UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Desarrollo de Transformaciones de Modelos para la Derivación de Arquitecturas de Producto en LPS

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Jesús Benedé Garcés

Tutor: Cesar Emilio Insfrán Pelozo

2014-2015

Agradecimientos

Especialmente quiero agradecer toda la ayuda y tiempo dedicado a este trabajo a una gran persona, pero mejor amigo, que desde que tuve el honor y placer de conocer no ha hecho otra cosa que ayudarme, no sólo en el ámbito académico sino también a afrontar problemas y situaciones personales.

Por todo esto, y mucho más, gracias Javier, y gracias Rosa.

También quiero agradecer a mis padres por su apoyo incondicional y ánimos durante la realización de este trabajo.

A Emilio, por sus consejos, y su actitud ante las dificultades que han surgido durante la realización del mismo.

A Marc, una gran persona con la que he tenido el placer de compartir muchos momentos en estos cuatro años de carrera.

A todos ellos, gracias.

Resumen

Producir SW de calidad es uno de los mayores retos que encontramos en el ámbito del desarrollo de software y más aún en el desarrollo de líneas de productos, y uno de los aspectos clave, es la gestión de la variabilidad, definida como la habilidad de un sistema para ser cambiado o configurado para su uso en un determinado contexto. Esta variabilidad habrá de ser resuelta en tiempo de derivación, a la hora de obtener un determinado producto a partir de los activos de la línea de productos.

Usar el paradigma Desarrollo de Software Dirigido por Modelos (DSDM) promueve el uso de modelos de software, a distintos niveles de abstracción, como artefactos principales en el desarrollo de software, y permite que estos modelos puedan ser transformados sucesivamente hasta la obtención del producto final. Aplicar el paradigma DSDM permite automatizar el proceso de derivación, mediante cadenas de transformaciones de modelos, para la obtención de la arquitectura del producto final.

Palabras clave: Desarrollo de Software Dirigido por Modelos, variabilidad.

Abstract

Nowadays the development of high-quality software is one of the greatest challenges that can be found in software development area, and it is even more challenging in the Software Product Line Development (SPL). In this environment, one of the key facets is the variability management. Variability is the ability of a software system or software artifact to be extended, customized or configured for (re-)use in a specific context. This variability should be resolved at derivation time, when we want to obtain a specific product from a set of SPL assets.

The Model Driven Software Development (MDSD) paradigm promotes the use of software models (in different abstraction levels), as main artifacts in software development, allowing us to transform models multiple times until we obtain the final product. Applying MDSD paradigm will allow us to automatize the derivation process by means of model transformation chains for obtaining the final product architecture.

Keywords: Model Driven Software Development, variability.

Resum

Produir software de qualitat és un dels major reptes que trobem dintre del desenvolupament de software, i encara més, dintre del desenvolupament en línies de producte software. Un dels aspectes clau és la gestió de la variabilitat, definida com l'habilitat d'un sistema per a ser canviat o configurat per al seu ús en un determinat context. Aquesta variabilitat haurà de ser resolta en temps de derivació a l'hora d'obtenir un determinat producte a partir dels actius de la línia de productes.

Utilitzar el paradigma de Desenvolupament de Software Dirigit per Models (DSDM), promou l'ús de models software, a distints nivells d'abstracció, com artefactes principals en el desenvolupament de software, i permet que aquests models puguen ser transformats successivament fins obtenir el producte final. Aplicar el paradigma DSDM permet automatitzar el procés de derivació mitjançant cadenes de transformacions de models, per a la obtenció de l'arquitectura del producte final.

Paraules clau: Desenvolupament de Software Dirigit per Models, variabilitat.

Tabla de contenidos

Índice de figuras

Índice de listados

1 Introducción

Producir SW de calidad, tanto en el tiempo adecuado como con costes razonables, es uno de los mayores retos que encontramos en el ámbito del desarrollo de software y lo es de forma más acusada en el desarrollo de líneas de productos software (LPS). Una línea de productos es un conjunto de sistemas software que comparten un conjunto de características específicas y que se desarrollan a partir de un conjunto de activos software de un modo preestablecido (Clements & Northrop 2001).

El desarrollo de LPSs comprende la ingeniería de dominio, donde se desarrollan los activos software que integran la línea de productos, y la ingeniería de aplicación, que trata la derivación de los productos a partir de los activos software que conforman la línea de productos (Clements & Northrop 2001). Asegurar la calidad del producto es una actividad de gran importancia en el desarrollo de LPS, ya que un defecto en un activo software puede impactar negativamente en la calidad de todos los productos de una línea de producto.

Uno de los aspectos clave del desarrollo de líneas de producto es la gestión de la variabilidad. La variabilidad en el ámbito de la ingeniería del software se define como la habilidad de un sistema para ser cambiado o configurado para su uso en un determinado contexto. Esta variabilidad habrá de ser resuelta en tiempo de derivación, a la hora de obtener un determinado producto a partir de los activos de la línea de productos.

El uso del paradigma Desarrollo de Software Dirigido por Modelos (DSDM) promueve el uso de modelos de software, a distintos niveles de abstracción, como artefactos principales en el desarrollo de software, y permite que estos modelos puedan ser transformados sucesivamente hasta la obtención del producto deseado.

Gracias a la aplicación del paradigma DSDM, los requisitos no-funcionales (requisitos de calidad) pasarán a ser un artefacto activo más a ser tratado durante los procesos de transformación para la obtención del producto final.

1.1 Motivación

El desarrollo de LPS, es una aproximación para el desarrollo de sistemas software complejos, y que recientemente está adquiriendo gran importancia como medio para mejorar la productividad y la calidad del producto software en desarrollo. Uno de los pilares de esta mejora es la reutilización masiva y sistemática de *activos software,* (aquellos que forman la base de la línea de productos; arquitectura, componentes reusables, modelos de dominio, etc.), con el fin de crear sistemas que aúnan un conjunto común de características. Estos activos incorporan variabilidad en si mismos (variabilidad interna), que da servicio a la variabilidad externa (la variabilidad visible por los usuarios y clientes).

Dada la importancia que adquieren tanto la variabilidad, como la calidad producto dentro del proceso del desarrollo de software, se hace imprescindible disponer de herramientas, que nos ayuden a tratar estos factores, y que mediante procesos de transformación vamos a poder gestionar. Por ello, la arquitectura software juega un papel fundamental.

Las arquitecturas software son el medio para poder obtener los requisitos de calidad del producto software en desarrollo. Bass et al. (1998) define la arquitectura software de la siguiente manera:

> *"La arquitectura de software de un programa o sistema de computación es la estructura o estructuras del sistema, que comprende los componentes de software, las propiedades externamente visibles de esos componentes y las relaciones entre ellos".*

En el desarrollo de LPS, identificamos dos arquitecturas software que tienen dos roles diferenciados:

- *La arquitectura de línea de productos*, cuya función es dar soporte a todos los posibles productos que podemos obtener a partir de la línea de productos y que para ello utiliza mecanismos de variabilidad con el fin de adaptarse a los requisitos de los productos (Clements & Northrop 2001).
- La arquitectura de producto, derivada a partir de la línea de productos resolviendo la variabilidad presente en la arquitectura de la línea de productos, con el fin de cumplir los requisitos del producto en desarrollo (Clements & Northrop 2001).

El desarrollo de las transformaciones de modelos, van a permitir la derivación de la arquitectura software en entornos de desarrollo de líneas de producto software, con el fin de obtener una primera versión de la arquitectura de producto en desarrollo. Esta, será generada a partir de una configuración previamente establecida.

Dichas transformaciones se realizarán utilizando ATL, un lenguaje para la transformación de modelos. ATL (Jouault et al. 2006) es un lenguaje híbrido imperativo-declarativo, basado en OCL desarrollado por el grupo ATLAS en respuesta a la definición del estándar QVT de la OMG.

La variabilidad en los modelos arquitectónicos se va a soportar mediante el lenguaje estándar *Common Variability Languaje (CVL)* (Object Management Group 2012) y las transformaciones de modelos a desarrollar en el contexto de este trabajo final de grado van orientadas a la obtención de los modelos de resolución CVL que resuelven la variabilidad en los modelos arquitectónicos como paso intermedio para la obtención de los modelos arquitectónicos finales mediante un paso de transformación CVL.

1.2 Objetivos

Desarrollar, o construir un producto software que incorpore elementos que pueden variar en función de nuestros intereses presenta numerosos retos. Algunos de ellos pueden ser resueltos mediante la aplicación del paradigma de desarrollo de líneas de producto y de principios y técnicas del desarrollo de software dirigido por modelos. El objetivo principal de este trabajo fin de grado, es crear una transformación de modelos para la resolución de la variabilidad en modelos arquitectónicos, para automatizar la derivación de arquitectura software en entornos de desarrollo LPS.

Para una configuración de producto dada, se quiere resolver la variabilidad presente en los artefactos software, con el fin de obtener un modelo arquitectónico sin variabilidad. Se deben tener en cuenta tanto los requisitos funcionales como los requisitos no funcionales, por eso debemos hacer uso de herramientas de modelado y aprovecharnos de las facilidades que nos proporcionan.

Basándose en dichas premisas surgen los siguientes objetivos:

- **Desarrollar transformaciones con el fin de resolver la variabilidad de producto, que permita derivar automáticamente modelos de resolución CVL.**
  Usar cadenas de transformaciones con el fin de poder resolver la variabilidad de un producto software con variabilidad incorporada, cuyos artefactos afectan al producto final. Esta configuración se representa como una selección sobre el modelo de características, usando transformaciones de modelos. Para desarrollar estas transformaciones se utiliza ATL, un lenguaje imperativo-declarativo basado en OCL.
  De este modo, se muestra como los principios del DSDM ayudan a resolver de forma automática la resolución de la variabilidad en la arquitectura de la línea de productos.

- **Obtener los modelos de resolución CVL a partir del modelo de características.**
  Aplicar las transformaciones sobre una línea de productos de ejemplo, situada en el dominio automovilístico, con el fin de obtener los modelos de resolución que resuelven la variabilidad arquitectónica de los sistemas de control del automóvil, en base a las características seleccionadas en el modelo de características.

- **Introducir los Requisitos No-Funcionales (RNF), en los procesos de toma de decisión como medio para asegurar la calidad del producto en desarrollo.**
  Asegurar el cumplimiento de los requisitos no funcionales es otro de los aspectos clave que se deben conseguir para asegurar el comportamiento correcto del producto, y asegurar que se satisfacen las necesidades del cliente.

Una vez descritos los objetivos principales que se quieren conseguir, se identifican estos subjetivos:

- **Estudio del proceso de desarrollo basado en líneas de producto.**
  Conocer en que se basan los principios del desarrollo basado en LPS, y qué beneficios se pueden obtener al aplicarlo.

- **Estudiar y comprender la estructura y funcionamiento del lenguaje de transformación ATL.**
  Dado que se trata de un lenguaje desconocido para el desarrollador, estudiar su funcionamiento y entender su comportamiento es vital.

- **Dar a conocer herramientas, métodos y procesos, que nos ayuden a poder desarrollar reglas y transformaciones de forma correcta.**
  Uso de ATL + CVL junto con el framework QuaDAI (González-Huerta 2014), para poder trabajar con el modelo de características, y tener una herramienta de apoyo para el proceso de obtención del modelo de resolución.

El ejemplo tratado en este trabajo, se basa en un caso de estudio CarCArSPL (González-Huerta 2014). Dicho caso de estudio engloba conjunto de sistemas de control del automóvil, donde encontramos varios tipos de sensores diferentes y nodos de

computación de distintos fabricantes, donde cada uno tiene su sistema software. Con el fin de poder abarcar todas las posibles configuraciones, se aplica el paradigma de líneas de producto software.

Gracias al uso de una serie de transformaciones vamos a ser capaces de obtener un modelo de resolución para una de configuración de producto dada. Este modelo de resolución se generará de forma automática, como resultado de las transformaciones desarrolladas en este Trabajo Fin de Grado y actúa como medio para derivar la arquitectura del producto. Este modelo representa la resolución de la variabilidad de la arquitectura de la PLA, y nos permite obtener de 'forma rápida' la arquitectura de producto que se ajusta a las características del producto, asegurando que se cumplen las necesidades del cliente (Requisitos funcionales y no funcionales).

1.3 Estructura del documento

El resto del documento se estructura de la siguiente manera: La sección 2 presenta el marco tecnológico en el que se engloba este trabajo fin de grado. La sección 3 analiza los trabajos relacionados en el campo de la derivación de arquitecturas software. En la sección 4 se presentan las transformaciones ATL desarrolladas para resolver la variabilidad arquitectónica. En la sección 5 se ilustra la aplicabilidad de la propuesta mediante un ejemplo de uso de la industria automovilística. Finalmente, en la Sección 6 se presentan las conclusiones y trabajos futuros.

2 Marco tecnológico

En este capítulo, vamos a presentar los pilares tecnológicos sobre los que se asienta este trabajo: líneas de producto software, arquitecturas software, desarrollo de software dirigido por modelos, ATL Transformation Language y Common Variability Language (CVL).

2.1 Líneas de producto software

La definición más comúnmente aceptada de una LPS procede de (Clements & Northrop 2001):

> *"Se definen las líneas del producto de software como un conjunto de sistemas software, que comparten un conjunto común de características (features), las cuales satisfacen las necesidades específicas de un dominio o segmento particular de mercado, y que se desarrollan a partir de un sistema común de activos base (core assets) de una manera preestablecida".*

Esta definición aglutina cinco conceptos fundamentales de las LPS, a saber:

- "…un conjunto de sistemas software…": el objetivo de una LPS no es el desarrollo de un producto, sino el de un conjunto de productos. Debemos delimitar y preestablecer el alcance de este conjunto (*scoping*).
- "…que comparten un conjunto común de características (*features*)…": este conjunto de productos se caracteriza por medio de características. Una característica es una peculiaridad del producto que los clientes consideran importante para describir y distinguir entre los distintos miembros de la LPS. Durante el proceso de ingeniería de dominio se lleva a cabo un análisis que permite identificar las diferencias entre los distintos productos que integran la línea de productos (Clauß 2001). Estas diferencias se expresan en términos de características. Estas características suelen organizarse en modelos de características (feature models) (Kang et al. 1990).
- "…satisfacen las necesidades específicas de un dominio o segmento particular de mercado…": una LPS se desarrolla orientándose a un segmento de mercado concreto, ya que los productos intentan satisfacer las necesidades específicas de un segmento de mercado. De la habilidad para acotar e identificar correctamente este mercado, dependerá el éxito de la LPS.
- "…se desarrollan a partir de un sistema común de activos base (*core assets*)…": los productos son desarrollados a partir de un conjunto común de activos reutilizables, que conforman la base sobre la que se construye el producto y que en sí mismos incorporan variabilidad.
- "…de una manera preestablecida.": los productos se construyen de una forma preestablecida; no sólo hay unos elementos comunes, sino que la estrategia para construir el producto está perfectamente establecida de antemano a través de un plan de producción.

Estos cinco conceptos forman la esencia de lo que se entiende actualmente por desarrollo basado en LPS.

Los beneficios de aplicar las LPS son varios y nos pueden ayudar en el ciclo de desarrollo del producto, favoreciendo a una más rápida, económica y mejor calidad en la entrega de productos software.

El incremento de la calidad de los productos, va a permitir una reducción de las tasas de defectos, por lo que afecta directamente a dos puntos muy importantes: el tiempo de entrega del producto y los costes del mismo, viéndose afectados de forma positiva.

Otro de los aspectos positivos a destacar es que también podemos distinguir beneficios tácticos y estratégicos (Krueger, 2006); reducción en el tiempo promedio de creación y entrega de nuevos productos, así como en el número medio de defectos por producto, también se reduce el esfuerzo requerido para desarrollar y mantener los productos, el costo final de producción de los productos disminuye y el número total de productos que pueden ser efectivamente desplegados y mantenidos, aumenta.

En el desarrollo de líneas de producto software es clave la gestión de la variabilidad, entendida como "habilidad de un sistema software para ser cambiado, personalizado configurado para su uso en un determinado contexto" (van Gurp & Bosch 2002). Las características identificables por el usuario definen la *variabilidad externa* (Pohl et al. 2005) en el espacio del problema, en oposición a la *variabilidad interna* (Pohl et al. 2005) en el espacio de la solución. *Variabilidad interna* es la variabilidad existente en los propios activos software que permiten su configuración para dar soporte a la *variabilidad externa* de la línea de productos.

2.1.1 Ventajas respecto a la productividad y coste

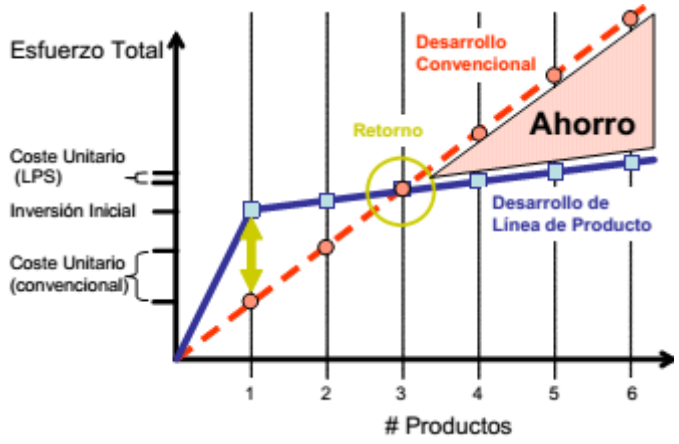


Figura 1. Desarrollo Convencional vs Línea de Producto (Garzás Parra & Piattini Velthuis 2010)

En la Figura 1, se puede apreciar la evolución y diferencias de utilizar un método de desarrollo convencional, a utilizar las líneas de producto. Si se aumenta el número de productos, también aumenta el esfuerzo y coste. Como analogía imaginemos dos productos software similares en el que el segundo es una copia del primero, pero, desde este momento la copia evoluciona de forma totalmente independiente al original. Como las dos aplicaciones son muy similares, se podría favorecer el desarrollo, pero no de la misma manera el mantenimiento, ya que cada una se mantiene de forma independiente incluso si el mantenimiento es similar para ambas. Por tanto, no explotamos las semejanzas entre las aplicaciones para reducir los costes asociados al mantenimiento.

Concluimos que, el desarrollo tradicional se centra en el producto, ya que cada producto tiene su mantenimiento. No estamos aprovechando las ventajas que tendríamos de poseer dos productos similares. El número de productos diferentes que somos capaces de gestionar es limitado.

Por otra parte, un entorno de líneas de producto está orientado a gestionar lo común y lo variable. Los beneficios de la reutilización aparecen y no de forma inesperada, ya que entra dentro de los objetivos de este enfoque. La incorporación de nuevas variantes se hace de forma sistemática y controlada, de forma que se agiliza tanto el desarrollo del producto como el mantenimiento. Reducir el coste del mantenimiento de un producto software, permite abarcar un número mayor de productos o variantes dentro de la línea de productos.

El enfoque o desarrollo basado en líneas de productos, realiza en primer lugar un estudio del dominio en el que se determinan las variables a soportar, y se desarrolla la plataforma LPS, antes incluso de realizar el primer producto. Gracias a esta forma de actuar, obtenemos información de interés, como podría ser la inversión o coste inicial y poder obtener el punto a partir del cual el uso de LPS resultaría rentable.

2.1.2 Ventajas relativas a la calidad

Los beneficios referentes a la calidad que aparecen al aplicar las líneas de producto se puede medir, de dos maneras distintas: la primera de ellas tiene en cuenta la precisión con que cada producto se adapta a las necesidades de cada cliente, y hay que tener en cuenta que esta medida depende del grado de variabilidad de línea de productos, ya que a mayor variabilidad la posibilidad de adaptar el producto a las necesidades del cliente es mayor.

Por otra parte debemos tener en cuenta que a mayor variabilidad el coste puede aumentar, por ello, es importante tratar de encontrar un equilibrio entre estos dos elementos.

La segunda manera de medir los beneficios que aportan las líneas respecto a la calidad es la tasa de defectos en los productos de LPS, donde los beneficios derivan de la reutilización de los elementos comunes, o *core assets*. La constante reutilización de estos elementos provoca que presenten un grado de depuración muy alto y por tanto detectar y solucionar o arreglar un defecto o problema en uno de estos elementos va a repercutir en todos los productos de la LPS.

2.2 Arquitectura software

Debido a la creciente complejidad que las aplicaciones están experimentando, junto con la aparición de nuevas tecnologías; como los sistemas distribuidos, o los sistemas abiertos y basados en componentes, es indispensable aislar los detalles de la implementación para poder interpretar correctamente el comportamiento de los elementos, y poder interactuar con ellos de forma adecuada.

Algunos de los objetivos que las arquitecturas software comprenden, son, entre otros ayudar a comprender y mejorar la estructura de las aplicaciones, además de permitir reutilizar dicha estructura (o partes de ella) para poder resolver problemas similares. Permitir la planificación de la evolución de la aplicación, identificando las partes

mutables e inmutables de la misma, así como los costes de los posibles cambios, también facilita el análisis de la corrección de la aplicación y su grado de cumplimiento respecto a los requisitos iniciales y además permiten el estudio de algunas propiedades específicas del dominio.

Pero, ¿de qué se ocupan las arquitecturas software?, en primer lugar, proporcionan un diseño o descripción de los elementos, es decir, muestran la organización a alto nivel del sistema, en los que se incluyen detalles como la descripción y análisis de las propiedades relativas a su estructura y control global.

Por supuesto, también han de relacionarse con la adaptabilidad a cambios y evolución, mantenibilidad, reutilización, composición, etc.

Desde la aparición del término *arquitectura software* en los años noventa, muchas definiciones han aparecido, pero la más aceptada es que la que propuso (Bass et al. 1998):

> *"La arquitectura de software de un programa o sistema de computación es la estructura o estructuras del sistema, que comprende los componentes de software, las propiedades externamente visibles de esos componentes y las relaciones entre ellos".*

Dado que la arquitectura software de sistema es un elemento complejo, no lo podemos describir de una forma simple. Es imprescindible documentar, es decir, debemos representar las vistas relevantes y añadir documentación que involucre a más de una vista, siendo este un tema que hoy en día todavía no tiene un consenso y presenta cierta controversia.

2.3 Desarrollo de Software dirigido por modelos

Entender y resolver problemas que cada vez son más complejos, mejorar el proceso de desarrollo de software, elevar el nivel de abstracción en la definición de los problemas planteados, y también en sus soluciones, etc. Son algunos de los objetivos principales que el *modelado* quiere ayudarnos a conseguir.

Por ello, se convierte en una herramienta fundamental en cualquier proceso científico o de ingeniería (Brambilla et al. 2012). Crear modelos que nos abstraigan los detalles más relevantes de un problema en concreto para poder analizar las posibles soluciones a dicho problema. Su uso en la ingeniería del software no es nuevo, de hecho se llevan utilizando mucho tiempo con la finalidad de documentar tanto la estructura interna del software como la estructura de los sistemas, interfaces o las estructuras de los datos asociados a los mismos.

El *Desarrollo de Software Dirigido por Modelos* (DSDM), se trata de desarrollar software a partir de modelos; con el correspondiente salto en el nivel de abstracción que esto conlleva. Abordamos el desarrollo de sistemas mediante refinamiento o la sucesiva transformación de modelos. Por esta razón, los modelos, 'evolucionan' ya que pasan de ser un mero elemento de documentación, a formar parte del propio software convirtiéndose en uno de los elementos determinantes para incrementar tanto la velocidad de desarrollo como aumentar la calidad del software obtenido (Stahl et al. 2006).

Usar transformaciones de modelos durante el desarrollo de un producto software, nos aporta múltiples ventajas:

- Fomento de la reutilización. Las transformaciones pueden ser utilizadas en sucesivas ocasiones para construir sistemas similares.
- La calidad obtenida en los modelos, aumenta.
- Se minimizan los errores en tareas de creación de un modelo a partir de otro.
- Encapsula el conocimiento de los expertos del dominio, facilitando la tarea a los desarrolladores, ya que de esta manera no necesitan conocer el proceso de transformación al detalle.

2.3.1 Model Driven Architecture

Con la finalidad de implantar la estructura anteriormente descrita, se hace imprescindible la implantación de estándares de modelado, metodologías, herramientas, etc. Que nos den soporte al ciclo de desarrollo permitiendo el uso de herramientas con capacidad de interactuar o interoperar entre ellas.

La iniciativa *Model Driven Architecture* (MDA), propuesta por la *Object Management Group* (OMG), establece una serie de estándares para dar soporte al desarrollo de sistemas independientes de la plataforma aprovechándose de las herramientas interoperables siguiendo el enfoque DSDM. Aunque MDA define varios y diversos estándares, se fundamenta en los siguientes:

- **Unified Modeling Languaje (UML)**, es el estándar de facto para el modelado de sistemas independiente del dominio.
- **Meta Object Facility (MOF)**, lenguaje común empleado para definir *metamodelos*, (descripción de la estructura que puede tener un modelo (Stahl et al. 2006); modelo de un lenguaje para expresar modelos (Favre 2004b), y representa el metamodelo a partir del cual se definen todos los lenguajes del modelado, e incluye UML.
- **Object Constraint Languaje (OCL),** se trata de un lenguaje declarativo y sin efectos colaterales, permitiendo la definición de restricciones como reglas en los modelos basados en MOF.
- **Query View Transformation (QVT),** lenguaje estándar para definir transformaciones en modelos basados en MOF.
- **XML Metadata Interchange (XMI),** mecanismo de persistencia para el almacenamiento e intercambio de modelos entre herramientas compatibles con MOF.

2.3.2 Transformaciones de modelos en DSDM

Este es uno de los puntos clave en el paradigma DSDM, ya que nos sirve como guía en el proceso de desarrollo, y gracias al cual podemos derivar código de la aplicación a partir de la especificación del sistema en modelos PIM, (*Platform Independent Model),* donde se ocultan los detalles de la plataforma de ejecución y permite utilizar la descripción del sistema con varias plataformas de tipo similar, para posteriormente en modelos PSM, (*Platform Specific Model),* en el que se combina la especificación definida en el PIM con los detalles concretos de la plataforma.

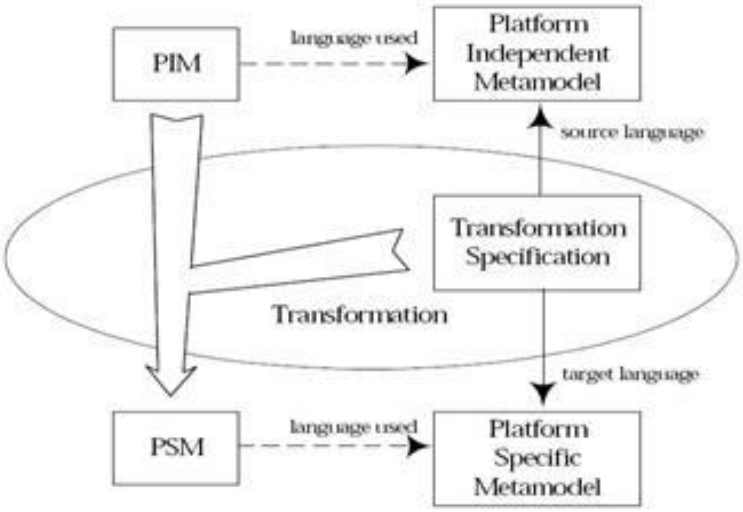


Figura 2. Transformaciones de PIM a PSM

En este proceso automatizado en que generamos un modelo destino a partir de uno origen, debemos usar reglas que especifiquen como un modelo, conforme a un *metamodelo,* se puede convertir en otro modelo.

Hoy en día los modelos están más y más presentes en gran parte de procesos de ingeniería. Sin embargo, en muchos existe poca documentación al respecto y es un motivo por el cual no acaban de estar altamente integrados en los procesos de ingeniería.

En contraposición a esta aproximación, el campo de MDE propone considerar los modelos como clases de primera entidad. También considera que las diferentes formas de manejar los recursos, (herramientas, repositorios, etc), pueden ser vistos y representados como modelos. MDE provee a los diseñadores y desarrolladores de modelos un conjunto de operaciones 'especiales' para la manipulación de los modelos.

En este contexto, la transformación de modelos parece ser una operación de gran relevancia a la hora de tratar los modelos: tiene como objetivo hacer posible la forma de especificar el camino de producir un número de modelos objetivo, basados en una serie de modelos de origen. Dentro del objetivo de MDE se asume que las transformaciones de modelos (como cualquier otra herramienta basada en modelos), puede ser modelada, es decir, que se debe considerar a sí mismo como un modelo.

El *metamodelo* define un lenguaje para especificar modelos, que describe los elementos que pueden ser usados en dicho modelo. Cada elemento que un modelador puede usar en su modelo está definido por el *metamodelo* del lenguaje. En UML por ejemplo, se pueden usar clases, atributos, asociaciones, estados, acciones, etc., porque en el *metamodelo* de UML hay elementos que definen qué es una clase, qué es un atributo, etc. Los *metamodelos* dentro de MDA son importantes por dos razones:

a) Un metamodelo define un lenguaje de modelado de forma tal que una herramienta de transformación pueda leer, escribir y entender los modelos.
b) La definición de transformación que describe cómo transformar un modelo en un origen en un modelo destino se basa en los metamodelos de ambos lenguajes.

La Figura 2, muestra la definición de una transformación entre un PIM y un PSM con sus correspondientes metamodelos.

2.4 ATLAS Transformation Language (ATL)

ATL es un lenguaje híbrido de transformación de modelos desarrollado como una parte de la plataforma AMMA (ATLAS Model Management Architecture). El entorno de programación, *Eclipse,* da soporte a ATL mediante una serie de herramientas: compilador, una máquina virtual, un editor y un *debugger*.

ATL, permite tanto aproximaciones imperativas, como declarativas (por eso se trata de un lenguaje híbrido), y éstas se pueden usar en las definiciones de las transformaciones dependiendo de las necesidades del problema. De forma declarativa, se pueden expresar los *mappings* entre los elementos origen y destino. Estos *mappings* pueden resultar complejos de declarar, por ello es necesario utilizar construcciones imperativas para lograrlo.

Hay que destacar que ATL únicamente soporta transformaciones unidireccionales, ya que los modelos origen solo se pueden leer y los modelos destino, son los que se modifican como resultado de dicha transformación.

2.5 Arquitectura del lenguaje ATL

La arquitectura del lenguaje ATL, consta de:

- Un 'Core', donde se describen los conceptos de ATL de forma abstracta,
- Un 'parser' y un compilador,
- Máquinas virtuales, que permiten realizar las transformaciones,
- Y un IDE, proporcionando funcionalidades como edición *debugger,* etc.

En la siguiente figura se muestra un esquema que describe los componentes ATL y su papel durante la ejecución de una transformación.

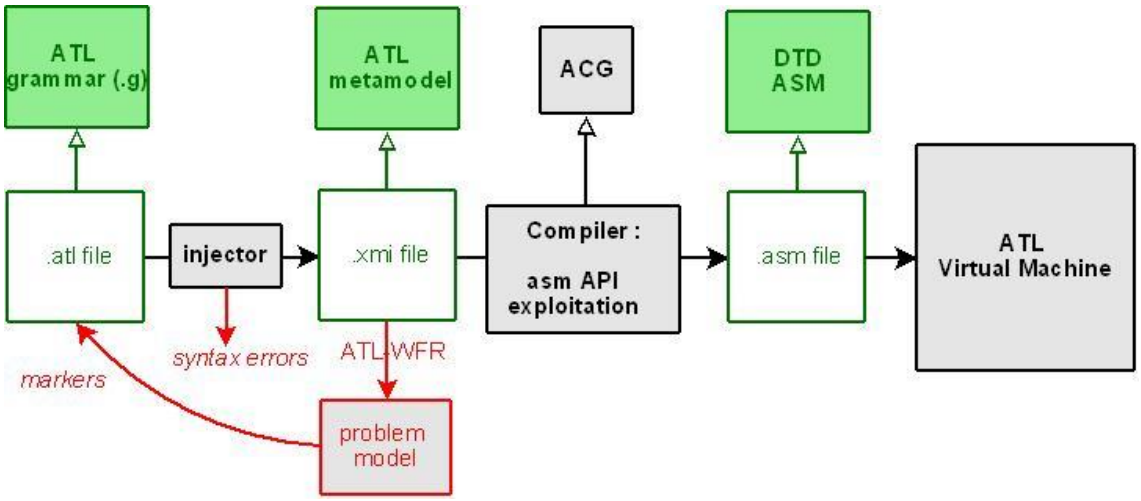


Figura 3. Componentes ATL

La máquina virtual de ATL está intercalada, o funciona de forma intercalada entre el compilador y los frameworks usados (EMF, MDR), permitiendo modularidad. De forma consecuente, cambios en el lenguaje ATL solo involucran al compilador de ATL.

2.5.1 ATL Core

La figura siguiente muestra la API presente en el núcleo de ATL y como interactua con herramientas como LaunchConfigurations o Ant tasks.

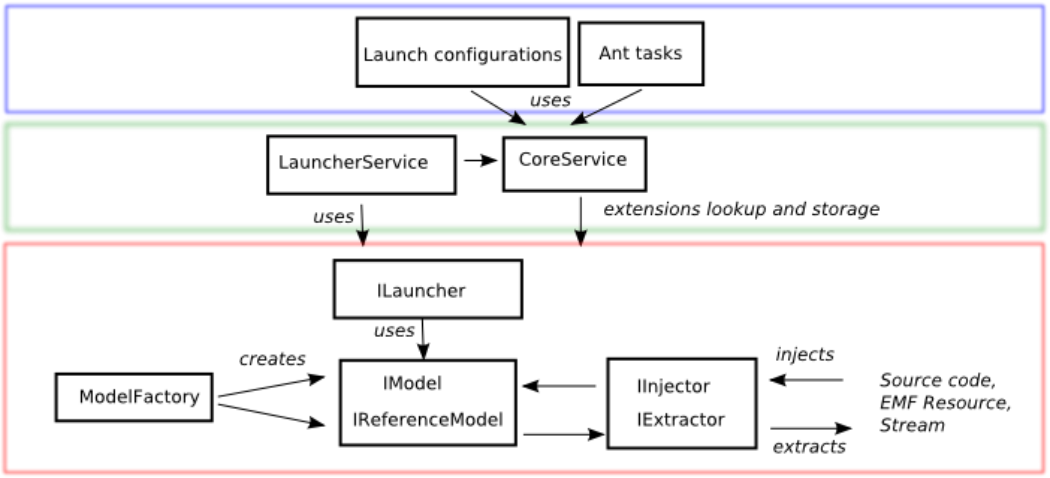


Figura 4. API presente en el núcleo ATL

- Un IModel es una representación de un modelo, que encaja en las trasnformaciones ATL. Provee métodos para 'vigilar' los elementos, crear nuevos, etc.
- La interfaz IReferenceModel extiende el IModel, y es una versión específica de un IModel que simboliza metamodelos. Define operaciones específicas en metamodelos, que son útiles en las transformaciones ATL.
- ModelFactory tiene como misión referenciar y crear modelos.
- Las interfaces, IInjector y IExtractor, proveen una forma de cargar y guardar modelos previamente creados por el modelFactory.
- La interfaz ILauncher tiene que ser implementado en (o por las) las máquinas virtuales: define métodos para parametrizar y lanzar transformaciones.

2.5.1.1 Servicios ATL

Con el fin de simplificar el uso del núcleo de ATL, y reducir la duplicación de código, se proporcionan dos servicios: CoreService y LauncherService.

Gracias al primer servicio, podemos buscar entre la extensiones disponibles dentro de Eclispe diferentes implementaciones para el Core. Gracias al LauncherService, podemos realizar una transformación a partir de una serie de parámetros. Este servicio está relacionado con la posibilidad de poder crear nuestras propias configuraciones para ejecutar transformaciones.

2.5.2 Máquina Virtual ATL

La máquina virtual de ATL es un *byte code* interprete que trabaja con OCL y ATL. (Object Management Group 2012). Esta especificación consiste en una precisa descripción de las funcionalidades de la máquina virtual, aunque no describe la implementación. La idea es que cualquier desarrollador pueda crear una máquina virtual ATL en cualquier lenguaje. La librería Nativa engloba todos los tipos de definiciones usadas por la máquina

virtual de ATL: tipos de OCL y tipos específicos de ATL. Ambos están definidos al mismo nivel, y usan reflexión. OCL aparece varias veces en la arquitectura ATL:

- En la implementación nativa de la librería.
- OCL está presente en los metamodelos ATL, ACG y TCS.

El siguiente esquema muestra la máquina virtual de ATL en funcionamiento:

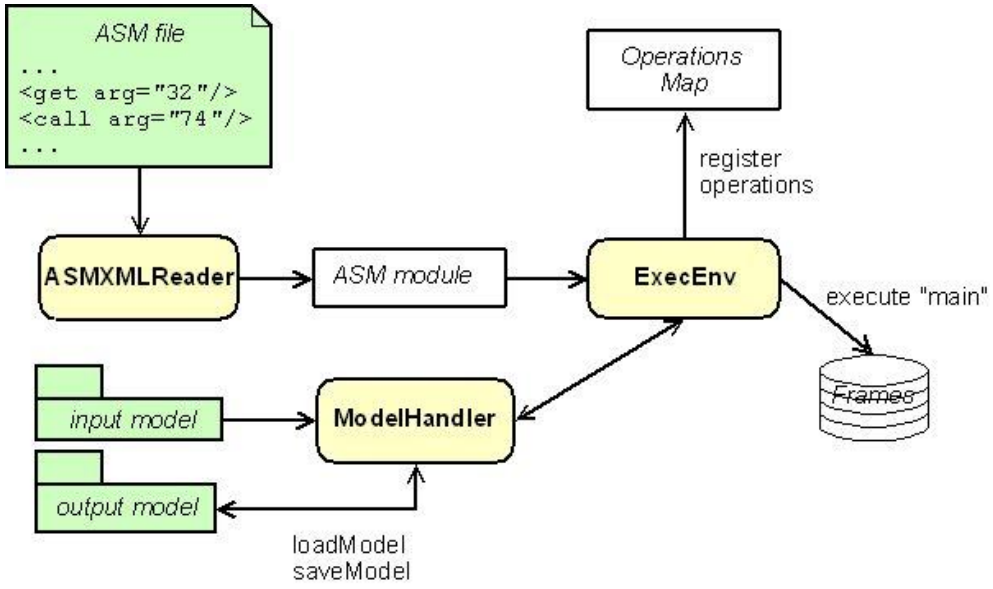


Figura 5. Máquina virtual ATL en funcionamiento

Durante la inicialización de la máquina virtual, cada operación se registra en el denominado 'Operations Map'. La clase *ExecEnv* es la que contiene el entorno de ejecución virtual. Este, trata con el 'mapa de operaciones' que se encarga de registrar todas las operaciones que son usadas en la transformación. Contiene toda la información usada, obtenida a partir de la ejecución, por ejemplo los modelos, y se recrea para cada ejecución. Las operaciones se ejecutan de forma secuencial en bloques, de acuerdo a su tipo. Por ejemplo, en ATL, una llamada al método append(), invoca directamente en el mapa de operaciones el método correspondiente en la clase ASMSequence.

Estos bloques pueden almacenar y también lanzar mensajes de error. ASMStackFrame, apoya a los métodos ASM, cuando la clase StackFrame, 'está ocupada' atendiendo a los métodos nativos. Los errores de ejecución provenientes de la máquina virtual cuando el método Frame.printstacktrace es invocado. En la actualidad hay dos tipos de implementaciones de la máquina virtual de ATL:

1. **Máquina virtual standard**: Se trata de la primera versión de la máquina virtual ATL. Esta implementación se abstrae del gestor de modelos presente en el framework. Esto se consigue gracias al uso de *model handlers*, que son una capa de abstracción dedicada a acceder al modelo. Este acceso, es implementado por dos clases: ASMModel y ASMModelElement.
   ATL contiene tres plugins que se corresponden con los diferentes Model Handlers: EMF, MDR y UML2. Cada uno de ellos implementa una clase abstracta:
   - ATLModelHandler, implementación de las tareas básicas de "newModel", "saveModel", "loadModel".

- ASMModel, implementación de métodos, como, getElementsByType, etc.
- ASMModelElement: Implementación de los métodos del tipo 'allInstances'.

Los modelos de entrada y de salida, se cargan en la misma API y se diferencian gracias a la propiedad 'isTarget'. Esta API implementa el método 'getMetaElementsByName', que se corresponde con la instrucción 'findme' de ASM.

Esta implementación de la máquina virtual, aún es usada en ATL, aunque presenta bastantes problemas de rendimiento, especialmente por la arquitectura presente en los *model handlers*.

2. **Máquina virtual específica. (EMF-specific VM):** Es una redefinición de la máquina virtual estándar que resuelve muchos de los problemas de rendimiento evitando el *wrapping* de EObjects.
   Su API permite considerar recursos EMF directamente como modelos, sin la necesidad de cargarlos previamente, tal y como se hace en la *Regular VM*.

2.5.3 Formato ASM

El lenguaje ASM es una clase de lenguaje ensamblador. Al tratarse de un lenguaje de bajo nivel, permite la modularidad con la intención de proveer las mayores facilidades a la hora de gestionar los modelos. Actualmente, el formato del fichero en el que se escribe código ASM es XML. De esta forma podemos *olvidarnos* de la sintaxis y *vigilar* solo en el *bytecode*. Un fichero ASM solo contiene nombres y constantes (en formato String). No hay referencias a ECore.

2.5.3.1 Serialization

Las transformaciones ASM se serializan con el fin de incrementar el rendimiento. La clase ASMXMLWriter se encarga de extraer el código ASM y de guardarlo en un fichero. La serialización se encarga de generar una lista de constantes ordenadas, en el fichero ASM, dónde las constantes, valores y llamadas a métodos se han resuelto. ASMWriter es la clase padre abstracta, que permite una implementación binaria que nos permite introducir y extraer código ASM.

2.5.3.2 Instrucciones

Todas las instrucciones se explican en la especificación de la máquina virtual de ATL, pero con el fin de comprender un poco más en detalle las mismas, se presenta un pequeño resumen de las más importantes:

- La instrucción getAsm retorna o devuelve el contexto en el que se encuentra el módulo ATL, por ejemplo, el 'thisModule', sería el equivalente en ATL.
- N simboliza un tipo nativo específico de ATL.
- TransientLink son links, o enlaces de trazabilidad.
- Todas las funciones del tipo 'getLinkBySourceElement', están implementadas en la nativelib.
- Respecto a las creaciones de objetos:
  - La instrucción 'new' toma 2 parámetros: el nombre del metamodelo y el tipo de clasificador. Luego, crea un elemento de este tipo en el modelo de salida (solo se permite crear un elemento). Los parámetros no están disponibles, en bytecode ya que se sacan de la pila antes de llamar a la instrucción.

- Podemos darnos cuenta, que ATL permite solo un modelo de salida, pero la máquina virtual de ATL, se puede extender (modificar) para poder permitir muchos otros.
- Una instrucción de eliminar, se puede implementar en la máquina virtual de ATL.
- ATL, proporciona el método, newInstance, que está directamente relacionado con el método model Handler. Este método no genera una nueva llamada dinámica. La principal ventaja es que este método se aplica directamente a la clase y no usa la pila de la máquina virtual de ATL para ello.

2.5.4 Intérprete o Parser

El *parsing* de ATL se realiza usando un parser definido en TCS, que devuelve como salida un modelo ATL conforme al metamodelo ATL. Posteriormente, una transformación ATL-WFR (interpretada por el motor o entorno), genera el 'modelo del problema'. Este modelo contiene errores a causa de la interpretación realizada por el parser, que se manifiestan en forma de *warnings* o marcas en el editor y que son visibles en cada compilación de cada fichero ATL. En caso que nos resultara de interés realizar esta interpretación de forma manual, deberíamos modificar la clase ATLParser. Hay que cercionarnos que el ATL parser, implementa las interfaces IInjector y IExtractor, se puede usar en ATL ant tasks con el fin de interpretar, o extraer ficheros ATL.

2.5.5 Compilador ATL

Actualmente existen disponibles dos versiones del compilador ATL. Las versiones 2004 y 2006, siendo ésta última la que utiliza ACG. La versión 2004 utiliza ATP, que es el predecesor de ACG. También podríamos compilar manualmente ficheros ATL, para ello, deberíamos editar la clase AtlDefaultCompiler.

2.5.5.1 ACG (ATL VM Code Generator)

ATL es un compilador DSL, que tiene como propósito facilitar la creación de un compilador que *se ajuste* a la máquina virtual ATL. Un compilador descrito con ACG genera ficheros ASM y contiene descripciones de instrucciones ASM para cada tipo de elementos de entrada obtenidos, o provenientes de un fichero compilado, por tanto, la entrada de este tipo de compilador, es un modelo que describe el contenido de un fichero compilado (por ejemplo, un fichero ATL).

Cuando un fichero ACG se compila, busca en el modelo de entrada usando el patrón de diseño visitador (Visitor pattern design). ACG, se inicia gracias a un fichero: ACG.acg, cuyo contenido describe el compilador ACG. Dado que los ficheros ACG describen de forma precisa instrucciones ASM, el fichero ACG.acg, lo podemos considerar trivial.

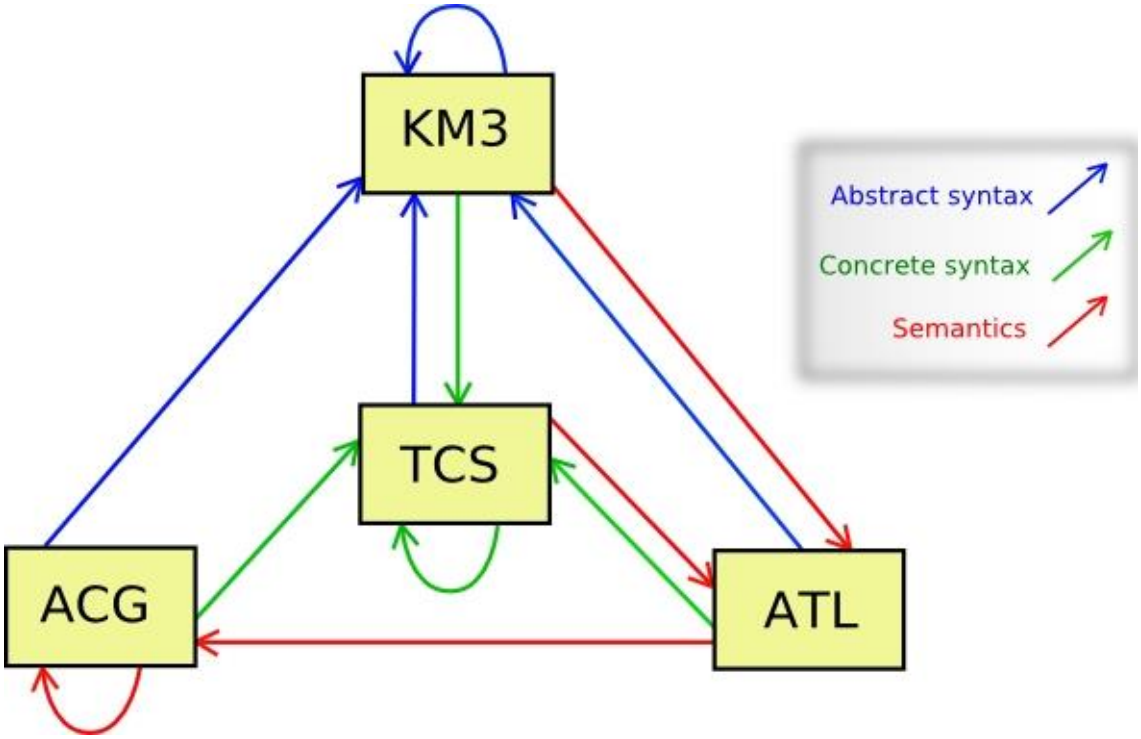


Figura 6. Situación de la plataforma AMMA

2.5.6 Modos de ejecución

Este lenguaje para la definición de transformaciones, presenta varios modos de ejecución, por una parte el **modo normal**, y por otro el modo **refinamiento** o ***refinning***. En el primero de los modos, debemos especificar completamente la forma en que cada uno de los elementos del modelo destino serán generados; mientras que en el modo refinamiento, solo se deben especificar las modificaciones a llevar a cabo entre el modelo origen y el destino. El resto de elementos son copiados automáticamente. Por ello, únicamente se emplean cuando el modelo origen y destino son "conformes" al mismo metamodelo.

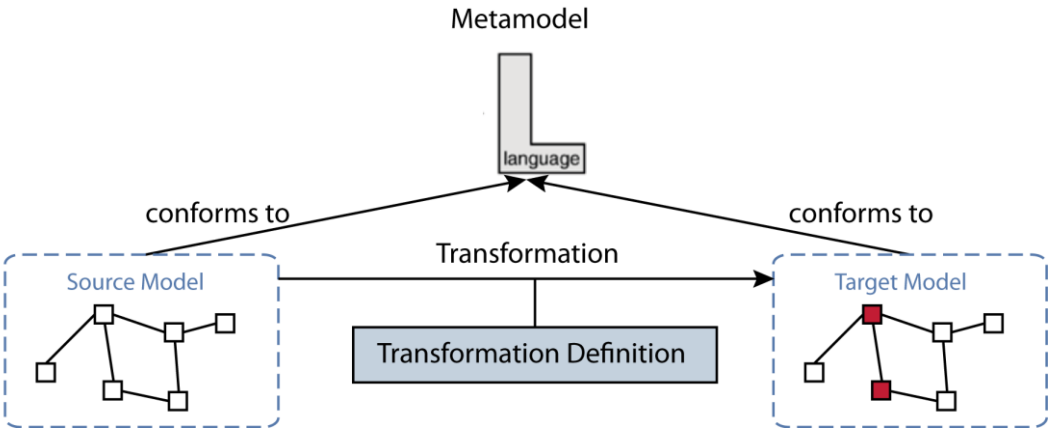


Figura 7. Modelo de entrada, metamodelo y modelo de salida

2.5.7 Tipos de Unidades ATL

Podemos definir tres tipos de unidades ATL en las que describir operaciones sobre modelos.

- **Query**: en los que se especifican un conjunto de consultas que permiten el cálculo de tipos primitivos de datos a partir de elementos de los modelos origen.
  - Son consultas ATL que permiten obtener un valor de tipo primitivo (*Boolean, String, Real, Integer*) desde un modelo.
- **Library**: ATL también ofrece la posibilidad de desarrollar bibliotecas independientes, que se pueden importar desde otras unidades ATL.
- **Module**: en los que se especifican las transformaciones como un conjunto de reglas definidas sobre los modelos origen y destino.

Estos tres tipos de unidades se definirán en ficheros que comparten la extensión .atl.

2.5.8 ATL Helpers

Podemos calificar a los *helpers* como el equivalente a los métodos Java. Gracias a ellos, podemos factorizar expresiones y pueden ser invocadas desde distintas partes de una transformación. Los *helpers* se definen mediante un nombre y un contexto de aplicación, además, pueden incluir parámetros y tienen un tipo de retorno.

```
helper [context context_type]? def : attribute_name :return_type
= exp;
```

El contexto de los helpers, se indica mediante el uso de la palabra reservada *context*, que define el tipo de elementos a los cuales el helper se aplica, es decir, el tipo de los elementos desde los cuales será posible invocar dicho helper. Debemos destacar que el contexto puede ser omitido en la definción de un helper. En ese caso, el helper es asociado al contexto global del módulo ATL. Esto quiere decir que para invocarlo, vamos a tener que utilizar la variable *self*.

Haciendo uso de la palabra reservada *def*, introducimos el nombre del helper. De la misma manera que su contexto, forma parte de la signatura del helper (junto con los parámetros y el tipo de retorno).

Un helper, puede aceptar un conjunto de parámetros que se indica mediante paréntesis a continuación del nombre del helper. La definición de un parámetro, incluye el nombre del parámetro y el tipo, tal y como se muestra a continuación.

```
parameter_name : parameter_type
```

Se pueden definir varios parámetros separándolos por una coma (“,”). El nombre del parámetro (parameter_name) es una variable identificador dentro del helper, motivo por el cual, cada nombre de parámetro debe de ser único. El tipo del contexto especificado, debe ser también el que tengan los parámetros y el tipo de retorno puede ser de cualquier tipo de dato que esté soportado por ATL.

EL cuerpo del helper se especifica como una expresión OCL, tal y como se muestra a continuación:

```
helper def : averageLowerThan(s : Sequence(Integer), value :
Real) : Boolean =

      let avg : Real = s->sum()/s->size() in avg < value;
```

Aquí, lo que se ha hecho, es definir un helper que tiene como nombre *averageLowerThan*. Este está dentro de un módulo ATL y como no detectamos la existencia de la etiqueta *context* en su definición, su contexto será el módulo ATL en el que se encuentre.

Este helper devuelve un tipo de dato booleano cuyo valor, dependerá de si la media de los valores contenidos en una secuencia de integers (parámetro s), es estrictamente menor que un valor real (que es el valor del parámetro que se le pasa). El cuerpo del helper consiste en una expresión "let", que define y inicializa la variable *avg*. Esta variable es comparada con el valor de referencia.

Por último, comentar que varios helpers pueden tener el mismo nombre, pero se van a diferenciar porque deben tener signaturas distintas, de esta manera son diferenciables por la ATL *engine*.

2.5.9 ATL Library

Gracias a las bibliotecas ATL, vamos a ser capaces de definir un conjunto de *helpers* que son invocados desde otras bibliotecas, módulos o consultas.

Estas bibliotecas no pueden ejecutarse de forma independiente, y los *helpers* definidos dentro de una ATL Library, deben estar explícitamente asociados a un contexto determinado.

```
library GeometryLib;

helper def: PIDiv180 : Real = 180.toRadians() / 180;

-- and some further geometric helper functions
```

2.5.10 ATL Module

Los módulos ATL, son las transformaciones modelo a modelo que presentan una estructura determinada, que se rige de la siguiente manera; (a) **Header**, donde definimos los atributos del módulo: nombre del módulo, metamodelos origen y destino, el modo de ejecución y si vamos a usar librerías ATL externas, (b) **Import**, se trata de una sección opcional en la que se permiten importar librerías ATL, (c) **Helpers**, gracias a los cuales podremos definir métodos y (d) **Reglas**, sección en la que se establecen el conjunto de reglas que definen como el modelo destino se obtiene a partir del modelo origen.

```
module module_name;
create outuput_models [from|refining] input_models;
uses extension_library_file_name;
```

2.5.11 Reglas ATL

Con las reglas ATL podemos definir cómo se transforman los elementos del modelo de entrada en los correspondientes elementos del modelo de salida, es decir, la

transformación deseada. En ellas podemos incluir condiciones, variables, asignaciones y otro tipo de sentencias.

Como ya hemos comentado en apartados anteriores, ATL es un lenguaje híbrido, por un lado, tenemos la parte declarativa y por otra la imperativa; podemos distinguir 3 tipos de reglas; *Matched Rules, Lazy Rules* y *Called Rules*.

2.5.11.1 Matched Rules

En la parte declarativa, ATL, nos proporciona las denominadas *Matched rules*, que se basan en *object pattern-matching*. Estas nos permiten expresar cómo se han de crear los elementos del modelo destino a partir de elementos del modelo origen. Se trata de reglas declarativas que se activan automáticamente mediante unificación (*pattern matching*) de los patrones del modelo origen.

Gracias a las *Matched Rules,* podemos especificar, las partes del modelo origen que deben unificar, el número y el tipo de los elementos del modelo destino generados y la manera en que esos elementos del modelo destino van a ser inicializados a partir del patrón que unifica en el modelo origen.

A continuación mostramos la sintaxis que una *Matched Rule* sigue. Podemos apreciar o diferenciar 3 partes:

- **From:** Refleja la sección del modelo origen.
- **To:** Se trata de la sección del modelo destino.
- **Do:** Se trata de la sección imperativa.

```
rule rule_name {
      from
            in_var : in_type [in model_name]? [(
                  condition
            )]?
            [using {
                  var1 : var_type1 = init_exp1;
                  ...
                  varn : var_typen = init_expn;
            }]?
      to
            out_var1 : out_type1 [in model_name]? (  bindings1   ),
            out_var2    :    distinct    out_type2    foreach(e    in
collection)(bindings2),
            ...
            out_varn : out_typen [in model_name]? (  bindingsn   )
      [do {
            statements
      }]?
}
```

2.5.11.2 Lazy Rules

Otros tipos de reglas que podemos encontrar en la parte declarativa, son las denominadas *lazy rules,* las cuales son invocadas explícitamente por otras reglas.

La sintaxis que estas presentan es muy similar a la anteriormente descrita, pero con una pequeña diferencia en la cabecera o definición de la misma.

Observamos también que las 3 partes anteriormente descritas, también las encontramos en la sintaxis de una *lazy rule*.

```
lazy rule rule_name {
      from
            in_var : in_type [in model_name]? [(
                  condition
            )]?
            [using {
                  var1 : var_type1 = init_exp1;
                  ...
                  varn : var_typen = init_expn;
            }]?
      to
            out_var1 : out_type1 [in model_name]? (  bindings1   ),
            out_var2    :    distinct    out_type2    foreach(e    in
collection)(bindings2),
            ...
            out_varn : out_typen [in model_name]? (  bindingsn   )
      [do {
            statements
      }]?}
```

2.5.11.3 *Called Rules*

En la parte imperativa de ATL, encontramos las reglas *Called*, que permiten generar documentos del modelo destino desde código imperativo. Han de ser invocadas desde bloques imperativos de otras reglas, y además, aceptan parámetros que han de ser especificados previamente (del mismo modo que en las invocaciones de los *helpers*).

```
[entrypoint]? rule rule_name(parameters){
      [using {
            var1 : var_type1 = init_exp1;
            ...
            varn : var_typen = init_expn;
      }]?
      [to
            out_var1 : out_type1 (
                  bindings1
            ),
            out_var2 : distinct out_type2 foreach(e in collection)(
                  bindings2
            ),
            ...
            out_varn : out_typen (
                  bindingsn
            )]?
      [do {
            statements
      }]?}
```

2.5.12 Tipos de datos en ATL

El esquema de datos definido en ATL es muy cercano, con el definido en OCL. El siguiente esquema nos muestra una vista de la estructura que siguen los tipos de datos en ATL. Los diferentes tipos de datos presentados en la Figura 8, representan posibles instancias de la clase *OclType*.

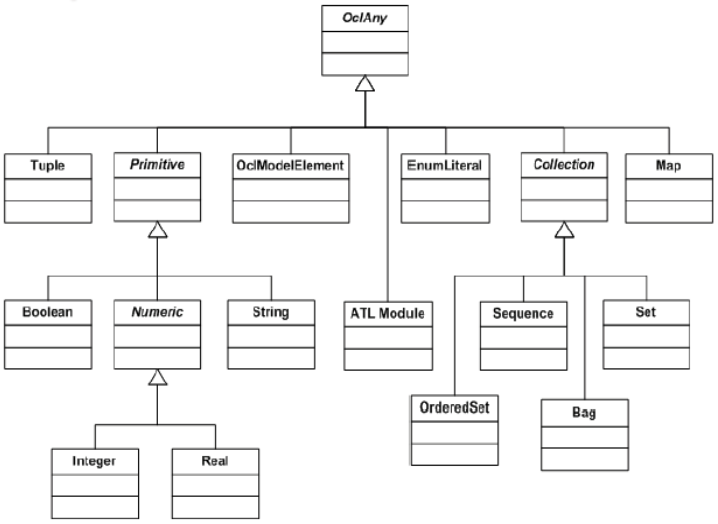


Figura 8. Tipos de datos en ATL

El elemento raíz de las instancias *OclType,* es el tipo abstracto *OclAny,* desde el cual cualquier otro tipo considerado, ya sea de forma directa o indirecta, hereda. ATL, considera 6 tipos principales de datos; primitivos, colecciones, tuplas, enumeraciones, el tipo *model element* y tipo *map* (debemos destacar que este tipo de dato, está implementado como un extra por parte de ATL, por eso no aparece en la especificación OCL).

2.6 El estándar CVL para la representación de la variabilidad

Common Variability Language o Lenguaje de representación de variabilidad común (CVL), es un estándar de la OMG para representar la variabilidad a nivel de modelos. Su finalidad es facilitar la representación y resolución de la variabilidad sobre instancias de cualquier modelo o lenguaje definido utilizando un *metamodelo* basado en Meta Object Facility (MOF). Este modelo, es el denominado *modelo base*.

En la Figura 9, podemos ver como las herramientas CVL y las DSL, (Domain Specific Language), se integran. *Calificamos* un lenguaje como Domain Specific Languaje, a cualquier lenguaje que esté especializado en modelar o resolver un conjunto específico de problemas. Este conjunto específico de problemas es el llamado dominio de aplicación o de negocio.

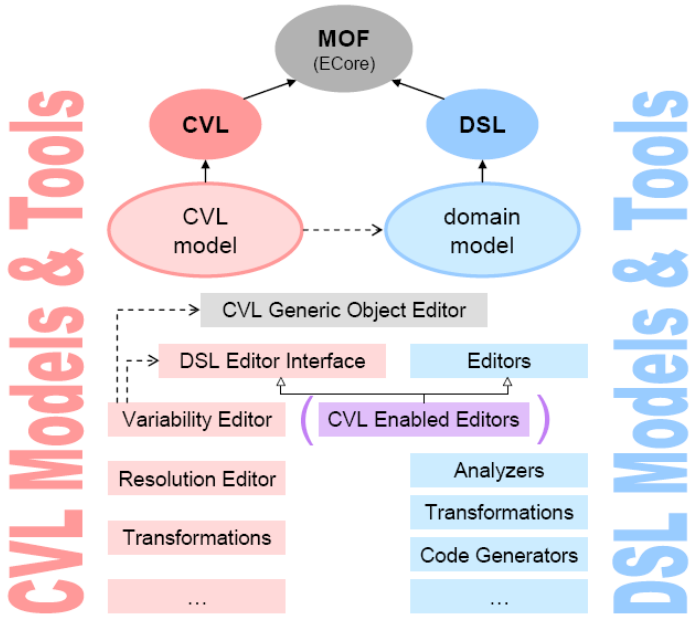


Figura 9. Herramientas CVL y DSL

En la parte derecha de la Figura 9, se muestra un DSL específico con una serie de herramientas asociadas al mismo. Podemos combinar el lenguaje CVL con otros lenguajes de propósito general que nos permitirá definir y resolver la variabilidad.

A nivel del modelo base, se especifica el modelo de variabilidad CVL y de esta forma, nos permite mediante el uso de modelos de transformación, (definidos en CVL), obtener de forma completamente automática modelos en el lenguaje del modelo base en los que la variabilidad ha sido resuelta.

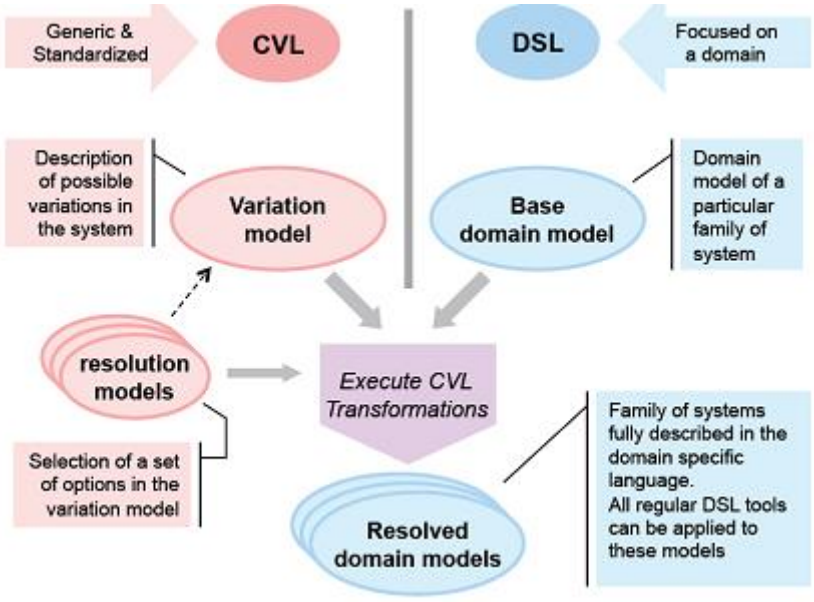


Figura 10 Especificación y resolución de la variabilidad mediante CVL

Como podemos observar en las imágenes anteriores, en CVL disponemos de tres modelos:

a) *Modelo base*: modelo descrito en DSL.
b) *Modelo de variabilidad*: Es en este modelo, donde definimos la variabilidad del modelo base.
c) *Modelo de resolución*: El modelo que define la manera de resolver el modelo de variabilidad, con el objeto de crear un nuevo modelo en el DSL base.

La obtención del modelo de resolución y materialización de la arquitectura de producto se definen de forma genérica a partir de la especificación disponible en el estándar. Trabajar en el estándar permite que los conceptos, artefactos y procesos sean adaptados de manera correcta y así asegurarnos ampliaciones futuras de manera más simple.

Con el fin de comprender en mayor profundidad la arquitectura CVL, se explican a continuación otras partes importantes que comprenden su estructura, representadas gráficamente en la Figura 11.

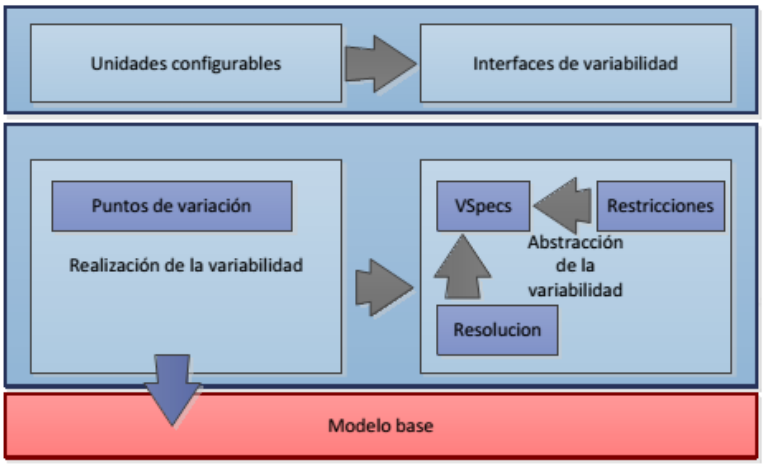


Figura 11. Arquitectura CVL

De la arquitectura CVL mostrada gráficamente, podemos distinguir varias partes:

a) *Modelo base*: modelo descrito en DSL. Desde el punto de vista de la arquitectura CVL, se trata de conjunto de elementos y enlaces entre ellos, expresado en un lenguaje de modelado distinto de CVL sobre el que se definirá la variabilidad.

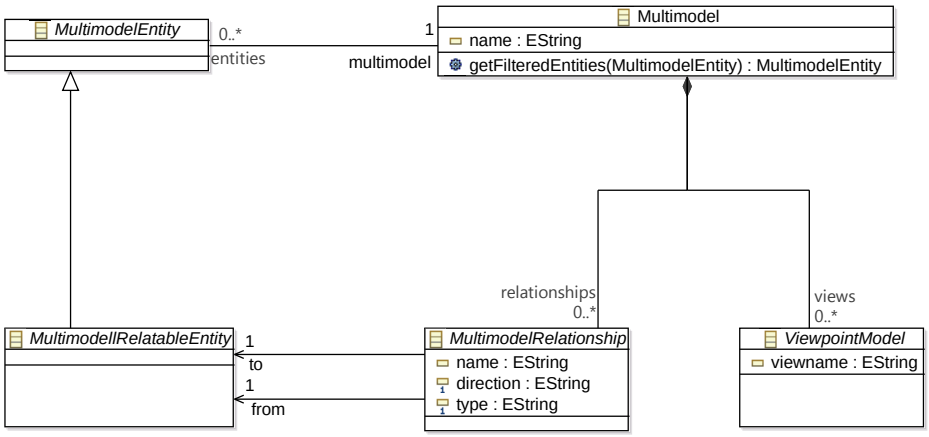


Figura 12. Metamodelo núcleo del multimodelo

b) *Abstracción de la variabilidad*: Es donde se ofrecen los constructos necesarios para especificar la variabilidad de manera abstracta. No se definen las consecuencias que dicha variabilidad va a tener sobre el modelo base, ya que aísla el componente lógico de CVL de las partes que modifican el modelo base. Para ello es necesario una entidad principal, que se corresponde con la especificación de variabilidad (VSpec), representando una opción binaria, un parámetro o un elemento de especificación que se puede invocar varias veces.

Dentro de este apartado podemos distinguir cuatro tipos de VSpecs. *Elección, variable, clasificador de variabilidad y especificación de variabilidad compuesta*. Cada una de ellas presenta las siguientes propiedades o características:

i. *Elección:* VSpec, cuya resolución necesita una decisión binaria y sus consecuencias no son conocidas a nivel de abstracción de la variabilidad.
ii. *Variable:* La resolución de esta VSpec, implica proporcionar un valor de tipo específico. Dicho valor se puede emplear en el modelo base, pero igual que en el caso anterior, desconocemos (en este nivel) donde y cómo se utiliza.
iii. *Clasificador de variabilidad:* Es un tipo de VSpec (VSClassifier); que instancia, provee y resuelve los sub-arboles de VSpecs para cada instancia.
iv. *Especificación de variabilidad:* (CVSpec). Hace referencia a los contenedores del modelo base, tratados como unidades configurables. Permite especificar una colección de declaraciones de variabilidad dentro de un contenedor del modelo base.

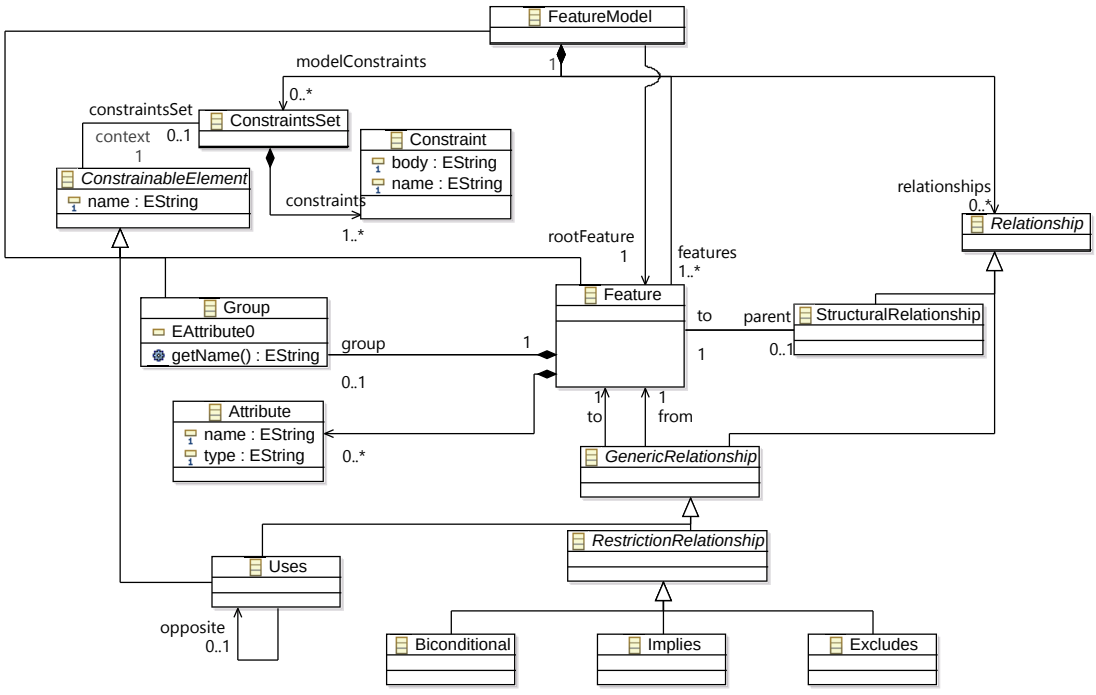


Figura 13. Extracto del metamodelo que da soporte a la vista de variabilidad (Gómez 2012)

c) *Realización de la variabilidad*: Provee de constructos con el fin de especificar los puntos de variabilidad sobre el modelo base. Un **punto de variación es una modificación aplicada al modelo base durante el proceso de materialización**. Estos puntos de variabilidad van a impactar directamente sobre el modelo base y hacen referencia al mismo mediante manejadores del modelo base. Estos puntos se asocian a las VSpecs con el fin de definir el grado de variabilidad abstracta que se va a materializar con dicho punto de variabilidad (formando un enlace entre la abstracción de la variabilidad y su realización, definiendo el efecto que va a tener el VSpec sobre el modelo base).
Estos mecanismos permiten la materialización de modelos a partir de la descripción CVL mediante la transformación del modelo base en modelos en los que la variabilidad ha sido resuelta.

Encontramos *cuatro* formas de definir las modificaciones realizadas al modelo base que se asocian a un punto de variabilidad:

i. *Existencia:* Especifica que la existencia de un objeto en concreto, enlace o valor en el modelo base, está en cuestión.
ii. *Sustitución:* Define el objeto del modelo que va a ser substituido por otro. Dicha substitución involucra dos objetos y presenta consecuencias tales como la necesidad de renombrar todas las conexiones en las que exista una relación objeto origen - destino. Cuando esto sucede, se elimina el objeto y se substituye por el nuevo.
iii. *Asignación de valor:* Indica el valor que puede ser asignado a un espacio específico de un objeto del modelo base.
iv. *Punto de variación opaco:* El punto de variación se asocia a variabilidad de dominio específico. Esta variabilidad se especifica empleando lenguajes de transformación de modelos como ATL.

Estos elementos anteriormente descritos, nos van a permitir definir variabilidad a partir de árboles de VSpecs y puntos de variación organizados de forma local sobre el modelo base utilizado. Están asociados al nivel más bajo en la arquitectura CVL como se puede observar en la Figura 11.

CVL también permite agrupar declaraciones de variabilidad CVL correspondientes a contenedores del modelo base mediante unidades configurables, estas unidades configurables exponen un interfaz de variabilidad permitiendo su configuración. Las unidades configurables, además de proveer de mecanismos de modularidad y reutilización, ofrecen la posibilidad de clonar y configurar varias veces una unidad configurable desde distintas partes de un mismo árbol de CVSpecs.

a) *Unidades configurables*: Podemos definir unidades configurables sobre los constructos de abstracción y realización de la variabilidad, permitiendo la especificación de componentes configurables y reusables. Proporcionan modularidad soportando la definición composicional jerárquica, reflejando la estructura composicional del modelo base. Estas unidades configurables, exponen una interfaz de variabilidad compuesta de VSpecs, que permite su configuración.

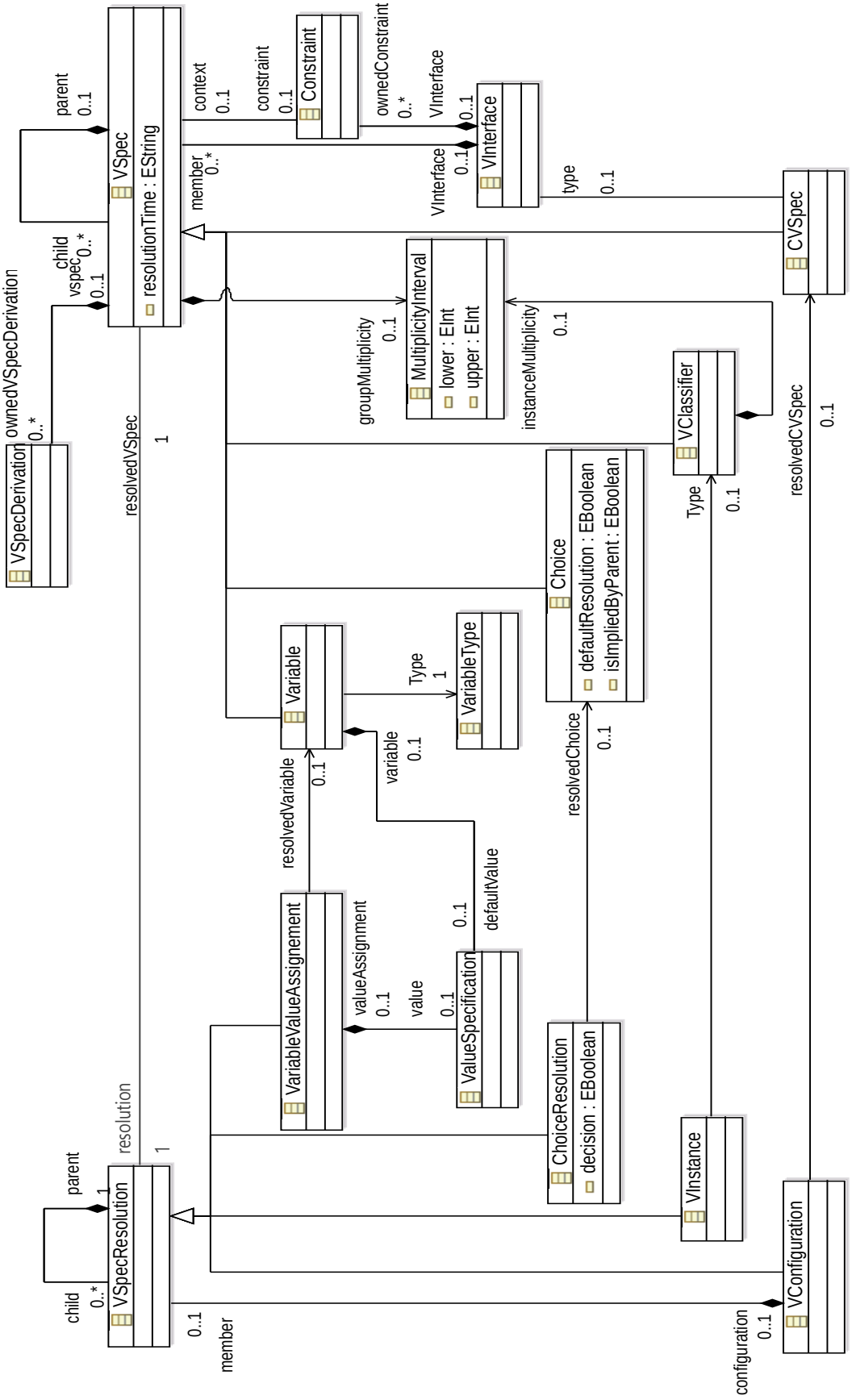


Figura 14. Extracto del metamodelo de CVL: VSpecs y VSpecResolution

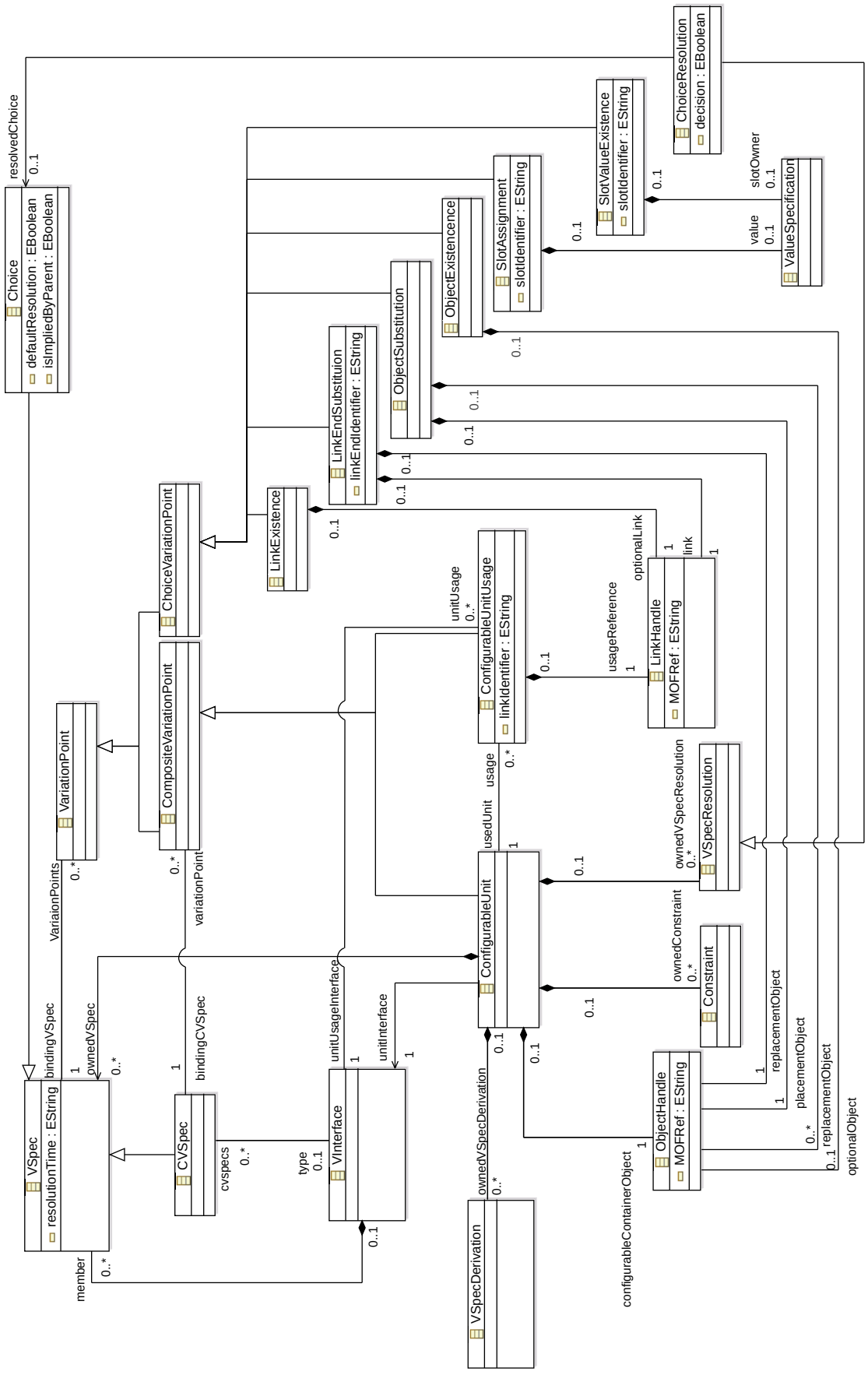


Figura 15. Extracto del metamodelo CVL, (CVSpecs y unidades configurables)

La interacción presente entre estos modelos resulta de gran interés; por una parte, el modelo base puede disponer de varios modelos de variabilidad y cada modelo de variabilidad puede tener varios modelos de resolución.

Una vez definidos de forma adecuada el modelo de variabilidad y el de resolución, podremos ejecutar transformaciones CVL genéricas de modelo a modelo para generar los nuevos modelos que se ajustan al DSL base.

La Figura 16 muestra el ciclo completo de materialización CVL. En la parte derecha, el modelo de decisión contiene las elecciones y los valores de las VSpecs de selección y los valores de las VSpecs variables.

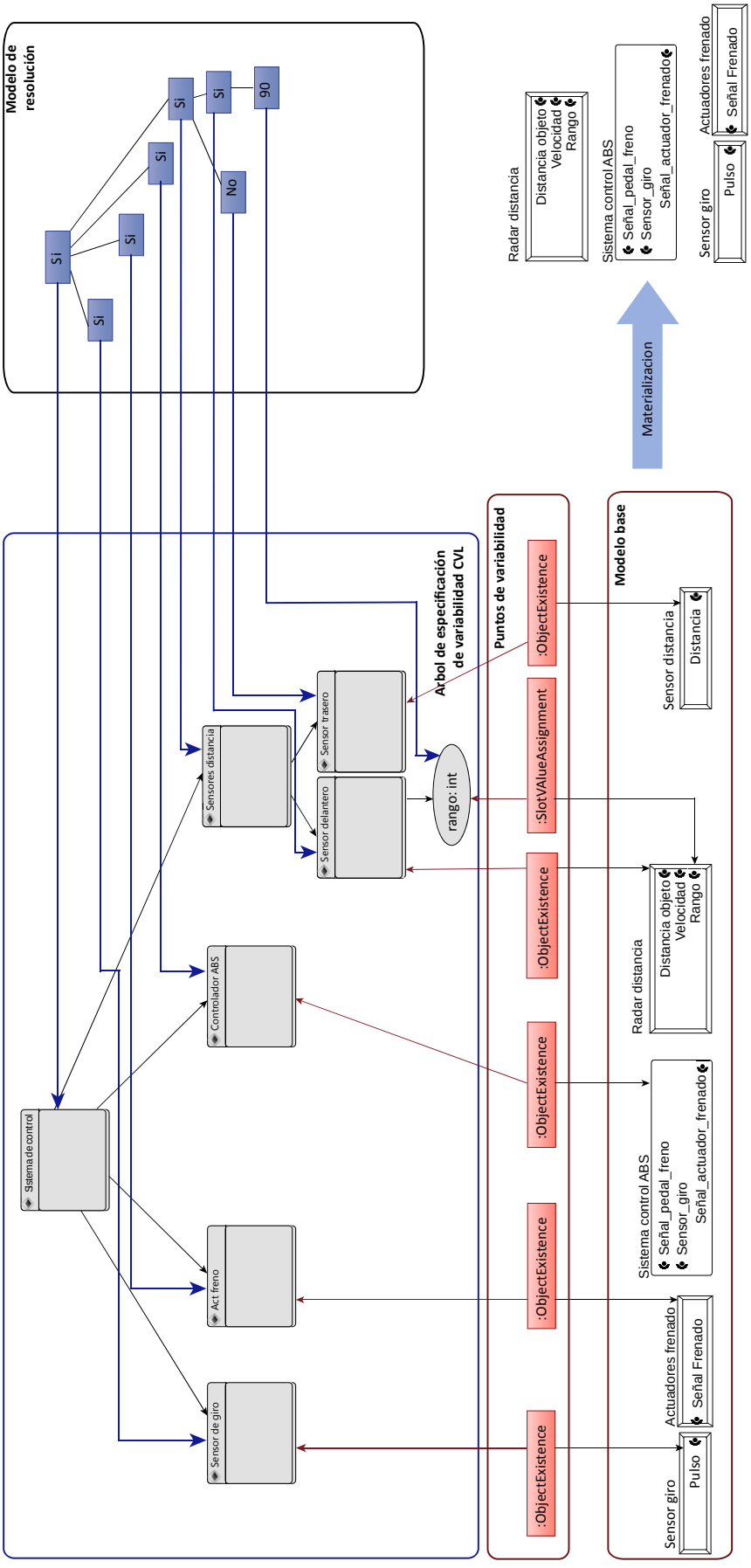


Figura 16. Árbol de variabilidad CVL, modelo de resolución y materialización (González-Huerta 2014)

2.7 Eclipse

Eclipse es una comunidad de código abierto, cuyos proyectos se centran en construir y desarrollar una plataforma donde podamos encontrar *frameworks, plugins,* herramientas y otro tipo de utilidades para desplegar y gestionar *software* a través del ciclo de vida.

El proyecto más relevante es *The Eclipse Platform,* el cual se puede integrar en los distintos entornos de desarrollo que se pueden usar para crear aplicaciones como páginas web, programas en JAVA, C++, y *Enterprise JavaBeans*.

The Eclipse Platform, permite descubrir, integrar y utilizar módulos llamados *plugins,* que una vez correctamente configurados e integrados con el IDE podremos hacer uso con el fin de aprovechar su potencial para poder desarrollar nuestras aplicaciones.

El papel principal de la plataforma es proveer herramientas con los mecanismos necesarios para usar las herramientas integradas. Estos mecanismos se muestran o son accesibles a través de APIs, clases, y métodos. También proporciona *frameworks* que facilitan el desarrollo de nuevas herramientas.

Esta modularidad es una ventaja a favor si comparamos con otros IDE's donde funcionalidad no se pueden usar o añadir.

Eclipse es de código abierto, flexible y de ámbito general. Para este trabajo nosotros mostramos interés especial en *Eclipse Modelling Framework* (EMF), un subproyecto de Eclipse. EMF es una herramienta que permite definir y trabajar con modelos y capaz de generar código a partir de una especificación de modelos.

La herramienta, desarrollada como un conjunto de plug-ins de *Eclipse Modelling Framework* (Eclipse 2013), da soporte a las actividades de creación y edición de multimodelos, a la configuración del producto, a la validación de la consistencia de la configuración, a la derivación del modelo CVL a partir de una configuración valida (González-Huerta 2014).

La implementación ha sido llevada a cabo utilizando el marco tecnológico Eclipse. Eclipse define Ecore, una implementación del estándar Essential MOF (EMOF) (Object Management Group 2006). Las herramientas de EMF para Eclipse permiten la definición de metamodelos que pueden instanciarse en los denominados modelos de instancia.

Esta solución emplea los niveles M1 y M2 previstos en la arquitectura MOF, es válida para la mayoría de los modelos.

2.8 QuaDAI, un Método para la Derivación, Evaluación y Mejora de Arquitecturas de Producto

Por último, el presente trabajo se engloba dentro de QuaDAI (Quality-Driven Architecture Derivation and Improvement) (González-Huerta 2014), un método de derivación, evaluación y mejora de arquitecturas de producto software en procesos de desarrollo de líneas de producto que siguen el enfoque de desarrollo dirigido por modelos.

2.8.1 Un Multimodelo para la representación de LPSs

Haciendo uso de un multimodelo, QuaDAI, es capaz de representar las vistas relevantes que permiten especificar la línea de productos y que será la columna vertebral del proceso. QuaDAI permite la derivación de una arquitectura de producto a partir de la arquitectura de la línea de producto mediante una cadena de transformación de modelos. Una vez derivada, la arquitectura será evaluada con el fin de evaluar el grado de cumplimiento de los requisitos no funcionales (que en QuaDAI se representan en la vista de calidad del multimodelo) y, en caso necesario, se aplicarán una serie de transformaciones arquitectónicas.

Dicho multimodelo está compuesto por una serie de vistas, relaciones que podemos definir entre los elementos que componen dichas vistas. El multimodelo permite representar diferentes vistas; de variabilidad, vista arquitectónica, vista de calidad y de transformaciones. Además permite definir las relaciones entre estas. El problema de resolver automáticamente la variabilidad arquitectónica a partir de una configuración requiere de, al menos, dos puntos de vista:

- **Punto de vista de la variabilidad (Variability Viewpoint)**, representa la variabilidad externa de las LPS expresando los puntos en común y variabilidad entre las líneas de producto.
- **Punto de vista arquitectónico (Architectural Viewpoint),** representa la variabilidad arquitectónica de la arquitectura de Línea de Producto que resuelve la variabilidad externa de las LPS expresadas en el variability viewpoint. Se expresa en términos de CVL y su elemento principal es la Variability Specification (VSpec) o punto de variabilidad arquitectónica.

En el multimodelo, nosotros podemos especificar como una característica dada, se realiza mediante (*is_realized_by*) una serie de VSpecs. Además, destacar que también podríamos establecer cómo dado un requisito no-funcional se realiza mediante (*is_realized_by*) una serie de VSPecs o cómo una VSpecs impacta positiva o negativamente en un atributo de calidad. Estos dos últimos casos modelarían el impacto que tienen los puntos de variabilidad arquitectónica sobre los requisitos no-funcionales y los atributos de calidad, lo que queda fuera de los objetivos a cubrir en el presente trabajo fin de grado.

Usando el multimodelo y la configuración de producto, que comprende las características del producto en desarrollo. En este trabajo, nos centramos en la definición de una serie de transformaciones ATL para dar soporte a la resolución de la variabilidad arquitectónica y a la generación automática de modelos de resolución en CVL.

2.8.2 Vista general proceso QuaDAI

Como anteriormente se ha comentado, QuaDAI, es un método integrado para la derivación, evaluación y mejora de arquitecturas software que define un artefacto (el multimodelo) y un proceso dirigido por transformación de modelos para obtener arquitecturas de producto software que cumplen, tanto los requisitos funcionales, como los no funcionales.

El mismo consta de dos fases:

a) Fase de derivación, donde se lleva a cabo **la derivación de la arquitectura de producto** a partir de la línea de productos haciendo uso de una cadena de transformaciones de modelos.
b) La siguiente fase, es la de evaluación y mejora. En primer lugar, la arquitectura será evaluada en la parte de **evaluación.** En caso de no cumplir los requisitos no funcionales es cuando se aplican las trasformaciones arquitectónicas necesarias para tratar de cumplir dichos requisitos. Esta actividad es la de **transformación.**

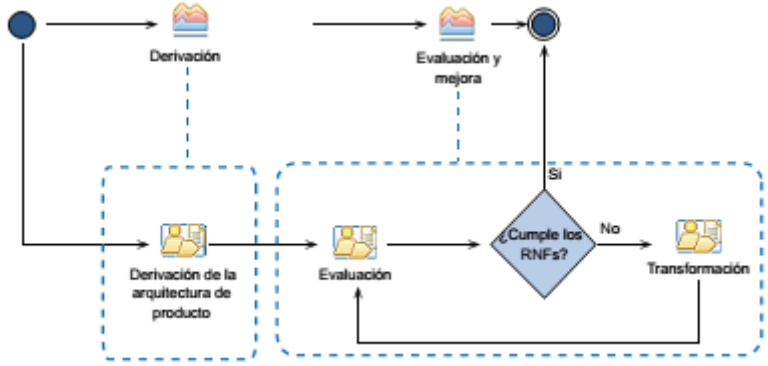


Figura 17. Proceso QuaDAI (Diagrama de actividad SPEM)(González-Huerta 2014)

La actividad de derivación, es generada a partir de la configuración (formada por características y requisitos no-funcionales), y que teniendo en cuenta las relaciones entre características, componentes de la arquitectura, requisitos no-funcionales y atributos de calidad presentes en el multimodelo, se va a obtener una *arquitectura derivada* que se trata, ni más ni menos, que de una primera versión de la arquitectura del producto en desarrollo.

La misma debe ser evaluada con el fin de determinar si se cumplen los requisitos no-funcionales.

En la fase *dos*, **evaluación y mejora,** se analiza la arquitectura y en caso de necesidad (por incumplimiento de los RNF), se llevan a cabo las transformaciones adecuadas y pertinentes para que esta situación no se produzca.

En este trabajo, únicamente nos vamos a centrar en la tarea de Derivación de la arquitectura de producto, concretamente en la obtención del modelo de variabilidad CVL.

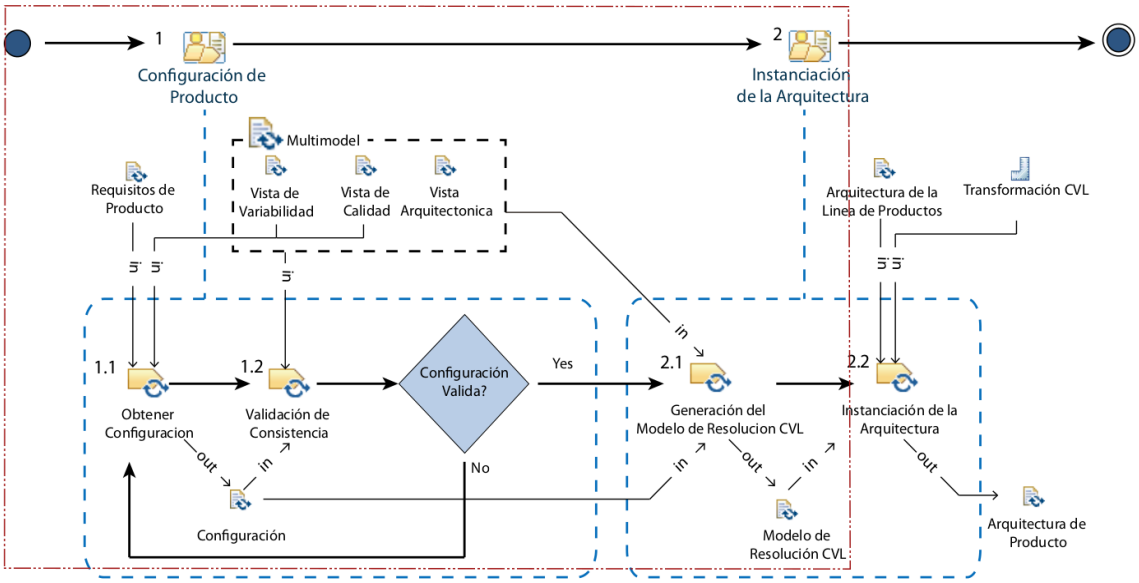


Figura 18. Amplitud en el proceso SPEM abarcado

Como se puede apreciar en la figura, el recuadro rojo muestra hasta donde vamos a tratar en este trabajo.

3 Trabajos relacionados

En un escenario de LPS es difícil manejar la derivación de la especificación de la arquitectura de un producto, especialmente cuando la LPS permite establecer un amplio rango de variabilidad. CVL ha sido usado en diversas aproximaciones para dar soporte a las resoluciones de la variabilidad arquitectónica y la derivación del producto en arquitectura en el desarrollo de LPS.

Nascimento et al. (2013), presentan una aproximación para definir arquitecturas de líneas de producto usando CVL. Aplican el método *Feature-Architecture Mapping Method* (FArM), para filtrar los modelos de características para considerar solo las características arquitectónicas relacionadas. Estas características formarán la especificación CVL que nos permitirá obtener los modelos arquitectónicos COSMOS*. Los autores no definen relaciones entre la variabilidad externa del modelo de características y la variabilidad arquitectónica expresada en CVL, obtienen los modelos arquitectónicos resolviendo manualmente los puntos de variabilidad sobre la propia especificación CVL.

Svendsen & Zhang (2010) et al. Presentan la aplicabilidad que tendría CVL para obtener los modelos de los productos para con sistema de control de trenes LPS usando un DSL. Ellos, solo consideran la definición explícita de la variabilidad interna y en consecuencia, al igual que en la aproximación anterior, la configuración debe establecerse directamente sobre la especificación CVL de la variabilidad interna.

Combemale et al. (2012), presentan una aproximación para especificar y resolver la variabilidad en *Reusable Aspect Models* (RAM), una serie de modelos de diseño interrelacionados. Ellos usan CVL para resolver la variabilidad en cada modelo y luego construir las correspondientes y reusables características usando RAM. También consideran solo la variabilidad interna y la configuración debe de hacerse sobre la especificación CVL.

Kobra (Atkinson et al. 2000) es una aproximación sistemática para el desarrollo de frameworks para el desarrollo de líneas de producto software basado en componentes que permite la derivación de la arquitectura del producto como un paso intermedio del ciclo de desarrollo. Kobra define el modelado de los componentes empleando distintos modelos, la fase de derivación está definida en la aproximación, pero no hay soporte automatizado para la misma por lo que no es posible resolver automáticamente la variabilidad.

Koalish (Asikainen et al. 2003) describe la integración del modelado de variabilidad con el lenguaje de descripción de arquitecturas software para líneas de producto Koala (Ommering et al. 2000). Además, Koalish añade nuevos mecanismos de variabilidad a Koala que solo cuenta en su definición con la parametrización y la compilación condicional. Koalish agrega la posibilidad de seleccionar el número y tipo de las partes de un componente a desplegar en la arquitectura. Los modelos arquitectónicos y de variabilidad son trasladados a un razonador que permite llevar a cabo la tarea de configuración. La salida obtenida del configurador es la descripción en Koala del producto. En este caso si se permite la obtención de modelos arquitectónicos a partir de la configuración, aunque la solución desarrollada es específica para el modelo de componentes Koala.

Cabello (2008) define un framework (BOM), con dos implementaciones distintas (*BOM-Eager* y *BOM-Lazy)* para la derivación, mediante transformaciones QVT-Relations, de arquitecturas de producto expresadas en PRISMA (Perez 2006). A partir de un modelo de variabilidad que expresa las características del dominio se obtiene un modelo conceptual expresado como diagramas de clase UML con restricciones OCL. Con este modelo conceptual, junto con el modelo de funcionalidad del sistema se obtiene el modelo de la arquitectura de la línea de productos. Al igual que en el trabajo anterior, a pesar de que se explotan las relaciones entre la variabilidad externa y la variabilidad arquitectónica, la solución aportada es específica para un determinado lenguaje de descripción arquitectónica.

Botterweck et al. (2009) proponen la automatización de la derivación de arquitecturas software mediante transformación de modelos en las que se explota la relación entre el modelo de características y componentes arquitectónicos formalizada por Janota y Botterweck (2008). La variabilidad arquitectónica es representada explícitamente sino que se establecen correspondencias entre una característica y su implementación mediante un lenguaje de modelado de la implementación (relación uno a muchos), sin permitir implementaciones alternativas. En el caso del establecimiento de relaciones entre características y variabilidad expresada mediante CVL permite mayor expresividad en la descripción de la variabilidad arquitectónica, mejorando así mismo la mantenibilidad de la arquitectónica.

Perovich et al. (2009) proponen una automatización de la derivación de la arquitectura del producto a partir de un modelo de configuración que representa las características seleccionadas para un producto. La premisa de partida es que la selección de una característica seleccionada en un modelo de configuración inspira un conjunto de decisiones arquitectónicas que guía la construcción de parte de la arquitectura del producto que incluye esa característica. Las decisiones se hacen de manera local a cada característica considerando únicamente su subárbol de características sin que haya una representación explicita de la variabilidad arquitectónica. En este caso la solución es específica para una vista, la vista componente-conector (Clements et al. 2011).

Duran-Limon et al. (2011) proponen una aproximación en la que la arquitectura del producto es derivada a partir de la arquitectura de la línea de productos mediante transformación de modelos. La variabilidad arquitectónica y el árbol de variabilidad se modelan mediante ontologías OWL (Horridge et al. 2004) y las reglas de transformación se seleccionan y componen mediante consultas a la ontología. La salida de las consultas a la ontología es la entrada a una transformación de modelos ATL que genera la arquitectura de producto. Al igual que la propuesta anterior, la solución es específica para una vista, la vista componente-conector.

En resumen, la mayoría de aproximaciones anteriormente descritas no hacen uso de relaciones entre la variabilidad externa en las LPS y la variabilidad arquitectónica (p.ej. Nascimento et al. (2013), Svendsen & Zhang (2010) o Combemale et al. (2012)) o bien son específicas para un determinado lenguaje de descripción arquitectónico (p.ej. Koalish o Cabello (2008)) o vista arquitectónica (p.ej. Perovich et al. (2009) o Duran-Limon et al. (2011)). Establecer relaciones entre el modelo de variabilidad externa y la variabilidad arquitectónica (interna) representada usando CVL nos permite resolver automáticamente la variabilidad usando transformaciones de modelos de manera

genérica, independientemente de los lenguajes de descripción arquitectónica en los que esté documentada la arquitectura o de la vista arquitectónica de interés.

4 Transformaciones para la resolución de la variabilidad arquitectónica

Una transformación consiste de una colección de reglas de transformación, que son especificaciones no ambiguas de la manera que un modelo (o parte de él), puede ser usado para crear otro modelo (o crear una parte de él). Si nos basamos en estas observaciones, definimos según (Kleppe et al. 2003):

> *Una transformación es la generación automática de un modelo destino desde un modelo fuente, de acuerdo a una definición de transformación.*
>
> *Una definición de transformación es un conjunto de reglas de transformación que juntas describen cómo un modelo en el lenguaje fuente puede ser transformado en un modelo en el lenguaje destino.*
>
> *Una regla de transformación es una descripción de cómo una o más construcciones en el lenguaje fuente pueden ser transformadas en una o más construcciones en el lenguaje destino.*

Para poder realizar las transformaciones, disponemos de herramientas de transformación, que tomando como entrada un modelo produce otro modelo como salida. Dentro de estas herramientas de transformación residen ciertos elementos que se involucran para poder realizar con éxito las transformaciones. Dentro de estas herramientas existe una definición en la que se describe como un modelo debe, (o debería), ser transformado.

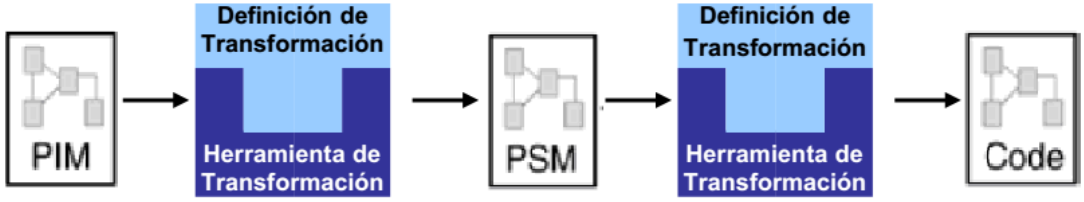


Figura 19. Definición de transformación dentro de la herramienta

Como se puede apreciar en la figura anterior, se detecta una diferencia entra la transformación en si misma: el proceso de generación de un nuevo modelo desde otro, y la definición de la transformación.

En el presente trabajo final de grado, se utiliza esta aproximación para poder desarrollar las reglas que definen las transformaciones y de esta manera obtener el modelo de resolución (modelo de salida), que es obtenido a partir del modelo de entrada en el que nosotros hemos seleccionado las características de nuestro producto software.

4.1 Estructura de un fichero .ATL

Con el objetivo de mejorar la comprensión acerca de las reglas desarrolladas, en este apartado se detalla la estructura de un fichero con transformaciones ATL. De esta manera, los conceptos clave y la forma en la que se procesan las reglas quedan explicadas dentro del ámbito del proyecto.

En la sección 2.5 se detalla la arquitectura del lenguaje utilizado para realizar las transformaciones, y a partir de la sección 2.5.7 en la que se detallan las operaciones sobre los modelos que podemos llevar a cabo en este lenguaje. Todas estas operaciones, son las que van a conformar los métodos utilizados por la herramienta, que nos va a permitir obtener el resultado deseado (siempre que las reglas estén correctamente definidas).

Haciendo uso de los Helpers o métodos auxiliares, librerías, módulos y las propias reglas, vamos a conformar un fichero, que presentará una estructura concreta, tal y como se contempla en la figura siguiente de forma genérica como sería la estructura de un fichero .atl en el que se hace uso de *lazy rules* y también se hace uso de *matched rules*.

Debemos indicar el nombre del módulo tal y como se indica en la línea 1. En la siguiente línea se debe crear el modelo de salida y se debe indicar el modelo de entrada así como el modo de ejecución que vamos a utilizar. El uso del create es la instrucción a usar para poder obtener lo anteriormente expuesto.

La diferencia entre el modo normal y el modo *refining* o refinamiento es que en el modo normal se debe especificar la forma en que cada uno de los elementos del modelo destino, van a ser generados.

No obstante, si usamos el modo refinamiento, solo se emplean cuando el modelo origen y destino son conformes al metamodelo. De esta forma solo hay que indicar las modificaciones a llevar a cabo entre el modelo origen y destino, ya que el resto de los elementos son copiados automáticamente.

```
1  module module_name;
2  create outuput_models [from|refining] input_models;
3  uses extension_library_file_name;
4
5  rule rule_name {
6          from
7                  in_var : in_type [in model_name]? [(
8                          condition
9                                  )]?
10                 [using {
11                         var1 : var_type1 = init_exp1;
12                         ...
13                         varn : var typen = init expn;
14                 }]?
15         to
16                 out_var1 : out_type1 [in model_name]? (   bindings1        ),
17                 out_var2 : distinct out_type2 foreach(e in collection)(bindings2),
18                 ...
19                 out_varn : out_typen [in model_name]? (   bindingsn        )
20         [do {
21                 statements
22         }]?
23 }
24 lazy rule rule_name {
25         from
26                 in_var : in_type [in model_name]? [(
27                         condition
28                 )]?
29                 [using {
30                         var1 : var_type1 = init_exp1;
31                         ...
32                         varn : var typen = init expn;
33                 }]?
34         to
35                 out_var1 : out_type1 [in model_name]? (   bindings1        ),
36                 out_var2 : distinct out_type2 foreach(e in collection)(bindings2),
37                 ...
38                 out_varn : out_typen [in model_name]? (   bindingsn        )
39         [do {
40                 statements
41         }]?}
```

Listado 1. Ejemplo de la estructura de un fichero ATL

Estas instrucciones se engloban dentro de lo que sería la cabecera del fichero (o header), tal y como se explica en la sección 2.5.102.5. También se puede indicar el uso de librerías externas ATL (como podemos ver en la línea 3).

A continuación, ya podemos añadir las reglas que conforman el fichero y que permite realizar las transformaciones.

4.2 Reglas en ATL para la transformación de modelos

En este apartado se presentan las reglas ATL desarrolladas en este proyecto, gracias a las cuales se obtiene la pertinente transformación de modelos.

Se adjunta al final del documento un anexo (Listado 10), donde se muestra un extracto del fichero utilizado para llevar a cabo las transformaciones. El fichero, se compone de dos *Helpers*, tres *matched rules* y tres *lazy rules*. Los *helpers* son los que podemos identificar con el nombre *featureSelected* e *iteratorSons*. Las reglas de tipo matched, son: *root, EVSpec e Iterators*. Por último, las reglas que llevan la etiqueta *lazy* son obviamente las que se corresponden con las reglas de tipo lazy, cuyos nombres son *VSpec, FragmentSubstitution y VarIterator*.

Dentro de la cabecera del fichero se observa en primer lugar, el nombre del módulo, que en este caso es **mymatchedrules**, y como hemos señalado en el apartado anterior, el siguiente paso es indicar en primer lugar el nombre del modelo que se va a generar y a continuación el modelo de entrada. Se puede apreciar como en las líneas 1 y 2 aparecen lo siguiente:

```
-- @nsURI MMin=http://es.upv.dsic.issi/multiple/automotiveMultimodelCVL
-- @nsURI MMout=http://org.variabilitymodeling.cvl
```

Se trata de un enlace a los modelos, es decir, gracias a esto, cuando en el create, indicamos que debemos tomar '*MMin*' como modelo de entrada, '*MMin*' tiene sentido, gracias a este enlace. El '*MMout*' es creado a partir del modelo de entrada, pero sus bases se asientan gracias al metamodelo general proporcionado por la (Object Management Group 2012).

Destacar que el modo de ejecución, va a ser en modo normal y no en refinamiento. Para ello deberíamos haber utilizado la etiqueta *refining* en la línea 4, pero en este trabajo no se va a hacer uso de tal modo de ejecución. Esto se debe a que el resultado obtenido al realizar las transformaciones no era el esperado, porqué se hacía uso de varios modelos de entrada y era en ese punto donde aparecían los problemas.

4.2.1 Helpers

Una vez definidos los atributos necesarios en la cabecera, es cuando podemos incorporar código; en este caso encontramos en la línea 15 el primer helper (Listado 10). Como ya se ha comentado los helpers son, por compararlo con otro lenguaje, como una función o método en Java.

Por este motivo, para ser utilizados, deben ser invocados, y esta llamada puede (o no) contener parámetros, la presencia de parámetros, va a depender de la definición del helper, y el tipo de la expresión de retorno también hay que indicarlo.

4.2.1.1 Helper featuredSelected

```
1  helper def: featureSelected(EVSpec: MMin!EVSpec): Boolean =
2  EVSpec.multimodel.relationships -> select(e | e.oclIsTypeOf(MMin!FeatureToVSpec)) ->
3  select(e | e.to = EVSpec and e.from.Selected = true).isEmpty();
```

Listado 2. Helper featureSelected

La secuencia de sucesos que se dan en el *helper* **featureSelected**, es la siguiente: en la llamada al mismo, este debe recibir un parámetro. Éste tiene como nombre '*EVSpec'*, y tipo del mismo debe ser *EVSpec*. El tipo del valor de retorno es un booleano, que va a ser true o false dependiendo de la evaluación final de la expresión:

- Dentro de las relaciones presentes en el multimodelo sean de tipo FeatureToVSpec. Para ello, se hace una selección de estos elementos mediante el uso de la instrucción que encontramos en la línea 16:
  - `select(e | e.oclIsTypeOf(MMin!FeatureToVSpec))`

- Esta instrucción, lo que hace es devolver una lista de elementos que sean de tipo FeatureToVSpec. Posteriormente con el uso de la instrucción de la línea 3, que no es ni más ni menos que otro select, que podemos entender como un filtrado, en el cual nos quedamos con aquellos elementos que el atributo '*to'*, es igual al parámetro de entrada (EVSpec).
  - `select(e | e.to = EVSpec and e.from.Selected = true)`

- No obstante, hay que cerciorarse que en efecto la característica para la cual la relación existe, debe estar seleccionada. Para ello, está la parte de:
  - `and e.from.Selected = true`

- Cuya misión es asegurar que la EFEature de la que proviene tiene el atributo *selected* marcado como true. En este punto, vamos a tener una lista que cumpla los requisitos anteriores. Pero, nosotros vamos a devolver un dato de tipo booleano, no una lista con elementos. Por ello la presencia del *.isEmpty(),* presente al final.
  - `select(e | e.to = EVSpec and e.from.Selected = true).isEmpty();`

- Por este motivo, si la lista que devolvemos contiene elementos, *isEmpty(),* se evalúa a false y por tanto sabemos que sí que existen características que están seleccionadas y este helper devolvería true.

4.2.1.2 Helper iteratorSons

```
1  helper def: iteratorSons(Iterator: MMin!Iterator): Boolean =
2  (Iterator.variabilitySpecification -> select(e | e.oclIsTypeOf(MMin!EVSpec) and not
3  (thisModule.featureSelected(e)))) -> size() > 0;
```

Listado 3. Helper iteratorSons

Una vez analizado el comportamiento que presenta el helper anterior, vamos a analizar el que presenta el siguiente: **iteratorsSons.**

- Como se puede observar el parámetro de entrada es un elemento de tipo Iterator y el tipo del valor de retorno es un booleano:
  - `helper def: iteratorSons(Iterator: MMin!Iterator): Boolean`

- El cómputo de la instrucción, o del helper, será true o false en caso que dentro del parámetro de entrada, exista un elemento o más que tengan como tipo EVSPec.
    - `Iterator.variabilitySpecification -> select(e | e.oclIsTypeOf(MMin!EVSpec)`
- Otra aspecto que debemos tener en cuenta, es que dentro de este helper se hace o realiza una llamada al helper descrito anteriormente (featureSelected), tal y como se puede apreciar:
    - `thisModule.featureSelected(e)`

- Esta operación lo que pretende es garantizar que dicho elemento no esté seleccionado, ya que como podemos ver la condición previa a la llamada es:
    - `and not(thisModule.featureSelected(e)`

- De este modo se obtiene una lista de elementos que por una parte cumplen que son de tipo EVSpec y que presentan el atributo *selected* como false. Si esta lista, tiene un tamaño mayor a cero, el valor de retorno de este helper es true.
    - `-> size() > 0;`

- El objeto final de este helper es determinar si existen elementos que 'cuelgan' del parámetro que se le pasa como entrada y de esta forma, por decirlo de alguna manera, vamos a poder incluir en el modelo de resolución los elementos que se corresponderían con los descendientes del iterador que se pasa como parámetro de entrada.

Como ya hemos comentado, estos son los dos helpers presentes en las transformaciones utilizadas. Nos ayudan a obtener cierta información que es necesaria para poder recorrer el modelo origen y posteriormente plasmarlo en la salida correspondiente.

Los helpers son invocados desde diferentes reglas, tal y como se puede apreciar en el Listado 5, la regla EVSpec (línea 7), que realiza una llamada al helper *featuredSelected,* y también en el Listado 6, la regla Iterators en la parte del to realiza una llamada al helper mencionado (línea 10).

Por otra parte, el segundo helper, iteratorSons, únicamente es invocado desde la regla Iterators en la parte del from.

4.2.2 Matched rules

Siguiendo con el análisis de las reglas presentes en el fichero, pasamos ahora a analizar las reglas de tipo matched, donde encontramos la regla *root*, *EVSpec* e *Iterators*.

4.2.2.1 Regla root

```
1  rule root {
2  from
3       VarViewpoint: MMin!VariabilityViewpointModel,
4       EVSpec: MMin!EVSpec,
5       Feature: MMin!EFeature,
6       FeatToVSpec: MMin!FeatureToVSpec
7       (
8         FeatToVSpec.from = Feature and FeatToVSpec.to = EVSpec and
9         Feature = VarViewpoint.rootFeature and
10        not EVSpec.oclIsTypeOf(MMin!FragmentSubstitution)
11      )
12 to
```

```
13      CVLModel: MMout!CVLModel (
14       name <- EVSpec.name,
15       resolutionSpecification <- Sequence{EVSpec},
16       variabilitySpecification <- EVSpec.refImmediateComposite().
17         variabilitySpecification -> select(e | not e.
18          oclIsTypeOf(MMin!FragmentSubstitution)) -> collect(e | thisModule.
19           VSpec(e))
20     )
21  }
```

Listado 4. Regla root

Se trata de una regla de tipo matched que en la parte del from recoge, o toma varios elementos. En primer lugar, indicamos por *donde* debemos navegar. Es decir, tal y como se aprecia en la línea 3, *VarViewPoint* hace referencia a la vista del modelo origen en la que vamos a ver o buscar los elementos, o características que deseamos. A continuación, especificamos o creamos los elementos que vamos a utilizar en la transformación y que deben estar presentes en el modelo origen, por lo que los elementos que encontramos en las líneas 4 y 5, se refiere a las características o *features* que deben estar contenidas dentro del Variability Viewpoint Model: EFeature y EVSpec, que utilizaremos posteriormente también en el modelo de salida, ya que 'vamos a tomar' información de estas variables para poder plasmarlas correctamente en la salida.

En la línea 6, encontramos *FeatToVSpec,* se trata de un elemento de tipo FeatureToVSpec, y dentro de él se define una restricción que va a hacer la función de filtro. En las líneas 8, 9 y 10 es donde se realiza dicha operación:

- En la línea 8, lo que se indica es que por una parte, el from del elemento actual (FeatToVSpec), coincida con la EFeature que debe estar dentro de este punto de vista y del mismo modo, establece que el *.to,* es igual al elemento EVSpec.
  - `FeatToVSpec.from = Feature and FeatToVSpec.to = EVSpec`

- En la línea 9, lo que especificamos o restringimos es que el primer elemento que cuelga del punto de vista de la variabilidad (Variability ViewPoint Model) **debe** ser igual a la Feature.
  - `Feature = VarViewpoint.rootFeature and`

- Por último, lo que debemos asegurar para el correcto funcionamiento de la regla es que EVSpec **no** sea de tipo *FragmentSubstitution*, tal y como vemos en la línea 10:
  - `not EVSpec.oclIsTypeOf(MMin!FragmentSubstitution)`

Una vez analizada esta primera parte de la regla, pasamos a la parte segunda, marcada por la palabra reservada **to**. En esta parte lo que vamos a hacer es en primer lugar crear un elemento llamado CVLModel.

- Podemos apreciar que el tipo es un CVLModel, y que vamos a *llenar* los atributos presentes en este elemento de manera que el producto final de la transformación resulte el adecuado:
  - `CVLModel: MMout!CVLModel (`

- Para ello, el nombre de este elemento que vamos a crear debe ser el que *hemos recogido* en la parte del from y hemos *guardado* en EVSpec (línea 14),
  - `name <- EVSpec.name,`

- El siguiente atributo a cumplimentar es *resolutionSpecification*, que se trata de una secuencia o lista con las EVSPec *recogidas* en la parte del *from*:
  - `resolutionSpecification <- Sequence{EVSpec},`

- Finalmente en las líneas 16 a 19, lo que hacemos es otra operación de filtrado en la que *obligamos* a que por una parte, el primer elemento que va a colgar del elemento raíz, es su hijo:
  - `variabilitySpecification <- EVSpec.refImmediateComposite().`

- Pero, debemos tener en cuenta otras restricciones, como por ejemplo, que este elemento no se trate de un objeto de tipo *FragmentSubstitution:*
  - `variabilitySpecification -> select (e|not e.oclIsTypeOf( MMin!FragmentSubstitution))`

- Y además se realiza una llamada a una *lazy rule*, con el fin de obtener la 'lista' de hijos que van a colgar del elemento raíz:
  - `—> collect(e | thisModule.VSpec(e))`

- Por ello se hace uso de la recursión en las reglas de tipo *lazy* y que funcionan las tres como un conjunto, y que gracias a su actuación se podrá establecer de forma correcta el modelo de resolución en el que se plasma la solución al problema de la variabilidad.

4.2.2.2 Regla EVSpec

```
1  rule EVSpecs {
2  from
3       EVSpec: MMin!EVSpec (
4       not(EVSpec.refImmediateComposite()
5        .oclIsTypeOf(MMin!ArquitecturalViewpointModel))and
6        if (not(EVSpec.multimodel.oclIsUndefined())) then
7        not thisModule.featureSelected(EVSpec)
8
9        else
10        false
11       endif
12      )
13
14 to
15     ResolutionElement: MMout!ResolutionElement (
16     name <- EVSpec.name,
17      resolution <- Sequence {} -> union(EVSpec.variabilitySpecification ->
18        select(i | i.oclIsTypeOf(MMin!Iterator)))
19    )
20 }
```

Listado 5. Regla EVSpec

La regla EVSpec, es la encargada de gestionar o tratar los elementos de tipo EVSpec, para ello es necesario realizar tal y como ya hemos aplicado anteriormente una serie de restricciones.

- Primero debemos asegurar que el 'espacio' en el que estamos *tratando* **no** es el punto de vista arquitectónico, por ello usamos la siguiente restricción en la que estamos *mirando* cada elemento de tipo EVSpec, y posteriormente confirmamos que el elemento contenedor de dicho elemento no está en *ArquitecturalViewPointModel:*
  - `rule EVSpecs {`
    `from`

```
EVSpec:
 MMin!EVSpec (not(EVSpec.refImmediateComposite()
  .oclIsTypeOf(MMin!ArquitecturalViewpointModel)
```

- Una vez sabemos el resultado de esta evaluación, y esta se evalúa a *true,* entonces es cuando debemos asegurar que el multimodelo esté definido, es decir, que exista y se debe hacer uso de la expresión oclIsUndefined() para tal fin:
  - `if (not(EVSpec.multimodel.oclIsUndefined())) then`

- Si el resultado de esta expresión booleana es true, entonces, lo que hacemos es uso del primer helper, del cual queremos obtener o revisar los elementos que **no** tienen el atributo *selected* marcado como *true*:
  - `not thisModule.featureSelected(EVSpec)`

- En caso que la primera condición del if no se cumpla, *cortamos* la ejecución de la siguiente manera, de forma que dejamos de explorar *el espacio* en el que nos encontramos:
  - ```
    Else
     False
    endif
          )
    ```

- Una vez cumplimentada la primera parte de la regla, es hora de pasar a la segunda, en la que vamos a crear un elemento, que presenta el tipo ResolutionElement, en el que le asignamos como nombre el que hemos recogido del elemento EVSpec explorado:
  - ```
    to
    ResolutionElement: MMout!ResolutionElement (
                        name <- EVSpec.name,
    ```

- El elemento ResolutionElement que se ha creado, tiene una serie de atributos, uno de ellos, aparte del nombre, es 'su resolución', es decir, los elementos o el elemento en sí, y que en este caso van a ser los Iteradores, ya que en el fondo lo que en este apartado hacemos es tratar con elementos tipo Iterator (líneas 17 y 18):
  - ```
    resolution <- Sequence {} ->
    union(EVSpec.variabilitySpecification -> select(i |
    i.oclIsTypeOf(MMin!Iterator)))
                )
    ```

4.2.2.3 Regla Iterators

```
1  rule Iterators {
2   from
3   I: MMin!Iterator (
4   thisModule.iteratorSons(I)
5  )
6  to
7   IR: MMout!IteratorResolution (
8   name <- I.name,
9    choice <- Sequence{} -> union(I.variabilitySpecification ->
10   select(e | e.oclIsTypeOf(MMin!EVSpec) and not (thisModule.featureSelected(e))))
11  )
12 }
```

Listado 6. Regla Iteratos

La siguiente regla que encontramos, es la regla Iterators y que va a *investigar* los elementos de tipo Iterator.
```

- Para ello, por cada elemento iterador que encontremos en modelo de entrada, vamos a realizar una serie de acciones, la primera de ellas, es hacer uso del helper *iteratorSons,* al que le vamos a pasar como parámetro el elemento actual:
  - ```
    rule Iterators {
    from
    I: MMin!Iterator (
     -- LLamada al helper
     thisModule.iteratorSons(I)
    )
    ```

- Una vez obtenida la respuesta del helper, pasamos a la segunda parte de la regla, en la que creamos un elemento de tipo IteratorResolution, y le asignamos el nombre 'del iterador padre':
  - ```
    To
    IR: MMout!IteratorResolution (
          name <- I.name,
    ```

- Los elementos de tipo Iterator, tienen un atributo que es *choice*, en el que hay una lista de elementos. Estos elementos deben de ser de tipo EVSpec y además deben cumplir la condición de **no** estar seleccionados en la configuración inicial:
  - ```
    choice <- Sequence{} ->
    union(I.variabilitySpecification -> select(e | e.
                        oclIsTypeOf(MMin!EVSpec)      and      not
    (thisModule.featureSelected(e))))
    ```

4.2.3 Lazy rules

Por último, las reglas que quedan por describir, son la de tipo *lazy*, que trabajan *en conjunto,* haciendo uso de la recursión entre ellas, de forma que nos permiten obtener el resultado esperado para poder obtener el modelo de resolución de forma correcta.

4.2.3.1 Lazy rule VSpec

```
1  lazy rule VSpec {
2   from
3    EVSpec: MMin!EVSpec -- Creamos una 'CompositeVariability'
4   to
5     VSpec: MMout!CompositeVariability (
6     name <- EVSpec.name,
7      variabilitySpecification <- EVSpec.variabilitySpecification ->
8       select(v | v.oclIsTypeOf(MMin!Iterator)) ->
9        collect(e |thisModule.VarIterator(e))
10   )
11 }
```

Listado 7. Lazy Rule VSpec

La primera *lazy rule* que encontramos es la que se muestra en Listado 7, nombrada VSpec. Esta es llamada por la regla *root* (Listado 4), en la parte del *to* de dicha regla, donde se pasaba como parámetro un elemento de tipo EVSpec.

- En esta regla, por cada EVSpec que le pasemos en las correspondientes llamadas, va a crear un elemento de tipo *CompositeVariability:*
  - ```
    lazy rule VSpec {
          from
           EVSpec: MMin!EVSpec
          to
    ```

```
VSpec: MMout!CompositeVariability (
```

- Este elemento *CompositeVariabilty,* nombrado VSpec, recoge ciertos atributos por parte del parámetro de entrada, como el nombre, pero lo realmente importante en esta regla, es que realizamos una operación de filtrado.
  En dicha operación, obligamos a que la especificación de la variabilidad del elemento VSPec, sea de tipo iterador (línea 8) y además se realiza una llamada a la *lazy rule VarIterator* (línea 9):
  - ```
    to
    VSpec: MMout!CompositeVariability (
          name <- EVSpec.name,
           variabilitySpecification <- EVSpec.
            variabilitySpecification ->
             select(v | v.oclIsTypeOf(MMin!Iterator)) ->
              collect(e | thisModule.VarIterator(e))
    ```

- La llamada a la regla *VarIterator,* tiene como misión obtener aquellos elementos de tipo EVSpec que contengan elementos de tipo *FragmentSubstitution*. Entre estas dos reglas se puede apreciar el uso de la recursión anteriormente mencionada, ya que tal y como se puede ver en el Listado 9 (línea 10), la regla *VarIterator,* realiza una llamada a la regla VSpec.

4.2.3.2 Lazy rule FragmentSubtitution

```
1 lazy rule FragmentSubstitution {
2  from
3   EVSpec: MMin!FragmentSubstitution
4  To
5   VSpec: MMout!FragmentSubstitution (
6    name <- EVSpec.name
7   )
8 }
```

Listado 8. Lazy rule FragmentSubstitution

La *lazy rule* FragmentSubtituion, es la encargada de generar los elementos de tipo FragmentSubtitution que van a estar presentes en el modelo de resolución.

- Como parámetro de entrada recibe un elemento de tipo Iterador, pero si nos fijamos en la regla desde la que la llamamos, lo que en realidad vamos a utilizar de este Iterator que recibe como parámetro, es aquellos elementos que están presentes, o contenidos dentro del iterador:
  - ```
    lazy rule FragmentSubstitution {
     from
      EVSpec: MMin!FragmentSubstitution
     to
      VSpec: MMout!FragmentSubstitution (
      name <- EVSpec.name
     )
    }
    ```

4.2.3.3 Lazy rule VarIterator

```
1 lazy rule VarIterator {
2  from
3   Iterator: MMin!Iterator
4  to
5   VSpec: MMout!Iterator (
6   name <- Iterator.name,
```

```
7     lowerLimit <- Iterator.refGetValue('lowerLimit'),
8     upperLimit <- Iterator.refGetValue('upperLimit'),
9      variabilitySpecification <- Iterator.variabilitySpecification ->
10      select(v | v.oclIsTypeOf(MMin!EVSpec)) -> collect(e | thisModule.VSpec(e)) ->

11       union(Iterator.variabilitySpecification ->
12        select(v | v.oclIsTypeOf(MMin!FragmentSubstitution)) ->
13         collect(f | thisModule.FragmentSubstitution(f)))
14    )
15  }
```

Listado 9. Lazy rule VarIterator

Por último dentro del apartado de análisis de las transformaciones, encontramos la regla *VarIterator*.

- Como parámetro de entrada recibe un elemento de tipo Iterador, y crea un elemento nombrado VSpec, que es de tipo Iterador, pero en su resolución, o especificación de la variabilidad nos quedamos con aquellos elementos que presentan un tipo concreto; EVSpec y FragmentSubstitution. Además esta regla realiza una llamada a la regla que hemos descrito anteriormente:
  - ```
    lazy rule VarIterator {
     from
      Iterator: MMin!Iterator
     To
      VSpec: MMout!Iterator (
          name <- Iterator.name,
          lowerLimit <- Iterator.refGetValue('lowerLimit'),
          upperLimit <- Iterator.refGetValue('upperLimit'),
    variabilitySpecification <- Iterator.
     variabilitySpecification ->
       select(v | v.oclIsTypeOf(MMin!EVSpec)) ->
        collect(e | thisModule.VSpec(e)) ->
         union(Iterator.variabilitySpecification ->
          select(v | v.oclIsTypeOf(
           MMin!FragmentSubstitution)->
            collect(f | thisModule.FragmentSubstitution(f)))
     )
    }
    ```

El uso de estas reglas para llevar a cabo las transformaciones, es el que permite obtener el modelo de salida con la resolución de la variabilidad. Su utilización permite recorrer el modelo de entrada, indagando en cada elemento presente, comprobando si se cumplen o no las condiciones establecidas en las reglas, de forma que por cada elemento encontrado, se crea otro en el modelo de resolución que, atendiendo a la configuración de entrada, va a presentar unas u otras características.

Es gracias al uso de concatenaciones, recursión y uniones lo que permite obtener y plasmar correctamente cada elemento en su posición o nivel.

Un uso inadecuado de estas operaciones, resuelve incorrectamente la variabilidad, ya que, podría por ejemplo, *colocar* elementos en un nivel no correspondido; por ejemplo, colocar al mismo nivel un iterador y sus hijos.
    ```

5 Ejemplo de aplicación

En esta sección se pretende mostrar el ejemplo de aplicación sobre el que se han llevado a cabo las transformaciones explicadas en el apartado anterior, con el fin de proporcionar un enfoque más visual a las reglas anteriormente descritas, a la par que mostrar una parte de lo que somos capaces de generar con lenguajes de este tipo.

También, pretende ser una especie de pequeño tutorial con el que poder recrear una configuración determinada para el modelo de entrada, y poder conocer un poco más a fondo la herramienta descrita en las secciones 2.7 y 2.8.

5.1 Modelo y descripción de características

Ya que partimos de un caso de estudio previo, CarCarSPL (González-Huerta 2014), es necesaria una pequeña explicación con el fin de poder comprender al máximo de qué se trata. CarCarSPL, es una compañía de software que desarrolla sistemas de control para automóviles, en el que se integran sensores y nodos de computación. En esta sección se presentan los diferentes componentes por los que está compuesto el sistema de control (*VehicleControlSystem*).

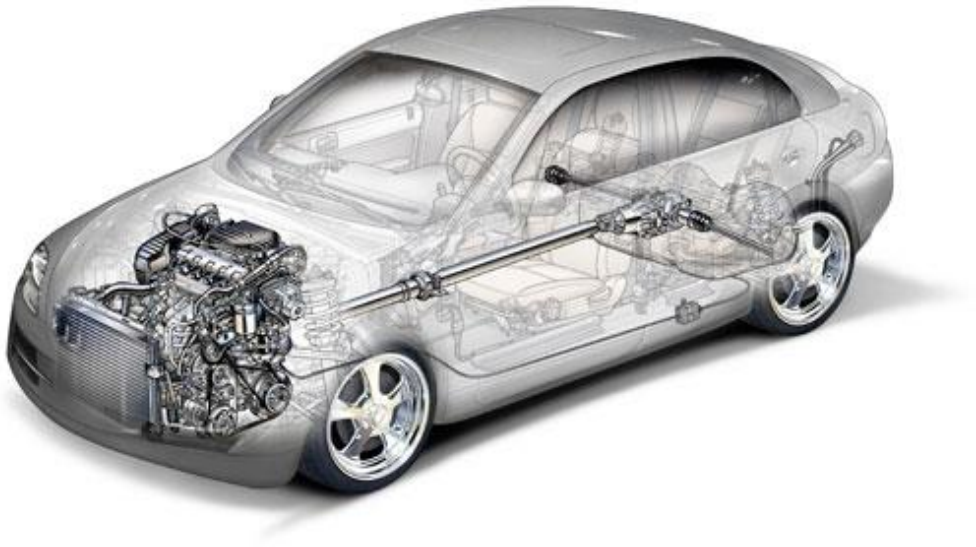
Dentro del sistema de control, encontramos diferentes elementos. Dichos elementos son características que presenta el vehículo, como el ABS, Control de Tracción, etc.

En esta sección se describen las diferentes características de los productos que integran el *VehicleControlSystem* SPL:

- **Sistema Antibloqueo de frenos (ABS):** El objetivo del ABS, es asegurarse que la máxima fuerza de frenado, es transmitida a las cuatro ruedas del vehículo, incluso en condiciones adversas como lluvia, nieve o hielo.
  El sistema de antibloqueo de frenos, funciona a través de sensores que detectan si las ruedas patinan o no durante la frenada, a través del sensor de giro, y ajustando la presión del freno para asegurar el máximo contacto entre las ruedas y la carretera, por medio de los actuadores electrónicos. En la mayoría de las versiones básicas, los sensores de rotación de las ruedas, se usan como entradas y la salida es la válvula de freno presente en cada línea de freno.

- **Control Sistema de Tracción (TCS):** La misión del TCS es evitar que las ruedas pierdan adherencia mientras aceleramos. TCS ha de controlar por una parte el eje frontal como el eje trasero. Utiliza datos del sensor de rotación de cada una de las ruedas del vehículo, compara dichos datos con la velocidad para detectar las ruedas que patinan y compensa esta situación reduciendo la velocidad. Esto se puede conseguir de dos formas, frenando la rueda que patina, o reduciendo la inyección de combustible con la finalidad de asegurar máximo contacto entre la carretera y las ruedas incluso en condiciones adversas, como hielo o nieve.

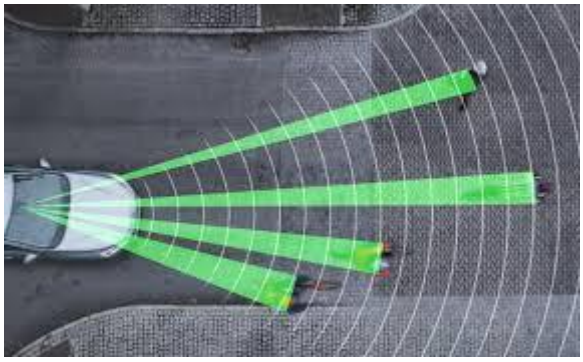
- **Sistema de control de estabilidad (SCS):** La función del SCS es mantener el vehículo en la dirección en la que el conductor está orientando el mismo. Para conseguir esto, el SCS, frena una rueda para ayudar a dirigir al vehículo en la dirección correcta. La diferencia entre SCS y TCS es en lo que pretenden evitar. El TCS actúa sobre el control de tracción de las ruedas con el fin de evitar que patinen al acelerar. El SCS, va un paso más allá, ya que detecta cuando el conductor ha perdido el control del vehículo y de forma automática intenta estabilizar el vehículo para ayudar al piloto a tener el control de nuevo.



- **Sistema de control de Crucero (CC):** El fin de este dispositivo es mantener la velocidad determinada por el conductor de forma constante. El CC mantiene la velocidad en un valor predeterminado guardando la velocidad de rotación de la rueda cuando éste se ha determinado. Si la inclinación de la carretera varía, la velocidad del vehículo cambia, por lo que la posición del acelerador debería cambiar con el fin de mantener la velocidad. El CC, detecta la diferencia de velocidad entre la actual y la deseada y a partir de este resultado determina si acelerar o desacelerar. El algoritmo encargado de realizar dicho cálculo se denomina *control law*. Dependiendo del sensor que esté disponible la funcionalidad que proporcione (el Control de crucero) puede diferir. Existen tres posibles configuraciones:

- **Control de Crucero básico:** Mantener la velocidad establecida de forma constante.
- **Control de Crucero autónomo:** Se trata de una extensión del control de crucero básico, ya que proporciona una nueva función: control de la distancia con el vehículo de delante. Para asegurar que la distancia se mantiene constante, se incluyen un sensor, *obstacle avoidance,* que ofrece la siguiente información: la distancia de nuestro vehículo respecto al vehículo *objetivo* y la velocidad relativa entre ambos.

- **Control de Crucero totalmente autónomo:** es una extensión del anterior, añadiendo una cámara que captura el comportamiento de los vehículos y objetos que se encuentran alrededor del sistema, esta información se procesa, y es capaz de detener el vehículo en caso que se detecte colisión inminente.



- **Asistente de parking:** Su función es informar de la presencia de un obstáculo durante las maniobras realizadas en el proceso de aparcar. Los sensores escanean la situación y cuando los sensores de detección de obstáculos (tanto en la parte trasera como delantera del vehículo), se percatan de la existencia de un obstáculo dentro del perímetro de seguridad, empiezan a emitir tanto señal acústica como luminosa. En caso de detectar que nos acercamos cada vez más a los obstáculos estas señales incrementan su intensidad.
- **Parking automático:** La función de este sistema es ejecutar las maniobras de aparcamiento de forma automática, de manera que el conductor no tenga que llevarlas a cabo. El sistema usa una serie de sensores de proximidad y sensores de detección de obstáculos con el fin de analizar el entorno. El sistema toma el control de la dirección del vehículo, los frenos y el acelerador para llevar a cabo la maniobra.

- **Sistema multimedia**: Este sistema encapsula una serie de características:
  - ***FMC_CD:*** Este sistema pretende controlar el sistema *stereo* básico, cuya composición se basa en una FM-Radio y una unidad de CD (single-CD). El sistema permite almacenar y sintonizar emisoras de radio, seleccionar la canción del CD a reproducir, pausar y para el sistema.
  - ***FM_CD_Charger*:** Se encarga de controlar el sistema de *stereo* de alta calidad, el cual está compuesto por la radio FM (FM-Radio) y seis unidades de carga para CDs. El sistema permite sintonizar, almacenar y obtener estaciones de radio, permite usar ecualizador, seleccionar la canción del CD a reproducir, pausarla y reanudarla.
  - ***B_W_OnboardComputer*:** Este sistema permite configurar otros sistemas presentes en el vehículo. Además, muestra estadísticas sobre el rendimiento del vehículo y avisa en caso de necesidad de tareas de mantenimiento. Tiene una pantalla en blanco y negro con una serie de botones que permiten al usuario interactuar con el sistema.
  - ***ColorOnboardComputer:*** Este sistema permite configurar otros sistemas presentes en el vehículo. Además, muestra estadísticas sobre el rendimiento del vehículo y avisa en caso de necesidad de tareas de mantenimiento. Dispone de una pantalla en color a alta resolución y una serie de botones que permiten al usuario interactuar con el sistema. Si el *VehicleControlSystem* se encuentra seleccionado, el ordenador de a bordo, incluirá al GPS.

- **GPS**: El GPS permite determinar la posición, trayectoria y velocidad a partir de la información proporcionada por el GPS. El sistema integra un módulo GPS que es capaz de calcular la posición del coche. ColorOnboardComputer, será el encargado de mostrar la información de navegación en caso que esta característica esté seleccionada.
- **Bluetooth**: El módulo Blluetooth permite la interconexión con un dispositivo móvil y realizar/recibir llamadas. El sistema integra un módulo GPS, y es capaz de interactuar con el conductor a través de una serie de botones, un micrófono y el sistema de sonido (FM_CD o FM_CD_Charger).
- **iSafe**: Este sistema de seguridad, en caso de accidente se encarga de enviar la posición GPS y de llamar a los sistemas de emergencia. El sistema es capaz de leer la información proveniente del GPS y usa el módulo Bluetooth para comunicarse con los sistemas de emergencia.

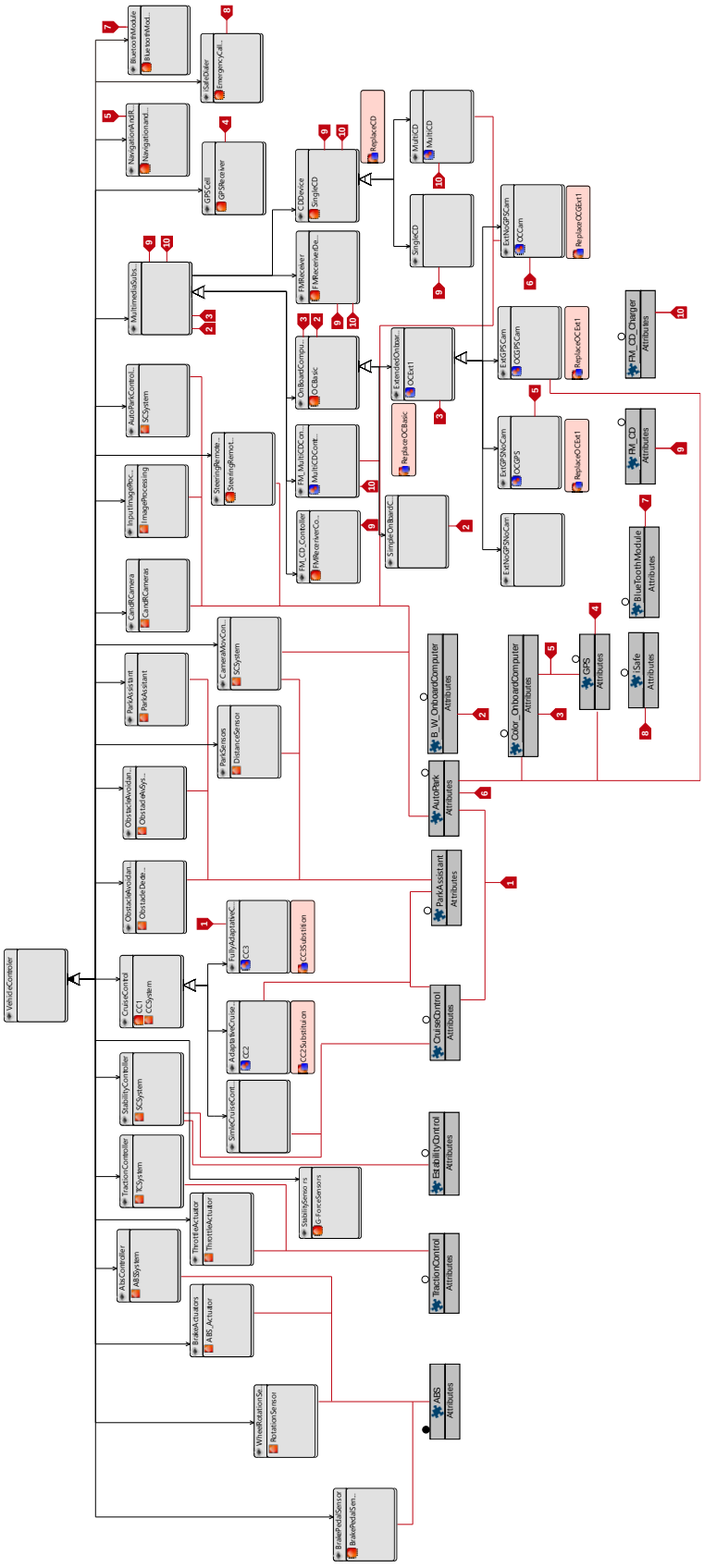


Figura 20. Realization Relationships entre Architectural Variability Points y Features

En el caso de estudio, encontramos un modelo de características con diferentes elementos: el control de antibloqueo de frenos ABS, el sistema de control de tracción o el control de crucero, etc. Dichas características comprenden una serie de sistemas software para la captura de las señales de entrada de los sensores, su procesamiento y transformación y el envío de la salida procesada a un actuador que potencialmente puede afectar al estado de otros sistemas o partes mecánicas del vehículo (como la posición del acelerador, el motor, el airbag o los frenos).

Por ejemplo, el control de crucero, incorpora su propia variabilidad. Esta variabilidad se resuelve en base a la selección hecha sobre el modelo de características (la selección del control de crucero junto con el asistente para aparcar, que implica la resolución positiva de una versión extendida del control de tracción).

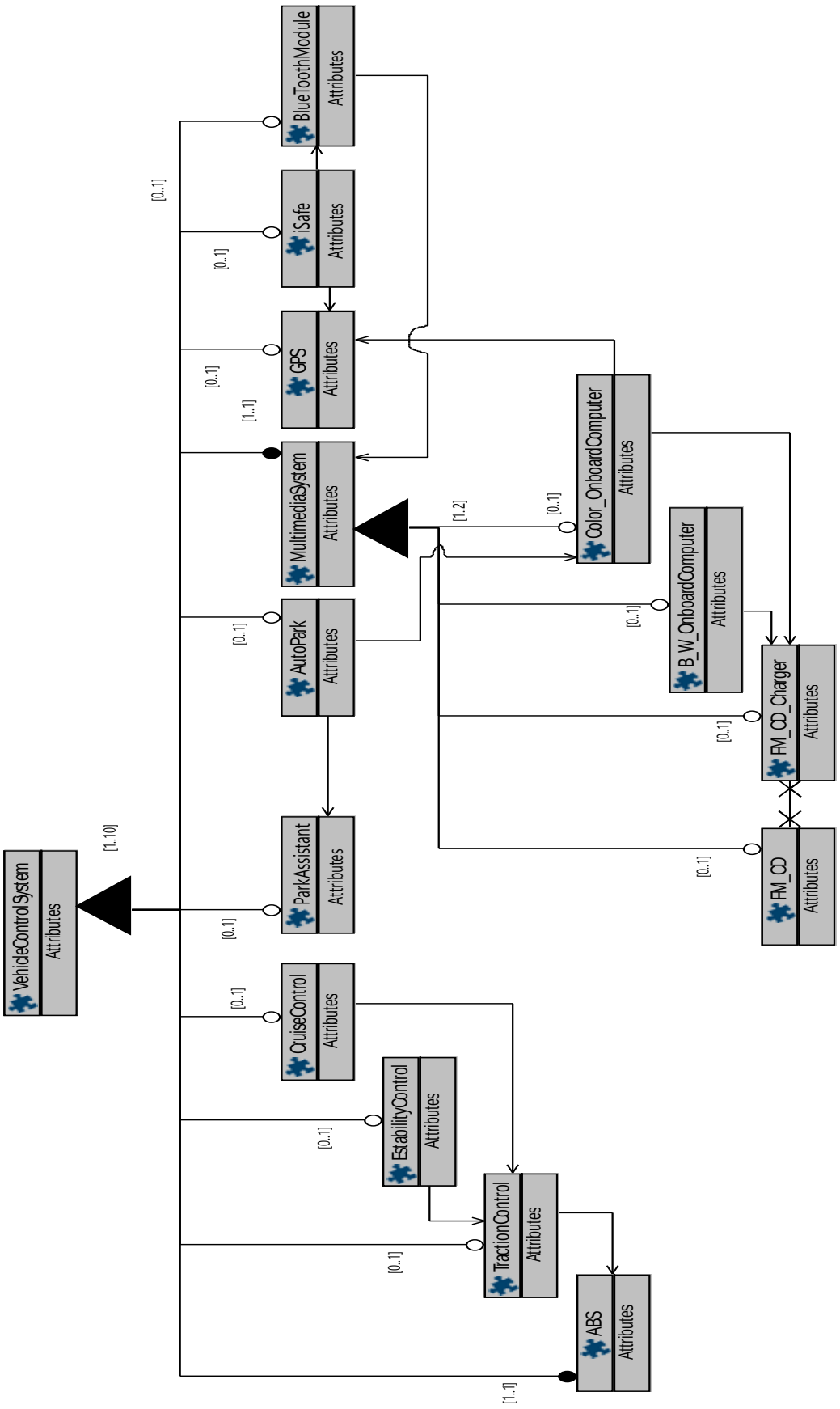


Figura 21. Modelo de características del Vehicle Control System (González-Huerta 2014)

5.2 Creando un proyecto ATL en Eclipse

En este apartado se presenta un pequeño tutorial que permite recrear la creación de un proyecto ATL en Eclipse, así como los pasos a seguir para realizar la ejecución de las reglas y conseguir el modelo de resolución.

En primer lugar, debemos descargar la herramienta a utilizar, QuaDAI framework seleccionando la plataforma a utilizar. Para este proyecto se ha utilizado el framework para Windows, ya que el funcionamiento en Mac OS no era correcto, encontrando múltiples problemas para poder configurar la herramienta y utilizarla adecuadamente.

Posteriormente es necesario instalar la plataforma ATL, por lo que es necesario instalar un *plugin* en Eclipse, lo podemos hacer de dos maneras, la primera es la forma tradicional, usando la opción de *install new software* (recomendada), o instalar ATL haciendo uso de la opción inmediatamente inferior que es *install modeling components* y seleccionar la opción de ATL.

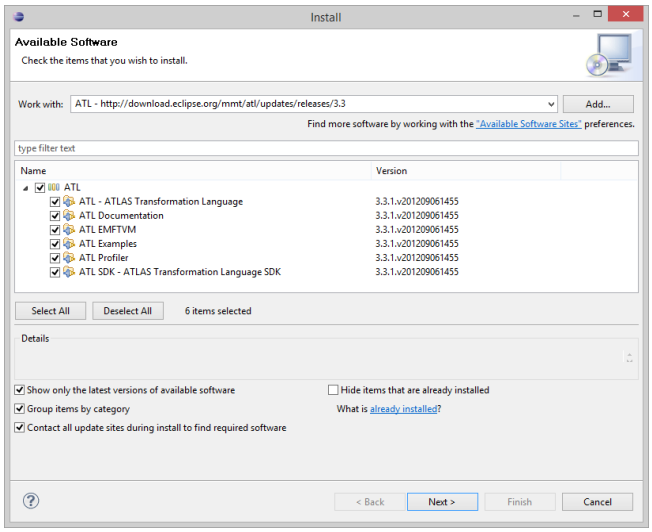


Figura 22. Instalación de ATL

Si elegimos la primera opción hay que usar el siguiente *update site:* http://download.eclipse.org/mmt/atl/updates/releases/3.3

Una vez esté la plataforma lista para ser usada en nuestro sistema, el procedimiento no es muy diferente de, por ejemplo crear un proyecto en Java usando este IDE, por lo que se trata de un proceso relativamente sencillo y familiar para aquellos desarrolladores que utilizan habitualmente Eclipse.

En primer lugar, para crear un nuevo proyecto ATL, debemos seguir este *camino* File < New < ATL Project.

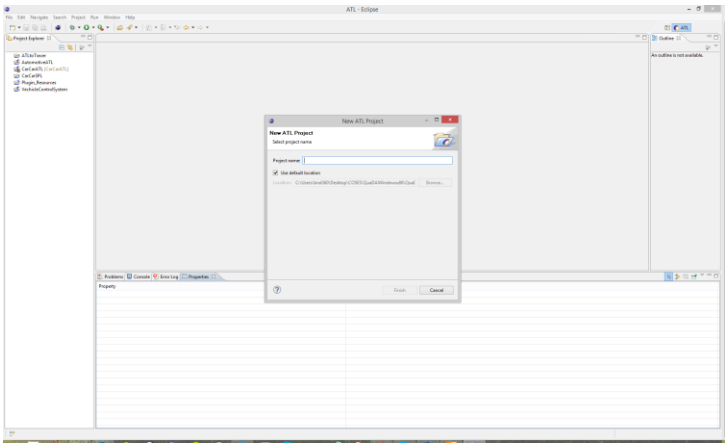


Figura 23. Crear un nuevo proyecto ATL

Una vez indicado el nombre del proyecto, que en este caso es AutomovilATL, vemos cómo se crea en el explorador de archivos que tiene Eclipse una nueva carpeta con dicho nombre.

Ahora, es el momento de añadir a nuestro proyecto el modelo de características. El fichero adjunto 'VCS_V3.automotivemultimodelcvl', es el que contiene el modelo. Sólo debemos importarlo a la carpeta creada.

El siguiente paso, es crear un fichero ATL, para poder plasmar en él las transformaciones y reglas a desarrollar. Basta con seleccionar la carpeta AutomovilATL, y con el botón derecho New < Other < ATL File.

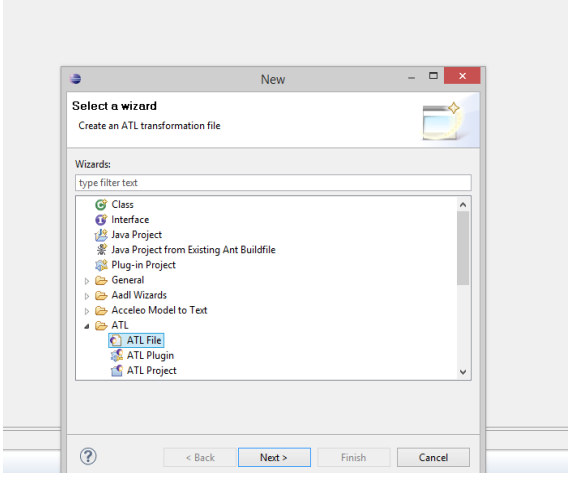


Figura 24. Crear un fichero ATL

Hacemos click en *next*, con el fin de poder configurar el fichero para que las reglas funcionen, es decir, hay que indicar el nombre del fichero (mymatchedrules), posteriormente los modelos, tanto el de entrada o modelo de características, como el modelo sobre el cual se va a apoyar el modelo de resolución.

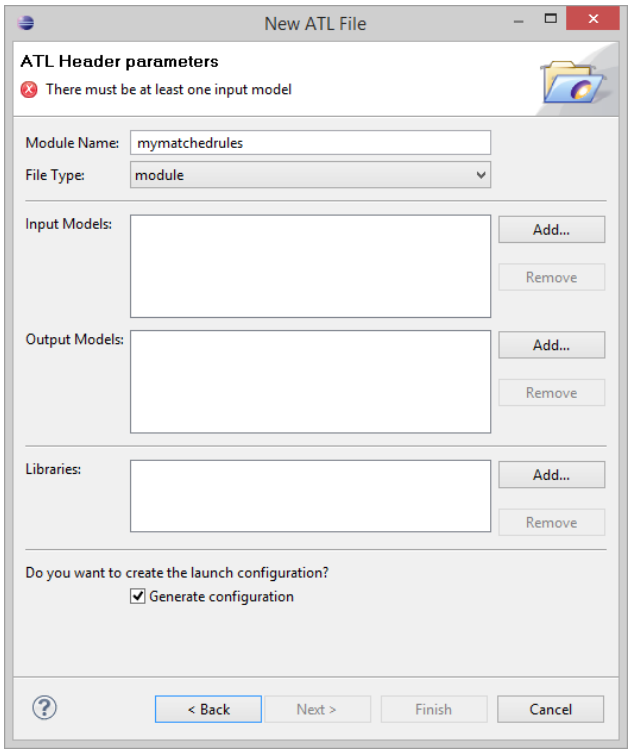


Figura 25. Configuración del fichero ATL

Para realizar correctamente lo expuesto, en primer lugar, al *clickar* sobre *Add* en el apartado de *Input Models,* aparece la siguiente ventana, en la que seleccionamos el multimodelo.

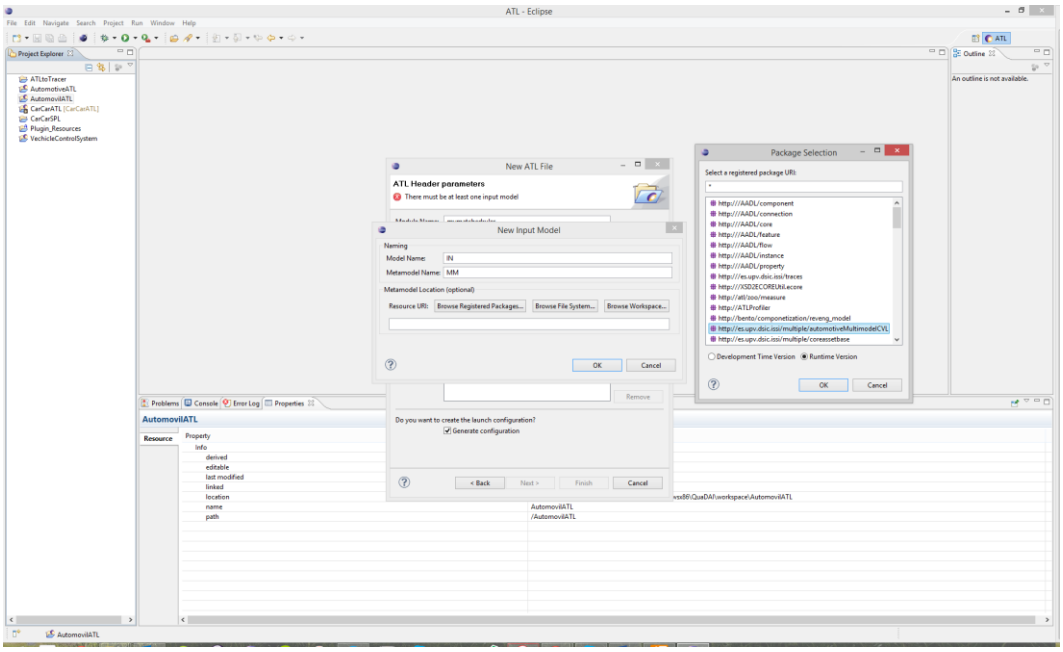


Figura 26. Selección del modelo de entrada

Para añadir el modelo de salida, debemos seguir el mismo proceso, pero en este caso no hay que seleccionar el mismo modelo. Hay que seleccionar el modelo estándar CVL.

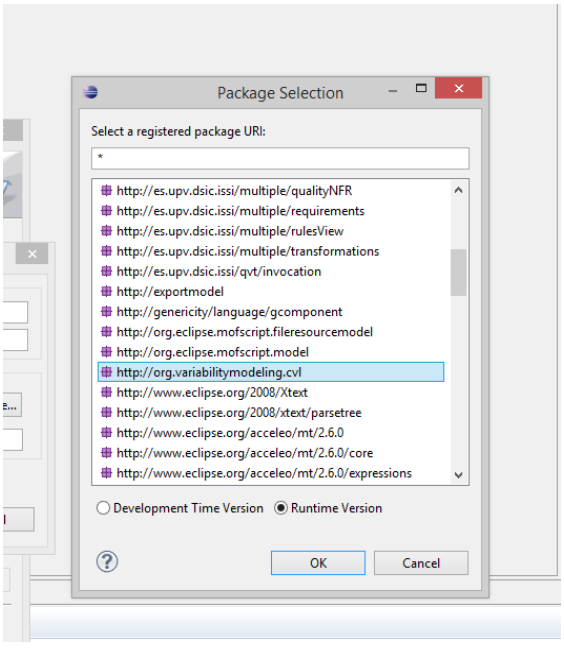


Figura 27. Selección del modelo de salida

En este punto, es el momento de establecer la configuración del proyecto para poder realizar la ejecución correctamente. Para ello, hay que acceder a las *run configurations*. Sobre la carpeta AutomovilATL y con el botón derecho, buscamos *Run as* para acceder a

la configuración e indicamos sobre qué fichero dentro de la carpeta se va a aplicar la configuración.

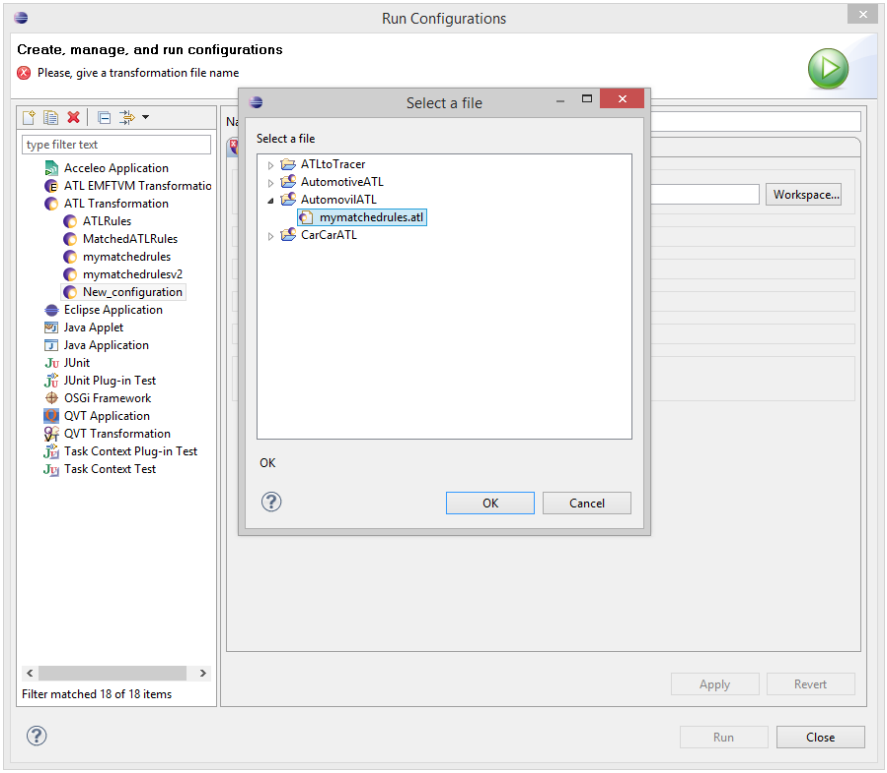


Figura 28. Run configuration

Posteriormente confirmamos que MMin y MMout se correspondan con los modelos que hemos añadido anteriormente, y ahora sólo hay que seleccionar el *Source Model*.

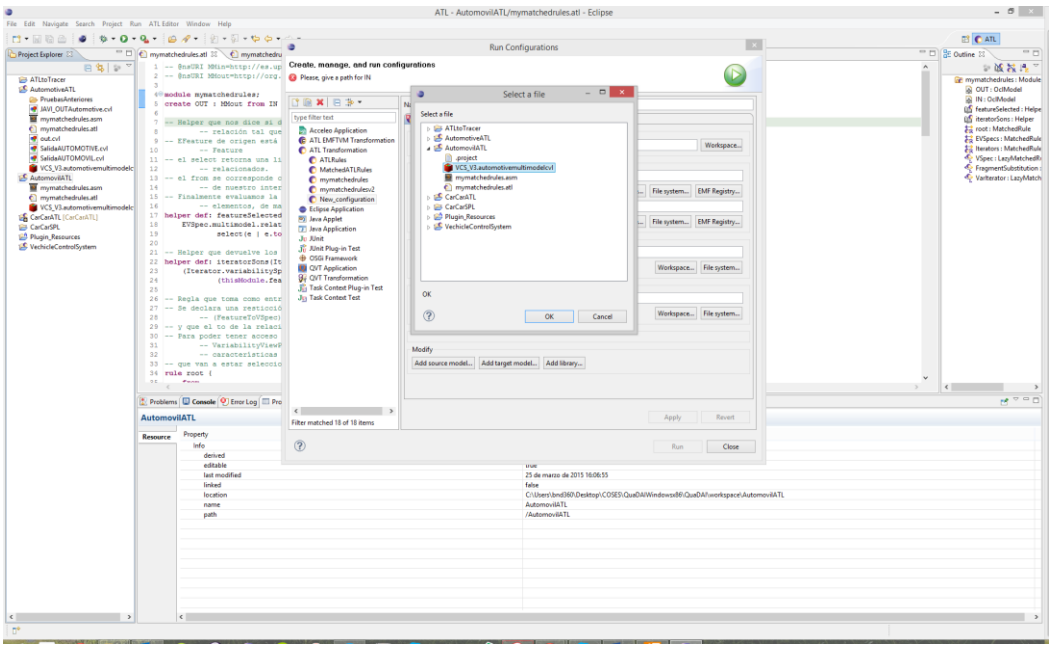


Figura 29. Selección del source model

A continuación se crea el modelo de salida, es decir, hay que darle nombre al fichero sobre el cual se va a *pintar* el modelo de resolución. Este debe tener la extensión .cvl y debe de estar dentro de la carpeta del proyecto.

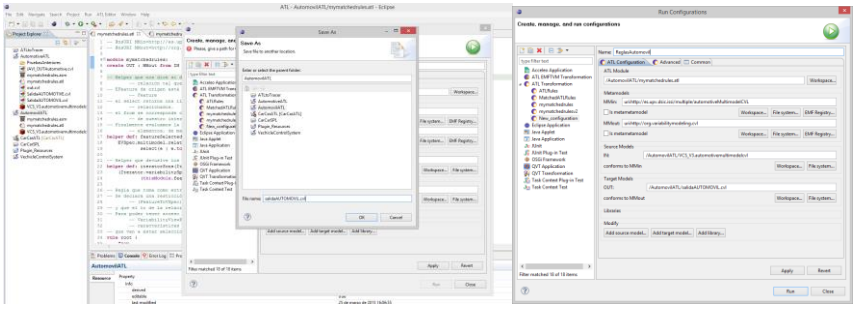


Figura 30. Creación del modelo salida y configuración final del fichero

Ahora ya somos capaces de ejecutar las transformaciones para obtener el modelo de resolución.

5.3 Ejemplos con diferentes configuraciones de producto

Para ilustrar el funcionamiento vamos a disponer de tres ejemplos con los que vamos a ver gráficamente el funcionamiento de la aplicación.

5.3.1 Configuración primera: ABS, MultimediaSystem y FM_CD

En primer lugar vamos a configurar nuestro producto seleccionando las siguientes características: ABS, MultimediaSystem y FM_CD presentes en el multimodelo de entrada tal y como se muestra en la Figura 31.

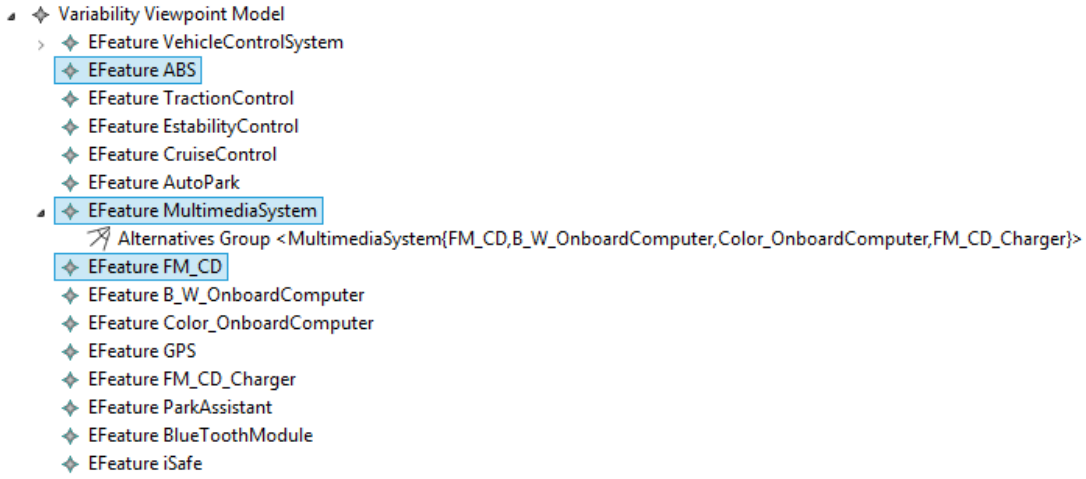


Figura 31. Elementos que vamos a seleccionar

Para indicar una característica en la herramienta de edición del multimodelo tendremos que darle valor verdadero a la propiedad seleccionada de dicha característica. A modo de ejemplo gráfico, mostramos en la Figura 32.

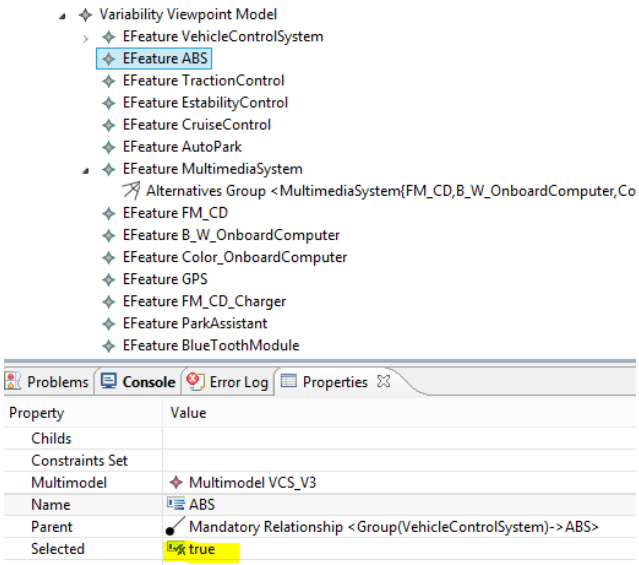


Figura 32. Selección de la EFeature ABS

Una vez establecida la configuración, se ejecutan las transformaciones pertinentes, esto es, ejecutar las reglas ATL que hemos definido para la resolución de la variabilidad y de esta forma obtener el modelo de resolución.

En la Figura 33 se muestra el modelo de resolución CVL para esta primera configuración obtenida. Hay que tener en cuenta que, se generan elementos que a priori, podríamos pensar que no deberían generarse ya que no los hemos seleccionado; por ello es importante fijarse en que dentro de las características seleccionadas pueden existir otros elementos, de manera que la selección del elemento contenedor implica de forma implícita la selección de estos.

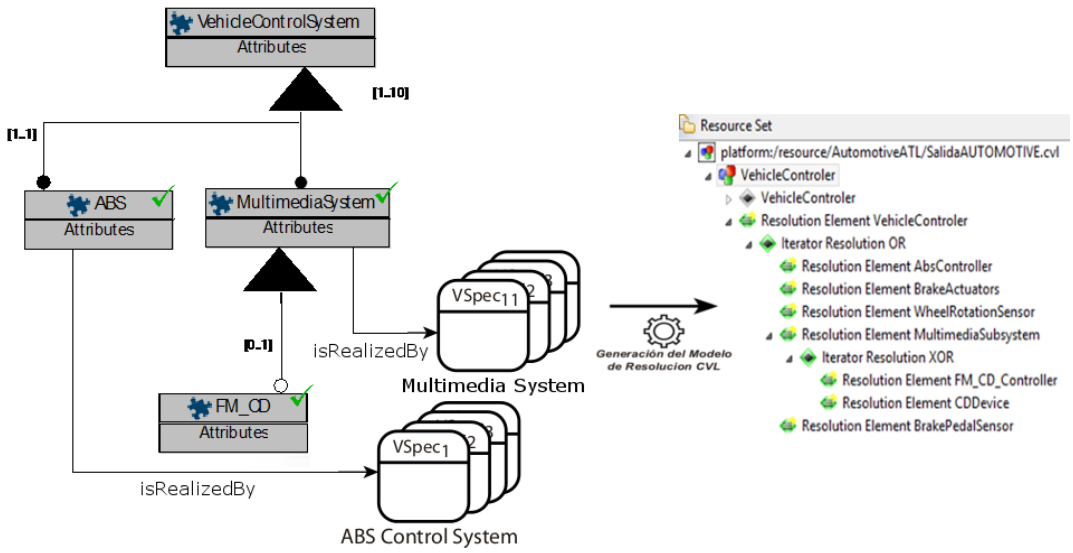


Figura 33. Extracto del modelo de características y su resolución

5.3.2 Configuración segunda: ABS, MultimediaSystem y GPS

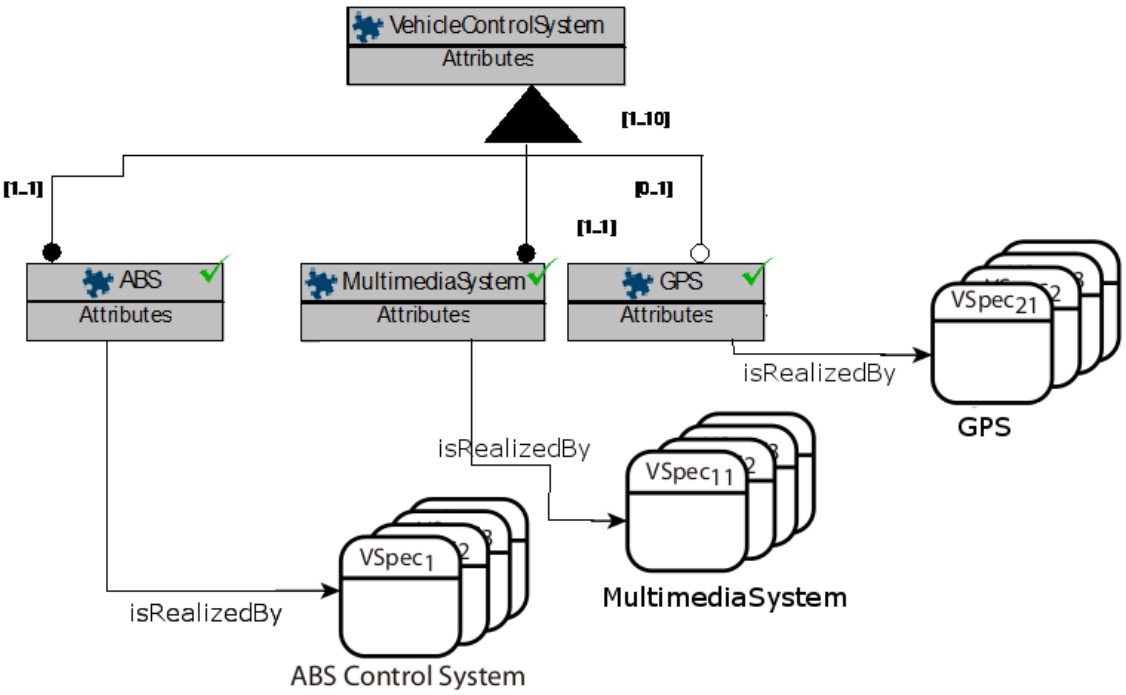


Figura 34. Extracto del modelo de características con los elementos seleccionados

Si pensamos en otra configuración posible para el modelo de características podría ser la siguiente: ABS, MultimediaSystem y GPS, tal y como mostramos en la Figura 34, donde se ha desmarcado el FM_CD como elemento de interés.

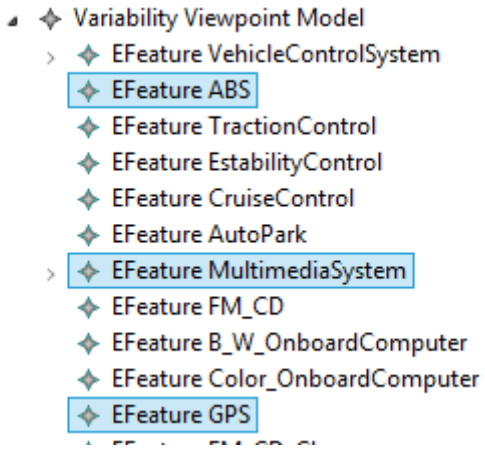


Figura 35. EFeatures seleccionadas en esta nueva configuración

Partiendo de esta nueva configuración de producto obtenemos el modelo de resolución que se muestra en la Figura 36.

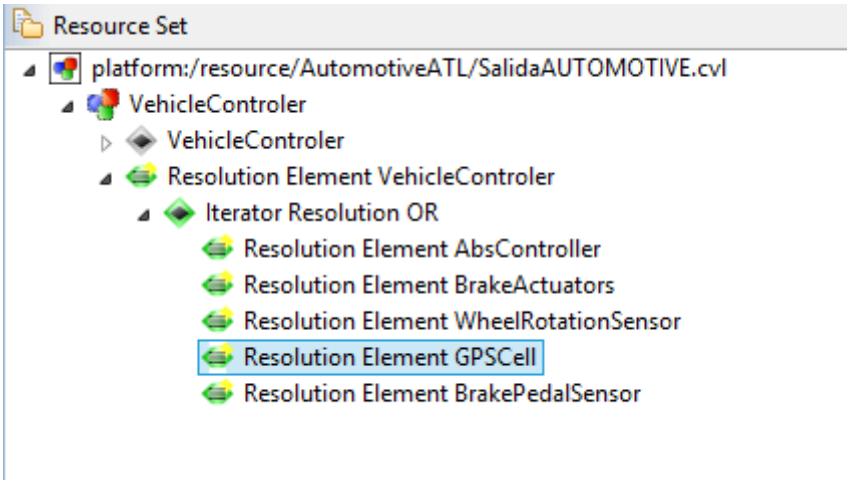


Figura 36. Modelo de resolución obtenido para la configuración de producto

Visible es que ahora éste, ha cambiado: en el caso anterior, dentro de la EFeature MultimediaSystem, existía un iterador con los elementos que ésta contiene. Como ahora ya no hay ningún elemento seleccionado dentro del contendor, ya no se crean los Resolution Elements asociados a esta característica, (FM_CD_Controller y CDDevice).

5.3.3 Tercera configuración: Configuración más completa posible

Para el último ejemplo que presentamos, vamos a seleccionar todas las características presentes en el modelo de entrada, excepto B_W_OnboardComputer y FM_CD.

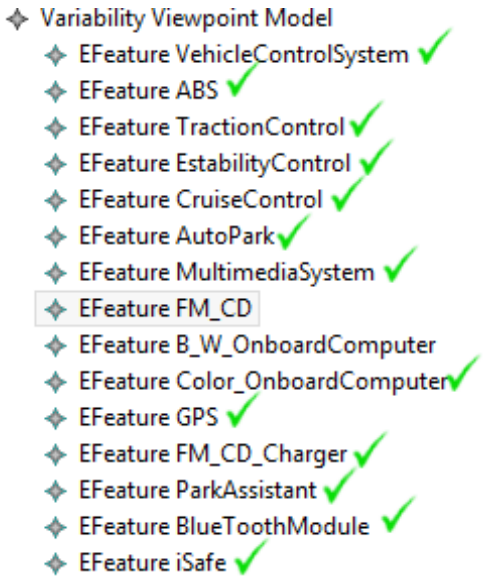


Figura 37. Feature que utilizamos en este ejemplo

Esta es la configuración más completa posible en la que se cumplen las restricciones del modelo de características que se muestra en la Figura 21.

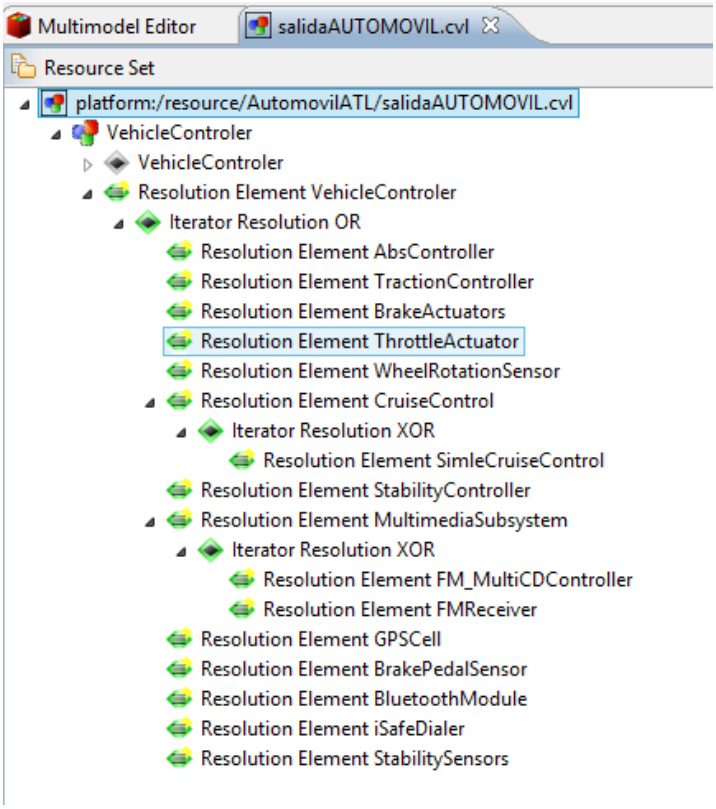


Figura 38. Modelo de resolución obtenido

Como se observa en el modelo CVL obtenido se tienen en cuenta los elementos previamente configurados en el modelo de entrada.

6 Conclusiones y trabajo futuro

En este trabajo se ha presentado una aproximación para obtener un modelo arquitectónico (sin variabilidad) a partir de un conjunto de artefactos software con variabilidad incorporada. La variabilidad se resuelve gracias al uso de una cadena de transformaciones de modelos compuestas por un conjunto de reglas, escritas en ATL, que permiten automatizar la derivación de la arquitectura software en entornos de líneas de producto, permitiendo obtener una primera versión del producto que se está desarrollando.

La aplicación de estas transformaciones, demuestra cómo utilizando los principios del Desarrollo de Software Dirigido por Modelos, se experimenta una mejora sustancial en el proceso de obtención del producto final, no obstante, hacer uso de las líneas de producto para afrontar el desarrollo de software no es una tarea trivial.

La problemática radica en los productos que admiten diversas variaciones: la capacidad para crear eficientemente múltiples copias de un producto determinado, sí es una tarea trivial, pero la capacidad para crear de forma eficiente múltiples variaciones de un producto, se convierte en un importante reto.

Sacar el máximo partido posible de las líneas de producto se puede conseguir si se aprovechan los elementos comunes y se gestiona de manera eficaz las variaciones. No se permite la flexibilidad presente en el desarrollo tradicional (pero que conlleva un gran coste), pero sí se optimizan los procesos para las variantes delimitadas por las líneas de producto.

Gracias a las líneas de producto se puede gestionar de manera eficiente y eficaz las diferentes variaciones que pueden existir entre los productos, permitiendo a las empresas centrarse en un dominio, y no en un producto. El reto está en delimitar el ámbito, en identificar las variaciones que se van a soportar y construir la infraestructura que permita producir el producto a un coste relativamente bajo pero manteniendo las cotas de calidad elevadas.

Los objetivos principales propuestos en la introducción se han cumplido:

- **Se ha desarrollado una cadena de transformaciones que resuelven la variabilidad de producto.**
  Se presenta una aproximación para derivar automáticamente modelos de resolución CVL para una configuración de producto dada usando transformaciones ATL.

- **El segundo objetivo, obtener el modelo de resolución CVL a partir del modelo de características, también se ha alcanzado.**
  Se utiliza una línea de productos software de ejemplo, a partir de la que se obtienen los modelos de resolución, es base a una configuración establecida.

- **El tercer objetivo, referente a los Requisitos No-Funcionales, aunque se considera cumplido, no lo está con el grado suficiente.**
  Si es cierto que se introducen los Requisitos No-Funcionales, y que el resultado obtenido es correcto, pero no aseguramos que para obtención del producto final

éstos se cumplan de forma adecuada. Este es uno de los puntos a mejorar en el presente trabajo.

En cuanto a los subjetivos que se identifican, afirmamos que también se cumplen de forma satisfactoria.

- Se ha realizado una pequeña introducción acerca del proceso de desarrollo basado en LPS y su funcionamiento.
- El desarrollador ha estudiado un nuevo lenguaje de transformación, desconocido para él hasta el momento, concluyendo que las operaciones y uso básico del mismo se comprende, se trata de un lenguaje con un gran potencial que debe seguir siendo investigado.
- Se presentan métodos y herramientas con el fin de facilitar el desarrollo de las reglas que van a permitir realizar las transformaciones.

Para finalizar estas conclusiones, se cree que el resultado final del proyecto es satisfactorio, teniendo en cuenta que el trabajo ha sido llevado a cabo por un desarrollador inexperto en la este campo. Ha habido momentos de estancamiento en los que se ha hecho difícil avanzar, ya que no se lograba encontrar posibles fallos y errores que provocaban problemas a la hora de obtener la solución esperada, (una regla puede interferir en el funcionamiento de otra, una llamada a un método en un lugar incorrecto puede llevar a resultados muy diferentes a los esperados, etc.), y conseguir corregirlos no ha sido tarea fácil.

El núcleo de este trabajo fue enviado al ACM-SRC (Student Research Competition) de la Conferencia *ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems,* en la en la categoría de *Undergraduate Student*, obteniendo el segundo puesto. El artículo se ha publicado en el volumen: *Joint Proceedings of MODELS 2014 Poster Session and the ACM Student Research Competition (SRC), Valencia, Spain, September 28 - October 3, 2014.* Publicado en CEUR-WS Vol. 1258. ISSN 1613-0073 con el título: *"Towards the Automatic Resolution of Architectural Variability in Software Product Line Architectures through Model Transformations"*.

Esta fue una experiencia única e irrepetible, que permitió conocer a grandes investigadores dentro del mundo del desarrollo de software dirigido por modelos.

La ACM SRC co-patrocinada por Microsoft Research, es un *fórum* en el que los estudiantes pueden mostrar sus trabajos de investigación, intercambiar ideas y competir por los premios que se otorgaban a los mejores trabajos.

Para poder optar a la participación en la conferencia, hay que ser estudiante y miembro de la Association for Computing Machinery (ACM). La competición presenta dos categorías, una para estudiantes graduados y otra para los que todavía no lo están.

Para poder participar, en primer lugar, se envía un resumen en inglés de la investigación, en el que los temas principales de la conferencia estuviesen presentes. Este *paper* debía contener los siguientes puntos: motivación y el ámbito de la investigación, trabajos relacionados, contribuciones y las conclusiones El contenido principal del documento, no podía excederse de las 5 páginas más referencias.

Una vez enviado el documento, una serie de expertos revisarían el trabajo, por el procedimiento *"peer-review"* y seleccionarían aquellos susceptibles de ser presentados en el ACMS-SRC del MODELS 2014.

Si se era aceptado se accedía a la competición, en la que había que presentar un poster. Esta presentación, ya se realizaba en la misma conferencia. Gracias a la presentación del poster, se proporcionaba una oportunidad de presentar la investigación a los asistentes a la conferencia. Los miembros del comité de evaluación, revisaban los posters además de realizar preguntas a los participantes acerca del trabajo, para posteriormente evaluar el mismo en base a la calidad, novedad e importancia. También tenían en cuenta la capacidad del estudiante a la hora de expresarse y realizar las explicaciones pertinentes.

Tras ser evaluado de forma satisfactoria en esta primera ronda, se podía seguir adelante en la competición, pasando a una segunda ronda en la que los estudiantes seleccionados, debían dar una pequeña presentación sobre su trabajo ante los miembros del jurado y los asistentes a la conferencia.

La evaluación final se realizaba en base a los conocimientos demostrados por los participantes (dentro de su campo de investigación), la contribución y la calidad de la presentación. En cada categoría había tres finalistas, recibiendo premios de 500, 300 y 200 US$, respectivamente.

Desde el punto de vista del participante se trata de evento único ya que permite vivir una experiencia totalmente diferente a lo que se pueda esperar de un evento de dichas características: permite conocer grandes investigadores dentro del mundo de la informática y del desarrollo de software dirigido por modelos, y asistir a sus presentaciones. Y por otra parte, proporciona la oportunidad de conocer a estudiantes que comparten tu campo de investigación, permitiendo compartir ideas, profundizar en las mismas, ampliar conocimientos, mejorar las habilidades de comunicación.

6.1 Trabajo futuro

Una vez expuestas las conclusiones, no se puede terminar sin presentar una propuesta de investigación para el futuro. Un proyecto final de carrera, o trabajo final de grado no debería quedar nunca 100% cerrado, sobretodo si se tiene en cuenta el ámbito de trabajo, que está en pleno auge.

Obviamente, uno de los puntos a cumplimentar en una continuación y extensión del trabajo es poder abarcar los atributos de calidad presentes, y estudiar cómo y qué relación existe entre los diferentes requisitos y poder establecer en qué grado uno, puede afectar a otro.

Por otra parte, *buscar* y estudiar la posibilidad de utilizar otras herramientas y frameworks que puedan facilitar la tarea de desarrollo. Dentro de punto, ya se ha indagado vagamente en algunas herramientas (ej, anATLyzer), aunque los posibles contratiempos que podrían surgir a la hora de realizar una migración de plataformas, o problemas de funcionamiento al añadir plugins, etc. se optó por seguir adelante con las herramientas actuales.

Otro trabajo futuro, que sería interesante contemplar es la posibilidad de mejorar la herramienta o framework para facilitar la tarea de configuración al máximo, con el fin de

minimizar posibles errores y/o reducir costes temporales, obteniendo una mayor productividad final.

Una vez realizada la extensión expuesta del presente proyecto, con todas las áreas cubiertas, estaríamos en disposición de dar un paso hacia adelante: Adaptar y desarrollas las transformaciones necesarias para ser aplicadas en un contexto de desarrollo de servicios *cloud*.

El fin de este nuevo nivel, es el de llevar a cabo la derivación de servicios cloud a partir de otros servicios cloud existentes que incorporan variabilidad (bien porque tenemos múltiples servicios con la misma funcionalidad y distintas propiedades no-funcionales o bien porque los propios servicios incorporan la variabilidad mediante interfaces de configuración) atendiendo tanto a los requisitos funcionales como a los no funcionales y que darán como resultado, los modelos de coreografía y orquestación del nuevo servicio.

Bibliografía

Asikainen, T., Soininen, T. & Männistö, T., 2003. A Koala-based approach for modelling and deploying configurable software product families. In *5th International Workshop on Product-Family Engineering*. Sienna, Italy, pp. 225–249. Available at: http://link.springer.com/chapter/10.1007/978-3-540-24667-1_17 [Accessed October 13, 2013].

Atkinson, C., Bayer, J. & Muthig, D., 2000. Component-based product line development: the KobrA approach. In *1st Internation Conference on Software Product Lines*. Denver, Colorado, pp. 289–309. Available at: http://link.springer.com/chapter/10.1007/978-1-4615-4339-8_16 [Accessed August 17, 2013].

Bass, L., Clements, P. & Kazman, R., 1998. *Software Architecture In Practice*, Addison-Wesley.

Botterweck, G., Lee, K. & Thiel, S., 2009. Automating Product Derivation in Software Product Line Engineering. In *Software Engineering Conference*. Kaiserslautern, Germany, pp. 177–182. Available at: http://subs.emis.de/LNI/Proceedings/Proceedings143/P-143.pdf?iframe=true&width=95%&height=95%#page=194 [Accessed August 14, 2013].

Brambilla, M., Cabot, J. & Wimmer, M., 2012. *Model-Driven Software Engineering in Practice Synthesis Lectures on Software Engineering*, Morgan & Claypool Publishers.

Cabello, M.E., 2008. *Baseline-Oriented Modeling: una Aproximación Mda Basada en Líneas de Productos Software para el Desarrollo de Aplicaciones*, PhD Thesis, Departamento de Sistemas Informáticos y Computación, Universitat Poltècnica de València.

Clauß, M., 2001. Modeling variability with UML. In *Young Researchers Workshop, Generative and Component-Based SW Engineering*. Messe Erfurt, Erfurt, Germany. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.109.3401&rep=rep1&type=pdf [Accessed August 17, 2013].

Clements, P. et al., 2011. *Documenting software architectures: views and beyond* 2nd Editio., Pearson Education. Available at: http://dl.acm.org/citation.cfm?id=599933 [Accessed August 4, 2013].

Clements, P. & Northrop, L., 2001. *Software Product Lines: Practices and Patterns*, Addison-Wesley Professional.

Combemale, B. et al., 2012. Using CVL to operationalize product line development with reusable aspect models. In *Workshop on Variability Modeling Made Useful for Everyone*. Innsbruck, Austria, pp. 9–14. Available at: http://dl.acm.org/citation.cfm?doid=2425415.2425418.

Duran-Limon, H.A., Castillo-Barrera, F.E. & Lopez-Herrejon, R.E., 2011. Towards an ontology-based approach for deriving product architectures. In *15th International Software Product Line Conference*. Munich, Germany, p. Volume 2, Article 29.

Available at: http://dl.acm.org/citation.cfm?id=2019169 [Accessed August 9, 2013].

Eclipse, 2013. Eclipse Modeling Framework Project (EMF). Available at: http://www.eclipse.org/modeling/emf/.

Favre, J.-M., 2004. Towards a Basic Theory to Model Model Driven Engineering. In *3rd International Workshop on Software Model Engineering*. Lisbon, Portugal.

Garzás Parra, J. & Piattini Velthuis, M.G., 2010. *Fábricas de Software: experiencias, tecnologías y organización* 2nd ed., RA-MA S.A.

Gómez, A., 2012. *Model Driven Software Product Line Engineering: System Variability View and Process*. PhD Thesis, Departamento de Sistemas Informáticos y Computación, Universitat Poltècnica de València.

González-Huerta, J., 2014. Derivación, Evaluación y Mejora de la Calidad de Arquitecturas Software en el Desarrollo de Líneas de Producto Software Dirigido por Modelos.

Van Gurp, J. & Bosch, J., 2002. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2), pp.105–119. Available at: http://dx.doi.org/10.1016/S0164-1212(01)00152-2 [Accessed August 17, 2013].

Horridge, M. et al., 2004. A Practical Guide To Building OWL Ontologies Using The Protégé-OWL Plugin and CO-ODE Tools Edition 1.0.

Janota, M. & Botterweck, G., 2008. Formal approach to integrating feature and architecture models. In *11th Conference on Fundamenteal Approaches to Software Engineering*. Budapest, Hungary, pp. 31–45. Available at: http://link.springer.com/chapter/10.1007/978-3-540-78743-3_3 [Accessed November 20, 2013].

Jouault, F., Allilaire, F. & Bézivin, J., 2006. ATL: a QVT-like transformation language. *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp.719–720.

Kang, K.C. et al., 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, CMU/SEI-90-TR-21 ESD-90-TR-222, Software Engineering Institute, Carnegie Melon University.

Kleppe, A., Warmer, J. & Bast, W., 2003. *MDA Explained: The Model Driven Architecture: Practice and Promise (Addison-Wesley Object Technology)*, Addison Wesley. Available at: http://www.amazon.co.uk/MDA-Explained-Architecture-Addison-Wesley-Technology/dp/032119442X [Accessed August 22, 2013].

Nascimento, A.S. et al., 2013. A Model-Driven Infrastructure for Developing Product Line Architectures Using CVL. In *7th Brazilian Symposium on Software Components, Architectures and Reuse*. Brasilia, Brazil, pp. 119–128.

Object Management Group, 2012. *Common Variability Language (CVL) OMG Revised Submission*,

Object Management Group, 2006. *Meta Object Facility (MOF) Core Specification*,

Ommering, R. van et al., 2000. Model for Consumer Electronics Software A component-oriented approach is an ideal way to handle the diversity of. *IEEE Computer*, 33(3), pp.78–85.

Perez, J., 2006. *PRISMA: Aspect-Oriented Software Architectures*. PhD Thesis, Departamento de Sistemas Informaticos y Computacion, Universitat Politècnica de València.

Perovich, D., Rossel, P.O. & Bastarrica, M.C., 2009. Feature model to product architectures: Applying MDE to Software Product Lines. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*. Cambridge, UK: Ieee, pp. 201–210. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5290806.

Pohl, K., Böckle, G. & van der Linden, F., 2005. *Software product line engineering*, Berlin: Springer.

Stahl, T. et al., 2006. *Model-Driven Software Development: Technology, Engineering, Management*, Wiley.

Svendsen, A. & Zhang, X., 2010. Developing a software product line for train control: a case study of CVL. In *14th Software Product Line Conference*. Jeju Island, South Korea, pp. 106–120. Available at: http://link.springer.com/chapter/10.1007/978-3-642-15579-6_8 [Accessed October 23, 2013].

Anexo I: Reglas para realizar las transformaciones

```
1.  @nsURI MMin=http://es.upv.dsic.issi/multiple/automotiveMultimodelCVL
2.  @nsURI MMout=http://org.variabilitymodeling.cvl
3.
4.  module mymatchedrules;
5.  create OUT: MMout from IN: MMin;
6.
7.  Helper que nos dice si dada una EVSpec (el parámetro de entrada es EVSpec), existe
8.  una relación tal que la EFeature de origen está seleccionada y existe dicho enlace,
9.  entre la EVSpec y la Feature el select retorna una lista con los featuretovspec tal
10. que el to y el from estén relacionados. El from se corresponde con la feature que
11. queremos que esté seleccionada, si no, no es de nuestro interés. Finalmente
12. evaluamos la lista que devuelve el select, para saber si esta contiene o n
13. elementos, de manera que cumplan las restricciones especificadas.
14.
15. helper def: featureSelected(EVSpec: MMin!EVSpec): Boolean =
16. EVSpec.multimodel.relationships -> select(e | e.oclIsTypeOf(MMin!FeatureToVSpec)) ->
17. select(e | e.to = EVSpec and e.from.Selected = true).isEmpty();
18.
19. Helper que devuelve los elementos que se corresponden con los hijos del iterador.
20. helper def: iteratorSons(Iterator: MMin!Iterator): Boolean =
21. (Iterator.variabilitySpecification -> select(e | e.oclIsTypeOf(MMin!EVSpec) and not
22. (thisModule.featureSelected(e)))) -> size() > 0;
23.
24. -- Regla que toma como entrada 4 'elementos'.
25. Se declara una restricción, en la que se indica que el from de la relación
26. (FeatureToVSpec) debe ser igual a la Feature y que el to de la relación debe
27. coincidir con el EVSpec.
28. Para poder tener acceso a la EFeature, hay que navegar por el
29. VariabilityViewPointModel, que es el que contiene el conjunto de características que
30. van a estar seleccionadas o no, dependiendo de la configuración.
31.
32. rule root {
33. from
34.      VarViewpoint: MMin!VariabilityViewpointModel,
35.      EVSpec: MMin!EVSpec,
36.      Feature: MMin!EFeature,
37.      FeatToVSpec: MMin!FeatureToVSpec
38.      (
39.        FeatToVSpec.from = Feature and FeatToVSpec.to = EVSpec and
40.        Feature = VarViewpoint.rootFeature and
41.        not EVSpec.oclIsTypeOf(MMin!FragmentSubstitution)
42.     )
43. -- En la parte del 'to', creamos un CVLModel con los datos de interés que
44. podemos recoger de la parte anterior.
45. En este caso, se crea un Modelo CVL (que va a actuar como contenedor de nuestros
46. elementos).
47. Esto se indica en la resolutionSpecification, que se trata de una secuencia de
48. EVSpecs
49.
50. to
51.     CVLModel: MMout!CVLModel (
52.      name <- EVSpec.name,
53.      resolutionSpecification <- Sequence{EVSpec},
54.      variabilitySpecification <- EVSpec.refImmediateComposite().
55.        variabilitySpecification -> select(e | not e.
56.         oclIsTypeOf(MMin!FragmentSubstitution)) -> collect(e | thisModule.
57.          VSpec(e))
58.    )
59. }
60.
61. -- Regla en la que se tratan los EVSpecs.
62. En primer lugar, lo que hacemos es definir una restricción en la que nos aseguremos
63. que el padre de dicha EVSpec, no se corresponda con el modelo Arquitectónico.
64. Para ello hacemos uso del refInmediateComposite(), que es la forma de acceder al
65. elemento (que contiene), el contenedor, del elemento actual.
66. rule EVSpecs {
67. from
68.      EVSpec: MMin!EVSpec (
69.      -- Si el padre no es del tipo ArquitecturalViewpointModel
70.      not(EVSpec.refImmediateComposite()
71.       .oclIsTypeOf(MMin!ArquitecturalViewpointModel))and
```

```
72.       -- Y si el multimodelo no es oclUndefined, entonces llamamos al helper que
73.          comprueba si las features con las que tiene relaciones is_realized_by con
74.          esa EVSpec estan seleccionadas.
75.       if (not(EVSpec.multimodel.oclIsUndefined())) then
76.       not thisModule.featureSelected(EVSpec)
77.
78.       else
79.        false
80.       endif
81.      )
82.
83. Creamos un resolutionElement, este elemento se va a encargar de 'gestionar' los
84. Iterators presentes en el modelo.
85. to
86.     ResolutionElement: MMout!ResolutionElement (
87.     name <- EVSpec.name,
88.     --Hacemos un Union con una secuencia vacia para que solo se consideren
89.     iteradores en caso de existir, lo cual es gestionado mediante la selección
90.     de los elementos de tipo iterator de la VSpec.
91.
92.      resolution <- Sequence {} -> union(EVSpec.variabilitySpecification ->
93.        select(i | i.oclIsTypeOf(MMin!Iterator)))
94.    )
95. }
96.
97. -- Regla en la que por cada Iterador que cumpla la restricción del helper, crea un
98. -- IteratorResolution.
99. rule Iterators {
100.     from
101.     I: MMin!Iterator (
102.    -- LLamada al helper
103.     thisModule.iteratorSons(I)
104.    )
105.    to
106.     IR: MMout!IteratorResolution (
107.     name <- I.name,
108.      -- El choice es una secuencia de EVSpecs.
109.      Este debe contener aquellos que hemos seleccionado en el multimodelo de
110.      Entrada. Por tanto, el choice lo que al final va a tener es una serie de
111.      ResolutionElements que cumplan las restricciones del helper featureSelected.
112.
113.      choice <- Sequence{} -> union(I.variabilitySpecification ->
114.      select(e | e.oclIsTypeOf(MMin!EVSpec) and not (thisModule.featureSelected(e))))
115.     )
116.    }
117.
118.    -- Lazy rule encargada de crear la especificación de la variabilidad.
119.
120.    lazy rule VSpec {
121.     -- Por cada EVSpec presente en el modelo de entrada,
122.     from
123.      EVSpec: MMin!EVSpec -- Creamos una 'CompositeVariability'
124.     to
125.       VSpec: MMout!CompositeVariability (
126.       name <- EVSpec.name,
127.       -- La restricción indica que que este elemento debe ser de tipo iterador
128.          Se hace una llamada a la lazy rule VarIterator.
129.          Es en esta regla en la que se gestionan los iteradores, puesto que va
130.          a devolver una lista (o secuencia) de iteradores que cumplan las
131.          restricciones que se especifiquen.
132.        variabilitySpecification <- EVSpec.variabilitySpecification ->
133.         select(v | v.oclIsTypeOf(MMin!Iterator)) ->
134.          collect(e |thisModule.VarIterator(e))
135.      )
136.    }
137.
138.    -- Lazy rule encargada de generar los fragmentSubtitution
139.    lazy rule FragmentSubstitution {
140.     from
141.      EVSpec: MMin!FragmentSubstitution
142.     to
143.      VSpec: MMout!FragmentSubstitution (
144.       name <- EVSpec.name
145.      )
146.    }
147.
```

```
148.    Regla encargada de generar un Iterador por cada Iterator presente en el modelo.
149.    lazy rule VarIterator {
150.     from
151.      Iterator: MMin!Iterator
152.     to
153.      VSpec: MMout!Iterator (
154.      name <- Iterator.name,
155.      lowerLimit <- Iterator.refGetValue('lowerLimit'),
156.      upperLimit <- Iterator.refGetValue('upperLimit'),
157.      -- Hacemos llamada a la lazy rule fragmentSubtitution, que devuelve los
158.      elementos de tipo FragmentSubtitution presentes. De estos se seleccionan
159.      aquellos que
160.       variabilitySpecification <- Iterator.variabilitySpecification ->
161.        select(v | v.oclIsTypeOf(MMin!EVSpec)) -> collect(e | thisModule.VSpec(e)) ->
162.          union(Iterator.variabilitySpecification ->
163.           select(v | v.oclIsTypeOf(MMin!FragmentSubstitution)) ->
164.            collect(f | thisModule.FragmentSubstitution(f)))
165.       )
166.    }
```

Listado 10. Extracto del fichero ATL con las transformaciones desarrolladas