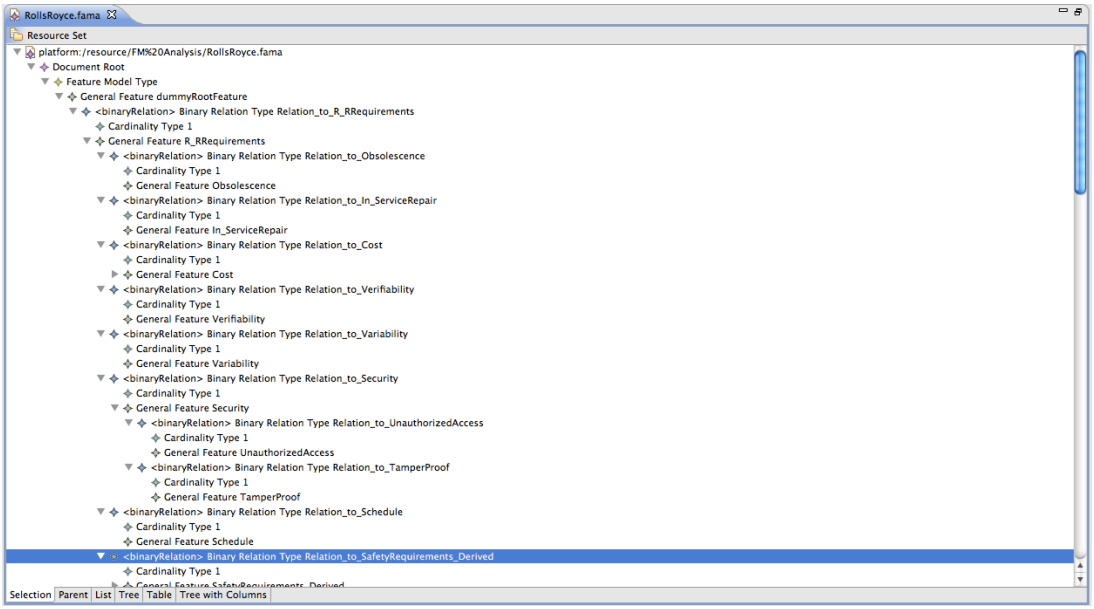


Figura 7.13 Contenido del documento FAMA del modelo de características

Figura 7.11 Contenido del documento FAMA del modelo de características

De la ejecución de las transformaciones, también obtenemos el modelo de trazas (**RollsRoyce.traces**). La Figura 7.14 muestra el modelo de trazas con el editor de trazabilidad por defecto. En él se muestran las correspondencias entre los dominios origen y destino.

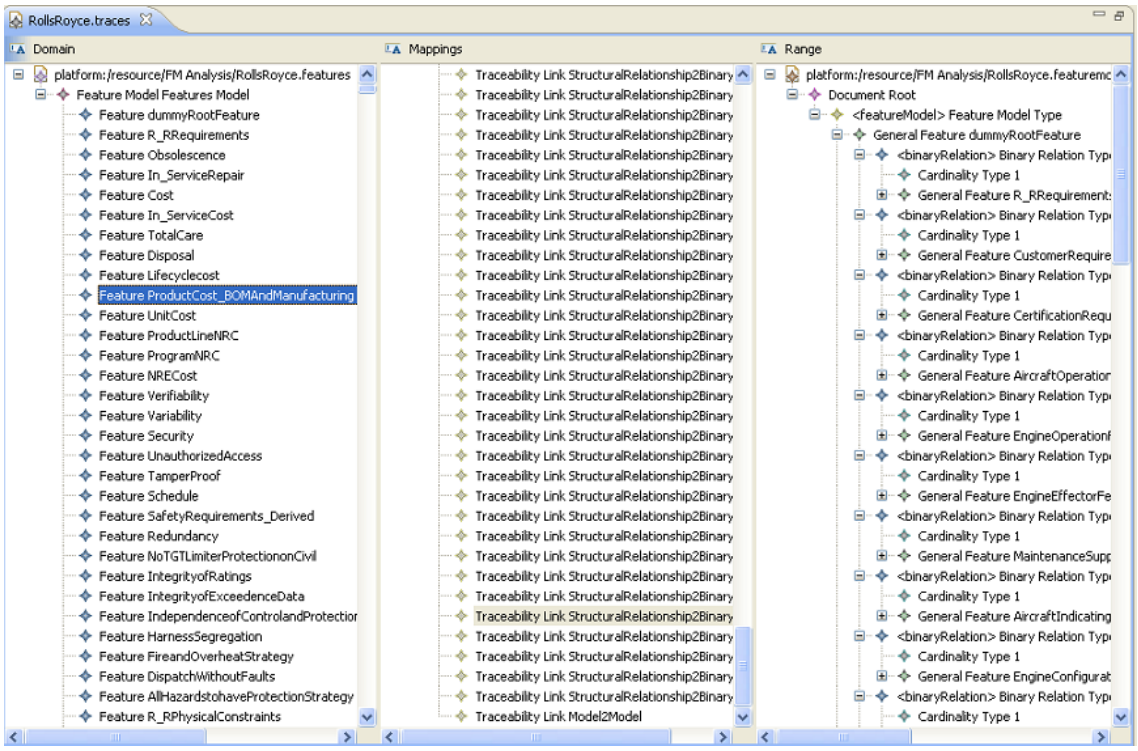


Figura 7.14 Editor de modelos de trazabilidad

7.1.2.5 Análisis con FaMa y resultados

Para ejecutar las operaciones de análisis de FaMa, hemos de hacerlo mediante el menú contextual (Figura 7.15). En la Figura 7.15 se muestran cuatro operaciones de análisis; detección y explicación de errores (*Detect and explain errors*), número de productos (*Number of products*) y productos potenciales (*Products*). Los resultados de estas operaciones de análisis irán mostrándose por nuestra consola (véanse figuras Figura 7.16, Figura 7.17 y Figura 7.18). En un primer desarrollo se implementaron más operaciones, como el cálculo del grado de variabilidad, de las características comunes y las variables. Pero estas operaciones no daban los resultados esperados (apartado 7.4) y para la versión ejecutable se dejaron aquellas operaciones que sí funcionaban bien.

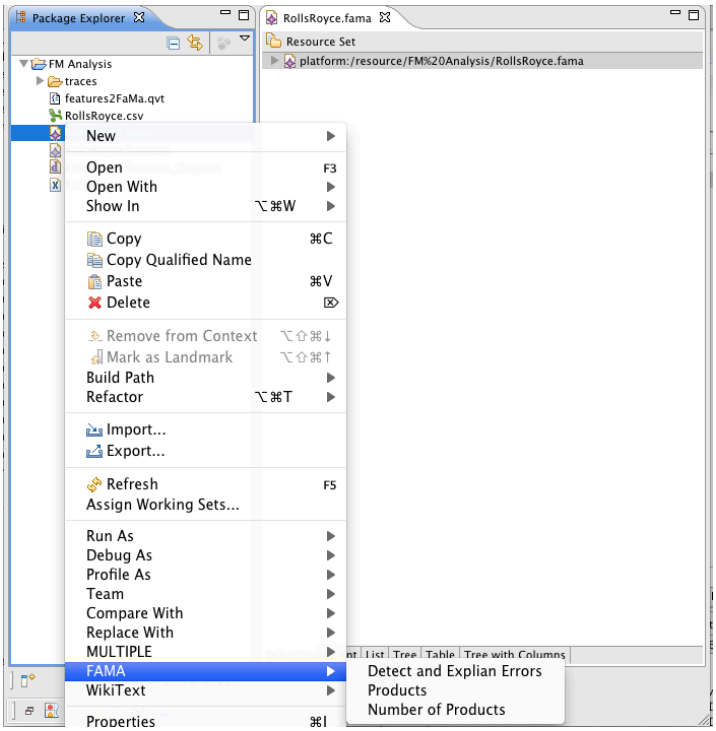


Figura 7.15 Ejecución de las operaciones de análisis de FaMa

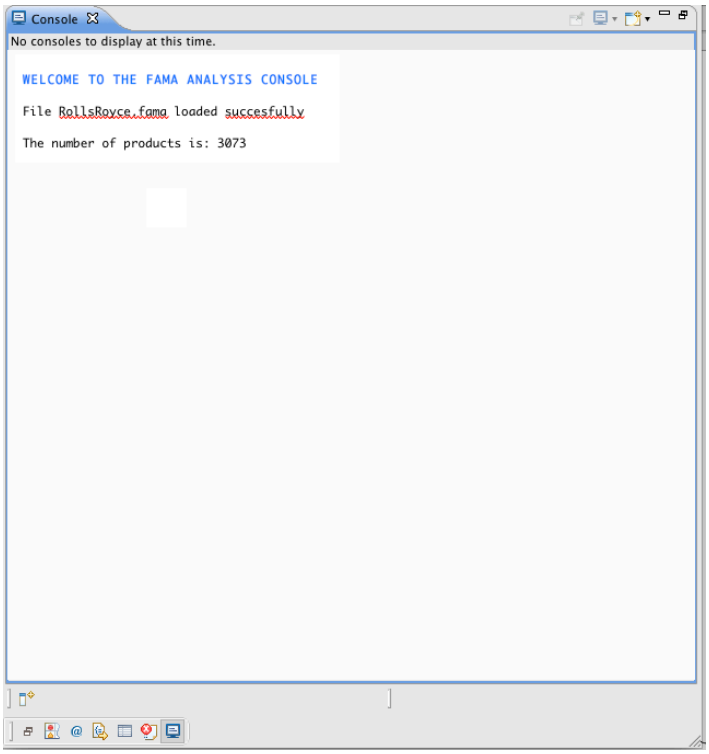


Figura 7.16 Consola Resultado de la operación Número de Productos

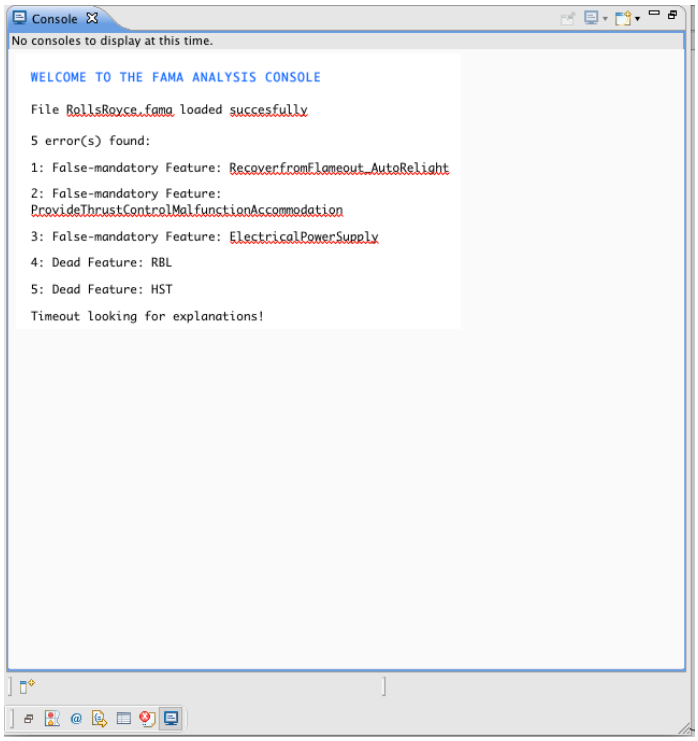


Figura 7.17 Consola resultado de la operación de detección y explicación de errores

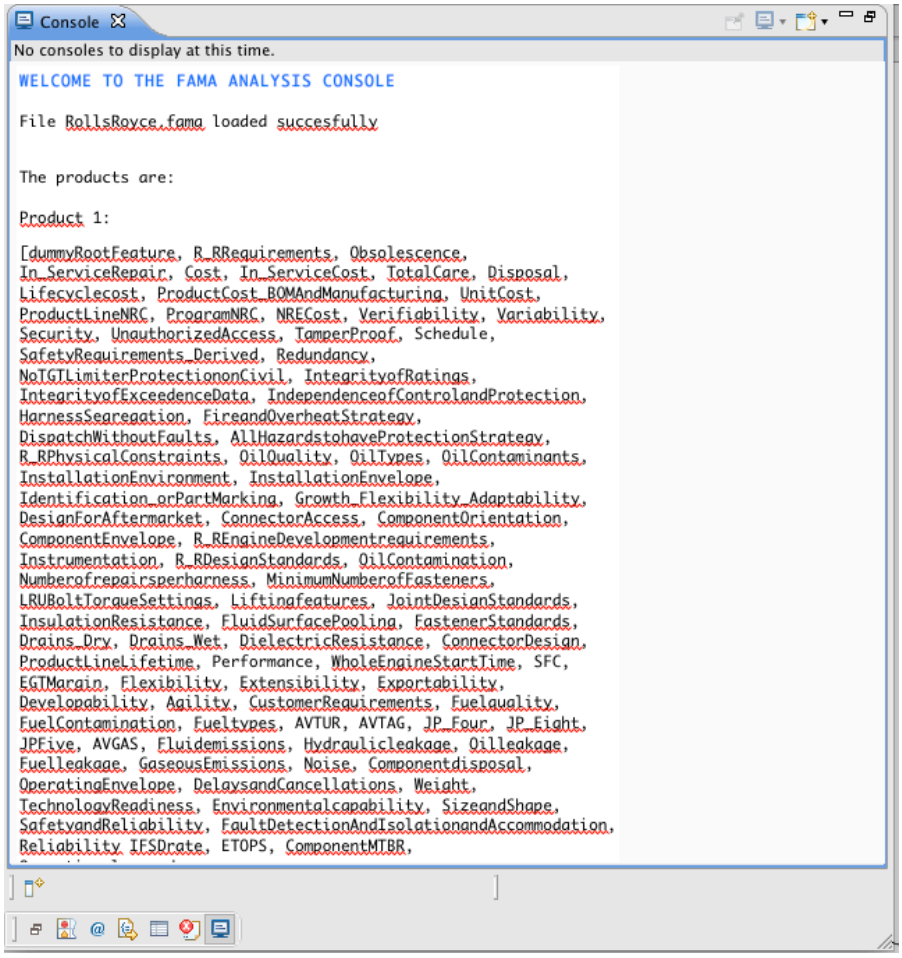


Figura 7.18 Consola resultado de la operación de análisis de cálculo de los productos potenciales

7.2 Interpretación de resultados obtenidos

En este apartado explicaremos los errores detectados durante el proceso de análisis, los clasificaremos, y propondremos soluciones.

7.2.1 Análisis sintáctico

Durante el proceso de adaptación del modelo origen en notación PLUSS a nuestra notación basada en cardinalidades, hemos detectado ciertos errores que hacen al modelo inconsistente. Estos errores pueden ser clasificados como sigue:

7.2.1.1 Características vacías (Empty Features)

Descripción

Hemos identificado que el modelo contiene características sin nombre (Figura 7.19a). Estas características 'vacías' y sus características padre, están definidas como *Mandatory*, de modo que aparecerán en cada uno de los productos de la SPL.

Cada característica del modelo ha de definir una funcionalidad que un usuario o cliente necesite para obtener un producto que se ajuste a sus necesidades. La presencia de estas características vacías no contribuye a describir la SPL.

Solución Propuesta

Hemos eliminado estas características del modelo. Además la herramienta muestra por la consola que se ha tomado esta medida y en qué línea del modelo se encontraba esta característica (Figura 7.19b).

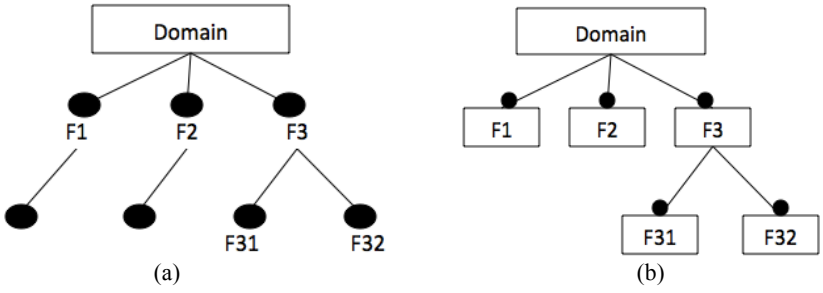


Figura 7.19 Características vacías extraídas del FM original (a) y su solución (b)

7.2.1.2 Características duplicadas

Descripción

El modelo original contiene algunas características duplicadas. Dado el tipo de modelo de entrada que tenemos, esto significa que existen distintas características en distintas localizaciones jerárquicas del modelo con el mismo nombre (Figura 7.20a).

Uno de los principales objetivos de los modelos de características es capturar las partes comunes y divergentes de los diferentes productos. Así, un FM refleja cada posible combinación de características que dan como resultado los productos de la SPL. Cada característica ha de ser única, y aunque sí es posible tener características clonadas (modelos basados en cardinalidades), no podemos tener una misma característica en diferentes partes de la jerarquía.

Solución Propuesta

Por defecto, dejamos la primera aparición de esta característica y eliminamos el resto de apariciones. Si la *feature* eliminada contenía hijos, estos también serán borrados para evitar ramas sueltas (Figura 7.20b).

Aunque para la obtención de un modelo representable es necesario tomar esta solución, consideramos que posiblemente el origen del error proviene de la definición de las características mientras se elaboraba el modelo. De manera que, quizá estas características no querían representar la misma cosa dentro del modelo y han sido erróneamente igual nombradas. Por ello, nuestra herramienta informa al usuario por consola de dónde se encuentra el error en el modelo origen y cuales son las medidas que se han tomado para resolverlo. Así, el experto del dominio es consciente del contenido erróneo del modelo y puede tomar la solución que considere más adecuada (aceptar los cambios realizados por la herramienta, modificar el modelo origen,…).

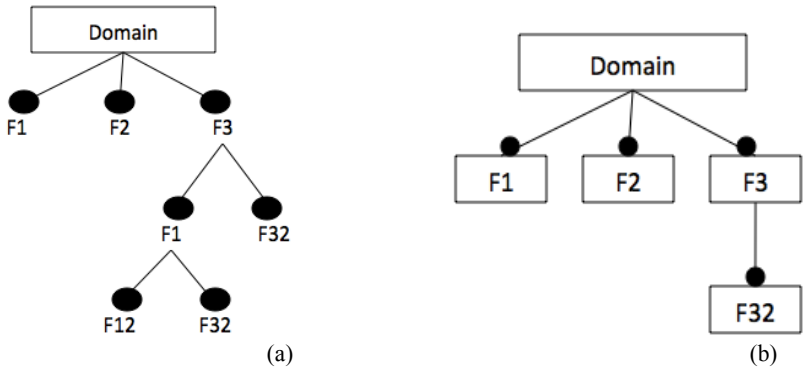


Figura 7.20 Características duplicadas extraídas del FM original (a) y su solución (b)

7.2.1.3 Campos del modelo que carecen de significado

Descripción

Como comentábamos al principio de este capítulo, el FM original está codificado como un archivo de texto plano (CSV) y las características de sus *features* están definidas como campos (a modo tabla).

En el modelo original nos encontramos con el campo llamado ‘Feature Node’. Éste describe si la característica forma parte de un grupo (*Single* o *Multiple)* o no. Puede contener dos valores: ‘Feature Group’ o ‘Feature’. Dentro del modelo, nos encontramos con *features* que se definen como grupos y que en ocasiones forman parte de un grupo y en otras ocasiones son hojas del árbol. Lo mismo pasa cuando se definen como *features*

simples. Además, en ciertos casos este campo está sin definir e incluso a veces, su valor parece dependiente del tipo de variabilidad definido. En conclusión, no existe ningún patrón lógico en el uso de este campo.

Solución Propuesta

En nuestro caso, podríamos haber hecho uso de este campo para la reconstrucción de las relaciones padre-hijo en el modelo pivote. Como el contenido de este campo no es fiable, hemos deducido estas relaciones de otros campos del modelo origen, como la variabilidad y el identificador de la *feature* padre ('Parent Number').

Además, en el caso concreto del modelo origen que se nos ha proporcionado, tomamos como una buena práctica el definir una *feature* como 'Feature Group' si tiene hijos y como 'Feature' si no los tiene.

La Tabla 7.1 representa un fragmento del modelo original. La columna 'Declared Feature Node' contiene los valores originales de este campo en el modelo origen, mientras que la columna 'Expected Feature Node' contiene los valores que se esperan correctos. Los valores en rojo, son ejemplos de los errores encontrados. Como podemos ver, la característica "F2" está definida como 'Feature' pero tiene dos hijos, mientras que "F21" está definida como 'Feature Group' y no tiene hijos. Lo mismo pasa con "F12".

Tabla 7.1 Fragmento del modelo original con campos que carecen de significado

ID	***Feature Node* Declarado**	***Feature Node* Esperado**
Domain	Feature Group	Feature Group
F1	Feature Group	Feature Group
F11	Feature	Feature
F12	**Feature**	Feature Group
F121	**VACÍO**	Feature
F122	**VACÍO**	Feature
F123	**VACÍO**	Feature
F2	**Feature**	Feature Group
F21	**Feature Group**	Feature
F22	Feature	Feature

La Figura 7.21, muestra la representación gráfica correcta de este fragmento del modelo de características en notación basada en cardinalidades.

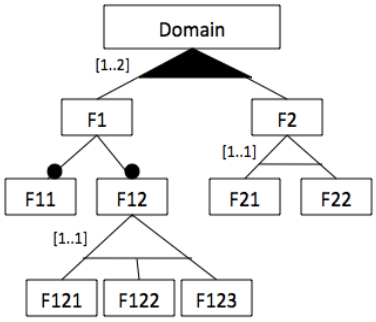


Figura 7.21 representación correcta con notación basada en cardinaldidades

7.2.1.4 Uso ambiguo de la variabilidad

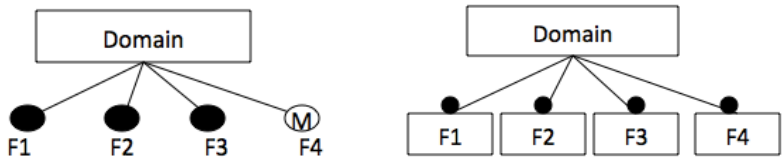
Descripción

Las características definidas con variabilidad *Multiple,* representan que ha de hacerse una selección de "al-menos-una-de-muchas" características a elegir.

En el modelo origen podemos encontrar el uso de esta variabilidad Multiple en grupos de sólo una feature. De este modo, no existe selección posible ya que sólo hay un hijo dónde elegir (Figura 7.22a). En estos casos podemos decir que se hace un empleo incorrecto de la variabilidad.

Solución Propuesta

En el caso en que una característica definida como Multiple es parte de un grupo en el que ella es el único miembro, sustituimos esta relación de grupo por una relación Mandatory (Figura 7.22b). Ya que el significado de la relación Multiple es "al-menos-una-de-muchas" creemos que Mandatory es la relación más precisa para estos casos.



(a) (b)

Figura 7.22 Uso ambiguo de la variabilidad extraídas del FM original (a) y su solución (b)

7.2.2 Análisis semántico

Como hemos comentado a lo largo de todo este documento, cuando trabajamos con modelos de características grandes, necesitamos herramientas automáticas para el análisis de su contenido. En nuestro caso, hemos escogido FaMa de entre las propuestas

disponibles en la literatura por los motivos comentado en el apartado 2.2.4, en el que además, repasábamos las operaciones que esta herramienta proporcionaba.

Con el objetivo de obtener resultados significativos, hemos seleccionado aquellas operaciones que arrojan un rayo de luz sobre la corrección del contenido del modelo de características. No hemos seleccionado aquellas que comprueban las configuraciones ya que nuestra herramienta, como se comentaba en [22][23] ya implementa esa funcionalidad.

A continuación, expondremos los resultados obtenidos al realizar las siguientes operaciones de análisis:

7.2.2.1 Validación (Validation)

El resultado de esta operación es que el modelo no es válido. El modelo de características original no es correcto ya que contiene dos relaciones *cross-tree* del tipo *Excludes* que no están bien definidas. Estas relaciones se establecen entre características del tipo *Mandatory* (Figura 7.23a).

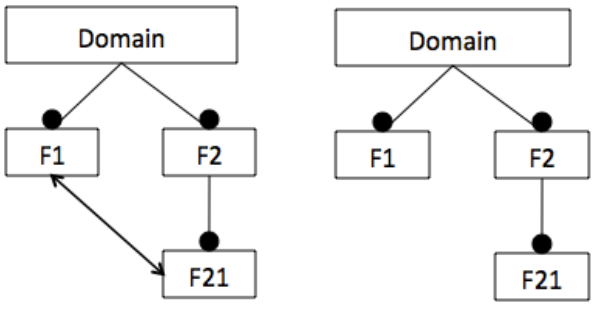
Cuando tenemos características definidas como *mandatory,* siempre habrá una caso en que éstas sean seleccionadas en un producto. Si dos de estas características están relacionadas por una restricción *Excludes* y al mismo tiempo están seleccionadas, tenemos como resultado una contradicción.

Solución Propuesta

Cuando tenemos este tipo de contradicciones hay dos maneras de proceder para atajarlas:

1. Eliminar la relación problemática (Figura 7.23b)
2. Comprobar la definición de las características en el modelo para ver si las características tenían que haber sido definidas como *optional* en vez de cómo *mandatory.*

La solución que nosotros hemos implementado en la herramienta es la primera, ya que es menos intrusiva con la estructura de la SPL. Además, la herramienta avisa dónde aparecen estas relaciones y la solución tomada.



(a) (b)

Figura 7.23 Relación inválida (a) y su solución (b)

7.2.2.2 Productos y Número de Productos (Products & Number of Products)

El resultado de esta operación, es que el modelo contiene 3073 productos potenciales diferentes. La operación *Products* además, nos proporciona las combinaciones de características que dan como resultado estos productos.

7.2.2.3 Detección de errores (Error)

Además de los errores obtenidos tras la validación, esta operación revela que el modelo presenta cinco errores más:

- **Tres características *false-mandatory***

  Son aquellas características que se comportan como *mandatory,* pero que no han sido definidas como tal.

  La *feature* "F21" en la Figura 7.24, es una característica *false-mandatory* ya que está definida como *optional* pero se comporta como una *mandatory* . Esto es debido a la presencia de la restricción *Requires* definida entre la característica *mandatory* "F11" y "F21".

Solución Propuesta

La solución a las características *false-mandatory* es cambiar su definición a *mandatory.*

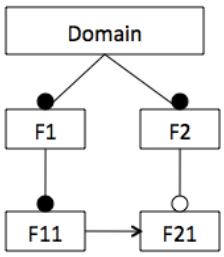


Figura 7.24 Características *false-mandatory*

- **Dos *dead-features***

  Las características muertas o *dead-fetures* [7] (características que no están presentes en ningún producto), suelen aparecer debido a la presencia de relaciones contradictorias en el FM. La Figura 7.25a muestra un ejemplo de este error extraído del modelo origen. Como podemos ver, las características hijas de "Domain" son parte de un grupo *XOR* o *Single*. La característica "F1" requiere a la "F2" y viceversa. En un grupo del tipo *XOR*, tan sólo podemos seleccionar una de sus *features*, así que estas dos relaciones del tipo *Requires* contradicen dicha lógica

Solución Propuesta

En este caso caben dos posibles soluciones, la más adecuada dependerá de lo que el modelo quiera reflejar.

1. Cambiar el grupo *XOR* por un grupo *OR* (Figura 7.25b) de manera que podamos seleccionar más de una *feature*.
2. Eliminar las restricciones *Requires* (Figura 7.25c)

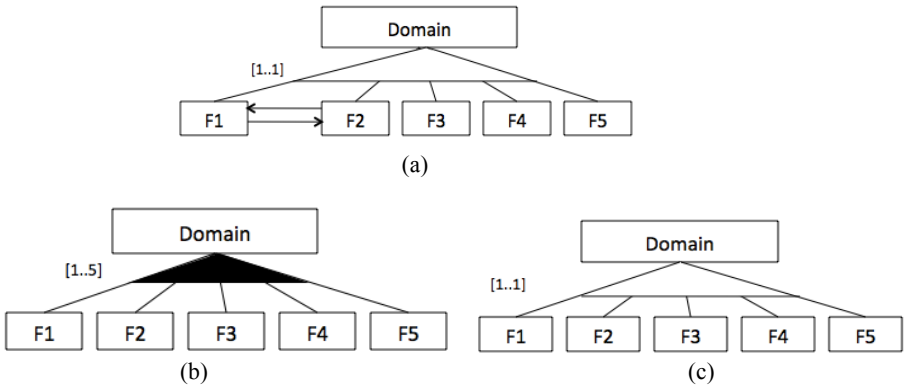


Figura 7.25 *Dead-features* extraídas del FM original (a) y su solución (b, c)

7.3 Conclusiones sobre los resultados del análisis

La Tabla 7.2 muestra los resultados del análisis del modelo industrial. En ella se reflejan los porcentajes de corrección del modelo de características de acuerdo a los siguientes criterios:

1. % Características **totales** bien definidas
2. % Características **opcionales** bien definidas
3. % Características ***mandatory*** bien definidas
4. % Características ***single*** bien definidas
5. % Características ***multiple*** bien definidas
6. % **Grupos** bien definidos
7. % Relaciones ***Implies*** bien definidas
8. % Relaciones ***Excludes*** bien definidas

En la siguiente tabla, la columna 'Source Model' hace referencia a los elementos contenidos en el modelo de características original (sin analizar y corregir). La columna 'Target Model' contienen los elementos del modelo resultado de su corrección tras su análisis. Los porcentajes de corrección se han calculado como sigue:

%Corrección = (Nº elementos correctos / Nº elementos en el modelo

Tabla 7.2 Porcentajes de corrección

	MODELO ORIGEN	**MODELO DESTINO**	**% corrección**
Número de Característias	1195	1016	85.02
Características *Optional*	295	263 (266-3 *false-mandatory*)	89.15
Características *Mandatory*	833	692	83.07
Características *Single*	53	53	100
Características *Multiple*	14	5	35.71
Número de Grupos	258	19	7.36
Implies	70	63 (65-2 rel. que causan *dead features*)	90
Excludes	6	4 (6-2 rel. que hacen al modelo *void*)	66.67
Total de elementos	1529	1099	71.88

Con los resultados obtenidos, podemos ordenar los elementos del modelo según su porcentaje de incorrección (de más a menos erróneos):

Grupos definidos > Características Multiple > Relaciones Excludes > Características Mandatory > Características Optative > Relaciones Implies > Características SIngle

7.4 Eficiencia y Limitaciones de la herramienta de análisis automático

En esta sección, repasaremos las limitaciones que hemos encontrado a la hora de aplicar algunas operaciones proporcionadas por FaMa sobre nuestro modelo. El origen de estas limitaciones viene dado por el gran tamaño de del modelo de características con el que trabajamos (1016 características).

La versión de FaMa que hemos usado para el análisis es FaMa 1.1.1b, publicada en 14 de abril de 2011.

A continuación listaremos las operaciones que llevamos a cabo, mencionando la eficiencia en su ejecución y los problemas encontrados.

7.4.1 Validación

Esta operación nos proporcionaba una respuesta rápida (3 seg. aprox.).

Limitaciones

Si el modelo es *void,* la herramienta te da la posibilidad de proporcionarte una explicación sobre los errores encontrados. Si le pedimos que nos proporcione esta información, la herramienta no proporciona respuesta en tiempo (pasa días sin responder).

7.4.2 Productos y Número de productos

La herramienta no encuentra ninguna dificultad en proporcionar respuesta a estas preguntas en tiempo (30 min. aprox.).

7.4.3 Variabilidad

Limitaciones

No hemos sido capaces de obtener respuesta de esta operación ya que la herramienta pasaba mucho tiempo sin proporcionar respuesta (días).

7.4.4 Detección de errores

Esta operación nos proporciona una respuesta inmediata, listando todos los tipos de errores que encuentra a lo largo del modelo.

Limitaciones

Al igual que en la operación de validación*,* la herramienta te da la posibilidad de proporcionarte una explicación sobre los errores encontrados. Si le pedimos que nos proporcione esta información, la herramienta no proporciona respuesta en tiempo (pasa días sin responder).

7.4.5 Características *Core (*comunes) y *Variant* (variantes)

Estas operaciones no están disponibles en la última versión pública de FaMa, mostrándonos un mensaje informando de que el modelo actual de FaMa no da soporte a dichas operaciones.

7.5 Conclusiones

En este capítulo, se detalla el proceso de análisis del modelo de características industrial proporcionado por Rolls Royce utilizando nuestra herramienta.

En primer lugar, se ha mostrado, paso a paso cómo ejecutar la herramienta y los resultados que se han obtenido. Además, se han definido y clasificado los errores encontrados durante las etapas del análisis (sintáctico y semántico), proponiendo soluciones a los mismos.

Los resultados obtenidos tras el análisis y la utilización de la herramienta, proporcionan a la empresa británica un mecanismo para automatizar el análisis de su modelo de gran escala. Siendo conscientes de las inconsistencias presentes en su modelo de características y con las conclusiones extraídas de nuestro análisis, podrán tomar acciones para corregir el modelo. Además, con todo este *background,* podrán incluso elaborar una serie de buenas prácticas a tener en cuenta a la hora de modificar la línea de productos, disminuyendo así, de manera incremental, la aparición de errores en su modelo de características.

Por otra parte, la herramienta FaMa nos ha sido de tremenda utilidad para extraer información relevante sobre el modelo que hubiéramos sido incapaces de extraer manualmente. Sin embargo, según nuestra experiencia, el uso de FaMa con modelos de gran tamaño revela algunas limitaciones a la hora de proporcionar respuestas en tiempo. Cuando trabajamos con modelos de características grandes, estamos tratando con estructuras muy complejas para analizar por los razonadores o *solvers* que se encargan de extraer la información. Creemos que estas limitaciones podrían resolverse ejecutando estos algoritmos en máquinas muy potentes. Sin embargo, un usuario corriente con una máquina estándar, no pude sacar todo el partido que FaMa nos ofrece (problemas de escalabilidad).

Parte IV

Conclusiones y trabajos futuros

8. Conclusiones y trabajos futuros

En la actualidad, el análisis de los modelos de características (que especifican las líneas de productos software), es uno de los campos en el que se nos presentan más desafíos. Ya contamos con métodos e incluso con herramientas que nos permiten analizar nuestro modelo mediante su representación lógica u otro tipo de lenguajes descriptivos. Pero el uso de estas herramientas de análisis, queda restringido a aquellas que se ajusten a la notación en la que nuestro modelo de características esté representado. Además, las herramientas de análisis existentes, no son especialmente interoperables ni fáciles de usar por un usuario corriente.

Por otra parte, no se automatiza un análisis de la forma (sintáctico) de los modelos. Este análisis sintáctico, es un aspecto importante cuando trabajamos con modelos grandes y con herramientas imprecisas que no controlan la introducción de errores en la creación y modificación de una línea de productos.

En este trabajo, se ha elaborado una herramienta prototipo que nos permite analizar un modelo de características real de gran tamaño en profundidad. El método empleado para hacerlo, nos permite coger el modelo original y analizarlo en forma (sintácticamente) y en contenido (semánticamente). Para hacer de este framework una herramienta amigable, usable e interoperable entre los distintos módulos que incorpora, se han implementado los siguientes componentes:

- Un *parser* o analizador sintáctico del modelo origen que además migra el modelo a nuestra representación (más completa).

- Una serie de transformaciones definidas en QVT-Relations para transformar nuestro modelo en un modelo entendible por la herramienta de análisis automático Fama.

- Incorporación de los mecanismos de análisis proporcionados por FaMa para analizar el modelo y mostrar los resultados.

Nuestra framework extenderá la funcionalidad de la herramienta MULTIPLE, desarrollada en nuestro grupo de investigación. Esta tecnología extensible para la explotación de líneas de producto, nos ofrece:

- Un editor gráfico de modelos de características extendidos (cardinalidades y atributos).

- La posibilidad de definir restricciones sobre este tipo de modelos.

- Validación de configuraciones de características.

- Implementa un proyector manual escrito en Java, que crea, a partir de los modelos definidos en MULTIPLE, un modelo analizable por la herramienta de análisis automático FaMa.

Sin embargo, no nos permite analizar el contenido de los modelos de características.

Por otra parte, el componente proyector, encargado de crear el modelo legible por FaMa, presenta los siguientes inconvenientes:

- No proporciona trazabilidad, ya que se crea el archivo legible por FaMa mediante código. No tenemos información sobre qué elemento del modelo origen se corresponde con qué elemento del modelo destino. Así que comprobar manualmente que nuestro modelo en FaMa está bien construido manualmente, sería una tarea igual de tediosa que elaborarlo a mano desde cero para FaMa.

- No ofrece interoperabilidad con FaMa. Su uso no se integra en la herramienta, de manera que para poder analizar el modelo, hemos de bajarnos FaMa y utilizarla en su versión de interfaz de línea de comandos. Lo que requiere además aprender el uso de otra herramienta.

Los componentes que añade nuestro desarrollo, nos permitirán llevar a cabo el análisis automatizado del un modelo de características. Estos, al haber sido desarrollados con Eclipse, se integrarán con la funcionalidad ofrecida por MULTIPLE, siguiendo un enfoque dirigido por modelos. Además, se modifican algunos de los componentes ya existentes en MULTIPLE para añadirles funcionalidad. Este es el caso del metamodelo de características definido por la herramienta. En nuestro desarrollo, añadimos las clases necesarias al metamodelo para que los elementos de sus instancias (modelos de características) puedan almacenar información extra en forma de anotaciones. También se hacen los cambios necesarios para que el editor gráfico de líneas de producto existente, permita representar y editar instancias de este nuevo metamodelo.

Por otra parte, se sustituye el proyector manual a FaMa por la definición de las transformaciones de nuestro dominio al dominio de FaMa usando el lenguaje QVT-Relations. Esto nos aportará como ventajas:

- QVT-Relations es un lenguaje declarativo que define reglas de transformación mediante patrones, lo que lo hace un lenguaje entendible para el usuario.

- Proporciona trazabilidad, de manera que podemos comprobar las correspondencias entre los elementos del modelo origen y destino.

Además, aplicamos la herramienta a un caso de estudio real. El caso de estudio consiste en el análisis del modelo de características de gran tamaño proporcionado por la compañía Rolls Royce. Con los resultados obtenidos, se han definido y clasificado los errores encontrados durante las etapas del análisis (sintáctico y semántico), proponiendo soluciones a los mismos.

Los resultados obtenidos tras el análisis y la utilización de la herramienta, proporcionan a la compañía británica, un mecanismo para automatizar el análisis de su modelo de gran escala. Siendo conscientes de las inconsistencias presentes en su modelo de características y con las conclusiones extraídas de nuestro análisis, podrán tomar acciones para corregir el modelo. Además, con todo este *background,* podrán

incluso elaborar una serie de buenas prácticas a tener en cuenta a la hora de modificar la línea de productos, disminuyendo así, de manera incremental, la aparición de errores en sus modelo de características.

Por otra parte, la herramienta FaMa nos ha sido de tremenda utilidad para extraer información relevante sobre el modelo que hubiéramos sido incapaces de extraer manualmente. Sin embargo, según nuestra experiencia, el uso de FaMa con modelos de gran tamaño revela algunas limitaciones a la hora de proporcionar respuestas en tiempo. Cuando trabajamos con modelos de características grandes, estamos tratando con estructuras muy complejas para analizar por los razonadores o *solvers* que se encargan de extraer la información. Creemos que estas limitaciones podrían resolverse salvando los problemas de escalabilidad y ejecutando las operaciones de análisis en máquinas muy potentes.

Pese a todas las ventajas que ofrece nuestro desarrollo, cabe destacar ciertos aspectos de la herramienta que podrían motivar trabajos futuros. En primer lugar, este desarrollo se ha realizado para un modelo con una notación concreta. Los *mappings* que el componente *parser* define, son totalmente dependientes de la estructura de este modelo original. Lo bueno de nuestro desarrollo es que los distintos componentes son independientes, luego podría aprovecharse el resto de partes del framework adaptando dicha parte a las necesidades concretas.

Por otra parte, el análisis que se realiza usando FaMa, es una versión restringida de toda la funcionalidad que puede ofrecernos la herramienta. Para el presente caso, hemos implementado aquellas operaciones que funcionaban bien con el modelo concreto con el que trabajábamos. Creemos que salvando los problemas de escalabilidad, se podría sacar mas provecho de la funcionalidad ofrecida por FaMa.

9. Agradecimientos

Agradecemos a Andy Nolan, Neil Robson y Francis Thom de Rolls Royce su colaboración con esta propuesta.

El desarrollo de esta tesis ha sido posible gracias al proyecto de investigación MULTIPLE, financiado con fondos públicos (MULTIPLE: Multipmodeling Approach For Quality-Aware Software Product Lines, Ministerio de Ciencia e Innovación, ref. TIN2009-13838).

Referencias

[1] Batory, D. 2005. Feature models, grammars, and propositional formulas. In *Software Product Lines, volume 3714 of Lecture Notes in Computer Sciences*, Springer Berlin / Heidelberg, 2005, 7–20.

[2] Batory, D., Benavides, D. and Ruiz-Cortés, A.. *Automated analysis of feature models: Challenges ahead*. Communications of the ACM, 49(12): 45–47, December 2006.

[3] Batory, D., Sarvela, J. and Rauschmayer, A. Scaling step-wise refinement. *IEEE Trans. Software Eng., 30* (6). 355–371.

[4] Benavides, D. 2007. *On the Automated Analysis of Software Product Lines using Feature Models: a Frameqork for Developing Automated Tool Support*. Doctoral Thesis. University of Seville, Spain, 2007.

[5] Benavides, D., Ruiz-Cortés, A., Smith, B., O'Sullivan, B. and Trinidad, P. *Computational issues on the automated analysis of feature models using constraint programming*. 2007.

[6] Benavides, D., Ruiz-Cortés, A., Trinidad, P. and Segura, S. 2006. A survey on the automated analyses of feature models. In *Jornadas de Ingeniería del Software y Bases de Datos,* (Sitges, Spain, 2006), 367–376.

[7] Benavides, D., Ruiz-Cortés, A., Trinidad, P. 2005. Automated Reasoning on Feature Models. In *Conference on Advanced Information Systems Engineering (CAISE)*, (Porto, Portugal, 2005).

[8] Benavides, D., Ruiz-Cortés, A., Trinidad, P. Using constraint programming to reason on feature models. In *The Seventeenth International Conference on Software Engineering and Knowledge Engineering*, SEKE, pages 677–682, 2005.

[9] Benavides, D., Ruiz-Cortés, A., Trinidad, P. *Generative and Transformational Techniques in Software Engineering*, volume 4143 of Lecture Notes in Computer Science, chapter Using Java CSP Solvers in the Automated Analyses of Feature Models, pages 389–398. Springer–Verlag, 2006.

[10] Benavides, D., Segura, S., Ruiz-Cortés, A. Automated Anlysis of feature Models 20 Years Later: A Literature Review.

[11] Bryant, R. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, *35* (8), 677–691.

[12] Cao, F., Bryant, B., Burt, C. Huang, Z., Raje, R., Olson, A. and Auguston, M. Automating feature-oriented domain analysis. In *International Conference on Software Engineering Research and Practice (SERP'03)*, pages 944–949, June 2003.

[13] Chastek, G., Donohoe, Kang, K.C. and Thiel, S. *Product Line Analysis: A Practical Introduction*. Technical Report CMU/SEI-2001-TR-001, Software Engineering Institute, Carnegie Mellon University, June 2001.

[14] Clements, P., Northrop, L. and Northrop, L.M. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, August, 2001.

[15] Czarnecki, K., Bednasch, T., Unger, P. and Eisenecker, U. *Generative programming for embedded software: An industrial experience report*. In Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, pages 156–172, 2002.

[16] Czarnecki, K. and Eisenecker, U.W. 2000. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[17] Czarnecki, K. and Eisenecker, U.W. Staged configuration using feature models. In *Proceedings of the Third Software Product Line Conference* 2004, volume 3154 of Lecture Notes in Computer Sciences, pages 266–282. Springer–Verlag, 2004.

[18] Czarnecki, K., Helsen, S. and Eisenecker, U.W. 2005. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice, 10* (1), 7–29.

[19] Czarnecki, K. and Kim, P. Cardinality-based feature modeling and constraints: A progress report. In *Proceedings of the International Workshop on Software Factories* At OOPSLA 2005, 2005.

[20] Deursen, A. and Klint, P. *Domain–specific language design requires feature descriptions*. Journal of Computing and Information Technology, 10 (1): 1–17, 2002.

[21] Eriksson, M., Börstler, J., Borg, K. 2005.The PLUSS Approach – Domain Modeling with Features, Use Cases and Use Case Realizations. In *Software Product Lines, volume 3714 of Lecture Notes in Computer Sciences*, Springer Berlin / Heidelberg, 2005, 33–44.

[22] Gómez, A., Ramos, I. 2010. Cardinality-Based Feature Modeling and Model-Driven Engineering: Fitting them Together. In *Proceedings of the Fourth International Workshop on Variability Modeling of Software-Intensive Systems*, (Linz, Austria, 2010), 61-68.

[23] Gómez, A., Ramos, I. 2010. Automatic Tool Support for Cardinality-Based Feature Modeling with Model Constraints for Information Systems Development. In *19th International Conference on Information Systems Development*, (Prague, Czech Republic, 2010).

[24] Griss, M., Favaro, J., d'Alessandro, M. 1998. Integrating Feature Modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse*, (Vancouver, Canada, 1998), 76-85.

[25] Gu, J., Purdom, P., Franco, J. and Wah, B. 1997. *Satisfiability Problem: Theory and Applications*, chapter Algorithms for the Satisfiability (SAT) Problem: A Survey. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997, 19–152.

[26] Iivari, J. *Why are CASE tools not used?* In Communications of the ACM, Volume 39 Issue 10, pages 94-103, 1996.

[27] Kang, K., Cohen, S., Hess, J., Nowak, W. and Peterson, S. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Software Engineering Institute at Carnegie Mellon University, Pittsburgh, Pennsylvania, 1990.

[28] Kleppe, A., Warmer, J. and Bast, W. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[29] Mannion, M. Using first-order logic for product line model validation. In *Proceedings of the Second Software Product Line Conference (SPLC2)*, volume 2379 of Lecture Notes in Computer Sciences, pages 176–187, San Diego, CA, 2002. Springer–Verlag.

[30] Riebisch, M., Böllert, K., Streitferdt, D. and Philippow, I. Extending feature diagrams with UML multiplicities. In *6th World Conference on Integrated Design & Process Technology (IDPT2002)*, June 2002.

[31] Riebisch, M., Streitferft, D. and Pashov, I. Modeling variability for object oriented product lines. In *ECOOP 2003 Workshop Reader,* volume 3013 of Lecture Notes in Computer Sciences. Springer–Verlag, 2004.

[32] Schobbens, P., Heymans, P. and Trigaux. J. Feature diagrams: A survey and a formal semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE)*. IEEE Computer Society, 2006.

[33] Schobbens, P., Heymans, P., Trigaux. J and Bontemps, Y. *Generic semantics of feature diagrams*. Computer Networks, 51(2):456–479, Feb 2007.

[34] Sun, J., Zhang, H., Li, Y.F. and Wang, H. Formal semantics and verification for feature modeling. In *Proceedings of the ICECSS05*, 2005.

[35] Trinidad, P., Benavides, D., Durán, A., Ruiz-Cortés, A. and Toro, M. *Agile debugging of feature models*. Journal of System and Software (conditionally accepted), 2007.

[36] Trinidad, P., Benavides, D., Segura, S. and Ruiz-Cortés, A. A first step detecting inconsistencies in feature models. In CAiSE Short Paper Proceedings, 2006.

[37] Trinidad, P., Benavides, D., Segura, S. and Ruiz-Cortés, A. 2007. FAMA: hacia el análisis automático de modelos de características. In *XII Jornadas de Ingeniería del Software y Bases de Datos*, (Zaragoza, Spain, 2007). FAMA Tool, available at: http://www.isa.us.es/fama/

[38] Trujillo, S. 2007. *Feature Oriented Model Driven Product Lines*. Doctoral Thesis. University of the Basque Country, San Sebastián, Spain, 2007.

[39] Tsang, E. 1995. *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego, 1995.

[40] van Deursen, A. and Klint, P. *Domain–specific language design requires feature descriptions*. Journal of Computing and Information Technology, 10 (1):1–17, 2002.

[41] von der Massen, T. and Litcher, H. Determining the variation degree of feature models. In *Software Product Lines Conference*, volume 3714 of Lecture Notes in Computer Sciences, pages 82–88, 2005.

[42] Wang, H., Li, Y., Sun, J., Zhang, H. and Pan, J. A semantic web approach to feature modeling and verification. In *Workshop on Semantic Web Enabled Software Engineering (SWESE'05)*, November 2005.

[43] Wookcock, J. and Davies, J. *Using Z: Specification, Refinement, and Proof*. Prentice–Hal, 1996.

[44] Zhang, W., Zhao, H. and Mei, H. *A propositional logic-based method for verification of feature models*. In J. Davies, editor, ICFEM 2004, volume 3308, pages 115–130. Springer–Verlag, 2004.

[45] Zhao, W., Bryant, B., Cao, F., Raje, R., Auguston, M., Burt, C. and Olson, A. Grammatically interpreting feature compositions. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004*), 2004.

[46] Apache CSV-Commons Library. Avaliable at: http://commons.apache.org/sandbox/csv/

[47] FAMA Tool, available at: http://www.isa.us.es/fama/.

[48] ikv++ technologies ag. ikv++ technologies ag website. http://www.ikv.de

[49] MDA Guide Version 1.0.1: 2003, from Object Management Group: http://www.omg.org/docs/omg/03- 06- 01.pdf.

[50] Meta Object Facility (MOF) 2.0 from Object Management. Core Specification (ptc/06-01-01), 2006: http://www.omg.org/cgi-bin/doc?formal/2006-01-01

[51] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification from Object Management. Core Specification (formal/11-01-01), 2011: http://www.omg.org/spec/QVT/1.1/PDF

[52] Object Management Group. http://www.omg.org.

[53] Object Technology International, Inc. Eclipse platform technical overview, 2003. http: //www.eclipse.org/whitepapers/eclipse-overview.pdf.The Eclipse Modeling Framework (EMF) Project. http://www.eclipse.org/downloads/

[54] The Eclipse Modeling Framework (EMF) Overview. http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html

[55] The Open Source Library for OCL Project. Oslo website. http://oslo-project.berlios.de/

Apéndice A: Proyector de nuestro modelo pivote a FaMa de MULTIPLE

```
public void run(IAction action) {

        ResourceSet resourceSet = new ResourceSetImpl();
        Resource resource = resourceSet.createResource(URI.createFileURI(file
                .getLocation().toString()));
        final IFile result = ResourcesPlugin
                .getWorkspace()
                .getRoot()
                .getFile(
                        file.getFullPath().removeFileExtension()
                                .addFileExtension("fm"));

        try {
             resource.load(Collections.EMPTY_MAP);
             final FeatureModel featureModel = (FeatureModel) resource
                       .getContents().get(0);
             Job convertJob = new Job("Convert to FAMA") {

                  @Override
                  protected IStatus run(IProgressMonitor monitor) {
                       monitor.beginTask("Convert to FAMA...",
                                 IProgressMonitor.UNKNOWN);
                       StringBuffer buffer = new StringBuffer();

                       buffer.append("%Relationships\n");
                       for (Feature f : featureModel.getFeatures()) {
                            if ((f.getChilds() != null && !f.getChilds().isEmpty())
                                      || f.getGroup() != null) {
                                 buffer.append(f.getName());
                                 buffer.append(": ");
                                 // Features hijas
                                 for (StructuralRelationship sr : f.getChilds()) {
                                      Feature cf = sr.getTo();

                                      if (sr.getLowerBound() == 0) {
                                           buffer.append("[" + cf.getName() + "] ");
                                      } else {
                                           buffer.append(cf.getName() + " ");
                                      }
                                 }
```

```
                    // Grupos
                    Group g = f.getGroup();
                    if (g != null) {
                        if (g.getUpperBound() == -1)
                            buffer.append("[" + g.getLowerBound() + ","
                                    + g.getChilds().size() + "]{");
                        else
                            buffer.append("[" + g.getLowerBound() + ","
                                    + g.getUpperBound() + "]{");
                        for (StructuralRelationship sr : g.getChilds()) {
                            Feature cf = sr.getTo();

                            buffer.append(cf.getName() + " ");
                        }
                        buffer.append("}");
                    }
                    buffer.append(";\n");
                }
            }

            boolean existConstraints = false;
            for (Relationship r : featureModel.getRelationships()) {
                if (r instanceof Implies || r instanceof Excludes) {
                    existConstraints = true;
                    break;
                }
            }
            if (existConstraints)
                buffer.append("\n%Constraints\n");

            for (Relationship r : featureModel.getRelationships()) {
                if (r instanceof Implies) {
                    Implies rr = (Implies) r;
                    buffer.append((rr.getFrom().getName())
                            + " REQUIRES " + (rr.getTo().getName())
                            + ";\n");
                } else if (r instanceof Biconditional) {
                    Biconditional rr = (Biconditional) r;
                    buffer.append(rr.getFrom().getName() + " REQUIRES "
                            + rr.getTo().getName() + ";\n");
                    buffer.append(rr.getTo().getName() + " REQUIRES "
                            + rr.getFrom().getName() + ";\n");
                } else if (r instanceof Excludes) {
                    Excludes rr = (Excludes) r;
                    buffer.append(rr.getFrom().getName() + " EXCLUDES "
                            + rr.getTo().getName() + ";\n");
                }

            }
            InputStream stream = new BufferedInputStream(
                    new ByteArrayInputStream(buffer.toString()
                            .getBytes()));

            try {
                if (result.exists()) {
                    result.setContents(stream, IResource.NONE, monitor);
                } else {
                    result.create(stream, IResource.NONE, monitor);
                }
            } catch (CoreException e) {
                e.printStackTrace();
            }
            return Status.OK_STATUS;
        }
    };
```

```
                    // Grupos
                    Group g = f.getGroup();
                    if (g != null) {
                        if (g.getUpperBound() == -1)
                            buffer.append("[" + g.getLowerBound() + ","
                                    + g.getChilds().size() + "]{");
                        else
                            buffer.append("[" + g.getLowerBound() + ","
                                    + g.getUpperBound() + "]{");
                        for (StructuralRelationship sr : g.getChilds()) {
                            Feature cf = sr.getTo();

                            buffer.append(cf.getName() + " ");
                        }
                        buffer.append("}");
                    }
                    buffer.append(";\n");
                }
            }

            boolean existConstraints = false;
            for (Relationship r : featureModel.getRelationships()) {
                if (r instanceof Implies || r instanceof Excludes) {
                    existConstraints = true;
                    break;
                }
            }
            if (existConstraints)
                buffer.append("\n%Constraints\n");

            for (Relationship r : featureModel.getRelationships()) {
                if (r instanceof Implies) {
                    Implies rr = (Implies) r;
                    buffer.append((rr.getFrom().getName())
                            + " REQUIRES " + (rr.getTo().getName())
                            + ";\n");
                } else if (r instanceof Biconditional) {
                    Biconditional rr = (Biconditional) r;
                    buffer.append(rr.getFrom().getName() + " REQUIRES "
                            + rr.getTo().getName() + ";\n");
                    buffer.append(rr.getTo().getName() + " REQUIRES "
                            + rr.getFrom().getName() + ";\n");
                } else if (r instanceof Excludes) {
                    Excludes rr = (Excludes) r;
                    buffer.append(rr.getFrom().getName() + " EXCLUDES "
                            + rr.getTo().getName() + ";\n");
                }

            }
            InputStream stream = new BufferedInputStream(
                    new ByteArrayInputStream(buffer.toString()
                            .getBytes()));

            try {
                if (result.exists()) {
                    result.setContents(stream, IResource.NONE, monitor);
                } else {
                    result.create(stream, IResource.NONE, monitor);
                }
            } catch (CoreException e) {
                e.printStackTrace();
            }
            return Status.OK_STATUS;
        }
    };
```

```
            convertJob.schedule();
        } catch (IOException e) {
            e.printStackTrace();
        }
```

Apéndice B. Parser de transformacion de CSV a nuestro modelo pivote basado en cardinalidades

```
package es.upv.dsic.issi.multiple.features.parser;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.StringTokenizer;

import org.apache.commons.csv.CSVParser;
import features.Annotation;
import features.AnnotationSet;
import features.Excludes;
import features.Feature;
import features.FeatureModel;
import features.FeaturesFactory;
import features.Group;
import features.Implies;
import features.StructuralRelationship;
import org.eclipse.emf.common.util.EList;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Display;
import org.eclipse.ui.console.ConsolePlugin;
import org.eclipse.ui.console.IConsole;
import org.eclipse.ui.console.MessageConsole;
import org.eclipse.ui.console.MessageConsoleStream;

public class Parser {

	private HashMap<String, Feature> map;

	MessageConsole console = new MessageConsole("parser console", null);
	MessageConsoleStream redStream = console.newMessageStream();
	MessageConsoleStream blackStream = console.newMessageStream();
	MessageConsoleStream greenStream = console.newMessageStream();

	public Parser() {

		this.map = new HashMap<String, Feature>();
	}

	/**
	 * @param stream
	 * @return FeatureModel
	 * @throws IOException
	 * @throws Exception
	 */
	public FeatureModel createFeatureModel(InputStream stream)
			throws IOException {

		ConsolePlugin.getDefault().getConsoleManager()
				.addConsoles(new IConsole[] { console });
		redStream.setColor(Display.getDefault().getSystemColor(SWT.COLOR_RED));
		blackStream.setColor(Display.getDefault().getSystemColor(
				SWT.COLOR_BLACK));
		greenStream.setColor(Display.getDefault().getSystemColor(
				SWT.COLOR_GREEN));
```

```
Feature feature = null;
HashMap<Integer, String[]> auxmap = new HashMap<Integer, String[]>();
auxmap.clear();
this.map.clear();

FeatureModel model = FeaturesFactory.eINSTANCE.createFeatureModel();
model.setName("Features Model");
// Set rootFeature
Feature rootFeature = FeaturesFactory.eINSTANCE.createFeature();
rootFeature.setName("dummyRootFeature");
rootFeature.setOwner(model);
model.setRootFeature(rootFeature);

// Processing stream 1st time
InputStreamReader inputReader = null;

try {
      inputReader = new InputStreamReader(stream);
      BufferedReader br = new BufferedReader(inputReader);
      String[] linea = null;
      CSVParser parser = new CSVParser(br);
      // skip first line that contains column values
      linea = parser.getLine();
      boolean condition = false;
      int numberLines = 0;
      int repeatedChilds = 0;
      ArrayList<String> names = new ArrayList<String>();
      greenStream
                  .println("WELCOME TO THE MULTILE SINTACTIC ANALYSIS CONSOLE");

      while ((linea = parser.getLine()) != null) {
            // blackStream.println("Reading line: " +
            // parser.getLineNumber());

            feature = FeaturesFactory.eINSTANCE.createFeature();
            String absoluteNum = linea[ParserUtility.IND_ABSOLUTE_NUMBER];
            String moduleNum = linea[ParserUtility.IND_MODULE_NUMBER];
            String featureID = moduleNum + absoluteNum;
            // check if contains heading
            if (condition
                        && linea[ParserUtility.IND_OBJECT_NUMBER].length() > repeatedChilds)
                  redStream
                              .println("Line: "
                                          + parser.getLineNumber()
                                          + " ,feature "
                                          + linea[ParserUtility.IND_HEADING]
                                          + "won't be processed too because its parent
                                          was invalid");
            else {
                  condition = false;
                  if (!linea[ParserUtility.IND_HEADING].equals("")) {
                        // check if feature already exists
                        if (names.contains(linea[ParserUtility.IND_HEADING])) {
                              redStream
                                          .println("Line: "
                                                      + parser.getLineNumber()
                                                      + " ,Feature: "
                                                      + linea[ParserUtility.IND_HEADING]
                                                      + " already in the model. Won't be
                                                       processed");
                              redStream
                                          .println("Children Won't be processed too");
                              condition = true;
                              repeatedChilds = linea[ParserUtility.IND_OBJECT_NUMBER]
                                          .length();
                        }
```

```
                        else {
                                names.add(linea[ParserUtility.IND_HEADING]);
                                feature.setName(fixString(linea[ParserUtility.IND_HEADING]));
                                // save the feature with name in map
                                this.map.put(featureID, feature);
                                // save the rest of the feature info in auxmap to
                                // retrieve
                                // it later
                                auxmap.put(numberLines, linea);
                                numberLines++;
                           }
                        } else {
                           redStream
                                .println("Line: "
                                        + parser.getLineNumber()
                                        + " this feature has an empty heading name.
                                        Feature won't be processed");
                        }
                     }
                }

                // saving features in the model
                saveFeatures(model, map, auxmap, rootFeature);

                // correction of the groups(multiple or single) that contains a
                // single child
                fixGroups(model);

           } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
           } finally {
                try {
                     if (null != inputReader) {
                          inputReader.close();
                     }

                } catch (Exception e2) {
                     e2.printStackTrace();
                     redStream.println("Could not read the file");
                }
           }
           return model;

      }

      /**
       * @param model
       * @param map
       * @param auxmap
       * @param rootFeature
       * @throws IOException
       */
      private void saveFeatures(final FeatureModel model,
                final HashMap<String, Feature> map,
                HashMap<Integer, String[]> auxmap, final Feature rootFeature)
                throws IOException {
           // processing auxiliar HAshMap
           int sizeMap = 0;
           String[] secondLine = null;
           while (sizeMap <= auxmap.size() - 1) {
                secondLine = auxmap.get(sizeMap);
                // blackStream.println("Feature " + sizeMap);
                Feature auxiliarFeature = null;
                Feature parentFeature = null;

                auxiliarFeature = this.map
                          .get(secondLine[ParserUtility.IND_MODULE_NUMBER]
                                    + secondLine[ParserUtility.IND_ABSOLUTE_NUMBER]);
```

```
            parentFeature = this.map
                        .get(secondLine[ParserUtility.IND_MODULE_NUMBER]
                                    + secondLine[ParserUtility.IND_PARENT_NUMBER]);
            for (int index = 0; index < secondLine.length - 1; index++) {
                  // blackStream.println(secondLine[index]);
                  secondProcess(model, auxiliarFeature, secondLine[index], index,
                              parentFeature, rootFeature, sizeMap);
            }
            sizeMap++;

            model.getFeatures().add(auxiliarFeature);
            auxiliarFeature.setOwner(model);
            // blackStream.println("Exit adding feature " +
            // auxiliarFeature.getName());
            if (parentFeature != null) {
                  model.getFeatures().add(parentFeature);
                  // blackStream.println("PARENT feature added" +
                  // parentFeature.getName());
                  // blackStream.println();
            }

      }
}

/**
 * @param model
 */
private void fixGroups(final FeatureModel model) {

      EList<Feature> featuresList = null;
      Feature feature = null;
      try {
            featuresList = model.getFeatures();
            for (int i = 0; i <= featuresList.size() - 1; i++) {
                  feature = featuresList.get(i);
                  if (feature.getGroup() != null) {
                        Group group = feature.getGroup();
                        if (group.getChilds().size() == 1) {
                              group.getChilds().get(0).setLowerBound(1);
                              group.getChilds().get(0).setUpperBound(1);
                              group.getChilds().get(0).setFrom(feature);

                              // remove the group
                              feature.setGroup(null);
                        }
                  }

            }
      } catch (Exception e) {
            e.printStackTrace();
      }
}

/**
 * @param model
 * @param feature
 * @param valor
 * @param indice
 * @param parent
 * @param root
 * @throws IOException
 */
private void secondProcess(final FeatureModel model, final Feature feature,
            final String value, final int indice, final Feature parent,
            final Feature root, final int line) throws IOException {
      String type = "";
      AnnotationSet set = null;

      switch (indice) {
      case ParserUtility.IND_MODULE_NAME:
            set = createAnnot(feature, ParserUtility.STR_MODULE_NAME, value);
```

```
                model.getModelAnnotations().add(set);
                break;
          case ParserUtility.IND_MODULE_NUMBER:
                set = createAnnot(feature, ParserUtility.STR_MODULE_NUMBER, value);
                model.getModelAnnotations().add(set);
                break;
          case ParserUtility.IND_ABSOLUTE_NUMBER:
                set = createAnnot(feature, ParserUtility.STR_ABSOLUTE_NUMBER, value);
                model.getModelAnnotations().add(set);
                break;
          case ParserUtility.IND_OBJECT_NUMBER:
                set = createAnnot(feature, ParserUtility.STR_OBJECT_NUMBER, value);
                model.getModelAnnotations().add(set);
                break;
          case ParserUtility.IND_PARENT_NUMBER:
                if (value.equals("")) {
                      setFeaturesToRoot(model, feature, root);
                      // blackStream.println("It's a root feature");
                }
                break;
          case ParserUtility.IND_VARIATION:
                if (value.equals(""))
                      redStream.println("Not variability specified at line: " + line);
                else
                      processVariability(model, feature, value, parent);
                break;
          case ParserUtility.IND_REQUIRES:
                type = "requires";
                processRestrictions(model, feature, value, type, line);
                break;
          case ParserUtility.IND_PROHIBITS:
                type = "prohibits";
                processRestrictions(model, feature, value, type, line);
                break;
          }
    }

    /**
     * @param model
     * @param feature
     * @param root
     */
    private void setFeaturesToRoot(final FeatureModel model,
                final Feature feature, final Feature root) {

          StructuralRelationship relationship = FeaturesFactory.eINSTANCE
                      .createStructuralRelationship();

          relationship.setLowerBound(1);
          relationship.setUpperBound(1);
          relationship.setFrom(root);
          relationship.setTo(feature);
          root.getChilds().add(relationship);
          feature.setParent(relationship);
          feature.setOwner(model);
          model.getRelationships().add(relationship);

    }

    /**
     * @param feature
     * @param tipo
     * @param strAnnot
     * @return
     */
    private AnnotationSet createAnnot(final Feature feature, final String type,
                final String strAnnot) {
          AnnotationSet annotSet = null;
          Annotation annotation = FeaturesFactory.eINSTANCE.createAnnotation();
          annotation.setKey(type);
          annotation.setValue(strAnnot);
```

```
            if (feature.getAnnotationSet() != null
                  && feature.getAnnotationSet() instanceof AnnotationSet) {
                annotSet = (AnnotationSet) feature.getAnnotationSet();
            } else {
                  annotSet = FeaturesFactory.eINSTANCE.createAnnotationSet();
                  feature.setAnnotationSet(annotSet);
                  annotSet.setAnnotContext(feature);
            }
            annotSet.getAnnotations().add(annotation);
            return annotSet;
      }

      /**
       * @param model
       * @param feature
       * @param restr
       * @param type
       * @throws IOException
       */
      private void processRestrictions(final FeatureModel model,
                  final Feature feature, final String restr, final String type,
                  final int line) throws IOException {
            StringTokenizer tokenizer = new StringTokenizer(restr, "|");
            String element = null;
            while (tokenizer.hasMoreTokens()) {
                  element = tokenizer.nextToken();
                  getRestrictions(model, feature, element, type, line);
            }
      }

      /**
       * @param model
       * @param feature
       * @param restr
       * @param type
       * @throws IOException
       */
      void getRestrictions(FeatureModel model, final Feature feature,
                  final String restr, final String type, final int line)
                  throws IOException {
            StringTokenizer tokenizer = new StringTokenizer(restr, ",");
            String moduleNum = "";
            String absoluteNum = "";
            int ind = 0;
            while (tokenizer.hasMoreTokens()) {
                  moduleNum = tokenizer.nextToken();
                  if (ind == 1) {
                        Feature restriction = null;
                        absoluteNum = tokenizer.nextToken();
                        restriction = this.map.get(moduleNum + absoluteNum);
                        if (restriction == null)
                              redStream
                                          .println("Constraint Relation doesn't have a valid target feature
"
                                                                                      +
                                                "in the model. Constraint Relation wo'nt be processed
at
                                                line: "+ line);
                        else {
                              if (type.equals("requires")) {
                                    Implies implication = FeaturesFactory.eINSTANCE
                                                .createImplies();
                                    implication.setFrom(feature);
                                    implication.setTo(restriction);
                                    model.getRelationships().add(implication);
                              } else {
                                    Excludes exclusion = FeaturesFactory.eINSTANCE
                                                .createExcludes();
                                    exclusion.setFrom(feature);
                                    exclusion.setTo(restriction);
                                    model.getRelationships().add(exclusion);
```

```
                    }
               }
               ind++;
          }

     }

     /**
      * @param model
      * @param feature
      * @param valor
      * @param parent
      */
     private void processVariability(FeatureModel model, final Feature feature,
               final String value, final Feature parent) {
          if (parent != null) {
               StructuralRelationship relationship = FeaturesFactory.eINSTANCE
                         .createStructuralRelationship();
               if (value.equals("Common")) {
                    relationship.setLowerBound(1);
                    relationship.setUpperBound(1);
                    relationship.setFrom(parent);
                    relationship.setTo(feature);
                    parent.getChilds().add(relationship);
                    feature.setParent(relationship);
                    model.getRelationships().add(relationship);

               } else if (value.equals("Optional")) {
                    relationship.setLowerBound(0);
                    relationship.setUpperBound(1);
                    relationship.setFrom(parent);
                    relationship.setTo(feature);
                    parent.getChilds().add(relationship);
                    feature.setParent(relationship);
                    model.getRelationships().add(relationship);

               } else if (value.equals("Single")) {
                    // alternativeGroup
                    Group xorGroup = null;
                    if (parent.getGroup() != null)
                         xorGroup = parent.getGroup();
                    else {
                         xorGroup = FeaturesFactory.eINSTANCE.createGroup();
                         xorGroup.setName(parent.getName() + "Type");
                         xorGroup.setLowerBound(1);
                         xorGroup.setUpperBound(1);
                         xorGroup.setParentFeature(parent);
                    }
                    // relationship from group to feature
                    relationship.setLowerBound(1);
                    relationship.setUpperBound(1);
                    relationship.setFrom(xorGroup);
                    relationship.setTo(feature);
                    // set parent to the feature
                    feature.setParent(relationship);
                    // set childs
                    xorGroup.getChilds().add(relationship);
                    // add group to parent
                    parent.setGroup(xorGroup);
                    model.getRelationships().add(relationship);

               }
               // "Multiple"
               else {
                    // SelectionGroup
                    Group orGroup = null;
                    if (parent.getGroup() != null)
                         orGroup = parent.getGroup();
                    else {
                         orGroup = FeaturesFactory.eINSTANCE.createGroup();
                         orGroup.setName(parent.getName() + "Type");
```

```
                        orGroup.setLowerBound(1);
                        orGroup.setUpperBound(-1);
                        orGroup.setParentFeature(parent);

                    }
                    // relationship from group to feature
                    relationship.setLowerBound(1);
                    relationship.setUpperBound(1);
                    relationship.setFrom(orGroup);
                    relationship.setTo(feature);
                    // set parent to the feature
                    feature.setParent(relationship);
                    // set childs
                    orGroup.getChilds().add(relationship);
                    orGroup.setUpperBound(orGroup.getChilds().size());
                    // add group to parent
                    parent.setGroup(orGroup);
                    model.getRelationships().add(relationship);

                }
            }

        }

        public String fixString(String string) {

            String fixedString = "";

            for (int x = 0; x < string.length(); x++) {
                if (string.charAt(x) == '(' || string.charAt(x) == '-'
                            || string.charAt(x) == '/' || string.charAt(x) == '['
                            || string.charAt(x) == ']')
                    fixedString += "_";
                else if (string.charAt(x) == '&' || string.charAt(x) == '+'
                            || string.charAt(x) == ',')
                    fixedString += "And";
                else if (string.charAt(x) == 'á' || string.charAt(x) == 'ä')
                    fixedString += "a";
                else if (string.charAt(x) == 'é' || string.charAt(x) == 'ë')
                    fixedString += "e";
                else if (string.charAt(x) == 'í' || string.charAt(x) == 'ï')
                    fixedString += "i";
                else if (string.charAt(x) == 'ó' || string.charAt(x) == 'ö')
                    fixedString += "o";
                else if (string.charAt(x) == 'ú' || string.charAt(x) == 'ü')
                    fixedString += "u";
                else if (string.charAt(x) == '%')
                    fixedString += "pc";
                else if (string.charAt(x) == '*' || string.charAt(x) == ')')
                    fixedString += "";
                else if (string.charAt(x) == '0')
                    fixedString += "Zero";
                else if (string.charAt(x) == '1')
                    fixedString += "One";
                else if (string.charAt(x) == '2')
                    fixedString += "Two";
                else if (string.charAt(x) == '3')
                    fixedString += "Three";
                else if (string.charAt(x) == '4')
                    fixedString += "Four";
                else if (string.charAt(x) == '5')
                    fixedString += "Five";
                else if (string.charAt(x) == '6')
                    fixedString += "Six";
                else if (string.charAt(x) == '7')
                    fixedString += "Seven";
                else if (string.charAt(x) == '8')
                    fixedString += "Eight";
                else if (string.charAt(x) == '9')
                    fixedString += "Nine";
```

```
                else
                    fixedString += string.charAt(x);
            }
            fixedString = fixedString.replaceAll(" ", "");
            return fixedString;

        }
```

Apéndice C. Transformación Features2Fama para MediniQVT

```
transformation Features2Features(featuresDomain:features, famaDomain:FeatureModelSchema){

    key BinaryRelationType {name};
    key SetRelationType {name,cardinality};
    key GeneralFeature{name};
    key FeatureModelType{feature};
    key DocumentRoot{featureModel};

    top relation Model2Model {

        checkonly domain featuresDomain fmodel : features::FeatureModel{
            rootFeature = root :features::Feature{}
            };
        enforce domain famaDomain fmodel2 : FeatureModelSchema::DocumentRoot{
            featureModel = model : FeatureModelSchema::FeatureModelType {
                feature = first:FeatureModelSchema::GeneralFeature{

                    name=root.name
                }
                    }
            };
        }

    top relation StructuralRelationship2BinaryRelation{

        checkonly domain featuresDomain model : features::FeatureModel {
        };

        checkonly domain featuresDomain feature : features::Feature {
            childs = relationship : features::StructuralRelationship {}
        };

        enforce domain famaDomain feature2 : FeatureModelSchema::GeneralFeature {
            name = feature.name,
            binaryRelation = binaryRelation :FeatureModelSchema::BinaryRelationType {
                name = 'Relation_to_'+relationship.to.name,
                cardinality= cardinality :FeatureModelSchema::CardinalityType{
                    max=relationship.upperBound,
                    min = relationship.lowerBound
                },
                solitaryFeature=generalfeatures : FeatureModelSchema::GeneralFeature {
                    name = relationship.to.name
                }
            }
        };
    }

    top relation Group2SetRelation {
        checkonly domain featuresDomain feature : features::Feature {
            group = group : features::Group {
                childs = childRelationship : features::StructuralRelationship {
                }
            }
        }.
```

```
        enforce domain famaDomain feature2 : FeatureModelSchema::GeneralFeature {
            name = feature.name,
            setRelation = setRelation :FeatureModelSchema::SetRelationType {}
        };
        enforce domain famaDomain setRelation :FeatureModelSchema::SetRelationType {
                name = 'Grouped_Relation',
                cardinality= cardinality:FeatureModelSchema::CardinalityType{},
                groupedFeature=generalfeatures : FeatureModelSchema::GeneralFeature {
                    name = childRelationship.to.name
                }
        };
        enforce domain famaDomain cardinality :FeatureModelSchema::CardinalityType{
                    max=group.upperBound,
                    min = group.lowerBound
        };
    }

    top relation ExcludesRelationship2ExcludesType{

        checkonly domain featuresDomain model : features::FeatureModel {
            relationships = excludesRelationship : features::Excludes {
                from = fromFeature : features::Feature {},
                to = toFeature : features::Feature {}
            }
        };

        checkonly domain famaDomain featureModel : FeatureModelSchema::FeatureModelType
        {
            feature= feature :FeatureModelSchema::GeneralFeature{
                name = model.rootFeature.name
                }
        };
        enforce domain famaDomain featureModel : FeatureModelSchema::FeatureModelType {
            excludes= exclude:FeatureModelSchema::ExcludesType{
                name= 'Excludes_from_'+from.name+'_to_'+to.name,
                excludes=from.name,
                feature=to.name
            }
        };
    }
    top relation IncludesRelationship2RequiresType{
        checkonly domain featuresDomain model : features::FeatureModel {
            relationships = requiresRelationship :features::Implies {
                from = fromFeature : features::Feature {},
                to = toFeature : features::Feature {}
            }
        };
        checkonly domain famaDomain featureModel : FeatureModelSchema::FeatureModelType
        {
            feature= feature :FeatureModelSchema::GeneralFeature{
                name = model.rootFeature.name
                }
        };
        enforce domain famaDomain featureModel : FeatureModelSchema::FeatureModelType {
            requires= require:FeatureModelSchema::RequiresType{
                name= from.name+'_requires_'+to.name,
                requires=from.name,
                feature=to.name
            }
        };
    }
```