Integración de Técnicas de Análisis de Dominio con Especificaciones RSL

Laura Felice

Director: Dr. Daniel E. Riesco

Codirector: Dr. Gustavo Rossi

Tesis presentada a la Facultad de Informática de la Universidad Nacional de La Plata para obtener el grado de Magíster en Ingeniería de Software.

La Plata, 2013
Facultad de Informática
Universidad Nacional de La Plata

Agradecimientos

Un especial agradecimiento a Daniel Riesco, por su permanente apoyo y disponibilidad, en especial por la claridad de sus ideas y su valioso criterio en todas las etapas del desarrollo de esta tesis.

Y también a mis amigas y colegas Carmen, Marcela y Virginia por el constante apoyo y sobre todo por su amistad.

INDICE

LISTA DE FIGURAS Y TABLAS

LISTA DE FIGURAS Y TABLAS

Capítulo 1

Introducción

1.1 Motivación

Las especificaciones formales y las componentes reusables son dos metodologías que muestran alto impacto potencial sobre la productividad y confiabilidad del diseño de software. Ambas, son mutuamente beneficiosas. Por un lado, los métodos formales para el desarrollo de software o, simplemente métodos formales, se utilizan durante todas las etapas del ciclo de desarrollo de software y tienen la característica que usan formalismos matemáticos para la representación o derivación de los elementos involucrados en cada etapa.

Por otro lado, el reuso tiene un gran impacto potencial en los estados tempranos del proceso de diseño. Usar especificaciones formales para representar componentes de software facilita la determinación de software reusable, ya que las especificaciones formales caracterizan mas precisamente la funcionalidad del software, y una sintaxis bien definida es beneficiosa para la automatización. La creación de componentes reusables de alta confiabilidad es una herramienta poderosa en la demanda de calidad, en la construcción de software confiable y en sistemas críticos en seguridad.

Los métodos formales han alcanzado un uso más masivo en la construcción de sistemas reales, ya que ayudan a aumentar la calidad del software y la fiabilidad. Las especificaciones formales pueden ser usadas a lo largo de todo el ciclo de vida del desarrollo de software y también este desarrollo puede ser automatizado por medio de herramientas de amplia variedad y propósito como model checking, verificación, animación, generación de datos para testing, como también refinamiento de especificaciones a implementaciones [46]. Cuando se usan en etapas iniciales del proceso, ayuda a revelar ambigüedades, omisiones, inconsistencias, errores o interpretaciones erróneas que podrían ser detectados durante pruebas costosas y en las fases de depuración.

Sin embargo, las especificaciones formales no son muy familiares para los stakeholders, cuya participación activa es crucial en los primeros estados del proceso de desarrollo de

software para entender y comunicar el problema [31]. Estas actividades son especialmente útiles en el análisis de dominio, pues su primera etapa consiste en capturar el conocimiento de un dominio en particular, lo que hace necesario disponer de un modelo que sea comprensible por los ingenieros de software y expertos del dominio.

Para contribuir a reducir esta brecha, hemos trabajado en la integración de una fase de análisis de dominio en el método formal de desarrollo de software RAISE [19], a fin de especificar una familia de sistemas para producir aplicaciones cualitativas y fiables en un dominio, promover la reutilización temprana y reducción de los costos de desarrollo.

Considerando lo argumentado anteriormente, pensamos en la utilidad de analizar una integración entre técnicas de análisis del dominio y especificaciones formales. De esta manera, se pretende reducir la brecha que existe entre estas etapas tomando las ventajas que ofrece cada técnica.

El método RAISE ha sido diseñado para uso en desarrollos reales. Este método provee facilidades para el uso industrial de métodos formales en el desarrollo de sistemas de software. Incluye un gran número de técnicas y estrategias para realizar desarrollo formal y pruebas, así como un lenguaje de especificación, el RAISE Specification Language (RSL) [18], y un conjunto de herramientas para la edición, checking, impresión, almacenamiento, transformación y razonamiento acerca de las especificaciones [20].

En particular nuestra propuesta tiene como objetivo la integración del método FORM [25] de representación de dominios, con el lenguaje de especificación formal RSL. Se ha definido un conjunto de heurísticas y pasos que guían en el proceso de transformación entre modelos del dominio y especificaciones RSL. Los modelos del dominio se representan mediante la técnica de Feature Modeling [12]. Se ha desarrollado parte del proceso de la transformación en lenguaje ATL [3] para la definición de especificaciones iniciales RSL. El proceso de transformación toma los modelos del Feature Model y define las especificaciones RSL con sus atributos y relaciones.

En la sección siguiente se describen los objetivos de nuestra propuesta y detalles de los procedimientos seguidos.

1.2 La propuesta

1.2.1 Objetivos

En este trabajo perseguimos un objetivo principal que consiste en la integración al método RAISE con técnicas de análisis de dominio.

Básicamente, el método RAISE consiste de:

- un lenguaje de especificación (RSL), el cual es un lenguaje de diseño y especificación muy poderoso;
- un método de desarrollo, que consta de un conjunto de técnicas que se aplican a cuatro procedimientos principales:
  - Especificación
  - Desarrollo
  - Justificación
  - Traducción

El énfasis está puesto en el estudio del primer procedimiento del método: las especificaciones de un sistema, a fin de introducir el análisis de dominio para contribuir al reuso de especificaciones en este nivel.

La propuesta de este trabajo se ajusta, en parte, en cómo aplicar resultados del análisis de dominio al desarrollo de especificaciones de dominio adaptables y reusables, proveyendo guías específicas. El análisis del dominio aplicado aquí está centrado en analizar y modelar aspectos comunes y aspectos variables de las aplicaciones en un dominio, en términos de “features applications” [12].

En la sección siguiente se describe en detalle la propuesta de este trabajo.

1.2.2 Descripción general

El análisis de dominio está estrechamente vinculado a la reutilización de software, que busca la construcción de nuevos sistemas informáticos a partir de componentes, diseños o especificaciones creadas anteriormente. Por lo tanto, disponer de una representación sistematizada del conocimiento hará mas sencilla la creación de módulos que puedan aplicarse en múltiples escenarios [6].

El análisis de dominio orientado a features puede facilitar la identificación de los factores que diferencian una aplicación específica de otra aplicación relacionada.

Existen diferentes formas sistemáticas de identificar y modelar distintos tipos de features y en mostrar como influencian el desarrollo de componentes reusables del dominio. Estas formas de derivar componentes desde el modelo de features brindan ventajas comparadas con otros métodos, como por ejemplo:

- Se proveen guías específicas para identificar objetos reusables vinculando categorías de features a categorías de objetos. Hay que proveer un mapping explícito o guías para analizar resultados del análisis a objetos reusables y desarrollo de arquitecturas. Los objetos reusables en este trabajo son las especificaciones RSL. Cuando se especifica usando el método formal RAISE, escribir una especificación inicial es una de las tareas más críticas, ya que esta especificación debe capturar los requerimientos en una forma precisa y formal [19] y [18].

- Se obtiene un conjunto de módulos RSL candidatos más sólido desde el modelo de features ya que podemos derivar las relaciones (agregación y generalización) entre objetos desde el modelo de features analizando las relaciones entre features (“compuesto-de”, “generalización” e “implementado-por”).

- Se pueden derivar arquitecturas del dominio (modelo del subsistema, modelo del proceso, modelo de módulos) desde el modelo de features aplicando las guías y el framework del método aplicado.

En este trabajo nos proponemos utilizar modelos de features para representar el análisis del dominio, ya que facilitan la personalización de los requisitos de software. En análisis de dominio, los features y relaciones entre los mismos (el modelo del dominio) se utilizan para organizar las necesidades de un conjunto de aplicaciones similares en un dominio de un sistema de software.

Nuestra propuesta tiene como objetivo principal comenzar el desarrollo de una especificación definiendo un modelo de features siguiendo una de las propuestas más difundidas que faciliten la construcción de estos modelos: el método Feature-Oriented Reuse Method (FORM). De este modelo se extrae un conjunto de módulos candidatos a transformarse en especificaciones RSL, aplicando un conjunto de heurísticas. Aplicando los principios del método RAISE se organizan los módulos para convertir dichas especificaciones en un modelo o arquitectura de niveles para convertirlas en una más concreta y luego obtener automáticamente un prototipo que valida la especificación. El uso de un modelo de features está motivado por el hecho de que los stakeholders expresan muy comúnmente las características del producto en términos de "las características del producto tiene y/o brinda", utilizando las mismas para comunicar sus ideas, necesidades y problemas. Cada módulo en la arquitectura de niveles RSL es diseñado utilizando los derivados del modelo de features.

La primera especificación (la especificación inicial) será generalmente un único módulo del sistema aplicativo. En niveles subsecuentes, habrá un módulo por cada uno del nivel anterior, pero puede haber otros agregados. Cada módulo se desarrolla a partir de sus predecesores de acuerdo a las técnicas del método RAISE. Básicamente, hay tres etapas involucradas:

- análisis: produce la especificación inicial
- diseño: produce la especificación final
- traducción: produce el programa ejecutable

Hay una noción de orden porque cada etapa es necesaria como una entrada para su sucesor, pero en la práctica el proceso es iterativo.

Se puede trabajar en forma top-down: se empieza con el primer nivel, la especificación inicial, seguida por la segunda, luego la relación entre la primera y la segunda y su justificación, etc. Si los requerimientos son claros, todo funcionará. Pero en la práctica, esto no es muy frecuente. Se necesita dedicar tiempo a la etapa de análisis. El proceso es extremadamente iterativo, lleva un tiempo definir la especificación inicial y es necesario probar diseños antes de decidir sobre la especificación inicial. Las especificaciones iniciales buenas son difíciles de escribir, debido a que la abstracción que es corta y simple y la adecuada para expresar propiedades importantes es difícil de encontrar y formular.

El análisis también involucra aspectos como la identificación de los objetos, sus atributos, las relaciones entre ellos, etc. Tales aspectos son tema generalmente de otros métodos llamados “tradicionales”, “estructurados” u “orientados a objetos”. Sin embargo, es efectivo empezar respondiendo estas preguntas y desarrollar la primera especificación para capturar las respuestas a ellas. Es poco probable que esta primera especificación sea la inicial, puede ser demasiado concreta e involucrar varios módulos. Pero ayudará a analizar los requerimientos.

Teniendo una primera especificación, quizás parcialmente bosquejada, hay dos opciones:
- completar la primera especificación y considerarla la inicial. Puede servir aún como la especificación final, en cuyo caso se estaría haciendo especificación formal en vez de desarrollo formal. Para problemas que involucran dominios bien conocidos, y para algunas componentes de problemas más grandes, esto es suficientemente efectivo.
- se puede formular una abstracción de la primera especificación para formar la especificación inicial. Luego se puede verificar que la primera es una implementación de la abstracción (en la práctica no es útil). Habiendo formulado una abstracción para la especificación inicial, la primera especificación se podría usar como un segundo nivel, pero lo que sucede con frecuencia es que se formula un nuevo segundo nivel, usando

algunas ideas de la primera especificación pero difiriendo en general en que se construye más cuidadosamente, más adecuada para futuros desarrollos. El objetivo de la primera especificación es análisis de requerimientos; el objetivo de la inicial y las subsecuentes es formalización de requerimientos y diseño.

Los **features** son características identificables unívocamente de un dominio de aplicación según el punto de vista del usuario o desarrollador [12]. Algunos de ellos, los que representan los conceptos del dominio o entidades, pueden ser modelados como objetos, otros, los que representan aspectos operacionales, pueden ser operaciones de los objetos; y otros tales como los aspectos no funcionales, restricciones del entorno, decisiones de diseño, etc. pueden ser modelados como parámetros que pueden instanciar un objeto genérico a objetos de aplicación específicos. El modelado basado en features, Feature Modeling, es el proceso por el cual se busca lo común y lo variable de los conceptos que definen una Línea de Productos de Software [7], así como las relaciones que puedan existir entre ellos, para posteriormente, organizarlo todo en un esquema jerarquizado denominado el Feature Model.

Este modelo describe todas las posibles variantes de productos software que se pueden obtener mediante la Línea de Productos a la que pertenezca. La generación de un determinado producto de software supone la eliminación de toda variabilidad posible que estuviese representada en el modelo de features, y a ello se llega mediante la selección o eliminación de los diferentes features y subfeatures, para obtener la configuración concreta del producto de software que se desea obtener.

El Feature Modeling tiene sus orígenes en el Software Engineering Institute (SEI), donde fue originariamente usado por el método FODA (Feature-Oriented Domain Analysis) de análisis de dominio [26]. Posteriormente han surgido otros enfoques del modelo, sobre todo extendiendo la notación original de FODA, como en [8], [9], [10]. Las extensiones allí propuestas, permiten un mayor grado de expresión y potencial para las aplicaciones en donde se aplica el modelo. El Feature Modeling ha sido usado para representar modelos de dominios de aplicación en muchos enfoques de la ingeniería del dominio. Entre algunos de los métodos que han abordado con mayor interés se encuentran: FORM, ODM [42] y Generative Programming [12]. Cada uno de estos métodos define un proceso para el análisis del dominio.

Con las diferentes categorías de features se puede direccionar a distintos niveles de abstracción de objetos. Los features que se encuentran en el denominado "*capability layer*" pueden ser usados para identificar objetos participantes en interacciones entre usuarios y el sistema, los que se encuentran en el "*operating environment layer*" son los que representan las variaciones del hardware o software que tienen las interfaces. Los que están en el denominado "*domain technology layer*" se usan para identificar objetos que contienen conceptos del dominio y algoritmos, y los que se hallan en el

"*implementation technique features*" pueden ser usados para identificar objetos que contienen decisiones de diseño y detalles de implementación [24].

Dentro del marco descripto, hemos analizado el caso de estudio correspondiente a la familia de productos de Electronic Shops (eShop) [33]. El árbol completo que modela el caso con sus constraints fue dividido para su estudio en varios grupos de acuerdo al análisis de cada uno de los casos. A partir del modelo del eShop se establecen los mappings para features que pertenecen al nivel de capability layer que propone el método FORM.

Se define además, parte del proceso que transforma los features en schemes RSL. Formalmente y dentro del paradigma de MDD [32], se implementa el proceso de transformación que toma los modelos del Feature Model y define schemes con sus atributos y relaciones, en el lenguaje ATL.

1.3 Trabajos relacionados

A pesar de la cantidad de investigación tanto en el área de métodos formales como en el análisis de dominio, las investigaciones que vinculan el objetivo principal de esta tesis no son tantas. Sin embargo, existen numerosos trabajos en el área de modelos de features en cuanto a propuestas de mappings hacia otros modelos, que se pueden considerar como relacionados, principalmente mencionamos los trabajos de Czarnecki y otros que se basan, en parte, en las propuestas de este autor.

El trabajo de Krzysztof Czarnecki y Michael Antkiewicz en [11] puede ser visto desde dos perspectivas: dar semántica a los features en los modelos de features haciendo un mapping a otros modelos, y también usar los features models para proveer una representación concisa de la variabilidad de los modelos. Este trabajo provee el mecanismo de traceability entre features y sus realizaciones en los modelos. Esto es especialmente útil en la etapa de requerimientos, aunque también la propuesta está para modelos en cualquier nivel (modelos de arquitectura e implementación). También se puede destacar el trabajo que hace Czarnecki y Kim en [10] donde emplean Object Constraint Language (OCL) para expresar restricciones adicionales del modelo. En este trabajo, los feature models son convertidos en diagramas UML donde las restricciones de integridad son aplicadas mediante expresiones OCL.

Valentino Vranic y Ján Snirc [50] presentan un enfoque para la integración de los modelos de features con UML, proponiendo la extensión de su metamodelo. Se considera crucial aquí, asegurar la abstracción de los elementos del modelo para el desarrollo de extensiones del mismo. Dicha extensión está basada en la notación de

Czarnecki-Eisenecker [10]. Desarrollaron un profile UML, referenciando múltiples trabajos relacionados con UML.

Por el lado de los métodos formales, específicamente con el lenguaje RSL, en [1] se ha trabajado en el desarrollo de diagramas inspirado visualmente por los diagramas de clases UML, pero semánticamente directamente relacionado con RSL. Los autores han desarrollado un plug-in para el editor de Eclipse, que permite al usuario dibujar diagramas y traducirlos directamente a RSL. La inspiración de la que hablan los autores es similar al espíritu de este trabajo: la combinación de los puntos fuertes entre ambos métodos. El autor de esta tesis discute también la integración de las técnicas gráficas con las especificaciones formales en la que describen las notaciones de los diagramas de secuencia y los Statecharts y se propone un método usando diagramas con estas notaciones para restringir una especificación expresada en RSL.

1.4 Publicaciones

Los siguientes artículos publicados son algunos de los resultados obtenidos respecto al tema de esta tesis:

- *Using ATL Transformations to Derive RSL Specifications from Feature Models*

Felice, Laura; Ridao, Marcela; Mauco, María Virginia; Leonardi, María Carmen.
Proceedings of The 2011 International Conference on Software Engineering Research & Practice. Volume I. pp: 273 – 279.
H. Arabnia, Hassan Reza, Leonidas Deligiannidis (Eds.) 18-21 July 2011. WorldComp´11. Las Vegas. USA.
ISBN:1-60132-199-6, 1-60132-200-3 (1-60132-201-1).

- *A Strategy to Derive RSL Specifications from Feature Models*

Laura Felice, María Carmen Leonardi, María Virginia Mauco, Germán Montejano, Daniel Riesco, Narayan Debnath.
Proceedings 18th International Conference on Software Engineering and
Data Engineering. SEDE 2009, June 22-24, 2009, Las Vegas, USA. 6 páginas.

- *A Feature model of E-Government Systems Integrated with Formal Specifications*

Narayan Debnath, Laura Felice, Germán Montejano, Daniel Riesco.
Proceedings de: ´5th International Conference on Information Technology : New Generations. ITNG 2008´. April 7-9, 2008, Las Vegas, USA.

- *Integrating Formal Method with a Reuse Technique*

Laura Felice, Carmen Leonardi , Virginia Mauco.

Proceeding de Information Resources Management Association International Conference. (IRMA 2007) Managing Worldwide Operations and Communications with Information Technology (Mehdi Khosrow-Pour ed.). Vancouver. Canadá. Mayo 2007. ISBN: 978-1-59904-929-8.

- *Using a Feature Model for RAISE specification reusability*

Riesco, D; Felice, L; Debnath, N; Montejano, G.
IEEE- IRI 2005 ´International Conference on Information Reuse and Integration
Proceedings de the 2005 IEEE International Conference on Information Reuse and Integration (IRI 2005) IRI - 2005, August 15-17, 2005. Las Vegas USA. IEEE Systems, Man, and Cybernetics Society 2005, ISBN 0-7803-9093-8. pp: 306-311.

- *Incorporating a reuse model to the RAISE Formal Method*

Riesco, D; Felice, L; Debnath, N; Montejano, G.
IEEE- IRI 2004 ´International Conference on Information Reuse and Integration
Proceedings de the 2004 IEEE International Conference on Information Reuse and Integration (IRI 2004) ISBN: 0-7803-8819-4 pp: 133-138.

1.5 Organización de la tesis

Este trabajo está organizado de la siguiente manera:

- Capítulo 2: se introduce a grandes rasgos las Líneas de Productos de Software donde se dan las nociones de Feature Modeling y el Desarrollo de Software dirigido por Modelos (MDD). También allí se describe el metamodelo propuesto del Feature Model con los elementos necesarios para establecer el mapping definido en el capítulo 4.

- Capítulo 3: presenta el método RAISE, y una breve descripción del RAISE Specification Language detallando los elementos necesarios para las especificaciones iniciales propuestas en este trabajo. Además, como en el capítulo 2, se presenta el metamodelo que describe los elementos necesarios del lenguaje RSL para establecer el mapping mencionado anteriormente.

- Capítulo 4: en este capítulo se revisan las construcciones presentes en un Feature Model (FM) y se discute el proceso que puede ser usado como transformación en expresiones RSL, mediante el análisis de un conjunto de casos. Dentro del paradigma MDD, se presenta además, la transformación en el lenguaje ATL para la definición de schemes RSL. Se ha definido el proceso para la mayoría de las transformaciones propuestas, el detalle de las mismas se encuentra en Anexo B.

- Capítulo 5: en este capítulo se recopilan las conclusiones a las que se ha llegado una vez se ha terminado el trabajo. Estas conclusiones pretenden plantear posibles líneas futuras de trabajo así como ampliaciones del trabajo realizado.

- Apéndice A: contiene el caso de estudio del modelo de Electronic Shop.

- Apéndice B: contiene el código ATL de las transformaciones definidas en el capítulo 4.

Capítulo 2

Líneas de Productos de Software–Feature Model

Antecedentes

En este capítulo se introduce a grandes rasgos las Líneas de Productos de Software y se describe qué es un Feature Model y el Desarrollo de Software dirigido por Modelos. También se presenta el metamodelo del Feature Model con los elementos que son fundamentales para establecer el mapping definido en el capítulo 4 de esta tesis.

2.1 Líneas de Productos de Software

En los últimos años, el análisis de dominio ha tenido una atención creciente debido a la importancia de la llamada Ingeniería de Líneas de Productos de Software (LPS) [44], cuyo objetivo es el desarrollo de aplicaciones informáticas a partir de una serie de características y activos clave para el área de actividad a la que va dirigida la aplicación.

Las LPS tienen su origen en el Software Engineering Institute (SEI) [43]. El SEI es un centro de desarrollo e investigación patrocinado por el Departamento de Defensa de USA y operado por la Universidad Carnegie Mellon en Pittsburgh.

El SEI ayuda a promover los principios y prácticas de la Ingeniería de Software, trabajando en estrecha colaboración con organizaciones de defensa y del gobierno, la industria y las universidades para la mejora continua de los sistemas de software. Su propósito principal es ayudar a las organizaciones a mejorar sus capacidades y a desarrollar o adquirir el software adecuado. El SEI transmite sus tecnologías a la comunidad de la ingeniería de software a través de cursos, conferencias, technical reports y una red asociada. Se ha convertido en el líder internacional reconocido en las LPS desarrollando la base de conocimientos necesarios para efectuar las prácticas sólidas de las líneas de productos. Las organizaciones han recurrido al SEI y sus recursos para poner en marcha las líneas y perfeccionar sus esfuerzos. En particular, el SEI ha desarrollado un framework denominado *Framework for Software Product Line*

Practice, un conjunto de patrones para guiar la aplicación de esas prácticas, métodos de diagnóstico denominados *Product Line Quick Look y el Product Line Technical Probe*, métodos y enfoques para realizar análisis, definir arquitecturas, desarrollar business cases, un modelo llamado *Structured Intuitive Model for Product Line Economics*, y modelos económicos adicionales, así como estrategias para la producción, manejo de variabilidad, planning y desarrollo en el contexto de otras tecnologías.

Las LPS tienen grandes similitudes con la ingeniería y el análisis de dominio, en los dos casos se trata de extraer las características comunes de un mismo tipo de sistemas. La principal diferencia está en que la LPS tiene como input el análisis de mercado, que no está presente en las técnicas de análisis de dominio tradicionales [29]. Se pueden considerar como un paradigma que ha guiado a las organizaciones al desarrollo de los productos a través de la estrategia de reuso de los llamados ´activos´ de la LPS (por ejemplo arquitecturas y componentes de software).

El desarrollo de activos de una LPS requiere analizar los aspectos comunes y variables entre productos de manera profunda, de modo que los aspectos comunes puedan ser utilizados para diseñar activos reusables y los aspectos variables puedan ser usados para diseñar activos adaptables y configurables.

Según Clements y Northrop [7], una LPS consiste en: "*Un conjunto de sistemas de software que comparten un conjunto común y gestionado de características (features) que satisfacen las necesidades específicas de un segmento particular de mercado y que se desarrollan a partir de un conjunto común de activos de una forma preestablecida*".

El objetivo de una LPS es crear la infraestructura adecuada para una rápida y fácil producción de sistemas similares, destinados a un mismo segmento de mercado. Las LPS se pueden ver análogas a las líneas de producción industriales, donde productos similares o idénticos se ensamblan y configuran a partir de piezas prefabricadas bien definidas, que son reutilizadas para la construcción de productos con características similares. Un ejemplo clásico es la fabricación de automóviles, donde se pueden crear decenas de variaciones de un único modelo de auto con sólo un grupo de piezas diseñadas y una fábrica específicamente concebida para configurar y ensamblar dichas piezas.

La variabilidad se refiere a la incorporación de mecanismos de variación, por ejemplo: componentes plugin, mejoras orientadas a aspectos, etc.; que permiten la construcción de un conjunto reutilizable de software representando el rango completo o la LPS. La derivación del producto es el proceso de construcción de productos de software específicos, dada una configuración específica, por ejemplo la selección de un conjunto válido de variantes siguiendo las directivas para la composición común y los conjuntos de software variables.

El desarrollo de las LPS se compone de dos procesos de desarrollo de software diferentes pero relacionados (figura 2.1) conocidos como Ingeniería del Dominio e Ingeniería de la aplicación.

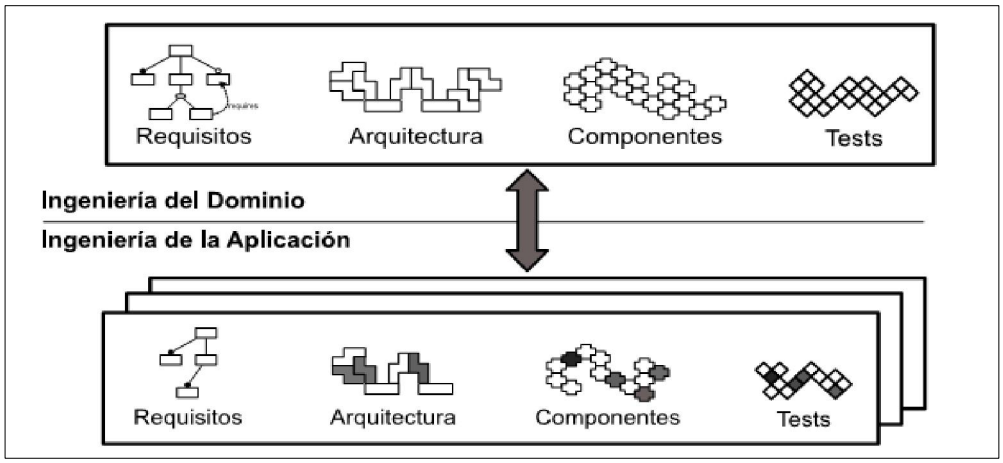


Figura 2.1 Proceso de desarrollo en LPS

En el nivel de ingeniería del dominio, se comienza por la documentación de requisitos que describen una línea de productos similares para un segmento de mercado específico. Luego, se diseña una arquitectura e implementación de referencia para esta familia de productos. Esta arquitectura de referencia contiene los elementos que son comunes para todos los productos en la familia, pero también debe contener mecanismos para permitir las diferentes variaciones de los diferentes productos pertenecientes a la misma familia.

En el nivel de ingeniería de la aplicación, se comienza con la documentación de requisitos de un producto específico. Esta documentación establece las variaciones específicas que deben de ser incluidas en este producto. Con esta información, se introduce las variaciones en la arquitectura de referencia y en la implementación, obteniendo como resultado un producto software único.

El principal beneficio de adoptar la metodología de LPS es la reducción en tiempo y esfuerzo en desarrollo para desarrollar productos específicos pertenecientes a la misma familia.

Para ser capaces de responder a las necesidades particulares de las demandas y producir LPS de forma sistemática, una de las cuestiones clave es establecer una forma de especificar los productos que una LPS es capaz de producir. La solución natural para el caso de estas líneas parecía simple: los productos de una LPS se diferencian por sus características **–features-**, siendo un feature un incremento en la funcionalidad del

producto o más formalmente: “una propiedad de un sistema que es relevante a algunos stakeholders y que es usada para capturar aspectos comunes o diferenciar entre sistemas de una misma familia” [12]. De esta manera, un producto se determina por los features que posee.

2.2 Feature Modeling

Para describir todos los productos que una LPS es capaz de producir, es necesario disponer de un modelo que permita describir todas las posibles combinaciones de features de estos productos. Para ello, se proponen los Feature Models (FM) como forma de describir una LPS. Los FM están reconocidos como una de las contribuciones más importantes en la ingeniería de LPS [44].

Uno de sus objetivos principales es capturar los aspectos variables y los aspectos comunes entre los distintos productos. Para ello los FM organizan el conjunto de features jerárquicamente mediante relaciones entre ellos:

- Relaciones entre un feature padre o composición y un conjunto de features hijos o subfeatures.
- Relaciones no jerárquicas: si el feature A aparece, entonces el feature B se debe incluir (o excluir).

Las primeras pueden ser consideradas como restricciones estructurales y a las segundas como restricciones de usuario.

La figura 2.2 ilustra un ejemplo simplificado de un FM inspirado en el dominio de la telefonía móvil. En el modelo se puede ver cómo se utilizan los features para especificar y desarrollar software para telefonía móvil. El software cargado en el teléfono está determinado por los features que posee. Por ejemplo, un sistema de esta línea puede ser especificado por el conjunto de features {Calls, Messaging, Connectivity}. Esto significa que el producto de software ofrece apoyo a realización de llamadas, a envío de mensajes y a conexión por medio de WiFi, respectivamente.

El Feature Modeling también es originario del SEI, donde fue inicialmente usado por el método FODA (Feature-Oriented Domain Analysis) de análisis de dominio.

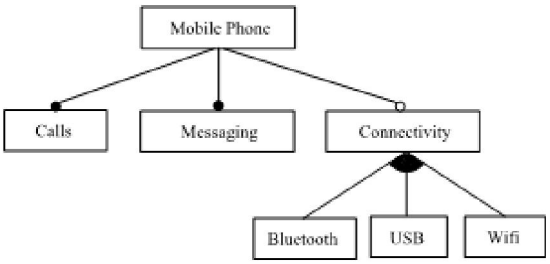


Figura 2.2 FM para sistemas de telefonía móvil

Posteriormente surgen otros enfoques del modelo. La notación original de FODA fue extendida por Czarnecki en [26] con respecto a la cardinalidad de un FM. También en [10], la notación se extiende con respecto a los atributos de un feature, referencias a los modelos y los atributos de referencia, las configuraciones multi-nivel y multi-estados, y en [9] la extensión es con respecto a las restricciones externas y la cardinalidad de features agrupados. Estas extensiones permiten un mayor grado de expresión y potencial para las aplicaciones en donde se aplica el modelo. El Feature Modeling ha sido usado para representar modelos de dominios de aplicación en muchos enfoques de la ingeniería del dominio. Entre algunos de los métodos que han abordado el modelo con mayor interés se encuentran: FORM [25], ODM [42] y Generative Programming [12]. También, ha sido la base del diseño multi-paradigma con Feature Modeling (MPDFM), el método introducido en [47].

Los FM se utilizan como entradas para otros procesos del desarrollo de LPS como la ingeniería de requisitos o la programación orientada a features. En este punto, el análisis automático de los FM puede ser de ayuda para aquellos procesos que se basan en ellos.

Cabe destacar que en los últimos años se ha añadido a los FM un simple pero importante concepto, como son los features *clonables*, features que pueden aparecer con un número diferente o cardinalidad dentro de un producto. Por ejemplo, el FM de una ´casa´ puede tener como feature clonable a la ´planta´ de la casa, dado que una casa posee un número variable de plantas es decir, usando la terminología, diferentes clones del feature planta. Esta modificación hace que se pueda modelar variabilidad estructural, tal como que una casa tenga un número variable de plantas, en los FM, acercando su potencia expresiva a la de los lenguajes de dominio específico para las LPS [39].

Hay otra característica de estos modelos que aporta simpleza y expresividad a los modelos, se trata de las *referencias*. Cuando un feature referencia a otro, significa que tiene los mismos descendientes que el feature referenciado. Un feature sólo puede

referenciar a otro, en cambio puede ser referenciado por cualquier número de features (siempre y cuando las referencias no provoquen un ciclo infinito).

2.2.1 Nociones básicas del modelo

A continuación se definen los conceptos fundamentales de los elementos que forman parte de un FM.

- **Concept**: el concept (concepto) es un punto de vista de una clase o categoría de elementos en un dominio. Los elementos individuales que corresponden a estos puntos de vista se llaman instancias del concepto.

- **Feature**: es una propiedad importante de un concept [12]. Un feature es un aspecto, cualidad o característica distintiva de un sistema de software visible para el usuario.

Un feature tiene propiedades que son los atributos que representan cualquier tipo de aspecto variable o medible. Pueden existir dependencias entre las propiedades y entre otros features u otras propiedades de features. Estas dependencias constituyen un tipo de flujo de información entre ellos. Las propiedades pueden tener un rol importante en el análisis de aspectos comunes o variables de un feature.

Los features pueden expresar funcionalidad, cualidades o características abstractas (a los cuales llamaremos features abstractos). Por medio de la relación provided-by, se puede expresar la realización de una abstracción [17].

Un feature puede ser común es decir que está presente en todas las instancias del concepto, o variable estando presente solamente en algunas instancias del concepto. Los features que están conectados directamente a un concepto o a un feature, son los denominados directos, los demás son features indirectos. Cualquier feature puede estar aislado y modelado como un concepto, por lo tanto un feature es una relación entre dos conceptos. Un feature puede ser un feature raíz siendo el único que no tiene padre.

Se define como feature **solitary** a aquellos que no están agrupados en ningún tipo de agrupamiento.

Sintetizando, los features pueden actuar como medios para:
- modelar grandes dominios,
- manejar la variabilidad de productos de una LPS,
- encapsular requerimientos del sistema,
- guiar el desarrollo de una LPS,

- futuros plannings,
- comunicación entre los stakeholders del sistema.

La representación gráfica de un FM es el llamado **Feature Diagram** (FD), pudiéndose completar la definición de un FM con información adicional acerca de concepts y features y, con restricciones y reglas de dependencia asociadas con los diagramas.

2.2.2 Feature Diagram

Un diagrama de features (FD en inglés) es la representación gráfica de un FM. Es un árbol dirigido cuya raíz representa un concept y el resto de los nodos representan los features. Los arcos son dibujados conectando subconjuntos disjuntos originados de un mismo nodo. Hay dos tipos de arcos, uno vacío y el otro lleno, utilizados para denotar features alternativos y features OR respectivamente. Un feature puede ser seleccionado desde un conjunto de features alternativos, y cualquier subconjunto de todos los features puede ser seleccionado desde el conjunto de los features OR. Si es opcional, cada alternativo seleccionado o feature OR puede quedar excluido. Un feature puede ser abierto, lo que significa que se espera que tenga nuevos subfeatures variables directos. Esto se indica directamente en un FD introduciendo el nombre del feature entre corchetes y opcionalmente los puntos suspensivos a sus subfeatures. Se utiliza la notación de Czarnecki descrita en [10].

El FD de la figura 2.3 muestra la notación. Los features f1, f2, f3, y f4 son features directos del concept C1, mientras que los demás son features indirectos. Los features f1y f2 son features alternativos mandatarios. El f3 es un feature opcional. Los features f5, f6, y f7 son features OR mandatarios, también son subfeatures de f3. El feature f3 es abierto. Además los puntos suspensivos expresan más precisamente que se pueden esperar nuevos features OR mandatarios en el grupo de f5, f6, y f7.

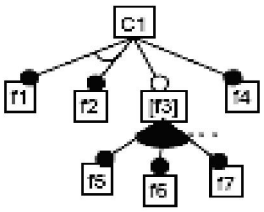


Figura 2.3 Feature Diagram

El Cardinality-based Feature Modeling [10] integra un número de extensiones de la notación original FODA, siendo el concepto de cardinalidad el más importante de las extensiones. La figura 2.4 correspondiente a la familia de Electronic Shops ilustra el

modelo con las extensiones mencionadas. Tanto el diagrama como la tabla 2.1 presentadas a continuación fueron extraídas de [10].

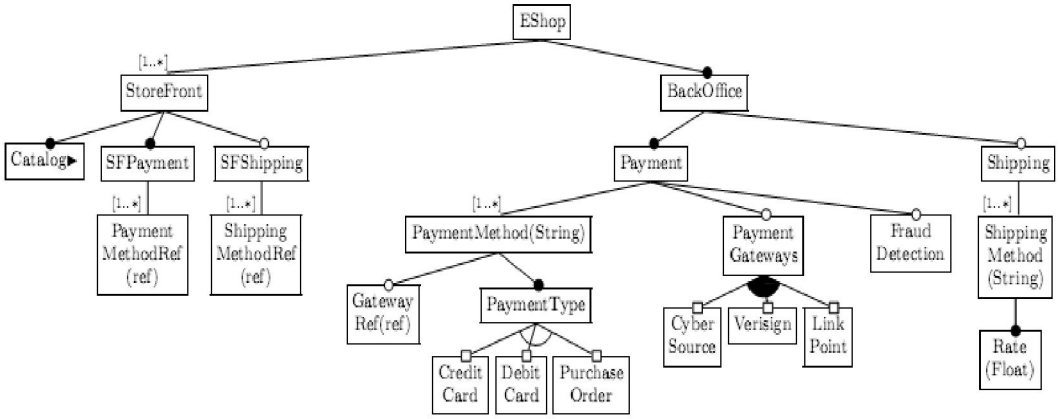


Figura 2.4 FD de Electronic Shop

El modelo cardinality-based es una jerarquía de features, donde cada uno tiene una cardinalidad. La cardinalidad del feature es un intervalo de la forma [m..n], donde m, n $\in Z \wedge n \in Z \cup \{*\} \wedge 0 \leq m \wedge (m \leq n \vee n = *)$. La cardinalidad denota cuantos clones del feature (con el subárbol entero) pueden ser incluidos como hijos de un feature padre cuando se especifique una configuración concreta. Se permite que la cardinalidad tenga un límite superior ilimitado. Los features con cardinalidad [1..1] son los features mandatarios, mientras que los features con cardinalidad [0..1] son los llamados opcionales. Según el ejemplo de la figura 2.4, Payment es un feature de tipo mandatory, mientras que Shipping es de tipo optional. Los features que tienen un límite más que 1 pueden ser clonados durante la configuración. El clonado es útil si la configuración necesita incluir múltiples copias de una parte, donde cada parte puede ser configurada de manera diferente. En el ejemplo presentado, la configuración de un Electronic shop puede incluir múltiples Store Fronts que pueden ser configurados de diferente manera, haciendo selecciones diferentes de métodos de pagos, y métodos de shipping por ejemplo.

Además, los features pueden ser agrupados en feature groups, donde cada feature group tiene una cardinalidad. La cardinalidad es un intervalo de la forma <m-n> donde m, $n \in Z \wedge 0 \leq m \leq n \leq k$, donde k es el número de features en el grupo. La cardinalidad del grupo denota cuántos miembros del grupo pueden ser seleccionados. Por ejemplo al menos y a lo sumo uno de los features CreditCard, DebitCard, y PurchaseOrder deben ser seleccionados como un subfeature de PaymentType. La tabla 2.1, extraída de [10] resume la notación.

Symbol	Explanation
	Solitary feature with cardinality [1..1], i.e., *mandatory* feature
	Solitary feature with cardinality [0..1], i.e., *optional* feature
	Solitary feature with cardinality $[0..m]$, $m > 1$, i.e., *optional clonable* feature
	Solitary feature with cardinality $[n..m]$, $n > 0 \wedge m > 1$, i.e., *mandatory clonable* feature
	Grouped feature with cardinality [0..1]
	Grouped feature with cardinality [1..1]
	Feature F with attribute of type T and value of *value*
	Feature model reference F
	Feature group with cardinality $\langle 1-1 \rangle$, i.e. *xor*-group
	Feature group with cardinality $\langle 1-k \rangle$, where k is the group size, i.e. *or*-group
	Feature group with cardinality $\langle i-j \rangle$

Tabla 2.1 Notación de un FD

Un feature tiene propiedades que son los atributos que representan cualquier tipo de aspecto variable o medible. Se permite a lo sumo un atributo por feature. Si fuera necesario más de un atributo, entonces se introducen varios subfeatures con un atributo cada uno. El tipo de atributo puede ser un tipo básico como String o Integer, o FRef, el cual denota el conjunto de todas las referencias a features en una cierta configuración. FRef se refiere a un atributo de referencia a un feature. Por ejemplo, en el FM de la figura 2.4 se usan los atributos String para representar los nombres de los tipos de pagos y métodos de envío, y un atributo Float para representar el tipo de un método de envío. Además, los atributos de referencia de un feature son usados en PaymentMethodRef y ShippingMethodRef para apuntar a pagos predefinidos y métodos de envío en el back office, es decir, clones de PaymentMethod y ShippingMethod. Aunque una referencia a un feature puede apuntar a cualquier feature en una configuración, se puede restringir que los atributos apunten a los clones de PaymentMethod o ShippingMethod usando restricciones adicionales.

Cuando se necesita trabajar con restricciones y reglas de dependencia por defecto entre mas de un concepto o feature, se pueden usar los nombres parametrizados [49]. De esta

manera, se evita la repetición de la expresión con cada concepto o nombre de feature. Un feature o concept parametrizado tiene la forma:

$p_1p_2....p_n$

donde cada $i \in [1..n]$ p_i es o un parámetro o un string específico y existe $j \in [1..n]$ tal que p_j es un parámetro. Por cada parámetro, pueden ser substituidos un conjunto de posibles strings. Los parámetros son introducidos entre <> para distinguirlos de strings específicos. Los nombres parametrizados son la única forma de expresar constraints y reglas de dependencia acerca de subfeatures de un feature abierto, ya que su número es desconocido. Por ejemplo en la figura 2.5 el feature f es un feature con sub-features de la forma f<i>, donde <i> es un número natural. Todos estos features pueden ser referenciados por el feature parametrizado con nombre f<i>.

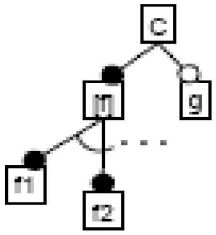


Figura 2.5 Features parametrizados

Los features parametrizados pueden aparecer solamente en FD de conceptos parametrizados. Un FM que contiene un concepto no parametrizado con features parametrizados en su FD sería inconsistente ya que definiría un conjunto de FD diferentes para un concepto simple. Por la misma razón, los conceptos parametrizados pueden no ser referenciados en FD de conceptos específicos (es decir: no parametrizados).

Finalmente, un nodo en un FM puede ser también una referencia a otro FM. Un FM ´X´ con una referencia a otro FM es semánticamente equivalente a una copia de X en la cual la referencia ha sido ´unfolded´, es decir sustituida por un FM al cual apunta la referencia. Las referencias a un FM permiten dividir FMs extensos en módulos menores. La figura 2.6 ilustra esta característica. El ejemplo presentado en la figura 2.4 tiene una referencia a un FM: Catalog.

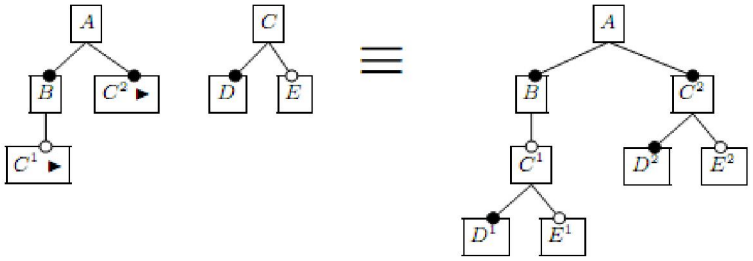


Figura 2.6 Unfolding de referencias a FM

2.2.3 Las restricciones

Los FD definen las restricciones principales sobre las combinaciones permitidas entre features. Dado que los diagramas se representan como árboles, es imposible expresar todas las restricciones sólo por medio de un diagrama. Las restricciones adicionales se pueden expresar mediante una lista asociada con el diagrama. Además, se asocia una lista de reglas de dependencias con cada FD a fin de especificar qué features deberían o no aparecer juntos por defecto.

Las restricciones y las reglas de dependencia por defecto son expresiones lógicas formadas por nombres específicos y parametrizados de conceptos y features, conectivos lógicos, cuantificadores y paréntesis. La intención de usar una lógica de predicados para expresar restricciones y reglas de dependencia por defecto es de evitar las ambigüedades del lenguaje natural.

Los cuantificadores comúnmente usados, el $\forall$ (cuantificador universal) y el $\exists$ (cuantificador existencial) y los conectivos $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implicación), $\veebar$ (xor), $\Leftrightarrow$ (equivalencia) deberían ser suficientes para expresar restricciones y reglas de dependencia por defecto, sin embargo, se pueden usar otros conectivos y cuantificadores si se mejora la claridad de las expresiones.

Una lista de restricciones asociadas con un FD es una conjunción de expresiones de las que consiste el modelo. Para que una instancia de un concepto sea válida, todas las restricciones asociadas con el diagrama de features deben estar completas, es decir, deben ser verdaderas. Las restricciones expresan exclusiones mutuas y requerimientos entre features, es decir, determinan cuales features no pueden aparecer juntos y cuales deben estar juntos respectivamente. Una expresión de restricción simple puede expresar varias exclusiones mutuas y requerimientos a la vez.

Las restricciones más comunes en un modelo son *´requires´* y *´excludes´*. En el modelo, la selección de features variables está restringida por las dependencias de configuración

requires o excludes. Estas restricciones son las denominadas restricciones fuertes del modelo. Ellas deben ser obedecidas ya que un feature depende de otro feature por su definición u operación. En otras palabras, un feature no trabajará sin el otro. Los ejemplos mostrados a continuación son fragmentos extraídos del FD Electronic Shopping que se pueden encontrar [33]. Por otro lado, las restricciones *´is_independent_of´* y *´recommend´* son restricciones que no necesariamente deben ser aplicadas y se denominan las restricciones débiles del modelo.

2.2.3.1 La dependencia *requires*

Un requires entre dos features significa que cuando uno de ellos es seleccionado para una LPS, el otro debe estar presente en la misma línea. Se puede observar en el fragmento del modelo de la figura 2.7 que cada Payment Method de tipo Debit o Credit Card debe tener una entrada específica (Payment Gateway). De esta manera, se dice que la selección de uno de los primeros requiere la selección de Payment Gateway y se expresa mediante una cross-tree constraint como cláusulas CNF de la siguiente manera:

$\vdash$ CreditCard $\vee$ Payment gateway y $\vdash$ DebitCard $\vee$ Payment gateway

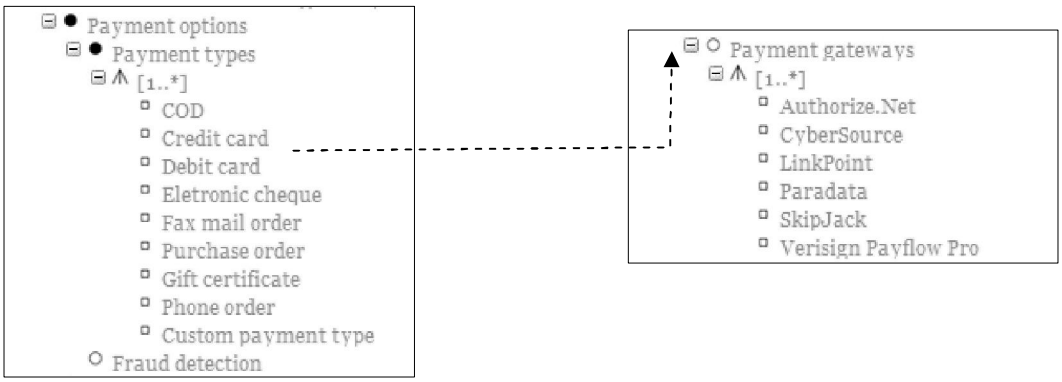


Figura 2.7 Relación requires

2.2.3.2 La dependencia *excludes*

Una dependencia excludes entre dos features significa que ambos features no pueden estar presentes en el mismo producto en el momento de la configuración. A *excludes* B significa que la selección de A implica que B deba ser eliminado. Por ejemplo: en el fragmento del modelo que se observa en la figura 2.8 si el feature Registered to buy (feature A) es seleccionado para su eliminación, entonces, no hay manera de forzar a un cliente a loguearse para hacer un checkout (Registered checkout -feature B-). Por lo

tanto, un checkout no puede ser soportado. Allí es donde se introduce la relación: feature A excludes feature B. La cláusula CNF correspondiente a la relación, es:

$\neg$ Registered checkout $\vee \neg$ Register to buy

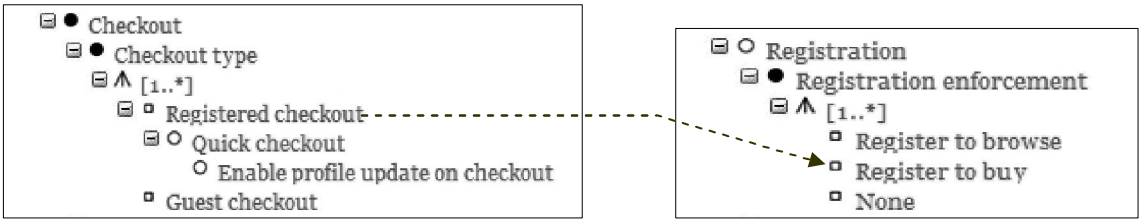


Figura 2.8 Relación excludes

Los aspectos comunes y variables de la información no son suficientes para desarrollar activos reusables y adaptables en una LPS. A pesar que las relaciones estructurales y las dependencias de configuración (requires y excludes) son esenciales en el desarrollo y en la configuración de los activos de la LPS, las dependencias operacionales entre features también tienen implicaciones significantes en el desarrollo de esos activos. Las dependencias operacionales son relaciones creadas implícita o explícitamente entre features durante la operación del sistema de tal manera que la operación de un feature es dependiente de los otros features [29]. Este tipo de análisis no es necesario en el contexto de este trabajo, ya que se abordaría con detalles que no van a tener una principal actuación en el proceso de transformación hacia otro modelo.

Estas relaciones no necesariamente tienen que ser bidireccionales. Lo mas común es que si un feature requiere a otro, la selección de este último recomiende la selección del primero, pero también puede haber independencia en un sentido, e incluso independencia parcial [30].

2.2.3.3 La dependencia *recommends*

Una dependencia recommends entre dos features A y B sucede cuando ocurre lo siguiente: A recommends B, significa si el feature A es seleccionado, entonces B debería también ser seleccionado ya que el único propósito de seleccionar A es para facilitar el uso de B. Por ejemplo: en el fragmento del modelo que se observa en la figura 2.9 siendo el feature A CreditCardInformation y el feature B CreditCard, A recommends B significa que el propósito de guardar la información de la tarjeta de crédito en un perfil sería de utilidad sólo si la operación acepta la tarjeta de crédito como una forma de pago.

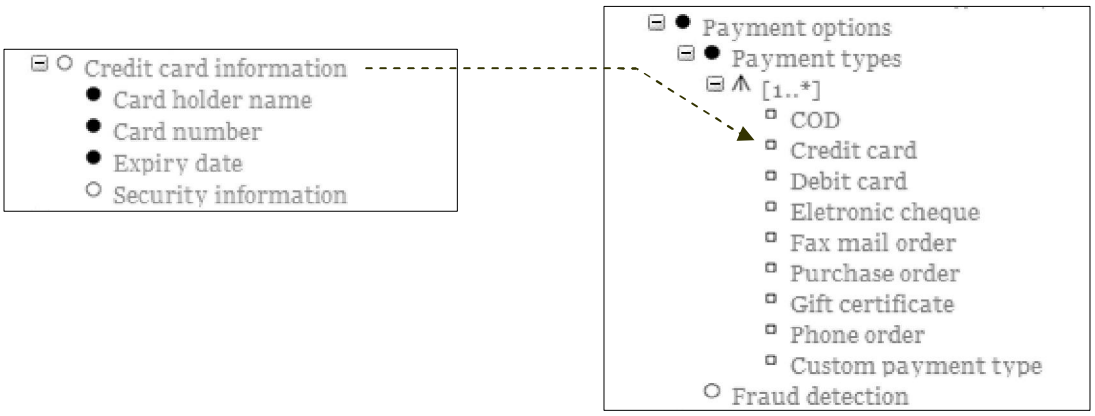


Figura 2.9 Relación recommends

2.2.3.4 La dependencia *is independent of*

La relación toma la forma A is independent of B, estableciendo que la selección de A no tiene implicancia sobre la selección de B. Existen casos en el modelo donde hay una fuerte relación por una parte, pero a su vez una relación de independencia de la otra parte. Significa que uno de los features tiene un alcance mucho más amplio de aplicación que el otro. De este modo, algunos features pueden depender del feature con el alcance más amplio.

Por ejemplo, en el fragmento de modelo que se muestra en la figura 2.10 se observa que si el modelo soporta Discounts (feature B) no tiene implicancia si el e-Shop decide mostrar Special offers (feature A) en la Home page. En este caso la relación es B is independent of A.

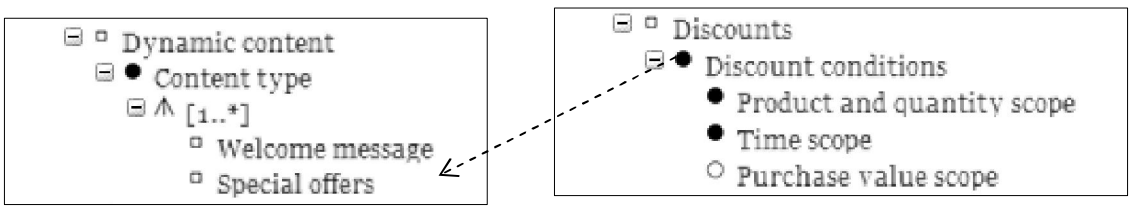


Figura 2.10 Relación is independent of

2.3 La Configuración de un FM

Un FM describe el espacio de configuraciones de una LPS. El usuario luego especifica un miembro de una familia de sistemas mediante la selección de los features deseados del FM dentro de las restricciones de variabilidad definidas por el modelo. Una **configuración** de un FM es el resultado de la selección de un cierto número de elementos del modelo y la eliminación de otros.

La configuración de un FM se corresponde con los requerimientos de un dominio específico de aplicación. Los diseñadores a menudo configuran un modelo tal que sean satisfechos el número máximo de los requisitos de los stakeholders y los requerimientos para una aplicación determinada. Sin embargo, no siempre se garantiza que todas las configuraciones sean válidas, a pesar de las restricciones de integridad que se imponen sobre el modelo.

Por otro lado, el proceso de **especialización** es una transformación que a partir de un FD se obtiene otro FD siendo el conjunto de las configuraciones denotadas por el diagrama último un subconjunto de las configuraciones del primero. Se dice que el último diagrama es una especialización del anterior. Esta relación es comparable a la relación clase-instancia en la programación orientada a objetos. Este proceso está basado en los requerimientos de los stakeholders y el dominio de aplicación destino. En muchos casos, los stakeholders sólo especifican un número limitado de features, sin embargo el resultado esperado es siempre obtener una configuración del FM acorde a los requisitos.

2.4 Herramientas que soportan FM

Actualmente, existen una multitud de herramientas para el modelado de features. La mayoría de estas herramientas fueron desarrolladas por distintas universidades como proyectos de investigación, y coinciden en la creación de modelos de features. Algunas permiten hacer sólo configuraciones gráficamente o validar modelos. Las siguientes son las herramientas que han sido creadas para estos objetivos.

FaMa[13] está desarrollado por un equipo de la Universidad de Sevilla. Permite el análisis automatizado de los modelos de features integrando algunos de los resolutores más comúnmente propuestos (BDD, SAT y CSP), siendo ésta una característica poco común. Dispone de un plugin de Eclipse gráfico desarrollado en EMF bajo licencia Open-Source. La última versión del proyecto es de Marzo del 2012.

FeatureIDE [27] está desarrollado por la University of Magdeburg de Alemania. Es un IDE basado en Eclipse que integra AHEAD, FeatureC++ y herramientas de FeatureHouse como herramientas de composición y compiladores entre otros. Es tanto un editor gráfico como de texto. Permite configuraciones y la creación de restricciones avanzadas. Como falencias, se puede citar que no dispone de features clonables, ni la posibilidad de referencias entre features.

MFM [35] es un método desarrollado en el Centro de Investigación en Métodos de Desarrollo de Software (ProS). Allí se ha desarrollado MOSKitt Feature Modeler (MFM), una herramienta Open-Source basada en Eclipse y desarrollada utilizando EMF, GMF y ATL. MOSKitt soporta edición gráfica de modelos, soporte de persistencia, transformación de modelos, traceability y sincronización, documentación y generación de código usando los modelos como input.

S2T2 [5] es el resultado de la investigación de las líneas de productos software por Lero (centro de investigación de Ingeniería del Software de Irlanda). Es una aplicación de Java Open-Source independiente que permite configuraciones y comprobación de restricciones básicas. No dispone de features clonables, referencias, atributos o restricciones avanzadas.

FMP [2] (Feature Modeling Plug-in) está desarrollado por la Universidad de Waterloo, Canadá, concretamente por K. Czarnecki, un referente para la mayoría de las herramientas citadas. FMP que es Open-Source permite ser utilizado en Eclipse como plug-in realizado en EMF o en el Rational Software Modeler (RSM) o Rational Software Architect (RSA) mediante el plug-in fmp2rsm. Posee numerosas características entre las cuales se destacan la creación de configuraciones, atributos y restricciones avanzadas. Es una de las herramientas más comúnmente utilizadas para el modelado de features.

SPLOT [33] cuyo objetivo principal es poner la investigación de las líneas de productos de software en práctica a través una herramienta on-line enfocada a usos académicos y a maestros del área. Está desarrollada por la Universidad de Waterloo de Canadá. SPLOT es una herramienta web, dispone de una base de datos con gran cantidad de modelos de features base. El usuario puede subir su propio modelo de features que debe de ser escrito en XSML. Además el usuario puede crear una configuración a partir de un modelo que esté en esa base de datos y validarla. Es una herramienta muy simple por lo que no dispone de features clonables, referencias, atributos, restricciones avanzadas o interfaz gráfica.

pure::Variants[4] es una herramienta creada por pure:systems desarrollada como plug-in de Eclipse. Permite perfilar y gestionar eficientemente todas las partes de los productos software con sus componentes, restricciones y términos de uso. No es una herramienta totalmente expresiva en lo que a modelos de features se refiere, puesto que carece de features clonables, referencias o atributos. En cambio tiene otras propiedades como la generación de código y la interoperabilidad con SAP o MatLab.

FMTools [22] tiene las ventajas de una integración directa con el IDE de Visual Studio y la posibilidad de la representación visual y manipulación de los features y las restricciones mutex/require. Puede generar estructuras de paquetes de la Product Line y archivos de configuraciones, directamente. Ha sido desarrollado con el entorno DSL Tools, herramienta de Microsoft integrada en Visual Studio SDK 2008, la cual permite al usuario definir su propio lenguaje específico del dominio.

2.5 El Método FORM

El método FORM (Feature Oriented Reuse Method) consiste de dos principales procesos ingenieriles: la ingeniería del dominio y la ingeniería de la aplicación (figura 2.11).

El proceso de la ingeniería del dominio consiste de actividades para analizar sistemas en el dominio creando arquitecturas de referencia y componentes reusables basadas en los resultados del análisis.

El proceso de la ingeniería de la aplicación consiste de actividades para desarrollar aplicaciones usando los artefactos creados en la ingeniería del dominio. Para aplicaciones típicas en un determinado dominio, la ingeniería de la aplicación debería ser trivial comparada al enfoque de desarrollo de la aplicación tradicional que no tiene una orientación al dominio.

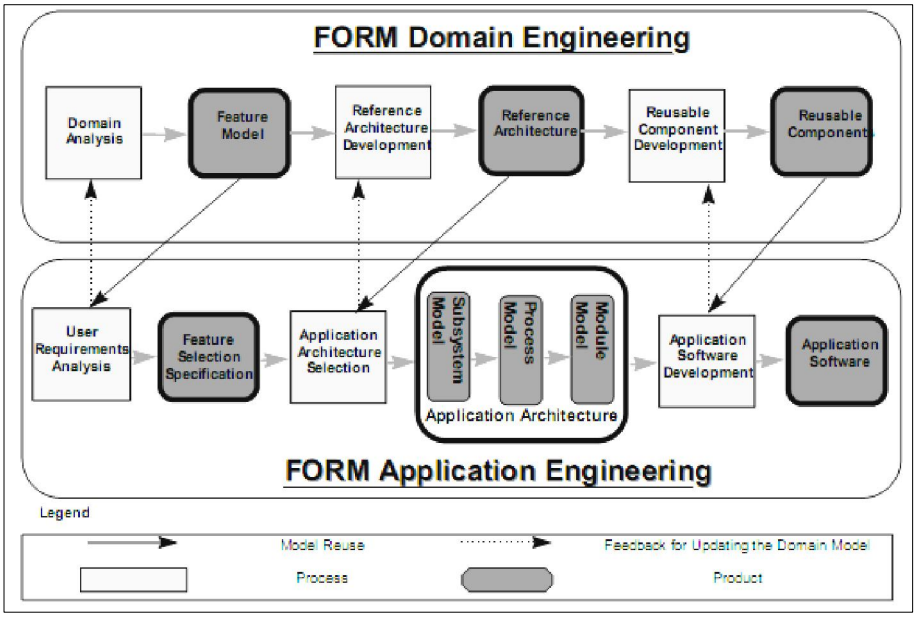


Figura 2.11 Procesos ingenieriles FORM

En este trabajo, el interés principal se centra en el resultado del proceso del Análisis de Dominio y del producto Feature Model, para obtener un análisis de dominio como entrada a un proceso de especificación en el lenguaje RSL. Por esa razón se brindan los conceptos fundamentales que se necesitan para comprender el modelo fuente de Análisis de Dominio y modelo destino de especificaciones RSL, propuestos aquí.

2.5.1 Ingeniería del dominio

En FORM, el conocimiento del dominio se organiza en una manera que intenta ser comprensible; proveyendo a los usuarios features comunes observables y arquitecturas de referencia del dominio objetivo, en el cual los roles de las componentes reusables están explícitamente descriptas. De esta manera, no sólo se incrementa la efectividad del reuso (por ejemplo, a través del uso del FM), sino también la 'adaptabilidad' de la componente reusable se incrementa por el mapping entre la arquitectura de referencia y el modelo de aplicación objetivo.

Hay tres fases en la ingeniería del dominio FORM: el análisis de contexto, el modelo del dominio (o features), y el modelo de la arquitectura. Durante el análisis de contexto se identifica primero, el rango exacto del dominio y el uso de la aplicación del dominio, las condiciones externas y posibles interacciones con el mundo exterior. Luego, durante la fase de modelo del dominio, se identifican los 'features' comprensibles por el usuario representativos del ya definido dominio objetivo y se modelan sus interrelaciones. Se crea un modelo de features y pueden existir varias especificaciones de features consistentes derivables del modelo. Mientras el análisis de contexto y el modelo de dominio definen un 'espacio de features' de selección del usuario, la arquitectura define el 'espacio del artefacto', donde se construyen los artefactos reusables (las componentes de software reusable, sus configuraciones y descomposición jerárquica).

El objetivo de la Ingeniería del dominio es establecer un mapping entre el espacio de decisión (FM) y el espacio del artefacto (modelo de arquitectura). Cada feature en el espacio de decisión restringe de alguna manera la selección del modelo de referencia final y este aspecto es modelado usando dos conceptos: uno por diferenciación entre el efecto de seleccionar features 'funcionales' y 'no-funcionales', y el otro considerando las diferencias en tipos de features siguiendo la jerarquía de cuatro niveles. Los features funcionales son usados principalmente para identificar componentes requeridas, mientras que los no-funcionales son usados para partición de componentes.

Basados en los objetivos antes mencionados del método en estas etapas, se propone en este trabajo un análisis similar para la obtención de una jerarquía de especificaciones reusables RSL a través de mappings. La figura 2.12 resume el objetivo.

Un método efectivo de encontrar el conjunto de features mas adecuado que represente el dominio es, siguiendo los cuatro niveles (layers) de jerarquía de features. Es decir, considerando los niveles propuestos por el método:

§ Capability layer (servicios funcionales y no funcionales): un servicio tiene un role funcional autocontenido que se alcanza por una secuencia de operaciones provistas por aplicaciones.

§ Operating Environment layer (entornos operativos): representan las variaciones del hardware o software que tienen las interfaces.

§ Domain Technologies layer: representan aquellos objetos que encapsulan decisiones de requerimientos. Estas decisiones son las que el analista del dominio hace a fin de desarrollar sus modelos. Representan diferentes formas de desarrollar modelos.

§ Implementation techniques layer: representan aquellos objetos que encapsulan métodos de comunicación, decisiones de diseño, métodos de implementación, etc. Los features de las técnicas de implementación son decisiones de diseño e implementación que deberían ser hechos para implementar objetos específicos del dominio. Esta clase de decisiones deberían estar separadas de los objetos específicos del dominio de modo que puedan ser fácilmente reusados o adaptados así como las estrategias sufren cambios.

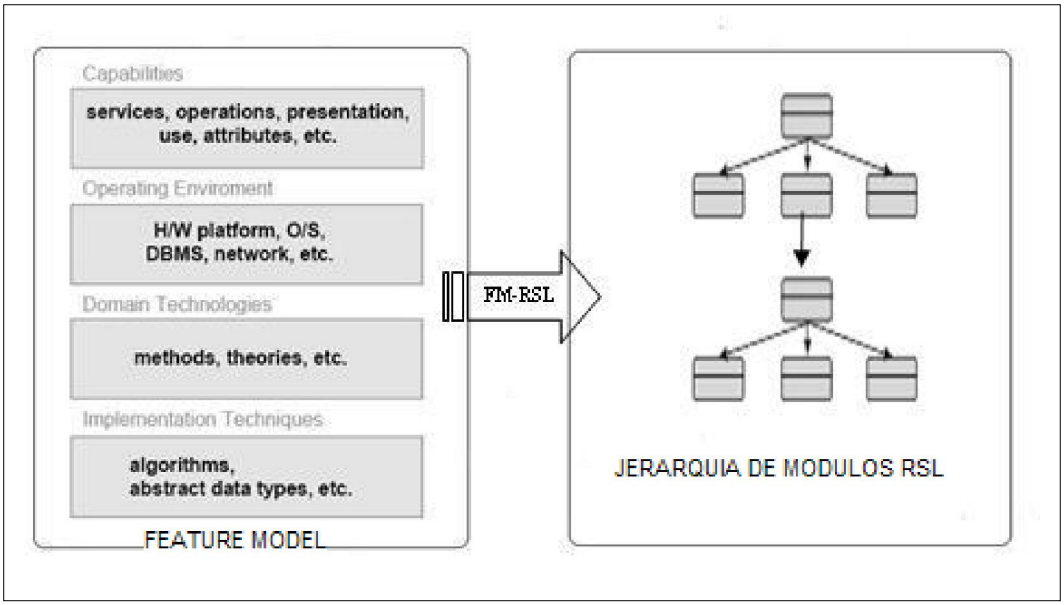


Figura 2.12 Mapping FM a módulos RSL

2.6 El Metamodelo del Feature Model

La definición de metamodelos constituye una parte esencial en los diversos enfoques de Desarrollo de Software Dirigido por Modelos (MDD -Model Driven Development-) [40]. Los conceptos resumidos a continuación son los ampliamente aceptados dentro de este paradigma, aunque la investigación sobre diversas técnicas para definir metamodelos está en continuo desarrollo.

2.6.1 Modelo

MDD asigna a los modelos un rol central: son tan importantes como el código fuente. Se generan desde un alto nivel de abstracción hasta niveles concretos a través de refinamientos y transformaciones, siendo éstas el motor de MDD.

MDD identifica los siguientes tipos de modelos:

- CIMs (Computational Independent Model) son los modelos de alto nivel de abstracción independientes de cualquier metodología computacional,
- PIMs (Platform Independent Model) son los modelos independientes de cualquier tecnología de implementación,
- PSMs (Platform Specific Model), son los modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica,
- Código, son los modelos de la implementación.

Un modelo define qué elementos pueden existir en un sistema o una parte del mismo. Un lenguaje define qué elementos pueden existir en un modelo. Un ejemplo es el lenguaje de modelado UML [45] que define los conceptos “Class”, “Attribute”, “Association”, etc., usados en los modelos estáticos UML. Se puede describir entonces un lenguaje por medio de un modelo, describiendo qué elementos pueden ser usados en el lenguaje.

En nuestro contexto de trabajo, la descripción en forma gráfica de lo que representan cada uno de los elementos que forman parte de un Feature Model se hace dentro de los modelos PIM.

2.6.2 Metamodelo

Un metamodelo es la definición de un lenguaje de modelado en forma precisa y sin ambigüedades. Un metamodelo describe los elementos que pueden ser usados en dicho

lenguaje. Cada elemento que un usuario puede usar en su modelo está definido por el metamodelo del lenguaje. Por ejemplo, en UML se pueden usar clases, atributos, asociaciones, etc., ya que el metamodelo de UML define qué es una clase, qué es un atributo, que son las asociaciones, etc.

En el paradigma de la MDD, la creación de los metamodelos es importante ya que ellos son el lenguaje origen y destino de las reglas de transformación. Estas reglas describen cómo se puede transformar un modelo en un lenguaje origen en otro modelo en un lenguaje destino, y se definen con los elementos de los metamodelos de los lenguajes origen y destino.

También se considera a los metamodelos como modelos gráficos, por esta razón deben ser escritos en un lenguaje bien definido. Estos lenguajes se denominan metalenguajes. De esta manera, se puede encontrar un metamodelo de un metamodelo, lo que se denomina meta-metamodelo y así sucesivamente.

La OMG (Object Management Group) [38] propone estandarizar los conceptos relacionados al modelado desde niveles abstractos a concretos por medio de la arquitectura de cuatro capas, denominados comúnmente con las iniciales M0, M1, M2, M3. Esta arquitectura va a permitir distinguir entre los distintos niveles conceptuales que intervienen en el modelado de un sistema.

- *El nivel M0 - Las instancias*: El nivel M0 modela el sistema real, sus elementos son las instancias que componen dicho sistema.

- *El nivel M1 – El Modelo del sistema:* es una instancia de un metamodelo. Esta capa define un lenguaje que describe los dominios semánticos. Este modelo contiene los elementos del modelo y las instancias de esos elementos. Existe una relación muy estrecha entre los niveles M0 y M1: los conceptos del nivel M1 definen las clasificaciones de los elementos del nivel M0, mientras que los elementos del nivel M0 son las instancias de los elementos del nivel M1.

- *El nivel M2 – El metamodelo* (modelo del modelo): es una instancia de un meta metamodelo donde cada elemento del metamodelo es una instancia de un elemento del meta metamodelo. El metamodelo define un lenguaje para especificar modelos. UML (Unified Modelling Language) y OCL [36] (Object Constraint Language) son ejemplos de metamodelos. Aquí también existe una gran relación entre los conceptos de los niveles M1 y M2: los elementos del nivel superior definen las clases de elementos válidos en un determinado modelo de nivel M1, mientras que los elementos del nivel M1 pueden ser considerados como instancias de los elementos del nivel M2.

- *El nivel M3 – El modelo de M2 (el meta-metamodelo*): la responsabilidad primaria de la capa de meta metamodelo es definir el lenguaje para especificar un metamodelo.

OMG ha definido un lenguaje para describir los elementos del nivel M3, que se denomina MOF (Meta-Object Facility) [34]. MOF puede considerarse como un lenguaje para describir lenguajes de modelado, como pueden ser UML, CWM (Common Warehouse Metamodel), o incluso el propio MOF. Tales lenguajes pueden ser considerados como instancias del MOF, ya que MOF proporciona el lenguaje con los constructores y mecanismos mínimos necesarios para describir meta-modelos de lenguajes de modelado (es decir, los elementos que van a constituir un lenguaje dado, y las relaciones entre tales elementos).

El metamodelo MOF está implementado mediante un plugin para Eclipse llamado Ecore. Este plugin respeta las metaclases definidas por MOF. Ellas mantienen el nombre del elemento que implementan y agregan como prefijo la letra “E”, indicando que pertenecen al metamodelo Ecore.

Ecore es, en sí mismo un metamodelo de EMF (Eclipse Modelling Framework) [3]. EMF es un marco de trabajo de Eclipse que unifica Java, XML y UML, permitiendo a los desarrolladores construir rápidamente aplicaciones robustas basadas en modelos simples. De este modo, Ecore es usado para representar modelos en EMF.

Es importante destacar que si no se provee un estándar para describir los metamodelos, las transformaciones que anteriormente se nombraron, no podrían ser definidas adecuadamente.

En el marco de este trabajo, el metamodelo del FM propuesto, está inspirado inicialmente por el metamodelo del plugin fmp de Czarnecki [2]. Luego de un análisis del mismo y con el objetivo de dar una transformación desde los FM a especificaciones RSL, se presenta el metamodelo que describe los FM con características específicas del método FORM. El metamodelo se creó en Ecore de acuerdo con las prácticas convencionales dentro de la ingeniería de lenguajes de modelado.

La figura 2.13 describe el metamodelo con los elementos que son fundamentales para establecer el mapping definido en el capítulo 4 de esta tesis.

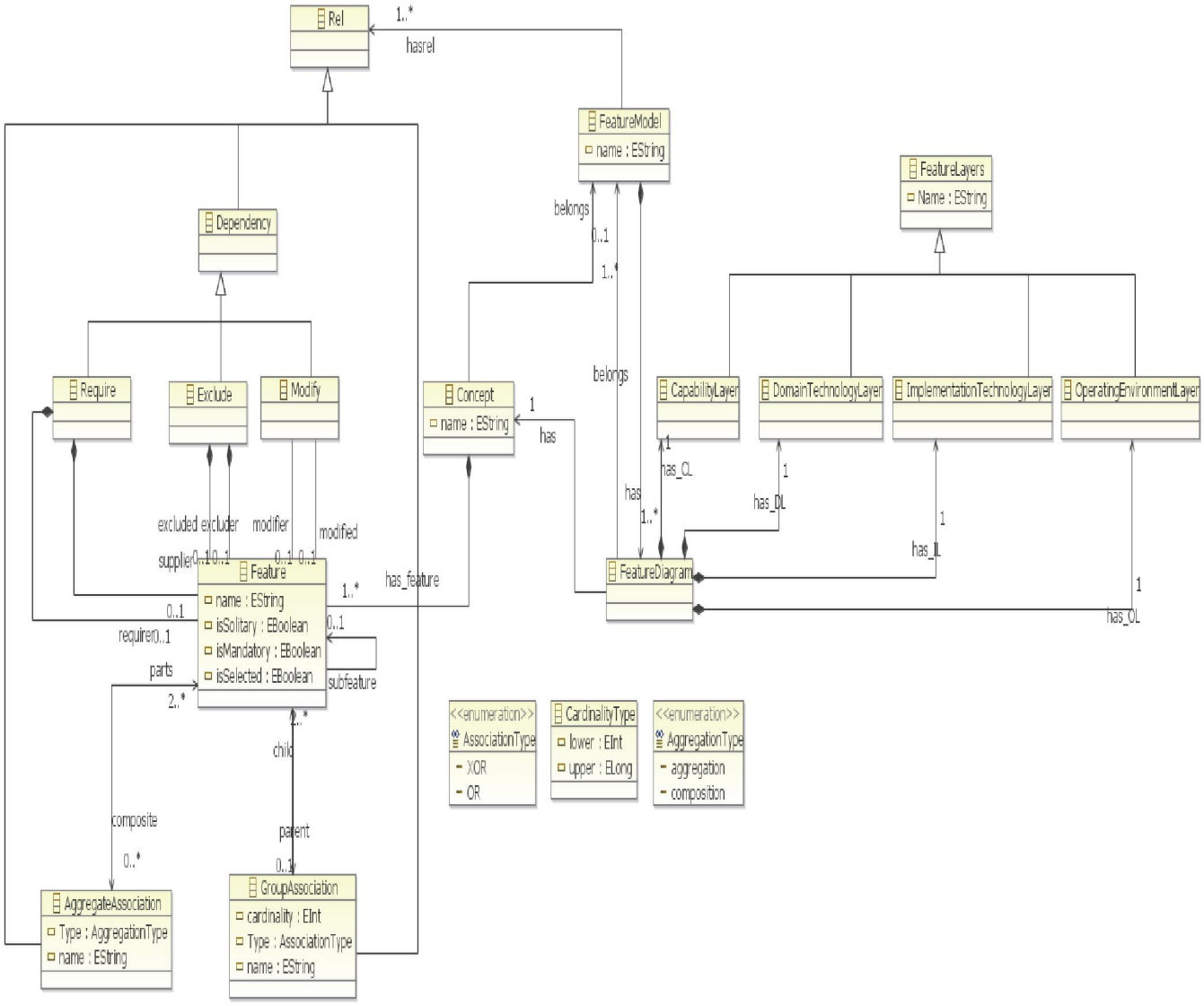


Figura 2.13 Metamodelo método FORM

2.6.3 Definición de los elementos del metamodelo

A continuación se definen las clases del metamodelo.

- **Feature**

Super Class: Feature

Descripción: Una propiedad de un sistema que es relevante a algunos stakeholders usada para capturar aspectos comunes o diferenciar entre sistemas de una misma familia.

Propiedades:

- name: EString. Especifica el nombre del feature.
- isSolitary: Boolean. Especifica si el feature es Solitary
- isMandatory: Boolean. Especifica si el feature es Mandatory.
- isSelected: Boolean. Especifica si el feature ha sido seleccionado en la configuración.

Asociaciones:

- composite: AggregateAssociation [*]. Referencia a la agregación de la cual puede formar parte un feature.
- subfeature: Feature [0..*]. Referencia a los features que pueden ser subfeatures de él.
- parent: GroupAssociation [0..*]. Referencia al feature que puede ser padre de un grupo XOR u OR.
- isRequired: Require [0..1]. Referencia a la relación que vincula al feature que requiere la relación.
- isSupplied: Require [0..1]. Referencia a la relación que vincula al feature que requiere la relación.
- excludes: Exclude [0..1]. Referencia a la relación que vincula al feature que requiere la relación.
- isExcluded: Exclude[0..1]. Referencia a la relación que vincula al feature que requiere la relación.
- belongs_to: Concept [0..1]. Referencia a la relación que vincula al feature que requiere la relación.
- isModified: Modify [0..1]. Referencia a la relación que vincula al feature que requiere la relación.
- modifies: Modify [0..1]. Referencia a la relación que vincula al feature que requiere la relación.

Restricciones: no posee restricciones adicionales.

- ## Concept

Super Class: no posee.

Descripción: El concept es un punto de vista de una clase o categoría de elementos en un dominio. Los elementos individuales que corresponden a estos puntos de vista se llaman instancias del concepto. Se puede pensar como un feature con características especiales, sin embargo no se aplicarían las propiedades descriptas para un feature cualquiera de un FD.

Propiedades:

- name**:** EString. Especifica el nombre que tiene el concept que se asociará al sistema mismo.

Asociaciones:

- has_feature: Feature [*].Especifica el conjunto de features de los cuales un concept está compuesto.
- belongs: FeatureModel [0..1]. Especifica el modelo al cual pertenece el concept.

Restricciones: no posee restricciones adicionales.

- ## FeatureDiagram

Super Class: no posee

Descripción: La clase FeatureDiagram representa el árbol dirigido cuya raíz es el concept y el resto de los nodos representan los features.

Propiedades: no posee propiedades adicionales

Asociaciones:

- belongs: FeatureModel [1..*]. Especifica que un FeatureDiagram pertenece a uno o más FeatureModel.
- has_CL: CapabilityLayer [1]. Especifica que un FeatureDiagram tiene asociado un CapabilityLayer.
- has_DL: DomainTechnologyLayer [1]. Especifica que un FeatureDiagram tiene asociado un DomainTechnologyLayer.
- has_IL: ImplementationTechnologyLayer [1]. Especifica que un FeatureDiagram tiene asociado un ImplementationTechnologyLayer.
- has_OL: OperationEnvironmentLayer [1]. Especifica que un FeatureDiagram tiene asociado un OperationEnvironmentLayer.

- has: Concept [1]. Referencia al concept que posee un diagrama.

Restricciones: no posee restricciones adicionales.

- ## Rel

Super Class: no posee

Descripción: La clase Relationship representa la abstracción de los distintos tipos de relaciones que existen entre features: la agregación, el agrupamiento y las dependencias.

Propiedades: no posee propiedades adicionales

Asociaciones: no posee asociaciones adicionales

Restricciones: no posee restricciones adicionales.

- ## FeatureModel

Super Class: no posee

Descripción: La clase FeatureModel representa al modelo del sistema que puede estar representado por uno o más diagramas.

Propiedades:
- name**:** EString. Especifica el nombre del modelo, generalmente asociado al dominio que modela.

Asociaciones:
- has: FeatureDiagram [1..*]. Especifica que un FeatureModel tiene uno o más FeatureDiagram.
- hasrel: Relationship [1..*]. Especifica que un FeatureModel posee una o mas relaciones entre sus features.

Restricciones: no posee restricciones adicionales.

- ## FeatureLayers

Super Class: no posee

Descripción: La clase FeatureLayers representa los cuatro layers en que se divide un diagrama según el método FORM. Es una clase abstracta.

Propiedades:
- name**:** EString. Especifica el nombre del layer.

Asociaciones: no posee asociaciones adicionales.

Restricciones: no posee restricciones adicionales.

- ## CapabilityLayer

Super Class: FeatureLayer

Descripción: La clase CapabilityLayer representa uno de los cuatro layers en que se divide un diagrama según el método FORM. Esta clase representa los features agrupados en el layer que corresponden a servicios funcionales y no funcionales.

Propiedades: no posee propiedades adicionales.

Asociaciones: no posee asociaciones adicionales.

Restricciones: no posee restricciones adicionales.

- ## DomainTechnologyLayer

Super Class: FeatureLayer

Descripción: La clase DomainTechnologyLayer representa uno de los cuatro layers en que se divide un diagrama según el método FORM. Esta clase representa los features que encapsulan decisiones de requerimientos que hacen los analistas del dominio.

Propiedades: no posee propiedades adicionales.

Asociaciones: no posee asociaciones adicionales.

Restricciones: no posee restricciones adicionales.

- ## ImplementationTechnologyLayer

Super Class: FeatureLayer

Descripción: La clase ImplementationTechnologyLayer representa uno de los cuatro layers en que se divide un diagrama según el método FORM. Esta clase representa aquellos objetos que encapsulan métodos de comunicación, decisiones de diseño, métodos de implementación.

Propiedades: no posee propiedades adicionales.

Asociaciones: no posee asociaciones adicionales.

Restricciones: no posee restricciones adicionales.

- ## OperatingEnvironmentLayer

Super Class: FeatureLayer

Descripción: La clase OperatingEnvironmentLayer representa uno de los cuatro layers en que se divide un diagrama según el método FORM. Esta clase agrupa a aquellos features que representan las variaciones del hardware o software que tienen las interfaces.

Propiedades: no posee propiedades adicionales.

Asociaciones: no posee asociaciones adicionales.

Restricciones: no posee restricciones adicionales.

- ## Dependency

Super Class: Rel

Descripción: La clase Dependency representa la abstracción de las restricciones fuertes (require, exclude y modify) entre features de un FM.

Propiedades: no posee propiedades adicionales.

Asociaciones: no posee asociaciones adicionales.

Restricciones: no posee restricciones adicionales.

- ## Require

Super Class: Dependency

Descripción: La clase Require representa la relación entre un feature supplier y su requester. El feature requester necesita la presencia del feature supplier en el mismo producto para su correcta operación.

Propiedades: no posee propiedades adicionales.

Asociaciones:
- supplier: Feature [0..1]. Referencia al feature que será el supplier de la relación.
- requirer: Feature [0..1]. Referencia al feature que será el que *requiere* de la relación

Restricciones: no posee restricciones adicionales.

- ## Exclude

Super Class: Dependency

Descripción: La clase Exclude representa la relación entre un feature supplier y su requester. Esta relación declara la incompatibilidad de dos features: un feature que excluye a otro hace entrar en conflicto la selección del segundo en el mismo producto (proceso de configuración).

Propiedades: no posee propiedades adicionales

Asociaciones:
- excluder: Feature [0..1]. Referencia al feature que será el excluder de la relación.
- excluded: Feature [0..1] Referencia al feature que será el excluído de la relación.

Restricciones: no posee restricciones adicionales.

- ## Modify

Super Class: Dependency

Descripción: La clase Modify representa la relación entre un feature modifier y su modified. Esta relación declara la relación de dos features: un feature que modifica a otro al ser seleccionado en el mismo producto (proceso de configuración).

Propiedades: no posee propiedades adicionales

Asociaciones:

- modifier: Feature [0..1]. Referencia al feature que será el modificador de la relación.
- modified: Feature [0..1]. Referencia al feature que será el modificado de la relación.

Restricciones: no posee restricciones adicionales.

- ## AggregateAssociation

Super Class: Rel
Descripción: La clase AggregateAssociation representa las formas de agrupamiento. Puede expresar tanto una relación *composition* como una relación *aggregate*.

Propiedades:

- Type**:** AggregationType. Especifica los dos tipos de relación que puede tomar un AggregationType.

Asociaciones:

- parts: Feature [2..*]. Referencia a las partes (features) que reúne un agreggate, pueden ser dos o mas.

Restricciones: no posee restricciones adicionales

- ## GroupAssociation

Super Class: Rel

Descripción: La clase GroupAssociation representa las formas de agrupamiento. Este tipo de agrupamiento es considerado como una relación '*is-a*' de features que pueden ser alternative, mandatory u optional, con un feature padre.

Propiedades:

- Type**:** AssociationType. Especifica los dos tipos de relación que puede tomar un AggregationType.
- Cardinality: CardinalityType. Especifica los valores límites que puede tomar el atributo cardinality.
- Name: EString. Especifica el nombre del grupo.

Asociaciones:

- child: Feature [2..*]. Referencia al grupo que asocia a dos o más features.

Restricciones: no posee restricciones adicionales

§ AssociationType

Super Class: no posee

Descripción: La clase AssociationType es una enumeración de los siguientes valores:

- XOR**:** especifica una relación OR exclusivo.
- OR: especifica una relación OR.

Propiedades: posee restricciones adicionales

Asociaciones: no posee

Restricciones: no posee restricciones adicionales

§ AggregationType

Super Class: no posee

Descripción: La clase AggregationType es una enumeración de los siguientes valores:

- aggregation**:** especifica la relación todo/parte.
- composition: este valor especifica la relación entre dos features donde se debe manejar la creación y destrucción de sus partes.

Propiedades: posee restricciones adicionales.

Asociaciones: no posee

Restricciones: no posee restricciones adicionales

§ CardinalityType

Super Class: no posee

Descripción: La clase CardinalityType es una enumeración de representa los valores que puede tomar los límites superiores e inferiores de la cardinalidad de un grupo.

- § lower**:** Int, especifica el valor entero mas bajo de la cardinalidad del grupo.
- § upper: de tipo unlimitedInt, especifica el valor entero más alto de la cardinalidad del grupo.

Propiedades: posee restricciones adicionales.

Asociaciones: no posee

Restricciones: no posee restricciones adicionales

Capítulo 3

RAISE

RAISE (Rigorous Approach to Industrial Software Engineering), fue originalmente el nombre de un proyecto ESPRIT diseñado para desarrollos de amplio espectro. Incluye un lenguaje de especificación y diseño denominado RAISE Specification Language (RSL), un método asociado y estrategias para el desarrollo formal. Brinda además un conjunto de herramientas para ayudar a la edición, verificación, impresión, almacenamiento y el razonamiento acerca de especificaciones formales. Una completa descripción del método y del lenguaje se pueden encontrar en los textos bibliográficos [19] y [18] como así también de las herramientas [20], y en el sitio web de la UNU/IIST (www.iist.unu.edu). En este capítulo se describe el método y sus principios fundamentales, como así también el lenguaje y los elementos necesarios para las especificaciones iniciales propuestas en este trabajo. Además se presenta un metamodelo que describe los elementos necesarios del lenguaje RSL para establecer el mapping definido en el capítulo 4 de esta tesis.

3.1 El Método

El método RAISE tiene su base en una serie de principios, ellos son:

- *Desarrollo separado*

El desarrollo de sistemas de software a gran escala es complejo. Por esa razón, es necesario descomponer su descripción en componentes y así construir el sistema a partir de las componentes desarrolladas. También es claro que para la mayoría de los sistemas, es necesario tener personas o grupos de personas trabajando sobre diferentes componentes al mismo tiempo. Habrá entonces entidades –archivos, documentos, etc- que serán compartidos, generando en consecuencia dificultades. Una de ellas es que debe estar claro quién es el responsable por actualizaciones tales como entidades compartidas, y cual es el estado de cada versión en un determinado tiempo. Esto es un problema de control de configuración standard y no específico de los métodos formales.

Otra dificultad que se presenta es que no debería haber ambigüedades acerca del significado de las entidades. Se necesitan entonces, sentencias no-ambigüas que actúen como un contrato entre el ingeniero de software y el usuario. Para el ingeniero, un contrato dice lo que él debe proveer; y para los usuarios dice lo que ellos pueden suponer. Una especificación de un módulo (o un grupo de módulos) puede actuar como un contrato precisando cuales son las propiedades esenciales de lo especificado. Es mucho mejor que algo escrito en un lenguaje de programación, ya que este puede establecer las propiedades esenciales e ignorar las irrelevancias.

El objetivo es poder alcanzar los requerimientos originales. Para ello, se requiere un conjunto de condiciones suficientes:

- Que los módulos iniciales alcancen los requerimientos, es decir tener todas las propiedades requeridas.
- Cada paso desarrollado de los módulos es un 'paso de implementación', es decir cada módulo implementa el inmediatamente precedente.

Desde el punto de vista de la reusabilidad y a fin de obtener una especificación inicial del sistema, se deberá componer las distintas especificaciones de los módulos obtenidos. El siguiente paso es establecer una taxonomía de subespecificaciones adecuadas, a fin de identificar especificaciones básicas en la biblioteca de componentes reusables.

- *Desarrollo paso a paso*

Desarrollar software en una secuencia de pasos es también una técnica importante. Se puede comenzar con una abstracción adecuada, decidir cuales son las decisiones de diseño principales que se necesitan hacer y las dependencias que hay entre ellas, y hacer un plan de orden. Típicas decisiones de diseño involucran:

- proveer definiciones explícitas para valores, dando sólo signaturas o definiciones implícitas o axiomas,
- proveer definiciones explícitas para variables y canales previamente referidos por *any*.
- dar definiciones concretas para tipos abstractos.
- cambiar las definiciones para tipos para permitir funciones mas detalladas o (potencialmente) mas eficientes sobre ellas (ej: listas para conjuntos)
- agregar nuevas definiciones o axiomas.
- agregar variables de estado, tanto localmente para salvar valores como globalmente para reemplazar parámetros.
- cambiar el estilo de especificación entre aplicativo e imperativo o entre secuencial y concurrente.

- agregar parámetros extras o canales para expresar mayor funcionalidad.
- hacer cosas mas generales (y por lo tanto mas reusables) – tales como agregar parámetros a esquemas o extender los tipos parámetros de funciones.
- remover constructores dificultosos de traducir a un lenguaje target seleccionado.

Tratar con una o más de estas decisiones significa hacer un paso de desarrollo, se genera una nueva especificación la cual podemos verificar conforme a la previa. Es importante realizar al menos una decisión de diseño en cada paso de desarrollo, para tratar con un problema por vez. Decidir cuales son las mejores decisiones de diseño también dá un plan general de actividades para el desarrollo.

En una especificación RAISE basada en este principio, donde cada paso representa un refinamiento del previo, es posible poder seleccionar componentes reusables en cada uno de esos pasos. La relación entre una especificación y la siguiente puede ser formalmente verificada, aplicando un tipo de refinamiento conocido como la teoría de la extensión.

- *Invent and verify*

Hay técnicas para el desarrollo de software que se basan en transformaciones. Al trabajar con este tipo de técnica, el ingeniero comienza con una expresión y aplica una regla de transformación que crea una expresión diferente pero equivalente. Así, el ingeniero se garantiza a priori que la nueva expresión es equivalente con la vieja porque las reglas de transformación preservan la equivalencia. De hecho, hacer justificación en RAISE es un ejemplo de una técnica transformacional, pero aplicada a expresiones lógicas en vez de expresiones de programas. Cualquier aplicación formal de reglas de prueba correcta produce un argumento correcto (no hay necesidad de verificarlo luego). “Invent and verify”, por otro lado, es un estilo que permite al ingeniero inventar un nuevo diseño. Luego, el ingeniero verifica su correctitud. ¿Cuáles son las ventajas de invent and verify?:

- Los sistemas transformacionales son muy largos para lenguajes acotados. El editor de justificación RSL actualmente tiene 2000 reglas. Es un pequeño sistema diseñado con un propósito muy particular de hacer justificaciones: la mayoría de las reglas son equivalencias. Un sistema transformacional debería ser considerablemente mas grande porque debería tener un propósito mas general y no estaría restringido a equivalencias.
- El enfoque invent and verify permite inventar varios pasos, quizá cambiando la forma de pensar algunas veces, antes de decidir sobre el correcto enfoque y

hacer las verificaciones. Con transformaciones el enfoque tiende a ser más formal desde el comienzo.

- En la práctica, algunos pasos de desarrollo no serán implementaciones y pueden, realmente producir una 'pequeña' relación formal con el nivel previo. Es bueno preservar el nivel previo como una guía para ver cómo se alcanza un nuevo diseño.

Si una especificación está basada en este paradigma de refinamiento, también se deberá verificar la correctitud de cada nueva especificación. Lograr la solidez en una especificación es esencial. En particular, la "relación de implementación" RAISE se debe conservar estrictamente.

La mayor parte del método consiste de técnicas para los cuatro procedimientos principales:

§ **Especificación**: comienza con la definición de los requisitos especificados en lenguaje natural, y produce una descripción en RSL que se denomina especificación inicial. Esta especificación define el comportamiento del sistema con el nivel de detalle necesario para abarcar todos los requisitos funcionales importantes. Sin embargo, sólo debe definir "qué" es el sistema en vez de "cómo" lo hace.

§ **Desarrollo**: partiendo de la especificación inicial, produce una especificación RSL nueva y más detallada, generalmente en un número de pasos de desarrollo. Esta nueva especificación, llamada especificación final, satisface a la original y está lista para ser traducida a un lenguaje de programación.

§ **Justificación**: una prueba total o parcialmente informal se denomina justificación. El desarrollo es difícil pero las pruebas lo son mucho más. No es práctico probar todo y por lo tanto se deben elegir sólo las propiedades que realmente necesiten una investigación más profunda.

§ **Traducción**: partiendo de la especificación final en RSL, produce un programa o conjunto de programas en algún lenguaje ejecutable.

Existen además dos aspectos del método a tener en cuenta:

§ qué es lo que se produce, es decir, los artefactos particulares,
§ el orden en que estos artefactos se producen.

Es fácil confundir estos aspectos, asumiendo que las dependencias entre cosas producidas implican que esas cosas se deben producir en un orden particular. Es cierto que algunas dependencias determinan un orden, pero en general existe una cierta flexibilidad que es frecuentemente muy útil.

En cuanto a lo que se produce, el objetivo general del método es desarrollar una secuencia de especificaciones de un sistema, donde una especificación es una colección de módulos RSL y su traducción en un lenguaje de programación. La primera especificación (especificación inicial) es generalmente abstracta aplicativa, es decir los módulos contienen tipos abstractos de datos, signaturas y axiomas, en lugar de definiciones explícitas, para algunas o todas las funciones.

En relación al orden de producción, se pueden identificar tres etapas principales:

- Análisis: produce la especificación inicial
- Diseño: produce la especificación final
- Traducción: produce el programa ejecutable

La etapa de Análisis produce la especificación inicial; el Diseño, la especificación final y la Traducción, el programa ejecutable. Si bien existe un orden entre estas etapas, ya que cada una es necesaria como entrada para la siguiente, en la práctica el proceso es iterativo. Esto se debe a que generalmente los requisitos no están claros o bien entendidos, y además no hay seguridad de cuál será el mejor diseño.

El Análisis involucra los aspectos siempre presentes en el desarrollo de software: determinar cuáles son los objetos, sus atributos y cuáles son las relaciones entre esos objetos. Es suficiente comenzar respondiendo estas preguntas y desarrollar la primera especificación para capturar las respuestas. Generalmente, se bosquejan las partes que parecen obvias y se pone especial atención en aquellas que se consideran posibles dificultades. Esta primera especificación está lejos, en general, de ser la especificación inicial ya que suele ser demasiado concreta. En este punto, hay dos opciones principales: completar la primera especificación y considerarla la inicial, o formular una abstracción de la primera especificación para formar la especificación inicial.

En síntesis, el método RAISE provee un *framework* para el proceso de desarrollo de software, permitiendo que las características de cada proyecto determinen las técnicas a aplicar y el nivel de rigor. De esta manera, los mismos usuarios pueden elegir el nivel de formalidad que es apropiado en cada proyecto.

3.2 El Lenguaje

El RAISE Specification Language (RSL) es un lenguaje de “amplio espectro” para especificar y diseñar sistemas de software usado en el método RAISE. El término amplio espectro se debe a que RSL permite estilos de especificación orientados a propiedades abstractas y estilos orientados a algoritmos concretos. Esto es, se puede escribir una especificación abstracta adecuada para ser leída por los usuarios y luego en

uno o más pasos se puede refinar en una especificación RSL concreta adecuada para ser traducida en algún lenguaje de programación. RSL permite que especificaciones y diseños de sistemas grandes sean modularizados y entonces permitir que los subsistemas se puedan desarrollar separadamente. También permite, que se pueda expresar los diseños operacionales a un nivel de detalle desde el cual el paso a código final sea simple. Esto significa que la mayor parte de la construcción de un sistema, desde la especificación al diseño, pueda ser hecha usando el mismo formalismo, teniendo entonces argumentos precisos y matemáticos para la correctitud de los pasos de desarrollo y de otras propiedades de carácter crítico.

En las secciones siguientes, se describen brevemente los conceptos básicos del lenguaje, necesarios para abordar el proceso de transformación descripto en el Capítulo 4. Las definiciones detalladas de estos conceptos se pueden encontrar en [18] y [21]. Los ejemplos utilizados para mostrar construcciones del lenguaje fueron extraídos de la especificación de University Library [37].

3.2.1 Expresiones básicas

Una especificación en RSL es una colección de módulos. Un módulo es básicamente una colección de declaraciones nombradas y puede ser tanto un *scheme* como un *object*. Una expresión de clase básica es una colección de declaraciones encerradas entre las palabras clave **class** y **end** y representa una clase de modelos. Cada declaración es una palabra clave seguida de uno o más definiciones apropiadas, detalladas a continuación:

- **object**: módulos embebidos
- **type**: tipos
- **value**: constantes y funciones
- **variable**: pueden almacenar valores
- **channel**: canales para input y output
- **axiom**: propiedades lógicas que deben valer siempre
- **test_case**: expresiones a ser evaluadas por un traductor o intérprete.

El anterior es el orden más común de uso de las declaraciones, aunque muchas clases sólo contienen declaración de tipos y valores.

3.2.2 Declaraciones de tipos

RSL es un lenguaje tipado (cada ocurrencia de un identificador representando un value, variable o channel debe estar asociado con un único tipo). Un tipo es una colección de valores lógicamente relacionados. Puede ser predefinido o definido por el usuario.

Puede estar definido siendo igual a algún otro tipo, o usando una expresión de tipo formada por otros tipos. Una declaración de tipo tiene la forma

type
  definición_tipo$_1$,
  ...
  definición_tipo$_n$,

para n ≥1

Una definición de tipo puede definir un tipo abstracto, es decir un tipo sin operadores predefinidos para generar y manipular sus valores, excepto por = que compara dos valores del tipo para verificar si son iguales. Un tipo abstracto se llama también sort. Todo tipo tiene asociados los operadores = y ≠. Otra forma de definir un tipo es

id = expr_tipo,

Un tipo que se obtiene aplicando el operador de tipo a uno o más tipos se denomina tipo compuesto. Ejemplo:

type
  Book_id,
  Copy_id,
  Book_key = Book_id x Copy_id

Define Book_id y Copy_id como tipos abstractos y Book_key como un tipo concreto. RSL tiene siete tipos built-in: Bool, Int, Nat, Real, Char, Text, Unit con sus correspondientes operadores y formas de construir tipos a partir de otros tipos (type constructors, record types, variante types, union types y subtypes).
Los constructores de tipos permiten la definición de tipos compuestos: products (x), functions ($\rightarrow$ para funciones y $\sim\rightarrow$ para partial functions), sets (**-set** para conjuntos finitos, **-infset** para conjuntos infinitos), lists (* para listas finitas y ^{w} para listas infinitas), y maps ($\rightarrow_m$ para maps finitos y $\sim\rightarrow_m$ para maps infinitos). Sets, lists y maps definen colecciones de valores del mismo tipo. Un set es una colección de valores distintos, mientras una lista es una secuencia de valores, posiblemente incluyendo valores duplicados. Un map es una estructura tipo tabla que mapea valores de un tipo en valores de otro tipo. Por ejemplo, la definición siguiente modela la colección de todos los libros de la biblioteca utilizando un type map.

type
  Book_id,
  Book,
  Books = Book_id $\rightarrow_m$ Book

Los records son muy similares a los records de los lenguajes de programación. Este ejemplo define el type Borrower como un record con tres componentes:

```
type
  Borrower_detail,
  Borrower_level == academic | non_academic | student,
  Br_copies,
  Borrower ::
        borr_detail: Borrower_detail ↔ chg_borr_detail
        level: Borrower_level ↔ chg_level
        copies: Br_copies ↔ chg_copies
```

Cada componente tiene un identificador, un destructor, y una expresión de tipo. Opcionalmente, un record puede tener un reconstructor. También provee una función constructora implícita para crear un value desde sus values componentes. Para el ejemplo presentado, se tiene el constructor mk_Borrower, el cual es creado colocando mk_ como prefijo del correspondiente identificador del tipo.

Las destructoras son funciones totales desde el record a sus expresiones de tipos componentes. Por ejemplo, se puede aplicar level a un valor del tipo Borrower para justamente, conseguir su level, y luego un valor borrower br, se escribe level(br).

Los reconstructores son funciones totales que toman su expresión de tipo componente y un record para generar un nuevo record. Si se escribe chg_level (student, br) se consigue un nuevo valor borrower con el mismo borr_detail y copies, pero con la componente level de student.

Los tipos variant permiten la definición de tipos con una elección de valores, posiblemente con diferentes estructuras. El tipo Borrower_level es un ejemplo de una definición de tipo variant.
La definición de un tipo Union permite hacer nuevos tipos a partir de otros que ya existen. Si B y C son tipos definidos en algún lugar, entonces se puede definir el tipo A como su Union:

```
type
  A = B | C
```

Los subtipos son tipos que contienen algunos de los valores de otro tipo, los que satisfacen un predicado. Por ejemplo, se puede definir el tipo Student como un tipo que contiene valores que satisfacen el predicado is_student.

type
Student = {| br: Borrower • is_student (br) | }

3.2.3 Values

Los values son constantes y funciones, y pueden ser implícitos o explícitamente definidos. En ambos casos, la definición debe incluir al menos la signatura, que es un nombre y tipos para el resultado, y para los argumentos, en caso de una función. Una declaración de values consiste de la palabra clave **value** seguida por uno o más definiciones de values separadas por comas. Por ejemplo, para especificar el máximo número de copias que un ´borrower´ puede estar leyendo en el ´reading room´, se debe dar la siguiente definición implícita:

value
reading_limit: Nat • ≤ 3

Sin embargo, si se conoce el valor exacto de la constante, se puede usar la definición explícita:

value
reading_limit: Nat = 3

Una función es un mapping desde values de un tipo a values de otro tipo, y puede ser total o parcial. Es total cuando está definida para cada valor de los argumentos, de otro modo, es considerada parcial. Las funciones también se clasifican como generadoras, cuando el tipo de interés aparece directa o indirectamente en el tipo resultado, u observadoras cuando no lo hace. Por ejemplo,

value
can_remove_item: Book_id x Books → Bool
can_remove_item (bi, bs) ≡ exist_id (bi,bs) ∧ B.has_copy (bs(bi)),

remove_item: Book_id x Books ~→ Books
remove_item (bi, bs) ≡ bs \ {bi}
pre can_remove_item (bi, bs)

la función remove_item es una función generadora parcial, mientras que can_remove_item es una función observadora total.

3.2.4 Axioms

Una definición de axioma es un predicado, opcionalmente precedido por un identificador entre corchetes. Los axiomas se introducen por medio de la palabra clave **axiom** y consiste de definiciones de predicados separadas por comas. Por ejemplo, en vez de definir:

value
reading_limit: Nat • ≤ 3

se podría escribir de la siguiente manera:

value
reading_limit: Nat

axiom
[Redding_copies_limit] reading_limit ≤ 3

De hecho, todas las definiciones de valores, pueden estar escritas en este estilo, un tipo mas un axioma. Aunque este estilo “axiomático” o “algebraico” pueden ser usados en RAISE, el método permite el uso de sets, lists, maps y products predefinidos.

3.3 Las especificaciones

El método RAISE brinda cuatro alternativas en los estilos de la escritura de las especificaciones:

- **aplicativo secuencial**: un estilo de “programación funcional”, sin variables ni concurrencia;
- **imperativo secuencial**: con variables, asignaciones, secuencias, loops, etc. pero sin concurrencia;
- **aplicativo concurrente**: programación funcional con concurrencia;
- **imperativo concurrente**: con variables, asignaciones, secuencias, loops, etc. y concurrencia.

Las especificaciones aplicativas concurrentes son poco adecuadas como base para las implementaciones en leguajes de programación. Por lo tanto, el estudio en nuestro

contexto tiene lugar en los otros tipos de módulo. El estilo aplicativo es el mas fácil tanto para formular y como para razonar en las justificaciones. De este modo, es fácil comenzar con especificaciones aplicativas y desarrollar mas tarde una especificación imperativa o concurrente.
También se puede distinguir entre un estilo abstracto y uno concreto. En las especificaciones abstractas se dejan tantas alternativas de desarrollo abiertas como sea posible. Las siguientes son opciones que no tienen por que ser absolutas, por ejemplo, un módulo puede ser abstracto en algunas partes y concreto en otras:

- **abstracto aplicativo**: módulos que usan tipos abstractos y signaturas y axiomas en vez de definiciones explícitas para algunas o todas las funciones;

- **concreto aplicativo**: módulos que usan tipos concretos y contienen más definiciones de funciones explícitas;

- **abstracto imperativo**: módulos que no definen variables pero que usan **any** en sus accesos y axiomas;

- **concreto imperativo**: módulos que definen variables y contienen más definiciones de funciones explícitas.

- **abstracto concurrente**: módulos que no definen variables ni canales pero que usan **any** en sus accesos y axiomas;

- **concreto concurrente**: módulos que definen variables y canales y contienen más definiciones de funciones explícitas.

Estas definiciones son relativas. Un módulo puede ser abstracto en algunas cosas y concreto en otras. Una especificación de un sistema contendrá módulos en distintos estilos y distintos grados de abstracción. Se usará el término axiomático para describir un estilo de definición de valores en términos de signaturas y axiomas.

3.3.1 La especificación inicial

Formular una especificación inicial se vuelve la tarea más crítica en el desarrollo de software, ya que si esta falla al cumplir con los requerimientos, las tareas siguientes serán más duras.

El problema principal es ´qué es lo que se entiende´ de los requerimientos. Generalmente, los requerimientos pertenecen a algún dominio en el cual el desarrollador no es experto, y los que escriben los requerimientos tienden a olvidar los detalles que

ellos consideran obvios. Además, en general los requerimientos se reciben expresados la mayor parte de las veces en lenguaje natural, que resulta ser impreciso.

El objetivo de una especificación inicial es capturar los requerimientos en un modo formal y preciso para obtener un modelo de lo que el sistema hará. Las siguientes son recomendaciones para quien escribe una especificación del sistema:

- Ser abstracto: la especificación debería omitir detalles tanto como sea posible.

- Utilizar conceptos del usuario: como la especificación debería describir el problema, y no su solución, entonces, no debería referirse a conceptos tales como bases de datos, tablas, registros. Los conceptos en la especificación deberían ser los mismos conceptos que usa el usuario.

- Especificaciones legibles: ya que las especificaciones están destinadas ser leídas por otras personas, se desea que sean lo mas legibles posibles. Las guías son muy similares a aquellas para los lenguajes de programación: identificadores significativos, comentarios, funciones simples, módulos que no estén acoplados.

- Detectar problemas: se debería concentrar en las cosas que parecen difíciles, extrañas u originales y diferir las más simples, de modo de evitar errores o encontrarlos luego rápidamente.

- Minimizar el estado: por estado de un sistema (módulo) se entiende la información que está almacenada, que persiste entre las interacciones con el. Se debería tratar de no incluir información dependiente del estado, es decir, información que pueda ser calculada desde otra información del estado. Por ejemplo, si C puede ser calculado desde A y B, entonces no debería modelarse C como parte del estado. Esto es una noción general que cuanto mas sencillo es el conjunto de condiciones de consistencia, será mejor diseñado el estado.

- Identificar condiciones de consistencia: aunque se trata de minimizar la información del estado, es usual que se necesite condiciones de consistencia y condiciones políticas. Las condiciones de consistencia son necesarias si los valores de estado no pueden ser correspondidos a la realidad, por ejemplo, dos usuarios de una librería piden prestado la misma copia de un libro simultáneamente. Los requerimientos de consistencia se deberían identificar primero porque algunas veces es posible diseñar un estado que reducirá la necesidad de condiciones de consistencia. Por ejemplo, en algunos oportunidades, la consistencia puede ser tratada con un subtipo (se puede registrar el número de libros que alguien puede prestar como un Nat para evitar que sea negativo).

3.3.2 Módulos

El tipo de desarrollo separado es uno de los principales en los cuales está basado en el método RAISE. Cuando se desarrolla un sistema de un determinado tamaño, se debe ser capaz de descomponer su descripción y componer el sistema desde las componentes.
Los módulos constituyen el medio para descomponer especificaciones en unidades reusables y comprensibles. Un módulo (module) puede ser un scheme o un object. Un scheme es una expresión de clase nombrada y un object es un modelo nombrado seleccionado desde una clase de modelos representada por una expresión de clase. Los objects pueden estar embebidos, es decir, definidos dentro de una expresión de clase o ser globales. Los objects embebidos son usados donde sea posible ya que permiten la visibilidad sólo en la expresión de clase dentro de la cual son definidos. Los schemes pueden ser parametrizados con objects.

Un módulo desarrollado puede ser usado en otros módulos de tres maneras diferentes. Si el módulo es un scheme, puede ser usado en un parámetro formal o para construir un object embebido. Los schemes y los global objects forman un espacio de nombres que pueden potencialmente ser usados en módulos.

Los global objects son declarados en un archivo separado. Tienen un amplio rango de visibilidad, son definidos para contener una colección de tipos que serán necesarios en varios lugares. Un tipo como *date* es candidato a ser definido como un global object así como los tipos que deben ser visibles a usuarios, es decir, tipos que son parámetros de funciones de usuario o que son resultados de funciones de usuario.

La mayoría de los módulos contendrán un tipo que modela el estado, junto con funciones que observan y generan valores del estado. El tipo se denomina tipo de interés del módulo. Tales módulos son definidos como schemes, y suelen ser instanciados como objects embebidos dentro de otros.

Por ejemplo, para modelar collection of books en una library se define un módulo (scheme) con tipo de interés Books y otro con tipo de interés Book, y se usa el scheme BOOK para tener al object B en el scheme BOOKS. La cláusula *context* del modulo BOOKS contiene el scheme BOOK, usado para definir el object embebido B. En la siguiente especificación se muestra los schemes.

```
scheme BOOK = class
    type
      Book
end
```

```
context: BOOK
scheme BOOKS =  class
    object
      B: BOOK
    type
      Book = Book_id ~ → B.Book
end
```

Los módulos están estructurados jerárquicamente a fin de entender una componente particular por referencia sólo a ella y sus servidores, limitar los efectos de cambios a un módulo a el y sus clientes, y limitar las propiedades de un módulo a el y sus servidores. Para lograr esto, cada módulo debería tener sólo un tipo de interés, los clientes deberían sólo extender sus servidores conservativamente, y los objetos globales deberían ser usados con cuidado. Luego, un módulo A menciona las entidades de un módulo B si A es un cliente de B, y los clientes sólo se refieren a las entidades de sus servidores inmediatos. Se pueden distinguir los módulos entre los llamados módulos del sistema y los módulos subsidiarios que se describen a continuación.

3.3.2.1 Módulos del sistema

Los módulos del sistema forman la parte principal de cualquier especificación. Son los módulos que se desarrollarán de abstracto a concreto y de aplicativo a imperativo y posiblemente concurrente. Se podrían llamar más precisamente módulos del sistema o sub-sistema ya que corresponden a lo que se verá como un sistema de software completo y sus subsistemas. Se espera que estos módulos se implementen finalmente como módulos de software con estado dinámico: en términos de orientación a objetos formarán los objetos del sistema de software.

Cada módulo tendrá un tipo de interés. Para módulos abstractos aplicativos corresponde al uso común del término. Para un módulo que especifica un tipo de datos abstracto es precisamente ese tipo (pila, cola, etc.). Para módulos que especifican sistemas de software o sub-sistemas es el tipo el estado del sistema o subsistema. Para módulos imperativos el tipo de interés es el producto de los tipos de las variables en el módulo y los tipos de interés de sus módulos componentes.

3.3.2.2 Módulos subsidiarios

Hay tres clases de módulos subsidiarios:

- Módulos de tipos: son útiles como un lugar para definir todos los tipos que deseamos usar en una especificación, junto con funciones aplicativas útiles sobre estos tipos. En una especificación grande puede ser útil tener varios de estos módulos, uno

para el desarrollo como un todo y uno para cada sub-desarrollo separado, de modo que si un equipo de trabajo desea poner un tipo nuevo pueda hacerlo en el módulo local. Se recomienda usar objetos globales para este propósito. Los equipos sólo deberán ponerse de acuerdo en cuestiones de nombres.

- Módulos auxiliares: son aplicativos, agrupan funciones auxiliares sobre tipos de datos concretos. Por ejemplo, un módulo que contiene algunas funciones útiles sobre listas (reversa, es_permutación, etc.). Estos no deberían definirse como parte del módulo de tipos de un sistema particular porque el módulo puede ser genérico, parametrizado por el tipo de elemento en las listas y quizás por la relación de orden.

- Módulos parámetro: se usan para definir parámetros a otros módulos; son generalmente aplicativos. Se utilizarán para dos propósitos: a) definir módulos genéricos, es decir módulos que se espera instanciar más de una vez con diferentes parámetros; b) permitir que los módulos compartan otros imperativos o concurrentes.

Para expresar la dependencia de un módulo sobre otro se usa a los objetos embebidos u objetos globales en vez de parametrización. Cuando es posible se usan los objetos embebidos ya que hacen visibles a los objetos en la expresión de clase en la que se definen y, si no se esconden, a otros usuarios del esquema u objetos definidos que usan esa expresión de clase. Este no es el único estilo posible ya que todas las dependencias se pueden hacer usando parametrización. En sistemas de un tamaño y complejidad considerables, las listas de parámetros pueden ser muy largas.

3.4 El Metamodelo de RSL

La figura 3.1 describe el metamodelo con los elementos que son fundamentales para establecer el mapping definido en el capítulo 4 de esta tesis.

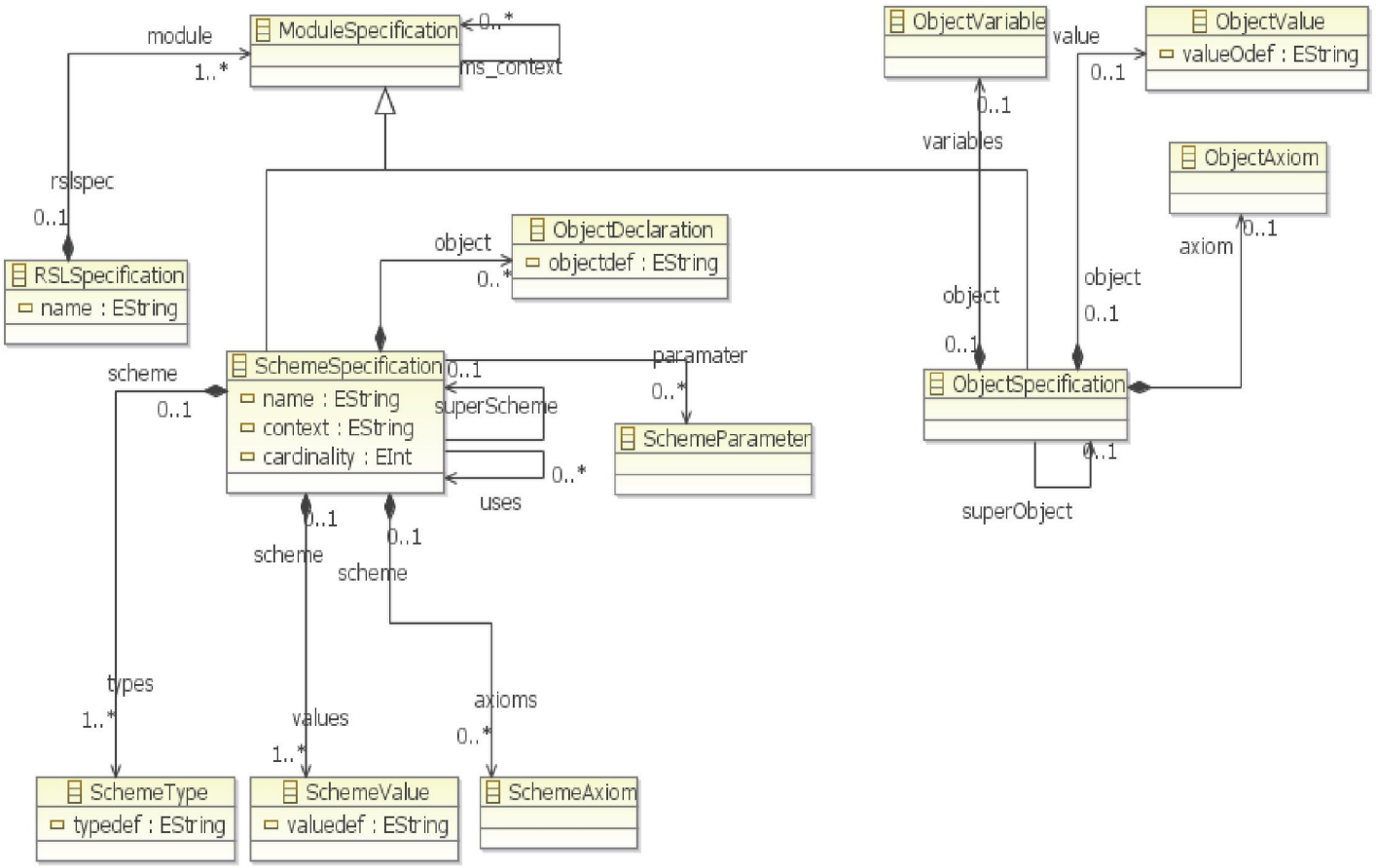


Figura 3.1 Metamodelo RSL

3.4.1 Definición de los elementos del metamodelo

A continuación se definen las clases del metamodelo.

- **RSLSpecification**

Super Class: no posee

Descripción: representa la colección de módulos.

Propiedades:

- name**:** EString. Especifica el nombre del modelo, generalmente asociado al dominio que modela no posee propiedades adicionales.

Asociaciones:

- module: ModuleSpecification [1..*]. Especifica el conjunto de especificaciones de módulos de los cuales una especificación RSL está compuesta.

Restricciones: no posee restricciones adicionales.

- **ModuleSpecification**

Super Class: ModuleSpecification

Descripción: representa el medio para descomponer especificaciones en unidades comprensibles y reusables.

Propiedades: no posee propiedades adicionales.

Asociaciones:

- ms_context: ModuleSpecification [0..*]. Referencia a los módulos que pueden formar parte como contexto de otro módulo.
- rslspec: RSLSpecification [0..1]. Referencia a la especificación RSL de la que forma parte.

Restricciones: no posee restricciones adicionales.

§ SchemeSpecification

Super Class: ModuleSpecification, SchemeSpecification

Descripción: representa las expresiones de clases nombradas.

Propiedades:

- § name: EString. Especifica el nombre del scheme.
- § context: EString. Especifica los nombres de los schemes que forman parte del contexto.
- § cardinality: EInt. Especifica el valor que puede tomar el atributo cardinalidad.

Asociaciones:

- § object: ObjectDeclaration [0..*]. Referencia al conjunto de objetos que puede tener embebido un scheme specification.
- § types: SchemeType [0..*]. Referencia al conjunto de types que se definen en un scheme specification.
- § values: SchemeValue [0..*]. Referencia al conjunto de constantes y funciones que puede tener un scheme specification.
- § axioms: SchemeAxiom [0..*]. Referencia al conjunto de declaraciones en forma de axiomas que puede tener un scheme specification.
- § parameter: SchemeParameter [0..*]. Referencia al conjunto de objetos que puede tener embebido una scheme specification.
- § superScheme: SchemeSpecification [0..*]. Referencia a los schemes de los cuales puede padre/hijo.
- § uses: SchemeSpecification [0..*]. Referencia a los schemes de los cuales tiene una relación cliente.

Restricciones: no posee restricciones adicionales.

§ ObjectSpecification

Super Class: ModuleSpecification, ObjectSpecification

Descripción: representa las expresiones de modelos nombrados representados por una expresión de clase.

Propiedades: no posee propiedades adicionales.

Asociaciones:

- variables: ObjectVariable [0..*]. Referencia al conjunto de variables que se definen en un object specification.
- value: ObjectValue [0..*]. Referencia al conjunto de constantes y funciones que puede tener un object specification.
- axiom: ObjectAxiom [0..*]. Referencia al conjunto de declaraciones en forma de axiomas que puede tener un object specification.
- superObject [0..*]. Referencia a los objects de los cuales puede ser padre / hijo.

Restricciones: no posee restricciones adicionales.

SchemeType

Super Class: no posee.

Descripción: representa a la colección de valores relacionados dentro de una scheme specification.

Propiedades:

- typedef: EString. Especifica el nombre del type.

Asociaciones:

- scheme: SchemeSpecification [0..1]. Referencia al scheme que lo contiene.

Restricciones: no posee restricciones adicionales.

SchemeValue

Super Class: no posee.

Descripción: representa al conjunto de constantes y funciones declarados en una scheme specification.

Propiedades:

- valuedef: EString. Especifica el nombre del value

Asociaciones:

- scheme: SchemeSpecification [0..1]. Referencia al scheme que lo contiene.

Restricciones: no posee restricciones adicionales.

- ## SchemeParameter

Super Class: no posee.

Descripción: representa al conjunto de objetos y schemes que son declarados como parámetros en una scheme specification.

Propiedades: no posee propiedades adicionales.

Asociaciones: no posee asociaciones.

Restricciones: no posee restricciones adicionales.

- ## SchemeAxiom

Super Class: no posee.

Descripción: representa al conjunto de predicados declarados en una scheme specification.

Propiedades: no posee propiedades adicionales.

Asociaciones:

- scheme: SchemeSpecification [0..1]. Referencia al scheme que lo contiene.

Restricciones: no posee restricciones adicionales.

- ## ObjectDeclaration

Super Class: no posee.

Descripción: representa al conjunto de módulos embebidos en una scheme specification.

Propiedades:

- objectdef: EString. Especifica el nombre del object.

Asociaciones: no posee asociaciones.

Restricciones: no posee restricciones adicionales.

§ ObjectVariable

Super Class: no posee.

Descripción: representa al conjunto de variables que pueden almacenar valores declaradas en una object specification.

Propiedades: no posee propiedades adicionales.

Asociaciones:

- § object: ObjectSpecification [0..1]. Referencia al object que lo contiene.

Restricciones: no posee restricciones adicionales.

§ ObjectValue

Super Class: no posee.

Descripción: representa al conjunto de constantes y funciones declarados en una object specification.

Propiedades: no posee propiedades adicionales.

Asociaciones:

- § object: ObjectSpecification [0..1]. Referencia al object que lo contiene.

Restricciones: no posee restricciones adicionales.

§ ObjectAxiom

Super Class: no posee.

Descripción: representa al conjunto de propiedades lógicas que deben ser verdaderas siempre declarado en una object specification.
Propiedades: no posee propiedades adicionales.

Asociaciones:

- object: ObjectSpecification [0..1]. Referencia al object que lo contiene.

Restricciones: no posee restricciones adicionales.

Capítulo 4

Desde Feature Models a schemes RSL

4.1 Introducción

El objetivo del análisis del dominio consiste en la identificación de aspectos comunes y variables entre los sistemas pertenecientes a un dominio y su representación en una forma que se pueda explotar. Dentro de las actividades correspondientes al análisis de los features que hace el método FORM, se encuentra la clasificación de los mismos. Esta clasificación tiene lugar según el tipo de información que estos features representan. El detalle de lo que estos layers están representando se encuentra en el capítulo 2 de esta tesis. La clasificación en cuatro categorías es la siguiente:

- Capability layer,
- Operating Environment layer,
- Domain Technonology layer e,
- Implementation Techniques layer.

Los features dentro del Capability layer son aquellos que caracterizan literalmente un servicio, operación, función o performance que una aplicación posee (para el dominio en cuestión). Estos features están representando un conjunto de operaciones que coordinan secuencias de interacciones entre usuarios y sistema. Ellos han sido creados en el modelo para coordinar las acciones de interacción. Cada feature que es un feature padre se entiende como una abstracción de servicios que son accedidos a través de un conjunto de interfaces.

Estos features son los que formarán parte de la transformación propuesta en este trabajo. Básicamente esta transformación derivará en schemes RSL que, en refinamientos sucesivos de la especificación, al factorizar el comportamiento común, agregarán funciones en un módulo padre. En caso que existan varios aspectos comunes entre features de este tipo con pocas diferencias, estos features se modelan como operaciones de un módulo, o simplemente como parámetros dentro de los cuales los comportamientos relacionados con los servicios son definidos y adaptados basados en los diferentes parámetros.

Una especificación final RSL de los features hijos, tiene un estilo más cercano a una implementación, con elementos tales como:

- un **scheme** con el mismo nombre del feature
- signaturas de las funciones que representan los servicios (**values**)
- funciones necesarias para acceder y modificar el tipo (**get´s , set´s**)
- funciones que servirán como pre-condiciones para el correcto funcionamiento de otras (**axiomas**)

En este capítulo se revisan las construcciones presentes en un Feature Model (FM) y se discute el proceso que puede ser usado como transformación en expresiones RSL, mediante el análisis de un conjunto de casos.

Se presenta además la transformación ATL [3] para la derivación de schemes RSL. El proceso de transformación toma los modelos del FM y define schemes con sus atributos y relaciones. La aplicación de este proceso proporciona una manera sistemática de definir una especificación RSL, conveniente en el marco del paradigma MDD. Sin embargo, una derivación manual generalmente produce una especificación RSL mas precisa. El objetivo en este trabajo es brindar un punto de partida para hacer frente a la gran cantidad de información del análisis de dominio. En este trabajo, se ha definido el proceso para alguna de las transformaciones propuestas, el detalle de las mismas se encuentra en Anexo B.

4.2 Pasos para convertir un FM en expresiones RSL

Una ***transformación*** define cómo un conjunto de elementos de un modelo origen puede ser transformado en otro conjunto de elementos de otro modelo, el modelo destino. Una transformación contiene un conjunto de reglas que especifican su comportamiento.

En este trabajo nos interesa definir una transformación a schemes RSL sintácticamente correctos. El metamodelo source (FM) se encuentra definido en el capítulo 2 y en el capítulo 3 el metamodelo target (RSL). Las descripciones de las especificaciones RSL son escritas utilizando un estilo aplicativo secuencial. Ellas juegan un rol importante como fundamento formal para expresar la semántica de las propiedades del lenguaje gráfico y la posibilidad de servir como referencia en las transformaciones escritas en ATL (Atlas Transformation Language).

Los elementos permitidos y sus reglas de formación (sintaxis) del modelo de FM son descriptos en el lenguaje RSL usando una representación más abstracta. En la sección siguiente se definen los mappings utilizando un estilo recursivo comenzando el análisis desde el FM, el Feature Diagram (FD), en adelante, analizando cada caso en particular.

4.3 Análisis de casos

La transformación inicial comienza con la traducción del FM a una jerarquía de módulos RSL. Luego, se transforma cada FD del modelo y se plantea la transformación de cada uno de los features de los diagramas, sus relaciones y las dependencias entre estos de una manera concisa. El metamodelo desarrollado en el capítulo 2 provee las bases para este tipo de razonamiento así como también para el desarrollo de herramientas de Feature Modeling.

De este modo, la transformación estará definida por los siguientes mappings:

- Un FM en el metamodelo origen será transformado en una o más jerarquías de módulos RSL en el metamodelo destino.

- Un FD en el metamodelo origen será transformado en una jerarquía de módulos RSL en el metamodelo destino.

- El Concept (feature raíz) del metamodelo origen se transforma en expresión de clase RSL (scheme) con un tipo de interés en el metamodelo destino.

- Los features del metamodelo origen que no son features Concept serán transformados a módulos RSL (schemes), pueden presentarse en un FD los siguientes tipos de features:
    - feature ‘mandatory’ (feaMand)
    - feature ‘optional’ (feaOpt)
    - feature ‘parameterized’(feaPar)

- Las Relaciones

    - Los *agrupamientos* de features se transforman en relaciones entre schemes RSL:
        - Aggregate
        - Agrupamiento OR (groupOR)
        - Agrupamiento XOR (groupXOR)

    - *Dependencias* entre features se transforman en predicados RSL expresando la restricción correspondiente:
        - Requires
        - Excludes
        - Modifies

Para visualizar los elementos involucrados en una transformación se utiliza una notación tipo package de UML [45] donde el nombre del package identifica la transformación, y el mismo contiene: el patrón source con sus metaclases y una especificación RSL que brinda una descripción formal del patrón target luego de la transformación.

Se establecen los mappings para features que pertenecen al nivel de Capability Layer que propone el método FORM. Además, para aquellos casos donde se detalla la transformación en ATL, se brinda un modelo de objetos correspondiente a una posible configuración del modelo, detallando los elementos involucrados en la misma y los módulos RSL derivados.

Los ejemplos de esta sección fueron extraídos del FD eShop de [33], el árbol completo con sus constraints fue dividido para su estudio en varios grupos de acuerdo al análisis de cada uno de los casos. Los ejemplos utilizados en cada caso se expresan exactamente como se encuentran en el FD de la referencia. No se han traducido al idioma castellano porque se considera al dominio con expresiones propias en el idioma original y la traducción daría lugar a palabras poco utilizadas y en consecuencia poco adecuadas. El FM y los features analizados como casos en las transformaciones descriptas en este capítulo se encuentran en el Anexo A.

4.3.1 Mapping *Feature Model*

Descripción: Esta transformación declara que un FM será derivado en una o más jerarquías de módulos RSL. El método RAISE provee las guías para estructurar jerárquicamente la especificación de un sistema. Estas guías permiten obtener una jerarquía de módulos cuya especificación estará basada en las transformaciones de cada FD que pertenece a un FM. La Figura 4.1 ilustra la formalización de la transformación en lenguaje RSL expresando:

- los **schemes** ($NAME_1$, $NAME_2$, …, $NAME_n$) que forman parte de cada estructura jerárquica (modules hierarchy, -MH-),
- los **schemes** del contexto (**context)** utilizados en la definición de otros en cada una de las jerarquías.

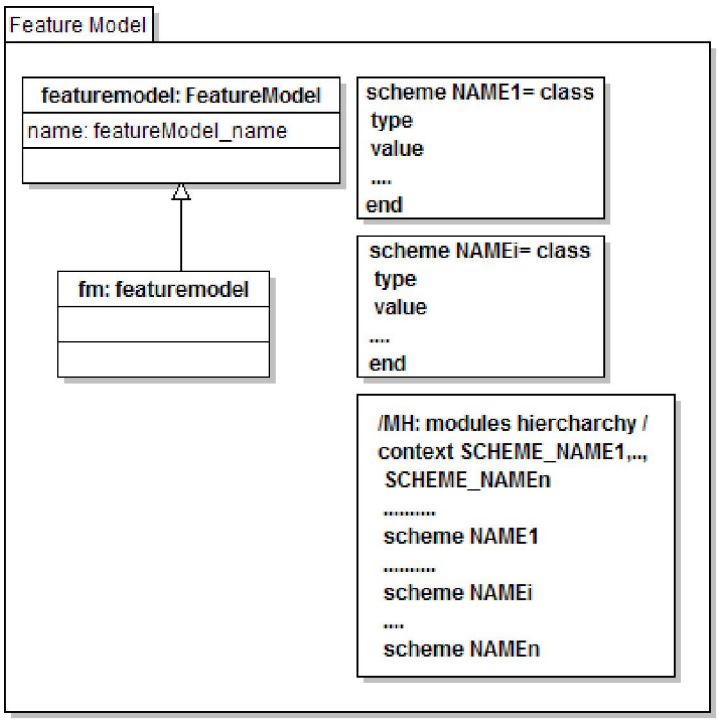


Figura 4.1 Transformación FM

Ejemplo: El ejemplo utilizado a lo largo del capítulo corresponde al sistema *Electronic shopping systems* (eShop) referenciado anteriormente y en [27]. El FM correspondiente a este sistema consta solo de un FD con el nombre eShop. El feature *concept* corresponde al feature eShop siendo el feature padre del diagrama para este caso.

4.3.2 Mapping *Feature Diagram*

Descripción: Esta transformación declara que una estructura jerárquica inicial de módulos RSL será derivada desde un FD perteneciente a un FM. La transformación de un FD dará como resultado un scheme RSL por cada feature en el diagrama y un módulo abstracto correspondiente al feature concept del FD. La jerarquía tendrá modificaciones mas tarde cuando se evoluciona con la especificación del modelo. Dentro del conjunto de schemes RSL derivados se incluyen los denominados **subsidiary**. Este tipo de módulos se utilizan para completar las definiciones de uno o más schemes de la jerarquía. Particularmente el scheme denominado ´TYPES´ que será usado para la definición de uno o más schemes del resto de la jerarquía.

Un FD está dividido en los layers que propone el método FORM, de esta manera cada módulo RSL derivado poseerá características propias del layer al que pertenece. Trabajamos aquí con elementos del FD pertenecientes al Capability Layer. La Figura 4.2 ilustra esta descripción escrita en lenguaje RSL con los elementos que forman parte de la transformación.

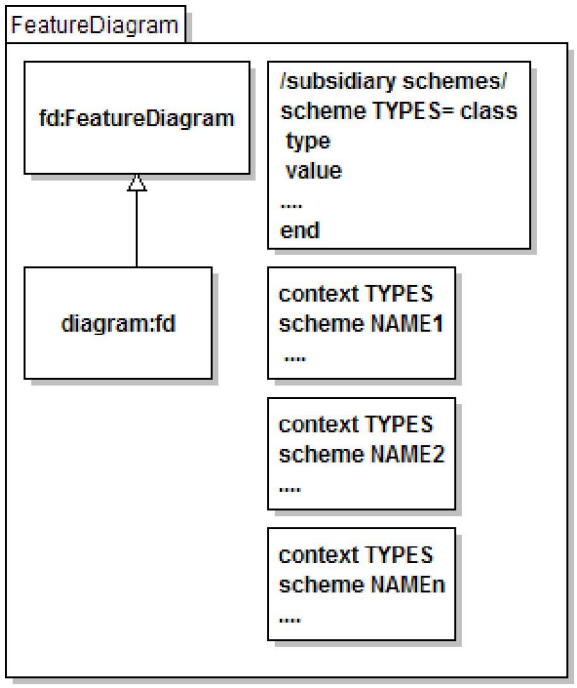


Figura 4.2 Transformación Feature Diagram

Ejemplo: el ejemplo abordado durante este capítulo tiene un solo FD, que corresponde al modelo del sistema eShop. Cada uno de los features que lo componen tendrá definida su propia transformación.

4.3.3 Mapping feature *Concept*

Descripción: El feature *concept* es el feature raíz de un FD. Representa el dominio del sistema de software que se modela y es el elemento de mayor jerarquía en un diagrama. La transformación de un feature de estas características a un scheme RSL tiene las siguientes características:

- Cada concept ***c*** en el modelo FM se transforma en un **scheme** correspondiente al modelo RSL con el mismo nombre.
- El scheme RSL tendrá un tipo de interés (**type**) cuyo nombre está representando el sistema que se modela. Las expresiones de clases con un tipo de interés RSL son usadas para especificar la jerarquía principal de módulos.

La Figura 4.3 ilustra los elementos involucrados en el mapping y una descripción en lenguaje RSL del mismo.

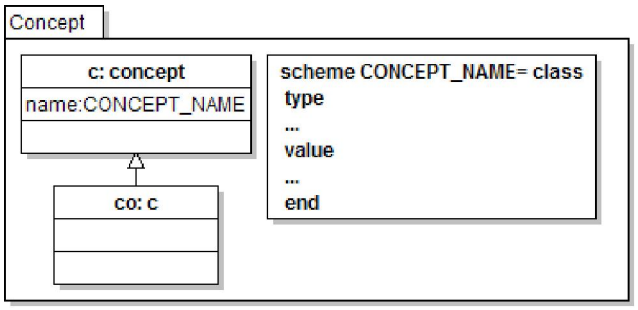


Figura 4.3 Transformación Concept

Ejemplo: en el ejemplo del eShop Systems, el feature e-Shop es el único feature concept del modelo, siendo la raíz y padre del único diagrama que posee el modelo (Figura 4.4).

Figura 4.4 Feature concept eShop

Se puede observar el modelo de objetos (Figura 4.5) que ilustra una configuración posible considerando el feature concept eShop. El mismo consta de los siguientes elementos:

- el objeto eShop es una instancia de la metaclase Concept,
- el objeto cuyo nombre representa al sistema eShop es una instancia de la metaclase FeatureModel,
- el enlace correspondiente desde el objeto concept eShop al feature model eShop que corresponde a la relación “belongs” entre un concept y un feature model.

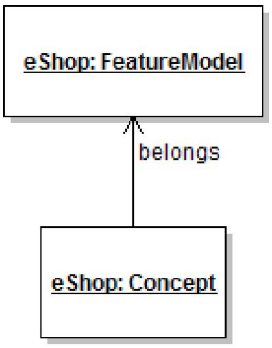


Figura 4.5 Modelo de objetos eShop - eShop

La existencia del feature concept está indicando que se generará un **scheme** que tendrá el mismo nombre del concept. El scheme derivado tiene la forma que se detalla a continuación (Figura 4.6).

```
scheme ESHOP= class
  type
    ...
  value
end
```

Figura 4.6 Módulos RSL derivados de la transformación

Transformación ATL

La transformación contiene una ´matched rule´ primaria que guía el proceso de esta transformación denominada *rule concept2Scheme*. Esta regla permite hacer el matching de los features concept a un scheme RSL. Para cada concept del modelo, la regla identifica su nombre y realiza las correspondientes asociaciones con sus atributos y relaciones. Para definir los atributos y relaciones se utilizan las ´lazy rules´ como así también los ´helpers´. El código ATL correspondiente se encuentra en el Anexo B.

En la sección siguiente, se detalla el mapping de cada tipo de features, se ilustra en la Figura 4.10 (a) el feature concept eShop. La Figura 4.10 (b) muestra el Sample Reflective Ecore Model para el mismo y la Figura 4.10 (c) el XMI correspondiente al output de esta transformación y del resto de los features.

4.3.4 Mapping *features*

4.3.4.1 Mapping *feature mandatory* (feaMand)

Descripción: los features de tipo mandatory son aquellos que formarán parte de todos los sistemas generados dentro de la familia, siendo esta propiedad heredada del feature padre. En la Figura 4.7 se muestran los elementos involucrados en la transformación, la especificación RSL expresa las componentes que estarán involucradas en el mapping:

- un **scheme** cuyo nombre viene dado por el nombre del feature.
- la cláusula **value** dentro del scheme donde se expresan las siguientes propiedades:
  - isSolitary con valor TRUE si no está formando parte de ningún agrupamiento o con valor FALSE si pertenece a algún agrupamiento,
  - isMandatory con valor TRUE,
  - isSelected con valor TRUE,
- la cláusula **extend** parent_name expresando que el feature mandatory mapeado es una extensión de un feature padre con un tipo de interés, si se está modelando una relación de herencia.
- la cláusula **context** parent_name expresando el feature que formará parte del contexto.
- el **scheme** parent_name correspondiente al feature padre ya ha sido derivado, pudiendo ser éste el feature concept o cualquier otro feature.

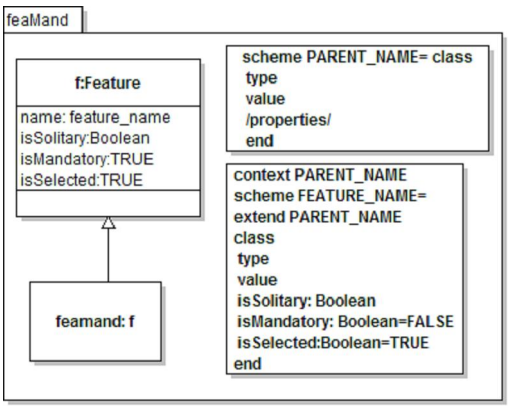


Figura 4.7 Transformación feature mandatory

Ejemplo: desde el punto de vista del metamodelo, de acuerdo a la fracción de ejemplo presentado en la Figura 4.4, se puede observar el modelo de objetos (Figura 4.8) que ilustra una configuración posible considerando el feature mandatory StoreFront. El modelo de objetos consta de los siguientes elementos:

- el objeto StoreFront es una instancia de la metaclase Feature
  - atributos:
    - isSelected: TRUE
    - isSolitary: TRUE
    - isMandatory: TRUE

- el objeto eShop es una instancia de la metaclase Concept

- el enlace correspondiente desde el objeto concept eShop al feature StoreFront que corresponde a una relación de agregación entre un concept y un feature.

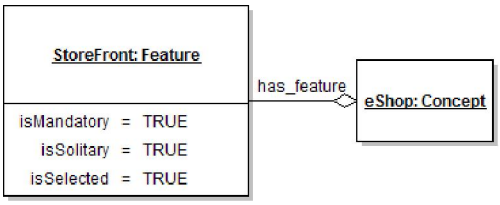


Figura 4.8 Modelo de objetos StoreFront - eShop

La existencia del feature con valor isMandatory=TRUE indica que se generará un **scheme** que llevará su mismo nombre del feature. El módulo correspondiente a eShop, ya que se encuentra derivado por ser el objeto Concept del sistema.

El mismo tratamiento tienen los features Business Managment y Customers, dando lugar a schemes RSL con la misma información. Los módulos derivados tendrán la forma que se detalla a continuación (Figura 4.9).

context: eShop **scheme** STOREFRONT= **class** **type** ... **value** isSolitary:Boolean=TRUE isMandatory: Boolean= TRUE isSelected: Boolean=TRUE **end**	**context: eShop** **scheme** BUSINESSMANAGEMENT= **class** **type** ... **value** ... **end**	**context: eShop** **scheme** CUSTOMERS= **class** **type** ... **value** ... **End**

Figura 4.9 Módulos RSL derivados de la transformación

Transformación ATL

La transformación contiene una ´matched rule´ primaria que guía el proceso de esta transformación denominada *rule fea2SchSpec*. Esta regla permite hacer el matching de los features de tipo mandatory a un scheme RSL. Para cada feature, la regla identifica su nombre y realiza las correspondientes asociaciones con sus atributos y relaciones. Para definir los atributos y relaciones se proponen las ´lazy rules´ como así también los ´helpers´. El código ATL correspondiente se encuentra en el Anexo B.

La Figura 4.10 a. ilustra el feature concept eShop y los features de tipo mandatory Store front, Business management y Customers. La Figura 4.10 b. muestra el Sample Reflective Ecore Model para estos features. Como ejemplo, se muestran las propiedades para el feature Business Management. La Figura 4.10 c. presenta el XMI correspondiente al output. Se puede ver que se genera un scheme RSL por cada feature.

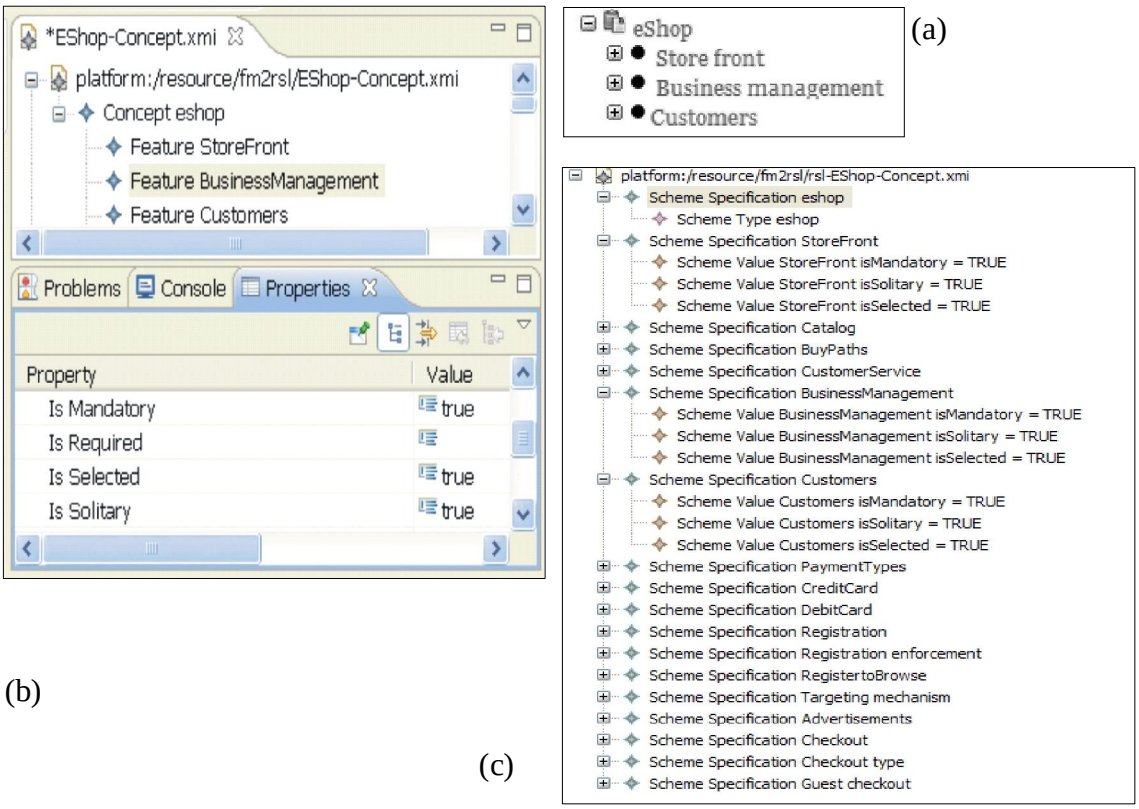


Figura 4.10 Transforamción ATL Feature Mandatory

4.3.4.2 Mapping *feature optional* (feaOpt)

Descripción: los features de tipo optional son aquellos que solo formarán parte de los sistemas generados dentro de la familia, si son seleccionados durante el proceso de configuración. La Figura 4.11 expresa las componentes que están involucradas en el mapping, y una especificación RSL formaliza esta descripción, con los siguientes los elementos involucrados en el mapping:

- un **scheme** cuyo nombre viene dado por el feature name,
- la cláusula **extend** *parent_name* expresando la que el feature optional mapeado es una extensión de un feature padre con un tipo de interés.
- la cláusula **value** dentro del scheme donde se expresan las siguientes propiedades:
  - isSolitary con valor TRUE si no está formando parte de ningún agrupamiento o con valor FALSE si pertenece a algún agrupamiento,
  - isMandatory con valor FALSE ,
  - isSelected con valor TRUE.
- la cláusula **context** parent_name expresando el feature que formará parte del contexto.
- el **scheme** correspondiente al feature padre ya ha sido derivado, pudiendo ser éste el feature concept o cualquier otro feature.

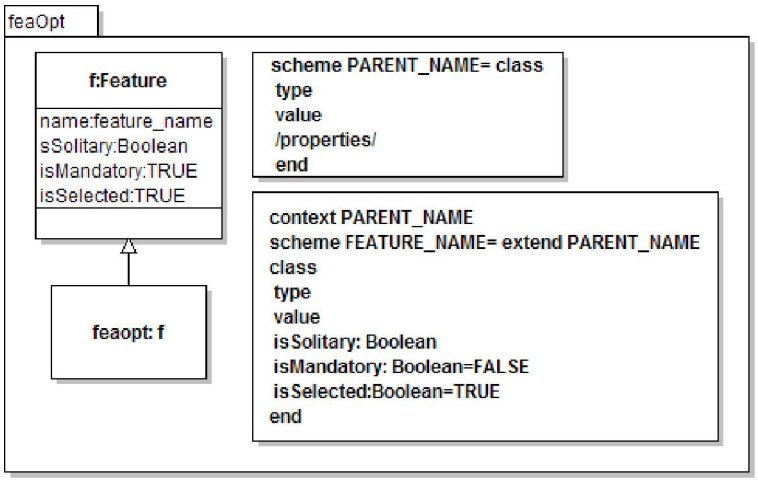


Figura 4.11 Transformación feature optional

Ejemplo: Desde el punto de vista del metamodelo y de acuerdo a la fracción de ejemplo presentado en la Figura 4.12 donde los subfeatures del feature Payment types son todos features de tipo optional, se presenta una configuración posible.

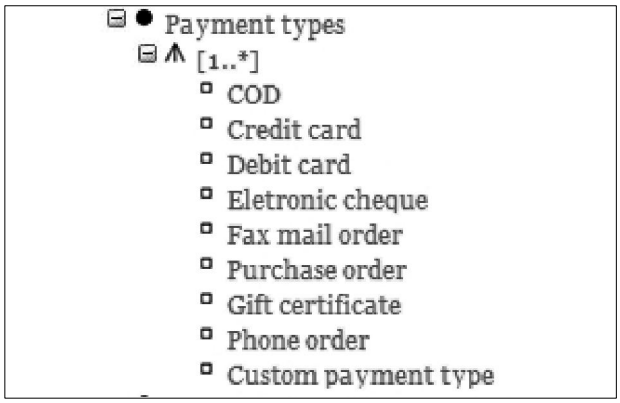


Figura 4.12 Features optional CreditCard, DebitCard, etc

Se puede observar en el modelo de objetos (Figura 4.13) que la transformación tendrá lugar con los siguientes elementos:

- el objeto CreditCard es una instancia de la metaclase Feature
  - atributos:
    - isSelected: TRUE
    - isSolitary: FALSE
    - isMandatory: FALSE

- el objeto Payment Types es una instancia de la metaclase Feature
  - atributos:
    - isSelected: TRUE
    - isSolitary: FALSE
    - isMandatory: TRUE

el enlace correspondiente desde el objeto feature CreditCard, el objeto feature PaymentTypes mediante la relación ***subfeature*** al objeto PaymentTypes y los distintos features que forman parte de esta asociación.

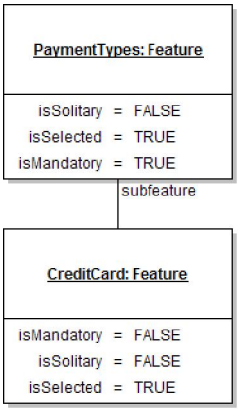


Figura 4.13 Modelo de objetos PaymentTypes - CreditCard

El feature Payment Types perteneciente a la categoría de los Capability features, se modela como un feature abstracto que relaciona distintos tipos de medios de pago para el eShop, entre ellos CreditCard, siendo éste derivado como un scheme RSL que extiende al tipo PAYMENTTYPES como se puede observar en la especificación que se muestra en la Figura 4.14.

```
context: PAYMENTTYPES
scheme CREDITCARD =
extend PAYMENTTYPES with
  class
    type
      value
        isSolitary: Boolean=FALSE
        isMandatory:Boolean=FALSE
        isSelected:Boolean=TRUE
        ...
      axioms
     .....
end
```

```
scheme PAYMENTTYPES= class
   type
     value
       isSolitary:Boolean= FALSE
       isMandatory:Boolean=TRUE
       isSelected:Boolean=TRUE
       ...
     axioms
       .....
end
```

Figura 4.14 Módulos RSL derivados de la transformación

Luego de un refinamiento en la especificación RSL, el módulo PAYMENTTYPES será un módulo perteneciente al conjunto de servicios que se acceden a través de un conjunto de interfaces. Las interfaces están divididas en operaciones Front-end y Back-end. Una especificación RSL completa para este tipo de features contendrá:

- el **scheme** PaymentTypes
- un conjunto de **funciones (values)** abstractas que identifican operaciones del front-end del Payment types como por ejemplo la de validación de los medios de pago (validate_pt ()).
- un módulo **scheme** "TYPES" como cuya especificación será válida para todos los tipos del modelo declarando un nombre de objeto de ese tipo para el uso en la especificación.
- la cláusula **context** TYPES permitiendo la visibilidad de este módulo a otros módulos.

Además, los features Credit card, Debit card, COD, Electronic cheque, y los restantes también serán mapeados como shemes RSL conservando el nombre. Estos features serán extensiones del scheme PaymentTypes. Los schemes contendrán records con campos representando atributos y una extensión (especialización) del PaymentTypes scheme. Los atributos en una especificación final de estos schemes representan la información relacionada con: nombre del titular, número, tipo, código de seguridad, fecha de expiración y demás elementos relacionados con este contexto. En las siguientes especificaciones se observa como se declaran las funciones de validación en cada una de las especificaciones: validate_cc(), validate_dc(), validate_cod(), validate_ech()) y funciones para acceder y modificar el tipo (getType(), SetState ()). La especificación final contendrá elementos como los que se muestran en la Figura 4.15.

RSL PaymentTypes	**RSL CreditCard**

scheme TYPES = **class**
  type
  Date,
  PY_ID,
  end
 	
context TYPES
scheme PAYMENTTYPES = **class**
  object T: TYPES
  type
  Payment types :: PTid: T. PT_ID
  ValidDate:T.Date
  value
  validate_pt : Payment types -> Bool
…
end
	scheme CREDITCARD=
	extend PAYMENTTYPES **with**
	class
	  object T: TYPES
	  type
	  Payment Type=
	  PAYMENT TYPE.Payment Type
	  CreditCard= {|o:Payment Type: is_a_CreditCard (o)|}
	  Payment types :: PTid: T. PT_ID
	  StateCard==Valid| Invalid | Stolen | Lost | Cancelled
	  TypeCard == International | Gold | Platinum
	  ….
	  value
	  validate_cc: CreditCard à Bool,
	  getType: CreditCard à TypeCard,
	  setState: CreditCard x StateCard à CreditCard
	axiom
	  …
	end

Figura 4.15 Especificaciones RSL PaymentTypes y CreditCard

Transformación ATL

Como para el caso de la transformación del feature mandatory, esta transformación contiene la misma ´matched rule´ primaria que se usa para la transformación descripta anteriormente: *rule fea2SchSpec*. Esta regla permite hacer el matching de los features de tipo optional a un scheme RSL. Para cada feature, la regla identifica su nombre y realiza las correspondientes asociaciones con sus atributos y relaciones. Para definir los atributos y relaciones se proponen las ´lazy rules´ como así también los ´helpers´. El código ATL correspondiente se encuentra en el Anexo B.

4.3.4.3 Mapping *feature parameterized* (feaPar)

Descripción: los features parametrizados son usados para representar valores simples. Este tipo de features serán derivados en un **scheme RSL parametrizado** con objetos. Estos objetos corresponden a los parámetros del feature. La Figura 4.16 expresa las componentes que están involucradas en el mapping, y una especificación RSL formaliza esta descripción, con los siguientes los elementos involucrados en el mapping:

- un **scheme** parametrizado cuyo nombre viene dado por el nombre del feature,
- la cláusula **context** con la lista de parámetros
- la cláusula **object** donde se declarará el tipo del objeto parámetro
- la cláusula **value** donde se expresan las siguientes propiedades:
  - isSolitary con valor TRUE si no está formando parte de ningún agrupamiento o con valor FALSE si pertenece a algún agrupamiento,
  - isMandatory con valor FALSE o TRUE,
  - isSelected con valor TRUE.

Se aplican las mismas reglas tenidas en cuenta para los features de un FD.

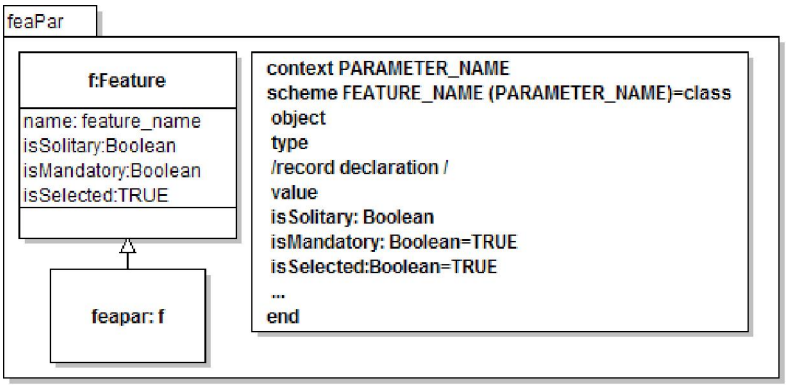


Figura 4.16 Transformación feature parametrizado

Ejemplo: la Figura 4.17 ilustra el feature UserBehaviourTrackingInformation siendo básicamente de tipo String.

El resto de los features: RegistrationEnforcement y RegistrationInformation son expresiones de clases especificadas de la misma manera que los schemes para los features mandatarios ya descriptos.

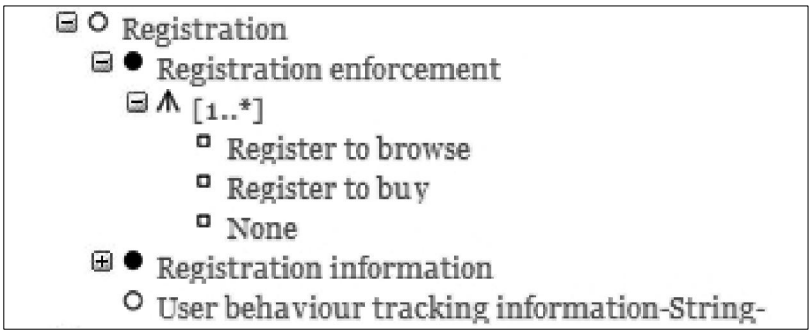


Figura 4.17 Feature Registration y sub-features

4.3.5 Mapping *RELACIONES*

Las asociaciones que se presentan en un FM y entre distintos FDs representan relaciones entre instancias de features. Puede observarse en el metamodelo propuesto, la línea que conecta features representando una relación. Las relaciones AggregateAssociation, GroupAssociation y Dependency son casos especiales de la relación *Relationship*.

La cardinalidad de la relación está representada con valores 1..*, indicando los límites superior e inferior de los objetos que participan en la instanciación del modelo.

A continuación se describe cada tipo de mappings. La descripción formal de los elementos que intervienen en la transformación se especifica en lenguaje RSL.

4.3.5.1 Mapping *AGRUPAMIENTOS*

En esta sección se describen los mappings del aggregate (básicamente composition) y de los agrupamientos OR y XOR.

4.3.5.1.1 Mapping *Aggregate*

Descripción: La AggregateAssociation es un tipo de relación que existe entre un feature padre y un grupo de dos o más features hijos. Puede expresar tanto una relación tipo *composition* como una relación tipo *aggregate*. Distinguir este tipo de asociaciones no es necesario desde un punto de vista estructural donde no se evidencia una característica distintiva entre ambas. Sin embargo, si es de interés cuando se modela el aspecto dinámico. Por un lado, el *aggregate* modela la relación todo/parte (un todo consiste de partes mas pequeñas). Por el otro, las partes de una *composition* pueden incluir features y asociaciones (el feature compuesto debe manejar la creación y destrucción de sus partes). Así, se puede tener en cuenta que una especificación final RSL contendrá la declaración de uno o mas **axiomas** expresando post-condiciones que aseguran que cuando un todo es eliminado, sus partes asociadas también lo serán. Se consideran las siguientes situaciones:

- Si el *feature parte* es un feature atómico (una hoja del árbol) y expresa tipos simples, el feature será mapeado como un componente de la definición de la clase padre. La componente es un **record** en la expresión RSL, ésta se obtiene con la simple referencia de la componente como un observador definido sobre el sort de la clase. Tiene el mismo tratamiento que el feature de tipo mandatary, optional o parameterized detallados anteriormente.

- Si el *feature parte* no es un feature atómico, las partes del aggregate se transforman directamente en expresiones de clase RSL según sea el tipo de features que se está modelando expresados como objetos embebidos de la expresión de clase que lo contiene. Estos objetos se declaran dentro de la cláusula **object** del feature que contiene la relación.

La Figura 4.18 expresa las componentes que estarán involucradas en el mapping, y una especificación RSL formaliza esta descripción.

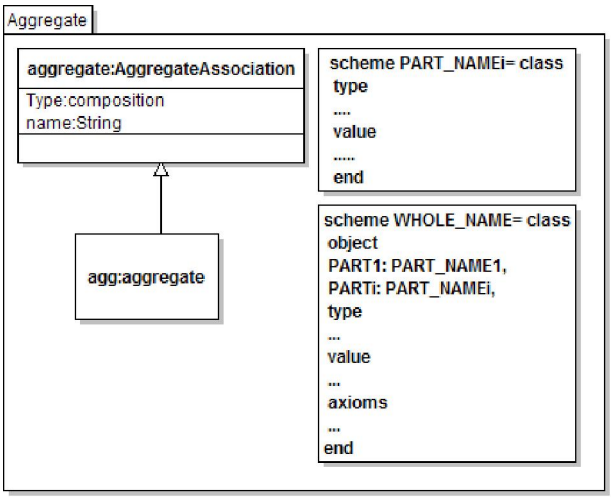


Figura 4.18 Transformación Aggregate

Ejemplo: En el fragmento del FD (Figura 4.19) que modela la parte StoreFront del sistema, se puede observar al feature padre (StoreFront) compuesto por tres subfeatures principales que son Catalog, Buy paths y Customer service. Los dos primeros son de tipo mandatory representando las dos partes principales de las que está compuesta cualquier configuración relacionada con e-commerce, mientras que el Customer service se considera de tipo optional. El resto de los features son de tipo optional.

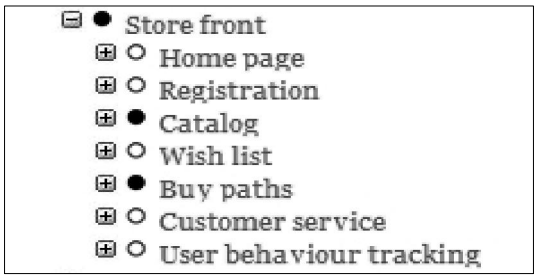


Figura 4.19 Feature StoreFront

Desde el punto de vista del metamodelo, de acuerdo a la fracción de ejemplo presentado, una configuración posible se puede observar en un modelo de objetos (Figura 4.20) con los siguientes elementos:

- el objeto StoreFront es una instancia de la metaclase Feature
  - atributos:
    - isSelected: TRUE
- el objeto Catalog es una instancia de la metaclase Feature
  - atributos:
    - isSelected: TRUE
    - isSolitary: FALSE, ya que es parte de un tipo de agrupamiento
    - isMandatory: TRUE, ya que debe ser seleccionado en la configuración correspondiente debido a lo que representa.
- el objeto Buy paths es una instancia de la metaclase Feature
  - atributos:
    - isSelected: TRUE
    - isSolitary: FALSE, ya que es parte de un tipo de agrupamiento
    - isMandatory: TRUE, ya que debe ser seleccionado en la configuración por la información que representa
- el objeto Customer service es una instancia de la metaclase Feature
  - atributos:
    - isSelected: TRUE
    - isSolitary: FALSE, ya que forma parte de un agrupamiento
    - isMandatory: FALSE, ya que no necesariamente debe ser seleccionado durante la configuración.

- los enlaces correspondientes desde el objeto StoreFront a la asociación composición (instancia de la metaclase AggregateType) a través de la propiedad ***composite*** y desde los objetos Buy paths, Catalog, y Customer service a la misma asociación a través de la propiedad ***parts***.

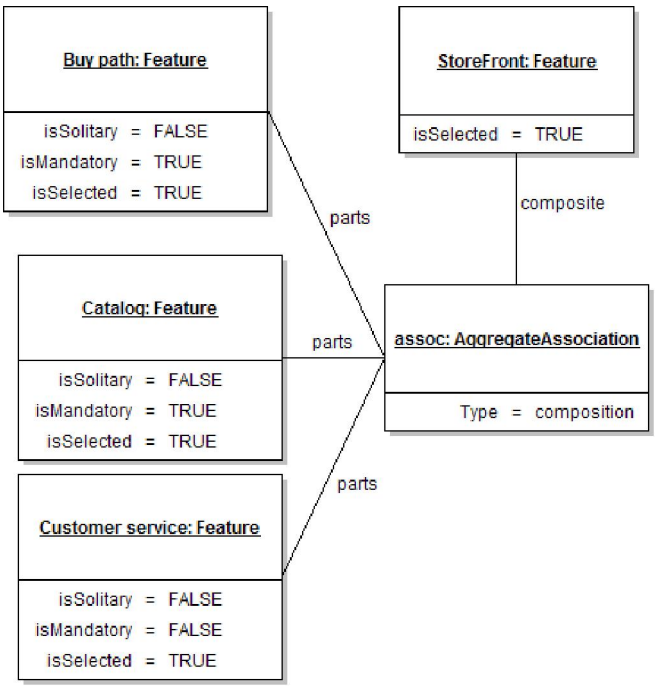


Figura 4.20 Modelo de objetos Aggregate-composite

La transformación de la asociación ´composition´ entre el feature padre StoreFront y los features Buy paths, Catalog, y Customer service dará como resultado los **schemes** correspondientes a cada uno de los features mencionados. El scheme StoreFront contendrá una estructura que refleja la composición y la misma se expresa mediante la definición de un short record RSL definido dentro la cláusula **type**, habiendo declarado los objetos Buy paths, Catalog y Customer service en la cláusula **object**.

Por otro lado, los features ´partes´ se definen de manera similar: por ejemplo el Catalog (Figura 4.21) está compuesto de varios subfeatures como por ejemplo ProductInformation, Categories, CustomsView, etc, siendo expresados de la misma manera mediante un short record como el StoreFront.

Figura 4.21 Feature Catalog

De la misma manera se puede razonar acerca de los subfeatures Buy paths y Customer service (Figura 4.22), se puede observar cada uno de los subfeatures en el caso desarrollado detalladamente en [33].

```
context: BUYPATHS, CATALOG, CUSTOMER SERVICE
scheme STOREFRONT=
  class
   object
     BP: BUYPATHS,
     CA: CATALOG,
     CS: CUSTOMER SERVICE
type
    StoreFront::
       bpaths: BP.Buypaths
       catalog: CA.Catalog
       custservice: CS.Customerservice
...
end
```

```
context:
scheme BUYPATHS=
class
  object
     PT: PAYMENTTYPE,
     FD: FRAUDDETECTION,
     ...
end
```

```
context:
scheme CATALOG=
class
   object
     PI:PRODINFORMATION,
     CA: CATEGORIES,
     CV: CUSTOMSVIEW
       ...
end
```

```
context:
scheme CUSTOMER SERVICE=
class
   object
      SST: SHIPSTTRACK,
      PR:PRODUCTRET,
      ...
end
```

Figura 4.22 Especificaciones RSL StoreFront, Buy paths, Catalog, Customer service

A pesar que esta transformación es una transformación estructural que no tiene en cuenta la funcionalidad completa de los schemes derivados, hay aspectos que son esenciales a la especificación de un sistema y pueden servir para razonar el posterior refinamiento de las especificaciones, siendo alguno de estos aspectos especificados como funciones dentro de cada uno de los módulos.

Por las razones anteriormente nombradas, es importante destacar que en este trabajo no se trata de representar el conjunto de funciones RSL completo, ya que se estaría especificando todos los aspectos dinámicos del sistema. Sin embargo, es importante en

esta etapa dar una especificación de la semántica de la composición basados en la propiedad del tiempo de vida coincidente del **todo** y sus **partes**. Para ello es necesario expresar una propiedad que restrinja la eliminación (función *del* con la restricción *can_del* como pre condición) de cada una de las partes (Buy paths, Catalog, Customer service). Estas funciones sirven para expresar la propiedad del *remove* del todo. Se puede especificar mediante una post-condición en la especificación del todo, que asegura que cuando el todo es eliminado, todas las partes que están asociadas con él, sean borradas también. Las siguientes especificaciones (Figura 4.23) muestran cómo se declaran estas funciones, producto de un refinamiento de las primeras especificaciones de los features partes obtenidas luego de la transformación.

SFPayment scheme

```
scheme BUYPATHS = class
  type
    ….
  value
     .....
    del_ BP: BUYPATHS x Syst -> Syst
     pre can_del_BP
     /declaración de la función can_del_BP/
    can_del_BP: …
 end
```

Catalog scheme

```
scheme CATALOG= class
  type
     ….
  value
      ….
     del_ Catalog: CATALOG x Syst -> Syst
     pre can_del_Catalog
     / declaración de la función can_del_Catalog/
     can_del_Catalog: ...
 end
```

SFShipping scheme

```
scheme CUSTOMERSERVICE = class
   type
     ….
   value
       …
   del_ CS:CUSTOMERSERVICE x Syst -> Syst
       pre can_del_ CS
  /declaración de la función  can_del_CS/
    can_del_CS: ...
end
```

Figura 4.23 Especificaciones RSL Buy paths, Catalog, Customer service

Transformación ATL

La transformación contiene una ´matched rule´ primaria que guía el proceso de esta transformación denominada *rule* **`rule`** *aggr2Sch.*. Esta regla permite hacer el matching correspondiente a la aggregation, los features parts y el composite a los schemes RSL. Para cada feature parts, la regla identifica su nombre y realiza las correspondientes asociaciones con sus atributos y relaciones. Para definir los atributos y relaciones se proponen las ´lazy rules´ como así también los ´helpers´. El código ATL correspondiente se encuentra en el Anexo B.

La Figura 4.24 a. ilustra el feature concept StoreFront como el padre del agrupamiento y las partes Buy path, Catalog y Customer service. La Figura 4.24 b. muestra el Sample Reflective Ecore Model para estos features. Como ejemplo, se muestran las propiedades para el feature Store front. La Figura 4.24 c. presenta el XMI correspondiente al output. Los features partes tendrán las propiedades correspondientes a cada uno y la derivación tiene la forma del mapping correspondiente a los features de tipo mandatory (feaMand) definido anteriormente.

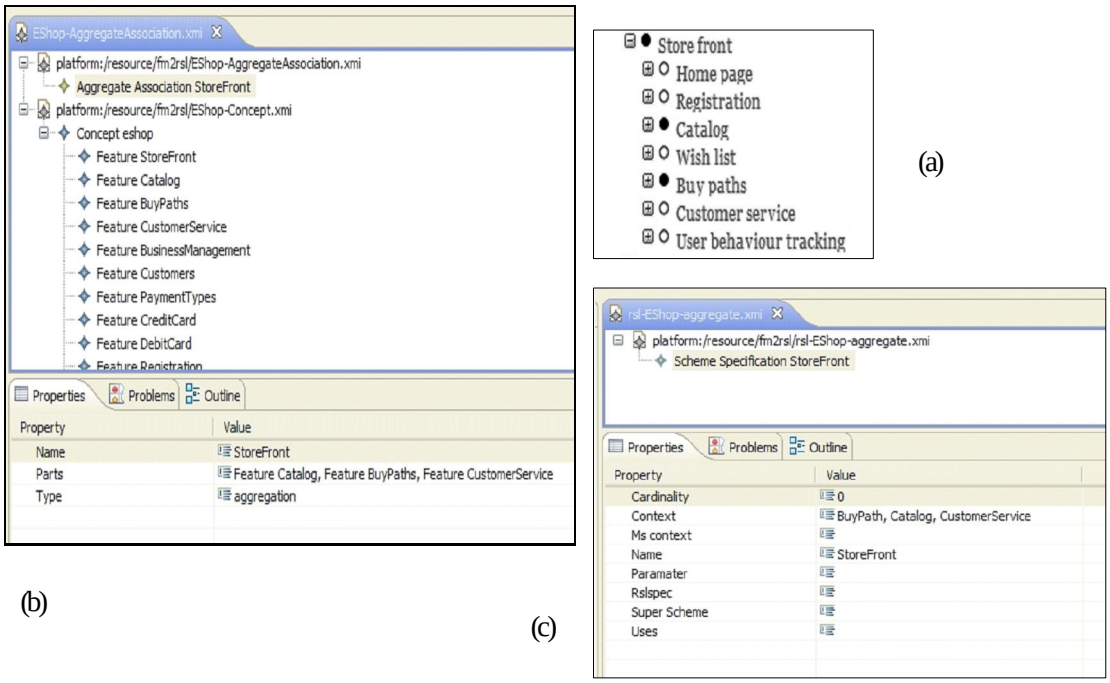


Figura 4.24 Transformación Aggregate

4.3.5.1.2 Mapping *GroupOR*

Descripción: Este tipo de agrupamiento es considerado como una relación '*is-a*' entre features de tipo alternative, mandatory u optional, con un feature padre. La estructura de este conjunto de features se mapea a una estructura jerárquica de módulos RSL. En este caso los subfeatures son considerados como extensiones o especializaciones de su feature padre. Se expresa en lenguaje RSL (Figura 4.25) una descripción del mapping propuesto con los elementos involucrados en el mapping descriptos a continuación:

- una expresión de clase (**scheme**) que tiene al menos una operación abstracta. Esto significa que esa operación está incompleta y no puede ser usada, por lo tanto sus expresiones de clases derivadas deben dar una implementación para ellas. En RSL, las operaciones abstractas son las que se especifican ocultando el nombre de la operación fuera del módulo que se está definiendo (cláusula **hide .. in**).
- la cláusula **type** con los valores que expresan la cardinalidad del grupo con un valor **lower** y un valor **upper**.
- por cada feature agrupado presente se crea una expresión de clase (**scheme**) con su mismo nombre
- cada especificación correspondiente a los features hijos llevará el nombre de la clase padre bajo la cláusula **context**. Será necesario luego redefinir todas las funciones de la especificación padre en cada subespecificación. Cada subfeature tiene su propia especificación, por lo tanto no habrá problemas de redefinición de funciones que puedan ocurrir en RSL.
- la cláusula **value** con las expresiones de restricción de consistencia (consistent).

El mapping definido para el feature abstracto en una clase RSL es el mismo que para un feature concreto. Sin embargo, es importante destacar que la diferencia que radica es que en el futuro (cuando se configura el modelo) serán expresiones de clases para las cuales no habrá objetos asociados y toda la funcionalidad que operará sobre esa clase no necesita ser especificada. En futuros refinamientos de estos módulos RSL, la restricción de consistencia se mappeará a una función booleana RSL, que expresa que ninguna instancia de la expresión de clase abstracta sea creada. Esto significa que todas las instancias que pueden estar en la clase corresponden a una de las expresiones de subclase concreta de la expresión de clase abstracta. Llamamos a la función boolena con el nombre "consistent": la unión de los dominios de las expresiones para los subfeatures es igual al dominio de la expresión para el feature abstracto.

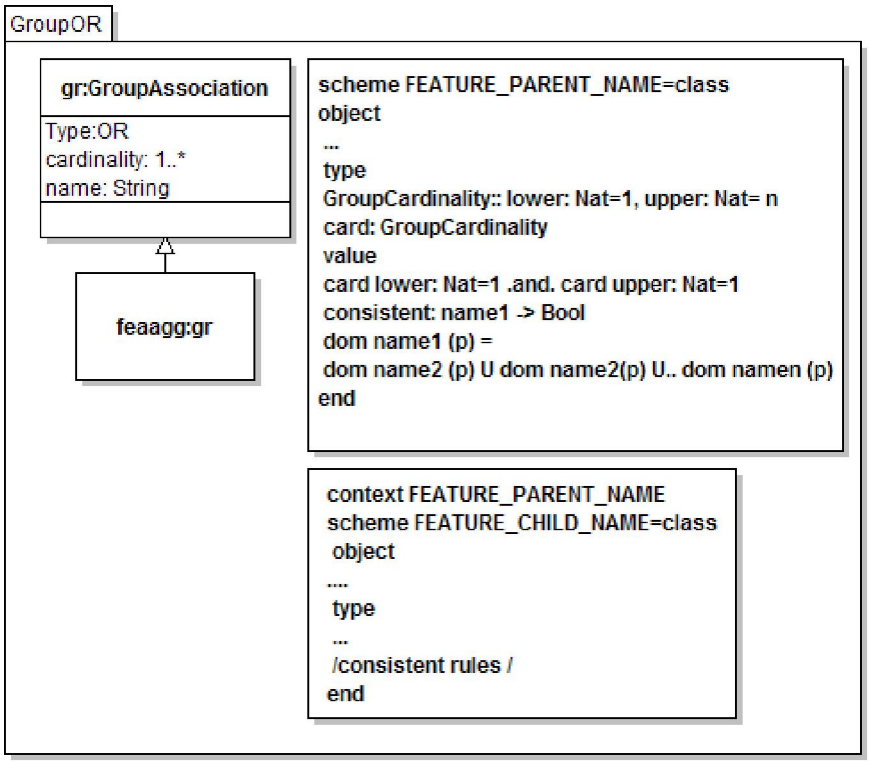


Figura 4.25 Transformación GroupOR2Schemes

Ejemplo: la Figura 4.26 ilustra al feature Registration enforcement que refleja una combinación de tres acciones posibles que pueden optar los usuarios durante una registración. Por las características de las acciones, se modela mediante un OR con cardinalidad**.** El feature Registration enforcement es un feature "abstracto", usado para expresar las posibles acciones si la registración se hace efectiva: Register to Browse, Register to Buy y None (No Register).

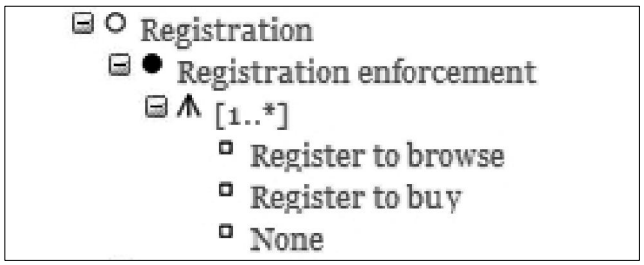


Figura 4.26 Feature Registration Enforcement

Según el metamodelo del FM y de acuerdo a la fracción de ejemplo presentado, una configuración posible se puede observar en un modelo de objetos (Figura 4.27) con los siguientes elementos:

- el objeto Regist_enforcement es una instancia de la metaclase Feature
  - atributos:
    - isSelected: TRUE
- el objeto Regtobuy es una instancia de la metaclase Feature
  - atributos:
    - isSelected: TRUE
    - isSolitary: TRUE|FALSE, ya que puede tomar cualquiera de estos valores.
    - isMandatory: FALSE, no es necesario que sea de tipo mandatory para ser seleccionado en la configuración.
- el objeto Regtobrowse es una instancia de la metaclase Feature
  - atributos:
    - isSelected: TRUE
    - isSolitary: TRUE|FALSE, ya que puede tomar cualquiera de estos valores.
    - isMandatory: FALSE, no es necesario que sea de tipo mandatory para ser seleccionado en la configuración.

los enlaces correspondientes desde el objeto Regist_enforcement al asociación (instancia de la metaclase AssociationType) a través de la propiedad OR y desde los objetos Regtoby y Regtobrowse a la misma asociación a través de la propiedad *child*.

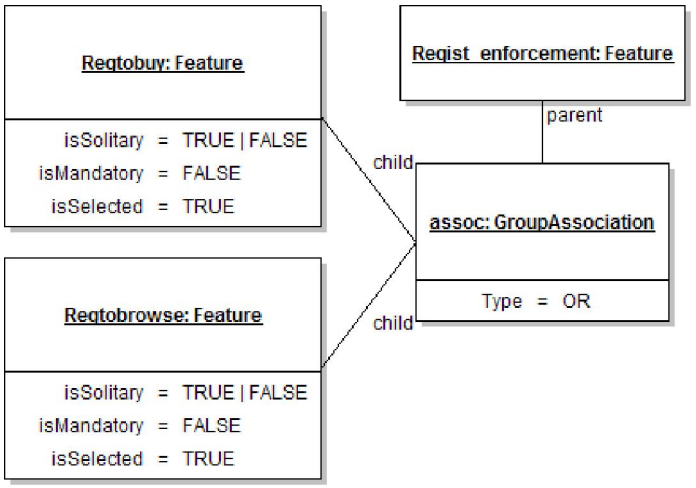


Figura 4.27 Modelo de objetos OR

La Figura 4.28 ilustra las especificaciones iniciales RSL correspondientes a estos features.

Registration to Browse RSL object	**Registration to Buy RSL object**
context REGISTENFORCEMENT, TYPES **scheme** REGTOBROWSE: **with** TYPES **in** **hide** operationName$_i$ **in** **class** **type** RegistEnforcement= REGISTENFORCEMENT.RegistEnforcement. RegToBrowse= {|o:RegistEnforcement **.** is_a_RegToBrowse (o)|} *// 0<i<=r ∧ Operation$_i$ is an abstract operation //* …. **value** **end**	**context** REGISTENFORCEMENT, TYPES **scheme** REGTOBUY: **with** TYPES **in** **hide** operationName$_i$ **in** **class** **type** RegistEnforcement= REGISTENFORCEMENT.RegistEnforcement . RegToBuy= {|o:RegistEnforcement **.** is_a_RegToBuy (o)|} *// 0<i<=r ∧ Operation$_i$ is an abstract operation //* …. **value** **end**

Registration enforcement RSL object
context TYPES **scheme** REGIST_ENFORCEMENT: **with** TYPES in **hide** /* operation list */ **in** **class** **type** Regist_Enforcement Regist_Enforcement :: Cardinality: GroupCardinality **value** /* hidden operation list */ … **end**

Figura 4.28 Especificaciones de los features partes (Regtobrowse, Regtobuy, Regist_enforcement)

Transformación ATL

La transformación ATL contiene la ´matched rule´ primaria que se usa para la transformación descripta anteriormente: *rule group2Scheme*. Esta regla permite hacer el matching de un grupo OR de features en schemes RSL. Para cada feature, la regla identifica su nombre y realiza las correspondientes asociaciones con sus atributos y relaciones. Para definir los atributos y relaciones se proponen las ´lazy rules´ como así también los ´helpers´. También se ejecutará la regla que transforma cada feature del grupo en schemes RSL y el feature Concept en otro scheme.

La Figura 4.29 a. el FD del grupo OR con Registration enforcement como feature padre, la Figura 4.29 b. muestra el Sample Reflective Ecore Model para los features que forman parte del ejemplo. Se muestran además, las propiedades para el feature Registration enforcement. La Figura 4.29 c. presenta el XMI correspondiente al output. Se observa que son generados todos los features que forman parte del grupo.

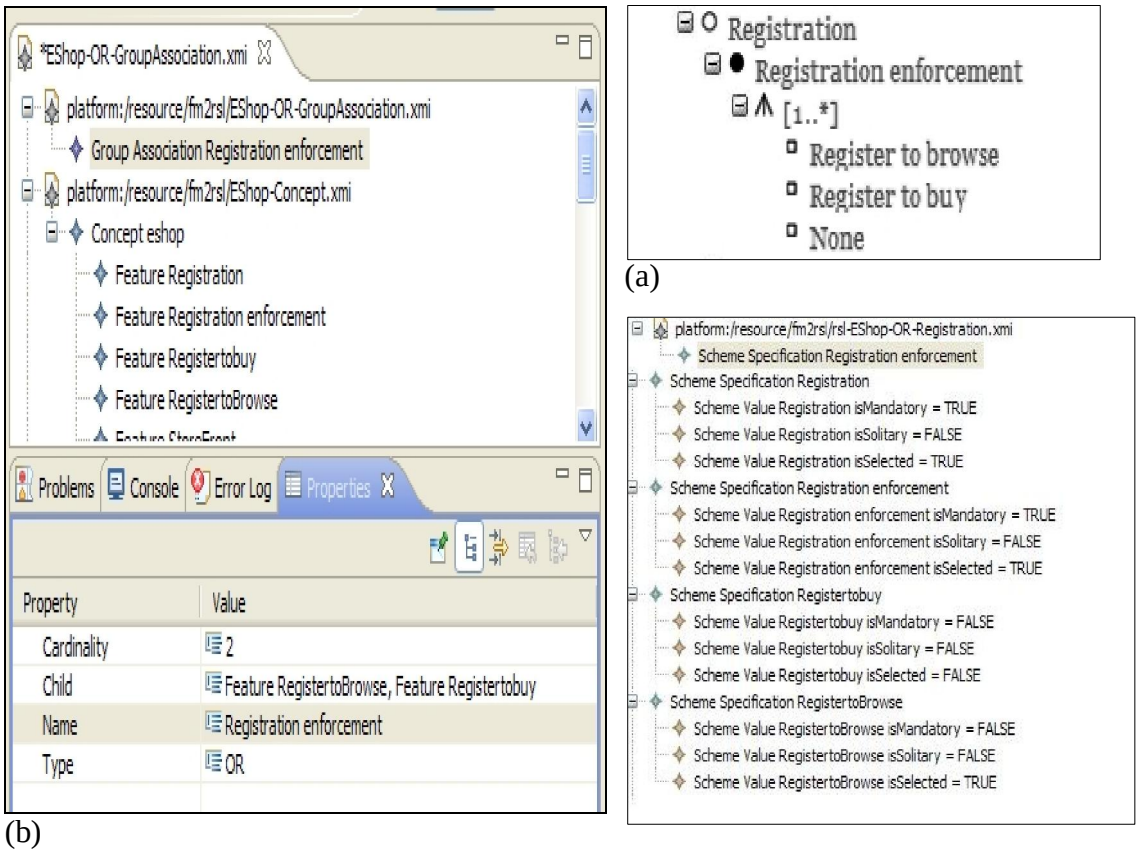


Figura 4.29 Transformación ATL Group OR

4.3.5.1.3 Mapping *GroupXOR*

Descripción: similar tratamiento para los grupos OR surgen para la construcción de los grupos XOR. Se expresa en lenguaje RSL (Figura 4.30) una descripción del mapping propuesto siendo los elementos involucrados en el mapping los siguientes:

- una expresión de clase (**scheme**) que tiene al menos una operación abstracta. Al igual que en el caso de los agrupamientos OR, esa operación está incompleta y no puede ser usada, por lo tanto sus expresiones de clases derivadas deben dar una implementación para ellas bajo la cláusula **hide .. in**.
- la cláusula **type** con la expresión de la cardinalidad del grupo con un valor **lower** y un valor **upper**.
- por cada feature agrupado presente se crea una expresión de clase (**scheme**) con su mismo nombre
- cada especificación correspondiente a los features hijos llevará el nombre de la clase padre bajo la cláusula **context**.
- la cláusula **value** donde se expresan las restricciones de consistencia (consistent)

El mapping para este caso de agrupamiento sigue el mismo razonamiento que para el tipo de agrupamiento OR, con la función “consistent”, creando para cada feature agrupado una expresión de clase similar RSL. Todos los features del grupo aparecen como opcionales. Sin embargo, al pertenecer a un grupo XOR, uno de los features debe ser seleccionado en tiempo de configuración.

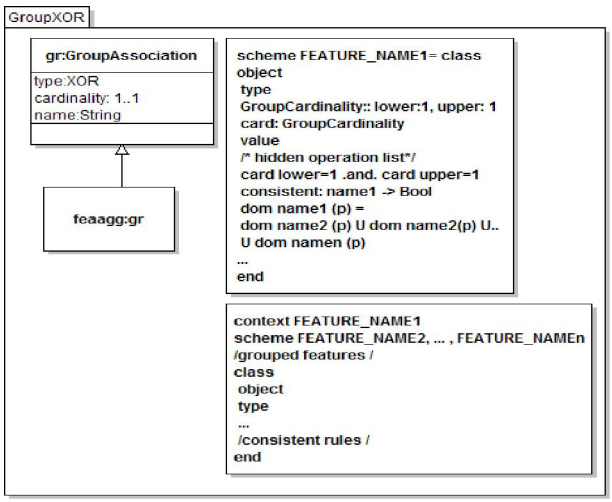


Figura 4.30 Transformación GroupXOR

Ejemplo: El feature PaymentType de la Figura 4.31 representa un grupo de XOR-features (CreditCard, DebitCard y PurchaseOrder) indicando que de esas opciones, exactamente una debe ser seleccionada cuando su feature padre lo ha sido. PaymentType es un feature "abstracto" y en este caso no hay cardinalidad que restrinja la selección en tiempo de configuración, ya que se podrá seleccionar solo una forma de pago como lo indican los subfeatures.

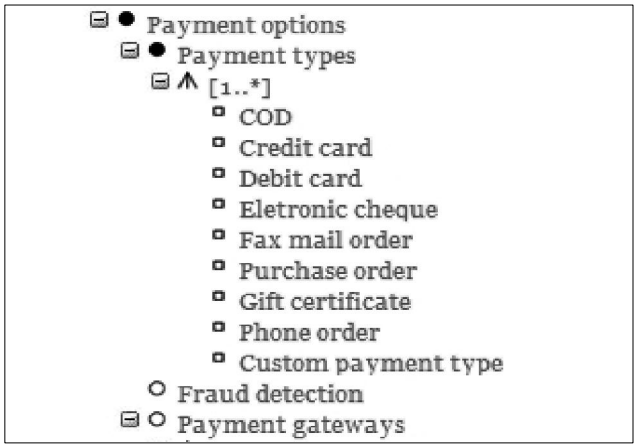


Figura 4.31 Feature Payment types

Según el metamodelo del FM y de acuerdo a la fracción de ejemplo presentado, una configuración posible se puede observar en un modelo de objetos (Figura 4.32) con los siguientes elementos:

- el objeto Payment types es una instancia de la metaclase Feature
  - atributos:
    - isSelected: TRUE
    - isMandatory: TRUE
    - isSolitary: FALSE
- el objeto CreditCard es una instancia de la metaclase Feature
  - atributos:
    - isSelected: TRUE
    - isSolitary: FALSE
    - isMandatory: FALSE, no es necesario que sea de tipo mandatory para ser seleccionado en la configuración.
- el objeto DebitCard es una instancia de la metaclase Feature
  - atributos:
    - isSelected: TRUE
    - isSolitary: FALSE

§ isMandatory: FALSE, no es necesario que sea de tipo mandatory para ser seleccionado en la configuración.

los enlaces correspondientes desde el objeto Payment types al asociación (instancia de la metaclase AssociationType) a través de la propiedad XOR y desde los objetos DebitCard y CreditCard a la misma asociación a través de la propiedad *child*.

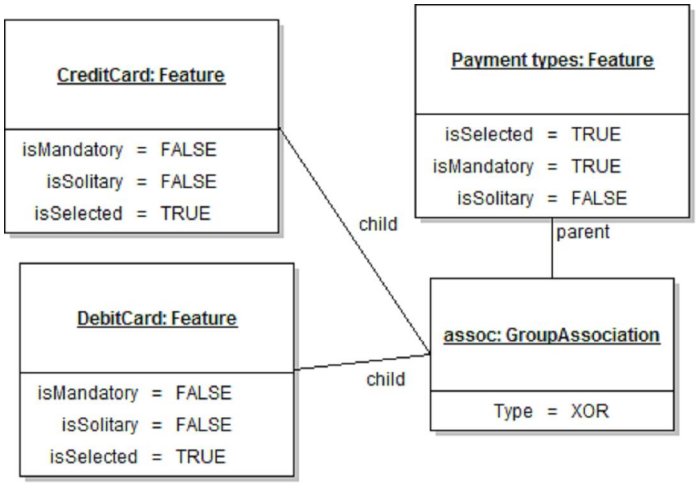


Figura 4.32 Modelo de objetos XOR

La Figura 4.33 muestra una especificación RSL inicial de estos features.

Payment type scheme

```
TYPES
scheme PAYMENTTYPE
  with TYPES in
  hide /* operation list */ in
  class
    type PaymentType
      …
    value
      consistent: PaymentType -> Bool
      consistent (r)≡ …
  end
```

context PAYMENTTYPE **scheme** CREDITCARD: **with** TYPES **in** **hide** operationName$_i$ **in** **class** **type** PaymentType= PAYMENTTYPE.PaymentType CreditCard= {|o:PaymentType. is_a CreditCard (o)|} *// 0<i<=r ∧ Operation$_i$ is an abstract operation //* …. **value** .... **end**	**context** PAYMENTTYPE **scheme** DEBITCARD: **with** TYPES **in** **hide** operationName$_i$ **in** **class** **type** PaymentType= PAYMENTTYPE.PaymentType DebitCard= {|o:PaymentType . is_a_DebitCard(o)|} *// 0<i<=r ∧ Operation$_i$ is an abstract operation //* …. **value** …. **end**

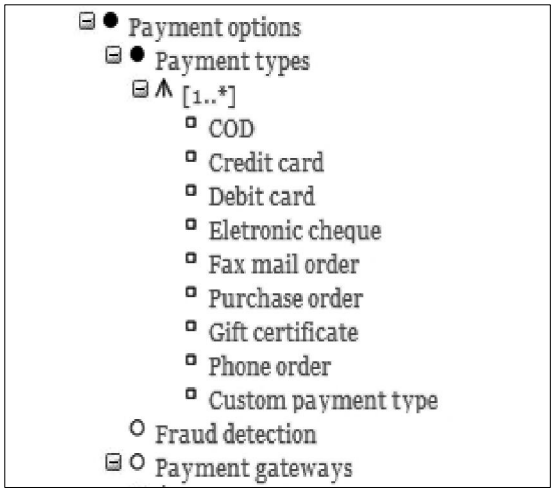
Figura 4.33 Especificación RSL PaymentType, Credit Card y Debit Card

Transformación ATL

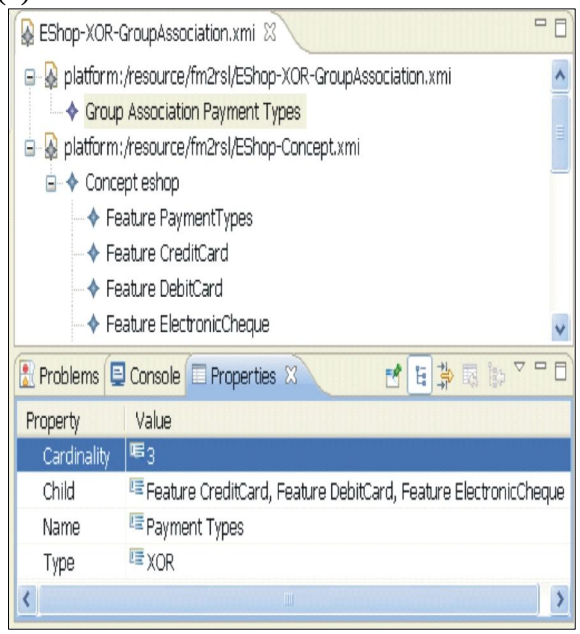
La transformación ATL contiene la ´matched rule´ primaria que se usa para la transformación descripta anteriormente: *rule group2Scheme*. Esta regla permite hacer el matching de un grupo XOR de features en schemes RSL. Para cada feature, la regla identifica su nombre y realiza las correspondientes asociaciones con sus atributos y relaciones. Para definir los atributos y relaciones se proponen las ´lazy rules´ como así también los ´helpers´. También se ejecutará la regla que transforma cada feature del grupo en schemes RSL y el feature Concept en otro scheme.

La Figura 4.34 a. muestra el Sample Reflective Ecore Model para los features que forman parte del ejemplo para el grupo XOR (Payment Types). Se muestran las propiedades para el Group Association Payment Types. La Figura 4.34 b. presenta el XMI correspondiente al output. Se observa que son generados todos los features que forman parte del grupo (DebitCard, CreditCard, Electronic cheque).

(a)



(b)



(c)

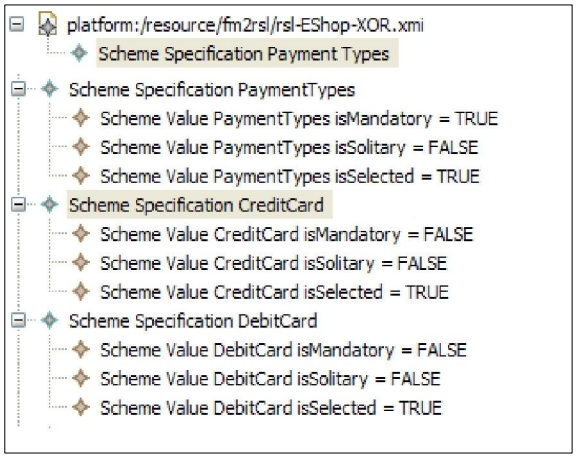


Figura 4.34 Transformación ATL Group XOR

4.3.5.2 Mapping *DEPENDENCY*

Los distintos métodos basados en Feature Modeling definen relaciones para soportar la consistencia del modelo completo y la correctitud de una selección particular de features al momento de la configuración. De este modo, además de las relaciones entre features y subfeatures, un FM puede también contener restricciones ´cross-tree´ entre pares de features. Ellas definen restricciones entre features que no están expresadas por el árbol del modelo y que toman la forma de una implicación. Estas restricciones se clasifican en dos tipos: restricciones "fuertes" y restricciones "débiles". Las primeras condicionan el modelo porque influyen activamente en él cuando se toman decisiones durante la configuración; son las relaciones *requires, excludes y modifies*; en tanto que las segundas son recomendaciones que no necesariamente deban ser aplicadas y son las llamadas *is_independent_of* y *recommend*.

Este tipo de relaciones no necesariamente tienen que ser bidireccionales; lo mas común es que si un feature requiere a otro, la selección de este último recomiende la selección del primero, pero también puede haber independencia en un solo sentido [28].

4.3.5.2.1 Mapping *REQUIRES*

Descripción: La relación ´*requires*´ es considerada como una de las restricciones fuertes del modelo ya que está forzando la existencia de un feature si se selecciona el feature que lo requiere en tiempo de configuración. Feature A *require* feature B indica que si se selecciona A entonces B debe ser seleccionado también. La Figura 4.35 ilustra los elementos vinculados en el mapping propuesto:

- una expresión de clase (**scheme**) para el feature supplier,
- una expresión de clase (**scheme**) para el feature requester
- una fórmula booleana ´**well_formed**´ usada para definir relaciones bien formadas en el scheme requester. En este caso esta fórmula debe describir la implicancia descripta anteriormente.

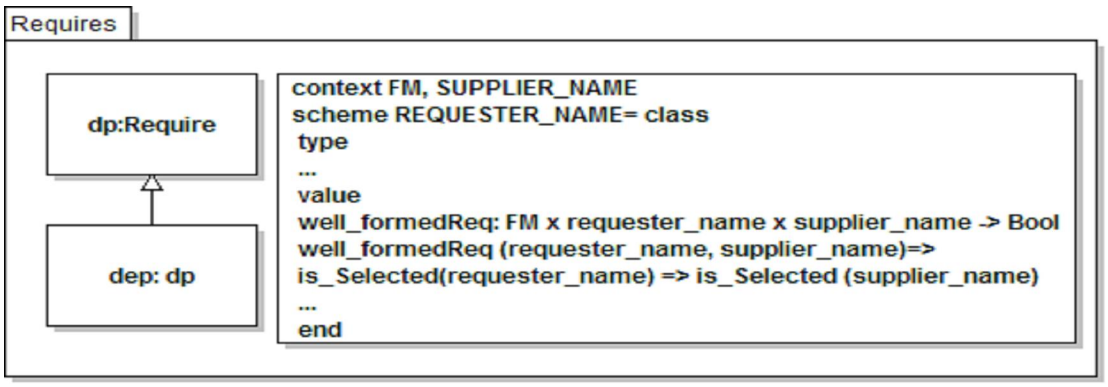


Figura 4.35 Transformación Requires

Ejemplo: El ejemplo de la Figura 4.36 consiste en: el feature Targeting está definido por tres features mandatarios, entre ellos el feature TargetingMechanisms. El grupo de features OR que forman parte del grupo de subfeatures contiene al feature Discount y el feature Advertisement. El feature Discount está modelando los mecanismos para definir y manejar ajustes de precios sobre productos. Por otro lado, el contexto del dominio expresa que SpecialOffers consiste de una serie de descuentos. De este modo, surge la necesidad de establecer la relación ´requires´entre SpecialOffers y Discount ya que para que pueda haber Special Offers deben existir los Discuounts. Se puede observar en la figura los features que intervienen en esta restricción pertenecientes al mismo Feature Diagram y la restricción expresada como una cross-tree constraint como cláusulas CNF.

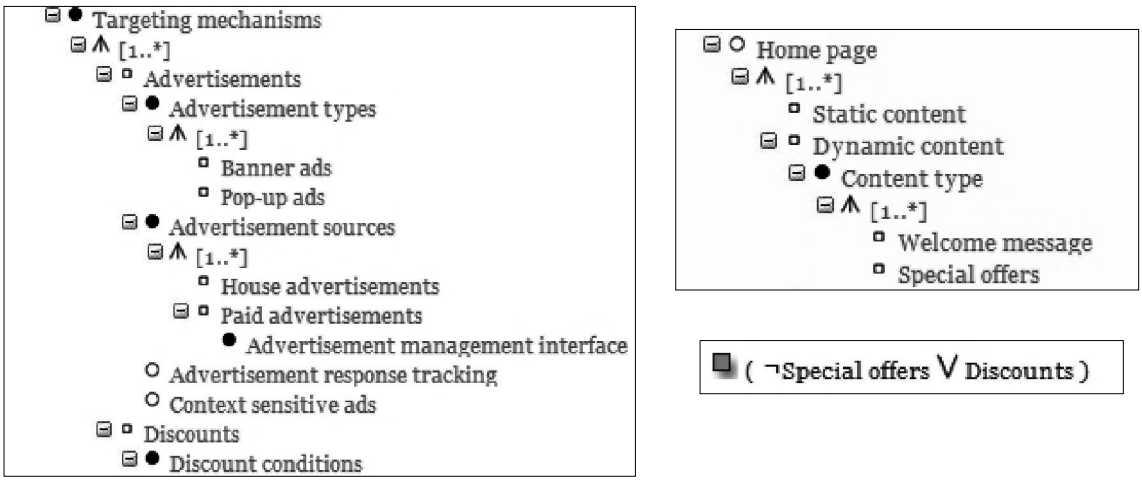


Figura 4.36 Features DynamicContent y Targeting mechanisms - Relación ´requires´

Según el metamodelo propuesto, de acuerdo a la fracción de ejemplo presentado, se observa la instanciación mediante el modelo de objetos (Figura 4.37), con los siguientes elementos:

- el objeto SpecialOffer es una instancia de la metaclase Feature
- el objeto Discount es una instancia de la metaclase Feature
- los enlaces correspondientes desde los objetos SpecialOffer y Discount a la relación Requires (instancia de la metaclase Dependency).

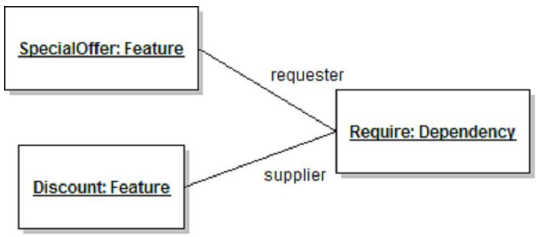


Figura 4.37 Modelo de objetos restricción Requires

La dependencia ´requires´ entre ambos features generará un **scheme** por cada feature tipo involucrado y la relación **use** desde el feature requester. El scheme SpecialOffer estará relacionado mediante la relación de uso con el scheme Discount (Figura 4.38).

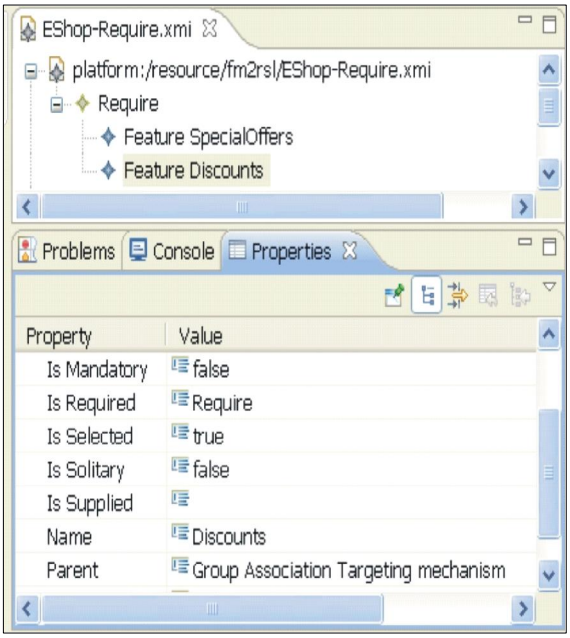
scheme SPECIALOFFER= **class** **type** specialoffer **use** DISCOUNT **value** **…** **end**	**scheme** DISCOUNT= **class** **type** discount **value** ... **end**

Figura 4.38 Especificaciones RSL Special offer y Discount

Transformación ATL

La transformación ATL contiene una matched rule primaria que guia el proceso de esta transformación denominada *rule Requires*. Esta regla permite hacer el matching de los features con su tipo correspondiente a los schemes RSL. La regla comienza analizando el tipo de Dependency, en este caso el Require, identifica su nombre y realiza las correspondientes asociaciones con sus atributos y relaciones. Para definir los atributos y relaciones se proponen las lazy rules como así también helpers.

La Figura 4.39 a. muestra el Sample Reflective Ecore Model para los features que forman parte del ejemplo. Se muestran las propiedades para el feature Discount. La Figura 4.39 b. presenta el XMI correspondiente al output. Se puede ver que se generan los features que la relación vincula.



(a)

(b)

Figura 4.39 Transformación ATL Requires

Otra de las restricciones fuertes del modelo es la relación ´excludes´. Las restricciones fuertes deben ser cumplidas ya que un feature depende de otro para su definición u correcta operación.

4.3.5.2.2 Mapping *EXCLUDES*

Descripción: Cuando dos features están vinculados bajo la restricción ´*excludes*´, la selección de uno de ellos (no importa cual) implica que el otro feature no será seleccionado en la configuración. El orden en el cual los features se seleccionan para participar en la relación es irrelevante, porque no importa cual feature es el feature fuente y cual es el feature target. La Figura 4.40 ilustra los elementos vinculados en el mapping propuesto:

- una expresión de clase (**scheme**) para el feature supplier,
- una expresión de clase (**scheme**) para el feature requester
- una fórmula booleana ´**well_formed**´ usada para definir relaciones bien formadas en el scheme requester. En este caso esta fórmula debe describir la implicancia descripta anteriormente.

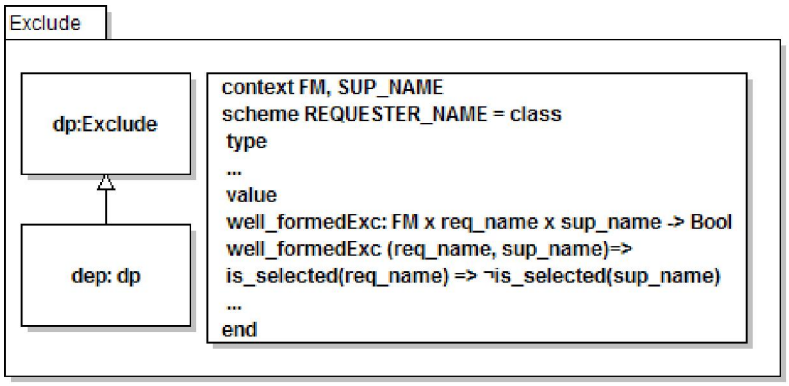


Figura 4.40 Transformación Exclude

Ejemplo: en el fragmento del modelo que se observa en la Figura 4.41 si el feature Registered to buy (feature A) es seleccionado para su eliminación, luego, no hay manera de forzar a un cliente a loguearse para hacer un checkout (Registered checkout -feature B-), por lo tanto un checkout no puede ser soportado. Allí es donde se introduce la relación: feature A excludes feature B. La figura expresa además la restricción como una cláusula CNF.

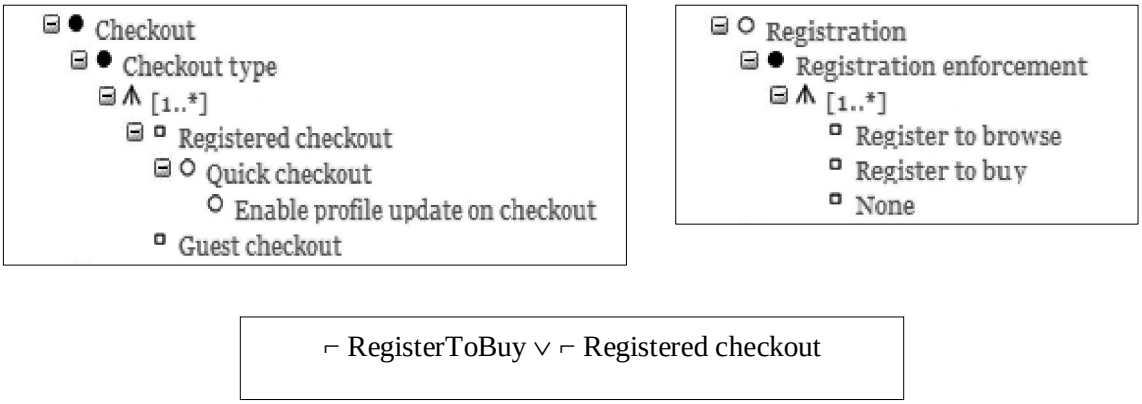
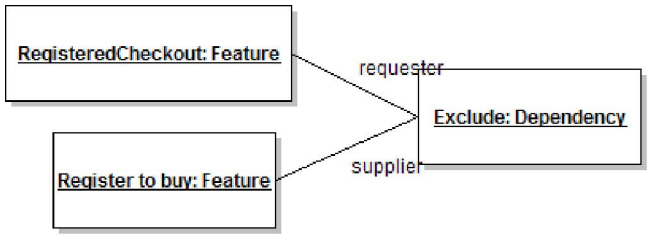


Figura 4.41 Features Registered checkout y Register to buy - Relación ´excludes´

Desde el punto de vista del metamodelo, en el ejemplo anterior se observa la instanciación mediante el modelo de objetos (Figura 4.42), con los siguientes elementos:

- el objeto RegisteredCheckout es una instancia de la metaclase Feature
- el objeto Register to buy es una instancia de la metaclase Feature
- los enlaces correspondientes desde los objetos RegisteredCheckout y Register to buy a la relación Excludes (instancia de la metaclase Dependency).



CFC= {RegisteredCheckout}

Figura 4.42 Modelo de objetos restricción Exclude

La dependencia exclude entre ambos features generará un solo **scheme** correspondiente al feature que requiere la dependencia, en este caso el feature RegisteredCheckout y no el scheme Registertobuy, ya que la dependencia exclude invalida la existencia del mismo (Figura 4.43).

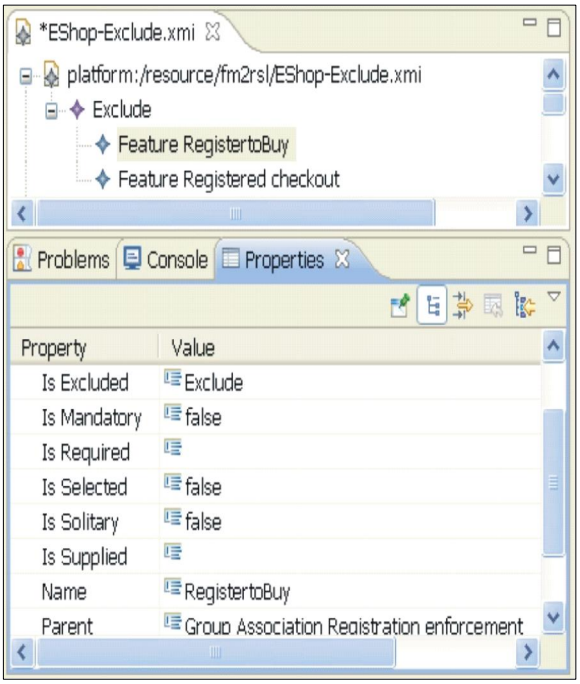
```
scheme REGISTEREDCHECKOUT = class
  type registeredcheckout
  value
  ....
end
```

Figura 4.43 Especificación RSL Registered Checkout

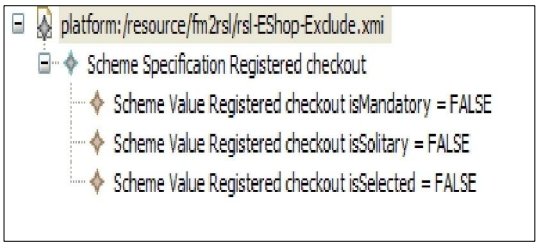
Transformación ATL

La transformación contiene una ´matched rule´ primaria que guía el proceso de esta transformación denominada *rule fea2SchSpec*. Esta regla permite hacer el matching de los feature verificando las dependencias entre los features. Para cada feature que se debe generar, la regla identifica su nombre y realiza las correspondientes asociaciones con sus atributos y relaciones. Para definirlos se implementan las ´lazy rules´ como así también los ´helpers´. El código ATL correspondiente se encuentra en el Anexo B.

La Figura 4.44 a. muestra el Sample Reflective Ecore Model para estos features. Como ejemplo, se muestran las propiedades para el feature RegistertoBuy. La Figura 4.44 b. presenta el XMI correspondiente al output.



(a)



(b)

Figura 4.44 Transformación ATL Exclude

4.3.5.2. Mapping *MODIFIES*

Descripción: La relación ´*modifies*´ es considerada como otra de las restricciones fuertes del modelo ya que tiene lugar cuando la inclusión de un feature modifica el valor de una propiedad de otro feature.

Al igual que la relación requires, se está forzando la existencia de un feature si se selecciona el feature que lo modifica en tiempo de configuración. Feature A *modifies* feature B indica que si se selecciona A entonce B debe ser seleccionado también.
La Figura 4.45 ilustra los elementos vinculados en el mapping propuesto:

- una expresión de clase (**scheme**) para el feature supplier,
- una expresión de clase (**scheme**) para el feature requester
- una fórmula booleana ´**well_formed**´ usada para definir relaciones bien formadas en el scheme requester. En este caso esta fórmula debe describir la implicancia descripta anteriormente.

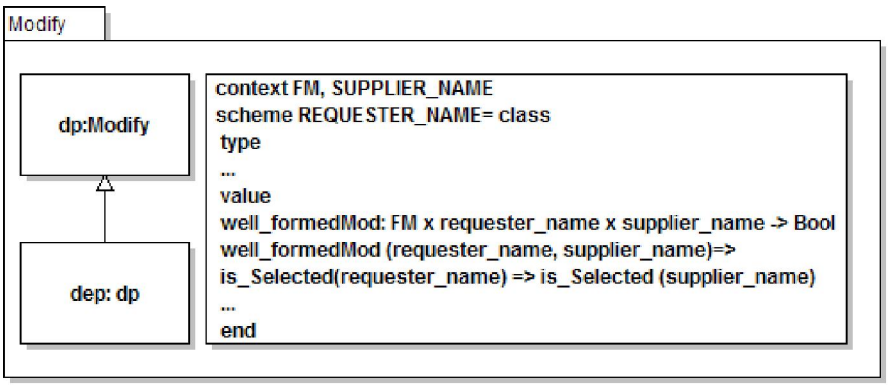


Figura 4.45 Transformación Modify

El caso de estudio propuesto no registra relaciones modify.

Una de las características claves del enfoque MDD es la noción de mapping. Hemos definido un conjunto de reglas de transformación con el objetivo de generar automáticamente el modelo destino de especificaciones iniciales RSL a partir del análisis de un conjunto de features y sus relaciones. Con respecto a especificaciones iniciales nos referimos a la estructura que tiene un módulo RSL y a su contexto (relaciones de uso-herencia) y a la forma que tomará en cuanto a los agrupamientos que tiene el modelo origen.

Capítulo 5

Conclusiones

Los métodos formales han alcanzado un uso masivo en la construcción de sistemas reales, ya que ayudan a aumentar la calidad del software y la fiabilidad. La principal ventaja del uso de este tipo de métodos como RAISE, es que pueden ser usados a lo largo de todo el ciclo de vida del desarrollo de software. Cuando se usan en etapas iniciales del proceso, ayuda a revelar ambigüedades, omisiones, inconsistencias, errores o interpretaciones erróneas que podrían ser detectados mediante pruebas costosas y en las fases de depuración.

Sin embargo, las dificultades de uso que acarrean los formalismos es un inconveniente especialmente en las primeras etapas que consisten en la captura del conocimiento de un dominio en particular. En este contexto y para contribuir a reducir esta brecha, en este trabajo hemos presentado una propuesta de integración de una fase de análisis de dominio con el método RAISE, a fin de especificar una familia de sistemas para producir aplicaciones cualitativas y fiables en un dominio, promover la reutilización temprana y reducción de los costos de desarrollo.

Para lograr el objetivo perseguido se ha trabajado definiendo un conjunto heurísticas y pasos que guían en el proceso de transformación entre modelos del dominio y especificaciones RSL del método RAISE. Se han representado los modelos del dominio mediante la técnica de Feature Modeling. Utilizar esta técnica permite capturar los aspectos variables y los aspectos comunes entre los distintos productos. Para ello, los modelos organizan el conjunto de features jerárquicamente mediante relaciones entre ellos:

- Relaciones entre un feature padre o composición y un conjunto de features hijos o subfeatures.
- Relaciones no jerárquicas: si el feature A aparece, entonces el feature B se debe incluir (o excluir).

Las heurísticas fueron creadas a fin de definir una transformación de features a schemes RSL sintácticamente correctos.

El Model Driven Development, conocido como MDD, constituye un nuevo paradigma de desarrollo de software. MDD hace hincapié en el uso de modelos en el ciclo de vida de desarrollo de software y argumenta automatización a través de la ejecución del modelo, transformación de modelos y técnicas de generación de código. Una de las principales características de este paradigma es la noción de transformaciones automáticas que describen como un modelo origen (modelo source) se puede transformar en uno o más modelos destino (modelo target). Con el fin de adaptarse a nuestra propuesta de mejorar el proceso de desarrollo con el método RAISE dentro del paradigma MDD, se ha desarrollado una transformación ATL que permite la derivación automática de una especificación inicial RSL abstracta de un dominio a partir de un modelo de features. Se definieron los metamodelos para automatizar la transformación. En el capítulo 4 se definió el metamodelo FM y en el capítulo 3 se define el metamodelo RSL. Ambos metamodelos fueron definidos parcialmente, es decir con los elementos necesarios para establecer los mappings.

5.1 Resumen

Los siguientes puntos resumen nuestro trabajo:

- La construcción de modelos de análisis utilizando la feature-orientation, lo que permite organizar los requerimientos de un conjunto de aplicaciones similares en un dominio de software. En un principio se desarrolló una estrategia informal la cual comienza definiendo el modelo de features siguiendo una de las propuestas que facilita la construcción de los mismos: el Feature-Oriented Reuse Method (FORM).

- Transformación del modelo utilizando heurísticas: el modelo definido anteriormente es transformado por medio de un conjunto de heurísticas manuales en especificaciones iniciales RSL, para luego ser refinadas en más concretas.

- El desarrollo de parte del proceso de la transformación en lenguaje ATL. Con el objetivo de adaptar nuestra propuesta de la integración de análisis de dominio con el método RAISE dentro del paradigma MDD, hemos desarrollado la transformación ATL que permite la derivación automática de una primera especificación abstracta RSL de un dominio a partir de un modelo de features. Las reglas ATL definen cómo los features y las relaciones entre ellos son mappeados para producir la especificación RSL. Las normas siguen los principios propuestos en el método RAISE, por lo que la primera especificación es una especificación incompleta que todavía debe ser refinada a fin de

convertirse en una más concreta siguiendo los pasos que propone el método RAISE.

Distintos artículos [14], [15], [16] han publicado el trabajo anteriormente mencionado, según la evolución de esta tesis.

5.2 Contribuciones

La contribución principal de esta tesis es la integración de un método de Análisis de Dominio con especificaciones RSL. Se derivan de esta contribución principal, otras que merecen ser mencionadas:

- El trabajo basado en familias de sistemas en etapas tempranas del proceso de desarrollo de software asegura la calidad de un producto desde su proceso de diseño. La aplicación de los principios de las familias de sistemas resulta en mejoras tanto en la eficiencia como en la eficacia del producto.

- Logro de mayor uso de los métodos formales partiendo de especificaciones de familias de productos: las especificaciones formales soportan descripciones de las LPS y por lo tanto los aspectos relevantes de un dominio de aplicación particular.

- La reutilización de especificaciones formales: la reutilización de especificaciones en general es un proceso que debe realizarse cautelosamente, sin embargo, un dominio debidamente especificado, puede ser almacenado en repositorios de elementos pertenecientes a la misma línea de productos para su posterior uso en la especificación de otro, reduciendo así la fase de especificación en el proceso de desarrollo.

- Métodos formales en la comunidad de los FM: la aplicación de los métodos formales para tarea particulares en las LPS tiene una creciente y relevante literatura y herramientas [41]. Czarnecki en [8] usa formalismos para capturar la semántica de los FM, creando un metamodelo más rico e introduciendo el clonamiento. En [23] se resaltan los desafíos de la aplicación de métodos formales en las SPL proveyendo guías que ayudan a la elección de herramientas o técnicas formales apropiadas para distintos tipos de problemas.

5.3 Trabajo Futuro

El estudio de la incorporación de un método de análisis de dominio presentado en esta tesis se encuentra en continuo proceso de desarrollo. Presentamos en esta sección futuros trabajos y extensiones que se pueden aplicar al trabajo desarrollado aquí:

- Mejora de las estrategias definidas y utilización en distintas familias de una Línea de Productos de Software sobre la base de los resultados que se van obteniendo.

- Obtención de una versión más detallada de la transformación ATL, permitiendo así, mejorar el proceso de desarrollo en el marco de MDD siguiendo el proceso de desarrollo RAISE, y obteniendo especificaciones más concretas trabajando dentro de los modelos PIM.

- Los FM son modelos fáciles de entender y proveen una manera genérica para representar la variabilidad, independiente del dominio de aplicación específico. Se han definido varios proyectos independientes que utilizan la plataforma Eclipse / EMF cada uno su propio meta modelo para los FM. Aunque estos meta modelos tienen considerables diferencias estructurales, su semántica básica es similar. Se propone trabajar bajo esta plataforma con el metamodelo definido en esta tesis para avanzar en una profundización de la utilización de estos modelos.

- Análisis profundo de las herramientas que soportan FM, a fin de profundizar en el área del análisis de dominio y la creación de nuevos productos en el marco de las LPS.

- Traceability juega un rol crucial en MDD [24]. Se propone la implementación de un proceso de transformación que incorpora un mecanismo de traceability sencillo, basado en [51] mediante la creación de una relación de traceability entre el origen y el destino de los elementos del metamodelo correspondiente de acuerdo a cada regla de transformación. Se debe incorporar un soporte de trace a fin de mantener un historial de las transformaciones de manera que sea útil en el proceso entero de desarrollo.

BIBLIOGRAFIA

[1] Andersen, S; Holmslykke, S. Integration of UML-ised Formal Techniques and Tools with RSL and the RSL toolset. Master of Science Thesis. Technical University of Denmark, DTU, 2005.

[2] Antkiewicz, M and Czarnecki, K. Featureplugin: feature modeling plug-in for eclipse. In Michael G. Burke, editor, ETX, pages 67–72. ACM, 2004.

[3] ATL Transformation Language. Octubre, 2012. Disponible en: http://www.eclipse.org/atl/.

[4] Beuche, D. Modeling and building software product lines with pure:: Variants. In SPLC, page 358. IEEE Computer Society, 2008.

[5] Botterweck, G; Janota, M and Schneeweiss, D. A design of a configurable feature model configurator. In David Benavides, Andreas Metzger, and Ulrich W. Eisenecker, editors. Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain. Proceedings, volume 29 of ICB Research Report, pp 165–168. Universit¨at Duisburg-Essen, 2009.

[6] Brun, R. Técnicas de análisis de dominio: organización del conocimiento para la construcción de sistemas software. Actas del VIII Congreso ISKO. España. pp 195-202, 2007.

[7] Clements, P., Northrop, L. Software Product Lines: Practices and Patterns, Addison-Wesley, Upper Saddle River, NJ, 2002.

[8] Czarnecki, K; Helsen, S; Eisenecker, U. Formalizing cardinality-based feature models and their specialization, In Software Process Improvement and Practice, Vol. 10, No. 1, pp. 7-29, 2005.

[9] Czarnecki, K; Helsen, S; Eisenecker, U. Staged configuration through specialization and multi-level configuration of feature models, In Software Process Improvement and Practice Vol. 10, No. 2, pp. 143-169, 2005.

[10] Czarnecki, K; Kim, C.H. Cardinality-based feature modeling and constraints: a progress report, In Proc, International Workshop on Software Factories, OOSPLA'05. Disponible en: http://www.ece.uwaterloo.ca/kczarnec/sf05.pdf.

[11] Czarnecki, K; Antkiewicz, M. Mapping Features to Models: A Template Approach Based on Superimposed Variants. International Conference on Generative Programming and Component Engineering (GPCE'05). ACM SIGSOFT/SIGPLAN, 2005.

[12] Eisenecker, U.W and Czarnecki, K. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[13] FaMa Framework. Octubre, 2012. Disponible en: http://www.isa.us.es/fama/.

[14] Felice, L; Riesco, D Debnath, N; Montejano, G Using a Feature Model for RAISE specification reusability. IEEE- IRI 2005 ´International Conference on Information Reuse and Integration Proceedings de the 2005 IEEE International Conference on Information Reuse and Integration (IRI 2005) IRI - 2005, August 15-17, 2005. Las Vegas USA. IEEE Systems, Man, and Cybernetics Society 2005, ISBN 0-7803-9093-8. pp 306-311.

[15] Felice, L; Leonardi, MC; Mauco, MV; Montejano, G; Riesco, D and Debnath; N. A Strategy to Derive RSL Specifications from Feature Models. Proceedings 18th International Conference on Software Engineering and Data Engineering. SEDE 2009, Las Vegas, USA, 2009.

[16] Felice, Laura; Ridao, Marcela; Mauco, María Virginia; Leonardi, María Carmen. Using ATL Transformations to Derive RSL Specifications from Feature Models. Proceedings of The 2011 International Conference on Software Engineering Research & Practice. Volume I. pp: 273 – 279. H. Arabnia, Hassan Reza, Leonidas Deligiannidis (Eds.). WorldComp´11. Las Vegas. USA. ISBN:1-60132-199-6, 1-60132-200-3 (1-60132-201-1), 2011.

[17] Fey, D; Fajta, R and Boros A. Feature Modeling: A Meta-Model to Enhance Usability and Usefulneess. SPLC 2002. LNCS 2379 pp: 198-216. Springer- Verlag Berlin Heidelberg, 2002.

[18] George, C; Haff, P; Havelund, K; Haxthausen, A; Milne, R; Nielsen, CB; Prehn, S.; and Wagner, K.R. The RAISE Specification Language. Prentice Hall, 1992.

[19] George, C; Haxthausen, A; Hughes, S; Milne, R; Prehn, S and Pedersen, JS. The RAISE Development Method. BCS Practitioner Series, Prentice Hall, 1995.

[20] George, C. RAISE Tools User Guide. UNU/IIST, Macau, Research Report 227. Octubre 2012. Disponible en: http://www.iist.unu.edu/newrh/III/3/1/page.html.

[21] George, C. Introduction to RAISE. Technical Report 249, UNU/IIST, Macau, Research Report 249. Octubre 2012. Disponible en: http://www.iist.unu.edu/newrh/III/3/1/page.html.

[22] González-Baixauli, B; Laguna, M. MDA e Ingeniería de Requisitos para Líneas de Producto. II Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones (DSDM 05), pp 11-20, 2005.

[23] Janota, Mikoláš; Kiniry, Joseph; Botterweck, Goetz: Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines. Lero Technical Report Lero-TR-SPL-2008-02 School of Computer Science and Informatics, University College Dublin, Dublin, Ireland.

[24] Jouault, F. Loosely Coupled Traceability for ATL. Proceedings of the Traceability of the European Conference on Model Driven Architecture, Workshop on Traceability, ECMDA, Germany, pages 29-37, 2005.

[25] Kang, K; Kim, S; Lee, J; Kim, K; E. Shin, E; M. Huh, M: FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5, pp 143-168, 1998.

[26] Kang, K; Cohen, S; Hess, J; Novak, W; Peterson, A. Feature-oriented domain analysis (FODA) feasibility study, Technical Report, Carnigie Mellon University, Software Engineering Institute, CMU/SE-90-TR-21, 1990.

[27] Kästner, C; Thüm, T; Saake, G; Feigenspan, J; Leich, T; Wielgorz, F; and Apel, S. FeatureIDE: A tool framework for feature-oriented software development. In ICSE, pp 611–614. IEEE, 2009.

[28] Lau, S. Master's thesis, Dept. Electrical and Computer Engineering, University of Waterloo, Canada, 2006. Disponible en: http://gp.uwaterloo.ca/files/2006-lau-masc-thesis.pdf.

[29] Lee, K; Kang, K; and Lee, J. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering . Software Reuse: Methods, Techniques, and Tools: 7th International Conference. ICSR-7, Austin, TX, USA, 2002.

[30] Lee, K and Kang, K. Feature Dependency Analysis for Product Line Component Design. J. Bosch and C. Krueger (Eds.): ICSR 2004, LNCS 3107, pp. 69–85, 2004. © Springer-Verlag Berlin Heidelberg, 2004.

[31] Leffingwell, D and Widrig, D. Managing Software Requirements: A Unified Approach. Addison-Wesley, 2000.

[32] Mellor, S., Clark, A., Futagami, T. Model-driven Development. IEEE Software Vol. 20 Nº V, 2003.

[33] Mendonca, M; Branco, M Cowan, D. S.P.L.O.T Software Product Lines Online Tools. In Comapanion to the 24th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOSPLA 2009. Florida, USA, 2009.

[34] MOF: Meta Object facility (MOF ™) 2.0. OMG Specification formal. Julio 2012. Disponible en: www.omg.org/mof.

[35] MOSKitt Feature Modeler. Octubre, 2012. Disponible en: http://oomethod.dsic.upv.es/labs/.

[36] OCL. (2006). OCL: Object Constraint Language. Version 2.0. OMG. Julio 2012. Disponible en: www.omg.org.

[37] Ok, P; Sul, R; and George, C. A Management System for a University Library. Technical Report 186, UNU/IIST, Macau. Febrero 2012. Disponible en: http://www.iist.unu.edu/newrh/III/3/1/page.html.

[38] OMG Model Driven Architecture (2012). Junio, 2012, Disponible en: http://www.omg.org/mda/.

[39] Pohl, K; Böckle, G and van der Linden, F. Software Product Line Engineering: Foundations, Principles and Techniques. Springer, 1st edition, 2005.

[40] Pons, C; Giandini, R; Pérez, G. “Desarrollo de Software Dirigido por Modelos. Conceptos teóricos y su aplicación práctica”. Editorial: McGraw-Hill Education y Edulp. Marzo 2010. ISBN 9789503406304.

[41] Schaefer, Ina; Hähnle, Reiner: Formal Methods in Software Product Line Engineering. Braunschweig University of Technology, Reiner Chalmers University of Technology. Published by the IEEE Computer Society 0018-9162/11.

[42] Simos, M.A. Organization domain modeling (ODM): Formalizing the core domain modeling life cycle. In: Proc. Of the 1995 Symposium on Software reusability, Seattle, Washington. USA. ACM Press, pp 196-205, 1995.

[43] Software Engineering Institute (SEI). Octubre, 2012. Disponible en: http://www.sei.cmu.edu.

[44] Software Product Line (SPL). Octubre, 2012. Disponible en: http://www.sei.cmu.edu/productlines.

[45] UML. (2009). UML: Unified Modeling Language: Superstructure. Version 2.2. OMG Specification. Julio, 2012. Disponible en: www.omg.org.

[46] van Lamsweerde, A. Formal Specification: a Roadmap. In Proceedings of the Conference on The Future of Software Engineering, pp 147-159. ACM, 2000.

[47] Vranic, V.: Feature modeling based transformational analysis in multi-paradigm design. Computer and Informatics (CAI), 2003.

[48] Vranic, V.: Reconciling Feature Modeling: A Feature Modeling Metamodel. NODe 2004. LNCS 3263, pp. 122-137. Springer-Verlag Berlin Heidelberg, 2004.

[49] Vranic, V. Multi-Paradigm Design with Feature Modeling. PhD Thesis. Marzo, 2012. Disponible en: http://www2.fiit.stuba.sk/~vranic/.

[50] Vranic, V ; Snirc, J. Integrating Feature Modeling into UML. NODe/GSEM 2006: pp: 3-15, 2006.

[51] Winkler, S; von Pilgrim. J.: A Survey of Traceability in Requirements Engineering and Model-Driven Development. Software and Systems Modeling, Vol. 9, No 4. pp. 529–565, 2010.

ANEXO A

El Feature Model Electronic shop (eShop)

En este anexo se describe el Feature Model correspondiente a la familia de Electronic shop (eShop) [33]. La figura A.1 ilustra el nivel más alto del árbol que representa el modelo.

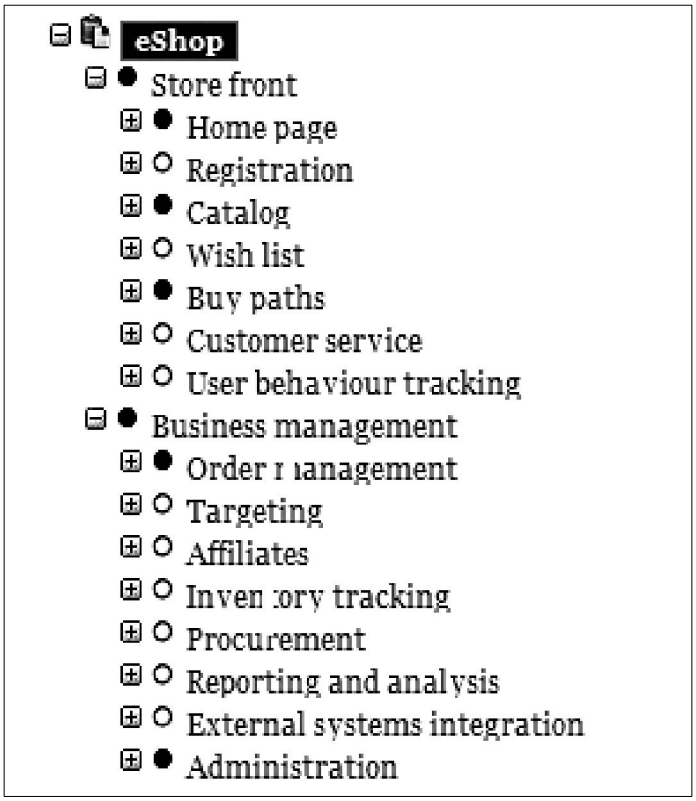


Figura A.1 Feature Model e-Shop

Los features descriptos en esta sección son los analizados como casos en las transformaciones descriptas en el capítulo 4.

1. Store front

El Store front es la interface que el cliente usa para acceder al e-Shop. El mismo consiste de un conjunto de features funcionales representando la Home Page, Catalog, y Buy path; siendo estos features de carácter mandatory. El Store front también puede incluir los features (de carácter optional) Registration, Wish list, Customer service y User behaviour Tracking.

1.1 Home page

Cada e-Shop tiene una Home page que sirve como bienvenida a la página. Es la página inicial que un cliente tendrá acceso cuando ingresa en el sitio. El e-Shop puede estar también configurado para redireccionar al cliente a la Home page si la URL apunta a una sesión que ha expirado o una página inválida o una página restringida. El contenido principal de una Home page es un Welcome message y los Featured products. La figura A.2 ilustra el árbol que representa este feature.

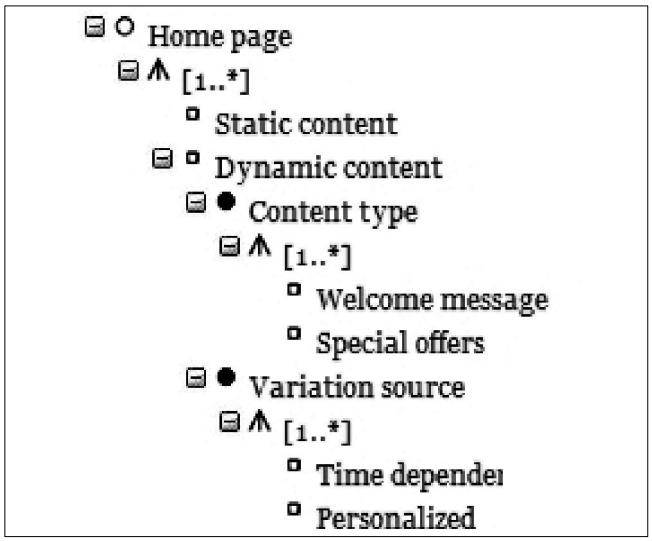


Figura A.2 Feature Home page

1.1.1 Static Content

El feature Static Content representa los contenidos de una home page con cambios poco frecuentes. En una implementación con contenido estático, los clientes observan

información estable de la home page, la misma se crea una vez, se almacena en el server y cualquier cambio requerirá la creación de una nueva.

1.1.2 Dynamic Content

El feature Dynamic Content representa los contenidos de una home page con cambios frecuentes. En una implementación con contenido dinámico, el contenido de la página es generado sobre demanda y cada cliente tiene una home page customizada en cada sesión. Dos parámetros se requieren para la customización: el content type y el variation source.

1.1.2.1Content type

Este feature describe qué elementos pueden ser dinámicamente generados. Los content type mas comunes son el Welcome message y special offers. Un welcome message es una bienvenida reproducida como texto. Special offers son promociones para el cliente, que incluyen por ejemplo descuentos sobre ordenes de compra. El soporte para special offers requiere la selección del feature Discounts; sin embargo, la selección del feature Discounts no tiene impacto sobre el feature Special offers.

1.1.2.2 Variation source

El feature Variation source representa la información que es usada para generar contenido. Los variation source mas comunes son time dependence y personalization. Time dependece genera contenido basado en el momento en que el cliente entra al e-shop. El momento se refiere a la hora del día, mañana o tarde, o a la época del año, verano o invierno. La personalización genera contenido basado en la información del cliente o información inferida. La información del cliente se almacena en su perfil. La información inferida son datos que no se proveen explícitamente, tales como el país desde el cual se visita la página.

1.2 Catalog

Un catalog contiene las cosas y/o servicios que un e-shop ofrece, de este modo, un e-shop debe soportar el uso de un catalog. Este provee un entorno para organizar la información para las cosas y servicios, que pueden influenciar la navegabilidad y usabilidad del e-shop. Debido a que cada e-shop ofrece productos, se debe definir un formato para la información del producto, el feature que describe estas características el

Product Information, siendo por lo tanto un feature de tipo mandatory. El resto de los features, que incluyen categoires, multiple catalogs, searching, browning, y custum views, son de tipo optional.

1.3 Buy paths

Buy paths es un agrupamiento de features relacionando el flujo de compras del cliente. Incluye acciones como mostrar los ítems y brindar la información de la orden de compra. Consiste de tres features de tipo mandatory: shopping cart, checkout y order confirmation. La figura A.3 ilustra el árbol de este feature.

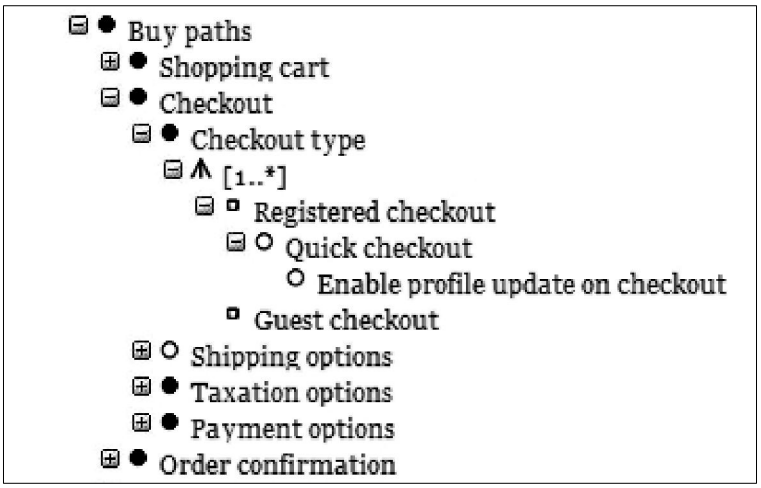


Figura A.3 Feature Buy paths

1.3.1 Shopping cart

El Shopping cart le permite al cliente realizar un seguimiento de los artículos que desea comprar durante su sesión. El carro (cart) contiene una lista de productos y cada producto está asociado con la cantidad que el cliente desea comprar. Ubicar un item en el Shopping cart implica el intento de hacer la compra, pero no la obligación para el cliente de completar la transacción. La selección del feature Shopping cart también requerirá que los siguientes features sean seleccionados: la política de manejo de inventario (inventory management policy feature) y el contenido del carro (cart conent page feature). Como features opcionales están el cart summary page y el cart saved after session.

1.3.2 Checkout

El feature checkout agrupa los features relacionados con el proceso de checkout. En el proceso, el cliente revisa los ítems que ha agregado a su shopping cart, ingresa su pago y la información de envío, selecciona los tipos de envío y las opciones de obsequios y confirma la orden. El proceso comienza cuando el cliente ha finalizado de seleccionar sus ítems y finaliza con el procesamiento de la orden. Checkout requiere que sean seleccionados (features de tipo mandatory): checkout type, taxation options, y payment options, en cambio no necesariamente necesita ser seleccionado el feature shipping options (feature de tipo optional). La figura A.4 ilustra el árbol del feature Checkout.

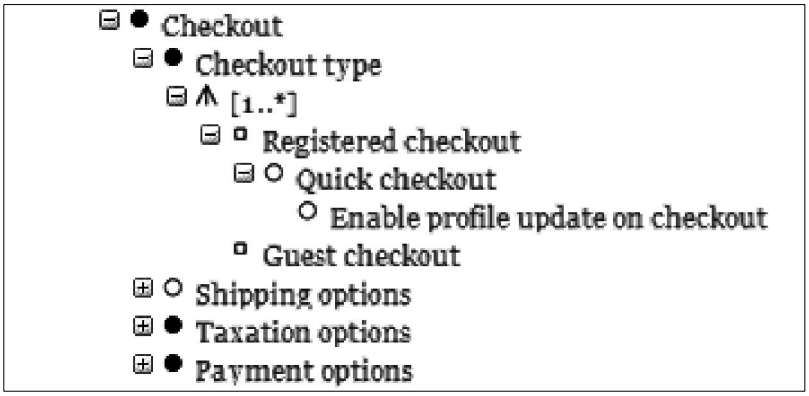


Figura A.4 Feature Checkout

1.3.2.1 Checkout type

Hay dos tipos de checkout: registered y guest. Un e-shop puede soportar los dos tipos simultáneamente, pero el cliente seleccionará que tipo de checkout usar durante la sesión.

1.3.2.1.1 Registered checkout

El registered checkout tiene por objeto hacer que los clientes se logueen antes que puedan comenzar el proceso de checkout. Durante el checkout, los clientes deben entrar o seleccionar su shipping y la información del pago de su orden.

1.3.2.2 Shipping options

El feature shipping options describe las opciones relacionadas al envío. Si se selecciona Shipping options (ya que es un optional feature), se requiere la selección de feature Shopping. No hay implicaciones sobre el feature shipping ya que el feature shipping options sólo determina el grado de control que el cliente tiene sobre el proceso de shipping.

1.3.2.3 Taxation options

El feature taxation options describe todas las opciones disponibles para aplicar las leyes de tax y calcular el monte del tax que debe ser cargado en una orden. El tax puede verse afectado por varias razones, incluso por la ubicación del cliente, la ubicación de la compañía del e-shop y de los ítems seleccionados.

1.3.2.4 Payment options

El feature payment options describe detalles pertenecientes a los pagos de la compra por parte del cliente. El feature payment types (de tipo mandatory) debe ser seleccionado ya que el e-shop no puede procesar ningún pago sin haber considerado este feature. Además las opciones de pago también pueden ser soportadas por los features fraud detection y payment gateways (ambos de tipo optional). La figura A.5 ilustra el árbol correspondiente.

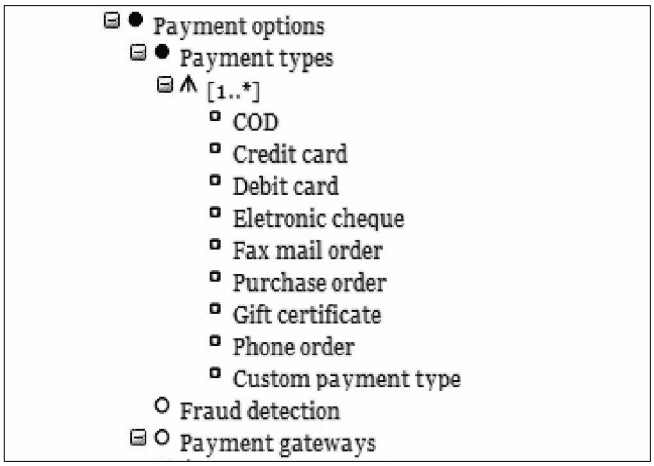


Figura A.5 Feature Payment options

1.3.2.4.1 Payment types

El feature payment types denota las formas de pago que pueden ser manejadas por el e-shop. Payment types puede incluir Cash On Delivery (COD), credit cards, debit cards, electronic cheques, fax mail orders, purchase orders, gift certificates, phone orders, y un custom payment type. El grupo es un grupo OR ya que el e-shop puede aceptar múltiples formas de pago para una sola orden. Por ejemplo, un cliente puede realizar una compra que excede el valor de un gift certificate; por lo tanto, se necesita otro método de pago para cubrir la compra.

1.3.2.4.2 Fraud detection

El feature fraud detection realiza los controles sobre la información del pago para la verificación de su autenticidad. Esto se logra a través de las autorizaciones de tarjetas y servicios de verificación. Este feature, si es seleccionado, requiere de experticia que está disponible generalmente a través de servicios externos, como los payement gateways.

1.3.2.4.3 Payment gateways

El feature payment gateways permite al e-shop tercerizar los servicios de pagos. Payment gateways son las terceras partes que manejan la verificación, la detección de fraudes y los arreglos de pagos con instituciones financieras. También este feature es un grupo OR y puede incluir Autorize.Net, CyberSource, LinkPoint, Paradata, SkipJack y VerisignPayflowPro.

1.4 Registration

Un e-shop puede habilitar la registración la cual permite que la información de un cliente sea solicitada, que persista y que sea reusada. Esto es conveniente para los clientes ya que no tienen que ingresar su información cada vez que desean hacer una compra. Además, esta información puede ser útil para crear estrategias de venta.

Figura A.5 Feature Registration

1.4.1 Registration enforcement

Si se habilita la registración, debe haber una política para determinar que acciones en el e-shop están restringidas a usuarios registrados. La figura A.5 muestra las tres políticas: Register to browse, Register to buy y None. Un e-shop puede estar configurado para soportar cualquier combinación de estos tres features cuando se construye el modelo, pero sólo una política tiene vigencia en tiempo de ejecución.

1.4.1.1 Register to Browse

Esta política restringe la navegación a los clientes registrados. Es la política más restrictiva. Hay varias maneras de definir los permisos de navegación. Una implementación de esta política restringe el acceso a todas las páginas a clientes registrados, pero permite a invitados a observar las listas de los productos.

1.4.1.2 Register to buy

Esta política requiere que los clientes se registren antes de hacer efectiva una compra. Esto se puede implementar requiriéndole al cliente loguearse antes de agregar un item a su carro o antes de comenzar el proceso de checkout.

1.4.1.3 None

Esta es una política sin restricciones. Cualquier visitante puede libremente navegar y realizar una compra en el e-shop sin pasar por el proceso de registración.

1.4.2 Registration information

El feature registration requiere que el cliente provea información acerca suyo. Esta información es almacenada en un perfil de cliente. Es un feature de carácter mandatory, sin embargo la mayoría de la información del perfil del cliente es de carácter optional. El único feature mandatory es el login credentials, que permite al cliente identificarse cuando hace un log in. El resto de los features son de tipo optional y pueden mencionarse: Shipping Address, Billing Address, Credit Card Information, Demographics, Personal Information, Preferences, Reminders, Quick Checkout Profile, Custom File.

1.4.3 UserBehaviourTrackingInformation

Este feature permite la e-shop asociar datos que guarda sobre acciones de usuarios a un perfil. La información adicional se puede usar para interpretar los datos desde una perspectiva de marketing. A pesar que es un feature de tipo optional, si es seleccionado, requiere la selección del feature user behaviour tracking;

1.5 Wish List

Una wish list es una vista del catálogo definido por el cliente. Le permite al cliente guardar registro de productos en el e-shop que desearía comprar o recibir como obsequio. Las wish lists también pueden ser usadas no sólo para guardar registro de los productos, sino también de sus precios. También es usada por el departamento de marketing para reunir datos de inteligencia empresarial para la orientación a los consumidores.

1.6 Customer service

El feature customer service contiene subfeatures que reflejan una mejora en las experiencias de compras de los clientes. Incluye features de carácter optional como questions and feedback forms, product returns, order status viewing, y shipment status trucking.

1.7 User behaviour Tracking

El feature user behaviour Tracking, permite al e-shop monitorear y registrar las acciones de un cliente mientras navega y realiza la compra. Este dato puede estar asociado con datos del perfil del cliente para analizar tendencias y comportamiento del consumidor. El feature user behaviour tracking requiere que el e-shop especifique que tipos de comportamiento son seguidos.
Un e-shop puede hacer un seguimiento de sus usuarios en forma anónima por sesión. Una sesión comienza cuando el visitante entra al sitio y finaliza cuando el visitante abandona el sitio. Los datos de una sesión constituyen el comportamiento de un simple usuario anónimo.

ANEXO B

Código ATL de las transformaciones

En este anexo se describe brevemente las construcciones más importantes del lenguaje de transformación ATL. Este lenguaje permite especificar la forma de generar un conjunto de modelos destinos a partir de un conjunto de modelos fuentes.

ATLAS Transformation Language (ATL)

ATL es un lenguaje de programación híbrido (imperativo y declarativo). Este lenguaje provee las construcciones que facilitan las especificaciones de los mapping entre el modelo fuente y el modelo destino. Las reglas que forman una transformación definen cómo elementos del modelo fuente son relacionados y navegados para crear los elementos del modelo destino. Está desarrollado sobre la plataforma Eclipse.

Un **módulo** ATL se corresponde a una transformación modelo a modelo. Los módulos permiten especificar la forma de producir un conjunto de modelos destinos a partir de un conjunto de modelos fuente. Los modelos fuente y destino de un módulo ATL deben satisfacer a sus respectivos metamodelos.

El metamodelo fuente (método FORM) se encuentra descripto en el capítulo 2, y el metamodelo destino (lenguaje RSL) se encuentra descripto en el capítulo 3.

Un módulo tiene un **header** definiendo los atributos vinculados al módulo de transformación, una sección **import** permitiendo importar librerías ATL, un conjunto de **helpers** similares a los métodos Java y un conjunto de **rules** que definen la forma en que son generados los modelos destino.

Helpers

Los helpers permiten definir código reutilizable que puede ser llamado en diferentes puntos en una transformación ATL. Al igual que un método, un helper se define mediante un nombre que corresponde al nombre del método, un tipo contextual que

define el contexto en el que el atributo está definido, un tipo de valor de retorno y un conjunto opcional de parámetros identificados por el par (parámetro, tipo).

Reglas

En ATL, existen dos tipos diferentes de reglas que corresponden a los dos modos diferentes de programación que provee ATL (declarativa e imperativa): los matched rules (la forma declarativa) y las lazy rules y called rules (la forma imperativa).

Las **matched rules** constituyen el núcleo de una transformación declarativa de ATL, éstas permiten especificar: 1) para cuáles tipos de elementos fuentes, los elementos destinos son generados y, 2) la forma en que los elementos destino son inicializados. Una matched rules comienza con la palabra reservada *rule*, que está compuesta por dos secciones obligatorias:

- Pattern fuente: se define después de la palabra reservada *from*, permite especificar las variables de los elementos del modelo que corresponden a los tipos de los elementos fuente que la regla vincula.y destino) y dos optativas (las variables locales y la sección imperativa).
- Pattern destino: se define después de la palabra reservada *to*, su objetivo es especificar los elementos que serán generados y cómo esos elementos generados son inicializados.

Las **lazy rules** se definen como las matched rules, pero son aplicadas únicamente cuando es llamada por otra regla.

Las **called rules** permiten utilizar programación imperativa. Se pueden considerar como un tipo de helper: tienen que ser explícitamente llamadas para ser ejecutadas y pueden aceptar parámetros. Sin embargo, a diferencia de los helpers, las called rules pueden generar elemento del modelo destino como las marches rules. Las called rules tienen que ser llamadas desde una sección de código imperativo, desde una marches rules o de otra called rules.

Al igual que las matched rules, las called rules se introducen después de la palabra reservada *rule* y también pueden tener una sección de variables locales. Sin embargo, como las called rules no tienen que vincular elementos del modelo fuente, no incluyen un pattern fuente. El pattern destino se introduce después de la palabra reservada *to*.

A continuación se detalla el código de las transformaciones ejemplificadas según las reglas descriptas en el capítulo 4. La misma consta de un solo módulo denominado *fm2rsl*.

Módulo fm2rsl

El módulo *fm2rsl* es un módulo simple que involucra varias ATL rules (matched rules y lazy rules) y también un conjunto de helpers. Utilizamos el metamodelo fuente FM de y el metamodelo destino RSL. La Figura B.1 muestra el código para los helpers.

```
module fm2rsl;
create OUT : rsl from IN : FM;
-- @path rsl=/FEATUREMODEL/rsl.ecore
-- @path FM=/FEATUREMODEL/FM.ecore

helper context FM!Feature def : getValueIMDef : String =
if self.isMandatory then
      self.name + ' isMandatory = TRUE'
else
      self.name + ' isMandatory = FALSE'
endif;


helper context FM!Feature def : getValueISelDef : String =
if self.isSelected then
      self.name + ' isSelected = TRUE'
else
      self.name + ' isSelected = FALSE'
endif;


helper context FM!Feature def: getValueISDef: String =
if self.isSolitary then
    self.name + ' isSolitary = TRUE'
else
      self.name + ' isSolitary = FALSE'
endif;


helper context FM!Require def: getSupplier: FM!Feature =
      (self.supplier );


helper context FM!Feature def: getUsed: FM!Feature =
if not self.isRequired.oclIsUndefined() then
    self.isRequired.supplier
else
      Set{}
endif;
```

Figura B.1 Helpers

```
helper context FM!Feature def: isExcluded(): Boolean =
if self.isExcluded.oclIsUndefined() then
   false
else
   true
endif;

helper context FM!Feature def: getContext: String =
if not self.parent.oclIsUndefined() then
    self.parent.name
else
      if not self.belongs_to.oclIsUndefined() then
          self.belongs_to.name
      else
            ''
      endif
endif;

helper context FM!Feature def:getAggr: String =
if not self.composite.oclIsUndefined() then
    (self.name)
else
      ''
endif;

helper context FM!GroupAssociation def: getType: FM!AssociationType =
      (self.Type);

helper context FM!AssociationType def: isOR() : Boolean =
if self.getType ='OR' then
      true
else
      false
endif;

helper context FM!GroupAssociation def: getContext: String =
      (self.name);


helper context FM!GroupAssociation def: getCard: Integer =
    (self.cardinality);


helper context FM!GroupAssociation def: getChild: Set(FM!Feature) =
      (self.Child);
```

Figura B.1 Helpers

```
helper context FM!AggregateAssociation def : getparts() :
Set(FM!Feature) =
if not self.parts.oclIsUndefined()then
      self.parts -> collect( elem | elem.isPartofAggregate(self.name))
  else
      Set{}
  endif;


helper context FM!Feature def: isPartofAggregate(name: String):
Boolean =
if (self.refimmediateComposite().oclIsUndefined() and
   self.refimmediateComposite().Type = 'aggregation' and
   self.refimmediateComposite().name = name) then
   true
else
   false
endif;
```

Figura B.1 Helpers

A continuación se detalla el código para las lazy rules necesarias (Figura B.2)

```
unique lazy rule Concept2Type {
from
      C: FM!Concept
to
      ST: rsl!SchemeType (
            typedef <- C.name ) }


unique lazy rule Feature2Object {
from
    F:FM!Feature
to
      SV: rsl!ObjectDeclaration (
            objectdef <-F.getAggr)
}

unique lazy rule Feature2ValueIsMandatory {
from
      F: FM!Feature
to
      SV: rsl!SchemeValue (
            valuedef <- F.getValueIMDef ) }
```

Figura B.2 Lazy rules

```
unique lazy rule Feature2ValueIsSolitary {
from
      F: FM!Feature
to    SV: rsl!SchemeValue (
      valuedef <- F.getValueISDef ) }

unique lazy rule Feature2ValueIsSelected {
from
      F: FM!Feature
to
      SV: rsl!SchemeValue (
            valuedef <- F.getValueISelDef ) }


unique lazy rule Feature2Obj {
from
      F: FM!Feature (F.isPartofAggregate())
to
      OD: rsl!ObjectDeclaration (
            objectdef <- F.name ) }
```

Figura B.2 Lazy Rules

La siguiente figura muestra el código para las matched rules necesarias para las transformaciones (Figura B.3).

```
rule Concept2Scheme {
from
      C: FM!Concept
to
      S: rsl!SchemeSpecification(
            name <- C.name,
            types <- thisModule.Concept2Type(C) ) }

rule fea2SchSpec {
from
      F:FM!Feature  (not F.isExcluded() )
to
      S: rsl!SchemeSpecification (
            name <- F.name,
            values <- thisModule.Feature2ValueIsMandatory(F),
            values <- thisModule.Feature2ValueIsSolitary(F),
            values <- thisModule.Feature2ValueIsSelected(F),
```

Figura B.3 Rules

```
            uses <- F.getUsed,
            context <- F.getContext)}


rule group2Sch{
from
      G:FM!GroupAssociation
to
      S:rsl!SchemeSpecification (
            cardinality <- G.getCard,
            name <-G.name)
}

rule aggr2Sch{
from
      A:FM!AggregateAssociation
to
      S:rsl!SchemeSpecification (
            name <-A.name,
            object <- A.getparts()->collect (feature |
thisModule.Feature2Obj(feature))
            )
}
```

Figura B.3 Rules