TRABAJO FIN DE MASTER

CURSO ACADÉMICO 2009 / 2010

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GESTIÓN DE LA VARIABILIDAD EN LÍNEAS DE PRODUCTO DINÁMICAS

Alumno: Alejandro Valdezate Sánchez
Tutor: Dr. Rafael Capilla Sevilla

A Mabel, por estar a mi lado

A Henry y Carmen, por tanto y tanto que me han dado

INDICE

1. Introducción

Hoy en día resulta impensable crear un software que no sea parametrizable de una forma u otra. Gran parte del valor añadido del software reside en su reutilización, y para ello nada mejor que poder aprovechar la creación de dicho software para diversos sistemas, generalmente relacionados. De esta manera, es posible ahorrar esfuerzo y costes cuando se pretenden crear productos software que sean similares.

Una de las técnicas de desarrollo con mayor éxito en la industria para crear múltiples productos software relacionados son las denominadas Líneas de Productos Software (SPL), las cuales posibilitan crear familias de productos similares entre a partir de una arquitectura común y que tratan de explotar los aspectos comunes y variables y fomentar la reutilización. Asimismo, esta reutilización de consigue creando en primer lugar dichos componentes reutilizables dentro de la línea de productos, y utilizar éstos para crear productos software configurables y adaptables a diferentes necesidades, pero todo ello dentro de segmentos de mercado concretos.

Actualmente, una mejora de las SPL tradicionales consiste en explotar la reconfiguración de las aplicaciones en tiempo de ejecución, lo que ha dado lugar al concepto de Líneas de Productos Software Dinámicas (DSPL). Este trabajo persigue describir los mecanismos característicos de una DSPL y contribuir con una solución para modelar la variabilidad software en tiempo de ejecución, mediante la adición de variantes y puntos de variación de forma dinámica, con el fin de favorecer su gestión en aquellos sistemas que varíen sus condiciones de contexto.

2. Estado del Arte

Hoy en día, las arquitecturas software son la pieza angular del proceso de diseño software para sistemas complejos. La construcción de arquitecturas de software es una tarea que se desarrolla de forma manual dado que es un proceso creativo asociado al diseñador o arquitecto software. Algunas definiciones del concepto de arquitectura del software son las siguientes:

"*Una arquitectura software implica la descripción de los elementos a partir de los cuales se construye un sistema, las interacciones entre ellos, un conjunto de patrones que guían su composición, y restricciones entre estos patrones*" [Shaw y Garlan, 1996].

"*La arquitectura software de sistema software es la estructura o estructuras del sistema, que comprende componentes software, las propiedades visibles desde el exterior de estos componentes y las relaciones entre ellos*" [Bass et al., 2003]"

Las arquitecturas software son básicamente un conjunto de componentes y conectores que describen un sistema software mediante diferentes vistas arquitectónicas [Soni et al., 1993], [Kruchten, 1995], [Soni et al., 1995], [Hofmeister et all., 2007], acorde a los intereses de diferentes actores interesados en la arquitectura. Actualmente, con la revisión del estándar IEEE 1471-2000 [IEEE 1471-2000], el nuevo estándar ISO/IEC 42010 [ISO/IEC 42010] considera a las arquitecturas como el resultado de un conjunto de decisiones de diseño [Kruchten, 1995], [Bosch, 2000].

Estas decisiones constituyen un conocimiento tácito que reside en la mente del arquitecto software y que es raramente documentado. Parte de la base de estas decisiones la constituyen los patrones y estilos arquitectónicos,

que son las piezas principales en las que se basa el proceso de diseño software. Además, existen arquitecturas de referencia que son una especie de plantillas o modelos para un determinado tipo de sistemas. Asimismo, en cuanto al proceso de creación arquitectónico, existen diversos pasos y métodos de realizarlo, siendo uno de éstos las líneas de productos software, tal y como describimos en el siguiente apartado.

2.1. Líneas de Productos Software

Una línea de productos software (SPL) es un método de desarrollo para un conjunto de productos que están estrechamente relacionados y centrados en un determinado segmente de mercado. Dado que los productos están relacionados (tienen funcionalidad o requisitos de usuario similares) hay un alto grado de aspectos comunes en una línea de productos [Sonnemann, 1995]. Una SPL comparte la misma arquitectura de las líneas de productos (PLA) para todos los productos y sus componentes. Estos componentes reutilizables se denominan “core assets” [Bosch, 2000]. Las SPL son también un conjunto de productos que comparten un conjunto común de requisitos, pero que exhiben una variabilidad significativa en sus requisitos [Griss, 2000].

La principal cualidad de las líneas de productos es que potencian la reutilización estratégica. Los *assets* son componentes de una línea de productos que van más allá de una mera reutilización de código. Cada producto de la línea de productos toma la ventaja del análisis, diseño, implementación, y pruebas.

Básicamente, el desarrollo de una SPL consiste en un ciclo de vida dual en el que en la primera fase (ingeniería del dominio) se construye la arquitectura de la SPL y los componentes reutilizables y en la segunda fase (ingeniería de aplicaciones) se construyen y ensamblan los productos, tal y como muestra la Figura 1 [Svein et al., 2008]

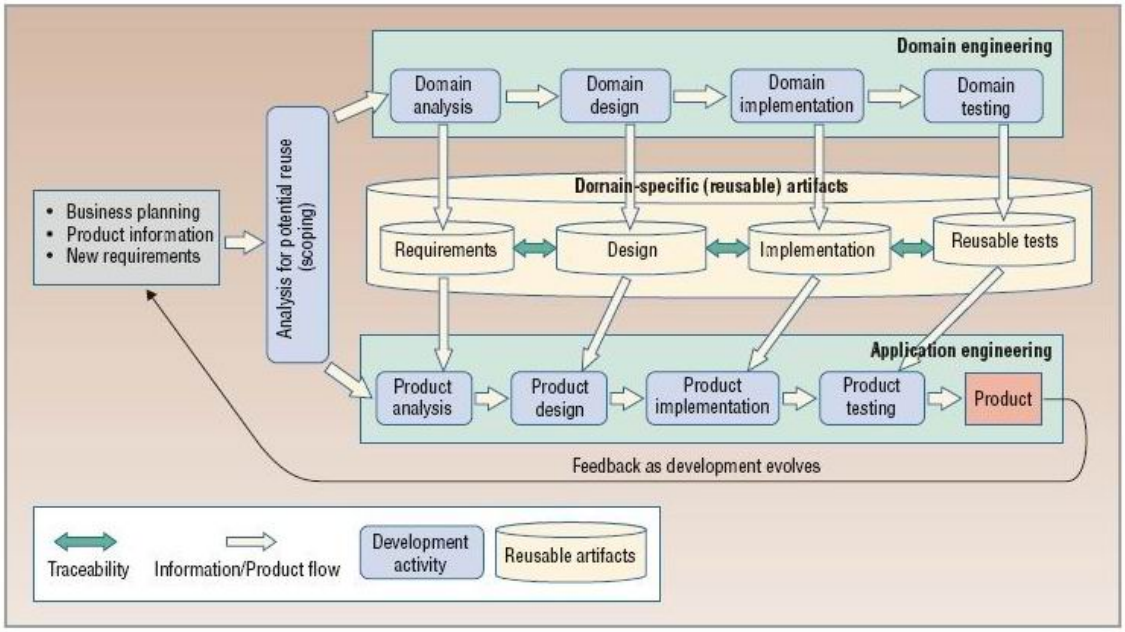


Figura 1. Enfoque del doble ciclo de vida aplicado a las líneas de producto.

El proceso de creación de la línea de productos de la Figura 1 se describe en los siguientes apartados.

2.1.1. Proceso de Ingeniería de Dominio

Es una actividad enfocada a la construcción de componentes reutilizables que se puedan utilizar posteriormente en el desarrollo de sistemas software. La principal ventaja del uso de componentes reutilizables es que puedan ser empleados en la construcción de varios sistemas de software en lugar de un único sistema.

Los procesos de ingeniería del dominio son procesos complejos que tratan de describir e implementar los conceptos de un dominio determinado (i.e.: por dominio entenderemos un conjunto de aplicaciones o sistemas relacionados que comparten un vocabulario común y unas características determinadas) y

producir una arquitectura común y un conjunto de componentes reutilizables. Los pasos principales de un proceso de ingeniería del dominio son los siguientes:

- **Análisis del Dominio**: El concepto de análisis del dominio tiene sus orígenes en los trabajos de James M. Neighbors sobre reutilización de software basados en el paradigma Draco [Neighbors, 1984] a principios de la década de los ochenta. Un proceso de análisis del dominio trata de identificar los objetos y operaciones de un dominio particular y obtener lo que se denomina modelo del dominio, consistente en una descripción más o menos formal de la información obtenida durante el proceso de análisis. Según Arango [Arango et. al., 1994] el Modelo del Dominio es un conjunto de definiciones de entidades, operaciones, eventos y relaciones clasificados que constituyen una taxonomía. En esta etapa se acota el dominio de la línea de productos y se descrien los conceptos. Hay que señalar que si el dominio es de sobra conocido no es necesario utilizar procesos complejos de análisis del dominio, los cuales pueden ser sustituidos por otros más sencillos.

.

- **Diseño del Dominio:** En esta etapa se desarrolla una arquitectura para la línea de productos que debe servir de referencia para todos los componentes y productos de la línea. Esta arquitectura implementa mecanismos de variabilidad software que permiten construir productos similares a partir de un único diseño. Para ello, es necesario realizar un análisis de comonalidad y variabilidad software con el fin de identificar los aspectos comunes y variables de los sistemas que van a formar parte de la línea.

- **Implementación del Dominio**: Es en esta fase donde se realiza la implementación de los componentes reutilizables de la línea de productos (denominados core assets) y se almacenan en un

repositorio. Estos componentes admiten mecanismos de personalización para adaptarlos a los diferentes productos. Estos componentes se derivan de la arquitectura de la línea de productos,

- **Pruebas del Dominio**: En estas pruebas se trata de comprobar que los core assets responden a la funcionalidad especificada.

2.1.2. Ingeniería de aplicaciones

Este fase trata del desarrollo de productos os sistemas software utilizando los componentes core desarrollados en la fase anterior.

- **Análisis del Producto**: En esta etapa de identifican y analizan los requisitos de un producto concreto que se pretende construir dentro de la línea de productos.

- **Diseño del Producto**: En esta etapa se personaliza la arquitectura software creada en la fase anterior para soportar los requisitos de los productos a construir. Esta personalización incluye la adaptación de los mecanismos de variabilidad implementados en la fase de ingeniería del dominio.

- **Implementación del Producto**: Esta etapa construye y ensambla el producto software a partir de los componentes core. La funcionalidad restante requerida y no soportada por los componentes core hay que desarrollarla de forma separada.

- **Test del Producto**: Finalmente, sólo resta realizar la fase de testing del producto final acorde a los requisitos especificados al inicio. Esta fase puede verse reducida ya que se supone que los tests de los componentes core han sido realizados en el ciclo anterior.

2.2. Variabilidad Software

Uno de los aspectos clave para el éxito de las líneas de producto es la implementación de mecanismos de variabilidad software en los productos que queremos construir. La variabilidad software [Svahnberg et al., 2001] representa la forma de describir las partes que diferencian a un componente o producto software de otro y que pueden personalizarse para construir productos similares con poco esfuerzo y anticipándose a cambios previsibles en el sistema. Una vez que en una PLA se han descritos los aspectos comunes a todos los productos de una SPL, la variabilidad permite diferenciar un producto de otro. La variabilidad software se describe en términos de puntos de variación y variantes, tal y como se define a continuación.

Punto de variación: Un punto de variación es un área de un sistema software afectado por variabilidad. Un punto de variación relaciona otros puntos de variación y variantes mediante una determinada fórmula.

Variante: Una variante es una característica visible de un sistema que puede variar y tomar un conjunto de valores admisibles especificados dentro del ámbito de la línea de productos. Por ejemplo, la potencia de un coche puede variar en un rango de valores admisibles según el modelo de vehículo que se pretenda fabricar.

La aplicación del concepto de variabilidad a las Líneas de producto permite modelar las partes comunes y variables de los sistemas y representarlos en la arquitectura a través de los denominados puntos de variación y variantes.

2.2.1. Representación de la Variabilidad

Para representar la variabilidad existen diversas formas y notaciones, siendo algunas de ellas las siguientes.

La metodología Featured Oriented Domain Análisis (FODA) [Kang, 1998] propone el uso de características (*features*) para describir la variabilidad de los sistemas. El concepto de variabilidad es un aspecto clave para la construcción de un conjunto de sistemas similares. La variabilidad debe reflejarse tanto en la arquitectura software como en los componentes reutilizables los cuales van a ser utilizados en la construcción de los diferentes sistemas software.

FODA pretende identificar características visibles para el usuario de manera prominente o distintiva dentro de una clase de los sistemas de software relacionados. Estas características conducen a la conceptualización del sistema de los productos que definen el dominio. Con FODA es posible representar relaciones entre componentes usando para ello diagramas de características y expresando las reglas de composición, decisiones y requisitos de diseño, todo ello a través de un catálogo de características de sistema.

FODA utiliza una representación en árbol para describir las características comunes y variables de los sistemas, las cuales modela con relaciones lógicas de tipo AND/OR/XOR y un conjunto de reglas o restricciones que deben cumplir los productos y que permiten relacionar las características que forman parte de un punto de variación y que son complejas de representar en el árbol. El nodo raíz del árbol FODA describe el producto software o tipo de productos cuya variabilidad se desea representar y las hojas o nodos sus características comunes y variables. Estas características suelen ser de tipo obligatorio, optativo o alternativo.

En Figura 2 se muestra un ejemplo de árbol FODA para la construcción de un sistema de monitorización de vehículos [Bontemps et al, 2004]. El producto a conseguir depende de dos puntos de variación que comprende dos características relacionada con la monitorización del motor y del combustible. El monitor de rendimiento del motor da lugar al monitor de temperatura (el cual se aplica al aceite, motor, transmisión, y de forma opcional, al refrigerante), al monitor de revoluciones y al monitor de niveles críticos. En el caso de del monitor de consumo de combustible, da pie a dos variantes, que son las medidas (a elegir entre litros por kilómetro recorrido o millas por galón) y métodos de medida (a elegir entre distancia, tipo de conducción y forma de conducción).

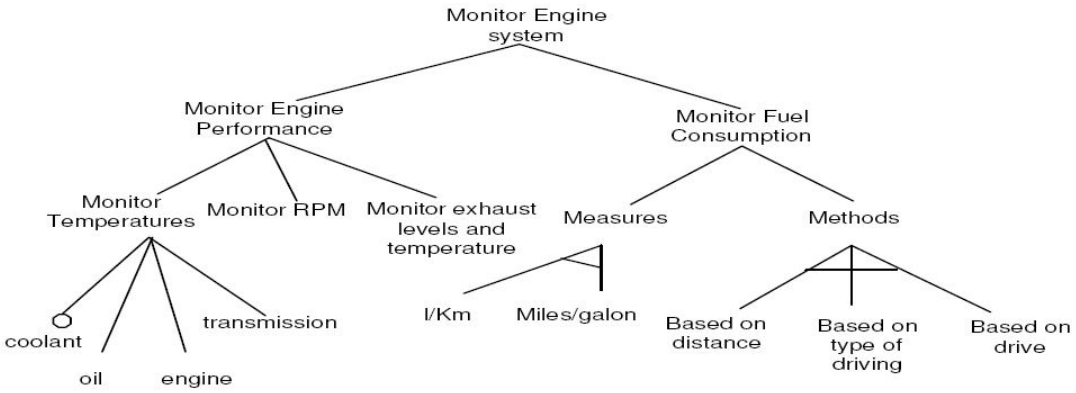


Figura 2. Árbol FODA para describir la variabilidad en la monitorización del motor de vehículos

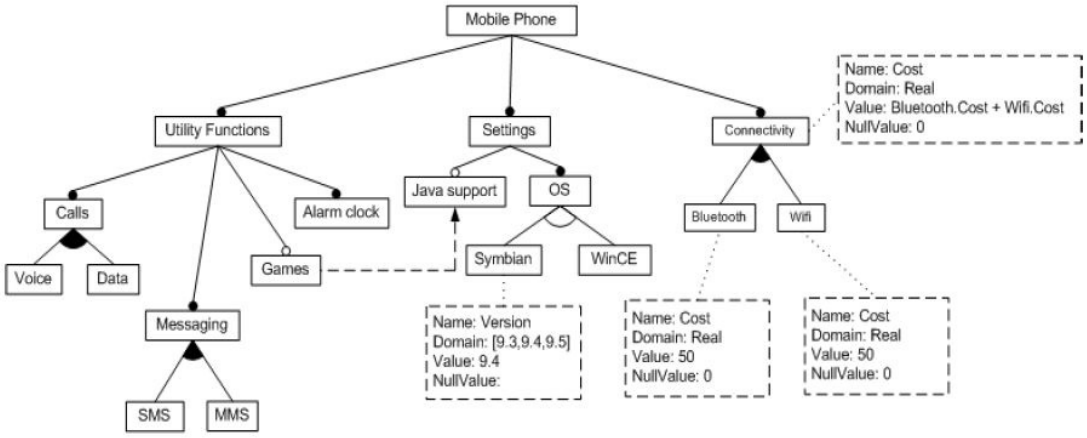
Por otro lado, a través de las reglas es posible expresar otras dependencias, como la exclusión de una característica debido a la presencia de otra, o la implicación de una característica cuando aparezca otra determinada. Las reglas son de tipo "*if-then-else*".

En la herramienta de modelado de características FAMA [Benavides et al. 2007] se describe una implementación de reglas if-then-else de esta forma:

Wifi.speed > 5 AND OS.version > 10 IMPLIES Bluetooth;

En la fórmula anterior se describe la siguiente condición: si se diera el caso de que la velocidad de la red WIFI fuera superior a 5 y la versión del sistema operativo fuera más reciente que la versión 10, el dispositivo de Bluetooth debería formar parte del teléfono móvil a construir.

Por otra parte, en la Figura 3 se muestra el modelado de reglas en un árbol FODA. Con una flecha discontinua se marcan las dependencias entre nodos. Además, en un apartado del sumario se describen las reglas del árbol. En este caso pueden verse tres reglas de dependencia (si el teléfono dispone de juegos, entonces debe llevar java, si la versión de Symbian es la 3, entonces no puede tener wifi, y si tiene Bluetooth y Wifi el coste sde la conectividad e calcula sumando el coste de Bluetooth y de Wifi.



% Constraints and invariants

Games IMPLIES JavaSupport;

Symbian.version==3 IMPLIES NOT wifi;

Bluetooth and Wifi IMPLIES Connectivity.cost= =(Bluetooth.cost +Wifi .cost);

Connectivity.cost == Bluetooth.cost + Wifi.cost;

Figura 3. Representación FODA y modelado de reglas de FAMA

Una extensión a FODA es el método FORM (Feature Oriented Reuse Method), que busca y captura características comunes y diferencias de aplicaciones en un dominio y utiliza los resultados del análisis para desarrollar arquitecturas del dominio y componentes. FORM se puede entender como la evolución de FODA hacia el paradigma de la orientación a objetos. FORM propone tres tipos de características (opcionales, obligatorias y alternativas) y tres tipos de relaciones entre ellas (compuesto-de, generalización e implementado-por). El tipo de relación implementado-por es realmente una dependencia entre requisitos de modo que la implementación de un requisito implica la implementación necesaria del segundo [Lee, 2000].

Debido a algunas limitaciones de los árboles FODA para representar valores cuantitativos, existen algunas mejoras que extienden FODA para incorporar dichos valores, permitiendo la inclusión de rangos de valores, repetición de un valor, y reglas que permiten modelar las relaciones para valores cuantitativos [Capilla, 2001].

Existen otros métodos para describir y representar la variabilidad software, de manera que la Figura 4 resume diversas notaciones propuestas por diversos autores.

FODA		FORM		GP		Featu-RSEB		J. Bosch		MEANING
	Manda-tory		Manda -tory f		Manda-tory		Compo-sed of		Compo-sition	**Mandatory** (if reachable, then f. must be chosen)
	Optio-nal		Optio-nal f		Optio-nal		Optio-nal f		Optio-nal f	**Optional** (if reachable, may be chosen, or not)
	Alterna-tive		Altern a-tive f		Alterna-tive		Vp-feature (XOR)		xor-speciali-zation	**One-of-many** choice from a group
-	-	-	-		Or-features		Vp-f use time bound (OR)		or-specia-lization	***n*-of-many** choice from a group
-	-	-	-	-	-	-	-		External feature	**External** Feature
-	-	-	-	[aFeature]	Open-feature	-	-	-	-	**Open** Feature
-	-	-	-		Prema-ture	-	-	-	-	**Premature** Feature

Figura 4. Diversas notaciones para representar la variabilidad

2.3. Implementación de la variabilidad

La variabilidad puede ser implementada a través de diferentes técnicas. [Anastasopoulos et al., 2001], [Sharp. 1999] , [Bosch, 2000], [Jacobson, 1997], [Karhinen et al. 97]. Algunas de ellas son las siguientes:

- Agregación
- Herencia
- Parametrización
- Macros
- Compilación condicional
- Configuración
- Carga dinámica de clases

La **Agregación** es una técnica orientada a objetos que permite a los objetos soportar virtualmente cualquier funcionalidad reenviando las peticiones que no pueden gestionar a los llamados objetos delegados, los cuales proveen del servicio solicitado. La variabilidad puede crearse poniendo la funcionalidad obligatoria en la clase delegante y la funcionalidad variante en la clase delegada. Esta técnica es recomendable para implementar características tanto opcionales como variantes, cuyo tiempo de ligadura es el tiempo de compilación. No obstante, el tiempo de ligadura puede pasarse hasta el tiempo de enlace usando la carga de clases dinámicas y bibliotecas de enlace.

La **Herencia** separa la funcionalidad común de las superclases y extensiones a las subclases. Hay al menos dos categorías de herencia soportadas por la mayoría de los lenguajes orientados a objetos.

- Herencia estándar (basada en clases): la funcionalidad común se mantiene en una superclase mientras que la funcionalidad variable se añade a las subclases que heredan de la superclase común.

- Herencia con funciones virtuales: Igual que la herencia estándar, pero las funciones pertenecientes a la clase virtual pueden definirse en la superclase y reemplazarse dinámicamente en las subclases.

Algunos lenguajes también soportan formas más raras de herencia como:

- Herencia múltiple: Una clase deriva de varias superclases.

- Herencia basada en mixins: Los mixins son similares a las clases ordinarias, pero solo definen diferencias par alas clases existentes. Los mixins se pueden combinar con clases existentes para ampliar su funcionalidad.

- Herencia basada en objetos: la herencia es transportada al nivel de los objetos en lugar de las clases.
- Herencia parametrizada: la superclase es un parámetro de una subclase la cual es un conjunto de una clase requerida donde la subclase es instanciada.

La separación de la variabilidad en clases derivadas se puede conseguir con todas las formas de herencia.

Parametrización. Una clase se parametriza con tipos que se determinan en la instanciación de la clase. La clase parametrizada contiene código común, el cual está diseñado para funcionar en tipos variables. Un ejemplo típico es la clase parametrizada “snack” la cual contiene elementos de un tipo que pueden ajustarse mediante un parámetro.

Macros. Hay dos formas básicas de macros que pueden usarse para conseguir la variabilidad. El uso de la macro de ajuste separa variabilidad en un uso de una macro predefinida. La funcionalidad común se guarda en la definición de la macro y la variabilidad se coloca en las llamadas a la macro, en lugares apropiados del código fuente usando los argumentos apropiados. La macro uso es parte de la funcionalidad común y la variabilidad se consigue con una implementación específica de la macro.

Compilación condicional. Se consigue controlando los trozos de código que serán incluidos o excluidos en el compilado del programa. Esta técnica separa la variabilidad en distintas secciones del código que son incorporadas usando variables de compilación. La funcionalidad común se coloca en áreas externas compiladas condicionalmente y ajustando apropiadamente las variables del compilador se incluyen las secciones del código deseadas.

Configuración. En configuración, el código fuente de cada variante es colocado en un fichero distinto. Las herramientas de gestión de la configuración se usan para alternar entre estas distintas implementaciones.

La **Carga dinámica de clases** es una propiedad de la máquina virtual de Java donde las clases no son cargadas en memoria hasta que deben ser usadas. El comportamiento del cargador de clases dinámicas puede ampliarse y controlarse en el código fuente para decidir en tiempo de ejecución que versiones de las clases deben cargarse.

Las **bibliotecas de enlace dinámico** (DLL) son bibliotecas que pueden cargarse en el espacio de direcciones de las aplicaciones a petición en tiempo de ejecución. La variabilidad puede gestionarse colocando variantes funcionales en DLL's separadas y escribiendo un código fuente que cargue la variante adecuada en la aplicación mientras está ejecutándose. Por tanto, todas las variantes no tienen que ser conocidos necesariamente en durante el tiempo de desarrollo. El enlazado dinámico es un servicio proporcionado por los principales sistemas operativos.

2.3.1. Binding Time

Lo que se conoce como "binding time" en líneas de producto es el momento en el que se realiza la variabilidad de un producto software. El binding time permite retrasar las decisiones de diseño [Svahnbert, 2005] a momentos más tardíos, lo que incrementa la flexibilidad de configuración y adaptación del producto software a distintas situaciones. Algunos de los "binding time" más relevantes son: Diseño, Implementación, Compilación, Integración y Ejecución. En la Figura 5 se describen posibles binding time para implementarlos en la variabilidad de los productos [Fritsch et al. 2002a].

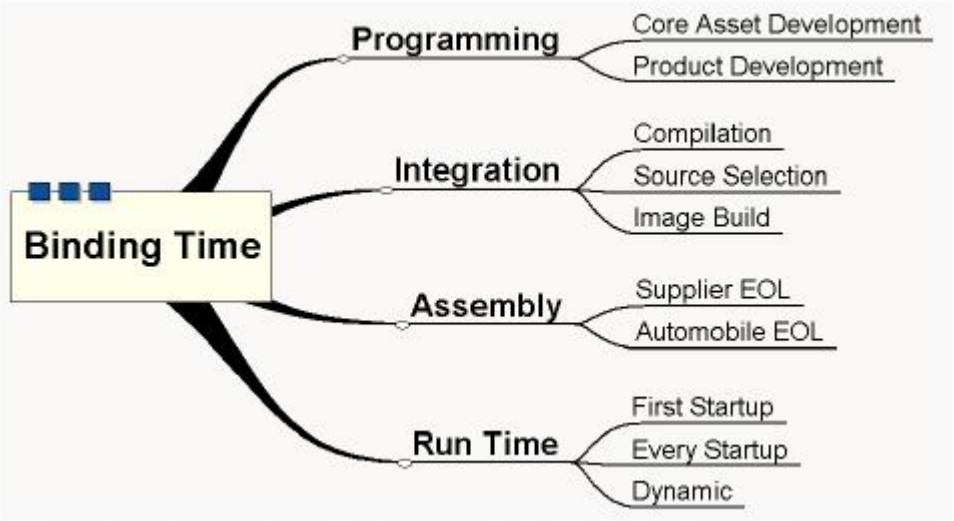


Figura 5. Ejemplos de binding time.

Esta figura está aplicada al binding time en la creación de software de un sistema integrado de automoción. Se consideran cuatro etapas, siendo estas programación, integración, ensamblado y tiempo de ejecución. En la etapa de programación, el desarrollo de componentes core puede tener lugar mucho antes que el desarrollo el producto, por lo que son binding times distintos. En la etapa de integración, se construye el ejecutable partiendo de varias fuentes (código, datos, ficheros de recursos y componentes de terceros). En el ensamblado, el software se puede configurar al final de la línea, sobrescribiendo partes de la memoria flash. En tiempo de ejecución, hay tres binding times. En la puesta en marcha inicial y sucesivas, donde el producto puede adaptarse al hardware presente, o de forma dinámica cuando se detecte un cambio de hardware.

2.4. Líneas de Producto Dinámicas

Hasta ahora, con las líneas de productos tradicionales, una vez tomada la decisión de generar el producto final, se producía el binding de la variabilidad y se realizaban las decisiones de diseño pospuestas, de manera que

cualquier cambio implicaba modificar el modelo de variabilidad descrito en la arquitectura.

Hoy en día, los aspectos dinámicos de los sistemas tienen cada vez más importancia, tales como aquellos sistemas sensibles al contexto, sistemas adaptativos, sistemas que se auto reparan, sistemas ubicuos o la propia naturaleza cambiante de los sistemas Web basados en aspectos de calidad que varían en diferentes condiciones. Estas mejoras suponen un ahorro económico importante, gracias a que los sistemas poseen dispositivos de autodiagnóstico que son capaces de detectar cambios y fallos y reaccionar para solventar dichos problemas ellos mismos. Por otro lado, también suponen un ahorro de tiempo y recursos, pues es posible aunar en un mismo dispositivo la funcionalidad de varios, de forma que el mismo sistema sepa qué rol debe desempeñar en cada momento, adaptándose a las condiciones particulares. Para ello, cada vez es más común que estos sistemas tomen información del exterior y elaboren tareas en función de dichos datos.

Estas nuevas necesidades han obligado a redefinir algunos aspectos de las SPL tradicionales para incorporar nuevas formas de desarrollo que soporten mejor el dinamismo del software. Por ello, se esta adoptando recientemente el concepto de DSPL o líneas de producto dinámicas. Algunas de las características más importantes de una DSPL [Hinchey et al., 2009] son las siguientes:

- Soporte para variabilidad dinámica, en la que la configuración y el binding ocurre en tiempo de ejecución. Los puntos de variación también cambian durante el tiempo de ejecución: se permiten añadir puntos de variación de forma dinámica.

- Son capaces de gestionar cambios inesperados.

- ❖ Capaces de tratar cambios producidos por los usuarios, tales como requerimientos funcionales o cualitativos (expresados a través de reglas).

- ❖ Conocimiento del contexto y de la situación y propiedades autónomas o autoadaptativas al entorno.

- ❖ Automatización de la toma de decisiones.

2.5. Experiencias para reconfigurar la variabilidad en tiempo de ejecución

En esta sección se describen algunas experiencias para gestionar la variabilidad de una línea de productos de forma dinámica.

2.5.1. Sistemas de transportes automatizados

En primer lugar, en [Helleboogh et all., 2009] se expone un sistema de mecanización y automatización del transporte en un almacén, en los que las unidades dedicadas al transporte se encargan de mover la mercancía de un lado a otro. Para ello, se sirve de tres tipos de subsistemas:

- Sistemas de Transporte Automatizados (ATS).
- Vehículos Guiados Automatizados (AGV).
- Asignaciones de Transporte (TA).

Un ATS posee una o más unidades AGV (Figura 6). En la misma línea, cada ATS posee un Punto de Variación llamado Asignación de Transporte, que será quien asigne las reglas a los Vehículos Guiados Automatizados.

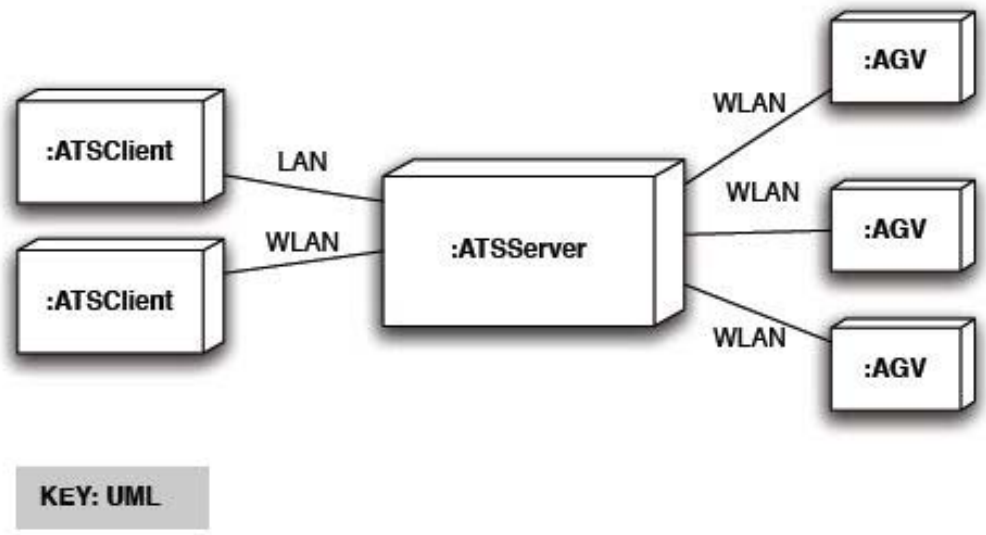


Figura 6. Configuración de un Sistema de Transporte Automatizado

El Servidor del ATS se conecta con unidades de AGV, la cuales poseen una serie de sensores que les permiten moverse de forma segura por la nave, mientras que un sistema de navegación impide que se salgan de la ruta establecida. Estos vehículos son capaces de establecer una conversación con otros vehículos similares que les permiten ponerse de acuerdo sobre qué destino tomará cada uno. El Servidor será el encargado de monitorizar el estado de cada AGV y enviarles tareas mediante la red wifi. Del mismo modo, dicho servidor lleva el control de todas las cargas que existen en el almacén para ser atendidas, así como de anunciar las nuevas tareas de transportes a los AGV y de tener guardada la ubicación de cada Vehículo en cada momento. Por otro lado, los clientes de los ATS consisten en una interfaz para los planificadores humanos que consiste en herramientas de visualización, diagnósticos, estadísticas y asignación manual de tareas a los AGVs. Estos clientes pueden ser ordenadores de sobremesa o dispositivos PDA que son utilizados para interactuar con la ATS.

Cada ATS difiere de otras en el número de AGVs disponibles, características o mercancía a tratar, la disposición de la nave, la relación con el ERP de la empresa, o el nivel de control humano. Estos ATS han sido diseñados siguiendo el esquema de una Línea de Productos de Software, donde se valora positivamente que la mayor parte de las decisiones puedan ser tomadas en tiempo de runtime de forma que la ATS no necesite ser detenida y permanezca parada el menor tiempo posible. Dependiendo de las diferentes condiciones que se puedan dar en la operativa particular de cada cliente, los AGV de un ATS concreto son configurados de un modo particular y exclusivo para dicho AGV, que puede cambiarse en tiempo de runtime.

El sistema propuesto dispone dos Puntos de Variación, tal y como se puede ver en la Figura 7, en los que el tiempo de ligadura tiene lugar en ejecución. Concretamente, el Punto de Variación Mecanismo de Asignación de Transporte tiene dos Variantes, que son Rubata y CNET. Rubata es un mecanismo de asignación de transporte basado en reglas mientras que CNET usa un protocolo de red para asignar los transportes en los AGVs. El segundo Punto de Variación es el Mecanismo de Ruta, el cual permite alternar entre dos tipos de selección de caminos, el algoritmo A* y el algoritmo A* Dinámico. De la misma forma, existe una regla de dependencia entre dos de las variantes.

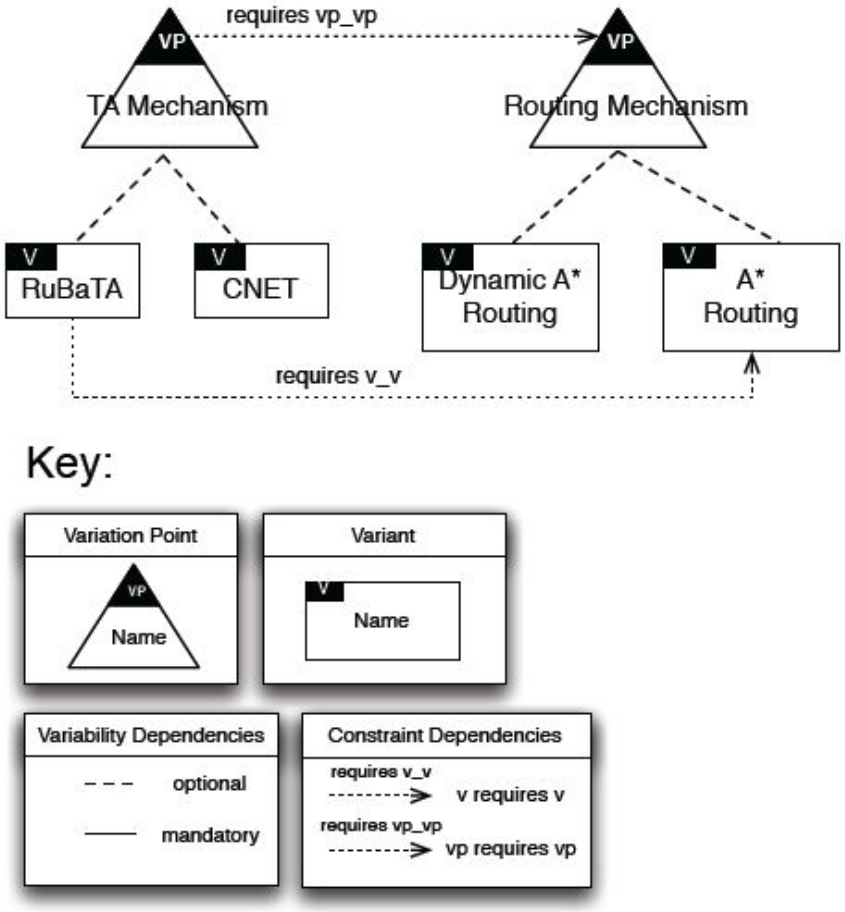


Figura 7. Modelo de variabilidad ortogonal de asignación de tareas y encaminamiento de ATS

La forma de abordar la variabilidad dinámica en este caso de estudio tiene lugar mediante un concepto de meta-variabilidad, por el cual se crea una capa adicional de abstracción que es susceptible de cambiar según el entorno. Los puntos centrales en los que se apoya este modelo, son los conceptos de Meta Punto de Variación y Meta Variante. Un Meta Punto de de Variación (MVP) es una representación de un elemento variable respecto al Modelo de Artefacto de Variabilidad.

Este modelo de meta-variabilidad queda ejemplarizado en este caso de estudio en la forma en que es posible añadir y quitar en tiempo de runtime unidades de transporte, sin tener que detener el sistema y reconfigurarlo de

acuerdo al nuevo número, y haciendo que los nuevos nodos tengan tareas asignadas o que asuman las tareas de los transportes eliminados, dependiendo del caso. Del mismo modo, las condiciones que gobiernan el sistema y a las que está sometido cada transporte pueden ser añadidas o eliminadas.

La reconfiguración de elementos y el flujo de información entre ATS y AGV se concretan mediante el uso de un artefacto llamado Gestor de Conexiones de ATS y AGV, el cual es responsable de la conexión y desconexión en tiempo de runtime de las distintas unidades. Estas unidades deben figurar previamente en un repositorio creado a tal efecto que funciona a modo de pool de transportes. Un Cargador de conexiones de AGV se encarga de hacer posible que un administrador manipule el grupo de AGVs disponibles, mientras que el Conector Distribuidor de AGVs se encarga de hacer llegar los cambios a todos los AGVs que están presentes en la ATS.

Como se puede ver, este caso de estudio afronta la variabilidad dinámica desde el punto de vista de añadir elementos que son homogéneos a los ya presentes, variando únicamente la cardinalidad de los mismos, pero dejando sin respuesta al hecho de añadir variantes de otros tipos.

2.5.2. Variabilidad dinámica en la gestión de una casa inteligente

El segundo caso de estudio que también aborda la variabilidad dinámica y que se va a explicar a continuación expone el supuesto de la aplicación de una línea de producto de software dinámica a una casa de salud mental inteligente.

La peculiaridad de este sistema es que debe ser capaz de tomar información del exterior y re-adaptar el sistema a dichas condiciones, tomando medidas tanto paliativas como preventivas, en algunas ocasiones

actuando antes de un hecho y en otras actuando tras producirse. Para ello, a través de la creación de un modelo orientado a objetivos denominado Tropos [Ali et al., 2009]. La característica más significativa de este sistema es la de dotarle de una perspectiva de una línea de producto de software donde los puntos de variación son vistos como objetivos a conseguir.

Inicialmente se define un conjunto de características principales y aquellas que inicialmente no formen parte del conjunto core, se dejan en segundo plano. Como el entorno es dinámico y va cambiando, las características del conjunto principal que funcionamiento de la casa se comprueben que no tienen utilidad pasan al conjunto secundario. De la misma forma, habrá características que pasen a tomar relevancia, y por ello deban pasar al conjunto principal.

Esto se consigue mediante la definición de contextos, que serán a su vez un punto de variación del sistema. Los tres tipos de contextos que sirven al sistema son, el contexto estimulable, el contexto requerido y el contexto de calidad, y cada uno de ellos tiene asociado un grupo distinto de puntos de variación del modelo de objetivos.

Cada objetivo, puede obtener mediante la realización de una o varias tareas, que deben estar definidas en función del objetivo a lograr. Las tareas están representadas en la figura como una serie de nodos hexagonales y con un nombre compuesto por “T” y un número. Las flechas orientadas indican el sentido de las acciones y el objetivo que se consigue. Del mismo modo, existen una serie de objetivos suaves, las cuales no pueden ser medidas cuantitativamente y sólo puede saberse si una objetivo actúa de forma positiva o negativa a su consecución. Esto se consigue a través de la evaluación de reglas.

Para resolver este problema, el modelo de este caso de estudio propone hacer una relación entre hechos, sentencias y ayudas. De este modo, el

entorno es monitorizado mediante sensores que proporcionan información al sistema de lo que acontece (hechos).

La variabilidad dinámica en estos sistemas viene representada a través de las reglas y la definición de contextos, que son las que ponen en funcionamiento unos u otros variantes en pos de conseguir dichos objetivos. Es decir, que los objetivos más usados y sus patrones de satisfacción se pueden modelar como una parte principal del sistema mientras que los objetivos menos utilizados se pueden percibir como apartados extras opcionales.

3. Planteamiento del Problema e Hipótesis

Los problemas que venían derivados de la aplicación de las líneas de producto estáticas son que la elección de los componentes que forman parte del producto final se hacen en la mayoría de los casos en tiempo de diseño, implementación y configuración. Una vez que el producto está creado, ya no es posible realizar cambios sobre el mismo y para dotar al sistema de nuevas funcionalidades o distinto comportamiento es necesario detenerlo y añadir o quitar las variantes necesarias, creando de esta manera un nuevo producto, que si bien es bastante parecido al anterior, resulta en un producto diferente.

Por ello, el planteamiento del problema que se propone en este trabajo es: *la incorporación en tiempo de ejecución de variantes y puntos de variación a un modelo de variabilidad y que puede tener aplicación en la construcción de líneas de producto dinámicas.* Los subproblemas que vamos a tratar en este trabajo son:

1. Inclusión de variantes en tiempo de ejecución con valores del mismo tipo o de diferentes tipos.
2. Modelado de supertipos que aglutinan variantes del mismo o de diferentes tipos básicos.
3. Inclusión de puntos de variación en tiempo de ejecución
4. Visualización y configuración de un árbol FODA

Como hipótesis principales a este trabajo consideramos las siguientes:

1. Los puntos de variación modificados incluirán variantes.
2. Si un punto de variación no se puede visualizar en el FODA, se representará de forma separada.
3. Los supertipos permiten visualizar partes de un árbol FODA.

4. Las restricciones permitidas serán la inclusión y exclusión de un componente respecto a otro.
5. Un punto de variación descrito con una regla de composición se define como el conjunto de todos sus componentes.
6. Un sistema generado como resultado de la generación de un producto de la línea de producto debe cumplir con las reglas establecidas para la línea de producto para poder ejecutarse.
7. No se representa la cardinalidad de los nodos del árbol por motivos de visibilidad.

4. Antecedentes del trabajo

El inicio de este modelo tiene su punto de partida en un trabajo anterior [Valdezate, 2006], cuyo cometido es proponer una herramienta Web para visualizar y gestionar la variabilidad en líneas de producto estáticas y cuya arquitectura se describe en la Figura 8.

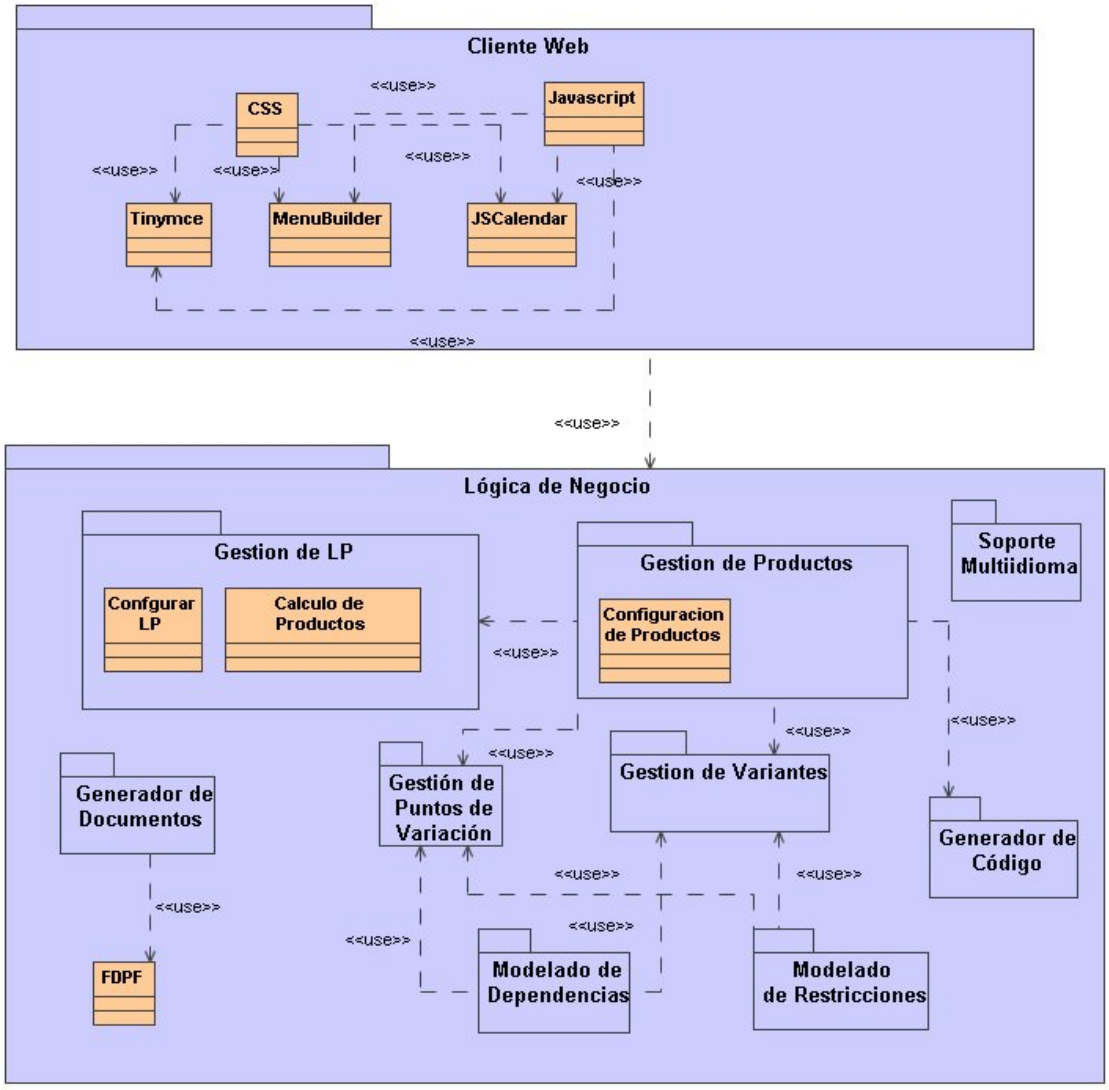


Figura 8. Arquitectura para Líneas de Producto estáticas

Esta herramienta tiene como objetivo servir a varios fines simultáneamente. Por un lado, servir como repositorio de componentes. Para ello, permite realizar la gestión completa de dicho repositorio, definiendo puntos de variación, variantes y valores, y permitiendo realizar el mantenimiento completo de dichas unidades. Por otro lado, permite la asociación de valores a variantes de manera que permanezcan ligados a la hora de ser incluidos en la línea de producto. Una vez que se dispone de un repositorio de componentes, es posible crear líneas de producto y configurarlas con los distintos puntos de variación y variantes creados previamente. Para ello es necesario modelar las características descritas en el modelo de variabilidad. Una vez que la línea de productos queda descrita, es posible añadir restricciones de dependencia 1:1 entre los variantes.

En la parte de representación, es posible visualizar tanto la línea de productos como el producto generado. La forma de representación es la de árbol esquemático.

Sin embargo, el alcance de esta herramienta sólo abarca el tiempo de diseño e implementación, no siendo posible enlazar componentes cuando el tiempo de ligadura llega más allá hasta el tiempo de ejecución. En caso de querer cambiar los componentes de un sistema en tiempo de ejecución, es necesario detener el sistema previamente y volver a generar un nuevo modelo de variabilidad que debe trasladarse a los productos de la línea de productos.

5. Solución propuesta

Como mejora del trabajo anterior, se propone un modelo que tiene como misión retrasar el tiempo de ligadura de los componentes hasta el último momento, es decir, hasta el tiempo de ejecución. La Figura 9 describe la arquitectura de la solución.

El modelo se divide en dos partes. Una describe la lógica de negocio y la otra la arquitectura del cliente web. La parte que afecta al tiempo de ligadura está íntegramente recogida en la lógica de negocio, y es la parte donde se ha realizado la investigación.

La lógica necesaria para hacer posible la reconfiguración de la variabilidad en tiempo de ejecución está recopilada en un paquete llamado *reconfiguración en runtime*, y que aparece en la parte inferior de la Figura 9. Esta entidad permite añadir puntos de variación y variantes a un producto que se encuentra en runtime, pues en tiempo de diseño la inserción se realiza en las entidades de gestión de puntos de variación y gestión de variantes, respectivamente.

Como la *reconfiguración en runtime* está ligada a la *gestión de productos*, ya que debe permitir la inserción de componentes dentro de los productos existentes. El proceso de reconfiguración precisa de la información de dependencias entre los distintos componentes del sistema, por lo que la *reconfiguración en runtime* también estará asociada al *modelo de dependencias*. El producto reconfigurado tiene que ser coherente con las restricciones definidas, por lo que esta relación también queda expresada en la figura.

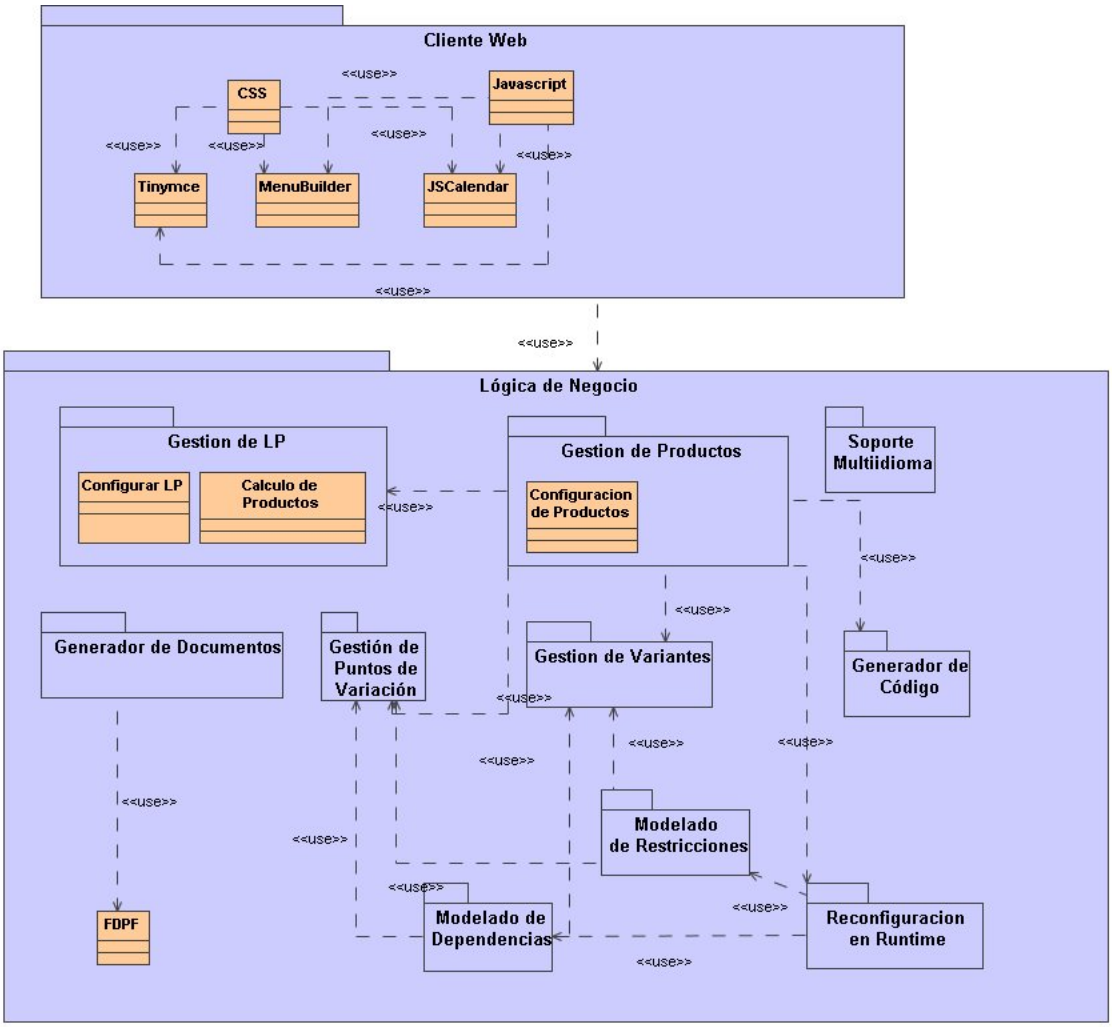


Figura 9. Arquitectura para gestionar variabilidad software en modo runtime

La solución propuesta se compone de las siguientes partes:

(a) Definición de supertipos que corresponden a ciertas características de un sistema software y que permiten seleccionar un conjunto de variantes y puntos de variación pertenecientes a un árbol FODA.

(b) Gestión de variantes en tiempo de ejecución
(c) Gestión de puntos de variación den tiempo de ejecución
(d) Visualización dinámica del árbol FODA.

5.1. Modelado de supertipos

Un supertipo es una forma de discriminar las características de un producto software de manera que los variantes y los puntos de variación que modela la variabilidad de un sistema pueden tener uno o varios supertipos. La notación de un supertipo queda representada en SP.

Cada punto de variación o variante puede tener asociado un supertipo. A su vez, un punto de variación puede necesitar ser provisto de un supertipo para lograr la funcionalidad requerida. En estos casos es necesario que alguno de los componentes que conforman el Punto de Variación le aporte el supertipo requerido.

Los supertipos nos permiten filtrar partes de un árbol FODA en el que intervienen variantes del mismo tipo o de distinto tipo. Por ejemplo, un supertipo “multimedia” para software de vehículos podría agrupar variantes relacionados que tengan tipos diferentes, como por ejemplo un variante que contenga un tipo numérico para especificar un rango de emisoras de radio y un tipo string para indicar el título de las canciones de un CD.

5.2. Variantes en tiempo de ejecución.

La solución propuesta incluye el modelado de variantes que pueden ser gestionados en tiempo de ejecución. En nuestro modelo, un variante V_i se

compone de una lista de valores V_a, un tipo básico y uno o varios supertipos que pueden ser opcionales, tal y como describe la fórmula siguiente:

$$V_l\ (Va_1,\ldots,Va_n, \text{tipo}, SP_1,..,SP_n)$$

Una variante puede añadirse a un punto de variación o directamente al producto. El proceso a seguir es el mismo en ambos casos. Para añadir una variante en tiempo de ejecución, se presta especial atención tanto al supertipo de la variante a añadir, como al supertipo de las variantes ya presentes y del punto de variación o producto al que va a conectarse, según la Figura 10.

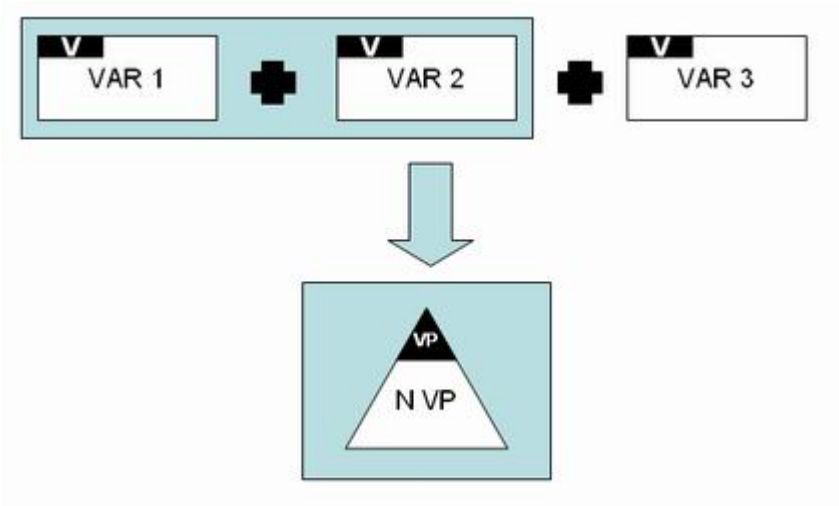


Figura 10. Inserción de una variante en tiempo de ejecución

En el caso de que el supertipo aportado por la nueva variante (VAR3) coincida con las de las variantes ya presentes (VAR1 y VAR2), y también con el supertipo admitido por el punto de variación, el resultado será que el punto de variación pasará a estar conformado por las tres variantes, tal y como se observa en la Figura 11.

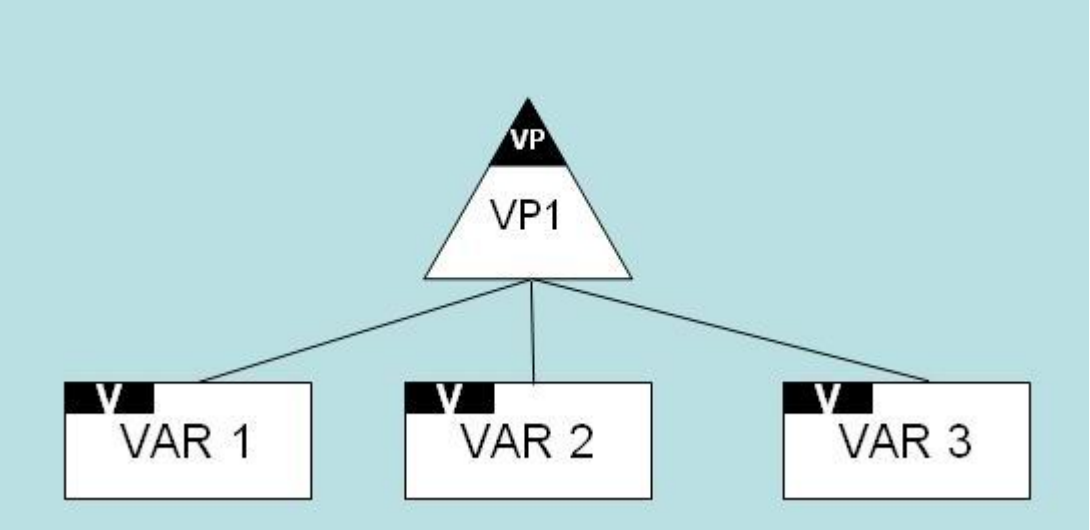


Figura 11. Inserción de variantes de supertipo homogéneo

Sin embargo, cuando el modelo de variabilidad está formado por variantes que no son homogéneas en cuanto a su supertipo, es posible agrupar aquellas que sí lo son para conformar un nuevo punto de variación y que concentra la funcionalidad aportada por el supertipo que es común a ellas. Esto queda reflejado en la Figura 12. La variante VAR3, que va a añadirse, tiene el mismo supertipo que VAR2, pero distinto del de VAR1. Por este motivo, se crea la necesidad de crear un nuevo punto de variación que agrupe variantes con igual supertipo. Por esta razón, VAR2 Y VAR3 conforman un nuevo Punto de Variación VP2 que representa el supertipo común a ambas.

VP1 -> VP2, VAR1

VP2 -> VAR2, VAR3

Algunos puntos de variación precisa de un supertipo determinado y otros dejarán abierto el camino a integrar variantes de cualquier supertipo. En este sentido no hay restricciones para añadir una variante. La metodología a seguir para insertar la variante será determinada por el punto de variación existente en función de si es necesario aportar un supertipo determinado, y

en caso de que la variante a añadir no pueda aportarlo, no podrá realizarse la inserción.

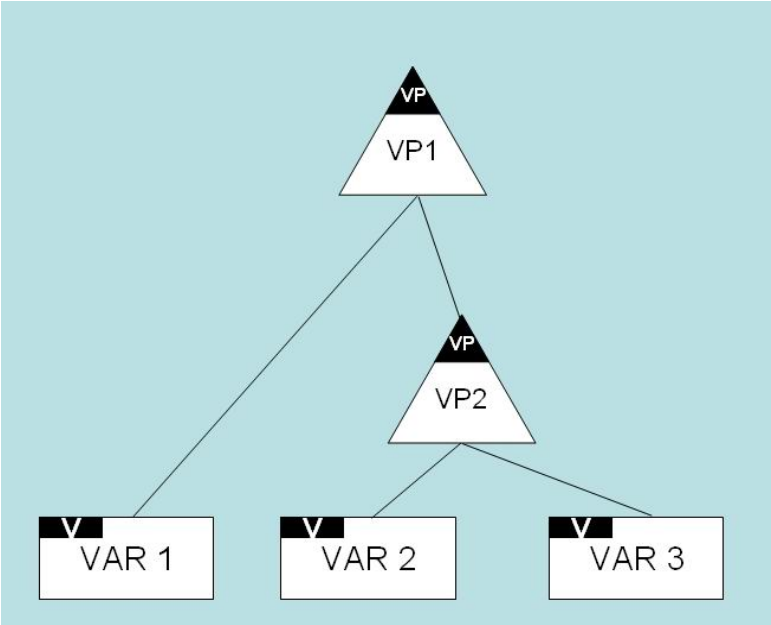


Figura 12. Inserción de variantes de tipo heterogéneo

Esta operación se puede ver más claramente mediante un caso práctico. Suponiendo una línea de productos en cuyo repositorio de características se encuentran las operaciones de suma, resta, multiplicación y división. A partir de dicha línea de productos se genera una configuración concreta para un producto que es capaz de sumar y restar, pero que es susceptible de aceptar otras operaciones no contempladas inicialmente, como puede ser dividir, multiplicar o realizar la raíz cuadrada. En el caso descrito, se puede ver como a las dos operaciones iniciales contempladas por el sistema se le añade una nueva (división). De esta manera el sistema tiene ahora tres variantes relacionadas y que entran dentro del ámbito de una línea de productos concreta. La Figura 13 es un ejemplo de lo expuesto aquí.

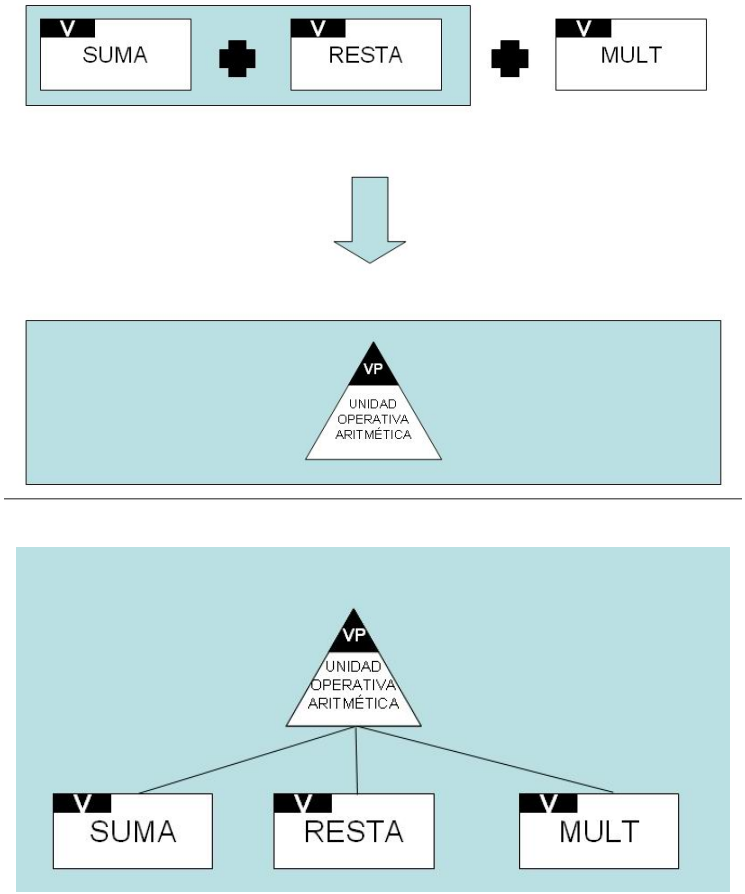


Figura 13. Caso práctico de variantes homogéneas: Unidad operativa aritmética

Otro ejemplo reflejado en la Figura 14, consiste en añadir una variante cuando los supertipos de los variantes presentes son heterogéneos. En este ejemplo, existe un punto de variación con varias variantes, siendo una variante un lector de discos compactos, que provee el supertipo *multimedia*. El resto de variantes tienen proveen otros supertipos. Al punto de variación se le añade un *lector USB* que provee también del supertipo *multimedia*. Como resultado de aunar ambos componentes se genera un nuevo punto de variación que representa el supertipo de estos dos componentes.

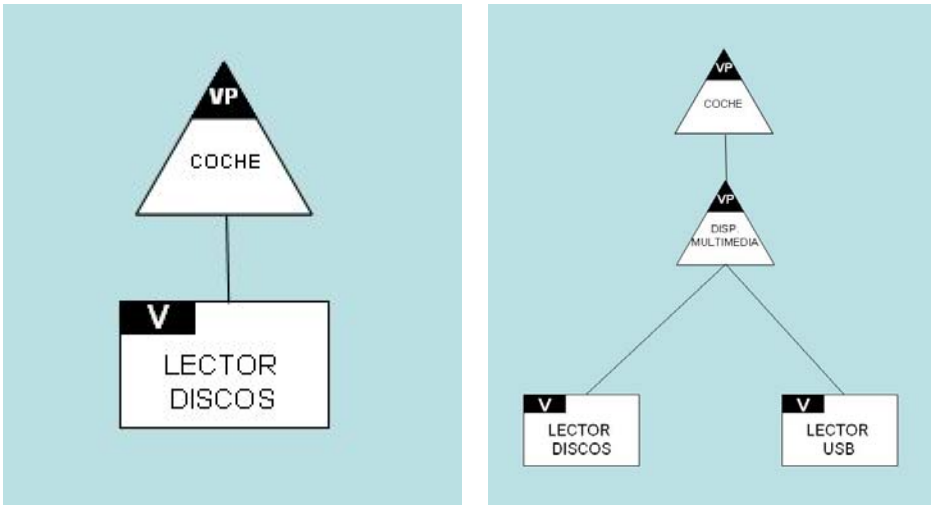


Figura 14. Inserción de variante con supertipos heterogéneos.

5.3. Puntos de variación en tiempo de ejecución

Asimismo, nuestra solución contempla la gestión de puntos de variación en tiempo de ejecución. En nuestro modelo, un punto de variación VPi se compone de una lista de variantes V y de Puntos de Variación VP. Los puntos de variación, al aglutinar a variantes y puntos de variación, pueden pertenecer a diversos supertipos, tal y como muestra la fórmula siguiente.

$$VP_I\ (VP_1,\ldots,VP_n,\ V_1,\ldots,V_n,,\ SP_1,..,SP_n)$$

De manera análoga a como sucedía en el caso de las variantes, a la hora de añadir un punto de variación habrá que prestar atención tanto al supertipo aportado por el punto de variación a insertar como el del punto de variación donde se inserta. También habrá que tener en cuenta los otros supertipos de los puntos de variación y variantes que ya estaban presentes.

La relación del nodo tiene especial relevancia, pues dependiendo de si el nodo es AND, OR o AND el mecanismo de inserción será ligeramente distinto.

En el caso de un nodo XOR, el punto de variación a insertar debe quedar activado. Por ello, en la configuración del sistema, debe figurar que el punto de variación que figura en el nodo padre figure el nuevo punto de variación como seleccionado. Para ello, es preciso deseleccionar el nodo que ya lo estuviera para esta relación XOR.

En el caso de un nodo OR o AND, el mecanismo es más simple. En el caso del nodo ON pueden coexistir varios nodos seleccionados al mismo tiempo. En el caso del nodo AND, todos deben estar seleccionados. En estos dos casos, la existencia de un nuevo punto de variación no altera el estado de los otros nodos hijos ya existentes en esta rama del árbol. Por este motivo, el nuevo punto de variación se puede insertar sin tener que realizar modificaciones sobre los nodos hermanos.

5.4. Modelado del árbol FODA

A la hora de realizar un modelo de características es muy importante tener un método de representación adecuado. Debido a que frecuentemente el número de puntos de variación, variantes y valores que conforman una línea de producto es muy grande, resulta muy complicado representar el total de la información perteneciente a cada componente. Por este motivo, el uso de supertipos resulta extremadamente útil.

La representación de nuestro árbol FODA muestra los nodos y relaciones entre nodos de tipo AND, OR, XOR. El uso de colores para diferenciar los puntos de variación, variantes y valores puede verse en la Figura 15.

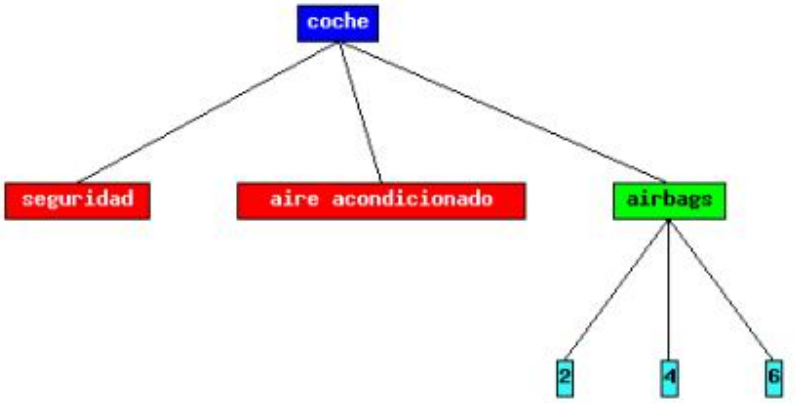


Figura 15. Ejemplo de variabilidad de una línea de productos de software de vehículos.

El uso de colores para mostrar las distintas opciones de configuración de un árbol aporta la posibilidad de conocer aquellos nodos donde existe variabilidad. El producto a generar se muestra en azul oscuro, los puntos de variación en color rojo, las variantes, en color verde y los valores que puede tomar la variante, en color azul claro. La variabilidad queda representada en aquellos nodos que presentan color verde.

6. Implementación

6.1. Arquitectura runtime

Nuestro sistema alterna los modos de diseño con el de ejecución, siendo necesario utilizar estructuras adicionales para realizar los ajustes sobre el producto cuando se añaden variantes o puntos de variación en tiempo de ejecución, reconfigurando así el modelo de variabilidad.

La Figura 16 describe la fase de diseño en términos de clases participantes. Una línea de producto está contiene una serie de entidades de puntos de variación, variantes y valores. La entidad de nodos permite representar una línea de productos de forma jerárquica en términos de puntos de variación, variantes y valores. Cada línea de producto tendrá asociado un conjunto de nodos, tantos como elementos contenga la línea de producto Cada línea de producto puede generar un conjunto de productos. Una línea de productos tiene asociado un conjunto de reglas. Un producto tiene asociado una única línea de producto, aunque una línea de producto puede generar varios productos. Un producto y su configuración quedan definidos por la línea de producto a la que pertenecen.

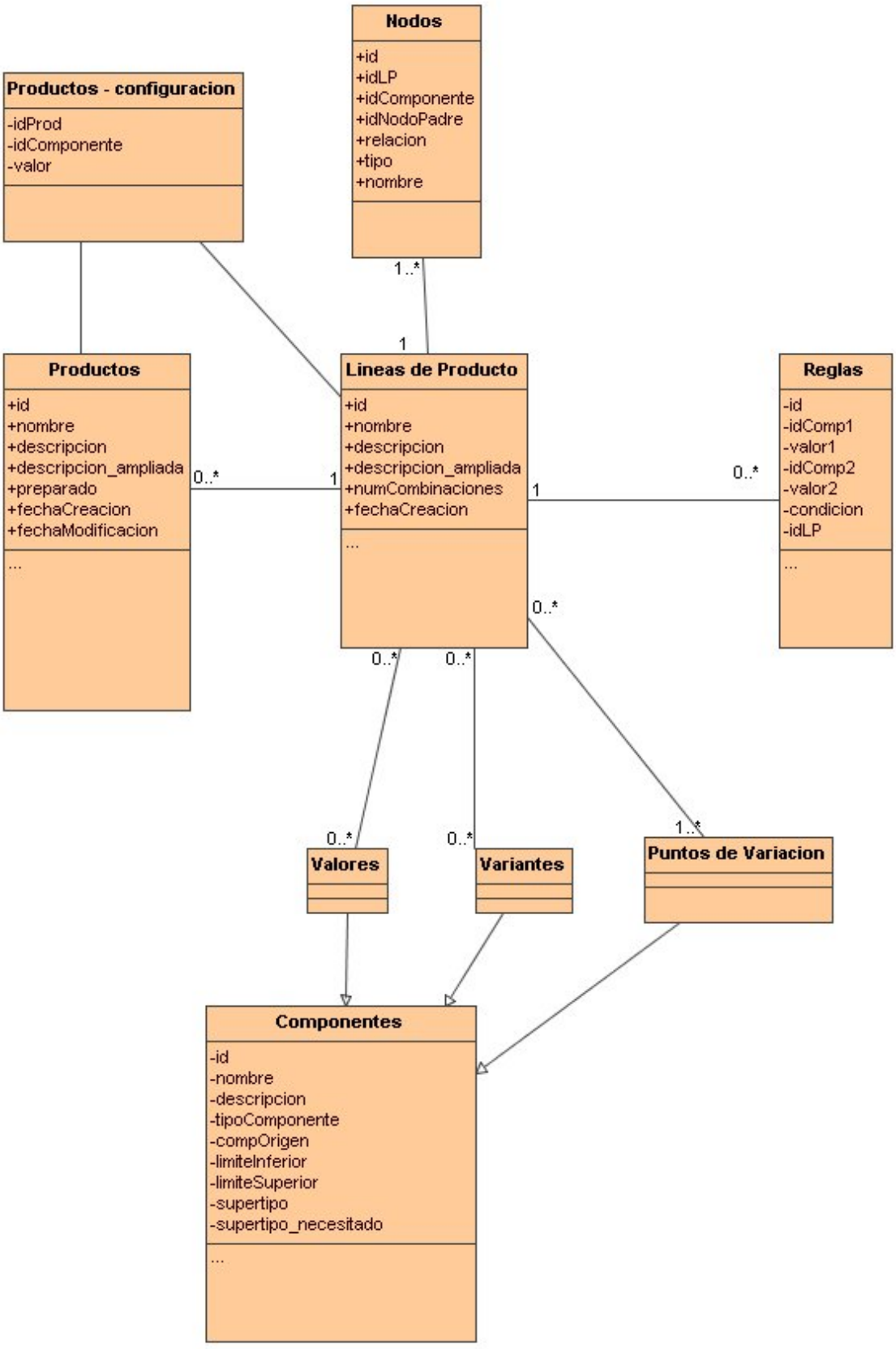


Figura 16. Entidades de datos en tiempo de diseño

El modo de ejecución incluye un mecanismo denominado *Mecanismo de Alteración del Producto en Runtime* (**MAPR**). El MAPR se ocupa de comprobar la viabilidad de un cambio en un sistema que está en ejecución. Si el sistema está detenido, el MAPR no interviene a la hora de realizar las modificaciones. El funcionamiento del MAPR se basa en gran parte en la aproximación mediante supertipos. Partiendo de un punto de variación o variante no existente en el sistema, el MAPR comprueba la relación del supertipo del componente y las de los otros componentes participantes (nodo padre y nodos hermanos), decidiendo si incluye el componente como un hermano más o si crea un punto de variación nuevo que aúne las variantes con mismo supertipo dentro del subárbol.

Para lograr este objetivo, el MAPR elabora un entorno paralelo en el que es posible realizar los cambios de forma segura y sin afectar al sistema principal. Este entorno paralelo necesita de los mismos componentes que el sistema principal, por lo que esos datos son replicados. De esta forma el MAPR puede tener exclusividad sobre los mismos sin que ello afecte al funcionamiento del sistema. Una vez desarrolladas las inserciones, el MAPR incorpora los nuevos datos al entorno real.

El MAPR necesita para funcionar las entidades *productos_configuracion_pre* y *nodos_pre*, que aparecen en la Figura 17.

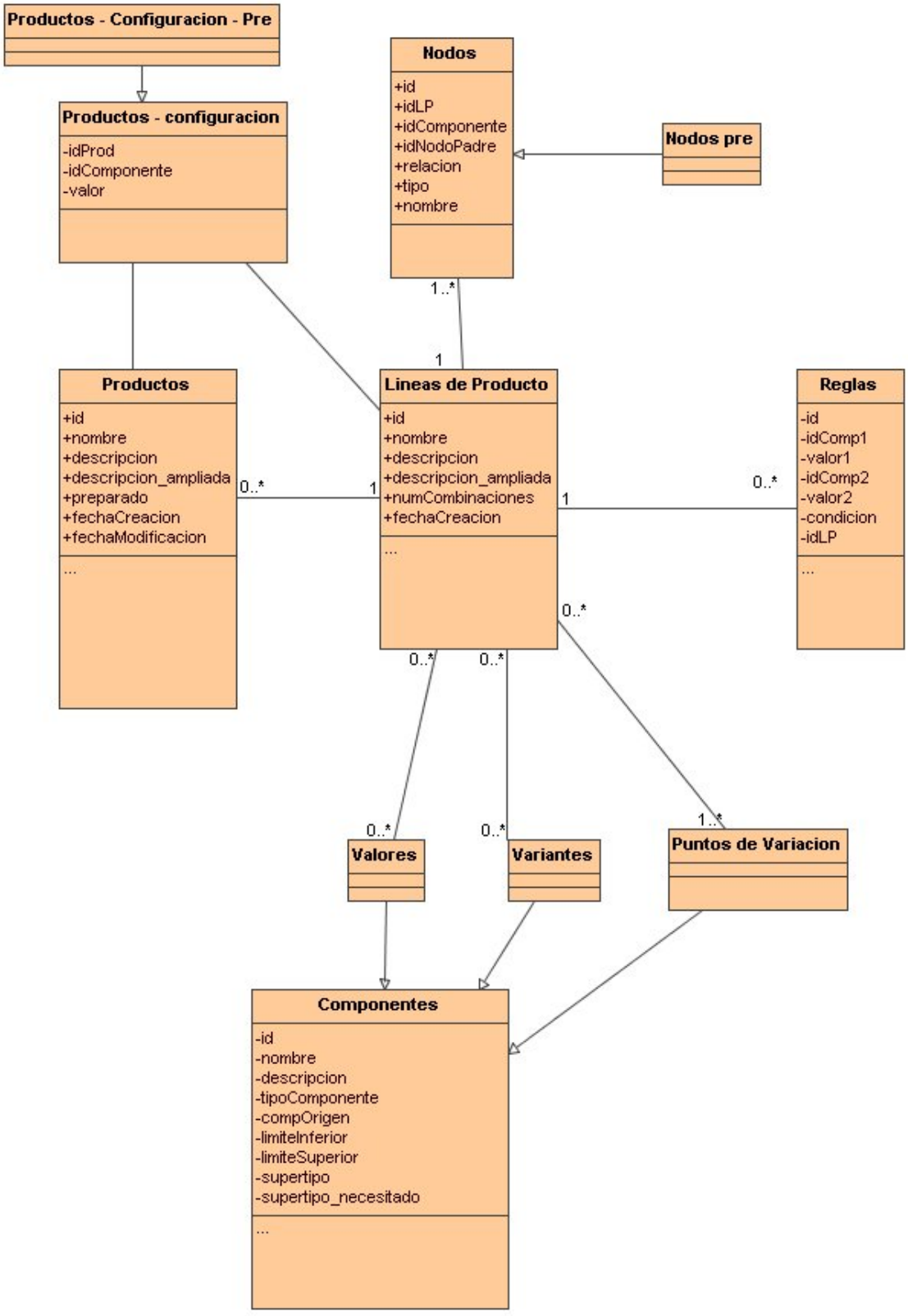


Figura 17. Entidades de datos en ejecución

6.2. Inserción de variantes

Para la inserción de variantes ha sido preciso desarrollar un algoritmo que contemple todas las opciones posibles en relación al supertipo de la variante a añadir, al supertipo del padre y de los hermanos que tendrá en el nodo. El proceso para insertar un variante en modo runtime es el siguiente.

1. Se trata de un nueva variante.
2. Si es nueva se inserta en el repositorio de componentes.
3. Se comprueba el supertipo de la variante.
4. Se comprueba el supertipo del nodo padre.
5. Se comprueba el supertipo de los nodos que serán hermanos de la variante.
6. Si el supertipo del padre necesita de un supertipo concreto, y la variante a añadir tiene este supertipo, se añade como hijo del nodo padre.
7. Si el supertipo del padre no necesita de un supertipo concreto, entonces se comprueban los supertipos de los nodos hermanos.
8. Si uno de los hermanos tiene el mismo supertipo que la variante a añadir, se crea un nuevo punto de variación que agrupe la funcionalidad de ambos y se hace que todos los nodos hermanos que compartían ese supertipo cuelguen de dicho nodo.

Una vez hecha la inserción se comprueba que el producto sigue cumpliendo las reglas definidas en la línea de producto a la que pertenece y si es así, se trasladan los datos de las tablas *productos_configuracion_per* y *nodos_pre* a *productos_configuracion_per* y *nodos_pre* respectivamente.

El código de cada parte significativa es el siguiente:

- Preparación de las tablas.

```
//preparamos la tabla tfm_config_pre y tfm_lp_nodos_pre
```

```
	$query = "delete from tfm_productos_config_pre;";
	$result = mysql_query($query, $conexion);
	$query = "insert into tfm_productos_config_pre (select idProd,
idVP,valor,texto from tfm_productos_config where
idProd=$idProducto);";
	$result = mysql_query($query, $conexion);
	$query = "delete from tfm_lp_nodos_pre;";
	$result = mysql_query($query, $conexion);
	$query = "insert into tfm_lp_nodos_pre (select
id,idLP,idVP,idNP,relacion,tipo,nombre from tfm_lp_nodos where
idLP=$idLP);";
	$result = mysql_query($query, $conexion);
```

- Comprobación de la existencia previa de la variante.

```
	// lo primero, dejamos el componente insertado en el
repositorio
	$nombre=$_POST['nombre'];
	$supertipo=$_POST['supertipo'];
	//verificamos que no existe
	$query = "select id from tfm_componentes where
nombre='$nombre';";
	saca($query);
	$result = mysql_query($query, $conexion);
	if ( mysql_num_rows($result)>0) {
		$hayError=true;;
		$mensajes.="Ya existe la variante.<br/>";
	}
```

- Inserción de la variante en el repositorio.

```
		if ($hayError==false) {
			//añadimos la variante a la lista de componentes
			$query = "INSERT into tfm_componentes
(nombre,supertipo,varvp) VALUES ('$nombre','$supertipo','var');";
```

```
                saca($query);
                $result = mysql_query($query, $conexion);

                $rollback_cmd="delete from tfm_componentes where
nombre='$nombre';\n".$rollback_cmd;
```

- Obtención de supertipos.

```
//sacamos el supertipo del padre
$supertipo_necesitado_padre=dameSupertipoNecesitado($vppadre);
if (strlen($supertipo_necesitado_padre)>0) {
      $haysupertipoNecesitadoPadre=true;
}else{
      $haysupertipoNecesitadoPadre=false;
}

//sacamos el número de hermanos del nodo
$hermanos=hermanosEnLP($idLP,$vppadre,'tfm_lp_nodos_pre');
saca ("llamada hermanos=hermanosEnLP($idLP,$vppadre)");
```

- Inserción de la variante con un supertipo determinado

```
if ($haysupertipoNecesitadoPadre==true)  {
      if ($supertipo!=$supertipo_necesitado_padre) {
            //si está definido
            //verificamos que el supertipo que provee el nuevo
variante es compatible con el del punto de anclaje
            $mensajes.="El tipo del variante a añadir no
concuerda con el punto de variación del punto de anclaje.<br/>";
            $hayError=true;
      }
      if ($hayError==false){
            //añadimos la variante a la LP
```

```
                $cons="insert into tfm_lp_nodos_pre
(idLP,idVP,idNP,relacion,tipo,nombre) values
($idLP,$idVP,$vppadre,'none', 'mandatory','$nombre')";
                saca($cons);
                $result = mysql_query($cons, $conexion);
                $rollback_cmd="delete from tfm_lp_nodos where
nombre='$nombre';\n".$rollback_cmd;

                //añadimos la variante al producto
                $relpadre=relacionNodo($idLP,$vppadre);
                saca($relpadre);
          }

          if
((($relpadre=="or")||($relpadre=="and"))&&($hayError==false)) {
                //si es un or, se activa directamente
                $cons1="INSERT INTO tfm_productos_config_pre
(idprod,idVP, valor) VALUES ($idProducto,$vppadre,'$idVP')";
                saca($cons1);
                $result = mysql_query($cons1, $conexion);
                $rollback_cmd="delete from tfm_productos_config
where idprod=$idProducto and idP=$vppadre and valor='$idVPNueva';\n"
.$rollback_cmd;

          }elseif($relpadre=="xor"){
                //se activa y se quita al otro
                $cons1="UPDATE tfm_productos_config_pre set
valor='$idVP' where   idProd=$idProducto and idVP=$vppadre ";
                $result = mysql_query($cons1, $conexion);
                saca($cons1);
          }

          if (comprobarReglas_pre($idProducto,$idLP)==false) {
                $mensajes.="El nuevo producto no cumpliría las
reglas, por lo que no se autorizan los cambios.";
                $hayError=true;
          }

          if ($hayError==false) {
```

```
            //actualizamos los datos en las tablas auténticas
                  actualizaTablas($idProducto,$idLP);

                  $mensajes.="<br/><br/>Se ha conectado correctamente
la variante $nombre.";
        }else {
        }
```

- Inserción de la variante sin que el punto de variación precise un supertipo concreto.

```
      //miramos a ver si los hermanos son del mismo tipo
      saca("hermanos=$hermanos");
      $hermanosDelMismoTipo=false;
      $hermanosDeDistintoTipo=false;

      if ($hermanos>0) {
            //$vppadre=padreNodo($idLP,$idVP);
            $cons="select idVP from tfm_lp_nodos_pre where
idNP=$vppadre and idLP=$idLP and idVP not in (select idVP from
tfm_lp_nodos where idVP=$idVP)";
            saca($cons);
            $result = mysql_query($cons, $conexion);
            $encontrado=false;
  while(list($miHermano) = mysql_fetch_array($result)) {
                  $supertipohermano=dameSupertipo($miHermano);
                  // miramos a ver si el supertipo de los hermanos
coincide con el del nuevo componente.
                  if ($supertipo!=$supertipohermano) {
                        $hermanosDelMismoTipo=false;
                  }else{
                        $hermanosDeDistintoTipo=true;
                        $encontrado=true;
                  }
            }
      }else{
```

```
            $hermanosDelMismoTipo=true;
            $hermanosDeDistintoTipo=false;
      }
```

- Creación del punto de variación que alberga variantes con mismo supertipo.

```
      if (($hermanosDelMismoTipo==true)
&&($hermanosDeDistintoTipo==true))
            if ($hayError==false)    {
                  $nombreNuevoVP=$supertipo;
                  $supertiponuevo="a";
                  $supertiponecesitadonuevo=$supertipo;
                  //añadimos la variante a la lista de componentes
                  $query = "INSERT into tfm_componentes
(nombre,supertipo,varvp,supertipo_necesitado) VALUES
('$nombreNuevoVP','$supertipo','vp', '$supertiponecesitadonuevo');";
                  $result = mysql_query($query, $conexion);

                  $rollback_cmd="delete from tfm_componentes where
nombre='$nombre';\n".$rollback_cmd;

                  //sacamos el id del nuevo VP
                  $query = "select max(id) from tfm_componentes";
                  saca($query);
                  $result = mysql_query($query, $conexion);
                  list($idVPNuevo)= mysql_fetch_array($result);
            }

            // cambiamos los componentes que ya existian para que
cuelguen de ese nodo nuevo (update tfm_lp_nodos)
            // esos componentes se guardan en un array
            $varid = array();
            $i=0;
            $cons="select idVP from tfm_lp_nodos_pre where idLP=$idLP
and idNP=$vppadre ";
            saca($cons);
```

```
            $result = mysql_query($cons, $conexion);
      while(list($id) = mysql_fetch_array($result)) {
                  $cons2="select supertipo from tfm_componentes where
id=$id ";
                  $result2 = mysql_query($cons2, $conexion);
                  list($elsupertipo) = mysql_fetch_array($result2);
                  if ($elsupertipo==$supertipo) {
                        $i++;
                        $varid[$i]=$id;
                  }
            }
            $limite=count($varid);
            for ($i=1;$i<=$limite;$i++) {
                  $cons="update tfm_lp_nodos_pre set idNP=$idVPNuevo
where idLP=$idLP and idNP=$vppadre and idVP=".$varid[$i];
                  $result = mysql_query($cons, $conexion);
            }
```

- Inserción de la variante en el punto de variación creado.

```
            // metemos el nodo en tfm_lp_nodos_pre y le ponemos de
padre el antiguo padre del que iba a colgar la variante.
            // la relacion será "AND" y el tipo "Mandatory".
            if ($hayError==false)               {
                  //añadimos el nuevo nodo a la LP
                  $cons="insert into tfm_lp_nodos_pre
(idLP,idVP,idNP,relacion,tipo,nombre) values
($idLP,$idVPNuevo,$vppadre,'xor', 'mandatory', '$nombreNuevoVP')";
                  saca($cons);
                  $result = mysql_query($cons, $conexion);
                  $rollback_cmd="delete from tfm_lp_nodos_pre where
nombre='$nombre';\n".$rollback_cmd;
            }

            // para los mismos componentes de antes se cambia
tfm_producto_config_pre
            // primero se hace un update de los nodos anteriores
(valor) para que el idVP sea el nuevo componente
```

```
            // luego se hace un insert con nuevo componente cuyo
padre sea el padre antiguo

            $limite=count($varid);
            for ($i=1;$i<=$limite;$i++) {
                  $cons="update tfm_producto_config_pre set
idVP=$idVPNuevo where valor='".$varid[$i]."' ";
                  $result = mysql_query($cons, $conexion);
            }

            $cons="INSERT INTO tfm_productos_config_pre (idProd,idVP,
valor) VALUES ($idProducto,$vppadre,'$idVPNuevo')";
            $result = mysql_query($cons, $conexion);
```

- Comprobación de restricciones.

```
            if (comprobarReglas_pre($idProducto,$idLP)==false) {
                  $mensajes.="El nuevo producto no cumpliría las
reglas, por lo que no se autorizan los cambios.";
                  $hayError=true;
            }
```

- Incorporación a las tablas principales.

```
      // si todo ha ido bien, se incorporan los cambios a las tablas
principales

      //actualizamos los datos en las tablas auténticas
      $query = "delete from tfm_productos_config where
idProd=$idProducto;";
      $result = mysql_query($query, $conexion);
      $query = "insert into tfm_productos_config (select idProd,
idVP,valor,texto from tfm_productos_config_pre);";
      $result = mysql_query($query, $conexion);
      $query = "delete from tfm_lp_nodos where idLP=$idLP;";
```

```
      $result = mysql_query($query, $conexion);
      $query = "insert into tfm_lp_nodos (select
id,idLP,idVP,idNP,relacion,tipo,nombre from tfm_lp_nodos_pre);";
      $result = mysql_query($query, $conexion);
      $mensajes.="<br/><br/>Se ha conectado correctamente la variante
$nombre.";
```

6.3. Inserción de puntos de variación

De manera similar a lo realizado en el caso de los variantes. En este caso la clave aquí será el supertipo y en función del mismo y del tipo de nodo donde se vaya a insertar se realiza la inserción de distinta forma:

(a) En el caso de nodos XOR para que el nodo seleccionado sea el punto de variación a insertar.

En un nodo XOR sólo uno de los nodos hijos puede estar seleccionado. Por este motivo, al insertar un nuevo punto de variación, el punto de variación que permanecía activado previamente pasa a estar desactivado. El nuevo punto de variación será el nodo que esté activado a partir de este momento.

(b) En el caso de los nodos OR el punto de variación a insertar estará seleccionado para que aparezca como parte del producto.

Los nodos OR permiten escoger varias opciones. Por este motivo, no es necesario desactivar los nodos existentes. El punto de variación se añade y se coloca como activo, sin que los hermanos del punto de variación deban sufrir ningún cambio.

(c) En el caso de un nodo AND, se añade al resto de hermanos como cualquiera de los otros.

En una relación AND todos los nodos tienen que estar seleccionados a la vez. Por esta razón, el punto de variación se inserta activado, de la misma forma en que se hacía con el nodo OR. El punto de variación insertado queda activado, y la configuración de los nodos hermanos existentes no necesita ser alterada.

El código de la inserción de cada tipo de nodo es el siguiente:

- Inserción en un nodo XOR

```
//se activa y se quita al otro
$cons1="UPDATE tfm_productos_config_pre set valor='$idVP' where idProd=$idProducto and idVP=$vppadre ";
$result = mysql_query($cons1, $conexion);
saca($cons1);
```

- Inserción en un nodo OR

```
$cons1="INSERT INTO tfm_productos_config_pre (idprod,idVP, valor) VALUES ($idProducto,$vppadre,'$idVP')";
$result = mysql_query($cons1, $conexion);
$rollback_cmd="delete from tfm_productos_config_pre where idprod=$idProducto and idP=$vppadre and valor='$idVPNueva';\n" .$rollback_cmd;
```

- Inserción en un nodo AND

```
$cons1="INSERT INTO tfm_productos_config_pre (idprod,idVP, valor) VALUES ($idProducto,$vppadre,'$idVP')";
$result = mysql_query($cons1, $conexion);
```

```
$rollback_cmd="delete from tfm_productos_config_pre where idprod=$idProducto and idP=$vppadre and valor='$idVPNueva';\n" .$rollback_cmd;
```

6.4. Visualización del árbol FODA

Para el visionado y la representación gráfica se ha creado un nuevo sistema para visualizar el árbol FODA consistente en permite realizar un gráfico guiado por código, de manera que es es posible generar el árbol nodo a nodo, explorando el árbol en profundidad y generando una estructura de datos multidimensional que contiene todas las variables necesarias de cada nodo para dibujar posteriormente el árbol. Los atributos utilizados por el algoritmo para describir el árbol FODA son:

- Nombre del componente
- Identificador del componente
- Identificador del nodo en la línea de producto
- Puntero al nodo superior
- Puntero al hermano más a la izquierda
- Puntero al hermano más a la derecha
- Número de hijos
- Nivel dentro del árbol

Estos atributos no son visibles por el usuario, pero permiten la correcta visualización del árbol. Los atributos que sí son representados externamente son:

- Tipo del nodo (Punto de variación, variante, valor)
- Relación nodo (AND, OR, XOR)
- Obligatorio u opcional

Para la representación gráfica se ha diseñado un motor de visionado específico para representar árboles FODA. El motor de visionado es una capa colocada encima de la biblioteca phpDiagram, que permite la representación de diversos tipos de gráficos en PHP apoyado por la implementación de la librería GD. A esta biblioteca de diagramas se le han añadido diversos métodos y funciones para que el resultado sea representar un árbol FODA.

Las funciones utilizadas para la representación son:

- buscarPadreCajas
- obtenerNivelArbol
- obtenerNodosNivelArbol
- obtenerPosArbol
- hermanosEnLP
- pintaDiagramaT
- pintaDiagramaT2
- pintaFlecha

De esta manera, una vez definidos los nodos y su relación entre ellos, este motor tiene la información necesaria para dibujar las líneas que unen los distintos nodos y distribuir dichos nodos en pantalla.

Para poder representar un árbol FODA, toma como entrada una estructura de nodos con una serie de atributos concretos permite realizar un guion cuya interpretación lleva a conseguir la representación gráfica de una línea de producto.

De esta manera, es posible tanto representar todas las opciones posibles de configuración de un producto acorte a una línea de productos. Mediante la representación gráfica se configuran los productos, mediante la selección de

componentes, nodos y otras opciones, utilizando únicamente el ratón para ello.

Uno de los problemas que tiene FODA es que los árboles no ofrecen suficiente información sobre los componentes que albergan los nodos, siendo necesario recurrir a otro tipo de representaciones para obtener mayor información. Mediante el enfoque dado en esta herramienta, es posible disponer de la información de cada nodo y componente presente, siendo sólo visible a petición cuando el usuario así lo requiere. Para ello, al posicionar el ratón encima de un nodo se muestra en una ventana la información del componente en cuestión, informando de si se trata Punto de Variación, una Variante o un Valor, si el componente es obligatorio (mandatory) u opcional (optional), y del supertipo que lleva asignado.

En la figura Figura 18 se representa un nodo de tipo XOR que alberga una variante obligatoria, la cual tiene tres valores.

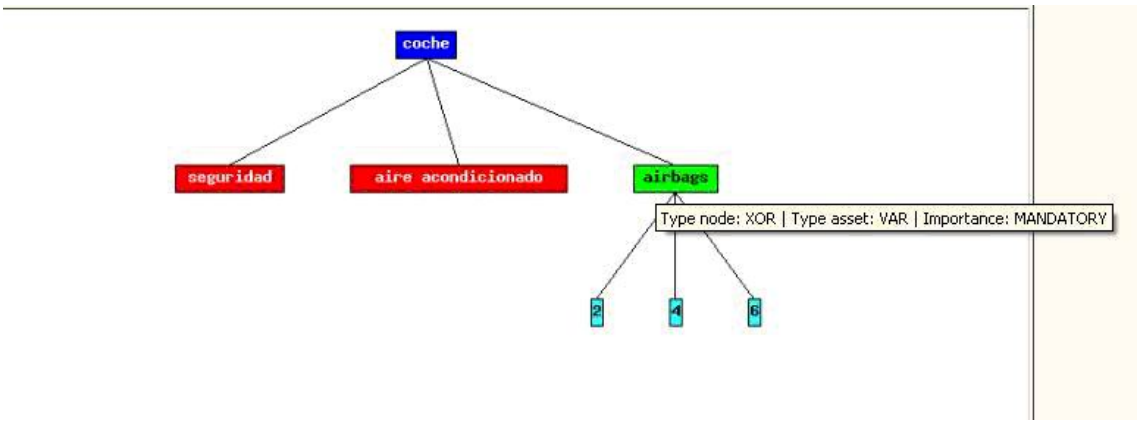


Figura 18. Información de un nodo

7. Caso de estudio

Para mostrar el funcionamiento de la herramienta, se va a elaborar un caso completo que ilustre todo el ciclo, desde la creación del repositorio hasta la inserción de elementos en tiempo de ejecución. Para que el caso sea fácil de seguir, usaremos el ejemplo de la línea de productos de coches, ya que esta línea de productos tiene unos puntos de variación bien definidos, y el árbol generado tiene todos los elementos necesarios para mostrar el potencial de la herramienta. En primer lugar, es necesario crear los diferentes componentes que van a utilizarse para construir la línea de productos. Para definir una línea de productos que produzca coches, vamos a insertar algunos puntos de variación, y entre otros estarán:

- Potencia
- Aire acondicionado
- Seguridad

Antes de dar de cada punto de variación, puede comprobarse si ya existe alguno con ese nombre pulsando en el botón Comprobar (Figura 19). Se rellenan el campo del nombre y aquellos campos que puedan ser necesarios para construir el árbol, como el supertipo.

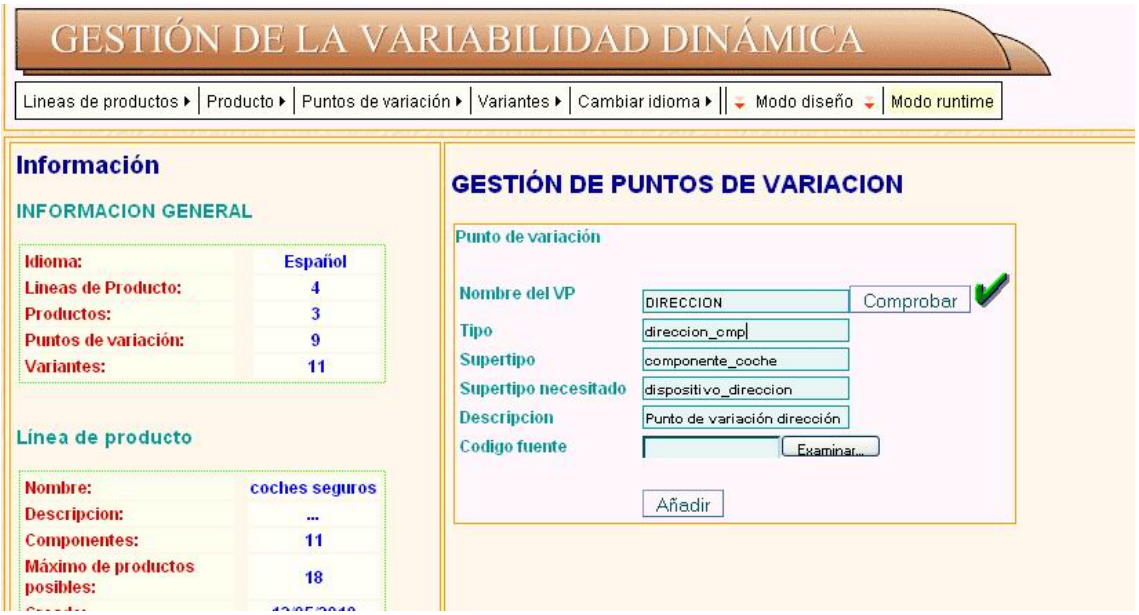


Figura 19. Inserción de puntos de variación.

A continuación se insertan las variantes. Cada variante tiene unos atributos (tipo, descripción,…) que hay que definir. Si la variante tiene un código fuente, se asocia como un archivo adjunto ligado a la variante. Las variantes que se van a insertar son:

- Airbags. Valores admisibles: 2, 4 y 6
- Potencia baja. Valores admisibles: 50, 55 y 60
- Potencia media. Valores admisibles:70, 80 y 90
- Potencia alta. Valores admisibles:105, 115 y 120

Desde los mismas pantallas de inserción y edición de variantes es posible también gestionar los posibles valores que tendrán estas variantes. Al igual que en el caso de la comprobación de la existencia del variante, la inserción de valores se realiza usando funciones de AJAX para evitar tener que cargar una y otra vez esta página (Figura 20).

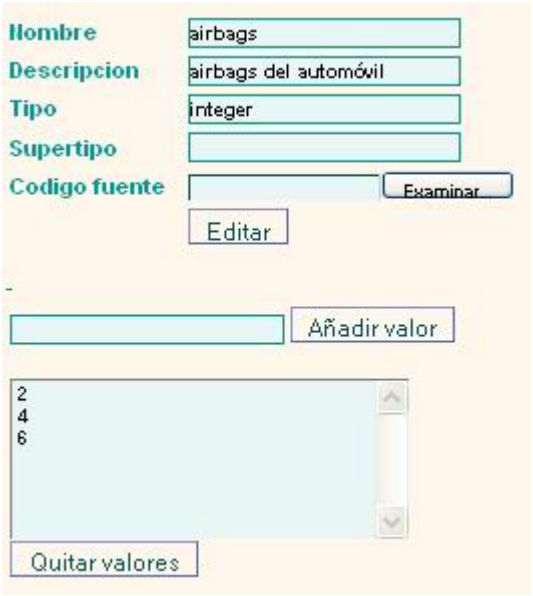


Figura 20. Edición de valores del variante *airbags*

Una vez introducidos algunos componentes ya es posible asignarlos a la línea de producto. Al dar de alta una línea de producto, hay que especificar un punto de montaje de la misma, es decir, cual es el tipo de productos que van a construirse y que conforman el nodo del árbol FODA. Si este punto de montaje aún no existe, hay que definirlo.

A continuación se definen los variantes y puntos de variación del modelo de variabilidad. Para ello se asignan los componentes a los diferentes nodos (Figura 21), indicando el tipo de nodo (AND, OR, XOR, NONE) y si nodo es opcional. Si el nodo no va a tener hijos (es un nodo hoja) se especifica que el tipo de nodo es NONE.

Por cada nodo de tipo AND, OR o XOR siempre se definirá una rama que permita seguir añadiendo nodos en cualquiera de las ramas del árbol. Para eliminar los nodos insertados, hay un botón en cada nodo ya insertado que permite hacerlo. Del mismo modo, es posible editar el tipo de un nodo, así como convertirlo de obligatorio a opcional y viceversa.

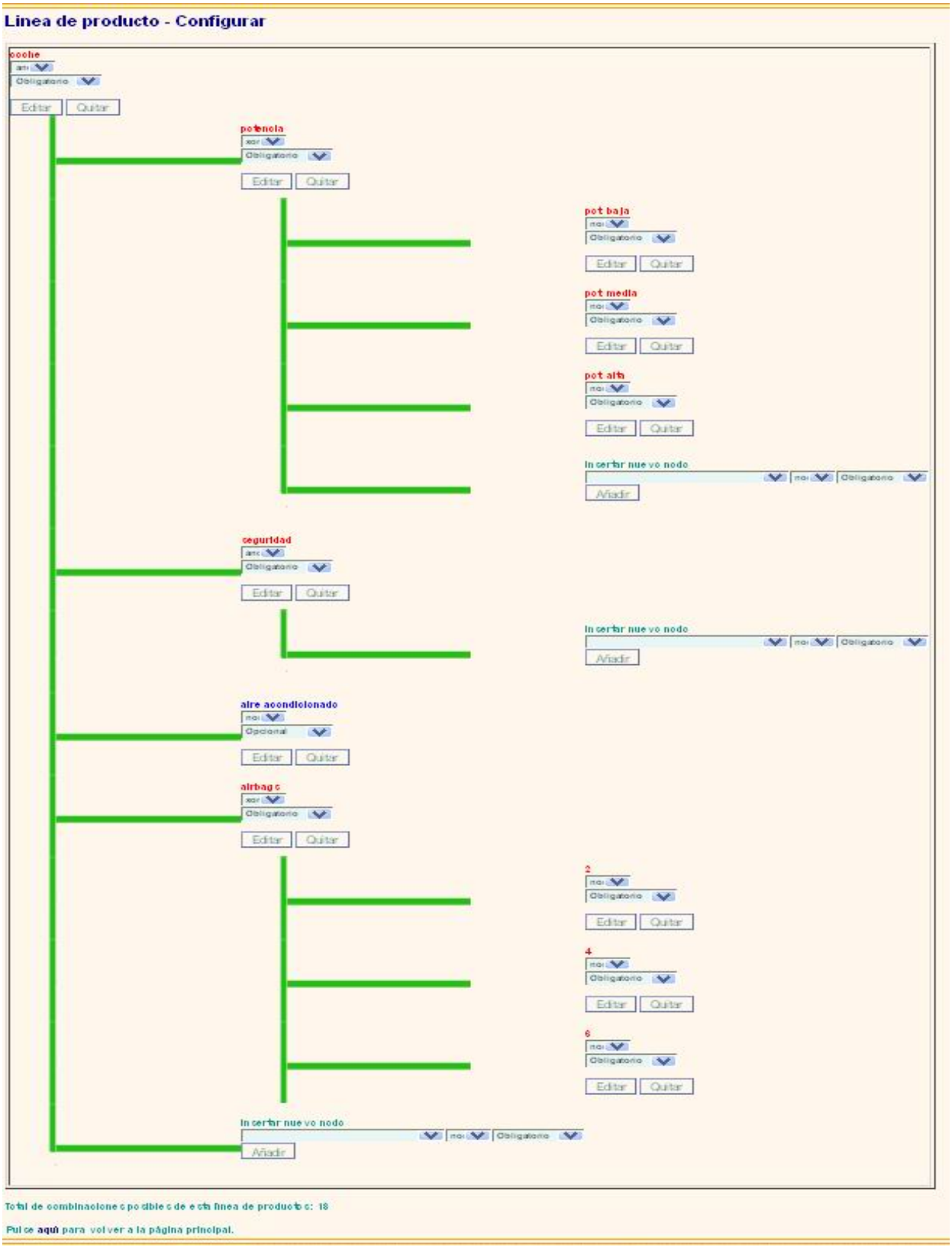


Figura 21. Configuración de la línea de producto

Tras ello, es el momento de especificar las restricciones. Para ello hay que ir al panel de configuración de reglas y seguir las instrucciones que figuran en pantalla (Figura 22).

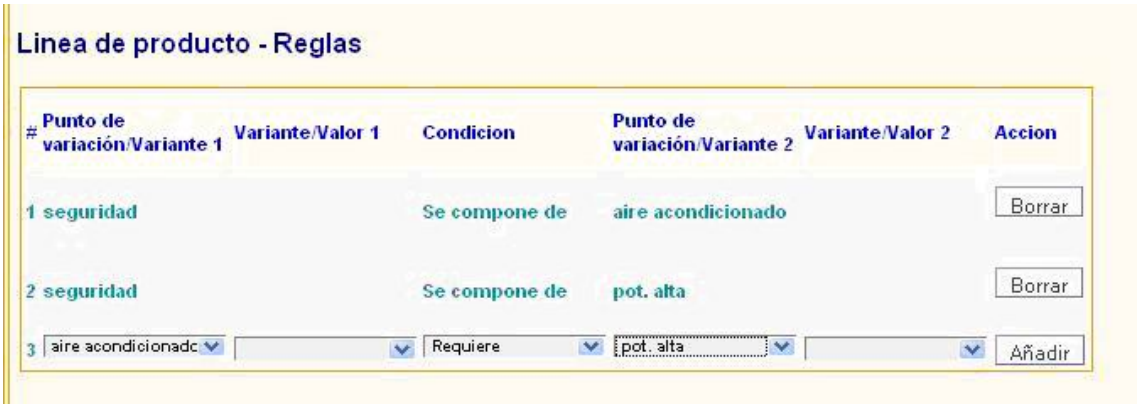


Figura 22. Regla *requiere*.

La pantalla de definición de reglas de la línea de producto permite modelar reglas *requiere* y *excluye,* además de permitir definir puntos de variación compuestos, como ya se vio anteriormente.

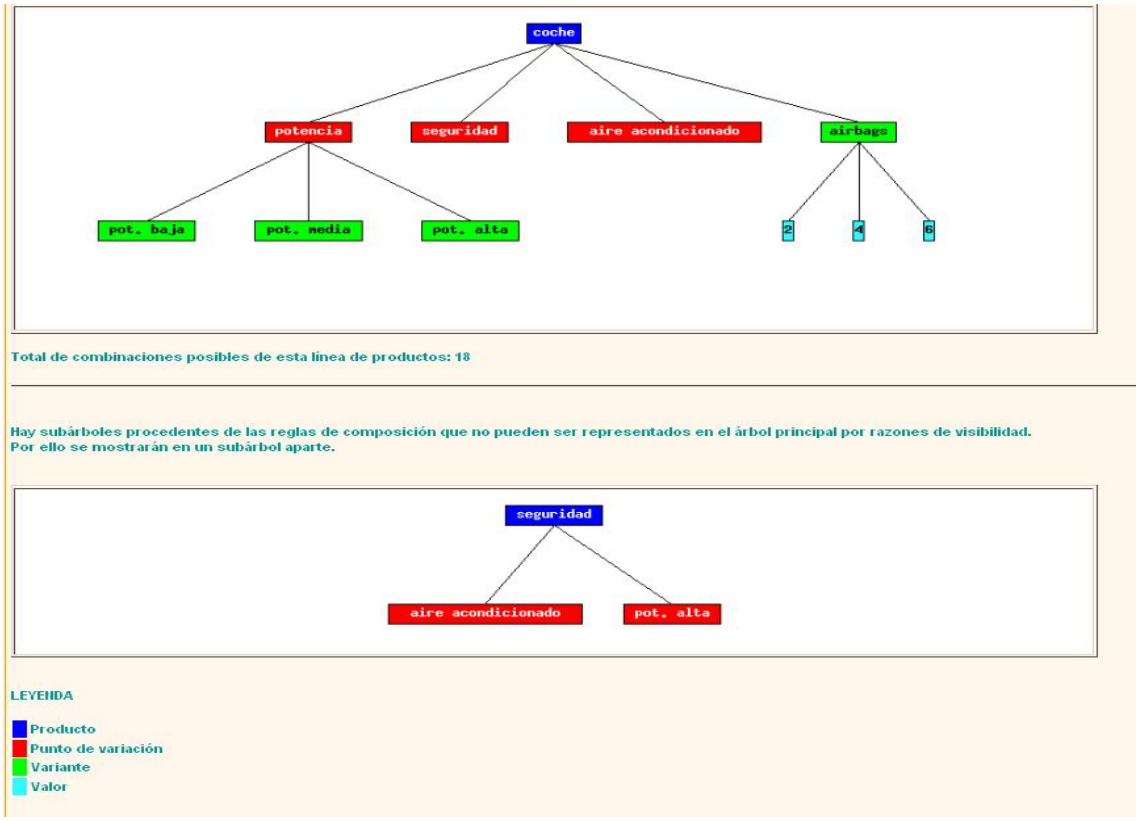


Figura 23. Representación gráfica de la línea de producto

Se ha utilizado un código de colores que permite identificar puntos de variación, variantes, y valores (Figura 23). De esta forma, de un simple vistazo es posible ver cuantos componentes hay de cada uno de ellos en el árbol. En esta pantalla se informa del número de combinaciones distintas que son posibles partiendo de esta línea de productos.

A continuación, es posible configurar los puntos de variación para construir un producto determinado (Figura 24). Los nodos que ya están seleccionados aparecen en verde, y los que no forman parte del producto aparecen en color blanco. Pulsando sobre un nodo, blanco, cambiará a verde, pasando a estar seleccionado. Si el padre del nodo es del tipo XOR, entonces desmarcará un nodo hermano y dejará marcado el nodo pulsado.

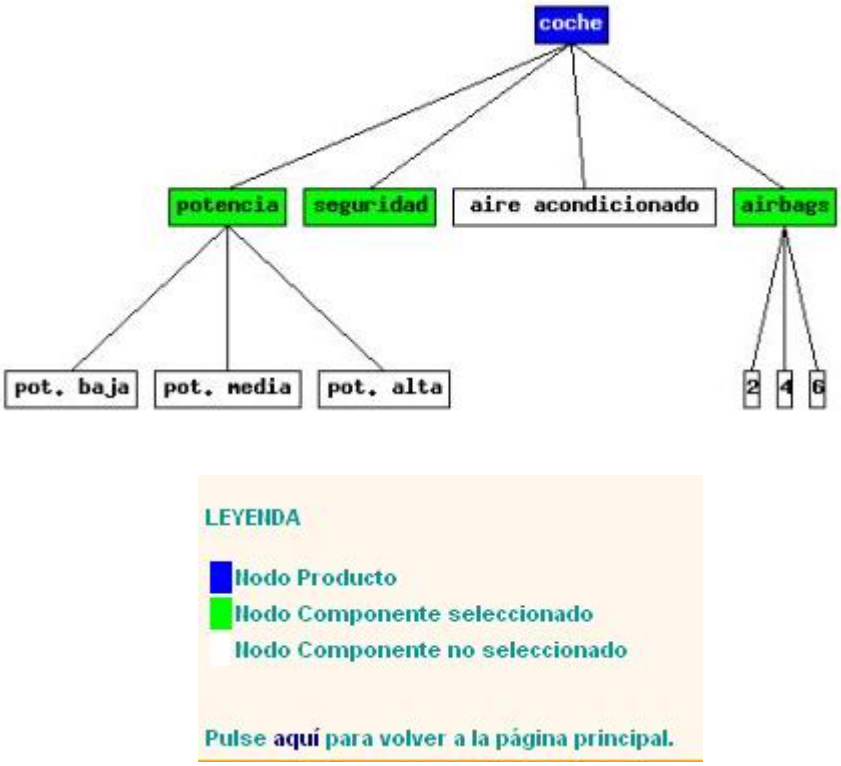


Figura 24. Configuración de un producto.

Si la relación es de tipo OR, la aplicación permite seleccionar varios nodos hermanos a la vez. En el caso del componente aire acondicionado, que es opcional, cada vez que se pulse sobre él se activará o desactivará sin que tenga impacto sobre el resto de nodos. Los nodos que están en un subárbol AND no pueden ser desmarcados y siempre aparecen en verde.

Una vez que el producto está configurado, hay que invocar la operativa de comprobación de cumplimiento de restricciones. De ser así, ya se puede pasar el producto a *modo runtime*, simulando su funcionamiento.

8. Conclusiones

La dificultad de modelar, gestionar y representar la variabilidad de los sistemas en tiempo de ejecución es altamente compleja, y las soluciones propuestas hasta ahora son escasas y parciales. Por ello, y debido a la enorme aplicación de estas técnicas en diferentes tipos de sistemas y metodologías de desarrollo software como son las líneas de productos, podemos extraer las siguientes conclusiones.

- La inclusión de variantes y puntos de variación en tiempo de ejecución se facilita si se incluyen supertipos o metatipos que permitan agrupar de forma lógica la variabilidad de partes funcionales del sistema.
- La definición de supertipos permite una mejor visualización de partes del árbol FODA, sobretodo en aquellos casos de disponer de un gran número de variantes y puntos de variación.
- La reconstrucción dinámica del árbol FODA permite al ingeniero controlar mejor aquellos cambios en el modelo de variabilidad del sistema.
- La creación de puntos de variación resulta altamente compleja ya que es una tarea propia del diseño. Para hacerla posible se ha creado un entorno virtual réplica del real con las mismas entidades, permitiendo de esta manera realizar los cambios necesarios y validándolos antes de incorporarlos al sistema real.

Finalmente, en cuanto a trabajos futuros podemos sugerir los siguientes:

- Aplicación del modelo de variabilidad dinámica a sistemas auto adaptativos.

- Uso más extensivo del modelo propuesto en líneas de productos dinámicas, indicando además que componentes arquitectónicos se ven afectados por cambios en la variabilidad.
- Asociar el modelo de variabilidad dinámica a decisiones de diseño en tiempo de ejecución, para permitir un back tracking a las decisiones que motivaron una configuración FODA determinada.

9. Bibliografía

[Ali et al., 2009]

Raian Ali1, Ruzanna Chitchyan2, Paolo Giorgini1

"*Context for Goal-level Product Line Derivation*",

3rd International Workshop on Dynamic Software Product Lines (DSPL09)

[Anfurrutia et al, 2006]

Felipe I. Anfurrutia, Student Member, IEEE, Oscar Díaz, y Salvador Trujillo

"Una Aproximación de Línea de productos para la Generación de Informes de Bases de Datos"

Revista IEEE América Latina Volume: 4, Issue: 2, Date: April 2006

[Anastasopoulos et al., 2001]

Anastasopoulos M., Gacek C.,

"*Implementing Product Line Variabilities*"

Symposium on Software Reusability (SSR'01), Toronto, Canada, Software Engineering Notes, Vol. 26, No. 3, May 2001, pages 109-117

[Arango et. al., 1994]

Arango, G

"Domain Analysis Methods "

W. Schäfer, et al., Software Reusability, Ellis Horwood, Hemel Hempstead, UK, 1994

[Bass et al., 2003]

L. Bass, P. Clements, R. Kazman,

"Software Architecture in Practice"

Addison-Wesley, 2nd ed, Addison-Wesley 2003**.**

[Batory, 2005]

D. Batory.

"*Feature models, grammars, and propositional formulas*"

Software Product Lines Conference, LNCS 3714, pages 7–20, 2005

[Benavides et al., 2005]

D. Benavides, S. Trujillo, P. Trinidad.

"*On the modularization of feature models*"

First European Workshop onModel Transformation, September 2005.

[Benavides et al. 2007]

David Benavides, Sergio Segura, Pablo Trinidad, Antonio Ruiz Cortés FAMA: "*Tooling a Framework for the Automated Analysis of Feature Models*"

VaMoS 2007: 129-134 (2007)

[Bencomo et al., 2008]

Nelly Bencomo, Gordon Blair, Carlos Flores, Pete Sawyer.

"Reflective Component-based Technologies to Support Dynamic Variability".

VaMoS Workshop. 2008.

[Blair et al., 2009]

Gordon Blair, Nelly Bencomo, and Robert B. France

"Models@run.time"

Computer, Vol. 42, Nr. 10, 2009,

[Bontemps et al., 2004]

Y. Bontemps, P. Heymans, P.-Y. Schobbens, and J.-C. Trigaux.

"Semantics of FODA Feature Diagrams"

Proc. of Workshop on Software Variability Management for Product Derivation (Towards Tool Support), Boston, 2004.

[Bosch, 2000]

J. Bosch,

"Design & Use of Software Architectures: Adopting and Evolving; a Product Line Approach", Addison-Wesley, 2000.

[Brugos et al., 2001]

José A. L. Brugos, Ángel Neira, Alfredo Alguero

"Agentes inteligentes y dimensiones de adaptación en informática educativa".

Simposio Internacional de Informática Educativa. Viseu, Portugal. 2001.

[Capilla et al., 2001]

Rafael Capilla, Juan C. Dueñas

"Modelling Variability with Features in Distributed Architectures"

Revised Papers from the 4th International Workshop on Software Product-Family Engineering, 2001

[Capilla et al., 2008]

Rafael Capilla, Muhammad Ali Babar

"On the Role of Architectural Design Decisions in Software Product Line Engineering",

2nd European Conference on Software Architecture (ECSA08), Cyprus, 2008

[Cetina et al., 2009]

Carlos Cetina, Pau Giner, Joan Fons y Vicente Pelechano

"*Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes* "

Computer, Vol. 42, Nr. 10, 2009,

[Clements et al., 2001]

P. Clements and L. Northrop

"Software Product Lines: Patterns and Practice".

Reading, MA: Addison Wesley, Boston, MA, 2001.

[Czarnecki et al., 2000]

K. Czarnecki and U. Eisenecker.

"Generative Programming: Methods, Tools and Applications"

Addison-Wesley, 2000.

[Fritsch et al. 2002a]

C. Fritsch, A. Lehn, T. Strohm

"Binding Time"

Robert Bosch GmbH (internal slides) 2002

[Fritsch et al. 2002b]

C. Fritsch, A. Lehn, R. Rashidi, T. Strohm:

"Variability Implementation Mechanisms – A Catalog"

Internal Paper. Technical report, Robert Bosch GmbH, 2002.

[Fritsch et al. 2002c]

Claudia Fritsch, Andreas Lehn, Dr. Thomas Strohm:

"Variability Implementation Mechanisms – A Catalog"

INTERNATIONAL WORKSHOP ON PRODUCT LINE ENGINEERING, 2., 2002, Seattle, USA. 2002. pp. 59-64

[Georgas et al., 2009]

John C. Georgas, André van der Hoek, Richard N. Taylor

"*Using Architectural Models to Manage and Visualize Runtime Adaptation* "

Computer, Vol. 42, Nr. 10, 2009,

[Gjerlufsen et al., 2009]

Tony Gjerlufsen, Mads Ingstrup, Jesper Wolff Olsen
"*Mirrors of Meaning: Supporting Inspectable Runtime Models* "
Computer, Vol. 42, Nr. 10, 2009,

[Goedicke et al. 2004]
Michael Goedicke1, Carsten Köllmann1, Uwe Zdun2:
"*Designing Runtime Variation Points in Product Line Architectures: Three Cases* "
Science of Computer Programming Volume 53, Issue 3, December 2004,

[Griss, 2000]
M. L. Griss,
"Implementing Product line Features with Component Reuse",
Proceedings of 6th International Conference on Software Reuse, Vienna, Austria, June 2000.

[Hallsteinsen, 2008]
Svein Hallsteinsen, Mike Hinchey, Sooyong Park, Klaus Schmid,
"Dynamic Software Product Lines"
IEEE Explore, April 2008.

[Helleboogh et al., 2009]
Alexander Helleboogh, Danny Weyns, Klaus Schmid, Tom Holvoet, Kurt Schelfthout, Wim Van Betsbrugge
"Adding variants on-the-fly: Modeling Meta-Variability in Dynamic Software Product Lines".
Third International Workshop on Dynamic Software Product Lines (DSPL 2009) at the Software Product Line Conference, 2009

[Hinchey et al., 2009]
Alexander Helleboogh, Danny Weyns, Klaus Schmid, Tom Holvoet, Kurt Schelfthout, Wim Van Betsbrugge

"Adding variants on-the-fly: Modeling Meta-Variability in Dynamic Software Product Lines".
Third International Workshop on Dynamic Software Product Lines (DSPL 2009) at the Software Product Line Conference, 2009

[Hofmeister et all., 2007]
S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid
"*Dynamic software product lines*"
Computer, vol. 41, no. 4, pp.93–95, 2008.

[Hofmeister et all., 2007]
C. Hofmeister, P. Kruchten, R.L. Nord, H. Obbink, A. Ran, P. America
"A General Model of Software Architecture Design Derived from Five Industrial Approaches",
Vol. 30, Issue 1, pp. 106-126, January 2007. © 2007 Elsevier

[Jacobson, 1997]
Jacobson, I., Griss, M., Jonsson, P.,
"*Software Reuse-Architecture, Process and Organization for Business Success*".
ACM Press, New York, NY, 1997.

[Jonas, 2006]
Jonás A. Montilva
"*Desarrollo de Software Basado en Líneas de Productos de Software*"
Doctorado en investigación, Universidad de Los Andes, Facultad de Ingeniería, Departamento de Computación, Mérida – Venezuela., 2007.

[Karhinen et al. 97]
Karhinen A., Ran A., Tallgren T.
"*Configuring Designs for Reuse*"

Proceedings of the International Conference on Software Engineering, Boston, MA. May 1997

[Kang, 1998]
K.C. Kang,
"FORM: a feature-oriented reuse method withdomain-specific architectures"
Annals of Software Engineering, V5, pp. 354-355. 1998.

[Korherr et al., 2007]
Birgit Korherr and Beate List,
"**A UML 2 Profile for Variability Models and their Dependency to Business Processes**"
1st International Workshop on Enterprise Information Systems Engineering (WEISE 07), September 2007, Regensburg, Germany, IEEE Press, 2007

[Knauber et al., 2002]
Peter Knauber, Giancarlo Succi,
"Perspectives on Software Product Lines"
ACM SIGSOFT's SEN, January 2002.

[Kruchten, 1995]
Philippe Kruchten.
"*Architectural Blueprints -The "4+1" View Model of Software Architecture*"
IEEE Software 12, November 1995.

[Laqua, 2002]
Roland Laqua.
"Concepts for a Product Line Knowledge Base & Variability"
Proceedings of NetObjectDays 2002, Erfurt, October, 2002.

[Lee, 2000]
Kwanwoo Lee, KC. Kang, W. Chae, and BW. Choi.

"*Feature-based approach to object-oriented engineering of applications for reuse*"
Software-Practice and Experience, 30(9):1025–1046, July 2000.

[Loughran, 2007]
N. Loughran, P. Sanchez, N. Gamez, et al,
"Survey on State-of-the-Art in Product Line Architecture Design"
AMPLE Project deliverable D2.1, April 2007

[Maes, 1987]
Pattie Maes
"Computational Reflection".
PhD Thesis, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[Mannion, 2002]
M. Mannion.
"*Using first-order logic for product line model validation*"
Proceedings of the Second Software. Product Line Conference (SPLC2), LNCS 2379, pages 176–187, San Diego, CA, 2002

[Massen et al., 2003]
T. von der Massen and H. Lichter.
"*Requiline: A requirements engineering tool for software product lines*"
Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5), LNCS 3014, Siena, Italy, 2003.

[Morin et al., 2009]
B. Morin, O. Barais, J. Jézéquel, F. Fleurey, A. Solberg
"M*odels@ Run.Time to Support Dynamic Adaptation* "
Computer, Vol. 42, Nr. 10, 2009,

[Neighbors, 1984]

J. Neighbors

"The Draco Approach to Constructing Software from Reusable Components"

IEEE Transactions on Software Engineering, 10, 5,654-574,1984.

[Robak, 2003]

Silva Robak (2003)

"Feature modeling notations for system families",

International workshop - ICSE'03. Portland, USA, 2003

[Sharp. 1999]

Sharp D.

"*Exploiting Object Technology to Support Product Variability*"

Proceedings of 18th Digital Avionics Systems Conference, IEEE, 1999

[Shaw and Garlan, 1996]

Mary Shaw, David Garlan.

"*Software Architecture: perspectives on an emerging discipline*"

Prentice Hall, 1996

[Soni et al., 1993]

D. Soni, R.L. Nord and Liang Hsu

"An empirical approach to software architectures**"**

International Workshop on Software Specifications & Design. Proceedings of the 7th international workshop on Software specification and design, 1993.

[Soni et al., 1995]

D. Soni, R.L. Nord and C. Hofmeister

"Software Architecture in Industrial Applications"

Proceedings of the 17th International Conference on Software Engineering, Seattle, Washington, pp. 196-207, April 1995.

[Sonnemann, 1995]
Sonnemann, R.,
"Exploratory Study of Software Reuse Success Factors"
Ph.D. Dissertation, George Mason University, 1995.

[Svahnberg et al., 2001]
Mikael Svahnberg, Jilles van Gurp, Jan Bosch
"On the Notion of Variability in Software Product Lines"
Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01) - Volume 00, 2001

[Svahnbert, 2005]
M. Svahnberg, J. v. Gurp, and J. Bosch.
"A taxonomy of variability realization techniques"
Software: Practice and Experience, 35(8):705 – 754, 2005.

[Svein et al., 2008]
Svein Hallsteinsen, Mike Hinchey, Sooyong Park, Klaus Schmid
"Dynamic Software Product Lines",
International Workshop on Dynamic Software Product Lines. DSPL 2008

[Tsang, 1995]
Edward Tsang
"*Foundations of Constraint Satisfaction*"
Academic Press, 1995

[Valdezate, 2006]
Alejandro Valdezate Sánchez
"Modelado y Gestión de la Variabilidad en Sistemas Software"

Proyecto de Fin de Carrera, Ingeniería Informática, Biblioteca de Móstoles, Universidad Rey Juan Carlos, 2006.

9.1. Páginas web referenciadas

Carnegie Mellon Software Engineering Institute. FODA
http://www.sei.cmu.edu/domain-engineering/FODA.html

[Program Transformation. Software Variability Management]
http://www.program-transformation.org/Variability/SoftwareVariabilityManagement

[FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures]
http://selab.postech.ac.kr/publication/1998_FORM_A%20Feature-Oriented%20Reuse%20Method%20with%20Domain-Specific%20Reference%20Architectures.pdf

[Context-aware pervasive systems]
http://en.wikipedia.org/wiki/Context-aware_pervasive_systems

[Ubiquitous computing]
http://en.wikipedia.org/wiki/Ubiquitous_computing

[ISO/IEC 42010]
http://www.iso.org/iso/catalogue_detail.htm?csnumber=45991

[IDI Software Definitions]
http://www.idi-software.com/resources/defns.html

Anexo I. An Analysis of Variability and Management Tools for Product line Development

An Analysis of Variability Modeling and Management Tools for Product Line Development

Rafael Capilla[1], Alejandro Sánchez[1], Juan C. Dueñas[2*]

[1] Department of Computer Science, Universidad Rey Juan Carlos
c/ Tulipán s/n, 28933, Madrid, Spain
rafael.capilla@urjc.es, valdezate@gmail.com

[2] Department of Engineering of Telematic Systems, ETSI Telecomunicación
Ciudad Universitaria s/n, 28040, Madrid, Spain
jcduenas@dit.upm.es

Abstract. Software variability is considered a key technique for modelling the variable parts of a software system. The importance for representing this variability is well recognized in product lines architectures. The advantage for producing multiple products under strong time to market conditions is reached through the definition of appropriate variation points as configurable options of the system. In industry, the development of large software systems requires an enormous effort for modelling and configuring multiple products. To face this problem, variability modelling tools have been proposed and used for representing and managing the variations of a system. In this work we analyze the current state of practice of some of these tools to identify their capabilities and limits and we provide suggestions for future second-generation tools.

* The work performed by Juan C. Dueñas has been partially done in the ITEA-SERIOUS project, under grant FIT340000-2005-136 of Ministerio de Industria, Turismo y Comercio de España.

1. Introduction

Software architectures have been widely used for almost 25 years for representing the main parts of a software system [2]. The success of software architectures comes from their ability for representing the common and variable parts of a set of related systems. The aim of software variability is to exploit the variable parts of a set of systems and to configure new products from the same family. Software variability has been widely used during the last years in product line architectures [6] for building multitude of products as a way to meet the market condition when multiple products have to be delivered in short time. Variation points are used to represent this variability as well as to discriminate and to configure different products. One of the main advantages is that variation points are used to delay the design decisions made at the beginning to the latest moment in the product lifecycle. Different alternatives for representing and configuring these variation points are possible, but modeling and managing hundreds of variation points constitutes a current problem, because dependencies between these have to be handled. The remainder of this paper is as follows. Section 2 describes the main concepts of software variability. Section 3 outlines variability representation and visualization techniques. Section 4 describes the main characteristics of a representative list of variability modeling tools. Section 5 mentions the impact of variability modeling tools in product lines. Section 6 discusses the limits of current tools and the features that might be implemented for future tools.

2. Software Variability Concepts

In order to analyze and understand the features implemented in the tools examined in this work, in this section we provide an overview of the main concepts belonging to software variability.

2.1 Origins

At the beginning of the ‘90s, the Feature-Oriented Domain Analysis (FODA) method [16] was proposed for modeling the variations of software systems. FODA

defines mandatory, alternative and optional features and composition rules for guiding the rationale in which the visible and external properties of systems are represented. Some approaches improve FODA capabilities. The Feature-Oriented Reuse Method (FORM) [17] extends FODA with domain features (i.e.: features specific to a particular domain) and performs a classification of different types of features. FORM is a systematic method that captures the commonalities and differences of application in a domain in terms of features and produces different feasible configurations of reusable architectures. Indeed, these feature trees are used as a "decision model" in the engineering process to obtain different product configurations. In [8] the authors propose to include quantitative values to FODA trees for specifying QoS parameters in distributed systems. The definition of quantitative values and ranges values among are quite useful for featuring certain applications (e.g. telecommunication systems). Today, the success of software product lines in the industry has raised the popularity of feature trees like FODA for describing the variability of the product family. A FODA tree describes at a high level (i.e.: conceptual level) the variations and alternatives that occur in a particular software product or in the entire product family as specified in the requirements. This high level description must be translated to the design and implementation levels in the form of variation points and variants.

2.2 Variation Points and Variants

The concrete specification of a feature tree is usually achieved by means of variation points (VP). We understand by variation points an area of a software system affected by variability. Each variation point is commonly defined in terms of variants that represent the alternatives for each variation point. At the end, each variant may define a set of allowed values that are selected at a given time. Variability in space defined the allowed configurations for a set of products. Furthermore, the extensibility of a variation point can be open or closed. In closed variation points all variants are defined at pre-deployment time and the selection of the choices is only possible between the built-in variants. Open variation points allow the inclusion of new variants to an existing variation points at runtime. Note than adding a variant to an existing variation point is quite different from adding a new variation point, but this

concept seems really hard to implement. Therefore, the evolution of the variation points has impact in the evolution of the products.

2.2 Binding Time and Implementation Mechanisms

The selection of a particular feature implies the selection of its corresponding variation point, variants, and values. The realization of the variability, that is, the moment in which the variability happens, is often called the binding time (i.e.: variability in time). This binding time may take place at different moments, like design time, compilation, programming, run-time, etc. In [11], the authors identify different binding times for which variability may occur. Complementary to the definition of each particular binding time, the variation points defined at the design level must be implemented at the code level. Some of the alternatives to realize the variability are the following.

- **Compiler directives and installation procedures**: The variation points are realized when the software is compiled and this is achieved through directives, (e.g.: #ifdef). These directives can be used during the installation of software packages (e.g.: operating systems) to install or configure a particular product.
- **Flow control sentences**: Flow control sentences (e.g.: if..then) can be used at programming and run-time for selecting concrete options in the product.
- **Parameterization**: Variation points and variants are specified using function parameters that are instantiated at a given moment.
- **Boolean formula**: The result of a variation point is computed by means of a specific formula, usually based on logical connectors, and checked afterwards to select a particular alternative or a different variation point. Variability is implemented at programming time and realized at run-time.
- **Configuration**: Configuration files are loaded at the beginning of the execution of the system and used to select between different options. This configuration file is usually written in the same language of the software product.

- **Generator files**: Under a generative approach, makefiles can be used to deploy automatically a software package or particular product configuration, which is usually built on the top of different base software packages.
- **Database**: This mechanism is sometimes used (e.g.: context-aware systems) and the variations are realized depending of the values stored in databases. An initial set-up of the values must be done prior to the execution of the system, so they can be checked afterwards to decide between different alternatives.

Other additional techniques are possible to enable variability in general (e.g.: aggregation, inheritance, overloading, macros), such as described in [23]. In many cases and due to the complexity of software systems, several variability implementation mechanisms might be combined to obtain the desired configuration. In [29], the authors present a taxonomy of variability realization techniques in which different ways to implement the variation points are described. The authors describe the motivation of each realization technique and they relate it with a concrete stage of the lifecycle as well as the proposed solution for each case.

2.3 Dependency Model and Traceability

An important characteristic that affects both the modeling and the implementation of the variability is the existence of dependencies that can be established between features. A dependency may be originated because some feature needs the existence of another or because the modification of a particular feature impacts on other features. System constraints can be modeled as dependencies which limits the number of allowed products. During the modeling process, new dependencies add a degree of complexity to the features model with a direct impact on the definition and selection of the variation points. Feature models can use logical connectors like AND, OR and XOR to model the relationships between features, and variation points can be defined using these logical connectors. More complex dependencies can be modeled in a different way. Jaring and Bosch [15] discuss a taxonomy of variability realization techniques as a major concern when modeling and configuring products in a product line context. The authors identify four main types of variability dependencies, each of

them consisting of four subtypes. Lee and Kang [20] suggest a classification of dependency types for feature modeling in product lines and they analyze feature dependencies that can be useful in the design of reusable components. Because feature modeling mainly focuses on structural dependencies, the authors propose to extend classic feature models by adding operational dependencies. An operational dependency is a relationship between features during the execution of the system. For instance, the usage dependency happens when a feature may depend on other features for its correct functioning. In [21], the authors describe a feature dependency model for managing static and dynamic dependencies for product line development. Three static dependencies and seven dynamic ones are defined. A directed graph is used to analyze domain requirements dependencies to produce the right product members in the product line. An algorithm generates the maximum connective dependencies graph but only direct dependencies are represented, no implicit ones. In general, variability modeling and management techniques are not enough powerful to support traceability, making it necessary to use other mechanisms to relate, for example, feature models with products. As stated in [7], capturing and representing this traceability is a challenging task. The authors propose a unified approach for successful variability management, in which trace links are defined to connect both the problem space (the feature model) with the solution space (instantiated architecture and code). Hence, several dimensions of variability can be defined to represent the variation points, the dependencies and the traces under a product line context. For the representation of these traces, and providing basic dependency types are handled, matrixes are a suitable option.

2.4 Variability Management

Variability is the ability to change or customize a system [32], and variability management includes the processes for maintaining the variability model across the different stages of the lifecycle in order to produce the right product configurations. Because features and variation points are not isolated, changes performed over a variation point may affect other variation points as well as the final product. Managing the variation points, variants and dependencies of a software system

constitutes a big challenge for current variability modeling tools. Typical management activities should include: maintain the variation points, variants and dependencies, constraint checking, traceability between the model and products, configuration processes, and documentation. In [32], the authors describe some activities concerning variability management in a product line context, such as: variability identification, introduction of the variability in the system, collecting the variants and binding the system to one variant. The authors state the existence of feature interaction but the maintenance of the dependencies between features should be described explicitly in the proposed tasks. The technical report described in [22] discusses family based development processes and how to express requirements in terms of features and features in terms of variation points. Variability should be managed not only in the problem space, but also in the solution space during the development phase because the product portfolio has a direct impact on variability management activities.

3. Variability Representation and Visualization Techniques

The notation of variation point was introduced in RSEB (Reuse-Driven Software Engineering Business) [13], but some other notations have been proposed and used. In [23], the author analyses and compares five different leading modeling notations for representing variability in software systems (i.e.: FODA, FORM, Generative programming, Feature-RSEB and Jan Bosch's notation). Some of the aforementioned notations share FODA and FORM feature types and relationships while others propose extensions to the feature modeling. For instance, Jan Bosch's model proposes the introduction of "external features" which does not fit in the usual classification. A feature tree is one of the most common presentation techniques used for describing the variability of a system. In addition, UML uses some extensions to describe the variability of systems, like: UML stereotypes, tagged values and constraints based on the OCL (Object Constraint Language) [10]. In general, standard UML suffers the lack of a more precise notation to express all the variability concepts needed, the same as the original FODA feature trees which are not enough powerful to represent complex relationships between the variation points. Practical usage, however, suggests

that FODA feature trees are well adapted for variability analysis, while the usage of UML profiles for variability has a clear focus on architecture, as described in [4].

3.1 Visualization

Variability management and modeling tools usually provide a graphical description of the variation points and variants for each product. Therefore, visualization and configuration facilities must be included, such as the following ones.

- **Tree view**: FODA trees are widely used for representing the variation points and variants for a particular product. One of the problems with this approach is the scalability when the number of alternatives grows.
- **Graph view**: Graphs are similar to trees but some kind of mechanism to expand the branches containing the alternatives is needed.
- **Matrix view**: Matrixes allow the visualization of a large number of items, but using this approach we can loose the perspective of the hierarchy and dependencies of the variation points and variants.

In addition, combo and check boxes or radio buttons can be used for the selection of the choices during product configuration. As the number of variation points and variants becomes unmanageable, the visualization of all the alternatives becomes a big problem. A way to solve this is to split the variability model into categories for which the user may select or visualize a portion of the tree. Another solution is to employ zooming tools to expand only those parts of the model in which the designer is interested on. The visualization of hundreds of variations points becomes an important problem in industrial product lines because it may hamper the scalability of visualization facilities.

4. State of Practice of Variability Modeling Tools

Once we have described the main concepts of software variability modeling, in this section we outline the main characteristics of some existing variability modeling and

management tools. The tools selected offer specific functionality for modeling variability. The assessment was performed, based on the information given by the tools' authors or providers, who were asked to complete a review form. There was no cross-comparison between tools since the main objective of the evaluation is not to identify the best tool, but to identify the most relevant items that offer practical relevance to the community. The scope of this analysis is mainly focused on variability modeling and management tools rather than those using MDA-MDD [34], domain-specific languages [30], and generative approaches.

4.1 GEARS

Gears is a commercial software product line development tool developed by BigLever Inc [19] (http://www.biglever.com) and enables the modeling of optional and varying features which is used to differentiate the products in the portfolio. The Gears feature model uses rich typing (sets, enumerations, records, Boolean, integer, float, character, string) distinguishes between "features" at the domain modeling level and "variation points" at the implementation level (source code, requirements, test cases, documentation). In Gears, Set types allow the selection of optional subsets, enumeration types allow selection of one and only one alternative, Boolean represent singular options, and records represent mandatory lists of features. Gears variation points are inserted to support implementation level variation. Components with Gears variation points become reusable core assets that are automatically composed and configured into product instances. Thus developers work in a very conventional way on Gears core assets, with the exception of implementing the variation points to support the required feature model variations that are in the scope of their asset.

Dependencies in Gears are expressed as relational assertions. Simple binary relations can be used to express the conventional require and excludes dependencies. Assertions can also contain 3 or more feature and relations such as "greater than". Variation points and feature models are fully user programmable to arbitrary levels of sophistication and complexity. User-defined compound features have anonymous types (i.e., with the advanced typing in Gears, explicit user defined types and reusable

types are not required).

The Gears approach defines product feature profiles for each product and selects the desired choices in the feature model. A product configurator automatically produces the individual products in the portfolio by assembling the assets and customizing the variation points within those assets to produce a particular product according to the feature profile. Gears modules can be mapped to any existing modularity capabilities in software. Gears modules can be composed into subsystems, which can be treated as standalone "product lines". Product lines can be composed from modules and other nested product lines. Aspect-oriented features are captured in Gears "mixins", which allow crosscutting features to be imported into one or more modules for use in implementation variation points in those modules. The tool supports also the definition of hierarchical product lines by nesting one product line into another.

Two views and editor styles are supported and can be switched dynamically: (1) syntactically and semantically well-defined text view and (2) context-sensitive structural tree view. Gears uses file and text based composition and configuration. This language-independent approach allows users to transition legacy variation as well as implement new variations. Gears has been used for all of the above and supports multiple binding times in one product line. For runtime binding, Gears typically influences the runtime behavior indirectly through statically instantiated configuration files or database setting, though these could also be set dynamically by making feature selections at runtime.

Gears technology eases the software mass customization process because enables organizations to quickly adopt a software mass customization for product line development. Gears supports three different models for product line adoption. Proactive, reactive and extractive approaches can be used depending of each particular organization, but they are not necessarily mutually exclusive. Gears has been used in systems with millions of LoC with no perceived limitation on scalability.

4.2 V-Manage

V-Manage from European Software Institute (ESI) (http://www.esi.es), is a tool for internal use that supports system family engineering in the context of MDA

(Model Driven Architecture) and consists of the following three modules [26].

- **V-Define**: Defines the variation model (i.e. decision model in V-Manage terminology) as well as the relationships. The elements of the feature model are HTML links.
- **V-Resolve**: Builds application models by setting the values of the decision model and produces a suitable configuration of the decision model. It supports the resolution of the model using the variation model.
- **V-Implement**: Supports the implementation of reusable components and links the variation parameters attached to the decisions to some external components or to other dependencies. V-Manage generates the result of a particular configuration to HTML, PLC-code, a UML model or requirements document.

As described in [4], with V-Manage the user specifies a variation model and a resolution model, and provides a mechanism to specify product line models to derive concrete system models. The V-define interface is used as the front-end to specify the variation model for a product line. The resulting model is an application model where all the variations have been resolved. Dependency rules guide the user during the configuration of the values for each variation element. Binding and refinement are supported by V-implement for the production of reusable components.

4.3 COVAMOF

The COVAMOF (ConIPF Variability Modeling Framework, http://www.covamof.com) approach is a variability modeling approach for representing variation points and variants on all abstractions layers, supports the modeling of relations between dependencies, provides traceability, and a hierarchical organization of variability. Five types of variation points supported in COVAMOF: optional, alternative, optional-variant, variant and value. The optional-variant variation point refers to the selection (zero or more) from the one or more associated variants. The COVAMOF Variability View (CVV) [25] represents the view of the variability for the product family artifacts and unifies the variability on all layers of

abstraction. The CVV models the dependencies that occur in industrial product families to restrict the binding of one or more variation points. Simple dependencies are expressed by a Boolean expression, and CVV specifies a function valid to indicate whether a dependency is violated or not. In addition to the Booleans, dependencies and constraints can also contain integer values, with operators ADD, SUBSTRACT, etc. Boolean and Numerical are used together in operators like the GREATER THAN, where numerical values are the input and Booleans are the output. Complex dependencies are defined in COVAMOF as dynamically analyzable dependencies and the CVV contains for each dynamically analyzable dependency the following properties [27]:

- **Aspect**: Each dependency is associated to an aspect that can be expressed by a real value.
- **Valid range**: This dependency specifies a function to {true, false} indicating whether a value is acceptable.
- **Associations**: The CVV distinguishes three types of associations for dynamic dependencies, which are: predictable, directional and unknown.

COVAMOF provides a graphical representation and a XML representation, used for communication between tools. The Mocca tool has been also developed to manage the COVAMOF Variability View, and allows for multiples views of CVV. Mocca supports the management of the CVV from the variation point view and the dependency view [26]. Mocca is implemented in Java as extension to the Eclipse 3.0 platform. Some recent improvements to COVAMOF, in particular to the derivation process, are supported by COVAMOF-VS tool suite [28], which is a set of add-ins for Microsoft Visual Studio.NET. The COVAMOF-VS provides two main graphical views, that is the variation point view and the dependency view, as a way to maintain an integrated variability model. Finally, specific plug-ins can be added for supporting different variability implementation mechanisms.

4.4 VMWT

VMWT (Variability Modeling Web Tool) is a research prototype developed at the

University Rey Juan Carlos of Madrid. This first prototype (http://triana.escet.urjc.es/VMWT/) is a web-based tool built with PHP and Ajax and running over Apache 2.0. VMWT stores and manages variation points and variants following a product line approach and enables to create product line projects for which a set of reusable existing assets can be associated. Before configuring a particular product, the variants that will be part of the variation points of the feature model must be added. Each variant can be associated to a particular code component and we can specify numeric values (quantitative values), ranges of values or a enumerated list can be specified. Once all the variants have been added, the variation points will be added to the code components. VMWT supports dependency rules and constraints for the variation points and variants already defined. The following Boolean relationships are allowed: AND, OR, XOR and NONE. In addition, more complex dependencies can be defined, such as requires and excludes. The tool allows constraint and dependency checking and we it computes the number of allowed configurations. This is quite useful when it is needed to estimate the cost of the products to be engineered. Finally, a FODA tree is visualized for selecting the options for each product and the selected configuration is then displayed to the user. The variation points and variants selected are included in a file attached to each code component. Documentation of the product line can be automatically generated as PDF documents.

4.5 AHEAD

The AHEAD (Algebraic Hierarchical Equations for Application Development) Tool Suite (AHEAD TS) (http://www.onekin.org) was developed to support the development of product lines using compositional programming techniques [3]. AHEAD TS has been used in distinct domains (i) to produce applications where features and variations are used in the production process [9] (ii) to produce a product line of portlets. The production process in software product lines require the use of features that have to be modeled as first-class entities. AHEAD distinguishes between “product features” and “built-in features”. The former characterizes the product as such. The latter refers to variations on the associated process. The production

processes are specified in using Ant, a popular scripting language from the Java community. AHEAD uses a step-wise refinement process based on the GenVoca methodology for incrementally adding features to the products belonging to a system family. The refinements supported by AHEAD are packaged in layers. The base layer contains the base artifacts and the lower layers allow the refinements needed to enhance the base artifacts with specific features. The AHEAD production process distinguishes between two different stages. The intra-layer production process specifies the tasks for producing a set of artifacts within a layer or upper layers. The inter-layer production process defined how layers should be intertwined to obtain the final product. An extension to AHEAD is described in [31], and a tool called XAK was developed for composing base and refinement artifacts in XML format. ATS was refactored into features to allow the integration with XAK The feature refactoring approach used in XAK decomposes legacy applications into a set of feature modules which can be added to a product line. AHEAD doesn't require manual intervention during the derivation process.

6. Discussion and Comparison

Complementary to the tools described before, we can find other approaches for feature and variability modeling. In [33], the authors present the Koala approach for modeling variability in software product families. Koala specifies the variability in the design by selecting the components and appropriate parameters. Another approach presented in [5] outlines the variability of a product family into two levels: the specification level and the realization level. The variability model defines the variation points and where these are implemented in the asset base. Static and dynamic variation points are allowed, whereas dependencies are modeled 1-to-1. In [12], the authors mention the Product-Line UML based Software Engineering Environment (PLUSEE), which is a tool for addressing multiple views of a software product line and check the consistency among these views. Variability is defined using the UML notation through different views (e.g.: case model, static model, collaboration model, statechart model and feature model) and each view is related to a particular stage in the software lifecycle. There are two versions of PLUSEE, one uses Rational Rose

and the other Rational Rose RT. PLUSEE is able to produce a consistent multiple-view model and an executable model using Rose RT.

The FeaturePlugin mentioned in [1] is an Eclipse plug-in for feature modeling. This tool follows the FODA approach and supports cardinality-based feature modeling, specialization of feature diagrams and configuration based on feature diagrams. The FeaturePlugin tool organizes features in trees and has some extra characteristics: ColorTypes, Depth and DisplayTypes which, for instance, are hidden in the V-Manage tool. The tree organization of FeaturePlugin supports very easily the scalability of the tool when new characteristics are added. The BVR model [24] (developed under the European FAMILIES project) defines a meta-model for modeling the variability in system families. This meta-model has three main parts. The Base model is any model in a given language. The Variation model which contains variation elements referred to the Base model element. The Resolution model resolves the variability for a system family model. The BVR approach uses a prototype tool called Object-Oriented Feature Modeler (OOFM) made as an Eclipse plugin for supporting the feature modeling process. The ASADAL (A System Analysis and Design Aid Tool) described in [18] supports the entire lifecycle of software development process of a software product line and based on the FORM method [17]. ASADAL is a more complete approach compared to variability management tools like COVAMOF but variability management is a key feature of ASADAL. Two feature analysis editors for feature modeling and feature binding are implemented in ASADAL, and product-specific design can be instantiated through feature selection. ASADAL is able to generate executable code from an architecture model.

Table 1 describes a comparative analysis of the representative tools introduced in Section 4, for modeling and managing variability. As evaluation items, we have chosen the specific concepts about variability modeling and management most widely accepted (which were defined in Section 2). The characteristics described in the table were obtained analyzing the information available for each tool. We couldn't perform a real evaluation of some of the tools because GEARS is a commercial tool and no demo is yet available whereas V-Manage is for internal use. The VMWT was tested in our university and we obtained some of the characteristics from the remainder tools by

interviewing the authors.

Table. 1. Comparative analysis of variability modeling tools

	GEARS	V-Manage	COVAMOF	VMWT	AHEAD
VP / Feature dependencies	Binary relations	AND, OR, NONE	Boolean Numerical	AND, OR, XOR, NONE	**AND, OR, XOR, NONE**
Complex dependencies	Relational assertions. Assertions can contain 3 or more feature and relations such as >, >=, <, <=, ==, subset, superset, AND, OR, NOT, +, -, *, / Require Excludes	----	Dynamic dependencies {aspect, valid range, associations}	Requires Excludes	**Require** **Excludes**
Feature / VP types	Set types allow selection of optional subsets. Enumeration types allow selection of one and only one alternative. Boolean types represent singular options Records are mandatory lists of features	Mandatory Alternative Optional Enabled Disabled	Alternative Optional Optional-variant Variant Value	Mandatory Alternative Optional	**Mandatory** **Alternative** **Optional**
Traceability	Yes	Yes	Yes	No	**Yes**
Feature-VP allowed values	Integers Floats Characters Strings Booleans Atoms Ranges are constrained by assertions	XML like data values	Ranges of values	Numerical Enumeration Boolean String Ranges of values	**-----**
VP visualization	Feature tree Text view	Feature tree Feature list Flowchart to be planned	Feature tree	Feature tree	**Feature tree**
Extensibility Opened / Closed VP	Open for dynamic loading & component swapping	Closed	Open & Closed (e.g: Open in SOA projects)	Closed	**Closed**

	Closed				
Variability Management facilities	5	4 (working on visual modeling)	5	4	**4**
Variability implementation mechanism	Composition Configuration	Parameterization Configuration Generation Macros Architectural design patterns Dynamic link libraries Dynamic class loading	Configuration Generation Macros Dynamic link libraries Parameterizatio n	Parameterization Configuration	**Inheritance Overloading**
Integration of VP with software components	5	4 (working on plug-ins architecture and eclipse integration)	5 (VP are specified in or even extracted from the source code)	3	**5**
Scalability for visualizing VP	5	5	5 (Plug-in for additional views. visualizes the VPs in Visio)	3	**4**
Scalability	5	5	5	4	**5**
Feature dependency / constraint checking	Yes	Yes	Yes	Yes	**Yes (SAT solver)**
Automatic VP generation	Yes	Yes	Yes	Yes	**Yes**
Binding Time	Pre-compilation Compilation Linking Installation Startup Runtime	Pre-Compilation	Pre-compilation Compilation Linking Installation Startup Runtime	Programming Runtime	**Pre-Compilation**
Statistical analysis and Reporting	Yes	Yes	Yes	Variability and Project PDF documents	**No**
Product derivation	Yes	Yes	Yes	Partially	**Yes**
Estimation of the number	Yes (Combinatory	Yes	No	Yes	**Yes (SAT solver)**

of products	reporting of estimated instances)				
Phases of the lifecycle covered	Analysis Design Implementation	Requirements Design Implementation	Analysis Design Implementation	Design Implementation	**Design** **Implementation**
Tool approach	Modeling Management	Modeling	Modeling Management	Configuration Modeling Management	**Configuration** **Modeling**
Development approach	Specialization Compositional	Compositional Derivation	Compositional Generative	Specialization Compositional	**Specialization** **Compositional** **Generative**
Platform / Technology	**Java** **Standalone version with Eclipse and Visual Studio**	**Java** **Ant** **XML** **Eclipse**	**MS Visual Studio.NET** **XML**	**PHP** **AJAX** **JavaScript**	**Java-Ant** **XML**

5. Impact on Product Line Development

Today, we have many examples of successful products lines (e.g.: Celsius Tech Cummins, Salion, Market Maker, Thales Naval, HP, etc) in which multiple products are designed and built under strong time to market conditions. The Nokia product line an example of successful product line which comprises multiple mobile phones organized around different families (e.g.: Nokia series 30, 60). A few examples of using the tools in real cases have been documented and the numbers of savings in terms of cost and effort have been reported. Some of the aforementioned tools have been tested both in academia and in industry. For instance, COVAMOF was used on the Intrada product family from Dacolian B.V., a small independent SME in The Netherlands for intelligent traffic systems [28]. Engenio is a firm dedicated to high-performance disk storage systems with approximately 200 developers distributed across four locations. Around 80% of the code is common to the 82 products of the firm. The increasing demand for Engenio's RAID storage server products led to the adoption of a product line approach and Gears was selected for this. Several successful results were obtained (see [14]). For instance, Engenio has experienced a 50% increase in development capacity. All these stories prove that software product lines constitute a successful approach for building software system families, and tools are strongly needed for managing the amount of variability required.

6. Current Limits and Second-generation Tools

The aforementioned tools constitute an important help for product line engineering. The lack of a unified approach for software variability leads to a certain number of tools with the same goal and similar characteristics, but with differences between them. There are some limitations of current tools as well as some issues that can be enhanced. For instance, the visualization of hundreds of variation points with their associated variants, in particular in industrial product lines, is a limitation of some of the existing tools, and new visualization facilities are welcome. As an example, we tested VMWT in two medium-size web projects and we observed that the tool needs better visualization capabilities when the number of variation points and variants scale up. Another issue, in particular for complex systems, is the need to handle complex dependencies. Some of the existing tools need to support more complex dependencies for describing all the relationships and constraints for any type of software system. Also, the extensibility of the variability model for supporting runtime variability (e.g.: adding variation points during the execution of the system) is a complex problem associated to the evolution of the system that could be alleviated by the usage of plug-ins mechanisms in runtime. The integration of variability modeling tools with traditional software engineering tools seems quite interesting for software product line engineering. In particular, unless there is a serious effort by the variability tools providers to integrate these tools with software configuration management tools and integrate the variability management activities in the practical development processes, the success of these tools will be in danger. The integration of variability tools with the configuration management community is an interesting issue to explore for configuration management purposes. Also, variability management tools could learn about the knowledge management community to incorporate more additional features. These and other capabilities seem to be interesting to be added and we believe a second-generation variability tools is a goal to pursuit. In this paper we have studied some representative tools for modeling and managing software variability in order to discover the needs for second-generation tools. Most of the tools analyzed share many common characteristics and most of them focus on specialization and compositional approaches. The estimation of the number of allowed products or right configurations

is an interesting issue for the defining production plans and for cost estimation. The majority of the tools cover design and implementation phases of the software lifecycle. Finally, integrated derivation processes from feature models to products are welcome to reduce the effort of configuration processes.

Acknowledgements

We thank to Charlie Krueger for his useful information about GEARS, which has been balanced between commercial and academic reasons; also Jason X. Mansell (ESI) who gave us many responses about V-manage. Many thanks to Marco Sinnema for providing useful information of COVAMOF-VS, and Salva Trujillo and Oscar Díaz who provide us the details of AHEAD TS in alive discussions

References

1. Antkiewicz, M. and Czarnecki K. FeaturePlugin: Feature Modeling Plug-in for Eclipse, OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, 2004
2. Bass, L., Clements P. and Kazman, R. Software Architecture in Practice, Addison-Wesley, 2nd edition, (2003).
3. Batory, D., Sarvela, J.N. and Rauschmayer, A. Scaling Step-Wise Refinement. IEEE TSE 30(6) 355-371, (2004).
4. Bayer, J, Gerard, S., Haugen, Ø, Mansell, J., Møller-Pedersen, B., Oldevik, J., Tessier, P., Thibault, J.P. and Widen, T. Consolidated Product Line Variability Modeling, In Software Product Lines Research Issues in Engineering and Management, Springer-Verlag, Timo Käköla and Juan Carlos Dueñas (Eds), pp. (2006).
5. Becker, M. Mapping Variability's onto Product Family Assets. Procs of the International Colloquium of the Sonderforschungberich 501, University of Kaisersalutern, Germany, (2003).
6. Bosch, J. Design and Use of Software Architectures, Addison-Wesley (2000).
7. Berg, K., Bishop, J. and Muthig, D. Tracing Software Product Line Variability – From Problem to Solution Space. Procs of the Annual Conference of the South

African Institute of Computer Scientists and Information Technologists (SAICSIT), pp. 111-120 (2005).

8. Capilla, R. and Dueñas, J.C. Modelling Variability with Features in Distributed Architectures, Procs of Product Family Engineering (PFE), Springer-Verlag LNCS pp. 319-329, (2001).
9. Díaz, O., Trujillo, S. and Anfurrutia, F.I. Supporting Production Strategies as Refinements of the Production Process. Procs of 9th Software Product Line Conference (SPLC), Springer-Verlag LNCS 3714 pp.210-221, (2005).
10. Dobrica, L. and Niemelä, E. Using UML Notation Extensions to Model Variability in Product-line Architectures. International Workshop on Software Variability Management (SVM), ICSE'03, Portland, Oregon, USA pp. 8-13 (2003).
11. Fritsch, C., Lehn, A. and Strohm, T. Evaluating Variability Implementation Mechanisms. Procs of International Workshop on Product Line Engineering (PLEES'02), Technical Report at Fraunhofer IESE (No. 056.02/E) 59-64 (2002).
12. Gomaa, H. and Shin, M.E. Variability in Multiple-View Models of Software Product Lines. International Workshop on Software Variability Management (SVM), ICSE'03, Portland, Oregon, USA pp. 63-68 (2003).
13. Griss M. L., Favaro J., d'Alessandro M., Integrating Features Modeling with the RSEB. 5th International Conference on Software Reuse, IEEE Computer Society (1998)
14. Hetrick, W.A., Krueger, C.W. and Moore, J.G. Incremental Return on Incremental Investment: Engenio's Transition to Software Product Line Practice. Conference on Object Oriented Programming Systems Languages and Applications, Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA), ACM pp. 798-804 (2006).
15. Jaring, M. and Bosch, J. Variability Dependencies in Product Family Engineering. 5th International Workshop on Product family Engineering (PFE), Springer-Verlag, LNCS 3014, pp. 81-97, (2004).
16. Kang K. C., Cohen S., Hess J. A., Novak W. E., Peterson A. S.. Featured-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-

21 ESD-90-TR-22, Software Engineering Institute, Carnegie Mellon University, Pittsburgh (1990).

17. Kang K. C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. FORM: A Feature-oriented Reuse Method with Domain Specific Software Architectures. Annals of Software Engineering, Vol 5(1) pp. 143-168, Springer (1998).
18. Kim, K., Kim, H., Ahn, M., Seo, M., Chang, Y. and Kang, K.C. ASADAL: A Tool System for Co-Development of Software and Test Environment based on Product Line Engineering. ICSE 2006, pp. 783-786, (2006).
19. Krueger, C. W. Software Mass Customization. BigLever Software Inc. Available at: http://www.biglever.com/extras/BigLeverMassCustomization.pdf, (2006).
20. Lee, K. and Kang, K.C. Feature Dependency Analysis for Product Line Component Design. 8th International Conference on Software Reuse (ICSR), Madrid, Springer-Verlag LNCS 3107, pp. 69-85, (2004).
21. Lee, Y., Yang, C., Zhu, C. and Zhao, W. An Approach to Managing Feature Dependencies for Product releasing in Software Product Lines. 9th International Conference on Software Reuse (ICSR), Turin, Italy, Springer-Verlag LNCS 4039, pp. 127-141, (2006).
22. Myllymäki, T. Variability Management in Software Product Lines. Technical Report 30. Institute of Software Systems, Tampere University of Technology (2002).
23. Robak, S. Feature Modeling Notations for System Families. International Workshop on Software Variability Management (SVM), ICSE'03, Portland, Oregon, USA pp. 58-62 (2003).
24. Shakari, P. and Møller-Pedersen, B. On the Implementation of a Tool for feature Modeling with a base Model Twist. Available at: http://www.himolde.no/nik06/articles/08-Shakari.pdf
25. Sinnema, M., Deelstra, S., Nijuis, J. And Bosch, J. COVAMOF: A Framework for Modeling Variability in Software Product Families. Procs of 3rd International Software Product Line Conference (SPLC), Springer-Verlag LNCS 3154, pp. 197-213, (2004).

26. Sinnema, M., de Graaf, O. and Bosch, J. Tool Support for COVAMOF. Procs of the 2nd Groningen Workshop on Software Variability Management (SVMG), Groningen, The Netherlands, (2004).

27. Sinnema, M., Deelstra, S., Nijuis, J. And Bosch, J. Modeling Dependencies in Product Families with COVAMOF. Procs of 13th International Workshop on Engineering of Computer Based Systems (ECBS'06), pp. 299-307 (2006).

28. Sinnema, M., Deelstra, S., Nijuis, J. and Hoekstra, P. The COVAMOF Derivation Process, 9th International Conference on Software Reuse (ICSR), Turin, Italy, Springer-Verlag LNCS 4039, pp. 101-114, (2006).

29. Svahnberg, M., van Gurp, J. and Bosch, J. A Taxonomy of Variability Realization Techniques. Software Practice & Experience, vol 35(8), 705-754, (2005).

30. Tolvanen, J-P., Kelly, S. Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences, Proceedings of the 9th International Software Product Line Conference, H. Obbink and K. Pohl (Eds.) Springer-Verlag, LNCS 3714, pp. 198 – 209, (2005).

31. Trujillo, S., Batory, D. and Díaz O. Feature Refactoring a Multi-Representation Application into a Product Line. Procs of 5th International Conference on Generative Programming and Component Engineering, pp. 191-200 (2006).

32. van Gurp, J., Bosch, J. and Svahnberg, M. Managing Variability in Software Product Lines. Procs of IEEE/IFIP Conference on Software Architecture, WICSA 2001, Amsterdam, The Netherlands, IEEE CS, pp. 45-54 (2001).

33. van Ommering, R. van der Linden, F., Kramer, J. and Magee, J. The Koala Component Model for Consumer Electronics Software. IEEE Computer, pp. 78-85, (2000).

34. Oldevik, J. Solberg, A., Haugen, Ø., -Pedersen, B. Evaluation framework for Model-Driven Product Line Engineering tools, In Software Product Lines Research Issues in Engineering and Management, Springer-Verlag, Timo Käköla and Juan Carlos Dueñas (Eds), pp. (2006).