HERRAMIENTA PARA ASISTIR LA EVOLUCIÓN DE SISTEMAS AUTO-ADAPTABLES EN EJECUCIÓN

Autor: Aarón Martín Bermejo
Directores: María Gómez Lacruz
Joan Fons i Cors

Máster Universitario en Ingeniería del Software, Métodos Formales y Sistemas de Información

13

ÍNDICE

ÍNDICE DE ILUSTRACIONES

1. INTRODUCCIÓN

Es una realidad que los sistemas software cada día están más presentes en el día a día y hacemos más uso de ellos. Sin embargo, estos sistemas no siempre se adaptan a las necesidades de los usuarios por las siguientes causas:

- Errores del sistema.
- Partes del sistema que fallan, no están disponibles o son externas al sistema y no puede controlarse su disponibilidad.
- Nuevas funcionalidades que demandan los usuarios y deben incluirse en el sistema.
- Actualizaciones del sistema costosas e incómodas para el usuario

Para satisfacer las necesidades de los usuarios, los sistemas han de cumplir unos requisitos de calidad: flexible, tolerante a fallos, configurable, seguro, sensible al contexto... Es por ello que cada vez se incrementa más el interés por sistemas capaces de adaptarse a distintas situaciones en función de su entorno, posibles fallos o incluso partes del sistema que estén temporalmente no disponibles.

Por ejemplo, móviles capaces de auto-silenciarse en determinadas situaciones, el sistema operativo que es capaz de actualizarse sin necesidad de reiniciarlo o un sistema de control crítico que necesite añadir o corregir una funcionalidad. Todos estos sistemas comparten una necesidad: ser capaces de adaptarse a la situación sin intervención humana. Una de las líneas más prometedoras en este sentido es la de los sistemas auto-adaptables [1]. Este concepto lo que quiere decir es que un sistema debería poder ser capaz de adaptarse a las posibles situaciones que se presenten con una intervención humana prácticamente nula.

1.1 PLANTEAMIENTO DEL PROBLEMA

La auto-adaptación tiene como objetivo principal obtener sistemas que sean capaces de adaptarse, sin intervención humana, a cambios en el propio sistema y el entorno de ejecución.

En el enfoque tradicional para construir sistemas auto-adaptables, los ingenieros identifican en tiempo de diseño los cambios que pueden aparecer en el entorno y en el sistema, y especifican las acciones que debe realizar el sistema para afrontar dichos cambios en ejecución. Sin embargo, en entornos altamente dinámicos, es imposible anticipar en tiempo de diseño todas las situaciones posibles a las que puede enfrentarse el sistema. Por ejemplo, nuevos usuarios, dispositivos o software, que no se han considerado en tiempo de diseño, pueden ser añadidos sin previo aviso, o una política de comportamiento del sistema puede aparecer o cambiar, pueden ocurrir fallos inesperados, nuevas necesidades de los usuarios, etc. A pesar de que los sistemas auto-adaptables han generado un considerable interés y han tenido un gran avance [2] [3], la mayor parte de las propuestas consideran un mundo cerrado, en el que todas las posibles situaciones han sido definidas a priori. Por tanto, es esencial dar

soporte a la evolución de sistemas auto-adaptables en ejecución, con el objetivo de manejar nuevos requisitos y no quedar obsoletos.

Tradicionalmente, la evolución de un sistema se realiza mediante una parada del mismo [4]. Este enfoque para sistemas auto-adaptables no es viable puesto que significa una pérdida de servicio durante la evolución. Además, es totalmente inasumible para sistemas críticos: gestión de infraestructuras, sistemas de los que dependan vidas, etc.
Modificar un sistema en tiempo de ejecución es complejo y propenso a errores [5] por dos motivos principales:

- En un sistema relativamente complejo muchos de sus módulos son dependientes en mayor o menor medida de otros. Modificar uno de estos componentes puede resultar catastrófico para el sistema si no se tienen en cuenta correctamente estas dependencias o si se desconocen. También existe la posibilidad de que se elimine un componente por accidente o porque no se conozcan las dependencias existentes con el mismo.

- Modificar un sistema en tiempo de ejecución no es una tarea sencilla ni recomendable sin unas garantías que aseguren que la modificación no afectará negativamente al funcionamiento del sistema.

En resumen, es necesario dar soporte a la modificación de sistemas auto-adaptables en tiempo de ejecución lo que hará posible que sea capaz de adaptarse a situaciones no previstas en tiempo de diseño y que se planteen en tiempo de ejecución. Esta tarea es compleja y por ello se necesitan técnicas para garantizar que la evolución es segura y sencilla.

1.2 SOLUCIÓN PROPUESTA

Modificar un sistema relativamente complejo alterando alguno de sus módulos o incluso eliminando alguno de ellos es una tarea compleja. El entendimiento humano no es capaz de captar toda la complejidad de un sistema de tamaño medio con detalle, por lo que no será capaz de determinar las posibles colisiones de las modificaciones realizadas sobre el sistema. Los ingenieros encargados de llevar a cabo esta tarea necesitan herramientas que proporcionen soporte a la evolución, tanto para reducir la complejidad y esfuerzo como para asegurar la estabilidad del sistema tras la evolución.

Para abordar la evolución de sistemas auto-adaptables esta tesis de máster propone una aproximación basada en técnicas de “Ingeniería Dirigida por Modelos”. Mediante el uso de esta filosofía se obtiene una representación abstracta del sistema (modelos) independiente de plataformas, lenguajes de programación e infraestructuras. Los modelos pueden tener un papel importante no únicamente como artefactos de primer orden en el desarrollo del sistema, si no también durante su ejecución. Esta iniciativa se conoce como “Modelos en Tiempo de Ejecución” (models@runtime) [6]. Los modelos capturan en todo momento las descripciones del sistema y su entorno necesarias para entender su situación actual, y las posibles necesidades de auto-adaptación.

El objetivo del trabajo de fin de máster es: desarrollar una herramienta para asistir al ingeniero a evolucionar un sistema software de forma sencilla y segura en tiempo de ejecución. Para abordar la evolución en tiempo de ejecución se ha utilizado el concepto de modelos en tiempo de ejecución. Gracias a que estos modelos los tendremos disponibles en tiempo de ejecución, tendremos una representación abstracta del sistema en tiempo de ejecución. Los modelos constituyen el conocimiento del sistema que permite razonar y tomar decisiones de forma autónoma. Este conocimiento podrá ser modificado para cumplir con las necesidades de evolución. Así la evolución se realiza a nivel de modelo (en vez de a nivel de implementación) permitiendo razonar sobre el sistema sin atender a detalles técnicos.

Para modificar el sistema se proponen "Patrones de evolución" que definen una serie de pasos que se aplican siempre que se realiza una modificación sobre el sistema. Estos patrones propagan cambios entre los distintos modelos que constituyen el conocimiento del sistema para garantizar que la evolución se ha realizado correctamente y de la forma más automática posible (con la menor intervención del ingeniero). Concretamente se proponen dos patrones, el patrón de "Añadir una nueva funcionalidad" y el de "Eliminar una funcionalidad", ya que ambos serán los que afecten de forma directa a la arquitectura del sistema.

En cuanto a la herramienta implementada nos basamos en MOSKitt4SPL (M4SPL) [7], una herramienta para el modelado de sistemas auto-adaptables. Se ha extendido esta herramienta para incluir los patrones de evolución en ella. Estos patrones se aplican mediante una serie de asistentes que guían al ingeniero paso por paso, requiriendo su intervención únicamente cuando sea necesario.

1.3 OBJETIVOS DE LA TESIS DE MÁSTER

El objetivo principal de esta tesis de máster es proponer mecanismos y herramientas de soporte para asistir a los ingenieros durante el proceso de evolución de los modelos de un sistema auto-adaptable. Este objetivo se descompone en los siguientes sub-objetivos:

1. Implementar patrones de evolución para guiar al ingeniero en el proceso de evolución del sistema. Concretamente, se han implementado los siguientes patrones:

    a. Eliminación de funcionalidades: al eliminar una funcionalidad, se detectan secciones que se deban y se puedan eliminar sin problemas del sistema. Además se detectan las dependencias que hay sobre ella para que el ingeniero decida fácilmente si hay que eliminar las funcionalidades dependientes.

    b. Adición de nuevas funcionalidades: al añadir una nueva funcionalidad al sistema, se asistirá al ingeniero para que decida si existen dependencias con esta nueva funcionalidad y/o si requiere funcionalidades adicionales (por ejemplo, la funcionalidad "Correo electrónico" necesita la funcionalidad "Acceso a internet").

2. Almacenar el conocimiento del sistema mediante el almacenamiento de distintas versiones de los modelos que lo representan.

3 Generar un plan de evolución entre distintas versiones del sistema para facilitar la tarea del ingeniero y exponer claramente los cambios necesarios a realizar entre distintas versiones de los modelos.

4 Validar la versión evolucionada del sistema aplicando una serie de refinamientos que aseguren su estabilidad y garanticen propiedades sobre el mismo.

5 Extender la herramienta de código libre MOSKitt4SPL para incorporar la funcionalidad implementada. Estas funcionalidades se añadirán en forma de asistentes que se lanzarán cuando sucedan los eventos anteriormente descritos (la eliminación o la adición de una funcionalidad).

6 Desarrollar un caso de estudio para validar la aplicación de la propuesta.

Al tener una herramienta gráfica que mediante asistentes guía al ingeniero ir paso por paso aplicando la evolución facilita enormemente una tarea que de otra forma debería ser manual y por tanto mucho más costosa y proclive a errores.

1.4 ESTRUCTURA DE LA TESIS DE MÁSTER

La estructura que se va a seguir en la tesis de máster será la siguiente:

2. **Contexto**: en este capítulo se presenta el contexto de la tesis. Es decir, se exponen los conceptos básicos sobre los que se basa y que será imprescindible conocer para comprenderla.

3. **Tecnologías:** en este punto se presentan las tecnologías utilizadas así como aquellas tecnologías que son la base de las utilizadas. Por ejemplo, M4SPL es la herramienta sobre la que se basa esta tesis y M4SPL a su vez se basa en Eclipse.

4. **Evolución del sistema:** en este capítulo se explica y se describe en detalle la aproximación propuesta para evolucionar sistemas auto-adaptables de forma automatizada y segura.

5. **Herramienta de soporte:** este capítulo constituye el núcleo de la tesis, ya que expone en detalle el desarrollo de la herramienta para dar soporte a la propuesta de evolución. Se presenta tanto la especificación de requisitos como la descripción detallada de la implementación de los mismos.

6. **Caso de estudio:** en este capítulo se expone un caso de estudio sobre el que se ha aplicado la propuesta de la tesis para demostrar su viabilidad práctica.

7. **Conclusiones y trabajo futuro:** en este capítulo se plantean las conclusiones extraídas de la tesis y se expone todo el trabajo que queda una vez finalizada.

2. CONTEXTO

En este capítulo se presentan los conceptos y disciplinas básicas sobre los que se basa la propuesta de esta tesis de máster y que son imprescindibles tanto para su desarrollo como para su comprensión.

2.1 COMPUTACIÓN AUTÓNOMA

El desarrollo y enorme avance de la tecnología en el campo de la informática en los últimos años es un hecho indudable. Es con toda seguridad el campo que más rápidamente haya avanzado en toda la historia de la humanidad.

Desde los primigenios sistemas "main frame" de los años 60 y 70 a la explosión de los ordenadores personales, el "boom" de Internet y, más recientemente, la proliferación de dispositivos móviles interconectados la informática ha avanzado sobremanera. Y no sólo en cuanto a sistemas de uso general para el público, sino por ejemplo en asumir la gestión de sistemas como la telefonía, la gestión de sistemas críticos como la gestión del agua, electricidad, gas, energía nuclear...[8].

Es evidente que la informática ha asumido un papel central en nuestra sociedad actual y sin la que sería inconcebible el mundo que nos rodea actualmente. Sin embargo, esta enorme proliferación de sistemas informáticos y su cada vez más creciente complejidad plantea un problema muy serio: ¿cómo vamos a gestionar una complejidad que crece sin final aparente y a un ritmo cada vez más veloz?

Como plantea Paul Horn en el manifiesto de la computación autónoma de IBM [9] sencillamente construyendo sistemas más complejos. Sin embargo y pese a la aparente contradicción esto simplemente significa que es necesario automatizar la monitorización de la complejidad de estos sistemas y la gestión de esta complejidad.

Paul Horn plantea que este tipo de sistemas ha de seguir el ejemplo del sistema nervioso autónomo humano: es capaz de decidir la velocidad de los latidos del corazón, comprobar el nivel de oxígeno en sangre, sudar si la temperatura aumenta demasiado… Todo ello sin realizar ningún tipo de esfuerzo consciente por nuestra parte, dedicando nuestros esfuerzos a la **toma de decisiones**.

Al inspirarse en el sistema nervioso autónomo, es de donde surge la denominación y el concepto de "computación autónoma", ya que se plantea a imagen y semejanza de este sistema del cuerpo humano. Debe ser:

- Invisible, de forma que el usuario sea totalmente inconsciente de su existencia o de su actuación.

- Conocer el estado del resto de sistemas gestionando una gran cantidad de información sobre ellos.

- Ser capaz de utilizar toda esta información de forma que mantenga todo el cuerpo humano gestionando eficientemente sus recursos en cualquier situación que se presente, sin que por ello la persona sea consciente de ello.

Evidentemente, el sistema nervioso autónomo (al igual que cualquier sistema humano) es de una complejidad extrema, así que es a esto a lo que se refiere con la frase "la solución a los sistemas complejos son sistemas más complejos".

Sin embargo, Horn plantea para lograr este objetivo con 8 características a cumplir por parte de los sistemas autónomos, para poder lograr la similitud con el sistema que los inspira:

1. Para ser autónomo un sistema ha de "**conocerse**", es decir, ha de tener un profundo conocimiento de sus componentes, su estado actual, los recursos disponibles… En caso contrario será imposible tomar decisiones sobre sí mismo.

2. Un sistema autónomo se **configura y reconfigura** bajo condiciones variables impredecibles.

3. Es capaz de **optimizarse** a sí mismo. Ya que tiene todos los datos de su funcionamiento debido a la característica uno, deberá ser capaz de utilizar esta información para realizar una gestión eficiente de sus recursos.

4. Puede "**auto-sanarse**" es decir, es capaz de recuperarse de una situación inesperada o de un mal funcionamiento de uno de sus componentes.

5. Puede "**auto-protegerse**", es decir, frente a ataques externos es capaz de reaccionar, defenderse y en caso necesario, recuperarse del ataque.

6. Además de conocerse a sí mismo, **conoce su entorno**. Debe conocer cuáles son los recursos externos que tiene disponibles y utilizarlos en consecuencia.

7. No puede existir en un **universo cerrado**. En un mundo cada vez más interconectado, un sistema cada vez tiene una mayor dependencia de otros para funcionar.

8. **Predecirá los recursos necesarios**. Una vez alcance cierto grado de conocimiento del uso de sí mismo, será capaz de predecir qué recursos serán necesarios en un futuro. Es decir, **asistirá a la toma de decisiones.**

Evidentemente, construir un sistema de estas características no es una tarea en absoluto trivial. A pesar de que se han realizado grandes avances desde el manifiesto de Horn y algunas de las características son mucho más factibles que entonces, aún quedan grandes retos por superar.

2.2 DESARROLLO DIRIGIDO POR MODELOS

El desarrollo dirigido por modelos propone la separación entre la especificación abstracta de un sistema mediante modelos de su implementación concreta como medio para el desarrollo de sistemas software.

Como definen Mellor, Clark y Futagami [10] un modelo es un conjunto de elementos formales que describen algo, construido con un propósito y que está relacionado con una forma de comunicación.

Un modelo se centra en aspectos concretos pudiendo ignorar el resto. Es decir, podremos modelar el motor de un coche ignorando aspectos como la transmisión o las ruedas. Se expresa en un lenguaje propio con un nivel de abstracción de forma que con muy poco se dice mucho.

Gracias a esta última característica, modelar un sistema será mucho más rápido y "sencillo" que implementarlo, ya que gracias a que modelar implica un alto nivel de abstracción no habrá que detenerse en los detalles de su implementación.

Como plantean también Mellor, Clark y Futagami [10], cualquier lenguaje de programación es en realidad un lenguaje de modelado. Esta afirmación puede resultar un tanto chocante, pero lo cierto es que un programa escrito en el lenguaje de modelado C ignorará tareas de la CPU o del compilador. Es decir, es una abstracción que nos permite expresar una especificación de una implementación concreta (código binario).

Así que, si un compilador nos permite convertir de la sintaxis de un lenguaje de programación, de un modelo, a un ejecutable que realmente será el producto, ¿qué nos impide convertir un modelo incluso más abstracto que el de un lenguaje, al producto definitivo?

Esto es lo que plantea el desarrollo dirigido por modelos, tener la capacidad de a partir de unos modelos abstractos, totalmente independientes de la implementación concreta, de plataformas y de lenguajes, ser capaces de obtener un producto software.

De hecho, es una evolución natural en la ingeniería del software ya que si durante estos años se ha conseguido avanzar de las fichas microperforadas a lenguajes de alto nivel como son los orientados a objetos, es lógico pensar que el siguiente paso sea incluso abandonar el código y desarrollar mediante el uso de modelos gráficos mucho más comprensibles para cualquier ser humano.

Un desarrollo de estas características nos proporciona unas grandes ventajas:

- **Independencia de plataformas**: desarrollar un sistema de forma independiente a una plataforma ya es un avance realmente considerable, ya que conseguimos reducir el tiempo de desarrollo para adaptarlo a distintas plataformas o adaptarlo a las que surjan.

- **Reutilización del conocimiento:** cuando un sistema se implementa por parte de un desarrollador, es complicado adaptar su desarrollo a lo largo del tiempo a las nuevas necesidades que vayan surgiendo. Sin embargo, utilizando modelos abstractos se simplifica esta tarea.

- **Mejor comprensión del sistema:** es evidente que un conjunto de modelos gráficos que representen el sistema será mucho más comprensible que el código que lo implemente

Este proceso está claro que no es en absoluto trivial, por supuesto. Como plantean Pedro Valderas y Vicente Pelechano [11] el proceso del desarrollo dirigido por modelos:

- Suele comenzar por definir un modelo de requisitos funcionales del usuario que defina todas las necesidades del mismo, sin tener en cuenta más que sus necesidades.

- Este modelo de requisitos se refina en varios modelos que especifican el sistema independientemente de los aspectos tecnológicos para realizarlo.

- Una vez obtenidos los modelos que especifican el sistema, serán convertidos directamente a código o a metamodelos que se utilizarán posteriormente para la generación del código.

A pesar de que pueda parecer que el paso de la generación del código sea el más complejo, en el estado actual del desarrollo dirigido por modelos este paso se ha superado por completo.

Gracias a ello podemos encontrar una gran cantidad de herramientas MDA (Model Driven Architecture – Arquitectura dirigida por modelos) que dan soporte al desarrollo dirigido por modelos y en las que la generación del código es un paso totalmente transparente al usuario.

Algunos ejemplos de ello pueden ser Enterprise Architect [12] o sin ir más lejos, M4SPL, la herramienta que se explicará en el capítulo y sobre la que se basa toda la propuesta de esta tesis de máster.

2.2.1 MODELOS EN TIEMPO DE EJECUCIÓN

Uno de los requisitos de los sistemas auto-adaptables es que sean capaces de adaptarse a cualquier situación que se presente. Sin embargo, es obvio que un ingeniero es incapaz de predecir cualquier situación posible que pueda plantearse al sistema que está diseñando. Mucho menos si utilizamos el paradigma de desarrollo que se aplica al desarrollo dirigido por modelos, es decir, especificación de requisitos, diseño e implementación (a partir de los modelos).

No por ello es un paradigma erróneo. Sin embargo, en el contexto de los sistemas auto-adaptables y teniendo en cuenta que es imposible definir todas las situaciones posibles en tiempo de diseño.

Además en el contexto de sistemas críticos, como pueden ser sistemas de control de infraestructuras o de navegación no se puede asumir una parada del sistema en caso de

detectar una situación ante la que el sistema deba adaptarse y no haya sido contemplada. O incluso un error en el sistema que deba ser corregido.

Habrá que realizar nuevas propuestas para sortear este obstáculo y poder utilizar el desarrollo dirigido por modelos junto con todas las ventajas que proporciona y una de las propuestas más prometedoras para sortearlo es la de los modelos en tiempo de ejecución (models@runtime).

Esta propuesta plantea que, dado que los modelos son en realidad el propio sistema, podremos tenerlos en tiempo de ejecución y en caso de que sea necesario, modificarlos.

Por ejemplo, en caso de que surja un nuevo requisito que el sistema deba cumplir, podríamos modificar los modelos para añadirlo, generar el código y con ello modificar el sistema en tiempo de ejecución para que lo cumpla.

Sin embargo, un sistema auto-adaptable tiene una complejidad excesivamente grande para ser abarcada en detalle por el entendimiento de un ser humano. Sin una comprensión en detalle, modificar el sistema es una acción considerablemente arriesgada porque desconocemos los efectos que pueda provocar una modificación de un componente en el conjunto del sistema.

Es por ello que es necesaria una herramienta que asegure que una modificación de los modelos durante el tiempo de ejecución sea completamente segura y deje el sistema en un estado estable, sin colisiones indirectas. De ahí esta propuesta de tesis de máster.

2.3 LÍNEAS DE PRODUCTO SOFTWARE

Las líneas de producto software fueron definidas por el Instituto de Ingeniería Software Carnegie Mellon [13] como un conjunto de sistemas software que comparten un conjunto de características comunes que satisfacen las necesidades de un segmento particular del mercado y que son desarrollados a partir de una serie de recursos principales básicos de una forma determinada.

Esta definición se puede dividir en los conceptos que la componen:

- **Producto**: el producto que las líneas de producto software buscan obtener cambia radicalmente el enfoque del desarrollo software. El objetivo es construir muchas aplicaciones, en lugar de una sola como ocurre en el desarrollo tradicional.

- **Características**: Las características comunes son las unidades (es decir, incrementos de funcionalidad de la aplicación) mediante las cuales se pueden construir productos distintos y definidos en una línea de producto software.

- **Dominio:** una línea de producto software se crea dentro del ámbito de aplicación de un dominio (de un segmento particular del mercado). Un dominio es un ámbito especializado de conocimiento, un área de especialización o un contexto cuyas funcionalidades están fuertemente relacionadas.

- **Elementos básicos**: los elementos básicos son aquellos que forma parte del núcleo de una línea de producto y que son utilizados en la creación de varios productos de la misma.

- **Plan de Producción:** Indica cómo ha sido producido cada producto. El plan de producción constituye una descripción de cómo los elementos básicos se van a utilizar para desarrollar un producto en una línea de productos y especifica cómo utilizar el plan de producción para construir el producto final [14].

Por tanto, el desarrollo de línea de producto software se referirá a todos los métodos, herramientas y técnicas ingenieriles dirigidas a la creación de productos software similares usando una serie de componentes comunes reutilizables.

2.3.1 LÍNEAS DE PRODUCTO SOFTWARE DINÁMICAS

Las líneas de producto software dinámicas son líneas de producto que soportan la variabilidad del sistema para adaptarse a cambios de requisitos que ocurran en tiempo de ejecución [15].

Las líneas de producto software dinámicas son un campo emergente y con una gran demanda cuyo objetivo es el de sistematizar el espacio de configuración de una línea de producto en un sistema software adaptable dinámicamente.

Para ello se gestiona la reconfiguración dinámica del sistema modelando las configuraciones explícitamente en una línea de producto tradicional y adaptándola pata utilizar el concepto de "variabilidad tardía" [16].

En una línea de producto tenemos características que serán los componentes centrales de un producto. En cambio, en una línea de producto software dinámica tendremos un conjunto de "características dinámicas", es decir, características que serán activadas o desactivadas en tiempo de ejecución.

En esta propuesta se utilizan modelos de configuración que indican qué características están activas en un determinado momento de tiempo. Además, se utilizan modelos de resoluciones en los que cada una de las resoluciones indicará, en función de una condición, qué características deberán desactivarse o activarse.

2.4 CONCEPTOS BÁSICOS

En esta sección se explican conceptos básicos que se utilizan a lo largo de la tesis de máster y que son necesarios para comprender la propuesta.

2.4.1 MODELADO DE SISTEMAS AUTOADAPTABLES

A continuación se exponen los distintos tipos de modelos que se utilizan en la propuesta para modelar un sistema auto-adaptable. Estos modelos constituirán el conocimiento del sistema.

2.4.2 MODELO DE CARACTERÍSTICAS

En el desarrollo software y en el contexto de la línea de producto software, un Modelo de Características es una representación de cada uno de los productos de una línea de producto software en términos de "características" [17].

Una línea de producto software es una familia de programas relacionados. Cuando las unidades de construcción del programa son características (incrementos en el desarrollo del programa o funcionalidades), todo programa en la línea de producto se identifica por su combinación única de características.

Por tanto un Modelo de Características agrupará un conjunto de características y constituirá uno de los componentes de un programa. Habitualmente se utiliza para generar otros artefactos como el Modelo de Arquitectura o piezas de código.

Las características dentro del modelo están conectadas en una estructura jerárquica de árbol mediante las siguientes relaciones de variabilidad:

- **Opcional**: indica que la activación del padre no implica la activación de la característica hija. La Ilustración 1 muestra un ejemplo de relación opcional.

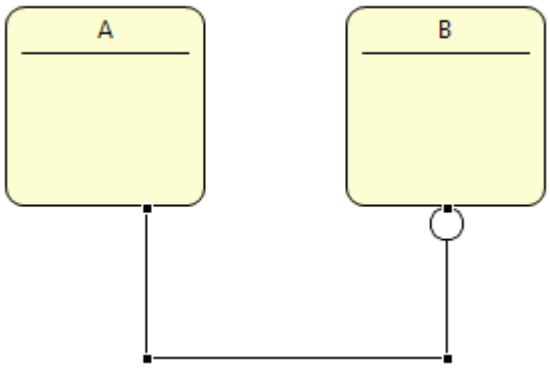

ILUSTRACIÓN 1 RELACIÓN OPCIONAL

- **Obligatoria:** indica que la activación de la característica padre implica la activación de la hija. La Ilustración 2 presenta un ejemplo de relación obligatoria entre dos características A y B.

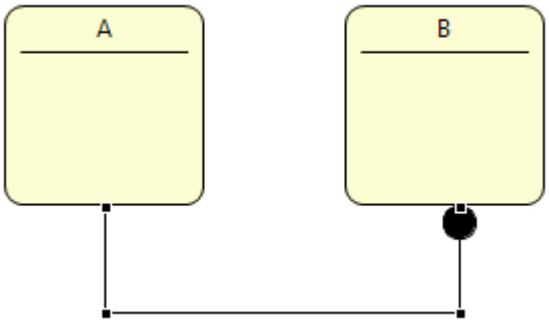

ILUSTRACIÓN 2 RELACIÓN OBLIGATORIA

- **OR**: indica que la activación de la característica padre activa una o más de las hijas (ver Ilustración 3):

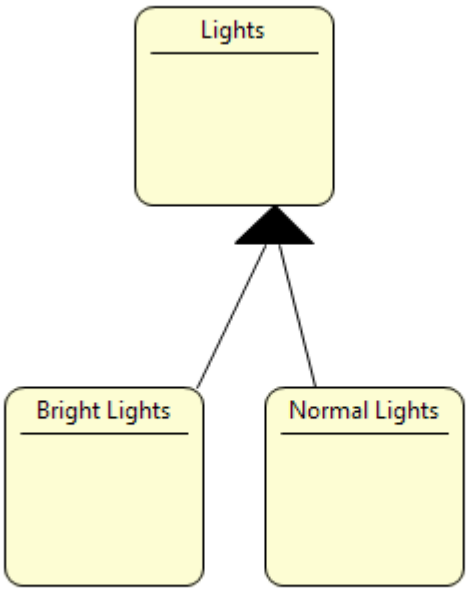

ILUSTRACIÓN 3 RELACIÓN OR

- **Alternativa** (XOR): sólo una característica hija podrá ser activada al activarse la padre (ver Ilustración 4)

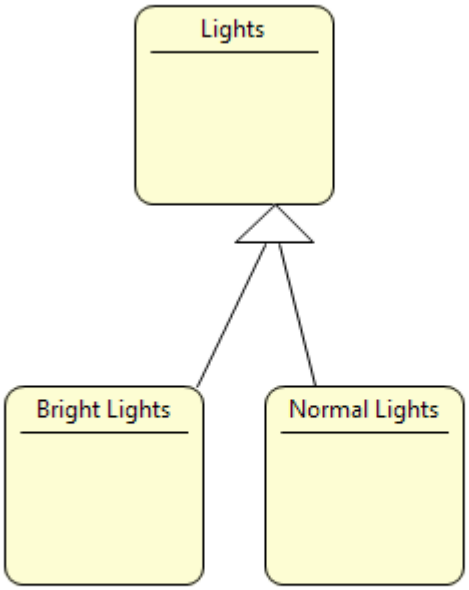

ILUSTRACIÓN 4 RELACIÓN ALTERNATIVA

Además de las relaciones jerárquicas, las características también pueden tener relaciones sin seguir ningún tipo de jerarquía:

- **Requiere**: indica que hasta que las características requeridas no sean activadas, no podrá ser activada la que las requiere.

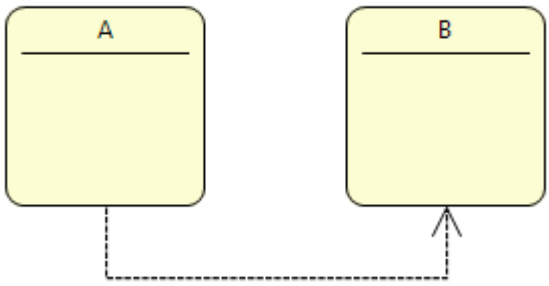

ILUSTRACIÓN 5 RELACIÓN REQUIERE

La Ilustración 5 muestra que la característica A no podrá ser activada hasta que se active la B.

- **Excluye**: cuando una de las dos características sea activada, excluirá la activación de la otra (ver Ilustración 6).

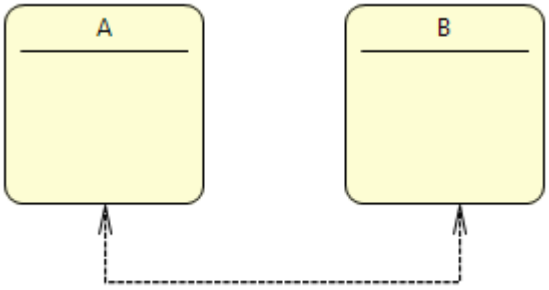

ILUSTRACIÓN 6 RELACIÓN EXCLUYE

La Ilustración 6 muestra que si la característica A se activa, no podrá hacerlo la B y viceversa.

Con esta última relación se muestran todas las relaciones posibles del Modelo de Características.

En la Ilustración 7 Modelo de Características Teléfono se muestra un ejemplo de Modelo de Características:

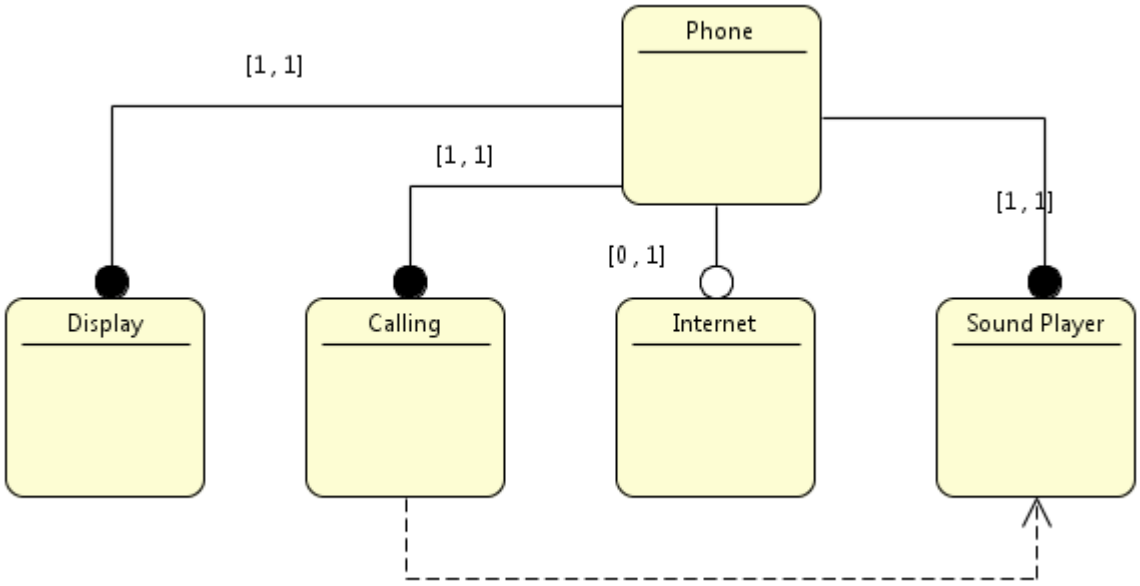

ILUSTRACIÓN 7 MODELO DE CARACTERÍSTICAS TELÉFONO

En la Ilustración 7 se describen las características de un teléfono móvil. Como se puede ver, el móvil es obligatorio que tenga:

- Un Display (generalmente, una pantalla)
- Ha de permitir llamar
- Ha de tener un reproductor de sonido (generalmente un altavoz)

Como características opcionales, puede tener acceso a Internet. Además, para que el usuario del teléfono pueda llamar deberá escuchar la llamada, con lo que la característica "Calling" requiere de la característica "Sound Player" para poder ser activada.

Es un Modelo de Características sencillo pero que ilustra las relaciones entre características y como se especifican las mismas.

2.4.3 MODELO DE CONFIGURACIÓN

Una configuración define el estado del sistema en un instante de tiempo. Cada configuración contendrá una característica y un estado de la misma.

El Modelo de Configuración tiene una referencia al Modelo de Características y contiene un conjunto de elementos "FeatureState". Cada "FeatureState" referencia una característica y contiene un atributo "state", que indica el estado de la característica. Hay dos posibles estados: Activa e Inactiva.

A continuación en la Ilustración 8 se expone un ejemplo de Modelo de Configuración:

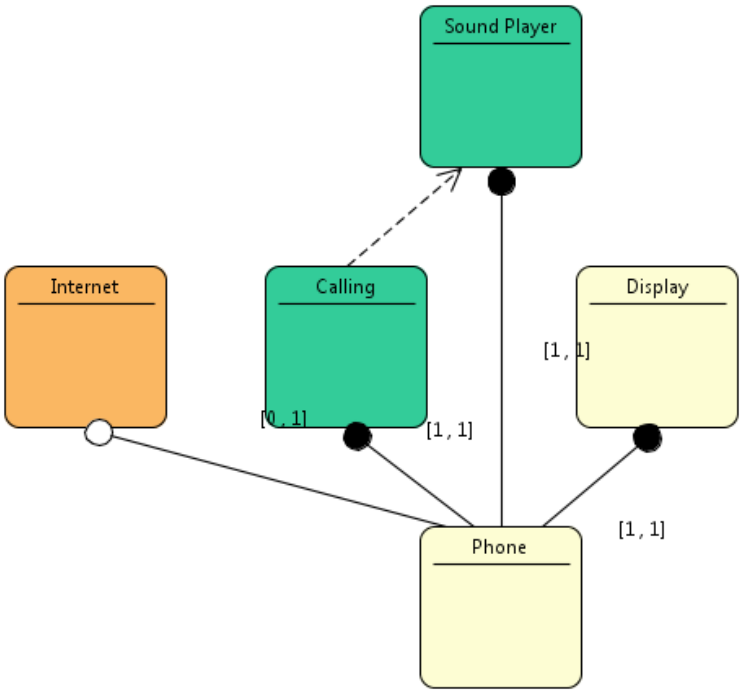

ILUSTRACIÓN 8 MODELO DE CONFIGURACIÓN MÓVIL

En la Ilustración 8 se muestra el Modelo de Configuración en base al Modelo de Características de la Ilustración 7. En este Modelo de Configuraciones se especifican como activas las características coloreadas de verde, es decir, "Calling" y "Sound Player". Las características desactivadas serán las coloreadas de rojo, concretamente la característica "Internet".

2.4.3 MODELO DE RESOLUCIÓN

Un modelo de resolución especifica los cambios necesarios que han de realizarse en la configuración de un conjunto de características ante un cambio de contexto. Se especifican reconfiguraciones de forma declarativa sobre las configuraciones del sistema.

Así, una resolución representará el conjunto de cambios a realizar sobre las características. Cada resolución contiene una lista de pares característica-estado y una condición que indicarán el estado al que debe modificarse una característica en caso de cumplirse la condición.

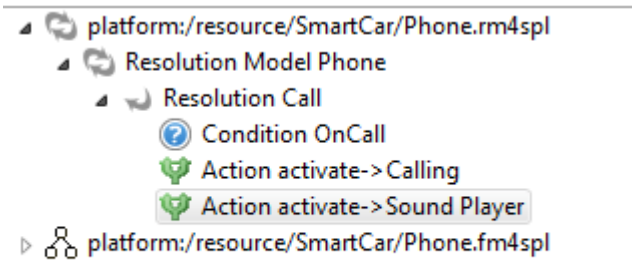

ILUSTRACIÓN 9 MODELO DE RESOLUCIÓN MÓVIL

En la Ilustración 9 se muestra un Modelo de Resoluciones con una sola resolución. Esta resolución llamada "Call" activa las características "Calling" y "Sound Player" en caso de que la condición "OnCall" se cumpla. Es decir, en caso de que haya una llamada, las características de llamar y de reproductor de sonido son activadas para posibilitarlo.

2.4.4 MODELO DE ARQUITECTURA

El Modelo de Arquitectura consiste en una especificación de todos los componentes software y hardware que realizarán las características especificadas en el Modelo de Características. En esta tesis de máster se ha utilizado PervML [18] para modelar la arquitectura del sistema.

PervML es un lenguaje específico de dominio desarrollado en el grupo de investigación PROs [19] para el desarrollo de sistemas sensibles al contexto [20]. Provee de un conjunto de primitivas conceptuales que permiten describir un sistema independientemente la tecnología que lo implemente. Además, cubre el proceso de desarrollo completo de un sistema sensible al contexto definiendo un método de desarrollo y proporcionando las herramientas de soporte para ello.

En esta propuesta para los modelos de arquitectura se utilizan 3 tipos de elementos arquitectónicos distintos:

- Servicios. Representan servicios del sistema.
- Componentes. Representan componentes del sistema.
- Canales. Representan canales de comunicación entre servicios y componentes.

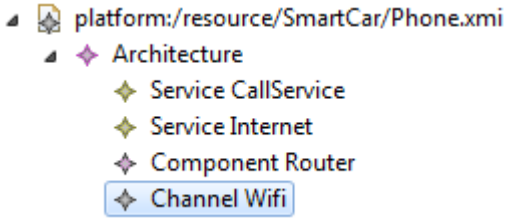

ILUSTRACIÓN 10 MODELO DE ARQUITECTURA MÓVIL

En la Ilustración 10 se especifica un Modelo de Arquitectura sencillo que contiene:

- El servicio “CallService”
- El servicio “Internet” para dar acceso a internet
- El componente “Router” que será el componente que posibilite el acceso a internet
- El canal “Wifi” que consumirá el servicio “Internet” a través del componente “Router”.

Así quedarán especificados los componentes de la arquitectura del móvil.

2.4.5 MODELO DE ENLACE

Un modelo de enlace o “Weaving Model” es un modelo que sirve para relacionar elementos de dos modelos. Los modelos originales permanecen sin modificaciones, mientras que las relaciones entre los elementos de los modelos se almacenan en un modelo aparte.

En esta propuesta se utilizará el modelo de enlace o “Weaving Model” para enlazar las características del Modelo de Características con los elementos arquitectónicos concretos que las realizan.

Para ello en primer lugar, enlazar un Modelo de Características con un Modelo de Arquitectura. Posteriormente se especifican los enlaces entre cada una de las características con los correspondientes elementos arquitectónicos que realicen cada una de ellas.

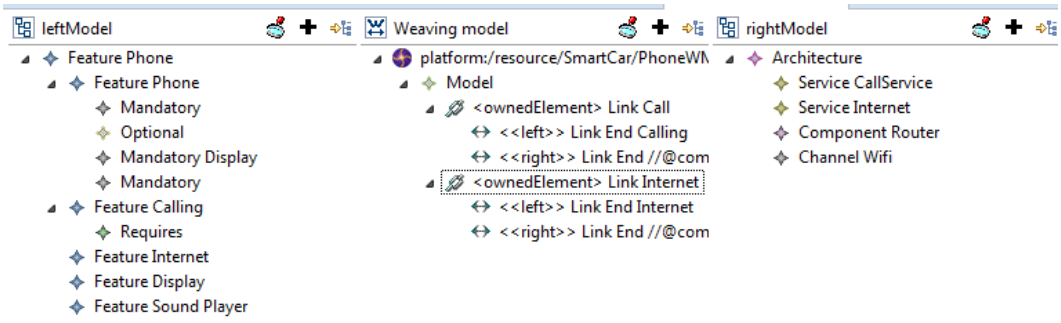

ILUSTRACIÓN 11 MODELO DE ENLACE MÓVIL

En la Ilustración 11 se muestra un ejemplo de modelo de enlace en el que se especifica que:

- La característica “Calling” será realizada por el servicio “CallService”
- La característica “Internet” será realizada por el servicio “Internet”

De esta forma es como se enlazan las características con los componentes de la arquitectura para especificar qué elementos de la misma realizan cada una de las características.

2.4.5 MODELO DE ESPACIO DE ADAPTACIÓN

La ejecución de una Línea de Productos Software Dinámica puede ser abstraída como una máquina de estados fuertemente conectada.

Cada estado se denomina “adaptación” y contiene una configuración y una lista de resoluciones. Aplicar cada una de las resoluciones sobre la configuración (que contiene el estado de las características del sistema) generará nuevas adaptaciones que estarán representadas por la configuración que se genera al aplicar la resolución.

El modelo de espacio de adaptación es un conjunto de adaptaciones, que representa todas las combinaciones posibles de configuraciones que se dan en el sistema. A continuación se muestra un ejemplo:

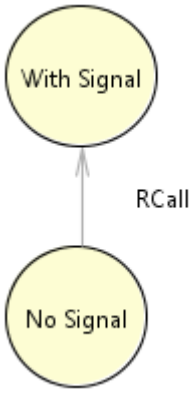

ILUSTRACIÓN 12 ESPACIO DE ADAPTACIÓN DE MÓVIL

En la Ilustración 12 se observa un ejemplo de espacio de adaptación donde el sistema se reconfigura mediante la aplicación de la resolución “Call” (que activa las características de “Calling” y “Sound player” como se ve en la Ilustración 9) pasando del estado “No Signal” a “With Signal”.

3. TECNOLOGÍAS

A lo largo de este capítulo se muestran las tecnologías que se han utilizado en el desarrollo e implementación de esta propuesta.

3.1 ECLIPSE

Eclipse es una entorno de desarrollo integrado (IDE) de código abierto multiplataforma que puede ser utilizado para crear aplicaciones como sitios web, programas java embebidos o programas C++.
Está disponible para su descarga gratuita y para el desarrollo de la propuesta se ha utilizado Eclipse Juno (4.2) [21].

Una característica que proporciona Eclipse y que será muy importante en el desarrollo de la propuesta son los asistentes.

Eclipse está muy orientado hacia su extensibilidad, de forma que proporciona la posibilidad de crear nuevos asistentes para automatizar un proceso seguido de pasos. En capítulos posteriores se expondrán ejemplos del uso de los asistentes y de su importancia en el desarrollo de la herramienta.

3.2 EMF

EMF o "Eclipse Modelling Framework" [22] [23] es un framework para construir herramientas y aplicaciones basadas en modelos de datos. Basándose en una especificación de modelo XMI [24] proporciona herramientas para la generación de clases java para el modelo, además de un conjunto de clases adaptadoras para habilitar la visualización, edición por comandos y un editor básico del modelo.

Los modelos pueden ser especificados utilizando java con anotaciones, documentos XML o herramientas de modelado como Rational Rose (importando a EMF). Además, EMF provee de la plataforma para la interoperabilidad de otras herramientas y aplicaciones basadas en EMF.

3.3.1 EMF ADAPTERS

Además de estas características de EMF cabe destacar una característica que será muy importante para la propuesta: los "Adapters". Un "adapter" consistirá en un escuchador [25] para captar todas las modificaciones respecto al objeto escuchado.

Esta característica que proporciona EMF será tremendamente útil para detectar cualquier tipo de modificación en el sistema y generar una respuesta automatizada al respecto. De hecho, será una de las tecnologías básicas para la propuesta.

3.3.2 EMF COMPARE

EMF además provee de un framework de comparación de cualquier tipo de modelo válido bajo el esquema XMI llamado EMF Compare [26]. Este framework también proporciona capacidades de fusión de modelos, sin embargo estas capacidades no son necesarias en nuestro enfoque.

Para la comparación de modelos proporciona dos servicios:

- **Matching:** itera sobre los dos modelos lógicos cargados de cada elemento para mapear los elementos dos a dos. Es decir, determina que el "Elemento1" del primer modelo corresponde al "Elemento1" del segundo.

- **Diferencing:** la fase de diferenciación recorre los elementos obtenidos en la fase de "matching" para determinar qué elementos son iguales y cuáles son diferentes.

Mediante el uso de estos dos servicios se podrán extraer todas las diferencias entre dos modelos. Esta característica será vital en el punto **Error! Reference source not found.** que se expondrá más adelante.

3.3.3 ATLAS MODEL WEAVER

La tecnología Atlas Model Weaver (AMW) [27] está desarrollada por el proyecto Generative Modeling Technologies (GMT – Tecnologías de Modelado Generativas) [28]. Este proyecto tiene como objetivo producir prototipos dentro del área del Desarrollo Dirigido por Modelos

AMW es una herramienta para establecer enlaces entre distintos modelos y que estos enlaces sean almacenados en un modelo llamado "Weaving Model".

En esta propuesta se ha utilizado la tecnología AMW para establecer enlaces entre las características del Modelo de Características y los elementos arquitectónicos que las realizarán.

3.3 MOSKITT4SPL

MOSKitt4SPL (M4SPL) es una herramienta libre y abierta que da soporte al modelado de líneas de producto software (SPL).

MOSKitt4SPL está basado en MOSKitt [29] y construido en Eclipse. Ha sido desarrollado por el Centro de Investigación en Métodos de Producción de Software (ProS) de la Universitat Politécnica de València (UPV). Ha sido construido utilizando las tecnologías de las *Eclipse Modelling Tools* como EMF, GMF o ATL.

Las características más importantes incluidas en MOSKitt4SPL son:

- **Modelado de Líneas de Producto Software:** las características fundamentales que nos ofrece MOSKitt4SPL para el modelado de líneas de producto software son:

    - **Editor de Modelos de Características**. Los *Feature Models* (FM) o Modelos de Características son una notación ampliamente utilizada para describir el conjunto de productos que se pueden derivar de una línea y para capturar la variabilidad que puede existir entre todos los productos.

    - **Felixibilidad en la notación**. La notación gráfica utilizada para representar los elementos puede ser cambiada de manera dinámica.

    - **Representación en árbol automática**. MOSKitt4SPL puede reorganizar la representación gráfica de un FM para ser mostrado en forma de árbol. Donde los elementos del nivel más bajo sean las hojas, el elemento más alto la raíz y el resto de elementos se encuentren ordenados por niveles.

    - **Configurador del Modelo de Características**. Las configuraciones de una SPL se representan como una selección de unas características del FM que constituyen la SPL. Cada configuración se representa como un conjunto de estados de las características. Los posibles estados definidos en MOSKitt4SPL son Activo (color verde), Inactivo (color rojo) y Descartado (color naranja).

- **Modelado de Líneas de Producto Software Dinámicas:** recientemente ha crecido la tendencia de posponer la configuración de los productos software hasta el tiempo de ejecución. Las DSPL o Líneas de Producto Software Dinámicas reconocen estos cambios, permitiendo a los sistemas adaptarse a las situaciones cambiantes del entorno en tiempo de ejecución.

  Las características fundamentales que MOSKitt4SPL ofrece para el modelado de líneas de producto software dinámicas son:

    - **Editor del Modelo de Resoluciones.** Las resoluciones definen el cambio de estado de un conjunto de características cuando ocurre un evento en el entorno.

    - **Generación de Espacio de Adaptación.** La ejecución de una DSPL puede ser abstraída como una máquina de estados altamente conectada donde los estados

representan las posibles configuraciones del sistema, mientras que las transiciones, representan rutas de migración. MOSKitt4SPL genera automáticamente el espacio de adaptación implícito (en forma de máquina de estados) a partir de la especificación de una DSPL.

La implementación realizada en este trabajo se ha realizado como una extensión de la herramienta M4SPL.

4. EVOLUCIÓN DEL SISTEMA

En este capítulo se presenta en profundidad la propuesta para evolucionar sistemas auto-adaptables, entrando en detalles de todos los objetivos que se han planteado en puntos previos.

4.1 VISIÓN GENERAL DE LA PROPUESTA

El tamaño y complejidad de los sistemas auto-adaptables va más allá de la habilidad de un ser humano para entenderlos en detalle. Además, modificar un sistema auto-adaptable es un procedimiento crítico en el que entender todos los posibles impactos de los cambios es una tarea muy complicada. Por tanto, será necesario asistir a los ingenieros en esta tarea.

El principal objetivo de la propuesta es asistir al ingeniero en la modificación de un sistema auto-adaptable en ejecución. Para ello se proporcionan mecanismos y herramientas para asistir al ingeniero y minimizar los posibles errores, reducir el esfuerzo necesario para evolucionar un sistema y que además esta evolución no introduzca inconsistencias o comportamientos indeseados.

Para ello, se toma como punto de partida el conocimiento del sistema representado por un conjunto de modelos. A partir de este conocimiento se propone un proceso de evolución formado por los siguientes pasos:

- Cuando se modifique un modelo del sistema, se realizará una **copia del conocimiento** del sistema para almacenar una versión del sistema. Esta versión constituye una copia de seguridad para utilizar posteriormente tras las modificaciones que se vayan haciendo del mismo y poder tener una trazabilidad de las mismas.

- Una vez tengamos la copia del conocimiento del sistema y por tanto una copia de seguridad, se podrá evolucionar el sistema. Esta evolución puede realizarse añadiendo o eliminando funcionalidades y para cada uno de los casos se aplicará un **patrón de evolución.** Un patrón de evolución define las acciones a realizar cuando se produce una modificación sobre el sistema. Es decir, un algoritmo a aplicar automatizado todo lo posible para que la intervención del ingeniero sea mínima

- Una vez tengamos una copia del conocimiento del sistema, estaremos en posición de comparar el sistema en la versión actual con una de las versiones previas y generar un **plan de evolución**. Este plan resumirá las modificaciones que se han realizado al sistema de forma que sean claras y queden explícitas para el ingeniero.

- Tras la obtención del **plan de evolución** se **validará** el sistema, para garantizar que no se ha llegado a un estado inestable que pueda llevar a fallos en el mismo.

- Finalmente tras validar el sistema, se podrán aplicar una serie de **refinamientos** para garantizar propiedades en el sistema, como puede ser que no se realicen reconfiguraciones en bucle, sin reversibilidad, etc.

Este proceso de evolución se resume en la Ilustración 13:

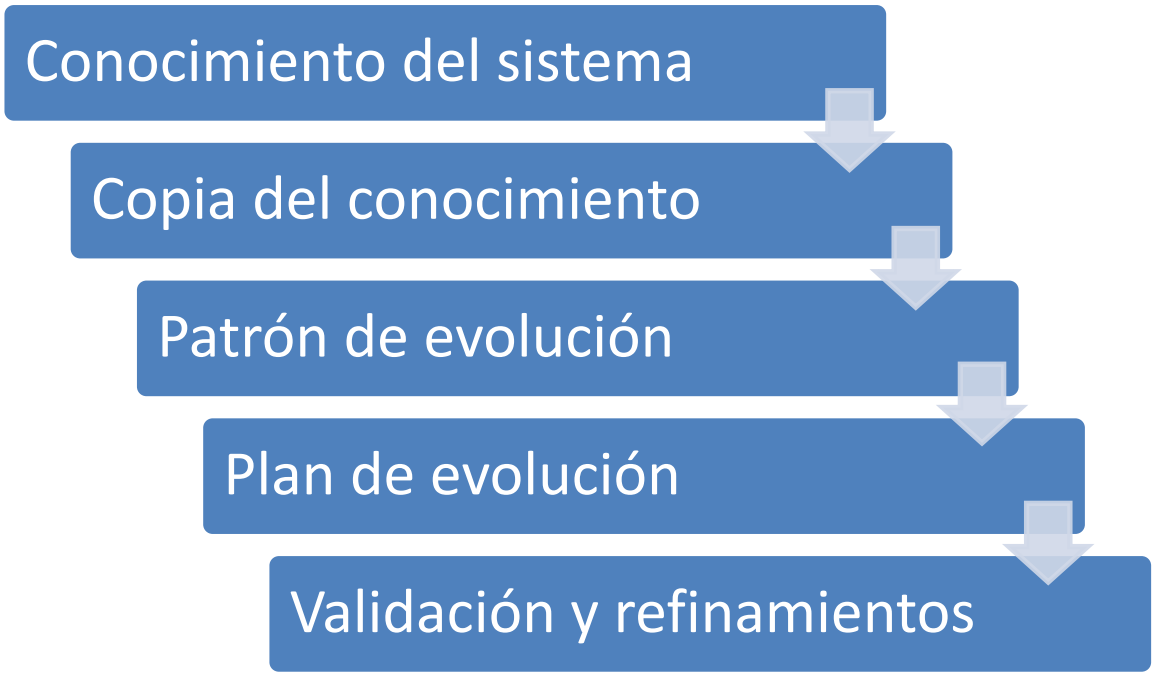

ILUSTRACIÓN 13. RESUMEN DEL PROCESO DE EVOLUCIÓN

Siguiendo este proceso garantizamos una modificación del sistema sencilla, clara y segura, con la menor intervención humana posible. Además siempre tendremos copias del conocimiento del sistema en caso de que sea necesario recuperarlas y planes de evolución que detallarán las modificaciones realizadas al sistema.

4.2 ALMACENAR EL CONOCIMIENTO DEL SISTEMA

Previamente a la modificación del sistema, se crea una copia del conocimiento actual del sistema, es decir, una copia de todos los modelos que describan el sistema. De esta forma se tendrá una copia de seguridad del mismo en caso de que las modificaciones llevasen al sistema a un estado inestable o incluso de mal funcionamiento.

Además de este uso, otro de los objetivos derivados de crear una copia del conocimiento del sistema es el que se explicará a continuación, el de generar un plan de evolución.

4.3 PATRONES DE EVOLUCIÓN

Para evolucionar un sistema auto-adaptable, el ingeniero ha de actualizar los modelos del conocimiento del sistema. Cuando se realice un cambio sobre cualquier modelo, los modelos relacionados deberán ser actualizados también.

Debido a que un sistema hoy en día puede estar compuesto de cientos de servicios, políticas de adaptación y componentes actualizarlos todos de forma manual es una tarea muy propensa a errores, además de implicar un gran esfuerzo por parte del ingeniero.

Para ello se han implementado los patrones de evolución, para asistir y guiar al ingeniero durante el proceso de evolución. Estos patrones definen las acciones a realizar cuando se produce una modificación sobre el sistema. Consisten en un algoritmo a aplicar que se ha automatizado todo lo posible en su implementación para que la intervención del ingeniero sea mínima. De esta forma se evitan errores humanos.

Concretamente, se han implementado dos patrones de evolución para asistir durante (1) la eliminación de funcionalidad y (2) la adición de nueva funcionalidad al sistema. La modificación de funcionalidad del sistema es considerada como una eliminación seguida de una adición.

Además se proporciona una herramienta con asistentes y editores gráficos para guiar a los ingenieros en la evolución del sistema auto-adaptable y aplicar los patrones de evolución, de forma que la evolución del sistema sea sencilla y la propagación de los cambios ante una modificación de un modelo sea segura.

4.3.1 PATRÓN 1: ELIMINACIÓN DE FUNCIONALIDAD

Una acción bastante común es la de eliminar una característica del sistema, ya sea porque los requisitos cambien y ya no sea necesaria, porque haya una nueva versión, porque sea una característica que esté fallando o sea problemática...

La evolución de la funcionalidad del sistema implica la evolución del modelo de variabilidad que representa las funcionalidades disponibles en el sistema. En nuestro enfoque, el ingeniero modifica el Modelo de Características eliminando la característica que representa dicha funcionalidad. Cualquier cambio que se realice en el Modelo de Características, como es en este caso la eliminación de una de ellas, deberá propagarse a los modelos relacionados: Modelo de Resoluciones y Modelo de Arquitectura.

Debido a ello, se propone un patrón de evolución que se ejecutará cada vez que se elimine una característica del sistema. El patrón consiste básicamente en un algoritmo a seguir de forma que se simplifique el proceso de eliminación de una característica y la propagación de este cambio a todos los modelos del conocimiento del sistema.

El patrón de evolución consiste en los siguientes pasos:

1. **Eliminar toda resolución que active o desactive la característica eliminada:** si una resolución está realizando una acción sobre una característica (ya sea desactivar o activar) que ha sido eliminada, la resolución debe ser eliminada también. Si no fuera eliminada se produciría un fallo en ejecución cuando se intentara aplicar la resolución sobre una característica que no existe.

2. Buscar todos los **elementos arquitectónicos relacionados con la característica eliminada**

    a. Preguntar al ingeniero si deben **eliminarse los elementos relacionados o hay que mantenerlos**: la modificación del Modelo de Características también tendrá efecto en la arquitectura. Sin embargo, es posible que un elemento arquitectónico sea necesario para dar soporte a otra característica. Incluso aunque no sea utilizado, quizás sea necesario mantener el elemento de la arquitectura porque se prevé su uso futuro. Es por ello que será necesaria la **participación del ingeniero** para decidir la eliminación del elemento de la arquitectura**.**

3. **Validar el sistema:** para garantizar que el sistema sigue siendo seguro, se lanzará un proceso de validación del mismo una vez realizadas las modificaciones.

A continuación en la Ilustración 14 Patrón de eliminación de funcionalidad se expone el pseudocódigo del patrón de eliminación de funcionalidad:

Algoritmo 1: Eliminación de funcionalidad

Requiere: Una Característica (C) se ha eliminado del *Modelo de Características*

```
1:  Para toda Resolución (R) ∈ Modelo de Resolución hacer
2:      si R activa/desactiva F entonces
3:          Eliminar R
4:      fin si
5:  fin bucle
6:  buscar elementos arquitectónicos en el Modelo de Arquitectura relacionados con F
7:  si se encuentran elementos arquitectónicos entonces
8:      preguntar al ingeniero si se deben eliminar/mantener los elementos
        arquitectónicos
9:  iniciar Proceso de Validación
```

ILUSTRACIÓN 14 PATRÓN DE ELIMINACIÓN DE FUNCIONALIDAD

De esta forma aseguramos que cuando el ingeniero elimina una característica se eliminan tanto las resoluciones asociadas a la misma así como los elementos arquitectónicos que el ingeniero juzgue que deben eliminarse.

Como comentario, para relacionar las características de un Modelo de Características y los elementos arquitectónicos (del Modelo de Arquitectura) se utiliza un Modelo de Enlace. En este modelo se relacionan características y elementos arquitectónicos de forma que en el

conocimiento del sistema también contiene información sobre qué características requieren qué elementos de la arquitectura para funcionar y viceversa.

A través de este modelo de enlace se obtienen todos los elementos arquitectónicos relacionados con la característica a eliminar y que serán los que se muestren al ingeniero para que decida si han de ser eliminados.

4.3.2 PATRÓN 2: ADICIÓN DE NUEVA FUNCIONALIDAD

Es algo muy habitual que en un sistema sea necesario añadir nuevas funcionalidades, ya sea para corregir funcionalidades que no están funcionando correctamente, para añadir una nueva versión de las mismas o sencillamente porque los requisitos cambien y sea necesario cumplir un nuevo requisito.

La adición de una nueva funcionalidad implica la evolución del Modelo de Características para incorporar nuevas variantes. La incorporación de estas variantes implica actualizar el Modelo de Resoluciones que permitan al sistema utilizar la nueva funcionalidad.

Además, el ingeniero también deberá indicar cuáles son los elementos arquitectónicos que realizan la nueva funcionalidad. Estos elementos pueden existir en el Modelo de Arquitectura o puede ser necesario que se añadan.

Finalmente, estos nuevos elementos de la arquitectura deberán ser accesibles, por lo que se deberá crear un enlace entre la nueva característica y el elemento de la arquitectura.

Para ello, igual que en el anterior caso se propone un patrón de evolución que consistirá en un algoritmo que se ejecutará siempre que se produzca la adición de una nueva característica. El patrón de adición consiste en los siguientes pasos:

1. **Preguntar al ingeniero si es necesario actualizar el Modelo de Resoluciones**: es muy probable que haya que añadir una resolución que des/active la nueva característica añadida para que el sistema pueda utilizarla. Además la utilización de la nueva característica puede requerir que otra característica existente se desactive porque sean incompatibles.

2. **Solicitar al ingeniero los elementos arquitectónicos que realizan la funcionalidad representada por la característica**: se deben proporcionar los elementos del Modelo de Arquitectura del sistema que realizan la funcionalidad representada por la característica.

    a. En caso de que el elemento arquitectónico no exista, **evolucionar el Modelo de Arquitectura** para incluir los nuevos componentes necesarios.

3 **Validar el modelo tras estas modificaciones:** para garantizar que los modelos del sistema siguen siendo válidos tras estas modificaciones habrá que validarlos tras las mismas.

A continuación en la Ilustración 15 se expone el pseudocódigo del patrón de Adición de funcionalidad:

Algoritmo 2: Adición de funcionalidad

Requiere: Una/s Característica/s (C) se ha/n eliminado del *Modelo de Características*

1: Preguntar al ingeniero si ha de actualizarse el *Modelo de Resoluciones*
2: Pedir al ingeniero que seleccione los elementos arquitectónicos del *Modelo de Arquitectura* que realicen la nueva característica
3: **si** los elementos arquitectónicos no existen **entonces**
4: evolucionar el *Modelo de Arquitectura*
5: **fin si**
6: iniciar Proceso de Validación

ILUSTRACIÓN 15 PATRÓN DE ADICIÓN DE FUNCIONALIDAD

El objetivo del patrón de evolución para añadir funcionalidad es asegurar que cuando el ingeniero añada una característica al sistema éste alcance un estado estable y que contenga todos los elementos necesarios para que la nueva característica funcione sin problemas en el sistema. Además, con la mínima participación del ingeniero posible.

4.4 GENERAR PLAN DE EVOLUCIÓN

El sistema en ejecución no se ha modificado todavía, los cambios se han aplicado en una copia de los modelos. Esta fase proporciona mecanismos para generar un Plan de Evolución que contiene las acciones necesarias para modificar el sistema en ejecución como resultado de la evolución ocurrida. Una vez realizadas las modificaciones en el sistema, habrá que generar un Plan de Evolución. El plan de evolución detalla cuáles son las acciones necesarias a realizar para evolucionar el sistema actual hasta la nueva versión (con las adiciones y/o eliminaciones de elementos que se hayan producido en cada modelo).

En primer lugar se computan las diferencias entre una versión del conocimiento del sistema que se haya almacenado y la versión actual del mismo. Se obtienen conjuntos de incrementos y decrementos de los modelos del conocimiento afectados por la evolución. Estos conjuntos ilustran las diferencias entre versiones en términos de elementos de modelos. Estas diferencias se utilizan para construir el Plan de Evolución encargado de actualizar los modelos usados por el sistema en ejecución.

4.5 VALIDAR EL SISTEMA EVOLUCIONADO

Además una vez realizadas todas las modificaciones sobre el sistema será necesario que el ingeniero tenga la posibilidad de validar el sistema para asegurar que se encuentra en un estado estable.

Para ello a partir del estado actual del sistema habrá que generar un Modelo del Espacio de Adaptación implícito que se deriva a partir de los modelos que constituyen el conocimiento del sistema. El Espacio de Adaptación se representa mediante un Modelo de Máquina de Estados que representa todos los estados posibles del mismo y las transiciones posibles entre los distintos estados.

A partir de este modelo se podrán aplicar una serie de refinamientos que irán orientados a garantizar en el sistema una serie de propiedades deseables de diseño y comportamiento. Las propiedades a garantizar son [30]:

- Determinismo: este refinamiento garantiza que el sistema de un estado dado y cuando una condición sea cumplida sólo podrá reconfigurarse a un solo estado. Para ello se modifica el Modelo de Resoluciones para evitar reconfiguraciones simultáneas.

- Reversibilidad: un modelo reversible es aquel modelo en el que a partir de una reconfiguración (de un estado) es posible volver a la reconfiguración anterior. Así mismo, un modelo no reversible es todo aquel modelo en el que a partir de una reconfiguración es imposible volver a la reconfiguración previa. Existirá un refinamiento para garantizar ambos casos, el de un sistema no reversible y el de un sistema reversible.

- No redundancia: un sistema es redundante cuando dos o más reconfiguraciones llevan al sistema de una misma configuración a la misma configuración destino. Para evitar esta situación existirá un refinamiento que encontrará todas las resoluciones redundantes y únicamente dejará las más simples de ellas. La resolución más simple será aquella que implique el mínimo número de modificaciones.

- Eliminación de bucles: existirá un refinamiento que evitará que se produzcan bucles de reconfiguración en el sistema.

Cada vez que se aplique uno de estos refinamientos se generará un Modelo de Resoluciones que garantizará las propiedades respectivas a cada uno de ellos. Además, es posible que se apliquen varios refinamientos combinados (determinismo y reversibilidad, por ejemplo).

5. HERRAMIENTA DE SOPORTE A LA PROPUESTA

En este capítulo se describe la herramienta implementada para dar soporte a las funcionalidades presentadas en el capítulo anterior. Esta herramienta se integra dentro de la herramienta MOSKitt4SPL. A continuación se describen las tecnologías utilizadas para realizar la implementación, las funcionalidades proporcionadas y nociones básicas de utilización de la herramienta.

5.1 PROCESO DE DESARROLLO

La filosofía de desarrollo que se ha seguido en el desarrollo de la herramienta es incremental [31], es decir, se han ido planteando primero requisitos sencillos y poco a poco se ha ido desarrollando uno tras otro cada uno de ellos.

Se ha tomado esta opción ya que era necesaria una inmersión en las plataformas de M4SPL y de EMF para abordar todos los requisitos y era imposible abordar, por ejemplo, los patrones de evolución sin tener unos conocimientos mínimos del desarrollo en EMF o del código de M4SPL.

Además mediante esta filosofía se han podido ir desarrollando y depurando cada uno de los requisitos de forma individual, siguiendo un ciclo en espiral para cada uno de ellos.

5.1.1 DESARROLLO INCREMENTAL

El desarrollo incremental nace ante las carencias del desarrollo tradicional o en cascada. El desarrollo en cascada supone que el desarrollo software se compone de una serie de pasos que finalizan siempre en la entrega del producto final.

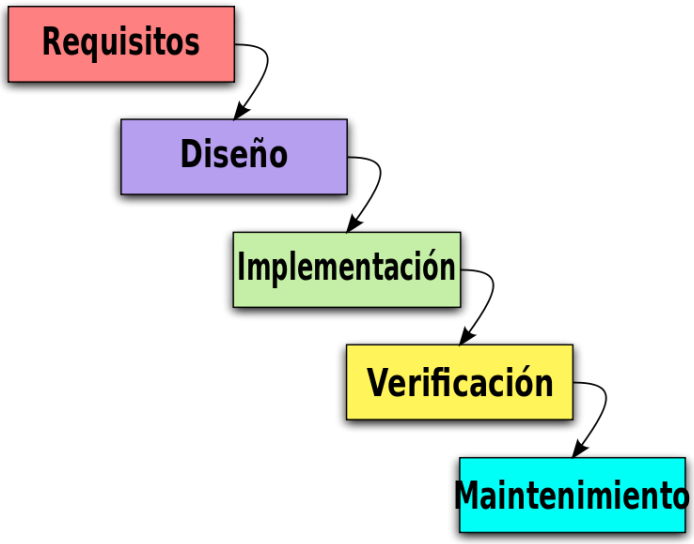

ILUSTRACIÓN 16 DESARROLLO EN CASCADA

La Ilustración 16 DESARROLLO EN CASCADA muestra el desarrollo tradicional o en cascada

En cambio, el desarrollo incremental o iterativo propone que el desarrollo del producto final se componga de diversas fases o iteraciones en las que en cada una de ellas se aplique un desarrollo en cascada. De esta forma en cada iteración se obtiene un artefacto software que es una versión del producto final o prototipo.

Así tras cada iteración se entrega al cliente el prototipo que va viendo cómo se va conformando el producto final. En caso de que detecte algún error en el desarrollo, será capaz de informar de ello al equipo de desarrollo antes de haber llegado al producto final.

Esta corrección puede ser especialmente valiosa en caso de ser un error de análisis, ya que corregir un error de análisis en un producto final puede ser extremadamente costoso o incluso imposible.

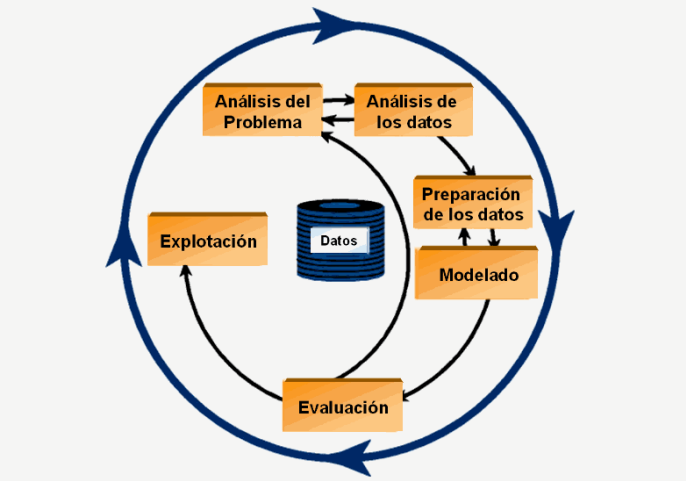

ILUSTRACIÓN 17 DESARROLLO ITERATIVO

La Ilustración 17 muestra el desarrollo iterativo o incremental

Mediante el uso de esta metodología de desarrollo se ha demostrado que se aumenta la productividad, se reduce el coste de los proyectos y se obtiene un mayor grado de cumplimiento en los proyectos. Sin embargo, no es la panacea para el desarrollo software.

5.2 INTERFAZ

Para la interfaz de la implementación de los requisitos previamente explicados, se ha optado por el uso de los “Wizards” o asistentes de Eclipse.

Una de las características de Eclipse es que permite añadir toda una capa de funcionalidades y plugins a su lista de características. Uno de los ejemplos de esta posibilidad de extensión son los asistentes. Eclipse permite desarrollar asistentes como los que él mismo utiliza, de forma que se pueden desarrollar muy fácilmente asistentes paso a paso para realizar procesos complejos.

Se ha optado por esta solución ya que como se ha visto en la visión general de la propuesta, todos los procesos a realizar son procesos que siguen inevitablemente una serie de pasos (escoger el modelo de resolución, escoger el de arquitectura, aplicar acciones sobre ellos…). Un asistente es precisamente una serie de ventanas que se van siguiendo en un orden determinado y en las que se van realizando acciones paso a paso.

5.3 LÓGICA

A lo largo de este punto se entrará al detalle de la implementación de cada uno de los requisitos, tanto a nivel teórico como con ejemplos de código y capturas de las pantallas finales de los asistentes.

5.3.1 IMPLEMENTACIÓN DE ALMACENAMIENTO DEL CONOCIMIENTO DEL SISTEMA

Para almacenar el conocimiento del sistema se ha creado una acción contextual (clicando con el botón derecho) que permita almacenar una copia de todos los modelos del conocimiento del sistema (extraídos del workspace actual).

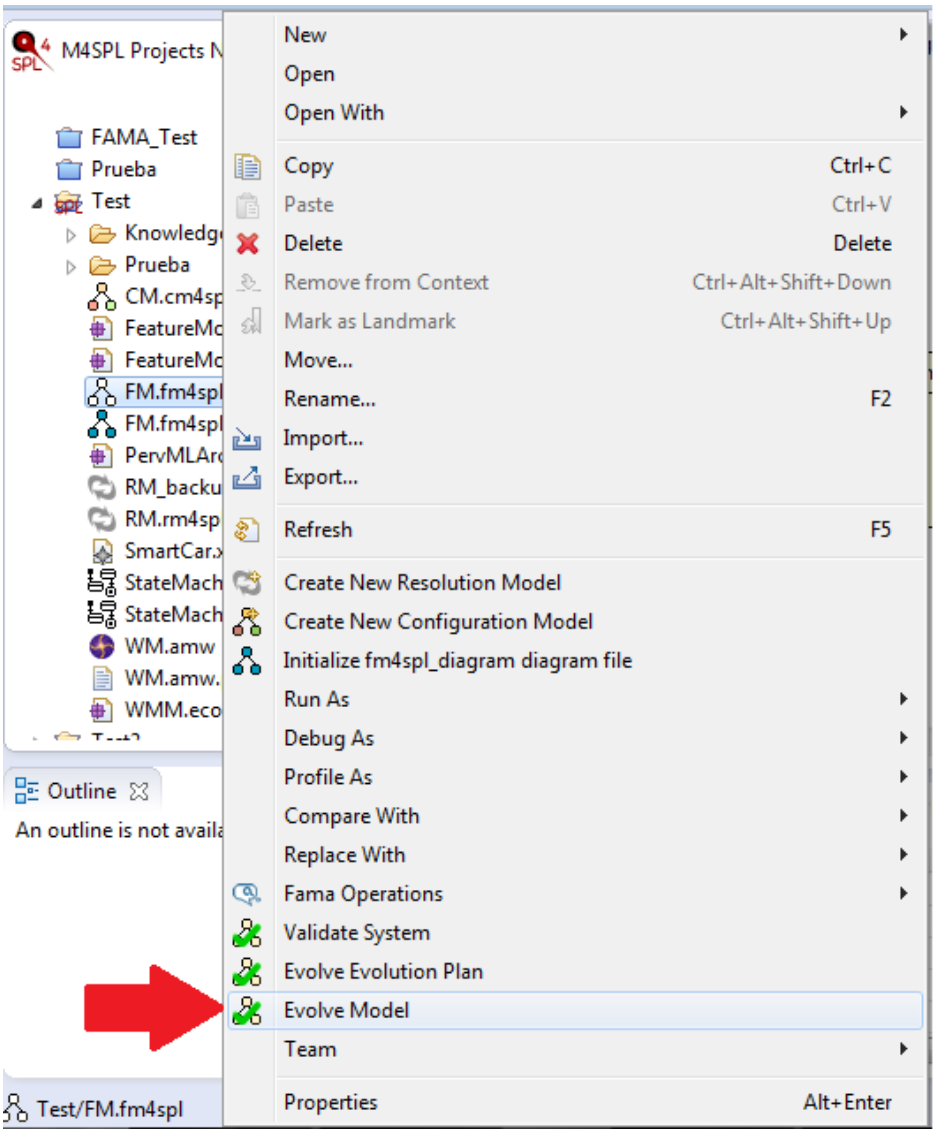

ILUSTRACIÓN 18 COPIA DEL CONOCIMIENTO DEL SISTEMA

La Ilustración 18 se muestra la acción contextual para copiar el conocimiento del sistema.

Todos estos modelos se almacenan dentro de una carpeta llamada Knowledge Versions (que habrá que crear si no existe) y a su vez dentro de una carpeta cuyo nombre sea la concatenación de la fecha y hora actuales (de forma que el nombre sea único dentro de nuestro workspace).

Deben almacenarse todos los modelos presentes, ya estén en la raíz del workspace o en subcarpetas. Además debe garantizarse que se copian exactamente con todos los contenidos que tenían originalmente.

Otro requisito que habrá que cumplir es que todas las referencias entre modelos deberán modificarse para que apunten a la nueva ruta donde se encuentren. Se ve más claro con la siguiente Ilustración 19:

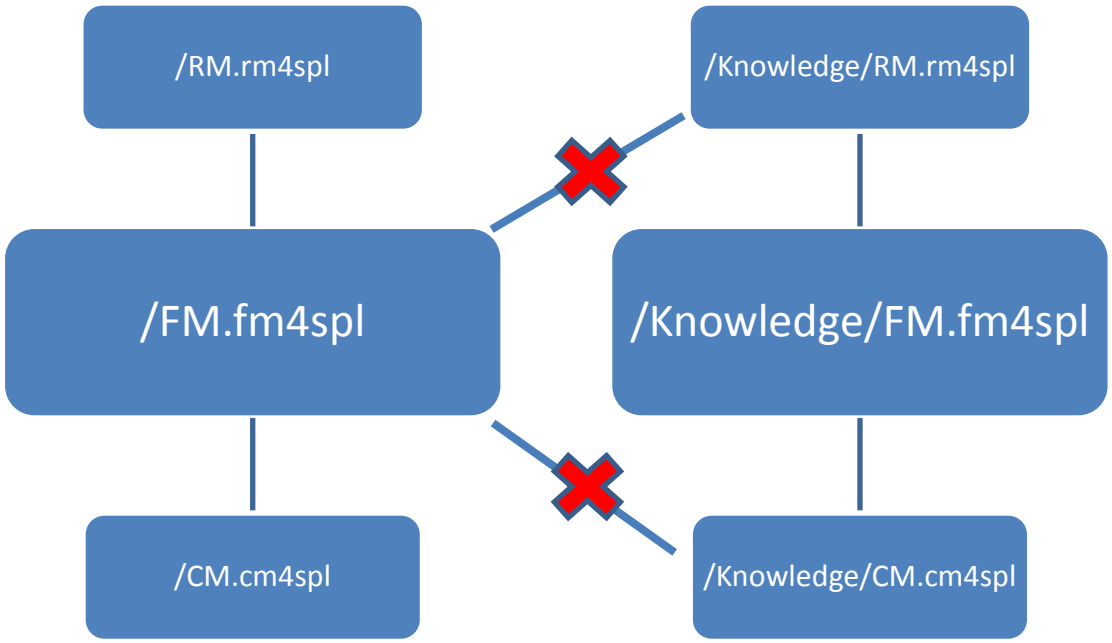

ILUSTRACIÓN 19 EJEMPLO DE COPIA DEL CONOCIMIENTO DEL SISTEMA

La Ilustración 19 Ejemplo de copia del conocimiento del sistema muestra en la columna izquierda el conocimiento del sistema y en la derecha una copia del mismo, en una carpeta "Knowledge". Lo que se quiere representar es que los modelos de la derecha (las copias) no deben contener ningún enlace a los modelos originales.

Para ello se han implementado una acción contextual llamada EvolveModel. Se ha incluido en el proyecto FAMA.Reasoner ya que es ahí donde se encuentran las acciones contextuales respecto a los modelos.

Esta acción se aplicará únicamente para los modelos de características y se ha añadido a la configuración del plugin del reasoner. Este es el XML en el que se añade esta acción:

```
      <objectContribution
            adaptable="true"
            id="es.upv.pros.M4SPL_Reasoning.FAMA_Reasoner.operations"
            nameFilter="*fm4spl"
            objectClass="org.eclipse.core.resources.IFile">
         <action

class="es.upv.pros.m4spl_reasoning.fama_reasoner.actions.EvolveModelAction"
               icon="icons/validateModelIcon.gif"
               id="es.upv.pros.M4SPL_Reasoning.FAMA_Reasoner.actions.evolve"
               label="Evolve Model"

menubarPath="es.upv.pros.M4SPL_Reasoning.FAMA_Reasoner.operations">
         </action>
```

Como se puede ver, se añade la acción "EvolveModelAction" de forma que se aplique a aquellos archivos cuya extensión sea "*fm4spl". Una vez hecho esto, cada vez que pulsemos el botón derecho sobre un Modelo de Características podremos seleccionar la acción "Evolve Model" y realizar una copia del conocimiento del sistema.

Esta operación lo que hará será recorrer todos los archivos que haya en el espacio de trabajo y los copiará a "/Knowledge Versions/EvolutionId", donde "EvolutionId" será un id de carpeta que se generará en función de la fecha y hora actuales. De esta forma también tendremos una trazabilidad clara respecto al momento en que se realizó la copia del conocimiento.

Este id de evolución se solicitará a la clase "AdapterHelper" que es una clase donde se han incluido las funcionalidades extra que se vayan necesitando. Para obtenerlo se llamará al método estático "getEvolutionId":

```
public static String getEvolutionId() {
      Calendar calendar = Calendar.getInstance();

      String result = "";

      result = calendar.get(Calendar.DAY_OF_MONTH) + "-"
                  + calendar.get(Calendar.MONTH) + "-"
                  + calendar.get(Calendar.YEAR) + "_"
                  + calendar.get(Calendar.HOUR_OF_DAY) + "-"
                  + calendar.get(Calendar.MINUTE) + "-"
                  + calendar.get(Calendar.SECOND) + "";

      return result;
}
```

Un ejemplo del resultado que se obtiene de esta función puede ser el id de evolución "11-5-2013_18-15-36". Una vez obtenido el id de evolución, ya podemos copiar todos los modelos del espacio de trabajo a una carpeta cuyo nombre sea este id.

Para ello se ha desarrollado el siguiente código de la clase "EvolveModelOperation":

```
public class EvolveModelOperation extends AbstractFMReasoningOperation {

	public EvolveModelOperation(String _reasonerID, BundleContext bc) {
		super("FAMA_EVOLVEMODEL", _reasonerID, bc);
}

@SuppressWarnings("restriction")
@Override
public Object executeOperation(Object parameters) {

	if (parameters instanceof IStructuredSelection) {
		IStructuredSelection selection = (IStructuredSelection) parameters;

		if (selection.getFirstElement() instanceof IFile) {
			IFile file = (IFile) selection.getFirstElement();

			IResource[] resources;
			try {
				resources = file.getProject().members();

				String evolutionId = AdapterHelper.getEvolutionId();

				copyResources(resources, evolutionId, "");

			} catch (CoreException e1) {
				e1.printStackTrace();
			}
		}
	}

	return "OK";
}
```

Como se puede ver, a partir del modelo seleccionado se obtiene el proyecto actual y a partir del él, los miembros que lo compongan. Es decir, se obtienen todos los modelos existentes, mediante la línea "`resources = file.getProject().members();`". Una vez obtenidos todos los modelos, se procederá a copiarlos en "/KnowledgeVersions/EvolutionId", dentro del espacio de trabajo.

Para ello se ha desarrollado el método copyResources:

```
@SuppressWarnings("restriction")
private void copyResources(IResource[] resources, String evolutionId,
		String aditionalPath) {
	Resource aux = null;
```

```
	Map<String, String> paths = new HashMap<String, String>();

	Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
			"xml", new XMLResourceFactoryImpl());

	for (IResource resource : resources) {

		if (resource.getType() == IResource.FILE) {
			copyFile(resource, evolutionId, aditionalPath);

		} else if (resource.getType() == IResource.FOLDER) {
			copyFolder(resource, evolutionId, aditionalPath);
		}
	}
}
```

Como se ve, se recorren los recursos y en función de si es un archivo o un directorio se llama a copyFile o copyFolder. El método copyFolder se utiliza para copiar los recursos que haya en subcarpetas, para ello se recorren sus archivos y a la hora de construir la ruta de destino se concatenerá el nombre de la carpeta.

Es decir, si tenemos un archivo en /Models/SubModels/FM.fm4spl, se copiará en /Knowledge Versions/EvolutionId/SubModels/Fm.fm4spl. Se mantendrá la jerarquía de directorios.

```
private void copyFolder(IResource resource, String evolutionId,
		String aditionalPath) {
	Folder folder = (Folder) resource;

	try {
		if (!folder.getName().toLowerCase().contains("knowledge versions"))
{

			String originalPath = aditionalPath;

			aditionalPath += "/" + folder.getName() + "/";

			copyResources(folder.members(), evolutionId, aditionalPath);

			aditionalPath = originalPath;
		}
	} catch (CoreException e) {
		e.printStackTrace();
	}
}
```

Pero el método realmente importante es el método copyFile. Este será el que realice la copia de los archivos:

```
private void copyFile(IResource resource, String evolutionId,
		String aditionalPath) {
	File file = (File) resource;
```

```
        Resource emfResource = null;

        ResourceSet rSet = new ResourceSetImpl();

        URI uri = URI.createFileURI(file.getLocation().toString());

        try {
                emfResource = rSet.getResource(uri, true);

                CopyModelHelper copyModel = new CopyModelHelper(emfResource,
evolutionId,
                                aditionalPath);

                copyModel.backupModel();

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

                copyPlainFile(uri, evolutionId, aditionalPath, file);
        }
}
```

Las dos líneas a destacar son "copyModel.backupModel()" y "copyPlainfile". CopyPlainFile se utiliza para el caso de que hayan archivos que no sean modelos y por tanto no cumplan la especificación de EOBject de EMF [32]. Sin embargo, el realmente importante será el "backupModel", ya que será el que copie los modelos.

Para ello y como se puede observar, se ha implementado una clase "CopyModelHelper" que encapsula las acciones a llevar a cabo para almacenar el conocimiento del sistema. Se ha hecho de esta forma porque deberá llevar a cabo dos acciones:

- **Copiar los modelos a la carpeta de "Knowledge Versions" a partir del id de evolución que se haya construido:** Esta tarea es relativamente trivial, ya que simplemente es realizar una copia de un archivo.

- **Eliminar referencias con los modelos originales**: esta será la parte más compleja. Habrá que realizarla porque por ejemplo un Modelo de Resoluciones tiene referencias a su Modelo de Características y ha de referenciar a la nueva copia y no al original. La Ilustración 19 explica gráficamente.

Para ello se ha implementado el siguiente código:

```
public EObject backupModel() {

        try {
                ResourceSet resourceSet = new ResourceSetImpl();

                URI uri = resourceToCopy.getURI();

                URI fileUri = URI.createFileURI(getDestinationUri(uri, evolutionId,
                                aditionalPath, versions));
```

```
		Resource resource = resourceSet.createResource(fileUri);

		if (modelToSave != null) {
			resource.getContents().add(modelToSave);
		} else if (resourceToCopy != null
				&& !resourceToCopy.getContents().isEmpty()) {
			if (resourceToCopy.getContents().get(0).eResource() != null)
{
				resourceToCopy.getContents().get(0).eResource()
						.setURI(resource.getURI());
			}

			modelToSave = resourceToCopy.getContents().get(0);

			redirectAllReferences(uri, fileUri, resourceToCopy
					.getContents().get(0));


	resource.getContents().add(resourceToCopy.getContents().get(0));
		}

		resource.save(Collections.EMPTY_MAP);

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

	// Refresh
	try {
		ResourcesPlugin.getWorkspace().getRoot()
				.refreshLocal(IResource.DEPTH_INFINITE, null);

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

	return modelToSave;
}

private void redirectAllReferences(URI originalUri, URI destinationUri,
		EObject resourceToCopy) {
	EList<EObject> references = resourceToCopy.eCrossReferences();

	for (EObject reference : references) {
		changeReferences(originalUri, destinationUri, reference, true);
	}

	TreeIterator<EObject> allContents = resourceToCopy.eAllContents();

	while (allContents.hasNext()) {
		EObject content = allContents.next();

		changeReferences(originalUri, destinationUri, content, true);
	}
```

}

Una vez copiados los modelos de forma que no contengan ninguna referencia a los modelos originales, habremos almacenado una copia del conocimiento del sistema. Esta copia del conocimiento podremos utilizarla para:

- Tener una copia de seguridad del sistema, frente a evoluciones del sistema que puedan llevar a un estado inestable al mismo.

- Tener una versión del conocimiento frente a la que poder comparar el estado del sistema tras su evolución. Esto se explica en la generación de planes de evolución.

5.3.2 IMPLEMENTACIÓN DE LOS PATRONES DE EVOLUCIÓN

En esta sección se muestra la implementación concreta de los dos patrones de evolución: "Eliminación de funcionalidades" y "Adición de funcionalidades". Se mostrarán los requisitos cumplidos, la interfaz y la lógica para aplicar cada uno de los patrones.

5.4.2.1 IMPLEMENTACIÓN DE ELIMINACIÓN DE FUNCIONALIDADES

Lo primero que se ha hecho para implementar el patrón de eliminación de funcionalidades es desarrollar un escuchador de EMF que detectará los cambios sobre los modelos de características y determinará si la acción que se está realizando es una eliminación de una funcionalidad.

Este "escuchador" es una clase llamada FeatureAdapter que extiende la clase AdapterImpl. Este escuchador se añade a todos los modelos de características que se generan modificando las clases FeatureModelPackageAdapterFactory (que es la factoría que ha de construir los escuchadores) y FeatureModelPackageFactoryImpl (que es la factoría que construye todos los objetos de tipo FeatureModel) de forma que siempre se le añada a todos los modelos de características nuestro escuchador. Este es el código (resumido) para ello:

```
public class FeatureModelPackageAdapterFactory extends AdapterFactoryImpl {

	public Adapter createFeatureModelAdapter() {
		return new FeatureAdapter();
	}
}

public class FeatureModelPackageFactoryImpl extends EFactoryImpl implements
		FeatureModelPackageFactory {

	public FeatureModel createFeatureModel() {
		FeatureModelImpl featureModel = new FeatureModelImpl();

		featureModelPackageAdapterFactory.adapt(featureModel,
FeatureModel.class);

		return featureModel;
	}
}
```

Una vez modificados estos dos métodos, todos los objetos de tipo FeatureModel tendrán nuestro escuchador asociado para detectar los cambios que nos interesen.

La clase FeatureAdapter (nuestro escuchador) sobrescribe el método notifyChanged. Este es el método que hay que sobrescribir para recibir las notificaciones de cambios sobre el objeto que estamos observando. Dentro de este método se filtran únicamente aquellas notificaciones cuyo tipo sea "Add", "Remove" o "Remove Many". Este último tipo se explicará su uso en este punto más adelante.

Para este caso (eliminación), se comprueba si la notificación es de tipo "Remove" y que además el objeto que lo está notificando (el "notifier") es un Modelo de Características (un FeatureModel).

Este es el código que lo realiza en la clase FeatureAdapter:

```
public class FeatureAdapter extends AdapterImpl {

protected NotificationFilter filter;

private RemoveAdapter removeAdapter;

private AddAdapter addAdapter;

public FeatureAdapter() {
	filter = NotificationFilter
			.createEventTypeFilter(Notification.REMOVE)

	.or(NotificationFilter.createEventTypeFilter(Notification.ADD))
			.or(NotificationFilter

	.createEventTypeFilter(Notification.REMOVE_MANY));
}

@Override
public void notifyChanged(Notification notification) {

	addAdapter = new AddAdapter(notification);
	removeAdapter = new RemoveAdapter(notification);

	Object notifier = notification.getNotifier();

	if (filter.matches(notification) && notifier != null &&
((Resource.Internal) ((EObject) notifier).eResource()) != null &&
!((Resource.Internal) ((EObject) notifier).eResource()).isLoading()) {

		switch (notification.getEventType()) {
		case Notification.REMOVE:
			handleRemove(notification);
			break;

		case Notification.ADD:
			handleAdd(notification);
			break;

		case Notification.REMOVE_MANY:
			handleRemoveMany(notification);
			break;
		}
	}
}

protected void handleRemove(Notification notification) {
```

```
	if (notification.getNotifier() instanceof FeatureModel) {
		removeAdapter.handleRemoveFeature(notification);
	}
}
```

Si se cumplen todas estas características se lanzará un asistente llamado "RemoveWizard". Este asistente se lanzará mediante una clase RemoveAdapter (cuyo uso se ve en el código). Es dentro de esta clase donde se ha desarrollado el código inherente a las tareas a realizar dentro de la eliminación de una característica del modelo, como se puede ver a continuación:

```
public RemoveAdapter(Notification notification) {
	helper = new AdapterHelper();
	this.notification = notification;
}

public Object[] handleRemoveFeature(Notification notification) {

	if (notification.getNotifier() instanceof FeatureModel &&
notification.getOldValue() instanceof Feature) {

		WizardDialog wizardDialog = new WizardDialog(PlatformUI
				.getWorkbench().getActiveWorkbenchWindow().getShell(),
				new RemoveWizard(this));

		wizardDialog.open();
	}

	return null;
}
```

En este código se observa que se lanza un asistente ("wizard") llamado RemoveWizard. Será este asistente el que vaya mostrando uno a uno los pasos a seguir en el patrón de eliminación de funcionalidades.

No se va a exponer todo el código del asistente, ya que es muy extenso e incluye mucho código de interfaz y validaciones sobre la misma. Sólo se comentarán los aspectos clave que se realicen respecto al patrón en sí y el paso entre pantallas o pasos del patrón. El asistente consistirá de dos pasos.

5.3.2.1.1 PASO 1: ELIMINAR RESOLUCIONES ASOCIADAS

En el primer paso se solicitará al ingeniero que seleccione el Modelo de Resoluciones a utilizar:

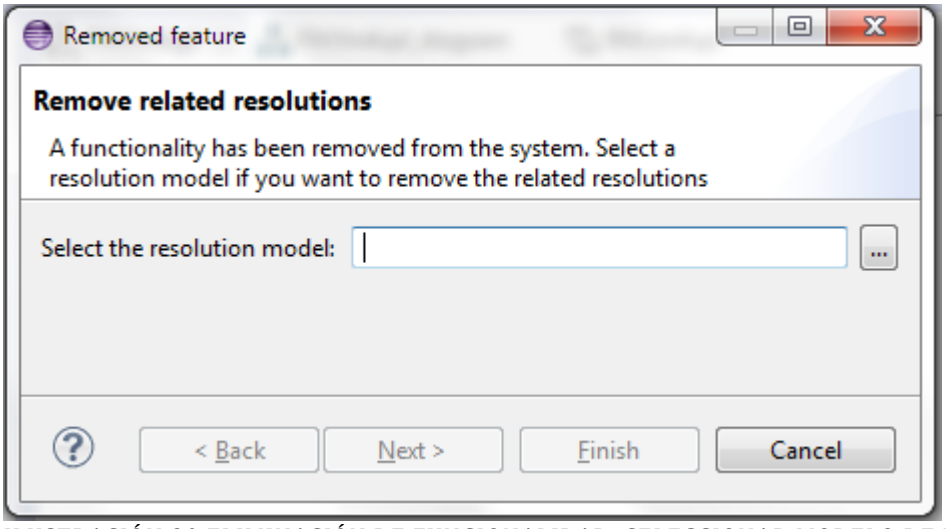

ILUSTRACIÓN 20 ELIMINACIÓN DE FUNCIONALIDAD: SELECCIONAR MODELO DE RESOLUCIÓN

En esta página del asistente se escogerá el modelo de resolución a utilizar ante una eliminación de resolución.

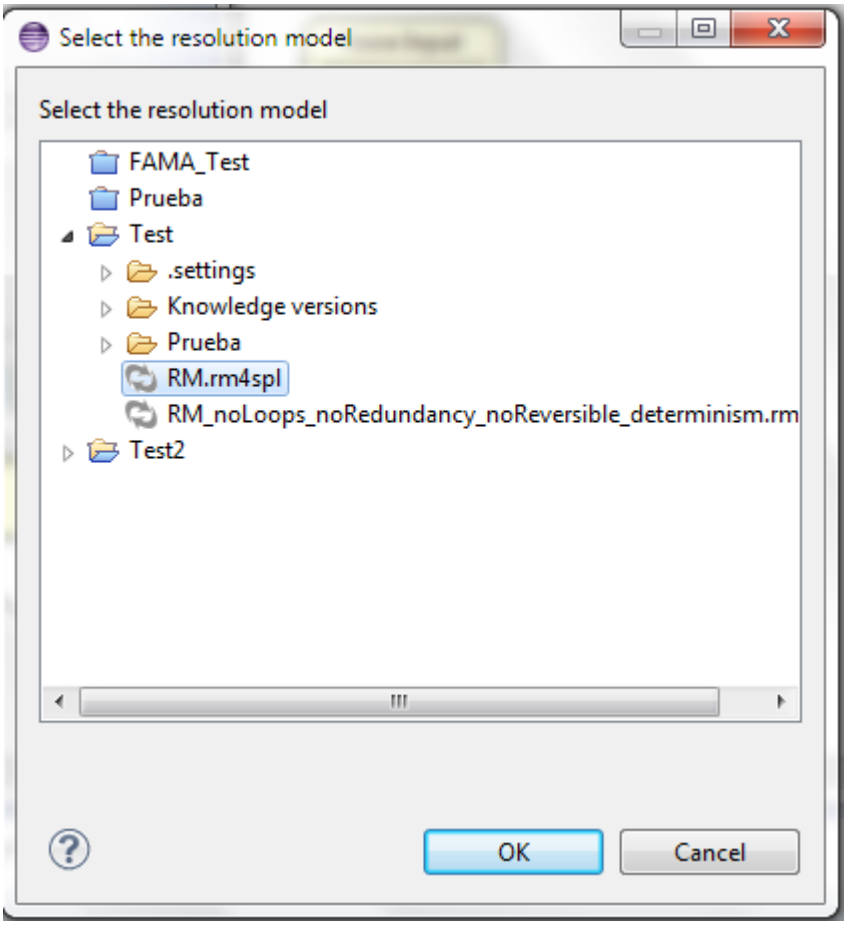

ILUSTRACIÓN 21 SELECCIÓN DEL MODELO DE RESOLUCIÓN DEL ESPACIO DE TRABAJO

En este diálogo se escogerá el modelo de resolución de entre los existentes en el espacio de trabajo. Se filtrarán los archivos de forma que sólo aparezcan los que tengan extensión rm4spl.

Como se ve en la Ilustración 21 sólo se permitirán seleccionar archivos cuya extensión sea rm4spl para evitar que el ingeniero pueda seleccionar otro tipo de modelo que no sea de resoluciones.

Además los botones de Next y Finish no estarán activos hasta que se seleccione el modelo de resolución, ya que es necesario borrar todas las resoluciones asociadas a la característica eliminada para no dejar el sistema en un estado inestable.

Una vez seleccionado el Modelo de Resoluciones, se buscará si hay resoluciones asociadas a la característica eliminada (la activen o la desactiven). Para ello lo que se hace es que se notifica de un evento de eliminación al escuchador del Modelo de Resoluciones.

```
public void removeResolutions(EObject resolutionModel) {
	if (resolutionModel != null) {
		Notification aux = new NotificationImpl(Notification.REMOVE_MANY,
				notification.getOldValue(),
notification.getNotifier());

		NotificationExtended notificationResolution = new
NotificationExtended(
				resolutionModel, aux);

		resolutionModel.eAdapters().get(0)
				.notifyChanged(notificationResolution);
	}
}
```

Este código crea una notificación en la que se indica el "oldValue" (la característica eliminada), y el "notifier" (el Modelo de Características). Una vez construida, se llamará al método "notifyChanged" para notificar del cambio. Esta notificación la manejará el escuchador del Modelo de Resoluciones mediante el siguiente código:

```
@Override
protected void handleRemoveMany(Notification notification) {
	if (notification.getNewValue() instanceof FeatureModel) {
		FeatureModel featureModel = (FeatureModel) notification
				.getNewValue();

		Feature featureRemoved = (Feature) notification.getOldValue();

		ResolutionModel resolutionModel = (ResolutionModel) notification
				.getNotifier();

		List<Resolution> toRemove = getRelatedResolutions(featureModel,
resolutionModel, featureRemoved);

		if (toRemove.size() > 0) {
```

```
                        showRemoveDialog(formatResolutions(toRemove).toArray());

                        resolutionModel.getResolutions().removeAll(toRemove);

                        try {

        resolutionModel.eResource().save(Collections.EMPTY_MAP);

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

private List<Resolution> getRelatedResolutions(FeatureModel featureModel,
ResolutionModel resolutionModel, Feature featureRemoved){
        List<Resolution> toRemove = new ArrayList<Resolution>();

        XmiModel xmiModel = new XmiModel(
                        FeatureModelPackagePackage.eINSTANCE, featureModel
                                        .eResource().getURI());

        for (Resolution res : resolutionModel.getResolutions()) {
                for (Action act : res.getActions()) {

                        Feature f = act.getFeature();

                        if(f.eIsProxy()){
                                f = getReferencedFeature(xmiModel, f);
                        }

                        if (f != null &&
featureRemoved.getName().equals(f.getName())) {
                                toRemove.add(res);
                                break;
                        }
                }
        }

        return toRemove;
}

private Feature getReferencedFeature(XmiModel xmiModel, Feature f){
        return (Feature) xmiModel.getModelElementByIDREF(((InternalEObject)
f).eProxyURI().fragment());
}

private List<String> formatResolutions(List<Resolution> resolutions){

        List<String> result = new ArrayList<String>();

        for(Resolution r : resolutions){
                if(r.getId() != null && !r.getId().equals("")){
                        result.add(r.getId());
```

```
                }
                else{
                        result.add("Empty Id");
                }
        }

        return result;
}

private void showRemoveDialog(Object[] relatedElementsToRemove) {
        Shell shell = PlatformUI.getWorkbench().getActiveWorkbenchWindow()
                        .getShell();

        ListDialog ld = new ListDialog(shell);
        ld.setAddCancelButton(true);
        ld.setContentProvider(new ArrayContentProvider());
        ld.setLabelProvider(new LabelProvider());
        ld.setInput(relatedElementsToRemove);
        ld.setTitle("Resolutions to remove.");
        ld.setMessage("The following resolutions are going to be removed:");
        ld.open();
}
```

Básicamente lo que se hace es lo comentado previamente, se buscan las resoluciones asociadas a la característica y se muestran en el siguiente diálogo para informar al ingeniero de que van a ser eliminadas.

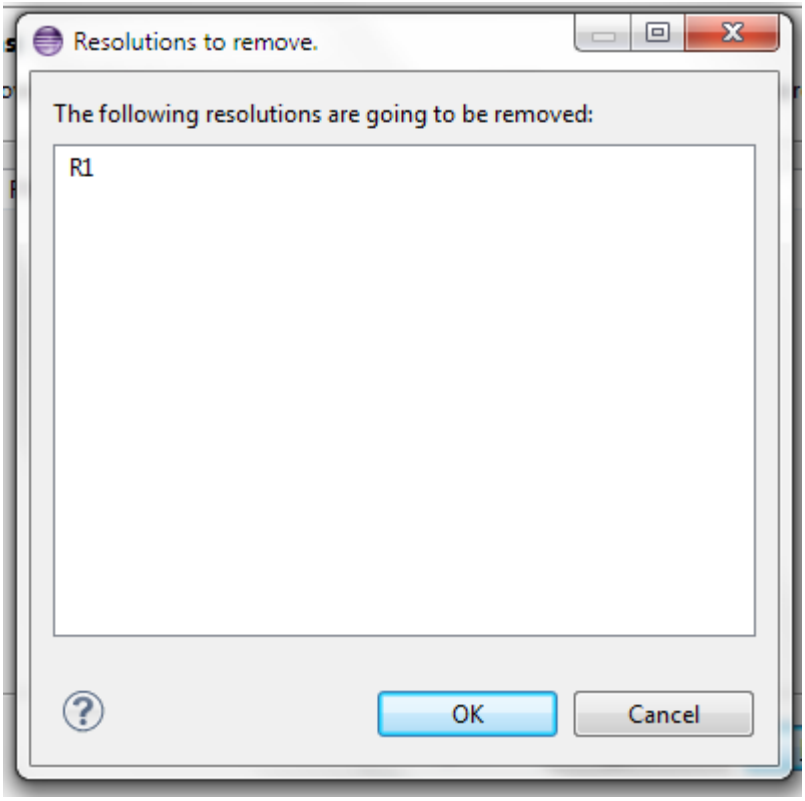

ILUSTRACIÓN 22 MOSTRAR RESOLUCIONES A ELIMINAR

En este diálogo se le mostrarán al ingeniero las resoluciones que serán eliminadas por estar relacionadas con la característica eliminada.

Una vez eliminadas las resoluciones asociadas, vamos al paso 2 del patrón de eliminación de funcionalidades.

5.3.2.1.2 PASO 2: ELIMINACIÓN DE ELEMENTOS DE LA ARQUITECTURA

En el paso 2 se eliminarán los elementos de la arquitectura asociados con la característica que se haya eliminado. Para ello lo primero que habrá que hacer es encontrar el modelo de enlace que asocie el Modelo de Características del que se ha eliminado la característica con el de arquitectura.

En este caso no se pedirá al ingeniero que indique qué modelo de enlace hay que utilizar, ya que sólo existirá uno en el espacio de trabajo. Por ello, habrá que buscarlo y en caso de que exista obtener el Modelo de Arquitectura asociado al Modelo de Características.

Para ello se utilizará el siguiente código:

```
public WModel getWModel(Notification notification) {

	Resource r = ((EObject) notification.getNotifier()).eResource();

	if (r == null) {
		r = resolveresource(r, notification);
	}

	URI uri = r.getURI();

	EPackage.Registry.INSTANCE.put(Mw_base_extPackage.eNS_URI,
			Mw_base_extPackage.eINSTANCE);

	try {

		java.io.File f = new java.io.File(getAbsoluteURI(uri));

		List<EObject> weavers = ModelUtils.getModelsFrom(f, "amw", null);

		if (weavers.size() > 0 && weavers.get(0) != null) {
			return (WModel) weavers.get(0);
		}

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

	return null;
}
```

Una vez obtenido el modelo de enlace, a partir de él podemos obtener el Modelo de Arquitectura asociado al Modelo de Características. A partir del mismo, se obtendrán los elementos de la arquitectura que tengan algún enlace con la característica eliminada:

```
public List<String> getRelatedArchitectureElements() {
	EObject removed = (EObject) notification.getOldValue();

	if (wModel != null) {
		List<String> removedFeatures = getRemovedFeaturesReferences(wModel,
removed);

		List<String> relatedFeatures =
getArchitectureElements(removedFeatures);

		return relatedFeatures;
	}

	return null;
}
```

El método "getRemovedFeaturesReferences" busca en el modelo izquierdo del modelo de enlace (el Modelo de Características) todos aquellos enlaces que contengan la característica eliminada y devuelve una lista con las referencias en el modelo derecho (Modelo de Arquitectura).

Un ejemplo de ello es la siguiente figura, que ilustra la eliminación de la característica "Localización" que está enlazada con el elemento de la arquitectura GPS:

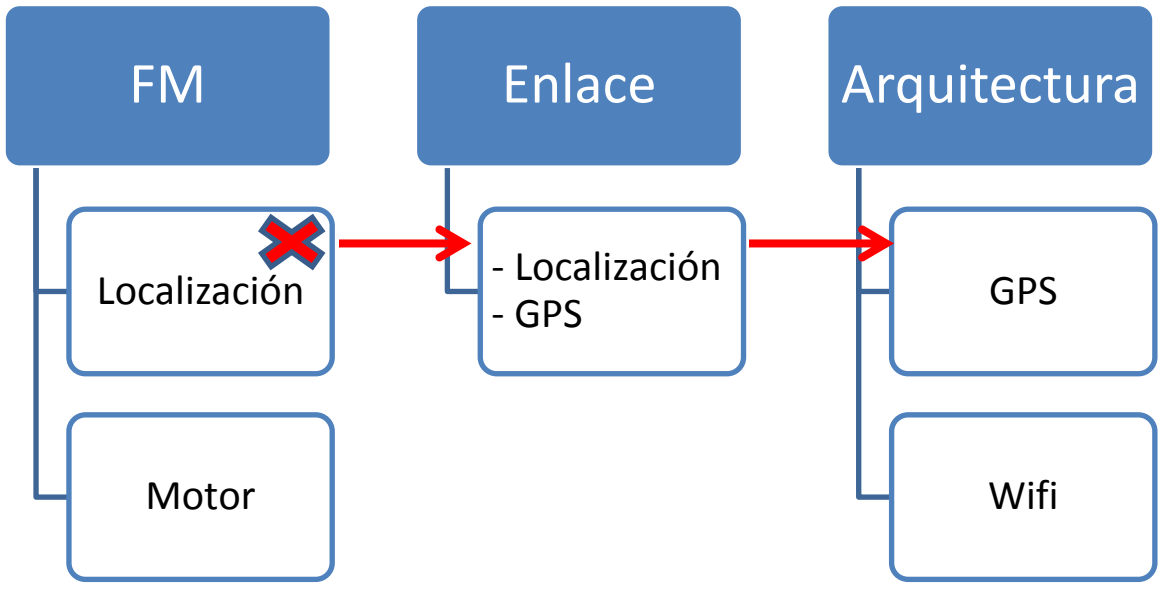

ILUSTRACIÓN 23 ELIMINACIÓN DE CARACTERÍSTICA – ENLACE CON ARQUITECTURA

En la Ilustración 23 se representa con una "X" la característica eliminada "Localización" y como ésta está enlazada con el elemento de arquitectura GPS.

De esta forma tendremos las referencias a eliminar del Modelo de Arquitectura y se mostrará el siguiente diálogo para que el ingeniero decida cuáles son los elementos de la arquitectura que hay que eliminar:

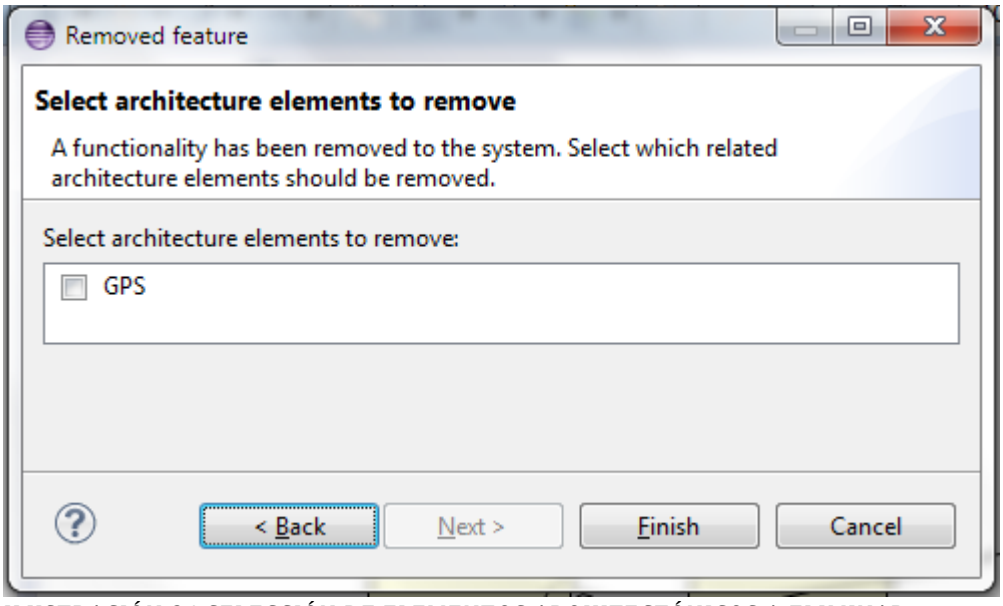

ILUSTRACIÓN 24 SELECCIÓN DE ELEMENTOS ARQUITECTÓNICOS A ELIMINAR

En esta página del asistente el ingeniero escogerá qué elementos de la arquitectura tendrán que ser eliminados ante la eliminación de una característica enlazada.

Una vez seleccionados los elementos de la arquitectura a eliminar, se eliminarán de la misma (en caso de que se seleccionen elementos):

```
public void removeRelatedArchitectureElements(List<String>
relatedElementsToRemove) {

	EObject removed = (EObject) notification.getOldValue();

	if (relatedElementsToRemove != null
			&& relatedElementsToRemove.size() > 0) {

		XmiModel xmiModel = helper.getRightModel(wModel);

		List<String> toRemove = getIdRefsFromIds(xmiModel,
relatedElementsToRemove);

		xmiModel.removeByIdRef(toRemove.toArray(new
String[relatedElementsToRemove.size()]));
	}

	removeRelatedLinks(wModel, removed);
}
```

```
private List<String> getIdRefsFromIds(XmiModel xmiModel, List<String>
idsToRemove){
	List<String> toRemove = new ArrayList<String>();

	Architecture arch = (Architecture)
xmiModel.resource.getContents().get(0);

	relatedsLoop:for (String related : idsToRemove) {
		boolean toContinue = false;

		for (Component comp : arch.getComponents()) {
			if (comp.getID().equals(related)) {
				toRemove.add(xmiModel.getEobjectIDREF(comp));
				toContinue = false;
				continue relatedsLoop;
			}
		}

		if (!toContinue) {
			continue;
		}

		for (Channel chan : arch.getChannels()) {
			if (chan.getID().equals(related)) {
				toRemove.add(xmiModel.getEobjectIDREF(chan));
				continue relatedsLoop;
			}
		}
	}

	return toRemove;
}
```

Básicamente lo que hace este código es buscar en el Modelo de Arquitectura los elementos seleccionados en el diálogo de la Ilustración 24, extraer el id de referencia de cada uno de ellos del modelo XMI y después eliminarlos utilizando este id de referencia.

Una vez eliminados los elementos de la arquitectura que el ingeniero haya seleccionado, habrá que eliminar todos los enlaces en los que apareciera la característica eliminada. Esto ha de hacerse independientemente de si se elimina el elemento del enlace en la arquitectura o no, ya que no pueden quedar elementos que la referencien.

Utilizando de nuevo el ejemplo de la Ilustración 23, pueden darse dos casos:

1. Se elimina el elemento de la arquitectura enlazado, en este caso GPS:

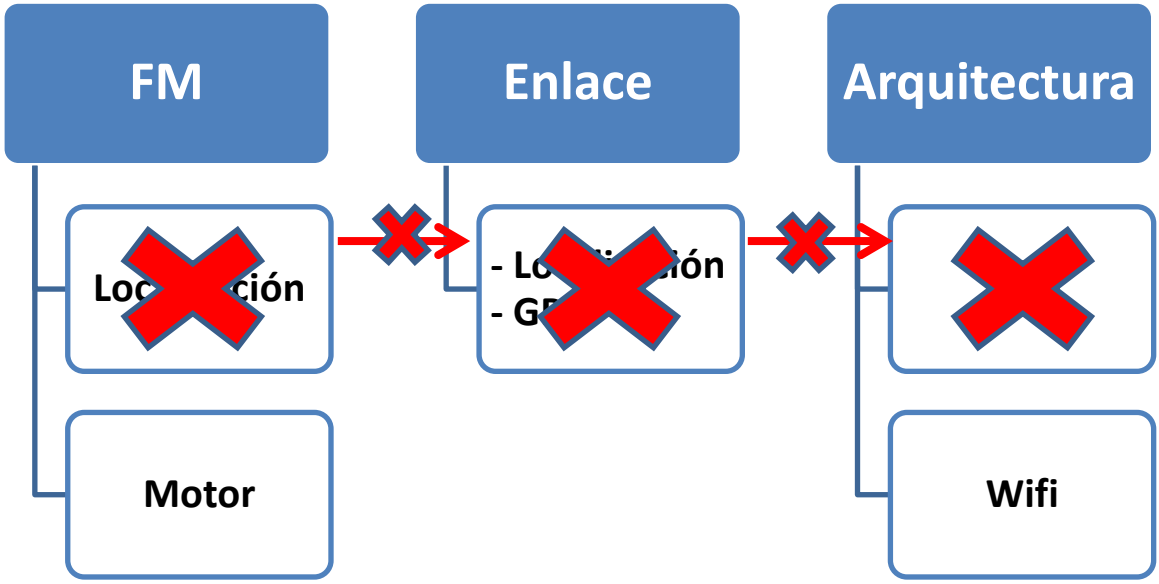

ILUSTRACIÓN 25 ELIMINACIÓN DE ELEMENTO ARQUITECTÓNICO

2. No se selecciona el elemento GPS de la arquitectura para su eliminación. Sin embargo, habrá que eliminar el enlace para que no exista un elemento que referencie a la característica "Localización" una vez eliminada:

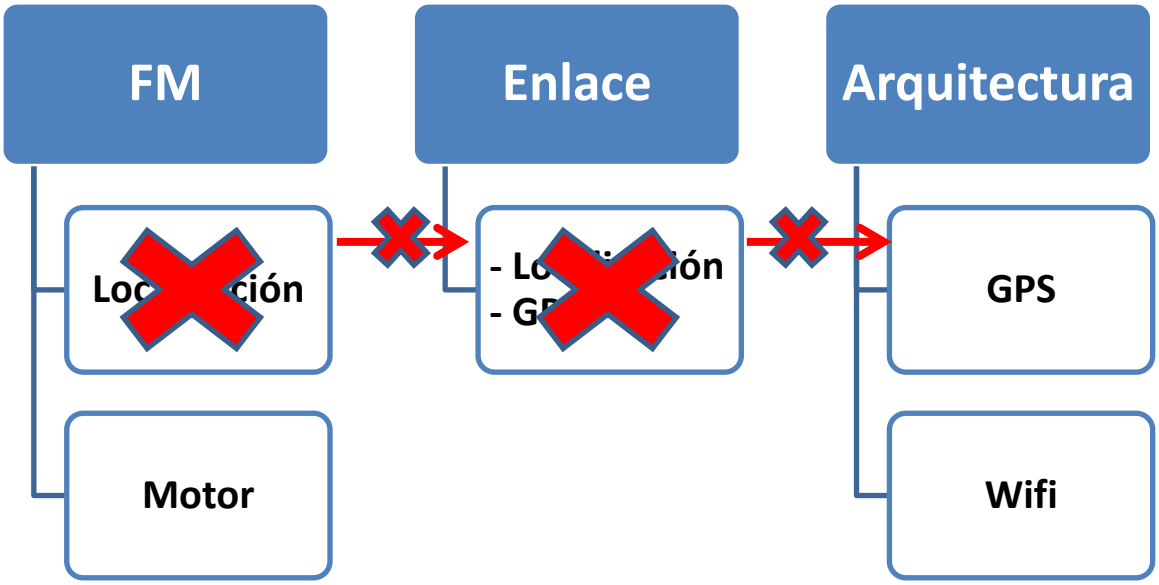

ILUSTRACIÓN 26 CONSERVACIÓN DEL ELEMENTO ARQUITECTÓNICO

Para eliminar los enlaces se utiliza el siguiente código:

```
public void removeRelatedLinks(WModel wModel, EObject removed) {
```

```
		if(wModel == null){
			wModel = this.wModel;
		}

		if(removed == null){
			removed = (EObject) notification.getOldValue();
		}

		@SuppressWarnings("unchecked")
		EList<Link> links = wModel.getOwnedElement();

		String id = ((NamedElement) removed).getName();

		for (int i = 0; i < links.size(); i++) {
			if (links.get(i) instanceof Link) {
				String name = links.get(i).getLeft().getName();

				if (name.equals(id)) {
					wModel.getOwnedElement().remove(links.get(i));
				}
			}
		}

		try {
			wModel.eResource().save(Collections.EMPTY_MAP);
		} catch (IOException e) {
			e.printStackTrace();
		}

		return;
}
```

Lo que se hace es, a partir del wModel, obtener la lista de enlaces que contiene y buscar dentro de ellos el id de la característica eliminada. En caso de que se encuentre algún enlace, se elimina de la lista de enlaces (ownedElement). Una vez eliminados todos los enlaces, se guarda el wModel.

De esta forma, habremos eliminado la característica, **todos** los elementos de la arquitectura enlazados a la misma y que el ingeniero **considere** que debían ser eliminados y **todos los enlaces** que usasen la característica.

Una vez completado este paso, habremos eliminado tanto las **resoluciones asociadas** a la característica eliminada como los **elementos de arquitecturas** escogidos por el ingeniero y todos los **enlaces** que referenciasen a la característica.

Hecho esto, habremos dejado el sistema en un estado estable y sin inconsistencias, eliminando posibles referencias a la característica ya eliminada y habiendo automatizado el proceso de forma que el ingeniero intervenga únicamente para escoger los elementos de la arquitectura a eliminar.

5.3.2.2 IMPLEMENTACIÓN DE ADICIÓN DE FUNCIONALIDADES

Para implementar el patrón de adición de funcionalidades, al igual que con el anterior patrón, se ha un escuchador (listener) que detecte cada ocasión en la que se añada una característica al Modelo de Características. Cuando se detecte, habrá que lanzar el patrón de adición de funcionalidades.

Para ello, se ha extendido el escuchador "FeatureAdapter" de forma que además de gestionar las notificaciones de tipo "Remove", se gestionen también las de tipo "Add".

Así como el escuchador y se ha añadido a todos los objetos de tipo "FeatureModel" como se comentaba en el punto anterior, no se ha hecho nada más que ampliar el código del adaptador para que gestione este tipo de notificaciones.

Para ello se ha desarrollado el código para que cuando recibamos una notificación de elemento añadido al Modelo de Características, se comprueba si el elemento que se añade es una característica y en caso de que sí, lanzar el patrón. Esta comprobación se hace para evitar que se lance el patrón al añadir atributos, relaciones entre características, etc.

A continuación se expone el código:

```
@Override
public void notifyChanged(Notification notification) {

	Object notifier = notification.getNotifier();

	if (filter.matches(notification)
			&& notifier != null
			&& ((Resource.Internal) ((EObject) notifier).eResource()) !=
null
			&& !((Resource.Internal) ((EObject) notifier).eResource())
					.isLoading()) {

		IWorkspace root = ResourcesPlugin.getWorkspace();

		try {
			root.getRoot().refreshLocal(IResource.DEPTH_INFINITE, null);
		} catch (CoreException e) {
			e.printStackTrace();
		}

		addAdapter = new AddAdapter(notification);
		removeAdapter = new RemoveAdapter(notification);

		switch (notification.getEventType()) {
		case Notification.REMOVE:
			handleRemove(notification);
			break;

		case Notification.ADD:
			handleAdd(notification);
```

```
                        break;

                case Notification.REMOVE_MANY:
                        handleRemoveMany(notification);
                        break;
                }
        }
}

protected void handleAdd(Notification notification) {

        if (notification.getNewValue() instanceof Feature) {

                addAdapter.handleAddFeature();
        }
}
```

El método "notifyChanged" es idéntico al del punto anterior, sólo que ahora el caso que nos interesa es el "Notification.ADD" y el método "handleAdd". Este método llama a un objeto llamado "addAdapter" que al igual que en el punto anterior, es un objeto de tipo "AddAdapter" que encapsula todas las funcionalidades necesarias para llevar a cabo el patrón.

Este objeto en el método "handleAddFeature" lanzará un asistente (wizard) de tipo "AddWizard" que será el que vaya guiando al ingeniero en las tareas de las que necesite su intervención. Este es el código que lo realiza:

```
public AddAdapter(Notification notification) {
        adapterHelper = new AdapterHelper();

        this.notification = notification;
}

protected void handleAddFeature() {
        WizardDialog wizardDialog = new WizardDialog(PlatformUI.getWorkbench()
                        .getActiveWorkbenchWindow().getShell(), new
AddWizard(this));

        wizardDialog.open();

}
```

Al igual que en el punto anterior también, no se expondrá todo el código del asistente. Éste consiste básicamente de tres pasos.

1. **Introducir nombre de la característica**: se pide el nombre de la característica añadida al ingeniero.

2. **Crear resolución para la característica** (opcional): se pide al ingeniero qué Modelo de Resoluciones desea utilizar para crear una para la nueva característica. Será opcional, porque no siempre es necesario añadir nuevas resoluciones.

a. En caso de que se seleccione el Modelo de Resoluciones, se permitirá crear una resolución para la nueva característica.

3. **Crear / enlazar elementos de la arquitectura** (opcional): se permite al ingeniero crear nuevos elementos de la arquitectura para la nueva característica o enlazar los ya existentes con la misma. Este paso también es opcional, ya que no siempre será necesario un elemento de arquitectura.

Estos tres pasos se convierten en más páginas del asistente, pero conceptualmente estos son los pasos que se realizan. A continuación se entra en el detalle de cada uno de ellos.

5.3.2.2.1 PASO 1: INTRODUCIR NOMBRE DE LA CARACTERÍSTICA

En el primer paso se solicitará al ingeniero que introduzca el nombre de la nueva característica, para poder utilizarlo en los siguientes pasos. Para ello se muestra el siguiente diálogo:

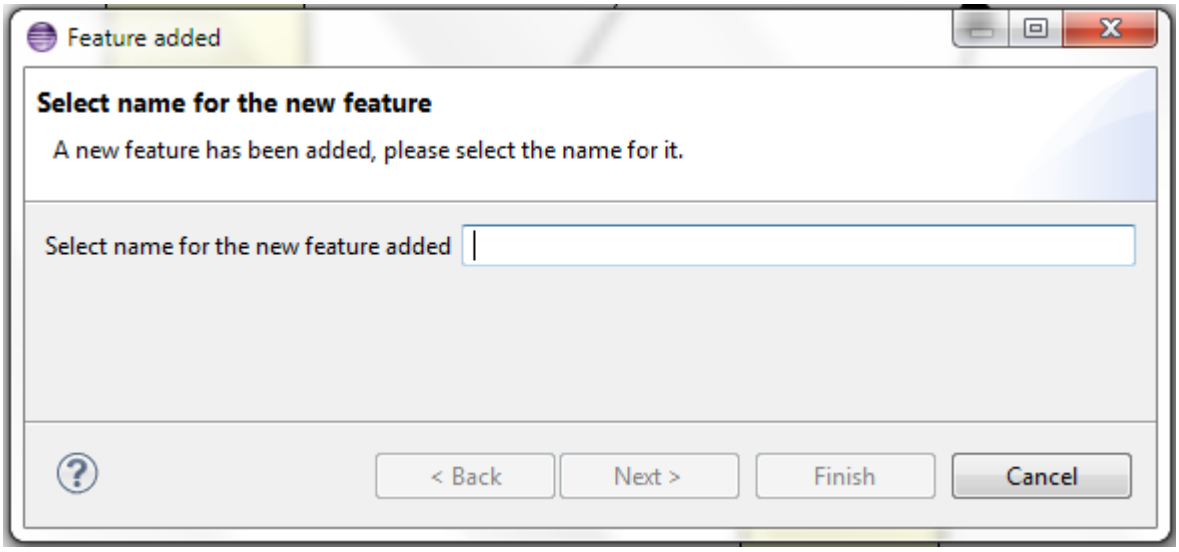

ILUSTRACIÓN 27 INTRODUCCIÓN NOMBRE DE NUEVA CARACTERÍSTICA

En la Ilustración 27 se muestra la página del asistente en la que introduciremos el nombre de la nueva característica.

Sólo cuando la caja de texto del nombre tenga algún carácter será cuando se habiliten los botones de siguiente y finalizar. Se habilitará también el botón de finalizar porque los siguientes pasos son opcionales. Es por ello que una vez introducido el nombre para la característica, ya podríamos finalizar el asistente y crearla.

5.3.2.2.2 PASO 2: CREAR RESOLUCIÓN PARA LA CARACTERÍSTICA

En este paso se solicitará al ingeniero que introduzca el Modelo de Resoluciones a utilizar:

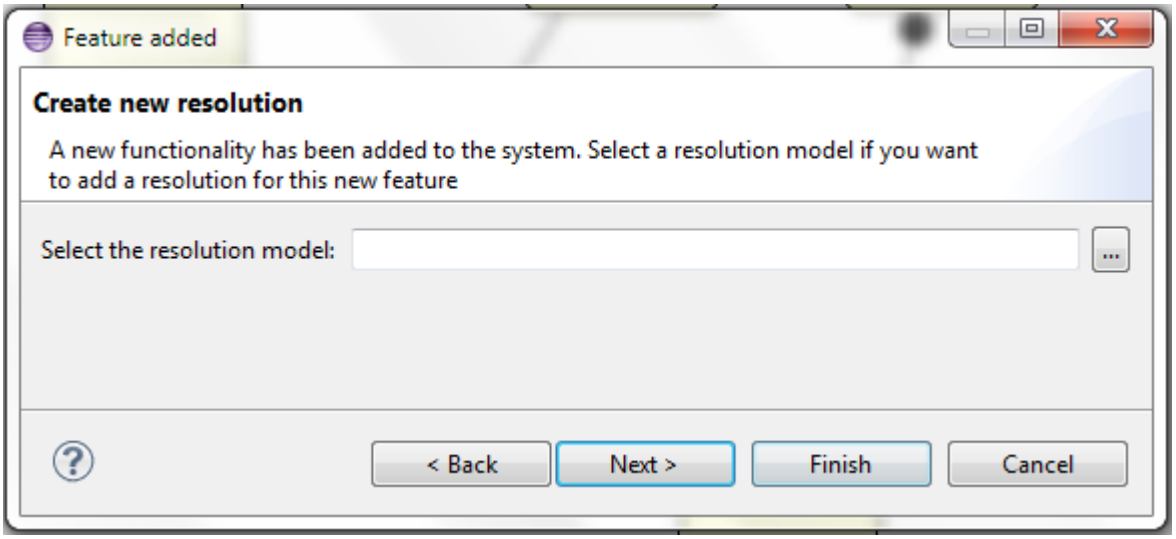

ILUSTRACIÓN 28 ADICIÓN DE FUNCIONALIDAD: SELECCIÓN DE MODELO DE RESOLUCIÓN

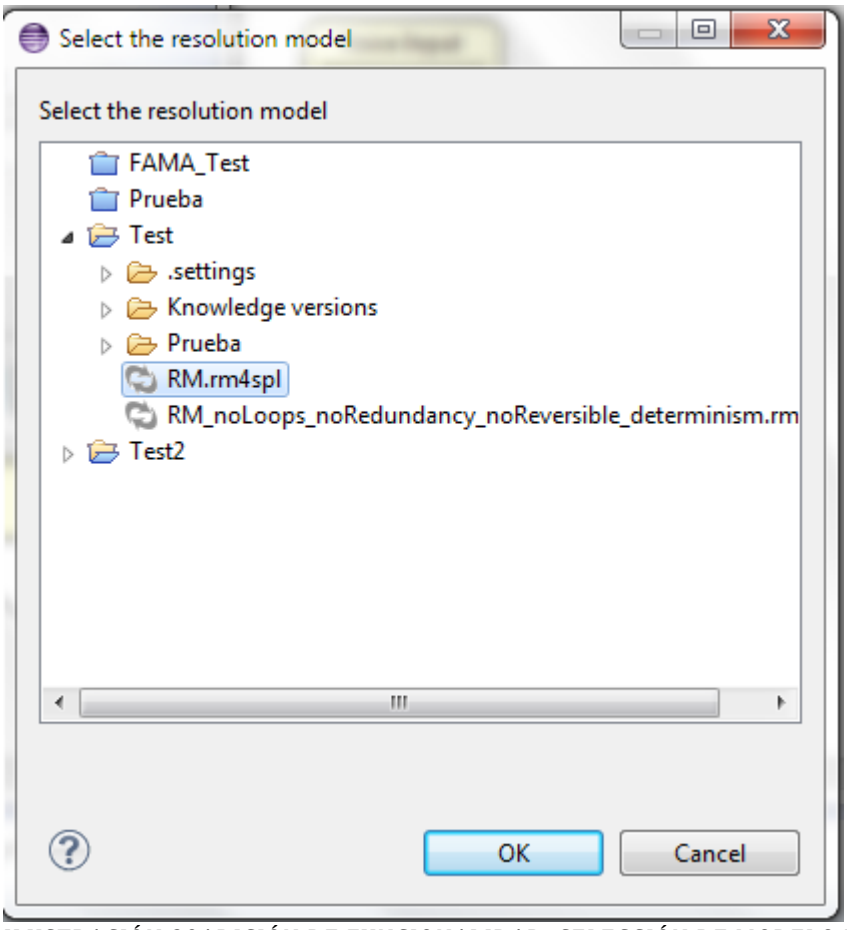

ILUSTRACIÓN 29ADICIÓN DE FUNCIONALIDAD: SELECCIÓN DE MODELO DE RESOLUCIÓN

Como se ve en la Ilustración 28 los botones de “Siguiente” y “Finalizar” están activos pese a no haber seleccionado el Modelo de Resoluciones a utilizar. Esto es porque, como se ha dicho previamente, este paso es opcional y puede pasarse al siguiente o finalizar el asistente.

En caso de que se seleccione un Modelo de Resoluciones, el asistente nos llevará a la siguiente página:

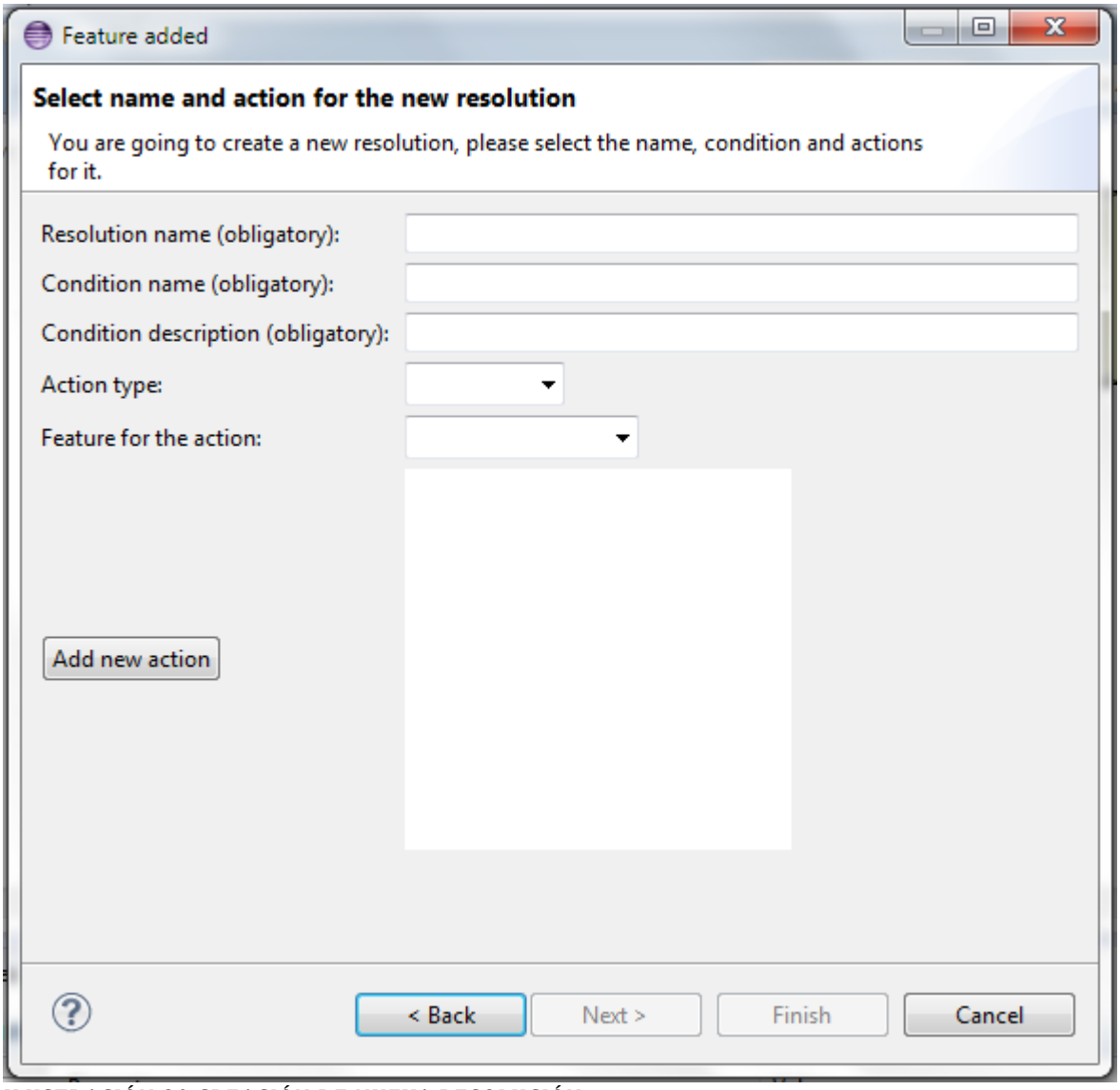

ILUSTRACIÓN 30 CREACIÓN DE NUEVA RESOLUCIÓN

Esta ilustración muestra la página del asistente en la que se crea una nueva resolución para la nueva característica añadida.

Los botones de siguiente y finalizar no se activan hasta que se introduzcan los campos obligatorios: nombre de la resolución, nombre de la condición y descripción de la condición.

La acción no es obligatoria, ya que en caso de que no se introduzca nada se generará automáticamente una acción de activación para la característica que se está añadiendo actualmente.

En cualquier caso, se pueden añadir varias acciones en esta resolución como ilustra la siguiente imagen:

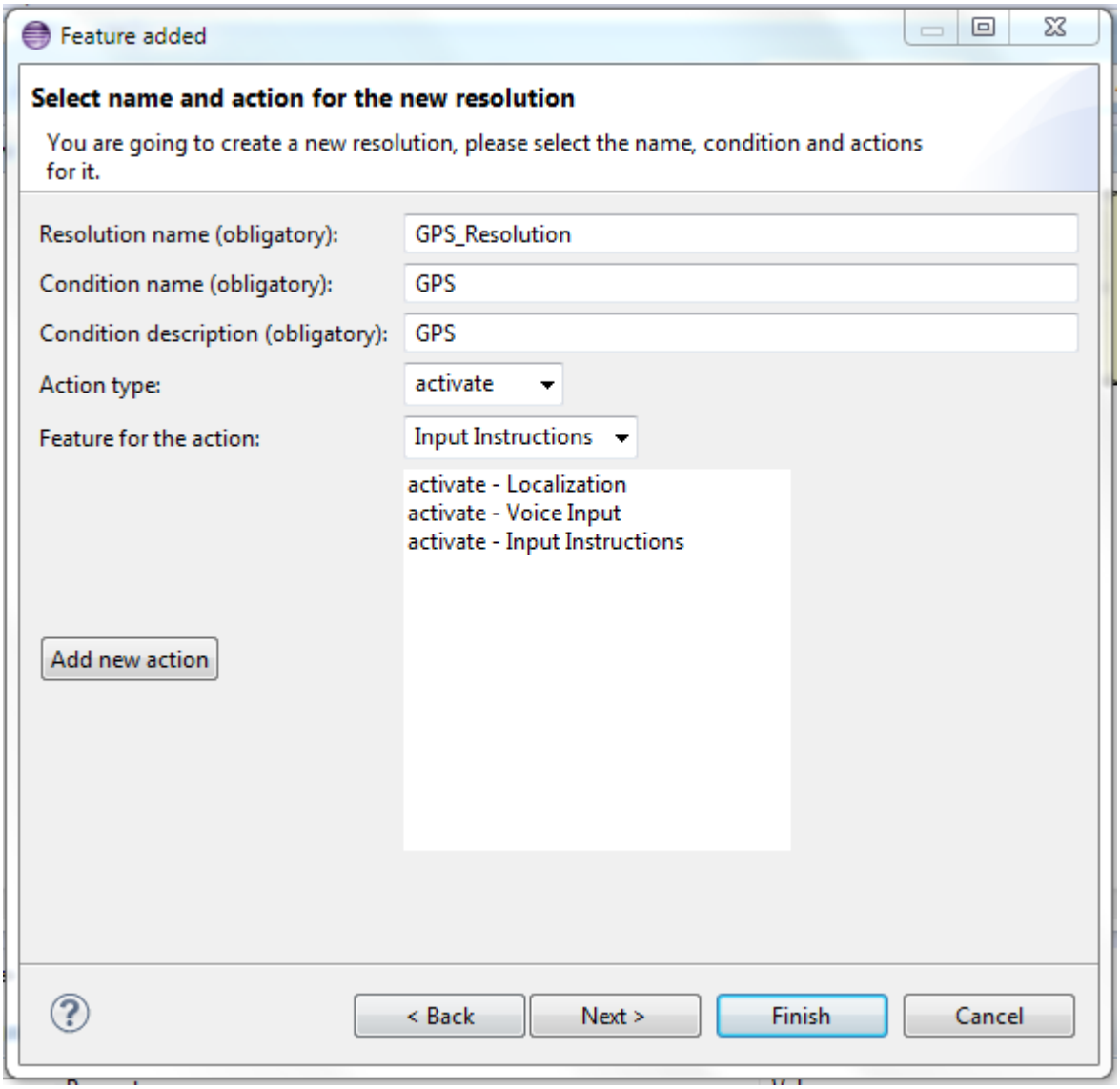

ILUSTRACIÓN 31 EJEMPLO CREACIÓN DE NUEVA RESOLUCIÓN

En esta ilustración se muestra un ejemplo de la creación de una nueva resolución con todos sus campos rellenados.

Una vez introducidos todos los campos, se añade la resolución con los datos introducidos. Para ello en la clase "AddAdapter" encontramos este método:

```
public void addResolution(EObject resolutionModel, String name, String
conditionName,
            String descriptionName, Map<Feature, String> actions) {
      if(resolutionModel != null){
            Notification aux = new NotificationImpl(Notification.ADD,
                        notification.getOldValue(),
notification.getNewValue());
```

```
            NotificationExtended notificationResolution = new
NotificationExtended(
                        resolutionModel, aux);

            notificationResolution.addExtendedData("Name", name);
            notificationResolution.addExtendedData("ConditionName", name);
            notificationResolution.addExtendedData("ConditionDescription",
                        descriptionName);

            if (actions != null) {
                  notificationResolution.addExtendedData("Actions", actions);
            }

            resolutionModel.eAdapters().get(0)
                        .notifyChanged(notificationResolution);
      }
}
```

Siguiendo el estilo del patrón de eliminar funcionalidad, en este caso lo que se ha hecho es algo muy similar. Se notifica al modelo de resolución de que se va a añadir una resolución al modelo y mediante el uso de la clase "NotificationExtended" que tiene un diccionario de datos adicionales, se le indican los campos introducidos en el asistente. Este es el código de la clase NotificationExtended:

```
public class NotificationExtended extends NotificationWrapper {

      private Map<String, Object> extendedData;

      public NotificationExtended(Notification notification) {
            super(notification);

            extendedData = new HashMap<String, Object>();
      }

      public NotificationExtended(Object notifier, Notification notification) {
            super(notifier, notification);

            extendedData = new HashMap<String, Object>();
      }

      public Object addExtendedData(String key, Object value){
            return extendedData.put(key, value);
      }

      public Object getExtendedData(String key){
            return extendedData.get(key);
      }

      public boolean containsExtendedData(String key){
            return extendedData.containsKey(key);
      }
}
```

Mediante esta clase se puede enviar una notificación con datos adicionales, que es justo lo que se necesita para notificar la creación de una resolución indicándole el nombre, condición, acciones, etc.

Una vez notificado el modelo, el "ResolutionAdapter" (el escuchador de los modelos de resolución) llama al método "handleAdd". Este método es llamado, porque como hemos visto en el "FeatureAdapter" se llamaba al mismo cada vez que se detectaba una notificación de tipo "Add" en el método "notifyChanged".

De esta forma lo que se ha hecho es que el "ResolutionAdapter" extienda al "FeatureAdapter", sobreescribiendo el método "handleAdd". Así se tendrá un comportamiento propio del Modelo de Resoluciones cuando se detecten notifcaciones de tipo "Add". Este es el código:

```
public class ResolutionAdapter extends FeatureAdapter {

@Override
protected void handleAdd(Notification notification) {
	if (notification.getNewValue() instanceof Feature
			&& notification instanceof NotificationExtended) {
		ResolutionModel resolutionModel = (ResolutionModel) notification
				.getNotifier();

		Feature newFeature = (Feature) notification.getNewValue();

		NotificationExtended extended = (NotificationExtended)
notification;

		Resolution resolution = buildResolutionFromNotification(extended);

		resolution.setCondition(buildConditionFromNotification(extended));

		List<Action> actions = buildActionsFromNotification(extended);

		if(actions.size() > 0){
			resolution.getActions().addAll(actions);
		}

		resolutionModel.getResolutions().add(resolution);

		try {
			resolutionModel.eResource().save(Collections.EMPTY_MAP);
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}

private Resolution buildResolutionFromNotification(
		NotificationExtended notification) {

	Resolution result = factory.createResolution();

	if (notification.containsExtendedData("Name")) {
```

```
        String name = notification.getExtendedData("Name").toString();

                result.setId(name);
        } else {
                result.setId(((Feature) notification.getNewValue()).getName());
        }

        return result;
}

private Condition buildConditionFromNotification(
                NotificationExtended notification) {
        Condition result = factory.createCondition();

        if (notification.containsExtendedData("ConditionName")) {
        String name = notification.getExtendedData("ConditionName")
                        .toString();

        result.setName(name);
}
```

Como se puede ver, los métodos "buildXXX" simplemente recogen los parámetros de la notificación extendida y construyen la resolución, la condición y las acciones a partir de los mismos.

Una vez creados los objetos a partir de los parámetros de la notificación, se añaden al modelo de resolución y se guarda, de forma que ya habremos creado la resolución como hemos indicado en el asistente.

De esta forma, se habrá acabado el paso de añadir una resolución para la característica añadida.

5.3.2.2.3 PASO 3: CREAR / ENLAZAR ELEMENTOS DE ARQUITECTURA

En este paso se mostrará primero una página donde el ingeniero elegirá si quiere crear nuevos elementos de arquitectura para la característica añadida o desea escoger de entre los ya existentes:

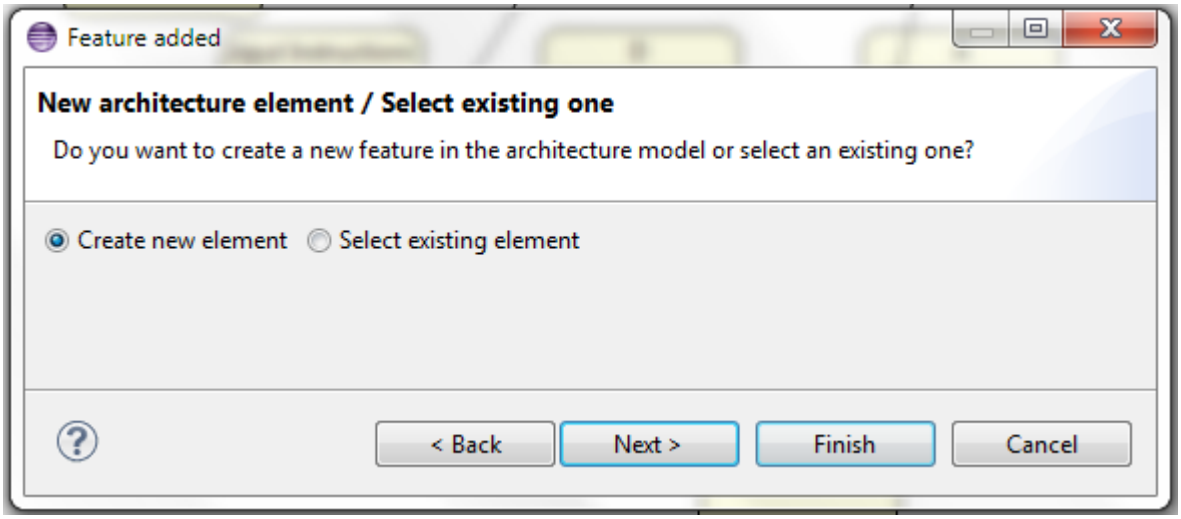

ILUSTRACIÓN 32 CREAR NUEVO / SELECCIONAR ELEMENTO DE LA ARQUITECTURA

En esta ilustración se muestra la página del asistente donde el ingeniero escoge entre crear un nuevo elemento de arquitectura o seleccionar uno ya existente.

5.3.2.2.3.1 PASO 3.1: CREAR ELEMENTOS DE ARQUITECTURA

En caso de que el ingeniero seleccione "Crear nuevo elemento" se le mostrará la siguiente página:

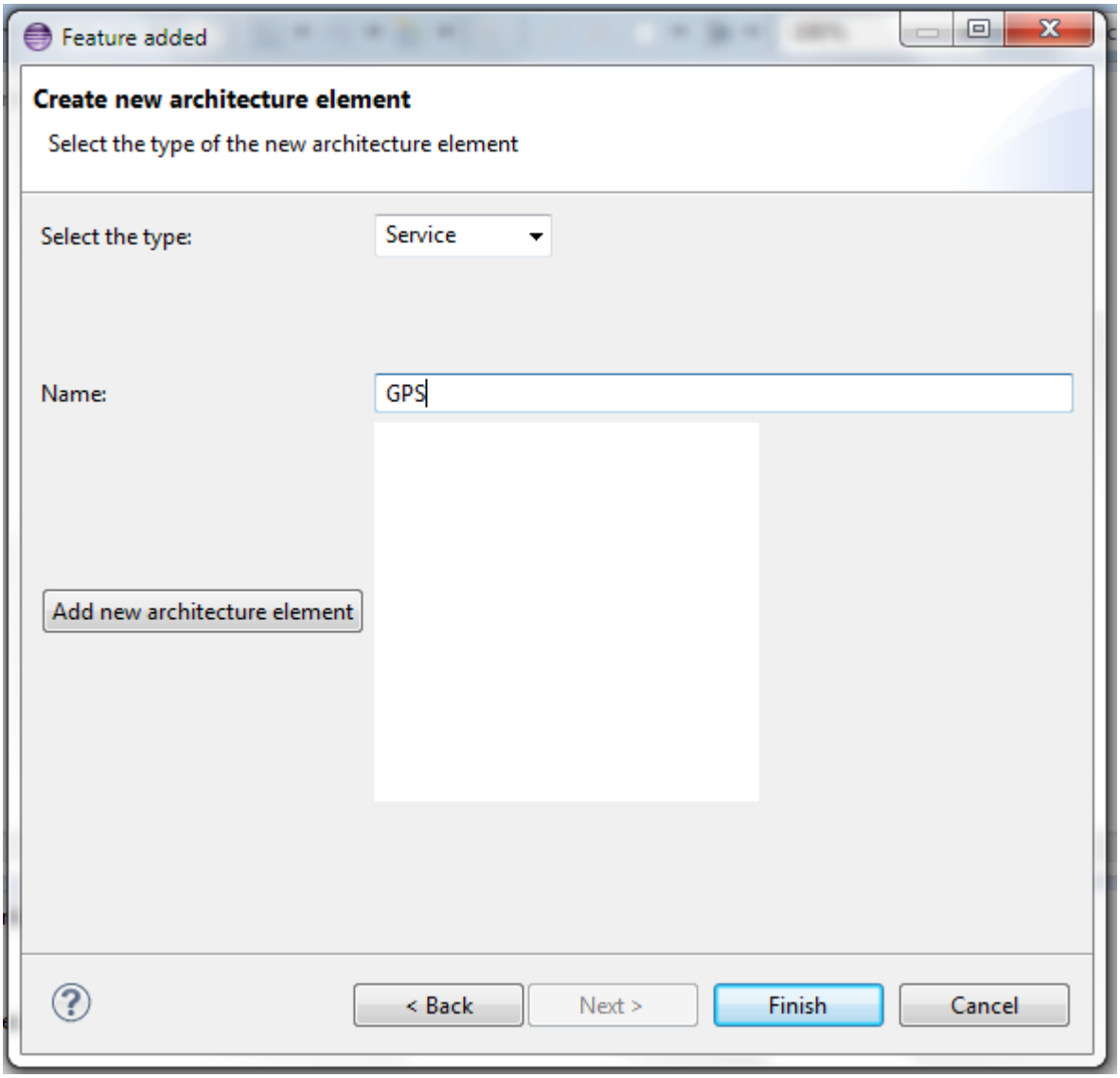

ILUSTRACIÓN 33 CREACIÓN DE NUEVO ELEMENTO DE ARQUITECTURA

Esta ilustración muestra la página del asistente donde el ingeniero crea un nuevo elemento de arquitectura para la nueva característica.

Como se puede ver, se puede seleccionar el tipo de componente (se podrá escoger entre "Componente", "Canal" y "Servicio") y su nombre. Hay un caso especial y es el que el componente sea de tipo "Canal". En tal caso se tendrá que permitir escoger el productor y el consumidor del mismo, como se ve a continuación:

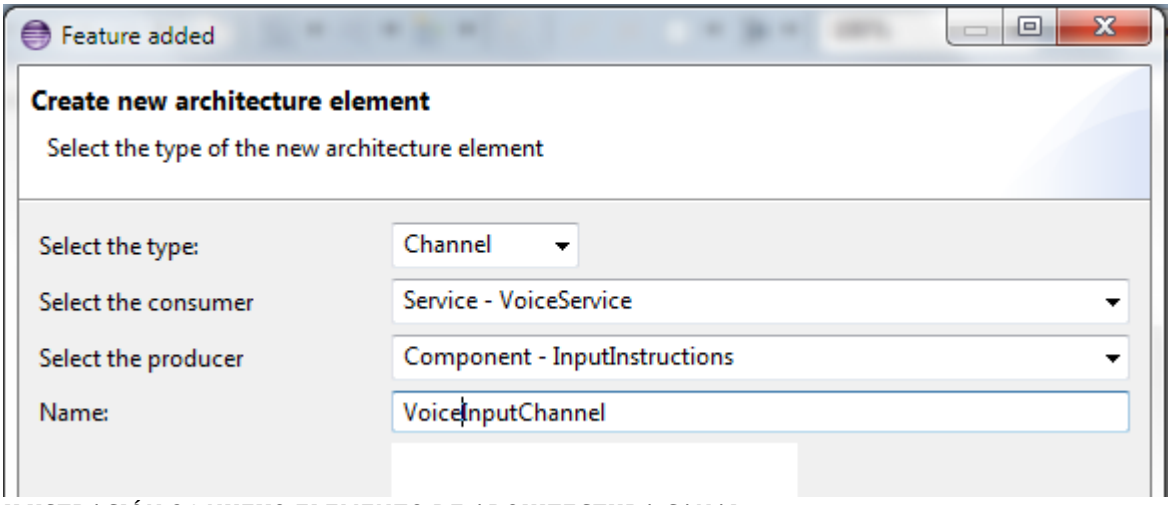

ILUSTRACIÓN 34 NUEVO ELEMENTO DE ARQUITECTURA CANAL

Esta ilustración muestra la página donde el ingeniero añade un nuevo elemento a la arquitectura en caso de que escoja como tipo "Canal", donde tendrá que escoger el consumidor y el productor del mismo.

Una vez se pulse el botón de añadir elemento de arquitectura, se creará el elemento. Para ello se utiliza el siguiente código de la clase "AddAdapter":

```
public void createNewArchitectureElement(String type, String name,
            String producer, String consumer) {

      if (wModel != null) {
            XmiModel rightModel = adapterHelper.getRightModel(wModel);

            EObject rightRefToAdd = null;

            Architecture architecture = adapterHelper
                        .readArchitecture(rightModel.resource.getURI()
                                    .toFileString());

            PervMLArchitectureFactory factory = new
PervMLArchitectureFactoryImpl();

            if (type.equals("Component")) {

                  Component comp = factory.createComponent();

                  comp.setID(name);

                  architecture.getComponents().add(comp);

                  rightRefToAdd = comp;
            }

            else if (type.equals("Channel")) {
                  Channel chan = factory.createChannel();

                  chan.setID(name);
```

```
                        chan.setConsumer(getComponentByName(architecture,
consumer));
                        chan.setProducer(getComponentByName(architecture,
producer));

                        architecture.getChannels().add(chan);

                        rightRefToAdd = chan;
                }

                else if (type.equals("Service")) {

                        Service service = factory.createService();

                        service.setID(name);

                        architecture.getComponents().add(service);

                        rightRefToAdd = service;
                }

                try {
                        architecture.eResource().save(Collections.EMPTY_MAP);

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

                addLinkFromArchitecture(rightRefToAdd);
        }
}
```

Como se puede ver, lo que se hace es crear un elemento en función del tipo que se le indique (el seleccionado en el asistente) y con el nombre introducido. Una vez creado el elemento del tipo correspondiente, se añade al Modelo de Arquitectura y se guarda el mismo.

Una vez añadido el elemento, hay que añadir un enlace entre el nuevo elemento y la característica añadida. Esta funcionalidad la realiza el método "addLinkFromArchitecture":

```
private void addLinkFromArchitecture(EObject elementToLink){
        XmiModel rightModel = adapterHelper.getRightModel(wModel);

        TreeIterator<EObject> features = rightModel.getAllFeatures();

        while (features.hasNext()) {
                EObject next = features.next();

                String id = getId(next);

                if (getId(elementToLink).equals(id)) {
                        elementToLink = next;
                }
        }
```

```
		Feature newFeature = (Feature) notification.getNewValue();

		addLinkInWeaverModel(newFeature,
				adapterHelper.getLeftModel(wModel), rightModel, wModel,
				elementToLink);
	}

	private void addLinkInWeaverModel(Feature newFeature, XmiModel leftModel,
			XmiModel rightModel, WModel wModel, EObject rightRefToAdd) {
		Mw_base_extFactoryImpl linksFactory = new Mw_base_extFactoryImpl();

		Link link = linksFactory.createLink();

		LinkEnd leftLink = linksFactory.createLinkEnd();

		ElementRef leftRef = createElementRef(wModel, leftModel, newFeature,
				newFeature.getName(), false);

		leftLink.setElement(leftRef);
		leftLink.setModel(wModel);
		leftLink.setName(newFeature.getName());

		link.setLeft(leftLink);
		link.setModel(wModel);

		LinkEnd rightLink = linksFactory.createLinkEnd();

		ElementRef rightRef = createElementRef(wModel, rightModel,
				rightRefToAdd, rightModel.getEobjectHREF(rightRefToAdd),
	true);

		rightLink.setElement(rightRef);
		rightLink.setModel(wModel);

		link.setRight(rightLink);

		wModel.getOwnedElement().add(link);

		try {
			wModel.eResource().save(Collections.EMPTY_MAP);
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
```

A pesar de que el código es bastante extenso, es muy sencillo de resumir. Simplemente se toma el elemento de arquitectura recién creado y se crea un enlace en el que el elemento izquierdo será la característica añadida y el izquierdo será el elemento arquitectónico.

Una vez creado, se añade al modelo de enlaces y se guarda, de forma que ya tendremos nuestro nuevo elemento enlazado con la nueva característica.

5.3.2.2.3.2 PASO 3.2: ENLAZAR ELEMENTOS DE ARQUITECTURA

En caso de que el ingeniero no desee crear nuevos elementos de arquitectura para la nueva característica si no seleccionar alguno de los ya existentes, se le mostrará la siguiente página del asistente:

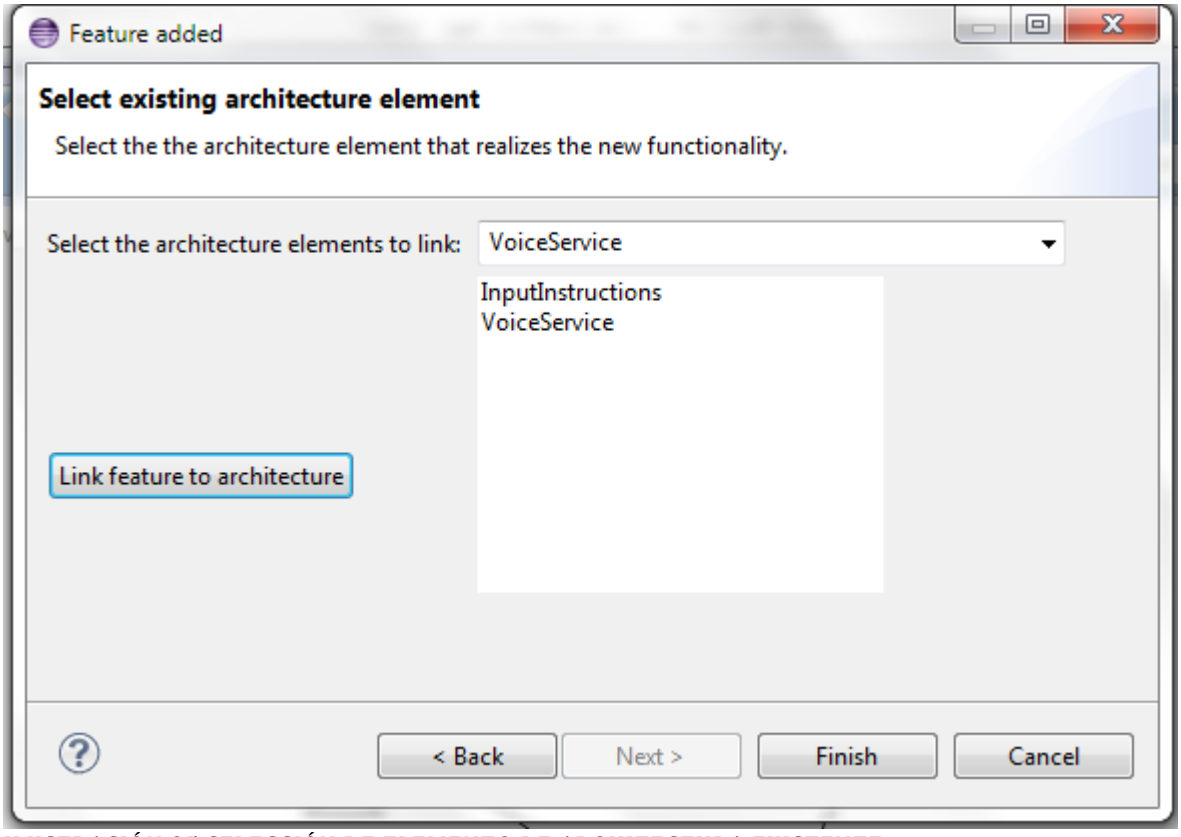

ILUSTRACIÓN 35 SELECCIÓN DE ELEMENTO DE ARQUITECTURA EXISTENTE

En esta página el ingeniero escoge qué elementos se deberán enlazar a la característica añadida para que sean estos elementos los que la realicen.

En ella se podrán seleccionar de entre los elementos ya existentes en la arquitectura, cuáles de ellos queremos enlazar con la nueva característica. En este ejemplo concreto, hemos enlazado la nueva característica con los servicios "InputInstructions" y "VoiceService".

Cada vez que se pulse en el botón "Enlazar característica" se creará un enlace entre la nueva característica y el elemento seleccionado. Para ello se hace uso del método "addLink" de la clase "AddAdapter":

```
public void addLink(String architectureElement) {

	XmiModel rightModel = adapterHelper.getRightModel(wModel);

	Feature newFeature = (Feature) notification.getNewValue();

	TreeIterator<EObject> features = rightModel.getAllFeatures();
```

```
        EObject rightRefToAdd = null;

        while (features.hasNext()) {
                EObject next = features.next();

                String id = getId(next);

                if (architectureElement.equals(id)) {
                        rightRefToAdd = next;
                }
        }

        addLinkInWeaverModel(newFeature, adapterHelper.getLeftModel(wModel),
                        rightModel, wModel, rightRefToAdd);
}
```

Lo que se hace es buscar en el Modelo de Arquitectura el elemento seleccionado y añadir un enlace en el modelo de enlaces mediante el uso del método "addLinkInWeaverModel". Este método, como se ha comentado previamente, simplemente crea un enlace entre los dos elementos que se indiquen y almacena el modelo de enlaces.

Una vez llegados a este punto, ya habremos creado la resolución para la nueva característica (si lo hemos creído oportuno) y habremos añadido un nuevo elemento arquitectónico o enlazado con uno ya existente la característica.

De este modo, habremos concluido el patrón de adición de funcionalidades automatizando esta tarea de forma que suponga para el ingeniero una serie de pasos donde tendrá que rellenar unos campos preestablecidos y validados en caso de que quiera añadir elementos que referencien a la nueva característica.

Como comentario, es muy obvio que tanto para este patrón como para el patrón de "Eliminación de funcionalidad" se ha seguido un esquema muy similar en ambos casos:

1. Detectar cambios en el modelo mediante el escuchador "FeatureAdapter"

2. Gestionar el cambio en función de su tipo:

    a. En caso de ser una eliminación, llamar al handleRemoveFeature para que se lance un asistente "RemoveWizard".

    b. En caso de ser una adición, llamar al handleAddFeature para que se lance un asistente "AddWizard"

3. Seguir una serie de pasos del asistente, validando y automatizándolos en todo lo posible.

Así, han quedado implementados los dos patrones de evolución del sistema.

5.3.3 IMPLEMENTACIÓN DE LA GENERACIÓN DEL PLAN DE EVOLUCIÓN

Una vez realizada la evolución del sistema y aplicados los patrones de evolución, necesitaremos un **plan de evolución** que contenga todos los cambios realizados en el sistema durante la misma.

Para ello se utiliza una copia del conocimiento del sistema realizada previamente. A partir de ella se comparan cada uno de los modelos del sistema en su estado actual con los modelos de la copia del conocimiento.

Para ello, se ha creado una nueva acción contextual (al igual que en 5.3.1 Implementación de almacenamiento del conocimiento del sistema) llamada "Evolve Evolution Plan".

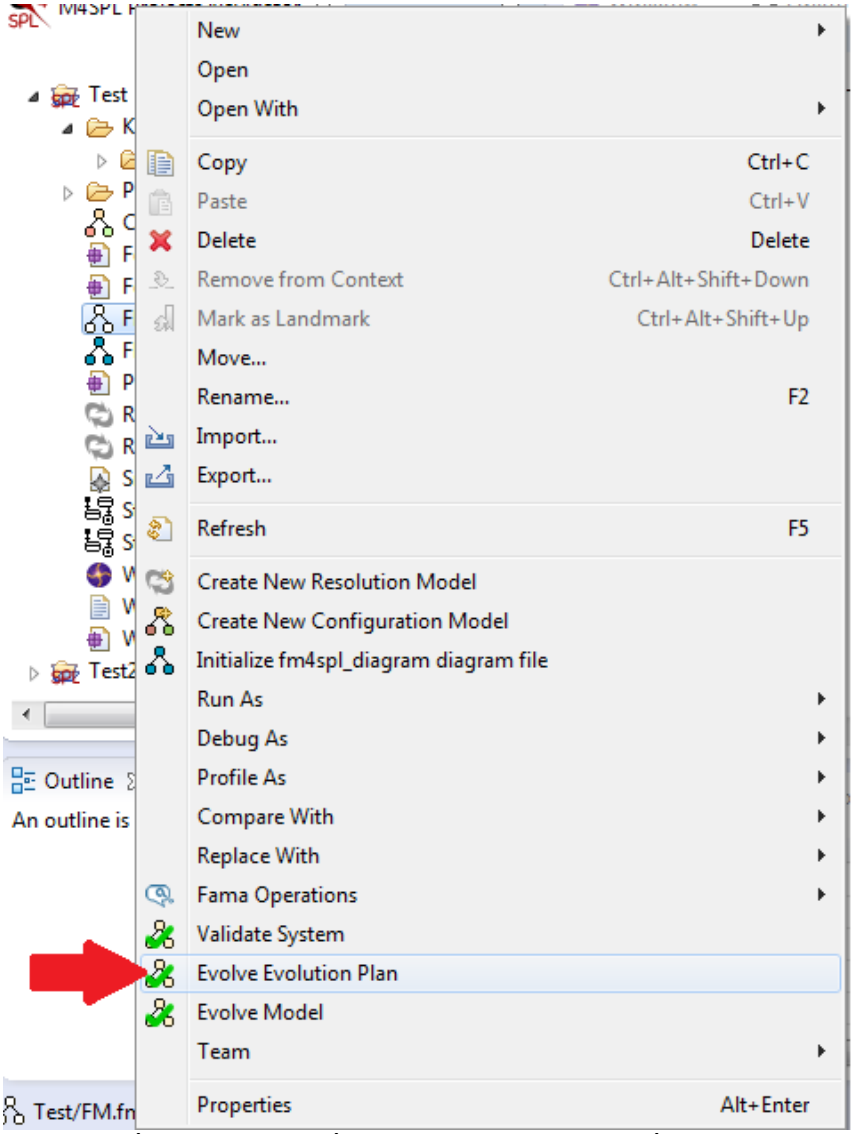

ILUSTRACIÓN 36 GENERACIÓN DE PLANES DE EVOLUCIÓN

Esta será la acción contextual para generar los planes de evolución.

Al igual que en el caso del almacenamiento del conocimiento, será una acción aplicable sobre los modelos de características (fm4spl) y también se ha añadido al proyecto FAMA.Reasoner. Este es un extracto del xml para añadirla:

```
   <extension
         id="FAMA_REASONING_POPUP"
         name="Fama Reasoning"
         point="org.eclipse.ui.popupMenus">
      <objectContribution
            adaptable="true"
            id="es.upv.pros.M4SPL_Reasoning.FAMA_Reasoner.operations"
            nameFilter="*fm4spl"
            objectClass="org.eclipse.core.resources.IFile">
         <action
class="es.upv.pros.m4spl_reasoning.fama_reasoner.actions.GenerateEvolutionPlanAc
tion"
               icon="icons/validateModelIcon.gif"

id="es.upv.pros.M4SPL_Reasoning.FAMA.Reasoner.generateevolutionplan"
               label="Evolve Evolution Plan"

menubarPath="es.upv.pros.M4SPL_Reasoning.FAMA_Reasoner.operations">
         </action>
```

Una vez el ingeniero solicita la generación de un plan de evolución, se le solicita qué versión del conocimiento del sistema quiere utilizar para comparar con el estado actual del mismo. Para ello se le mostrará el siguiente diálogo:

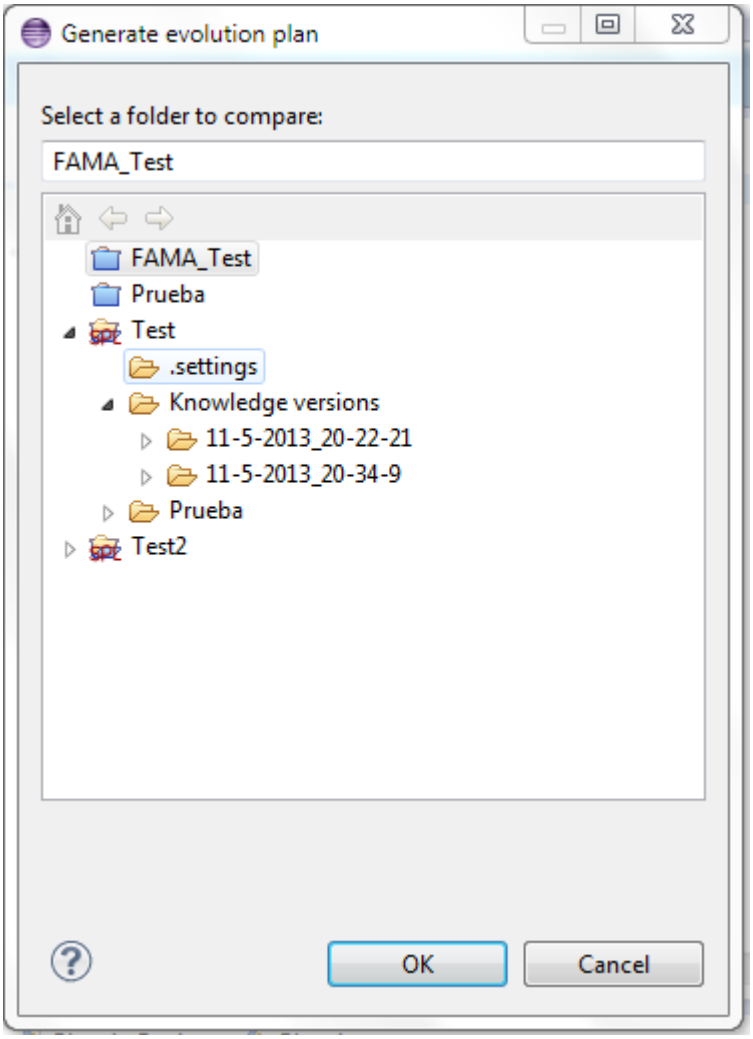

ILUSTRACIÓN 37 SELECCIÓN DE VERSIÓN DEL CONOCIMIENTO DEL SISTEMA

En este diálogo el ingeniero seleccionará que versión del conocimiento del sistema se utilizará para generar el plan de evolución.

Un diálogo que se muestra ejecutando el siguiente código:

```
private IFolder showSelectVersionFolder(IFile selectedFile) {

	IFolder folderBackup = null;

	Shell shell = PlatformUI.getWorkbench().getActiveWorkbenchWindow()
			.getShell();

	ContainerSelectionDialog dialog = new ContainerSelectionDialog(shell,
			null, true, "Select a folder to compare:");
	dialog.setTitle("Generate evolution plan");
	dialog.open();

	if (dialog.getResult() != null && dialog.getResult().length > 0
			&& dialog.getResult()[0] instanceof Path) {
```

```
            Path path = (Path) dialog.getResult()[0];

            folderBackup = selectedFile.getWorkspace().getRoot()
                        .getFolder(path);
      }

      return folderBackup;
}
```

Una vez seleccionada la versión, se deberán comparar todos los modelos del estado actual del sistema con los existentes en la versión seleccionada.

Para ello se recorrerán todos los modelos existentes tanto en la carpeta de la versión seleccionada como los existentes en el estado actual del sistema y se compararán. El código para ello es relativamente trivial, así que no expondrá.

Sin embargo, el código realmente importante es el de la clase "ModelComparer" que será la encargada de realizar la comparación entre dos modelos. Para ello se servirá de la EMFCompare, tecnología expuesta en 3.3.2 EMF Compare.

Esta tarea de comparación tendrá dos pasos:

- **Obtener diferencias entre modelos**: Obtener los elementos distintos entre los dos modelos, haciendo uso del "MatchService" y del "DiffService" que proporciona EMFCompare para obtener diferencias entre modelos.

- **Generar el plan de evolución**: consiste en formatear las diferencias entre los modelos de forma que sean comprensibles por el ingeniero. Una vez formateadas, almacenar el plan de evolución en la carpeta de la copia del conocimiento del sistema seleccionada por el ingeniero.

5.3.3.1 OBTENER DIFERENCIAS ENTRE MODELOS

Para obtener las diferencias entre dos modelos, la clase "ModelComparer" hace uso del siguiente código:

```
public ModelComparer(EObject before, EObject after) {
…

}

public ModelChanges diff() {

      MatchModel matchModel = null;

      try {

            matchModel = MatchService.doMatch(after, before, null);
```

```
		} catch (InterruptedException e) {

			e.printStackTrace();

		}

		if (matchModel != null) {

			DiffModel diff = DiffService.doDiff(matchModel);

			if (diff != null && diff.getOwnedElements().size() > 0) {

				ModelChanges changes = new ModelChanges();

				for (DiffElement diffElement : diff.getOwnedElements()) {

					if (diffElement instanceof DiffGroup) {
						getDiffElements((DiffGroup) diffElement,
changes);
					}
				}

				return changes;
			}
		}

		return null;
}

private ModelChanges getDiffElements(DiffGroup diffGroup,
			ModelChanges changes) {

		for (DiffElement diffElement : diffGroup.getSubDiffElements()) {

			if (diffElement instanceof DiffGroup) {
				changes = getDiffElements((DiffGroup) diffElement, changes);
			}

			else {

				if (diffElement instanceof AttributeChange) {

					AttributeChange change = (AttributeChange)
diffElement;

					if (change.getKind() == DifferenceKind.ADDITION) {

						changes.addAddition(change.getRightElement());
					}

					else if (change.getKind() == DifferenceKind.DELETION)
{
						changes.addRemoval(change.getRightElement());
					}
```

```
                    }

                    else if (diffElement instanceof
ModelElementChangeLeftTarget) {
                           ModelElementChangeLeftTarget change =
(ModelElementChangeLeftTarget) diffElement;

                           if (change.getKind() == DifferenceKind.ADDITION) {
                                  changes.addAddition(change.getLeftElement());
                           }

                           else if (change.getKind() == DifferenceKind.DELETION)
{
                                  changes.addRemoval(change.getLeftElement());
                           }

                    }

                    else if (diffElement instanceof
ModelElementChangeRightTarget) {
                           ModelElementChangeRightTarget change =
(ModelElementChangeRightTarget) diffElement;

                           if (change.getKind() == DifferenceKind.ADDITION) {
                                  changes.addAddition(change.getRightElement());
                           }

                           else if (change.getKind() == DifferenceKind.DELETION)
{
                                  changes.addRemoval(change.getRightElement());
                           }

                    }
             }
      }

      return changes;

}
```

Como se puede ver el constructor del ModelComparer requiere de dos modelos a comparar (versión previa y posterior). Una vez tengamos estos dos modelos a comparar, podremos utilizar el método “diff”.

Este método hará uso del “MatchService” y del “DiffService” para obtener los elementos añadidos y los elementos eliminados. Como se puede observar este método es bastante extenso debido a dos motivos:

- La comparación ha de ser recursiva, ya que una característica del sistema además de tener atributos puede contener también otras características. Es por ello que habrá que comparar todos los elementos hijos de cada uno de los elementos de los modelos.

- Además la implementación del "DiffService" es un tanto compleja y es bastante complicado generalizar un método para gestionar los resultados que devuelve en las comparaciones entre elementos.

En cualquier caso, se devolverá un objeto de tipo "ModelChanges" que contendrá todos los elementos añadidos y eliminados entre cada una de las versiones de los modelos. Este es el código de la clase:

```
public class ModelChanges {

	private ArrayList<EObject> removals;
	private ArrayList<EObject> additions;

	 public ModelChanges() {
		removals=new ArrayList<EObject>();
		additions=new ArrayList<EObject>();
	}

	 public void addRemoval(EObject featureToRemove){
		if(!removals.contains(featureToRemove))
		removals.add(featureToRemove);
	 }

	 public void addAddition(EObject featureToAdd){
		if(!additions.contains(featureToAdd))
		additions.add(featureToAdd);
	 }

	 public ArrayList<EObject>getRemovals(){
		return removals;
	 }

	 public ArrayList<EObject> getAdditions(){
		return additions;
	 }

}
```

Una vez realizada esta comparación, quedará la segunda parte del proceso. Formatear los resultados para que sean comprensibles por el ingeniero.

5.3.3.2 GENERAR EL PLAN DE EVOLUCIÓN

Tras obtener las diferencias entre los modelos ya podremos generar el plan de evolución. En primer lugar, habremos de formatear las diferencias de forma que sean comprensibles para el ingeniero.

Para ello se recorrerán todos los cambios detectados entre una versión y otra del modelo y se agruparán entre adiciones y eliminaciones de elementos. Para ello la clase "ModelComparer" utiliza el método "formatChangesForEvolutionPlan":

```
private String formatChangesForEvolutionPlan(ModelChanges changes,
            String evolutionPlan) {

      if (changes.getAdditions().size() > 0
                  || changes.getRemovals().size() > 0) {

            evolutionPlan += "-" + modelName + "- \n Increments: ";

            if (changes.getAdditions().size() > 0) {

                  evolutionPlan = appendElements(changes.getAdditions(),
                              evolutionPlan);
            }

            evolutionPlan += "\n Decrements: ";

            if (changes.getRemovals().size() > 0) {

                  evolutionPlan = appendElements(changes.getRemovals(),
                              evolutionPlan);
            }

            evolutionPlan += "\n\n";
      }

      return evolutionPlan;
}

protected    abstract    String    appendElements(List<EObject>    obj,    String
evolutionPlan);
```

Como se puede ver, se recorren las adiciones y las eliminaciones y se va construyendo la variable “evolutionPlan”. A su vez, se llama al método “appendElements”.

Como se puede ver, este método está sin definir, ya que se han implementado comparadores concretos a cada uno de los tipos de modelos que existen en nuestro enfoque: modelos de características, resoluciones, arquitectura…

De esta forma, cada comparador identificará qué tipo de elemento es el que se ha modificado en función del modelo al que está asociado el comparador.

Como ejemplo de ello, a continuación se expone el código del comparador de características:

```
public class FeatureModelComparer extends ModelComparer {

      public FeatureModelComparer(EObject before, EObject after) {
            super(before, after);

            modelName = "Feature Model " + ((FeatureModel)before).getName();
      }
```

```
	@Override
	protected String appendElements(List<EObject> obj, String evolutionPlan)
{
		for (EObject addition : obj) {

			if (addition instanceof BinaryRelationship) {

				String literal = "";

				BinaryRelationship binaryRel = (BinaryRelationship)
addition;

				if (addition instanceof Optional) {
					literal = "Optional";
				}

				else if (addition instanceof Mandatory) {
					literal = "Mandatory";
				}

				else if (addition instanceof Regular) {
					literal = "Regular";
				}

				evolutionPlan += literal + "Relationship_"
						+ binaryRel.getName() + "-From_"
						+ ((Feature)
binaryRel.getFrom()).getName() + "-To_"
						+ binaryRel.getTo().getName() + ",";
			}

			else if (addition instanceof Attribute) {
				Attribute att = (Attribute) addition;

				evolutionPlan += "Attribute_" + att.getName()+ ",";
			}

			else if (addition instanceof Alternative) {
				Alternative alt = (Alternative) addition;

				evolutionPlan += "AlternativeRelationship_" +
alt.getName()+ ",";
			}

			else if (addition instanceof OR) {
				OR or = (OR) addition;

				evolutionPlan += "ORRelationship_" + or.getName()+
",";
			}

			else if (addition instanceof Constraint) {
				Constraint constraint = (Constraint) addition;
```

```
                        String literal = "";

                        if (constraint instanceof Requires) {
                              literal = "Requires";
                        } else if (constraint instanceof Excludes) {
                              literal = "Excludes";
                        }

                        evolutionPlan += literal + "Constraint_" +
constraint.getName()+ ",";
                  }

                  else if (addition instanceof Feature) {
                        evolutionPlan += "Feature_" + ((Feature)
addition).getName()
                                    + ",";
                  }

            }

            evolutionPlan = evolutionPlan.substring(0,
                        evolutionPlan.lastIndexOf(','));

            return evolutionPlan;
      }
}
```

Como se puede ver y como se comentaba, se recorren las diferencias comprobando de qué tipo es cada una de ellas en función de los posibles tipos de elementos del modelo. En este caso concreto, los tipos de elementos son relaciones entre características, atributos, restricciones… ya que estos son los elementos que puede contener un Modelo de Características.

Otro ejemplo es el comparador de arquitecturas, mucho más sencillo que el de características:

```
public class ArchitectureModelComparer extends ModelComparer {

      public ArchitectureModelComparer(EObject before, EObject after) {
            super(before, after);

            modelName = "Architecture Model";
      }


      @Override
      protected String appendElements(List<EObject> obj, String evolutionPlan){
            for (EObject addition : obj) {

                  if (addition instanceof Service) {
                        evolutionPlan += "Service_"
                                    + ((Service) addition).getID() + ",";
                  }

                  else if (addition instanceof Channel) {
```

```
                        evolutionPlan += "Channel_"
                                    + ((Channel) addition).getID() + ",";
                  }

                  else if (addition instanceof Component) {
                        evolutionPlan += "Component_"
                                    + ((Component) addition).getID() + ",";
                  }
            }

            evolutionPlan = evolutionPlan.substring(0,
evolutionPlan.lastIndexOf(','));

            return evolutionPlan;
      }

}
```

Una vez aplicados cada uno de los comparadores en función del tipo de modelo que estemos comparando, se obtendrá como resultado un archivo "txt" que contendrá la evolución del sistema entre las dos versiones seleccionadas. Un ejemplo podría ser el siguiente:

```
*** EVOLUTION PLAN ***
-Feature Model FM-
 Increments:
 Decrements: Feature_D,Feature_1106,Feature_1230,RegularRelationship_B-From_-
To_A

-Resolution Model FM_Resolutions-
 Increments:
 Decrements: Resolution_null,Resolution_R1,Resolution_1236

-Architecture Model-
 Increments:
 Decrements: Component_InputInstructions,Component_Prueba,Service_1107_S_alt
```

Una vez obtenido el plan de evolución, el ingeniero tendrá un documento en el que se expongan los cambios realizados al evolucionar el sistema.

5.3.4 IMPLEMENTACIÓN DEL PROCESO DE VALIDACIÓN DEL SISTEMA

Para validar el sistema tras todo el proceso de evolución, se ha implementado una acción contextual mediante la cual el ingeniero podrá lanzar un proceso de validación sobre todo el sistema.

Esta acción se podrá aplicar sobre los modelos de características y a continuación se expone el código xml para añadir su funcionalidad a las acciones de M4SPL:

```
        <action

class="es.upv.pros.m4spl_reasoning.fama_reasoner.actions.ValidateSystemAction"
              icon="icons/validateModelIcon.gif"
              id="es.upv.pros.M4SPL_Reasoning.FAMA.Reasoner.action1"
              label="Validate System"

menubarPath="es.upv.pros.M4SPL_Reasoning.FAMA_Reasoner.operations">
        </action>
```

Para ello, en primer lugar se generará el Espacio de Adaptación a partir del Modelo de Características, el Modelo de Resoluciones y el Modelo de Configuración.

Ya que la acción se aplica sobre los modelos de características, al iniciar el proceso de validación ya conoceremos qué Modelo de Características desea el ingeniero que se utilice para validar el sistema.

Sin embargo, deberemos solicitarle que indique qué Modelo de Configuración y de resoluciones habrá que utilizar. Aunque en caso de sólo existir uno de cada tipo, automáticamente se seleccionará el único existente.

Esta acción contextual lanzará un asistente que nos guiará en los pasos necesarios para lanzar el proceso de validación del sistema. Estos pasos consistirán en:

1. **Seleccionar el Modelo de Configuración** a utilizar para generar el espacio de adaptación.

2. **Seleccionar el Modelo de Resoluciones.**

3. **Generar el espacio de adaptación** a partir del Modelo de Características, configuración y resoluciones.

4. **Aplicar refinamientos al espacio de adaptación** para garantizar propiedades en el sistema.

Será mediante estos diálogos como el ingeniero escogerá el Modelo de Configuración:

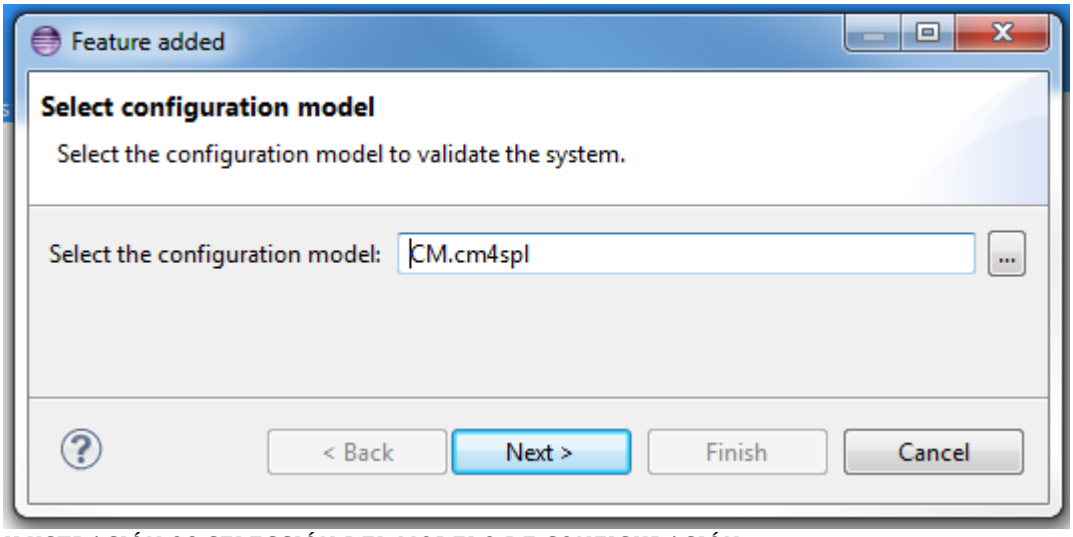

ILUSTRACIÓN 38 SELECCIÓN DEL MODELO DE CONFIGURACIÓN

Diálogo para seleccionar el Modelo de Configuración.

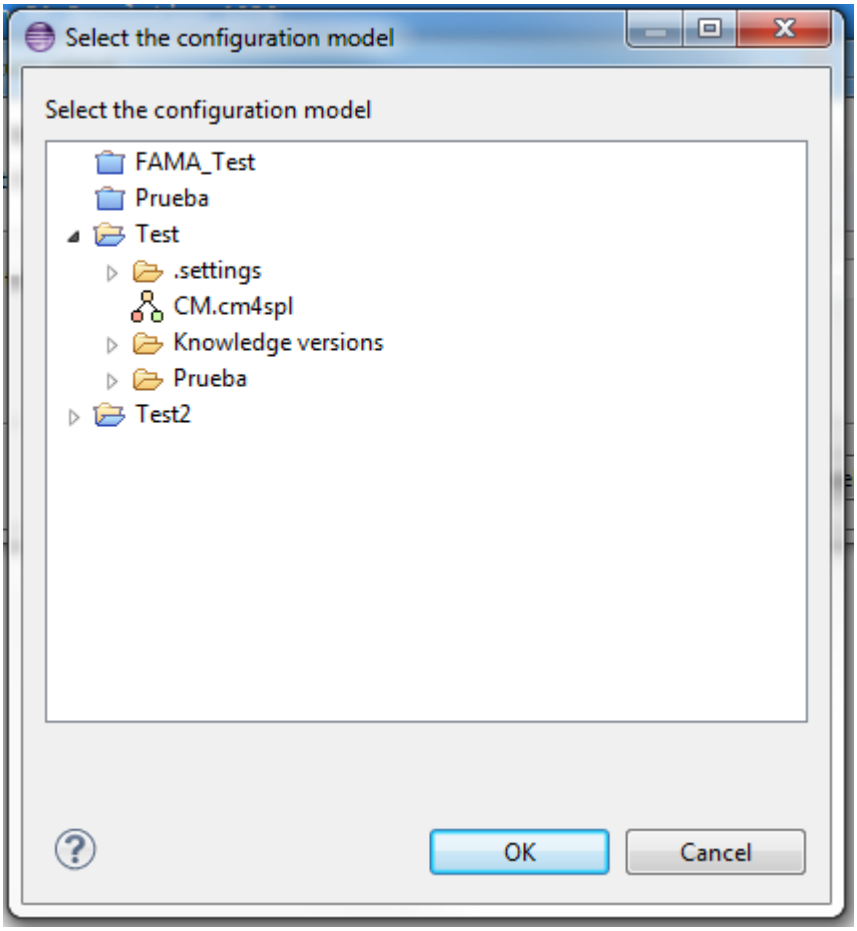

ILUSTRACIÓN 39 SELECCIÓN FILTRADA DEL MODELO DE CONFIGURACIÓN

Diálogo de selección del Modelo de Configuración del espacio de trabajo filtrado por su extensión (cm4spl).

Una vez seleccionado el Modelo de Configuración, se nos permitirá avanzar al siguiente paso para seleccionar el Modelo de Resoluciones:

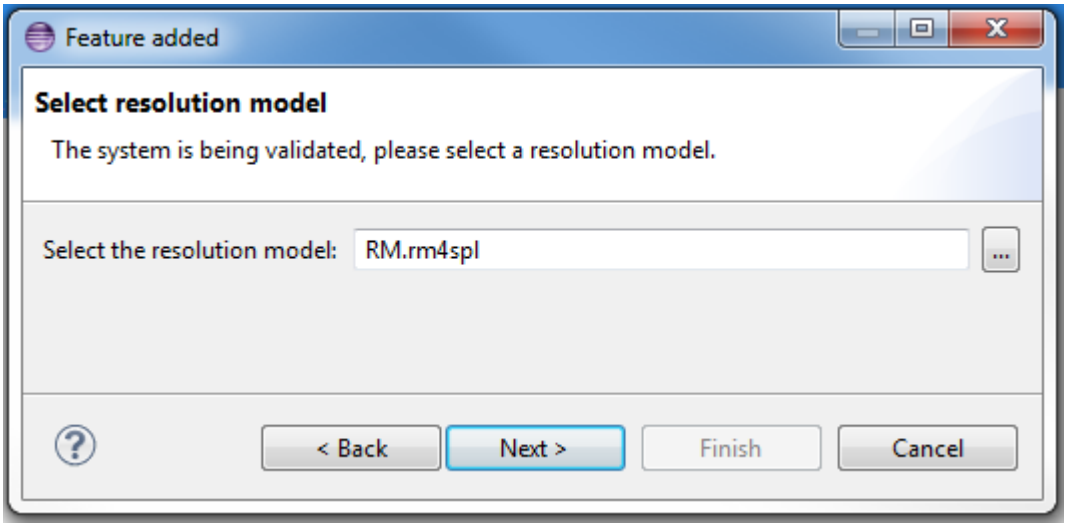

ILUSTRACIÓN 40 SELECCIÓN DEL MODELO DE RESOLUCIONES
Diálogo de selección del Modelo de Resoluciones.

Una vez el ingeniero haya seleccionado el Modelo de Configuraciones y el Modelo de Resoluciones, ya tendremos los 3 modelos necesarios para generar el espacio de adaptación: el de características, el de configuración y el de resoluciones.

Dado que M4SPL ya incorpora un plugin para generar el espacio de adaptación automáticamente, una vez seleccionados todos los modelos lo que se hace simplemente es llamar a este plugin para que lo genere. Este es el código para ello:

```
public StateMachineModel fromCMRMToSM(EObject featureModel,
            EObject configurationModel, EObject resolutionModel) {

      if (featureModel != null && configurationModel != null
                  && resolutionModel != null) {

            CalculateSM c;
            try {

                  c = new CalculateSM();

                  c.loadModels(featureModel.eResource().getURI().toString(),

      configurationModel.eResource().getURI().toString(),

      resolutionModel.eResource().getURI().toString());

                  c.setCMPath(configurationModel.eResource().getURI()
                              .toFileString());

                  c.run(null);
```

```
                    return c.getStateMachineModel();
                } catch (IOException e) {
                    e.printStackTrace();
                } catch (ATLCoreException e) {
                    e.printStackTrace();
                }
        }

        return null;
}
```

Evidentemente, se han tenido que hacer pequeñas modificaciones a la clase "CalculateSM" ya que estaba pensada para ejecutarse a partir de una acción contextual que seleccionaba el Modelo de Configuraciones. Sin embargo y a pesar de estas modificaciones el código para calcular el espacio de adaptaciones es básicamente el mismo.

Para generar el espacio de adaptación se nos solicitará un nombre para el nuevo modelo que se va a generar. Para ello se mostrará el siguiente diálogo:

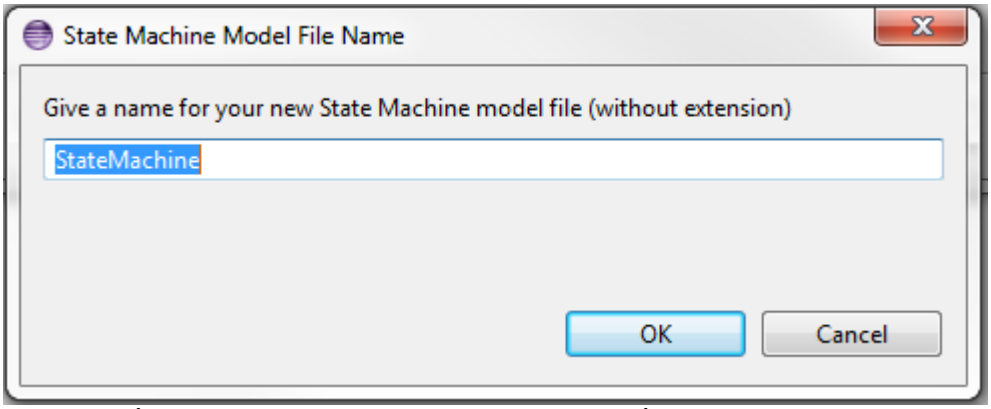

ILUSTRACIÓN 41 NOMBRE DEL ESPACIO DE ADAPTACIÓN

Y una vez se introduzca el nombre, se muestra información acerca de la generación del espacio.

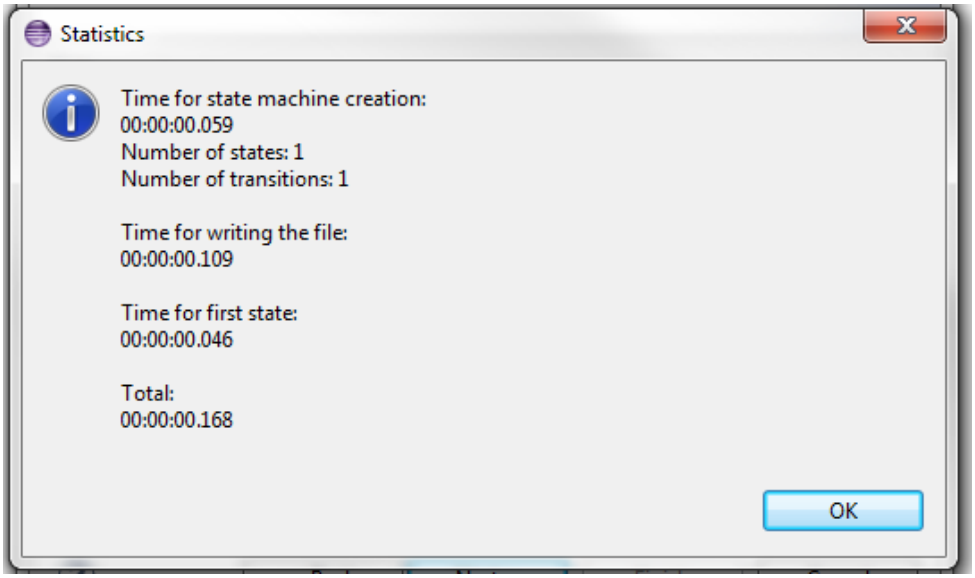

ILUSTRACIÓN 42INFORMACIÓN DE LA GENERACIÓN DEL ESPACIO DE ADAPTACIÓN

En esta ilustración se muestra la información que se proporciona al ingeniero sobre el espacio de adaptación. En ella se indica el tiempo de creación de la máquina de estados, el número de estados, transiciones…

Una vez hecho esto, ya se ha generado el espacio de adaptación del sistema. A partir de este espacio de adaptación se podrán aplicar los refinamientos explicados en el capítulo "4.5 Validar el sistema evolucionado".

Para ello se muestra un diálogo como el siguiente en que el ingeniero podrá aplicar los refinamientos que seleccione sobre el espacio de adaptación:

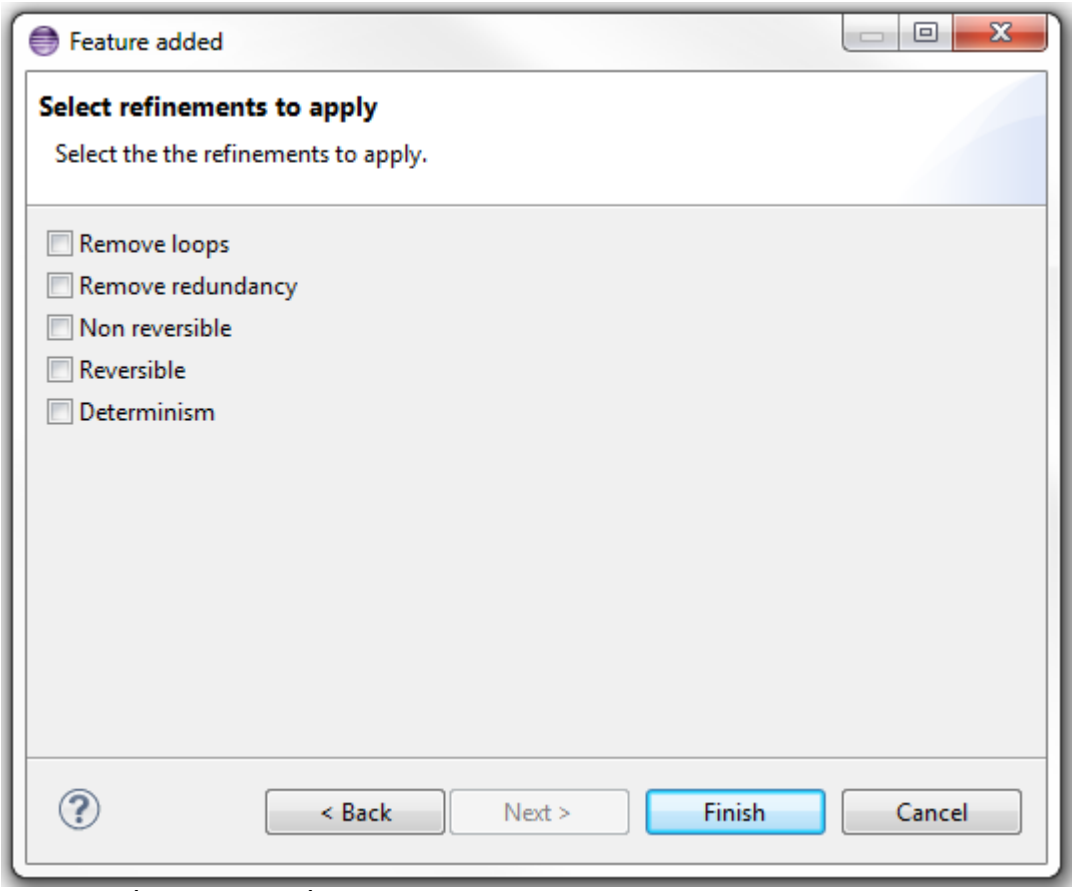

ILUSTRACIÓN 43 SELECCIÓN DE REFINAMIENTOS

En este diálogo el ingeniero selecciona los refinamientos que se aplicarán sobre el espacio de adaptación para garantizar que el sistema cumpla unas determinadas propiedades.

M4SPL también soporta la aplicación de refinamientos mediante varios plugins que aplican cada uno de ellos. Sin embargo, hasta ahora sólo era posible aplicar un refinamiento en solitario, generando un Modelo de Resoluciones para cada uno de los refinamientos.

Sin embargo, en el diálogo es posible seleccionar varios refinamientos por lo que se han modificado los plugins que aplican los refinamientos de forma que sea posible aplicar varios de ellos y generar un solo modelo de resolución con varios refinamientos.

El código de cada refinamiento sin embargo, es bastante extenso como para exponerlo. Básicamente lo que se ha hecho es modificarlos de forma que se apliquen los refinamientos de forma secuencial y cuando se ejecute uno, le indique al siguiente refinamiento a ejecutar sobre qué Modelo de Resoluciones ha de aplicarlo.

Por ejemplo, en caso de querer aplicar el refinamiento “Eliminar bucles”, “Eliminar redundancia” y “Determinismo” marcarían los 3 checks como se ve a continuación:

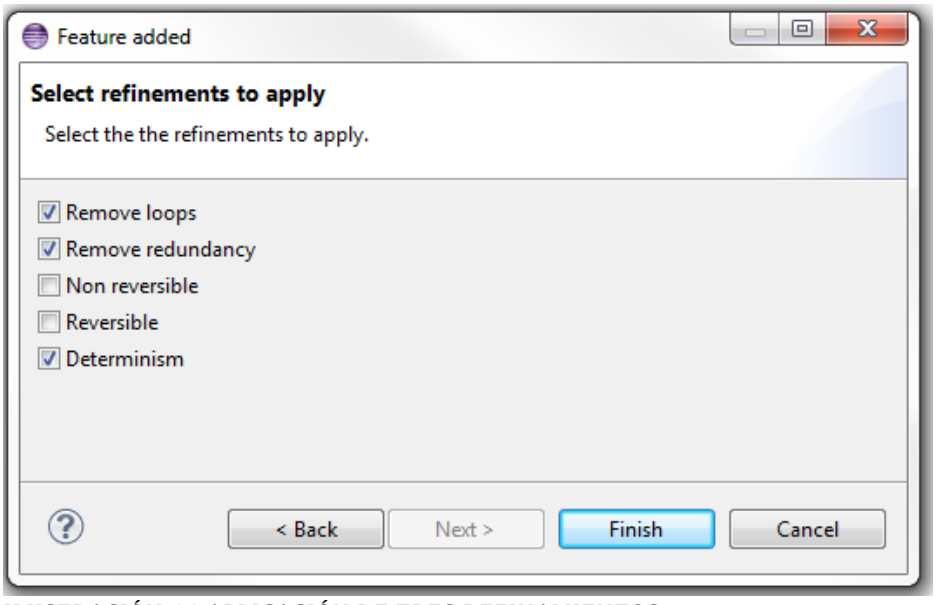

ILUSTRACIÓN 44 APLICACIÓN DE TRES REFINAMIENTOS

En esta ilustración se muestra cómo se pueden seleccionar tres refinamientos a aplicar.

Una vez seleccionados pulsaríamos en "finalizar" y se aplicarían los 3 refinamientos como se muestra en la Ilustración 45:

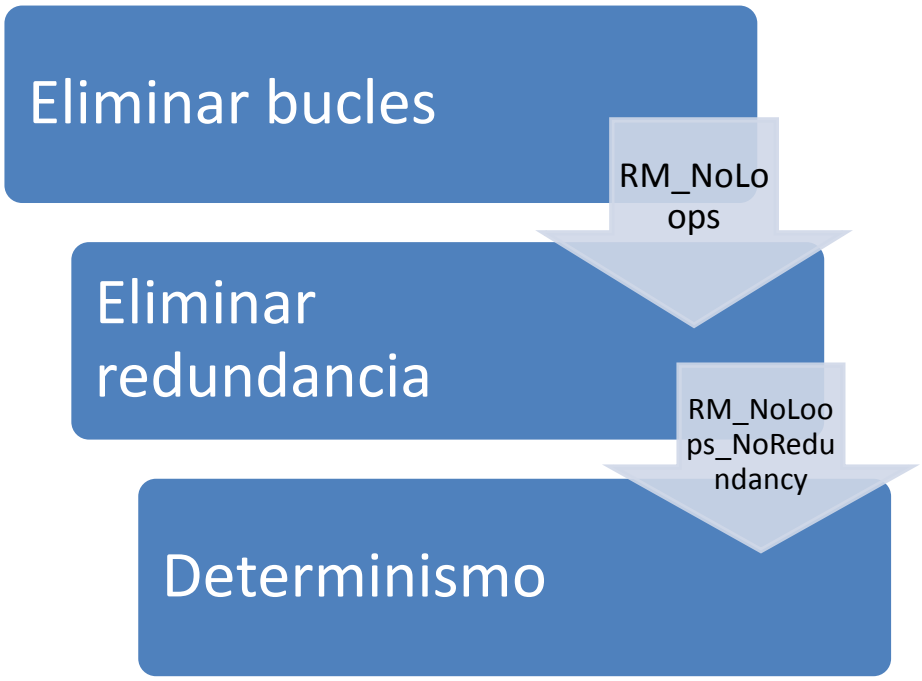

ILUSTRACIÓN 45 APLICACIÓN DE TRES REFINAMIENTOS

En esta ilustración se muestra como se aplican tres refinamientos secuencialmente y como indican al siguiente refinamiento a aplicar sobre qué Modelo de Resoluciones ha de aplicarse.

A continuación se expone un extracto del código que realiza esta aplicación de refinamientos:

```
if (removeLoopsCheck.getSelection()) {
```

```
        removeLoops = new RemoveLoops();

        removeLoops.setInputModelSMPath(validateHelper
                    .getStateMachineModel().eResource().getURI()
                    .toFileString());

        removeLoops.setSave(checkShouldSave(selecteds,
                    removeLoopsCheck));

        removeLoops.run(null);

        resolutionModel = removeLoops.getOutputResolutionModel();
}

if (removeRedundancyCheck.getSelection()) {
        removeRedundancy = new RemoveRedundancy();

        removeRedundancy.setInputModelSMPath(validateHelper
                    .getStateMachineModel().eResource().getURI()
                    .toFileString());

        removeRedundancy.setSave(checkShouldSave(selecteds,
                    removeRedundancyCheck));

        removeRedundancy.setOutputResolutionModel(resolutionModel);

        removeRedundancy.run(null);

        resolutionModel = removeRedundancy
                    .getOutputResolutionModel();

        removeRedundancyCheck.setSelection(false);
}

if (reversibleCheck.getSelection()) {
        reversible = new Reversible();

        reversible.setInputModelSMPath(validateHelper
                    .getStateMachineModel().eResource().getURI()
                    .toFileString());

        reversible.setSave(checkShouldSave(selecteds,
                    reversibleCheck));

        reversible.setOutputResolutionModel(resolutionModel);

        reversible.run(null);

        resolutionModel = reversible.getOutputResolutionModel();

        reversibleCheck.setSelection(false);
}
```

Como se puede ver, sencillamente se construye un objeto que aplicará el refinamiento (RemoveLoops, RemoveRedundancy, Reversible) y al que se le indica el espacio de adaptación sobre el que aplicarlos y el modelo de resolución de salida.

Internamente lo que harán es que si el modelo de resolución de salida no es nulo, aplicarán los refinamientos sobre el mismo. En caso de que sea nulo, se creará un nuevo modelo de resolución con el refinamiento aplicado sobre el mismo.

Una vez aplicados los refinamientos seleccionados, generan un Modelo de Resoluciones cuyo nombre indicará qué refinamientos se han aplicado para generarlo (en este caso, RM_noLoops_noRedundancy_noDeterminism.rm4spl). Además este Modelo de Resoluciones garantizará las propiedades que se seleccionen.

Una vez generado el espacio de adaptación y aplicados los refinamientos, habremos validado sobre el sistema las propiedades deseadas y por tanto habremos realizado una evolución segura del sistema.

6. CASO DE ESTUDIO

En este capítulo se presenta un caso de estudio real para aplicar sobre él todos los conceptos, procedimientos y la herramienta desarrollada que se han expuesto y detallado a lo largo de los capítulos anteriores.

El principal objetivo del capítulo es el de comprobar el uso de la herramienta ante un problema real y de esta forma exponer su aplicación sobre casos de estudio reales. Para ello se va a utilizar el caso de estudio de un Smart Car.

6.1 PRESENTACIÓN DEL CASO DE ESTUDIO

El caso de estudio escogido consiste en la definición del software para un Smart Car o coche inteligente. Se ha escogido un coche inteligente como caso de estudio por las siguientes razones que justifican su interés de cara al estudio de un sistema auto-adaptable:

- Un vehículo necesita tener una gran cantidad de datos sobre su estado actual para garantizar la seguridad tales como temperatura del motor, nivel de combustible en el depósito, inflado de las ruedas… Es decir, necesita **conocerse a sí mismo**.

- Necesita adaptarse y configurarse en función de **múltiples e imprevistas situaciones**. Debido a que un vehículo está siempre en movimiento, las situaciones a las que puede enfrentarse son realmente imprevisibles. Además, deberá reaccionar correctamente a un accidente, que es precisamente la definición de situación imprevista.

- **Ha de auto-optimizarse:** Un vehículo ha de ser capaz de gestionar sus recursos eficientemente. Un claro ejemplo es el gasto de combustible.

- **Debe auto sanarse:** dado que es un sistema crítico (dependen vidas humanas de su funcionamiento) un vehículo ha de ser capaz de recuperarse del fallo de uno de sus componentes sin que ello afecte al sistema al completo.

- **Ha de protegerse:** dado que un vehículo circula por el mundo real hay una miríada de elementos que pueden causarle daño voluntaria o involuntariamente: animales que se crucen en su camino, otros vehículos que provoquen un accidente, fenómenos meteorológicos… Es por ello que ha de ser capaz de protegerse en la medida de lo posible frente a estos elementos agresivos.

- **Ha de conocer su entorno:** un vehículo ha de tener toda la información posible de su entorno para garantizar su seguridad y su funcionamiento eficiente. Desde encender las luces cuando la visibilidad sea baja, detectar la proximidad de los otros vehículos, etc.

Como se puede ver, un vehículo inteligente prácticamente cumple punto por punto las características que un sistema auto-adaptable que plantea Horn en su manifiesto de la computación autónoma[9]. Esto hace de este caso de estudio del coche inteligente un dominio ideal para hablar de sistemas autoadaptables.

Un vehículo es también un sistema crítico, como se ha comentado, ya que de él dependen vidas humanas. Es por ello que es imposible plantearse una parada del sistema para corregir algún error o actualizarlo. Por tanto, será necesaria la modificación del sistema en **tiempo de ejecución**.

Además, un ejemplo muy claro de línea de producto es el de la cadena de producción de vehículos. Si a ello le añadimos que cada día más se está incluyendo software en los mismos para hacerlos más inteligentes como plantea el caso de estudio y que para modificarlo será necesario hacerlo en tiempo de ejecución, no habrá otro contexto mejor para aplicar la **línea de producto software dinámica.**

6.2 FUNCIONALIDAD DEL SMART CAR

En la Ilustración 46 se expone el diagrama de las características especificadas para el caso de estudio del Smart Car, representadas mediante un Modelo de Características. Evidentemente, es un conjunto reducido ya que definir todas las características necesarias para un Smart Car sería excesivamente complejo.

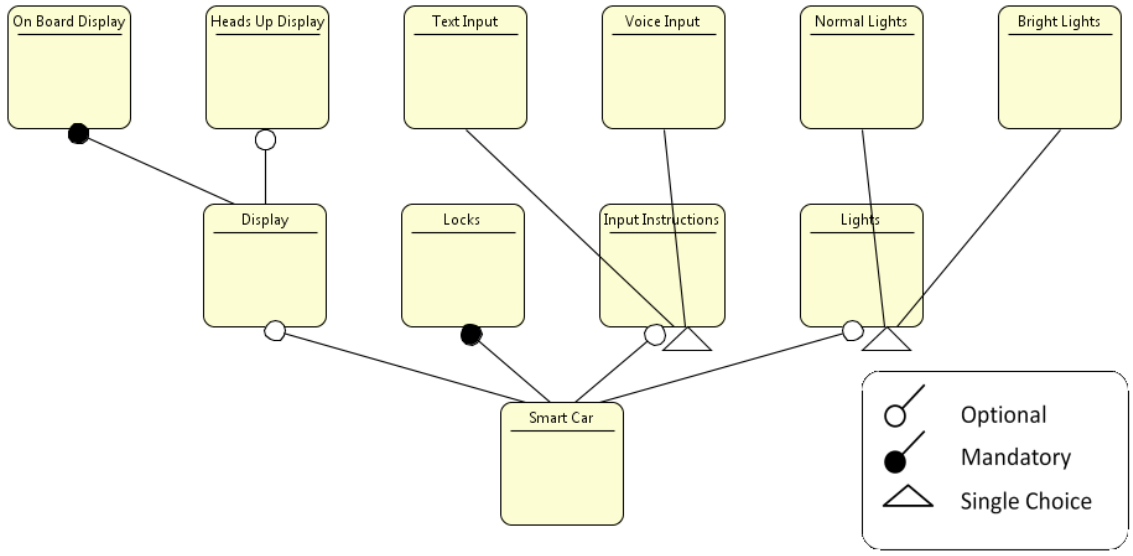

ILUSTRACIÓN 46 MODELO DE CARACTERÍSTICAS DEL SMART CAR
Esta ilustración muestra el diagrama de características de un Smart Car.

Las principales características serán las del primer nivel en el árbol:

- **Display**: el vehículo ha de mostrar información al usuario, ya sea acerca del propio vehículo (velocidad, combustible…) así como de la información contextual (proximidad de otros vehículos, avisos meteorológicos…)

- **Locks**: el vehículo ha de estar cerrado mientras esté en marcha o tras permanecer un cierto tiempo en reposo

- **Input instruccions:** ha de proprocionarse una entrada al usuario para interactuar con el vehículo

- **Lights:** el vehículo tiene que tener luces para iluminar la carretera en caso de poca visibilidad.

Las características de segundo nivel, como se verá a continuación, son refinamientos de las características principales:

- **Display**

    - **On Board Display:** el vehículo ha de disponer de un panel de información donde mostrar los datos al usuario.

    - **Heads up display:** en caso de ser necesario, el vehículo ha de disponer de un panel de avisos en el que mostrarle información importante al usuario como puede ser "combustible bajo" o "revisión de aceite necesaria". Estos avisos deberán tener un alto grado de alerta, ya que son avisos importantes.

- **Input instructions**

    - **Text input:** se puede proporcionar al usuario una entrada de texto para indicar datos al vehículo.

    - **Voice input:** se puede proporcionar una entrada de información del usuario mediante voz, de forma que sea menos incómoda y más segura durante la conducción.

- **Lights**

    - **Normal lights:** el vehículo dispondrá de luces normales o cortas.

    - **Bright lights:** también dispondrá de luces brillantes o largas.

Además las características tienen relaciones entre ellas:

- **Display:** mostrar información será una característica opcional del vehículo inteligente. Sin embargo, en caso de que se active será obligatorio que se active el panel de información (**on board dislay**) y opcional que se activen los avisos (**heads up display**).

- **Locks:** será obligatorio para el vehículo que se activen la característica de cierres.

- **Input instructions**: la característica de entrada de instrucciones será opcional. Sin embargo, en caso de que se active, sólo podremos activar la entrada de texto (**text input**) o la entrada de voz (**voice input**) ya que tienen una relación "alternativa". Esto es debido a que el usuario no puede introducir instrucciones a la vez por voz y escribiendo.

- **Lights:** las luces serán opcionales. Sin embargo e igual que en el caso anterior, debido a que hay una relación alternativa sólo podrá estar activa la característica de "luces cortas" (**normal lights**) o "luces largas" (**bright lights**).

6.2.1 MODELO DE CONFIGURACIÓN

En la Ilustración 47 se expone el Modelo de Configuración inicial del Smart Car. Las características verdes representan características activas mientras que las características rojas representan características inactivas.

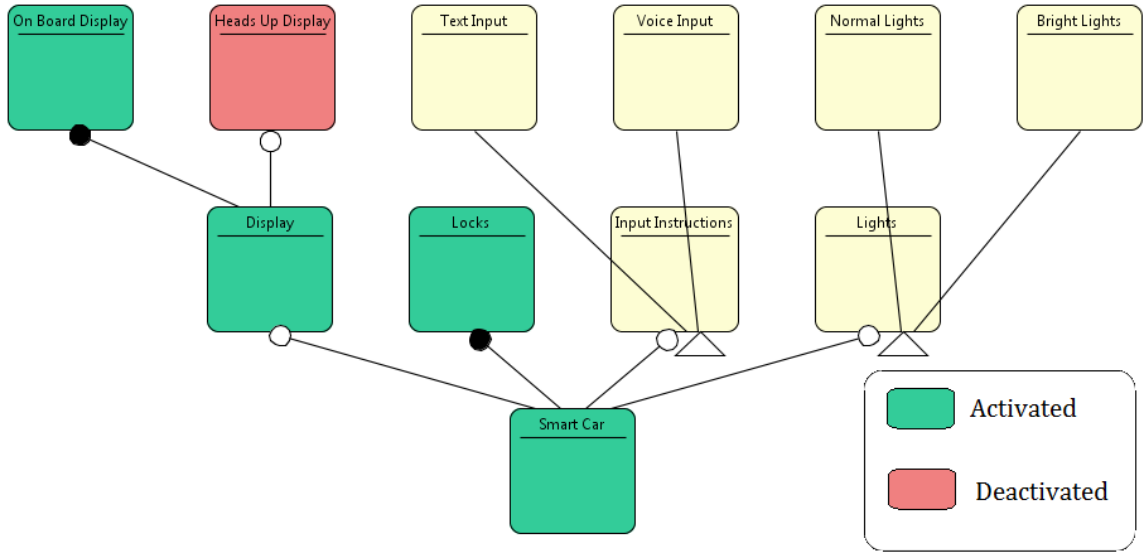

ILUSTRACIÓN 47 MODELO DE CONFIGURACIÓN DEL SMART CAR

Este Modelo de Configuración es bastante sencillo de entender. Para el producto final inicialmente se tendrán activadas las características de "Display", "On Board Display" y "Locks". "Heads Up Display" estará desactivada inicialmente.

6.2.2 MODELO DE RESOLUCIONES

A continuación se expone el modelo de resoluciones (ver Ilustración 48 Modelo de Resoluciones del Smart Car):

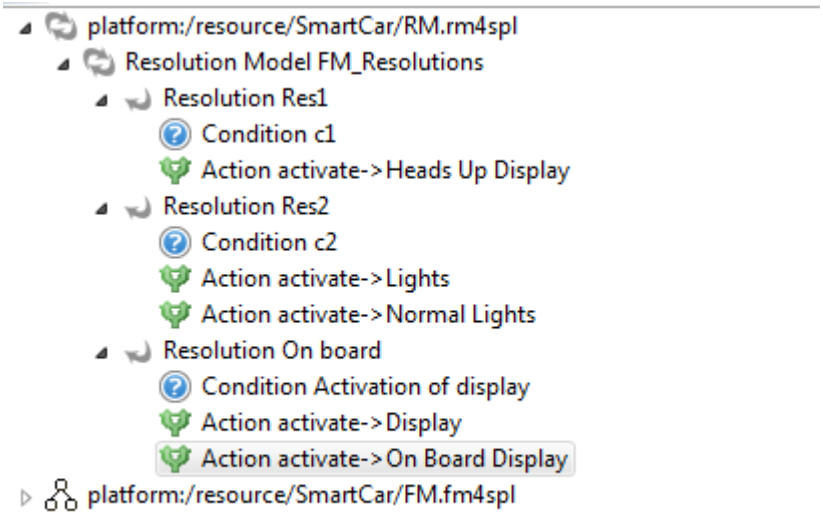

ILUSTRACIÓN 48 MODELO DE RESOLUCIONES DEL SMART CAR

En este modelo se especifican tres resoluciones:

- **Res1**: en caso de que se cumpla la condición c1, habrá que activar la característica "Heads Up Display" (panel de avisos).

- **Res2:** en caso de cumplirse la condición c2, se activará la característica "Lights" y "Normal Lights".

- **On board:** en caso de cumplirse la condición de "Activation of display", es decir, que se active el mostrar información al usuario, será necesario activar las características de "Display" y "On Board Display".

6.2.3 MODELO DE ARQUITECTURA

En el Modelo de Arquitectura se muestra qué elementos arquitectónicos serán los que realicen cada característica. En la Ilustración 49Ilustración 50 se muestra el Modelo de Arquitectura del Smart Car.

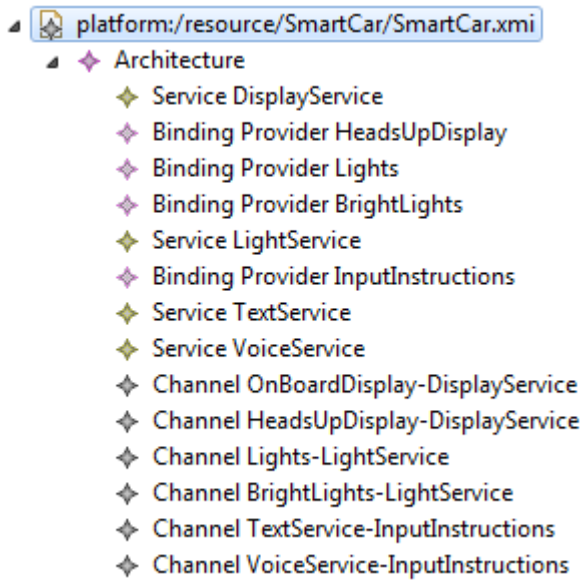

ILUSTRACIÓN 49 MODELO DE ARQUITECTURA SMART CAR

En el Modelo de Arquitectura se ven los servicios que realizan las características, como el servicio "DisplayService", "TextService", "LightService"... Es en el modelo de enlace donde se relacionan cada uno de los servicios con las características.

6.2.3 MODELO DE ENLACE

En el modelo de enlace se muestra qué elementos arquitectónicos serán los que realicen cada característica. El modelo de enlace relaciona características del Modelo de Características con elementos arquitectónicos del Modelo de Arquitectura. En la Ilustración 50 se muestra el modelo de enlace del Smart Car.

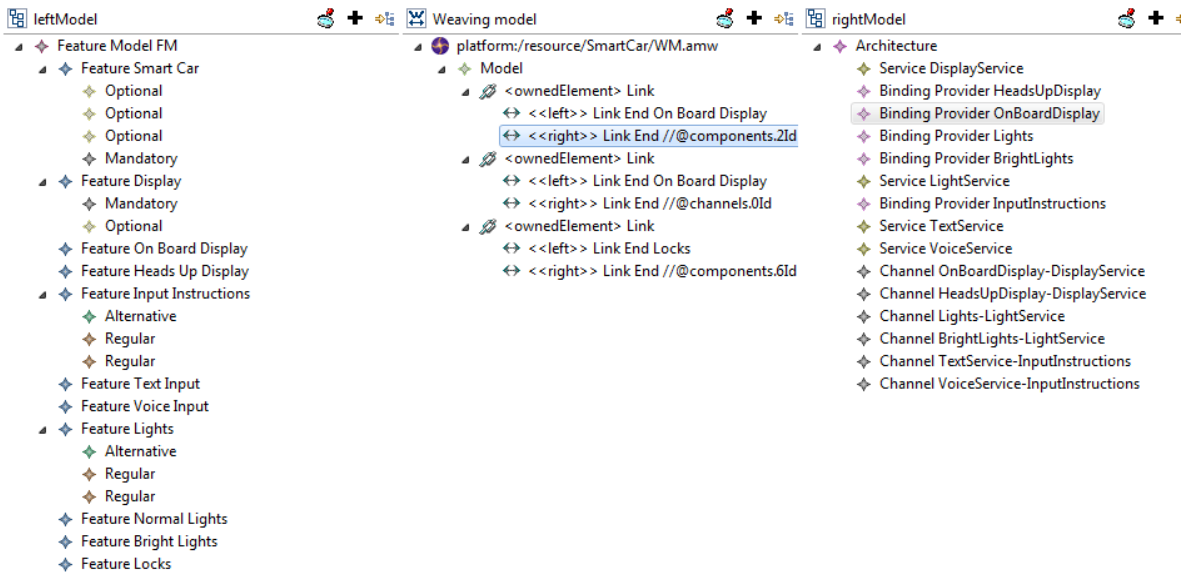

ILUSTRACIÓN 50 MODELO DE ENLACE DEL SMART CAR

En este modelo se muestran tres enlaces, dos referentes a la característica “On Board Display” y uno para la característica “Locks”.

En cada enlace se especifica además cual es el elemento asociado de la arquitectura. Por ejemplo, en el caso de la Ilustración 50 Modelo de enlace del Smart car el primer enlace tiene como referencias la característica “On Board Display” y como elemento arquitectónico que la realiza “Binding Provider OnBoardDisplay”.

Como se puede ver por la especificación de características, configuraciones, resoluciones, enlaces y Modelo de Arquitectura, el vehículo no es especialmente “inteligente”. Tiene características que prácticamente cualquier coche hoy en idea contiene.

Sin embargo el objetivo del caso de estudio no es el de modelar un vehículo inteligente en toda su extensión y el de generar un modelo extenso y detallado, sino es el de mostrar como la eliminación o adición de una característica puede afectar al resto del sistema y como la herramienta desarrollada asiste a este proceso.

Es por ello que los modelos utilizados son relativamente sencillos para poder seguir los procesos que se van a aplicar sobre él utilizando la herramienta.

6.3 APLICACIÓN DE LA PROPUESTA

En esta sección se van a aplicar sobre el caso de estudio dos modificaciones: la eliminación de una de sus características y la adición de una nueva. Así, se podrá observar perfectamente como la herramienta desarrollada da soporte a ambas modificaciones y como al finalizar el proceso de modificación, el sistema sigue siendo válido.

6.3.1 ELIMINACIÓN DE LA FUNCIONALIDAD "ON BOARD DISPLAY"

En este punto del capítulo se muestra la eliminación de la característica "On Board Display". Se ha optado por esta característica porque como se puede ver en los modelos que especifican el Smart Car previamente explicados, está presente en todos ellos.

De esta forma, se podrá exponer como se propaga la eliminación de la característica a todo el conocimiento del sistema y como finalmente, se obtiene un sistema válido.

El primer paso que se realizará será el de realizar una copia del conocimiento del sistema. Para ello tendremos que pulsar con el botón derecho sobre el modelo "FM.fm4spl" que es el Modelo de Características del Smart Car y seleccionar la acción "Evolve Model".

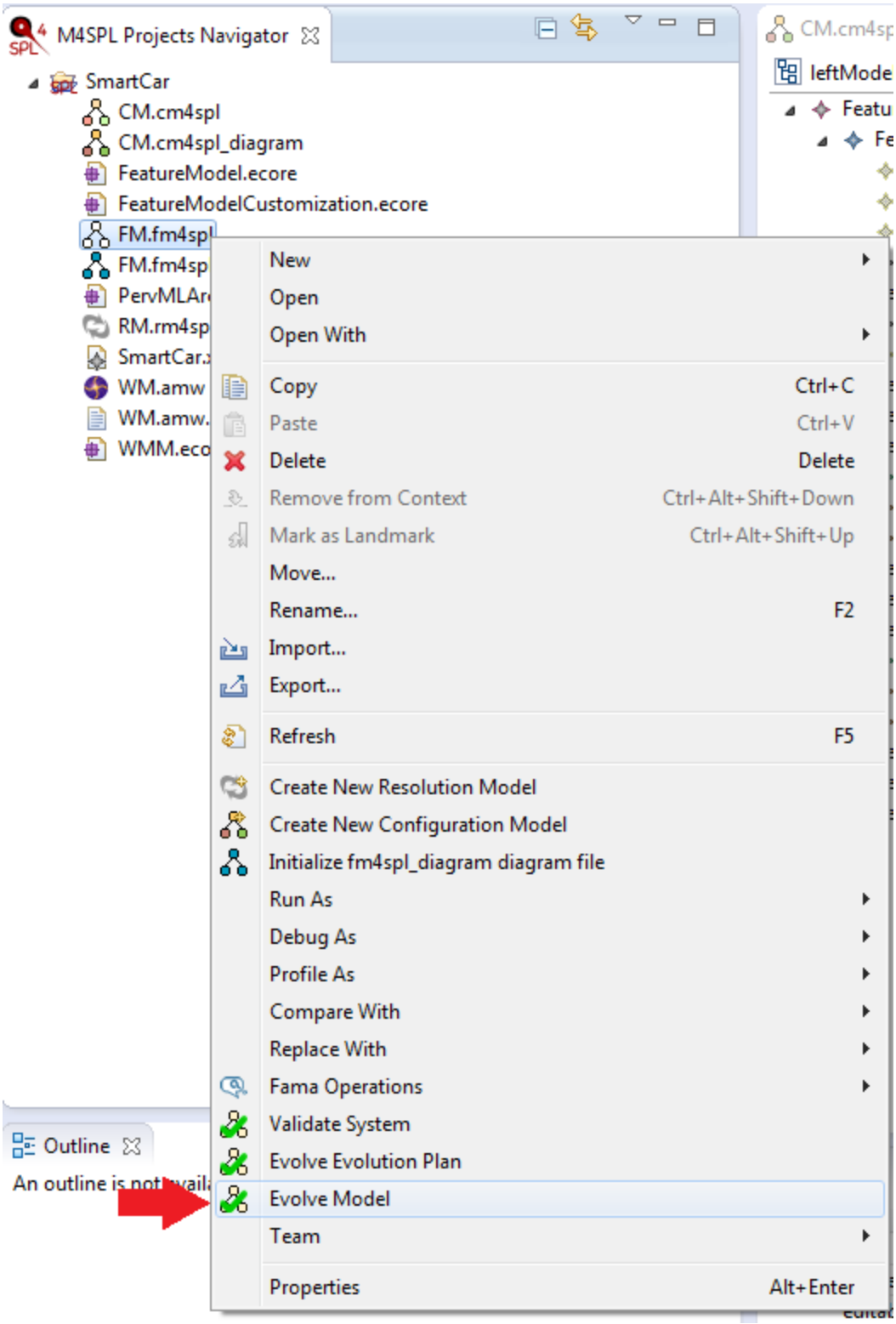

ILUSTRACIÓN 51 COPIA DEL CONOCIMIENTO DEL SMART CAR

En Ilustración 51 Copia del conocimiento del Smart Car se muestra como se realiza una copia del conocimiento del Smart Car.

Una vez pulsemos en esta acción se genera una carpeta con nombre “Knowledge Versions” en el espacio de trabajo. Dentro de ella se crea una carpeta con un id de evolución donde se almacena una copia de todos los modelos.

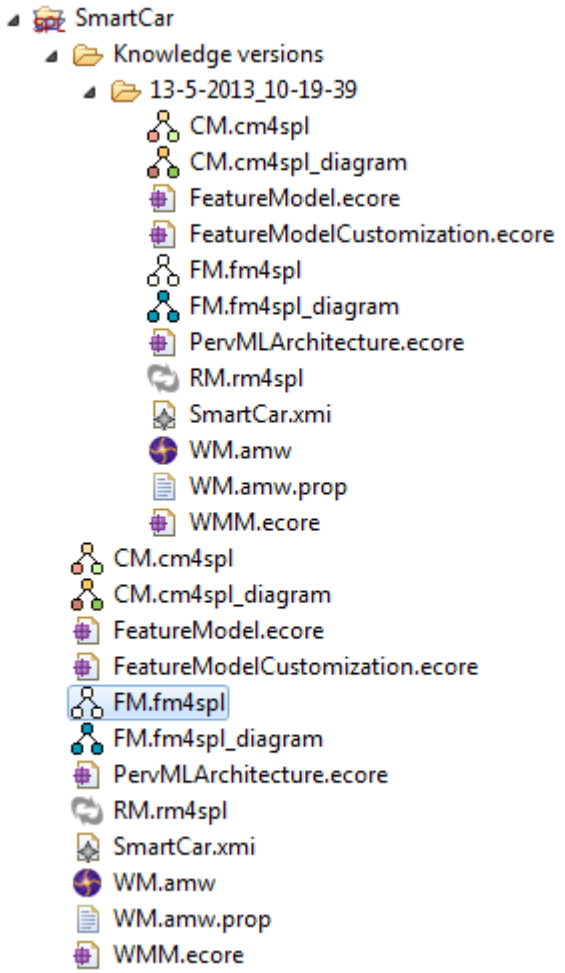

ILUSTRACIÓN 52 KNOWLEDGE VERSIONS DE SMART CAR

En la Ilustración 52 Knowledge versions de Smart Car se muestra la copia de todos los modelos del conocimiento del smart car en la carpeta /Knowledge versions/13-5-2013_10-19-39

Una vez almacenada la copia del conocimiento procederemos a eliminar la característica “On Board Display” del Modelo de Características. Al hacerlo, se lanzará el patrón de evolución “Eliminar funcionalidad” y nos irá mostrando las distintas ventanas del asistente.

Primero, nos solicita el Modelo de Resoluciones a utilizar:

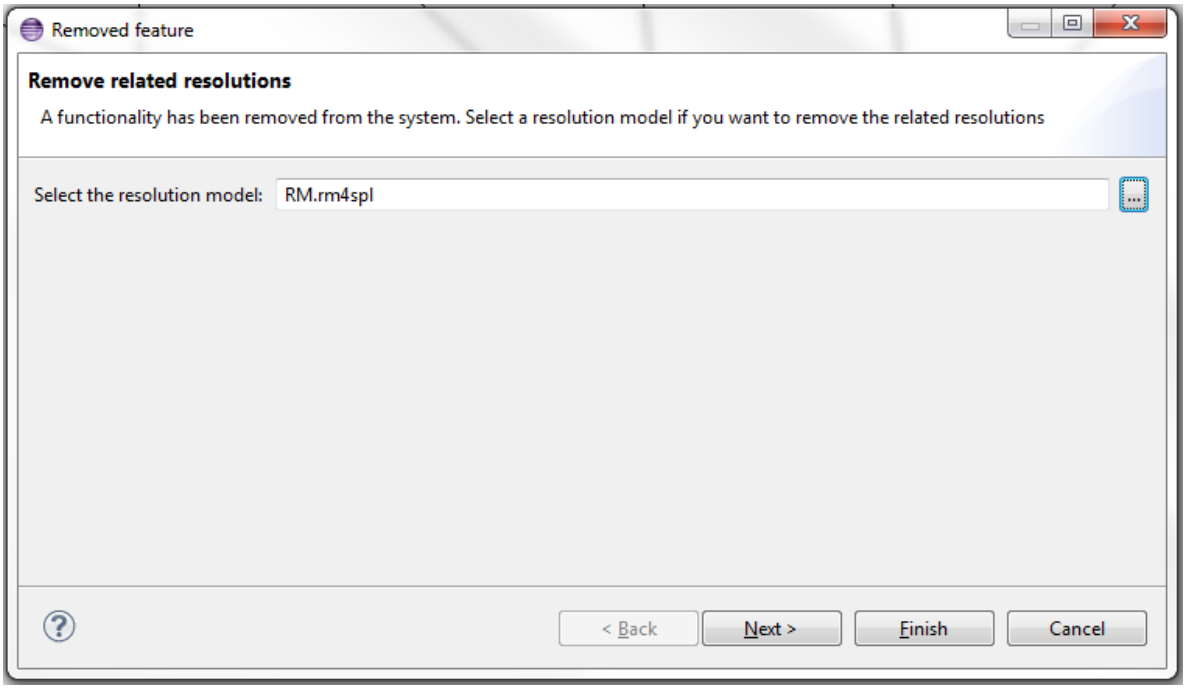

ILUSTRACIÓN 53 SELECCIÓN DE MODELO DE RESOLUCIÓN, ELIMINACIÓN DE LA CARACTERÍSTICA "ON BOARD"

En esta ilustración se muestra el primer paso del asistente de eliminación de funcionalidad, la selección del modelo de resolución.

Una vez seleccionado, nos mostrará las resoluciones que hacen uso de la característica y que van a ser eliminadas. En este caso, la resolución On Board:

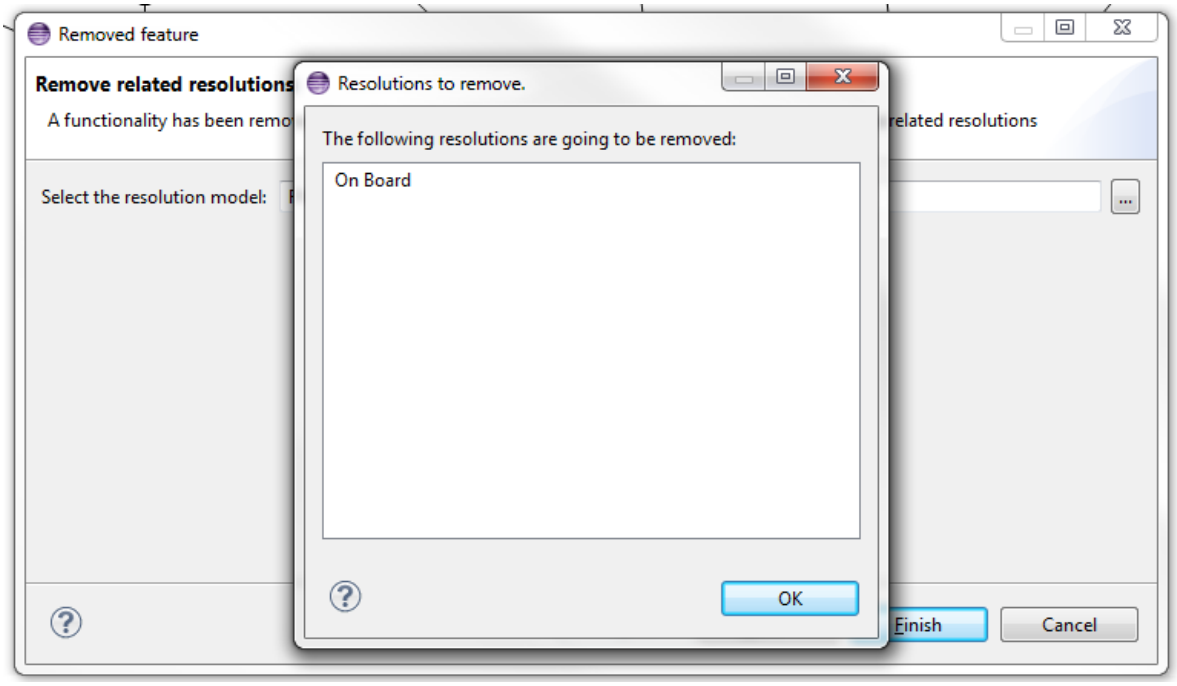

ILUSTRACIÓN 54 ELIMINACIÓN DE RESOLUCIÓN "ON BOARD"

Esta ilustración muestra el diálogo en el que se informa al ingeniero de las resoluciones asociadas a la característica eliminada. En este caso se eliminará la resolución On Board.

Tras eliminar la resolución asociada (On Board) a la característica eliminada (On Board Display) se muestra al ingeniero un diálogo con los elementos arquitectónicos que tengan un enlace con la característica eliminada:

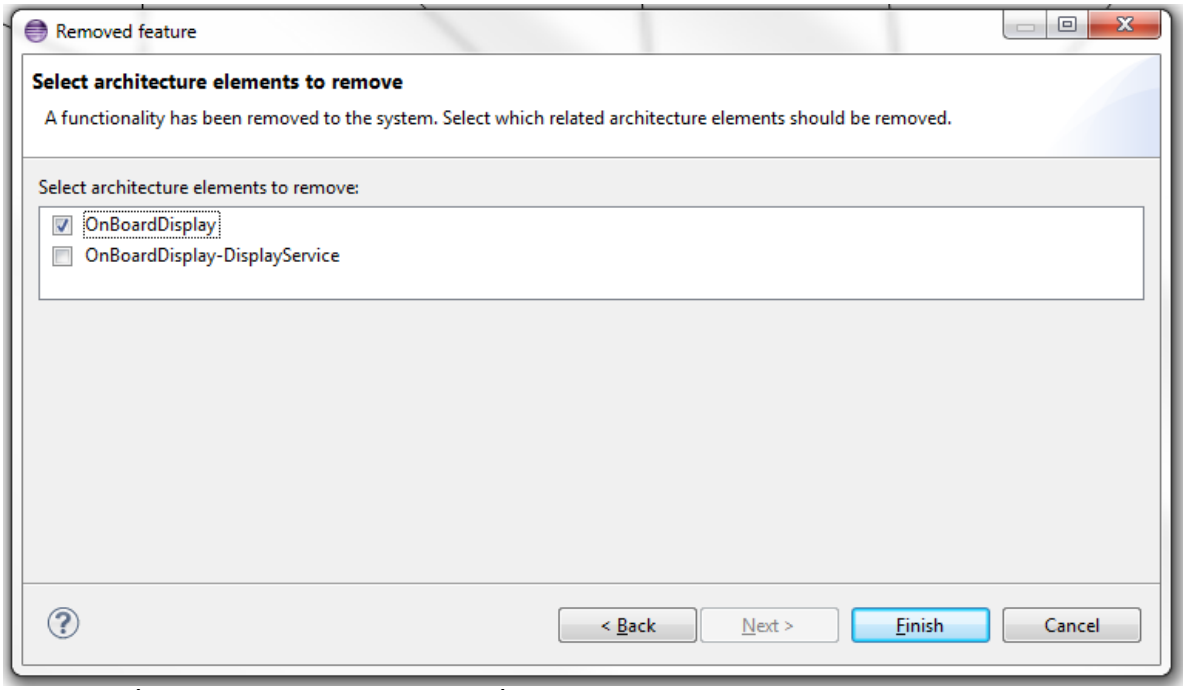

ILUSTRACIÓN 55 ELEMENTOS ARQUITECTÓNICOS A ELIMINAR

En esta ilustración se muestra el diálogo donde el ingeniero selecciona los elementos arquitectónicos a eliminar enlazados a la característica eliminada. En este caso, OnBoardDisplay y OnBoardDisplay-DisplayService.

Como se ve en la ilustración, se muestran los dos componentes de la arquitectura que tenían un enlace con la característica OnBoardDisplay (como se puede ver en la Ilustración 50 Modelo de enlace del Smart car). En este caso seleccionaremos únicamente el servicio "OnBoardDisplay". Este servicio será eliminado de la arquitectura y su enlace será eliminado del modelo de enlaces.

La razón de que sólo se seleccione uno de ellos es para mostrar que al eliminar la característica, además de eliminar los componentes arquitectónicos que el ingeniero seleccione también se elimina cualquier enlace que haga referencia a la característica eliminada. En la siguiente ilustración se puede ver como únicamente queda un enlace y el componente OnBoardDisplay ha sido eliminado de la arquitectura:

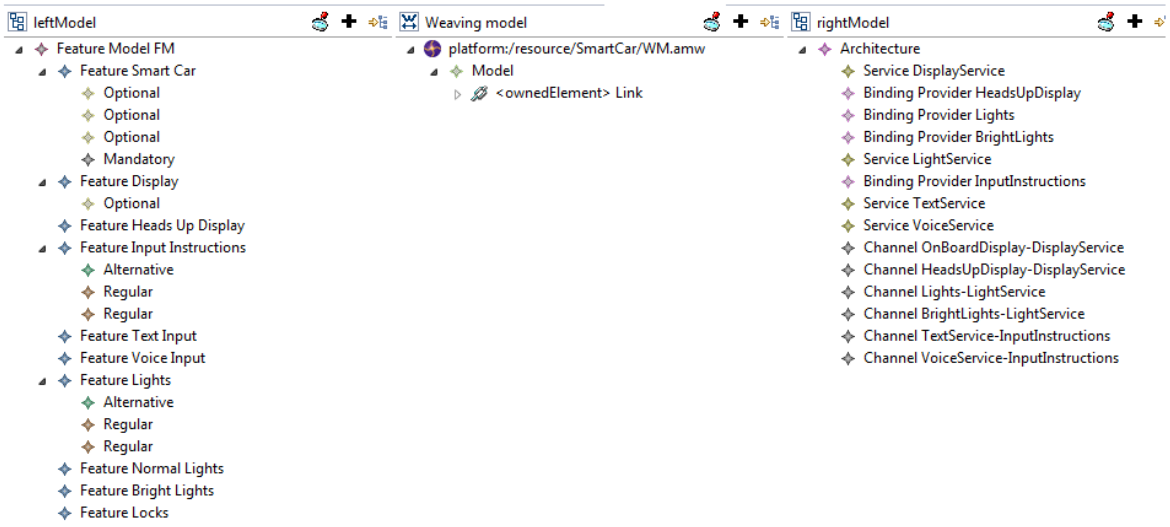

ILUSTRACIÓN 56 ONBOARDDISPLAY ELIMINADA

En esta ilustración se muestra el resultado del Modelo de Características, enlace y arquitectura tras la eliminación de la característica OnBoardDisplay

Y este es el Modelo de Resoluciones resultante una vez eliminada la resolución On Board automáticamente:

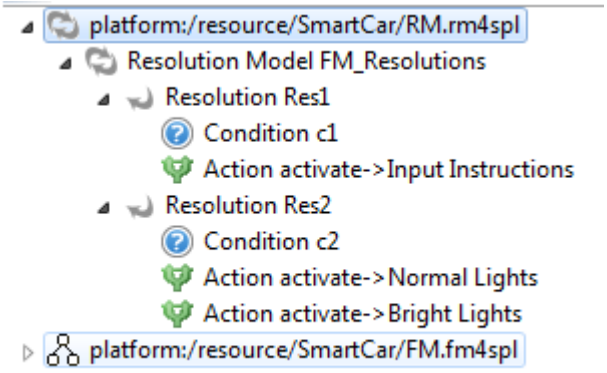

ILUSTRACIÓN 57 RESOLUCIÓN ON BOARD ELIMINADA

Modelo resultante tras la eliminación de la resolución On Board.

Una vez hemos propagado la eliminación de la característica OnBoardDisplay obtendremos el plan de evolución mediante el uso de la acción contextual “Evolve Evolution Plan”:

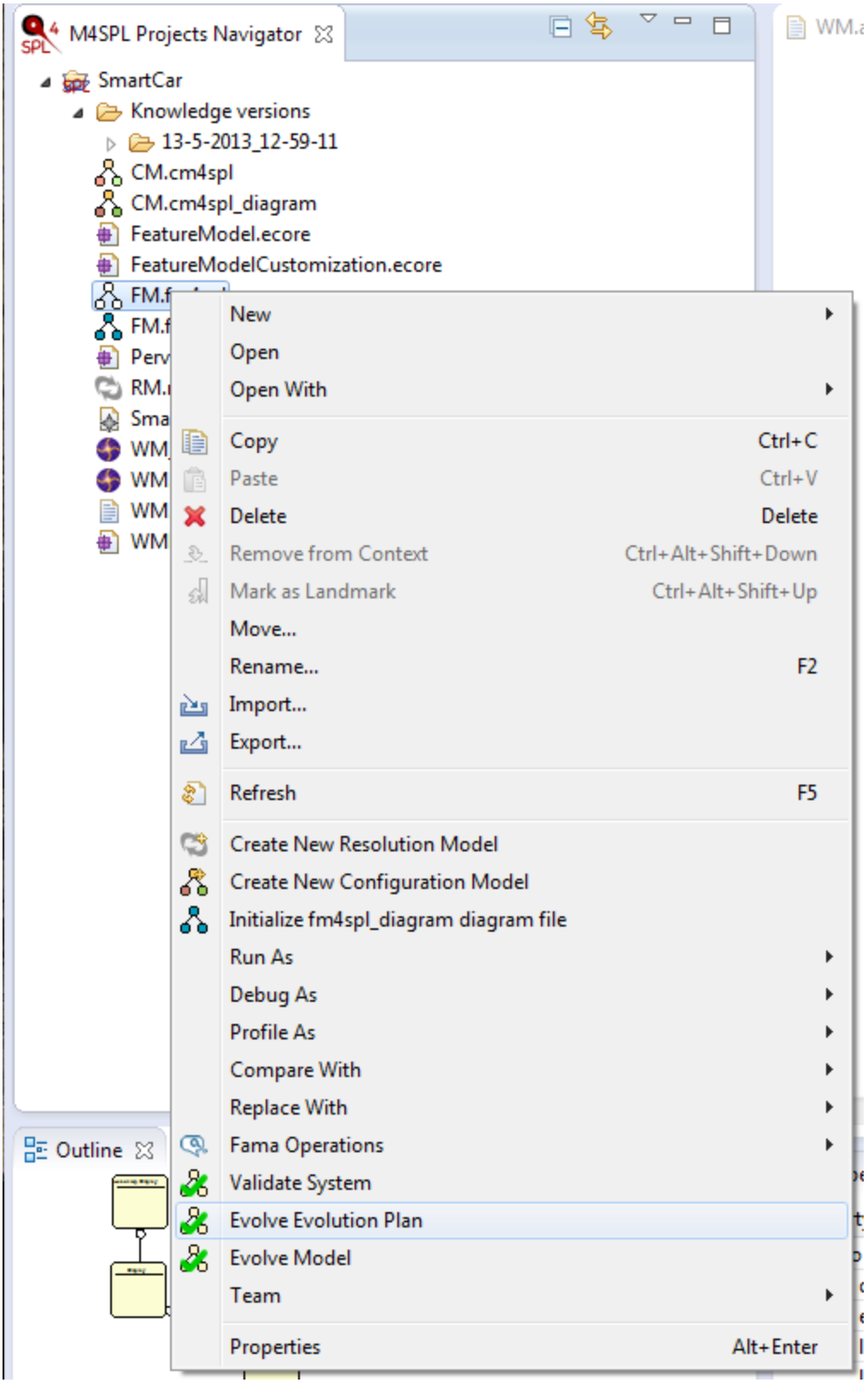

ILUSTRACIÓN 58 GENERAR PLAN DE EVOLUCIÓN

Una vez propagada la eliminación del OnBoardDisplay, el ingeniero genera el plan de evolución

Nos mostrará un diálogo para seleccionar con qué versión del conocimiento queremos comparar el estado actual del sistema:

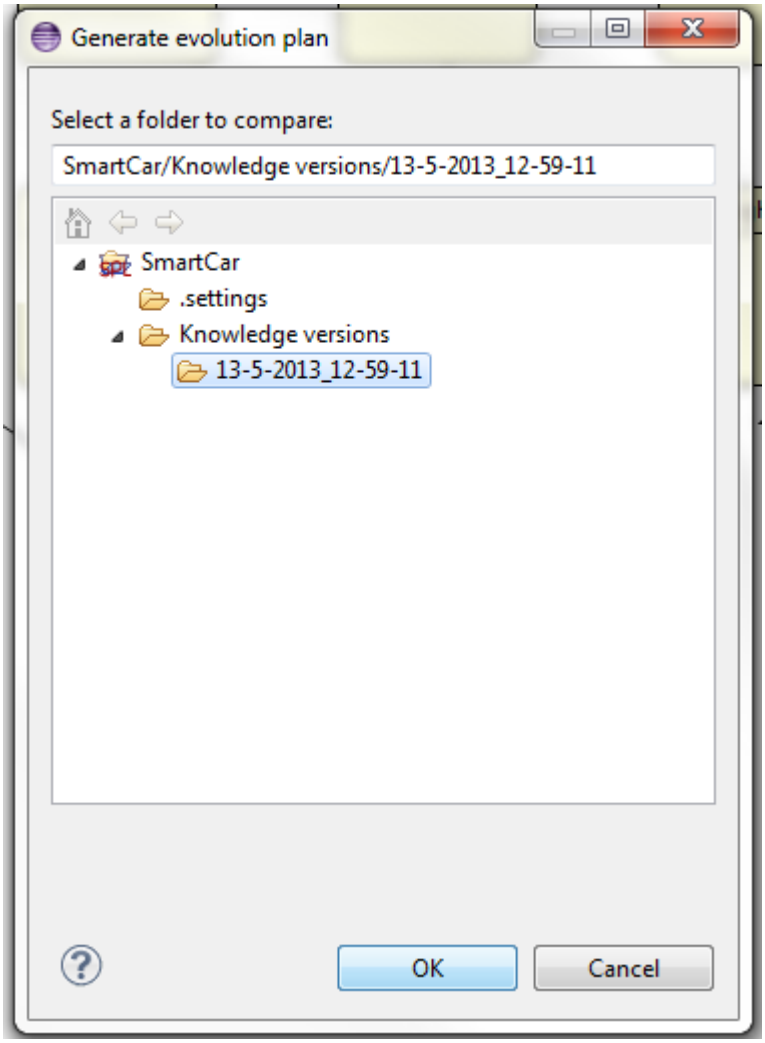

ILUSTRACIÓN 59 SELECCIÓN DE LA VERSIÓN DEL CONOCIMIENTO

Y finalmente genera el plan de evolución con el siguiente resultado:

```
*** EVOLUTION PLAN ***
-Feature Model FM-
 Increments:
 Decrements: Feature_OnBoardDisplay,MandatoryRelationship_null-From_Display-
To_OnBoardDisplay

-Resolution Model FM_Resolutions-
 Increments:
 Decrements: Resolution_On Board

-Architecture Model-
 Increments:
 Decrements: Service_OnBoardDisplay
```

6.3.2 ADICIÓN DE NUEVA FUNCIONALIDAD

Ahora se expondrá el caso en que se añada una funcionalidad nueva llamada “Location”. Esta funcionalidad será opcional y su objetivo es localizar la posición del vehículo.

En primer lugar, se realiza una copia del conocimiento del sistema como se comenta en el punto anterior.

Una vez hecho esto, se añade una característica al modelo y comienza el asistente. En primer lugar se pide el nombre de la característica (ver Ilustración 60):

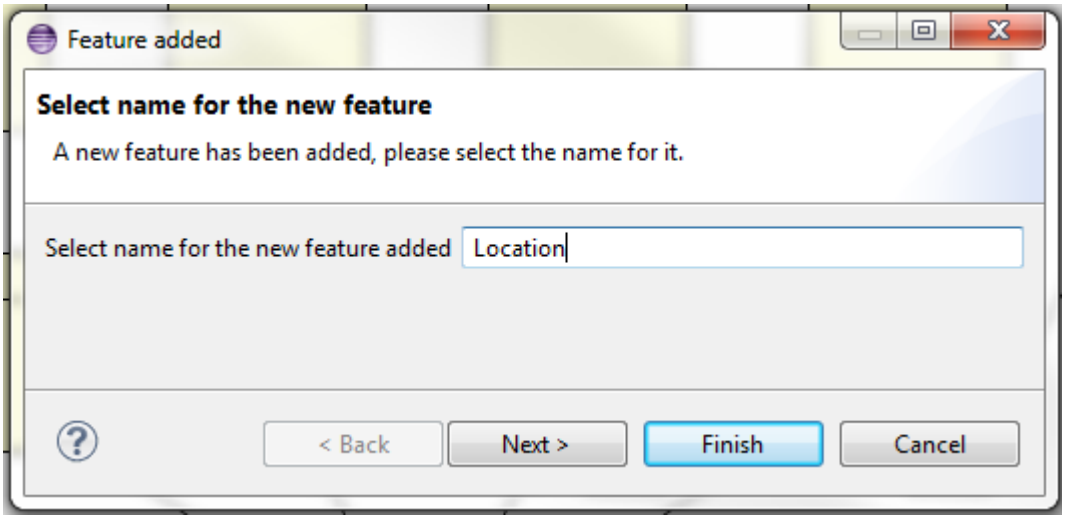

ILUSTRACIÓN 60 AÑADIR FUNCIONALIDAD LOCATION

A continuación se solicita el Modelo de Resoluciones (ver Ilustración 61):

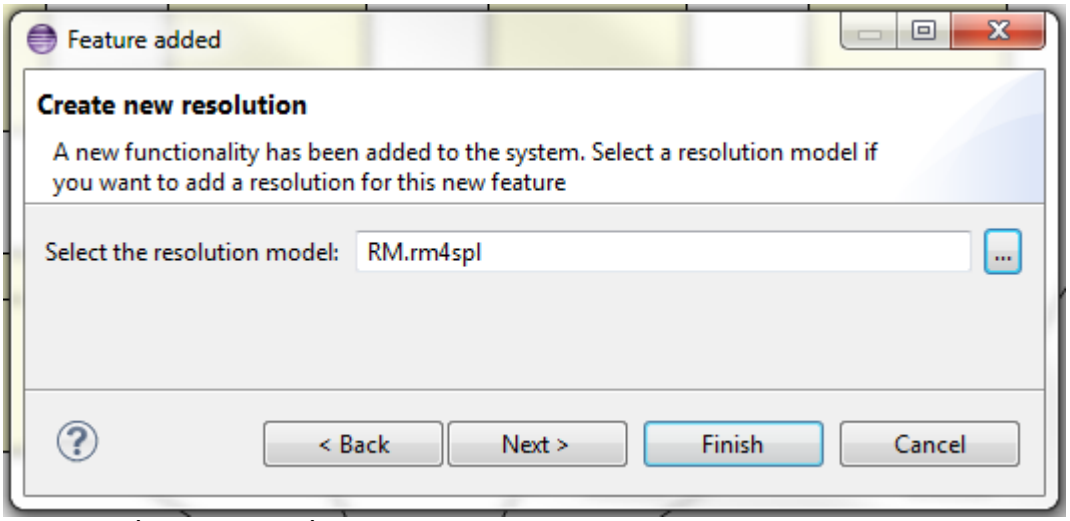

ILUSTRACIÓN 61 SELECCIÓN DEL MODELO DE RESOLUCIONES

Y una vez seleccionado, se añade una resolución para la nueva característica:

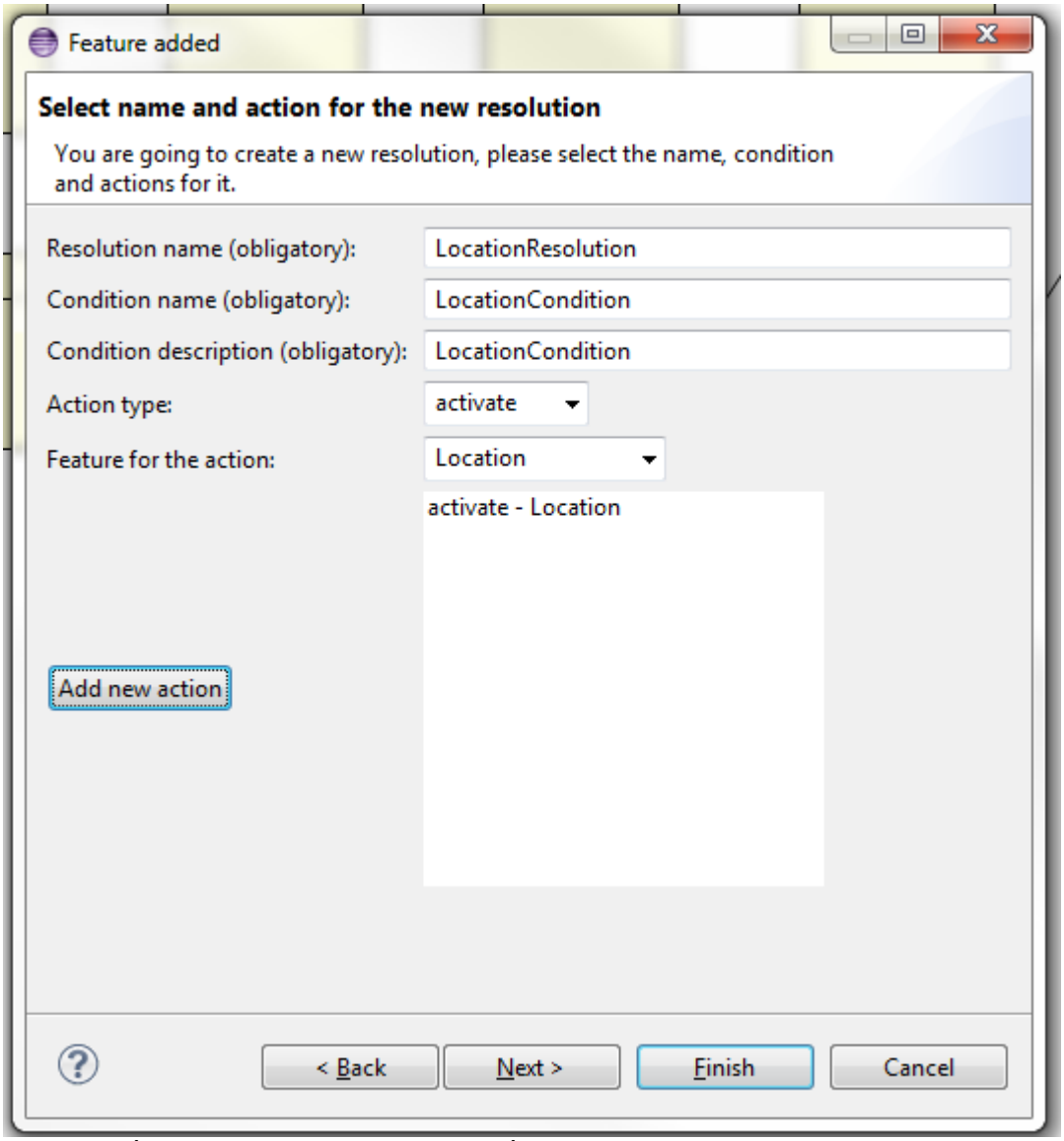

ILUSTRACIÓN 62 AÑADIR NUEVA RESOLUCIÓN LOCATION

Una vez añadida la resolución, el ingeniero opta entre añadir un nuevo elemento arquitectónico o seleccionar uno de los ya existentes. Dado que no existe ningún componente en la arquitectura para realizar la característica "Location". Por ello se escoge crear un nuevo componente:

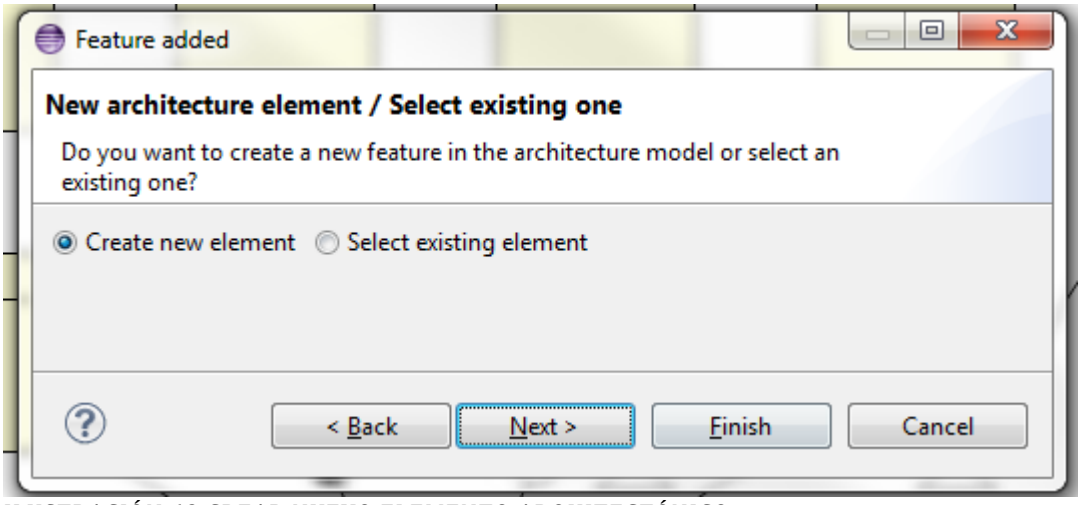

ILUSTRACIÓN 63 CREAR NUEVO ELEMENTO ARQUITECTÓNICO

Y se introducen sus datos:

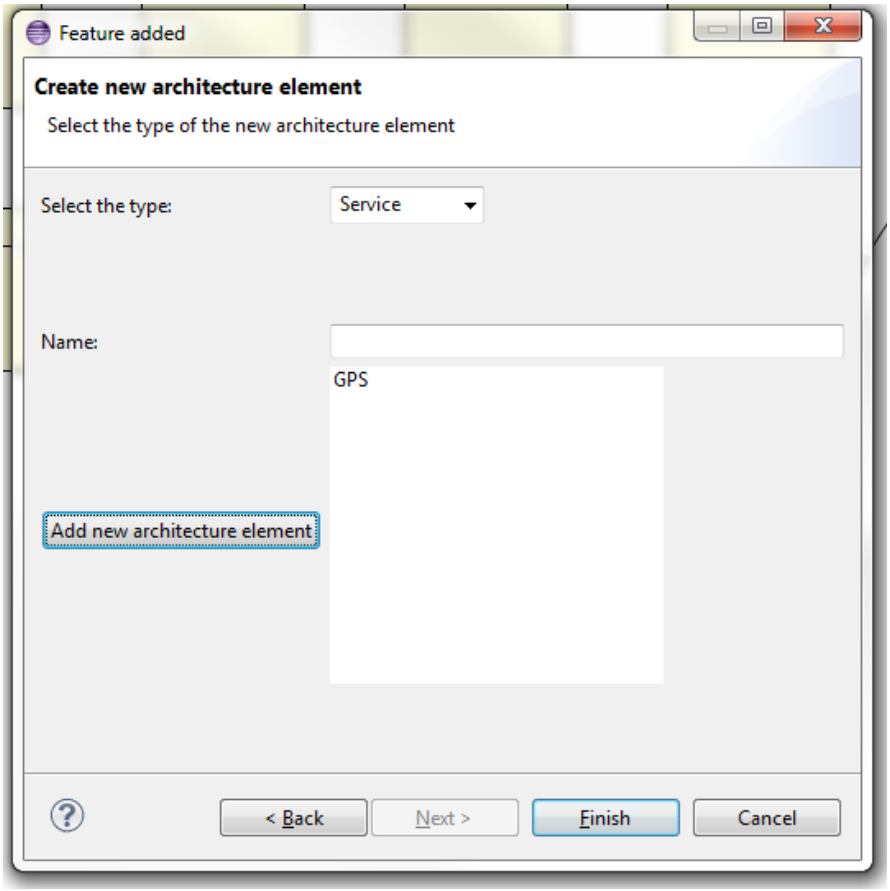

ILUSTRACIÓN 64 CREAR SERVICIO GPS

Una vez añadido el servicio de GPS, podremos generar el plan de evolución para comprobar los cambios realizados y este es el resultado:

```
*** EVOLUTION PLAN ***
-Feature Model FM-
 Increments: Feature_Location
 Decrements:

-Resolution Model FM_Resolutions-
 Increments: Resolution_LocationResolution
 Decrements:

-Architecture Model-
 Increments: Service_GPS
 Decrements:
```

7. CONCLUSIONES Y TRABAJO FUTURO

En este capítulo se presentan las conclusiones extraídas una vez desarrollada la propuesta que motivó esta tesis de máster, así como el trabajo que aún queda por delante una vez completada.

7.1 CONCLUSIONES

Las aproximaciones actuales para desarrollar sistemas auto-adaptables asumen que todos los posibles comportamientos del sistema han sido anticipados en tiempo de diseño. Sin embargo, con el paso del tiempo todo sistema software requiere evolucionar para dar soporte a cambios, mejoras y extensiones.

En este trabajo fin de máster se ha propuesto una aproximación y herramienta de soporte para evolucionar sistemas auto-adaptables mientras están en ejecución. La solución propuesta está basada en técnicas de ingeniería dirigida por modelos.

Concretamente, se proponen mecanismos y herramientas para ayudar a los ingenieros a minimizar errores y esfuerzo durante la evolución de sistemas auto-adaptables. Se han implementado una serie de patrones de evolución que guían el proceso de evolución. Estos patrones propagan los cambios realizados en un modelo al resto de modelos que constituyen el conocimiento del sistema.

Además se proporcionan refinamientos para garantizar automáticamente propiedades deseables de diseño y comportamiento en los sistemas evolucionados. Finalmente, la herramienta permite guardar copias de las distintas versiones del sistema y la generación del plan de evolución, para pasar de una versión a otra.

7.2 TRABAJO FUTURO

El trabajo desarrollado en esta tesis de máster no es un trabajo cerrado si no que existen distintas líneas de trabajo que deben ser abordadas para completar la propuesta. Como principales trabajos futuros se proponen los siguientes:

- Conexión con el motor de reconfiguración. Uno de los principales trabajos futuros es la conexión de la herramienta de evolución desarrollada con el motor de reconfiguración MoRE (Model-based Reconfiguration Engine) [33]. Con esta integración se conseguiría un funcionamiento completamente real de la evolución en ejecución de un sistema auto-adaptable.

- Implementación de generadores de modelo a código. Como trabajo futuro se planea implementar generadores de modelo a código para transformar automáticamente el plan de evolución generado en términos de acciones sobre los modelos al código fuente del sistema.

- Ampliación con una herramienta de control de versiones. Finalmente, se propone ampliar la herramienta implementada con una herramienta de control de versiones que permita gestionar las distintas versiones del conocimiento del sistema generadas y restaurar el sistema a versiones anteriores cuando sea necesario.

8. REFERENCIAS

[1] De Lemos, R., Giese, H., Müller, H., Shaw, M.: Software Engineering for Self-Adpaptive Systems: A second Research Roadmap. In: Software Engineering for Self-Adaptive Systems. No. 10431 in Dagstuhl Seminar Proceedings (2011)

[2] Brun, Y., Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Software engineering for self-adaptive systems. chap. Engineering Self-Adaptive Systems through Feedback Loops, pp. 48–70. Springer-Verlag, Berlin, Heidelberg (2009)

[3] IBM: An architectural blueprint for autonomic computing. (2003)

[4] Mens, T., Demeyer, S., Mens, T.: Introduction and roadmap: History and challenges of software
evolution. In: Software Evolution, pp. 1–11 (2008)

[5] J. Kramer and J. Magee, \Analysing dynamic change in software architectures: a case study," in Proc. of 4th IEEE international conference on configurable distributed systems, (Annapolis), May 1998.

[6] Blair, G., Bencomo, N., France, R.B.: Models@run.time. Comp. 42, 22–27 (2009)

[7] http://www.pros.upv.es/m4spl/

[8] Software Engineering. Addison-Wesley. ISBN 0-201-14229-5

[9] Paul Horn: Autonomic Computing, IBM's Perspective on the State of Information Technology

[10] Mellon, Clark, Futagami: Model Driven Development

[11] Introducing requirements traceability support in model-driven development of web applications
Pedro Valderas, Vicente Pelechano Department of Information Systems and Computation, Technical University of Valencia

[12] http://www.sparxsystems.com.au/platforms/mda_tool.html

[13] http://www.sei.cmu.edu/productlines/

[14] G. Chastek and J. D. McGregor. Guidelines for developing a product line production plan.Technical report, CMU/SEI, June 2002.

[15] Dynamic Software Product Line Approach using Aspect Models at Runtime: Tom Dinkelaker, Ralf Mitschke, Karin Fetzer, Mira Mezini

[16] J. van Gurp. Variability in software systems: the key to software reuse. PhD thesis, Dept. of Software
Engineering & Computer Science, Blekinge Institute of Technology, 2000.

[17] http://en.wikipedia.org/wiki/Feature_Model

[18] www.pros.upv.es/labs/projects/pervml/

[19] www.pros.upv.es

[20] Pervasive Computing: A Paradigm for the 21st Century. Debashis Saha (Indian Institute of Management-Calcutta), Amitava Mukherjee (IBM Global Services, Calcutta).

[21] http://www.eclipse.org/

[22] http://www.vogella.com/articles/EclipseEMF/article.html

[23] http://www.eclipse.org/modeling/emf/

[24] Mastering XMI: Java Programming with XMI, XML Timothy J. Grose, Gary C. Doney, Stephen A. Brodsky

[25] http://en.wikipedia.org/wiki/Observer_pattern

[26] http://www.eclipse.org/emf/compare/

[27] http://www.eclipse.org/gmt/amw/

[28] http://www.eclipse.org/gmt/

[29] http://www.moskitt.org/

[30] Moskitt4SPL: Tool Support for Developing Self-Adaptive Systems. XVII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2012). Almería, España, 2012

[31] http://es.wikipedia.org/wiki/Desarrollo_iterativo_y_creciente

[32] http://java.dzone.com/articles/emf-reading-model-xml-%E2%80%93-how

[33] Cetina, C., Giner, P., Fons, J., & Pelechano, V. (2009). Autonomic computing through reuse of variability models at runtime: The case of smart homes.*Computer*, *42* (10), 37-43.