Universidad Politécnica de Valencia


Facultad de Informática


Departamento de Sistemas Informáticos y de Computación

Generación automática de código y ejecución de arquitecturas software orientadas a Aspectos

Proyecto Final de Carrera
Julio 2007

Autor:
Javier Guillén Martín

Dirigido por:
Isidro Ramos Salavert
Jennifer Perez Benedí

Agradecimientos

A mis padres y a mis hermanos por estar siempre a mi lado y apoyarme en todo momento.

A mis amigos por estar a mi lado cuando mas hace falta y por ser casi una familia

A mis amigos de la Facutlad, ya que estos años a su lado nos han convertido en mas que compañeros.

A Jennifer por todo lo que he aprendido a su lado y sobre todo por todo el esfuerzo realizado

A Isidro por haber compartido su tiempo y concocimiento.

A Cristobal, Carlos, Rafa, Nour e Ismael ya que este proyecto continua todos sus trabajos y por su compañerismo y amistad.

A todo el grupo ISSI por el apoyo a este proyecto.

INDICE

TABLA DE FIGURAS

Capítulo 1

INTRODUCCIÓN

Contenido del capítulo

1 INTRODUCCIÓN

1.1 Motivación

PRISMA es un enfoque de desarrollo que tiene como objetivo principal el dar soporte al desarrollo de arquitecturas software orientadas a aspectos e independientes de tecnología. PRISMA ofrece un modelo que combina las arquitecturas software y el Desarrollo de Software Orientado a Aspectos (DSOA), y un Lenguaje de Descripción de Arquitecturas Orientado a Aspectos, lo que permite dar soporte al desarrollo de arquitecturas software complejas aumentando el mantenimiento, reusabilidad, trazabilidad, etc.

Para dar soporte a este enfoque se ha desarrollado un prototipo de herramienta CASE que hasta el momento esta compuesta por dos herramientas. La primera de ellas, PRISMANET, define un middleware sobre el cual se pueden lanzar a ejecución arquitecturas software basadas en PRISMA. La segunda herramienta, PRISMADSL, permite la especificación de aplicaciones en base al modelo PRISMA. Esta especificación se realiza modelando de forma grafica los modelos arquitectónicos orientados a aspectos de dichas aplicaciones. Hasta ahora el paso desde el modelado realizado en PRISMADSL hacia la generación de código en PRISMANET se hacia de forma manual, es decir, el usuario debe implementar manualmente el código correspondiente a la especificación desarrollada.

Con el objetivo de paliar este problema de trazabilidad entre los modelos arquitectónicos y el código definido, y de seguir la tendencia en la comunidad internacional en lo que respecta a la materia de Ingeniería del Software, se ha optando por dar soporte al Desarrollo de software Dirigido por Modelos (MDD) mediante la aplicación de técnicas de generación automática de código (Paradigma de la Programación Automática). De este modo, a partir de los modelos es posible generar código ejecutable sin necesidad de que un programador escriba ni una línea de código.

1.2 Objetivos

El objetivo principal de este proyecto final de carrera es el desarrollar un compilador de modelos que una la herramienta PRISMADSL con la herramienta PRISMANET y así solventar el gap entre modelos y código que presenta la metodología PRISMA. Por lo tanto, este proyecto pretende dar soporte a las necesidades de la generación automática de código de PRISMA CASE.

Aunque el principal objetivo del proyecto es la realización de un compilador de modelos para PRISMA, tambien se han realizado una serie de tareas relacionadas con el middleware PRISMANET para realizar la generación de código de una forma más sencilla, así como otras tareas de optimización mediante técnicas de refactorización.

En la parte del proyecto relacionada con la realización del compilador, se quiere dar un paso más adelante, y no proveer una mera generación de código de los modelos

arquitectónicos y de los aspectos, ya que estos son tipos y no son ejecutables sobre .NET. Por ello, se quiere dar soporte a la configuración de la arquitectura para un sistema específico, y así poder configurar sus instancias y su topología y generar su código, para posteriormente lanzarla a ejecución.

Por último, se quiere proveer al arquitecto de una IGU genérica que le permita la interacción con la arquitectura software definida mientras está en ejecución. De este modo, el arquitecto podrá comprobar su funcionamiento sin al necesidad de diseñar la interfaz gráficas de usuario de la aplicación. Esto es algo importante, ya que en este proyecto se aborda la generación automática de código en base a la lógica de la aplicación, pero no en base a la interfaz gráfica de usuario, algo que quedaría pendiente como trabajo futuro.

Con todos estos objetivos lo que se prentende es dar forma a una herramienta, llamada PRISMACASE, que ofrecerá al usuario de la siguiente funcionalidad:

- Modelado Gráfico
- Generación Automática de Código
- Lanzamiento a ejecución del modelo
- Interactuar con el modelo definido

Como ejemplo de la utilización de la herramienta resultante se ha llevado a cabo la resolución de un caso de estudio. Se trata de modelar el funcionamiento de un robot, que es un robot a pequeña escala de los robots que son utilizados en tareas de reparación y mantenimiento de cascos de buques. Una vez modelado el sistema, se deberá generar automáticamente el código y ver cómo es posible interactuar a través del código generado con dicho robot.

1.3 Estructura

El proyecto se encuentra dividido en los siguientes capítulos:

En el capitulo 2, PRISMA, se muestra el enfoque PRISMA y su modelo, los cuales están soportados por la herramienta PRISMACASE. PRISMACASE esta formada por PRISMANET y PRISMADSL. En este capítulo se van ha presentar cada una de ellas, realizando una introducción a las mismas y dando una visión de su estado previo a la realización de este proyecto.

El capitulo 3 describe el robot *TeachMover*, el cual ha sido utilizado como caso de estudio para poder modelar una arquitectura software orientada a aspectos y comprobar el correcto funcionamiento de la herramienta definida mediante la generación automática de su código y su posterior ejecución.

En el capitulo 4 se explican las mejoras realizadas en el *midlleware* PRISMANET mediante técnicas de refactoring para así proporcionar facilidades en su extensión y en la generación automática de código.

El capítulo 5 presenta el desarrollo del compilador para PRISMA mediante una serie de patrones software y cómo éste ha sido integrado en PRISMANET. También en este capítulo muestra la Consola que se ha desarrollado para interactuar con el sistema definido. Por último, el capítulo detalla cómo se ha dado soporte a la inclusión de COTS en el compilador prisma.

Finalmente, en el capítulo 6 se enumeran las conclusiones del proyecto, así como los posibles trabajos futuros.

Junto a todos estos capítulos se añaden dos anexos (A y B) en los que se muestran los resultados del uso de la herramienta para el caso de estudio del robot *TeachMover*. El anexo A presenta los modelos gráficos de tipos y de configuración a partir de los cuales se ha generado el código y el anexo B presenta el código generado.

Capítulo 2

PRISMA

Contenido del capítulo

2 PRISMA

En este capitulo se muestra el enfoque PRISMA y su modelo, los cuales estan soportados por la herramienta PRISMACASE. PRISMACASE esta formada por PRISMANET y PRISMADSL. En este capítulo se van ha presentar cada una de ellas, realizando una introducción a las mismas y dando una visión de su estado previo a la realización de este proyecto.

2.1 El modelo PRISMA

El modelo PRISMA permite la definición de sistemas software complejos. En dicho modelo se integran dos aproximaciones de desarrollo de software como son el Desarrollo de Software Basado en Componentes (DSBC) y el Desarrollo Software Orientado a Aspectos (DSOA), con lo que se integran las ventajas que cada una de estas aproximaciones para obtener un mejor desarrollo de sistemas software.

Debido a que las arquitecturas software utilizan componentes para describir sistemas complejos, PRISMA se presenta como un enfoque para el desarrollo de arquitecturas software orientadas a aspectos.

En PRISMA se introducen los aspectos como ciudadanos de primera especie en su Lenguaje de Definición de Arquitecturas (LDA). Por este motivo, PRISMA proporciona un Lenguaje de Descripción de Arquitecturas Orientada a Aspectos (LDAOA), para especificar arquitecturas software. PRISMA crea un nuevo concepto que modela los distintos asuntos comunes de los requisitos de un sistema software, como pueden ser (distribución, seguridad, coordinación, etc). Todos estos asuntos comunes serán aspectos arquitectónicos que conformarán cada uno de los componentes del sistema software.

Un aspecto en PRISMA es un asunto que entrecruza la arquitectura software. Esto viene dado por el hecho de que un mismo aspecto puede ser importado por varios elementos arquitectónicos de una arquitectura software. (ver Figura 1)

El poder utilizar un mismo aspecto en distintos elementos arquitectónicos posibilita un mayor grado de reutilización dentro del desarrollo de la arquitectura software.

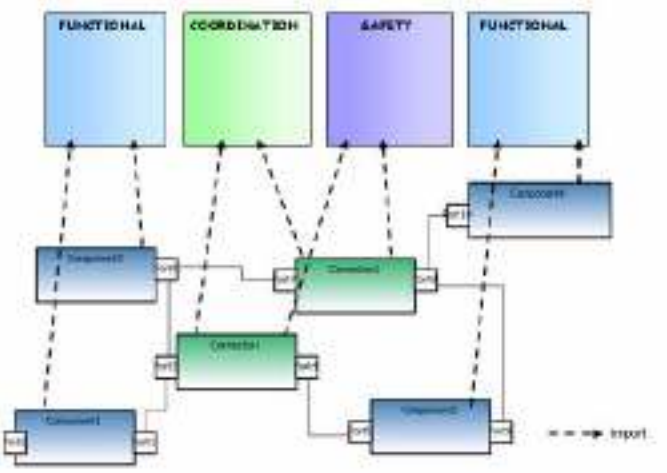


Figura 1. Arquitecturas Orientadas a Aspectos PRISMA

Un elemento arquitectónico en PRISMA puede observarse según dos vistas: una interna y otra externa. La vista externa de un elemento arquitectónico encapsula su funcionalidad como una caja negra, desde la cual se publica una serie de servicios que son ofrecidos por dicho elemento arquitectónico (ver Figura 2). Dichos servicios están agrupados en diversas interfaces que publican dichos servicios mediante los puertos de cada elemento arquitectónico. Cada puerto tiene asociada una interfaz que contiene una serie de servicios que pueden ser provistos o requeridos a través del puerto. Por lo tanto, los puertos son el punto de interacción con los elementos arquitectónicos.

Figura 2. Figura Vista caja negra de un elemento arquitectónico

La vista interna (caja blanca) de un elemento arquitectónico se puede ver como un prisma (ver Figura 3). Cada cara del prisma es un aspecto que el elemento arquitectónico importa. De esta forma, los elementos arquitectónicos estan representados por un conjunto de aspectos y sus relaciones entre si, llamadas weavings.

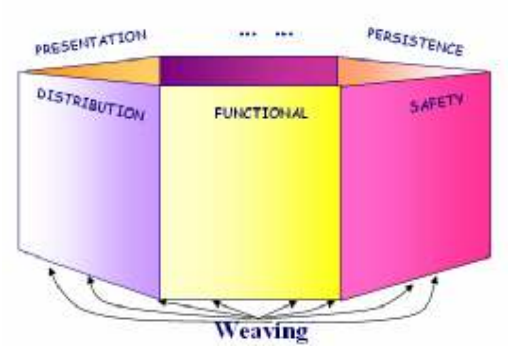


Figura 3. Figura Vista caja blanca de un elemento arquitectónico

Las relaciones de weaving entre los aspectos permiten establecer como la ejecución de un servicio en un aspecto puede desencadenar la ejecución de servicios en otros aspectos del elemento arquitectónico. Para tener un mayor grado de independencia entre

los aspectos, las relaciones de weavings se definen fuera de ellos. Los elementos arquitectónicos son los encargados de definir los weavings entre el conjunto de aspectos que los forman. De este modo dos elementos arquitectónicos con los mismos aspectos pueden tener comportamientos distintos dependiendo de los weavings que tengan definidos entre ellos.

La comunicación entre la vista como caja negra y como caja blanca es posible a través de las interfaces. Estas estan asociadas a los puertos y son utilizadas por los aspectos. De este modo la solicitud de un servicio entra en el elemento arquitectónico utilizando los puertos y es procesada por un aspecto que usa la misma interfaz. (ver Figura 4)

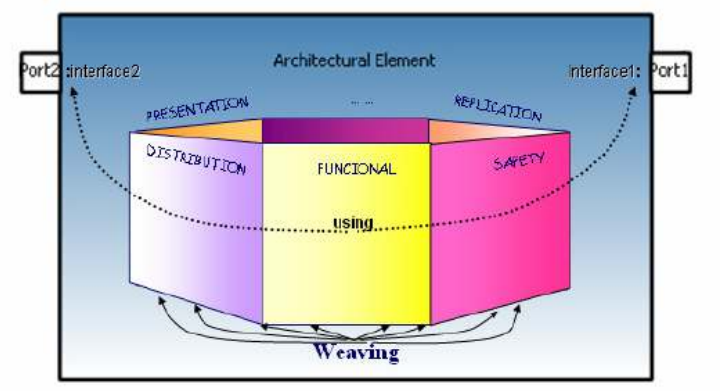


Figura 4. Figura Comunicación entre caja negra y caja blanca

A continuación se muestran cada uno de los elementos que forman parte del modelo PRISMA.

2.1.1 Interfaces

Una interfaz publica un conjunto de servicios. Describe la signatura de los servicios que pueden ser invocados o requeridos a través de una interfaz.

La signatura de lo servicios especifica su nombre y sus parámetros. Para estos últimos hay que indicar el tipo así como si el parametro es de entrada o de salida. Los parámetros de entrada son los párametros necesarios para poder ejecutar el servicio en el cual estan definidos. La forma de devolver resultados en la invocación de un servicio se realiza mediante el uso de los parámetros de salida, los cuales obtendrán un valor tras la ejecución del servicio correspondiente.

2.1.2 Aspectos

Un aspecto define la estructura y el comportamiento de un asunto específico dentro del sistema software. Algunos ejemplos de estos asuntos del sistema son: *functionality*, *coordination*, *safety*, *distribution*, etc.

La estructura de los aspectos se define en base a un conjunto de atributos. El valor que tengan los atributos en cualquier momento de la ejecución del aspecto definen el estado

del mismo. Pueden definirse restricciones (*constraints*) que deben satisfacerse durante la ejecución de un aspecto.

En un aspecto se declara el uso de interfaces y se define la semántica para cada uno de los servicios que son publicados a través de dichas interfaces. Esta semántica captura cuando un servicio puede ser ejecutado, como cambia el estado tras su ejecución y el orden en el cual es ejecutado. Dicha semántica se define mediante precondiciones, valuaciones y played_roles, respectivamente.

Precondición: Establece una condición que debe satisfacerse para poder proceder a la ejecución de un servicio.

Valuación: Las valuaciones establecen como la ejecución de un servicio afentan al cambio de estado del aspecto.

Played_roles: Los played roles indican como y cuando pueden ejecutarse los diversos servicios.

A parte de los servicios de las interfaces, un aspecto puede definir servicios privados. Estos son definidos internamente en un aspecto sin poder ser accedidos desde fuera del mismo. El uso de servicios privados permite estructurar la funcionalidad dentro de un aspecto, con lo cual se obtiene un código más legible. Mediante el uso de los servicios privados se encapsula cierta funcionalidad que es interna al funcionamiento de un aspecto y que solo puede ser accedida por el mismo.

El comportamiento de un aspecto se define a través del protocolo. En el protocolo se define como pueden ser ejecutados los diversos servicios. El protocolo puede verse como una maquina de transición de estados, en la cual en cada nodo se especifican que servicios pueden ejecutarse y por que played_role, así como al siguiente estado en el cual se quedará el aspecto.

Dentro de los servicios que se pueden definir dentro de un aspecto, hay unos servicios especiales que son las transacciones. Las transacciones establecen la ejecución de varios servicios pero realizandose siguiendo las propiedades ACID: Atomicidad, Consistencia, Aislamiento y Durabilidad.

2.1.3 Elementos arquitectónicos

PRISMA tiene tres tipos de elementos arquitectónicos: componentes, conectores y sistemas. Los componentes y conectores son simples. Los sistemas representan componentes complejos de la arquitectura software.

Un componente es un elemento arquitectónico que captura la funcionalidad del sistema software y no actua como coordinador de otros. En cambio, un conector es un elemento arquitectónico que actua como coordinador entre otros elementos arquitectónicos.

Los componentes o conectores no tienen referencias entre ellos sobre sus conexiones. Esto es posible gracias a la definición de canales de comunicación entre los componentes y los conectores, los cuales tienen las referencias hacia los componentes y

conectores que conectan. El concepto de PRISMA que proporciona esta funcionalidad son los *Attachments*. Cada *Attachemnt* se define como la comunicación entre un puerto de un conector y un puerto de un componente, (ver Figura 5).

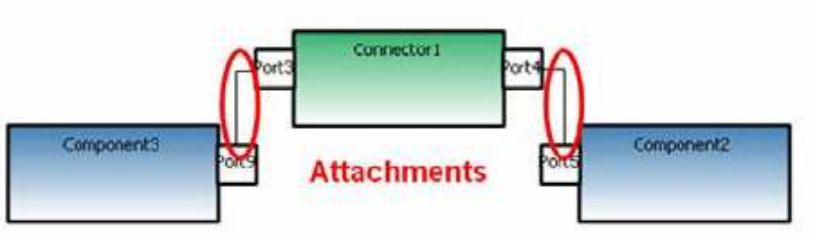


Figura 5. Attachments

Un sistema PRISMA es un componente que incluye un conjunto de elementos arquitectónicos (conectores, componentes y otros sistemas) que estan correctamente conectados a través de *Attachments*. Un sistema como todo componente puede tener un comportamiento propio definido por sus propios aspectos y weavings entre ellos mismos. Para poder realizar la conexión entre los elementos arquitectónicos que forman un sistema y el mismo, se utilizan las relaciones de comunicación llamadas *Bindings*. Estas comunicaciones son configuradas entre un puerto del sistema y un puerto de alguno de los elementos arquitectónicos que forman el sistema, (ver Figura 6).

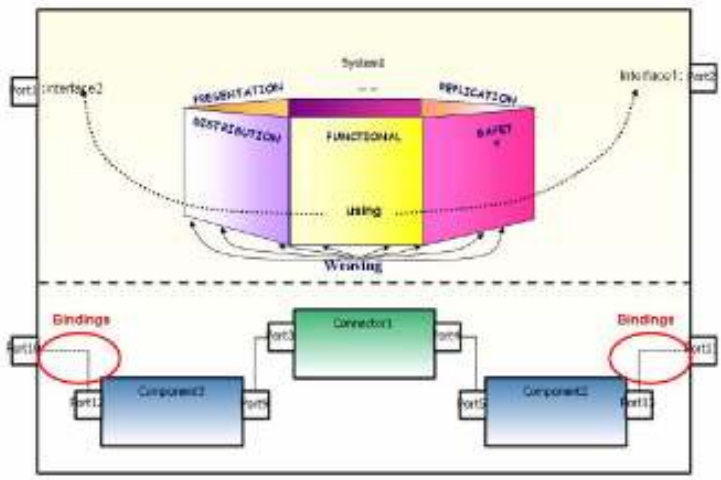


Figura 6. Figura Sistemas

2.2 PRISMACase

PRISMACase es una herramienta que se ha desarrollado para dar soporte al enfoque de desarrollo PRISMA para la definición de arquitecturas software orientadas a aspectos. El objetivo de esta herramienta es proporcionar al usuario un marco de trabajo para el desarrollo y ejecución de arquitecturas basadas en PRISMA.

Esta herramienta sigue el enfoque del Desarrollo Dirigido por Modelos (DDM). DDM propone la definición de los sistemas software a un nivel más abstracto que los lenguajes de programación. Este nuevo nivel son los modelos, que hasta ahora jugaban

un papel descriptivo en las primeras fases del desarrollo software, pero que no permitían la obtención de especificaciones ejecutables. El Object Manegement Group (OMG) propone MDA (Arquitecturas Dirigidas por Modelos) para dar soporte a DDM. Por otra parte, Microsoft ha hecho otra propuesta que son las Softwares Factories. PRISMACASE sigue este último enfoque.

La idea principal de esta nueva forma de desarrollar sistemas software es la de realizar modelos y obtener automáticamente código ejecutable de los mismos. Para ello es necesario aplicar técnicas de generación automatica de código. Por lo tanto, PRISMACASE utiliza dichas técnicas para la generación automática de código en C#.

En esta sección se muestra la estructura de esta herramienta, así como el estado previo a la realización de este proyecto, indicando en cada caso las nuevas tecnologías que han sido necesarias para su desarrollo.

2.2.1 DSL Tools

En primer lugar y antes de que se muestren las partes que forman PRISMACASE, se muestra la tecnología que ha sido necesaria para su desarrollo.

Como se ha mostrado anteriormente el proposito de PRISMACASE es dar soporte al desarrollo se software basado en modelos. En el caso de PRISMACASE la tecnología utilizada ha sido DSL Tools (Domain Specific Languages tools) ya que es la propuesta por Microsoft para dar soporte a las Software Factories.

DSL Tools es un plugging de Visual Studio 2005 que esta diseñado para la definición de lenguajes específicos de dominio. Esta herramienta permite al usuario definir un modelo especifico de dominio sobre el cual crear posteriormente un modelo concreto y a partir de estos últimos generar ficheros con la información que uno desee, según el objetivo que se persiga.

La herramienta permite la definición de una solución donde se especifica el modelo a utilizar, su representación gráfica y las transformaciones para la generación de código.

Cuando se crea un nuevo proyecto, el asistente permite elegir cuatro tipos de proyectos:

- *Minimal Language:* Permite la definición de un modelo específico por parte del usuario.
- *Activity Diagrams:* Crea un proyecto con las definiciones básicas de los diagramas de actividad de UML.
- *Class Diagrams:* Crea un proyecto con las definiciones básicas de los diagramas de clases de UML.
- *Use Case Diagrams:* Crea un proyecto con las definiciones básicas de los diagramas de casos de uso de UML.

Todo proyecto en DSL Tools tiene la siguiente estructura. Consta de dos proyectos bien diferenciados como son el “Designer” y “Domain Model”. Estos dos proyectos

posibilitan al desarrollador la definición de un modelo sobre el cual construir modelos conforme al modelo de dominio definido y obtener su código automáticamente.

A continuación se ve con más detalle cada uno de estos proyectos para ver que tareas se pueden realizar en cada uno de ellos.

En primer lugar, el "Domain Model", es el proyecto en el cual se define el modelo de dominio a utilizar. Se dispone de un fichero llamado DomainModel.dslt, el cual proporciona un papel tapiz en el cual se puede ir dibujando cada uno de los elementos necesarios y sus relaciones entre si.

En la Figura 7 se puede observar los elementos de que se disponen para la inserción de cada uno de estos conceptos: clases, relaciones, herencia, etc.

Figura 7. Figura - Toolbox DSL Tools

Las clases pueden tener propiedades con un valor y un tipo especificado. Ademas pueden componerse entre ellas. Entre las relaciones que se disponen para relacionar unas clases con otras hay dos que son de tipo asociación y otro tipo especialización, es decir la realización de una jerarquía de herencia como en cualquier lenguaje orientado a objetos.

En las relaciones establecidas entre las clases del modelo se puede definir información referente a la cardinalidad de los distintos elementos que participan en la misma, así como si se pretende tener navegabilidad bidireccional o no. Esta propiedad es importante, ya que posteriormente permitirá recorrer el modelo que genere el usuario. Si de estas relaciones emergen nuevas propiedades, se pueden mostrar estas relaciones como clases, y como tal, se les pueden asociar tanto propiedades, como una composición de clases.

Cada una de las clases definidas en el modelo, generan a su vez una serie de ficheros que contienen una definición parcial de estas clases. El desarrollador puede ampliar la funcionalidad que por defecto tienen. En el caso de PRISMADSL, se han implementado una serie de reglas de verificación, así como un conjunto de reglas de borrado, inserción y modificación de elementos en el modelo. Estas ampliaciones al modelo tienen como objetivo facilitar, al usuario final de la herramienta, el uso de la misma.

Una vez definidos los elementos del modelo, se pueden asociar las representaciones graficas que se deseen a través del proyecto "Designer", el cual a través de un fichero XML permite establecer esta conexión.

Ejecutando este proyecto, aparece un nuevo proyecto en el cual se puede realizar la creación del modelo de dominio antes definido. Para realizar el modelado se utilizan principalmente tres componentes:

- *ToolBox*: Se pueden seleccionar cada uno de los elementos del modelo de dominio que tienen representación gráfica para ir definiendo el modelo y sus relaciones.

- *Solution Explorer*: En el se pueden ver en forma de árbol cada uno de los elementos del modelo de dominio y sus correspondientes instancias en el modelo. A través del mismo se pueden realizar más opciones del modelado para todas aquellas características que no tienen representación gráfica.

- *Properties*: A través de las *properties*, como su nombre indica, se pueden consultar y/o modificar las diferentes propiedades del elemento que este seleccionado en cada momento, ya sea en la parte gráfica o mediante el *Solution Explorer*.

Una vez realizado el modelado utilizando la herramienta, lo que se pretende es obtener información del mismo en el formato que el desarrollador desee. Por ejemplo se puede generar HTML, C#, C++, Java, etc.

Para obtener la transformación del modelo al código se utilizan una serie de plantillas de transformación de código. En estas plantillas se define por un lado el código necesario para poder recorrer el modelo creado, basándose en el modelo definido previamente. Por otro lado se define el código que se obtendrá como resultado de la transformación. Al inicio de estas plantillas se especifica información sobre el tipo de fichero resultante.

2.2.2 Estructura de PRISMACase

PRISMACase esta formada por dos herramientas ya definidas como son PRISMANET y PRISMADSL. A continuación veremos con más detalle que función tiene cada una de ellas.

Brevemente y como introducción, PRISMADSL permite el modelado de arquitecturas PRISMA y el Middleware PRISMANET permite la ejecución de código basado en PRISMA sobre la plataforma .NET. El problema viene dado por la falta de unión de las dos herramientas, ya que el usuario puede modelar arquitecturas software según PRISMA pero tiene que generar de forma manual el código que corresponde al modelo creado, para su posterior ejecución sobre el middleware PRISMANET, (ver Figura 8).

El principal objetivo de este proyecto es crear el nexo de unión entre estas dos aplicaciones, generando de una forma automática código ejecutable sobre el middleware PRISMANET a través de los modelos definidos por el usuario en PRISMADSL.

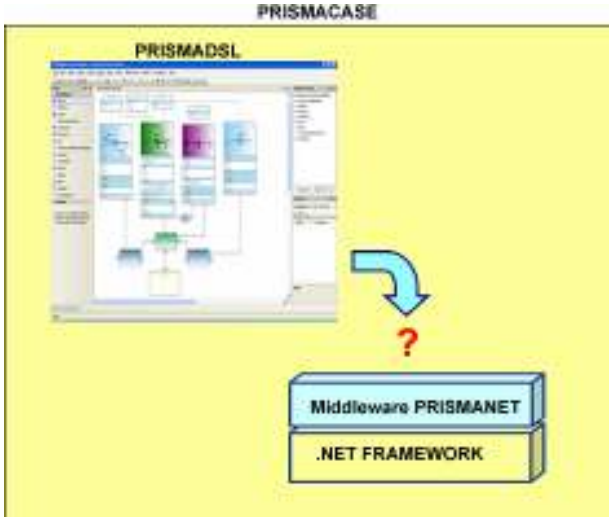


Figura 8. Figura - PRISMACase

2.2.2.1 PRISMANET

Uno de los objetivos que persigue el modelo PRISMA es poder generar automáticamente aplicaciones ejecutables desde sus especificaciones a través de modelos. La plataforma específica elegida ha sido la plataforma .NET de Microsoft© (como pudiese ser cualquier plataforma tecnológica existente). Sin embargo, la plataforma .NET no da soporte a las primitivas que PRISMA proporciona como en el caso de los aspectos, los elementos arquitectónicos, etc. Por este motivo, se ha tenido que desarrollar un middleware en C# para dar soporte a estas primitivas denominado PRISMANET.

PRISMANET es la capa que implementa las correspondencias entre el modelo PRISMA y la plataforma .NET. Para construir una arquitectura PRISMA ejecutable el compilador de modelos PRISMA utilizará PRISMANET para extender las clases que corresponden con las primitivas PRISMA, incluyendo en ellas los servicios necesarios para su ejecución. PRISMANET proporciona, a las aplicaciones PRISMA generadas, la ejecución concurrente de los elementos arquitectónicos, la comunicación distribuida, y en un futuro cercano la movilidad y la evolución dinámica.

El diseño realizado da soporte a la gestión de los aspectos, los elementos arquitectónicos, *attachments* y *bindings*. Además provee a los elementos arquitectónicos la concurrencia, la comunicación distribuida y la reutilización. Éstas son las características del modelo arquitectónico PRISMA que proporcionan a sus elementos una gran potencia expresiva.

En la presente sección se pretende dar una visión de cómo se ha implementado el middleware PRISMANET en la plataforma .NET, para conocer el estado previo a la realización de algunas tareas que pueden verse en capítulos posteriores de este proyecto.

Diseño de la Arquitectura

La plataforma sobre la que se ha creado el middleware es el framework .NET de Microsoft©. La arquitectura del framework .NET está dividida en tres capas. .NET

proporciona un lenguaje intermedio (MSIL, MicroSoft Intermediate Language) que es interpretado por una máquina virtual (CLR, Common Languaje Runtime) para la obtención del código nativo. Los diferentes compiladores de lenguajes que funcionan en la plataforma .NET generan código intermedio haciendo uso de las reglas que proporciona el CTS (Common Type System) a través del CLS (Common Language System) para la generación de código interoperable. Una vez elegido el lenguaje se contemplaron varias aproximaciones en relación al nivel en que se debía ubicar el middleware PRISMA.

• **Modificación del CLR:** Introducir mecanismos que extendiesen la funcionalidad del CLR. Al modificar en este nivel, las herramientas de depuración existentes no pueden ser utilizadas ya que el CLR ha visto modificado su comportamiento.

• **Alteración del código intermedio generado:** Manipular de alguna forma el lenguaje intermedio generado. El principal inconveniente, al igual que en el caso anterior, es que tampoco es posible depurar el código fuente.

• **Extensión del CTS/CLS (Extender los lenguajes de programación):** El problema que presenta es que los compiladores desarrollados deben actualizarse al mismo ritmo que la evolución de los lenguajes .NET que extienden, y además, se pierde la independencia del lenguaje

Las tres aproximaciones descritas tienen el inconveniente adicional de crear una dependencia con la plataforma .NET, con lo que futuras versiones supondrían una actualización de dichas aproximaciones para mantener la compatibilidad.

En cambio, PRISMA actúa a un nivel superior, ya que utiliza un lenguaje de descripción de arquitecturas independiente de cualquier plataforma de desarrollo, lo que le permite incorporar funcionalidad adicional y un mayor nivel de abstracción. Por lo tanto, la implementación de PRISMA se ha realizado basándose en los lenguajes estándar definidos, sin añadir ninguna extensión a la plataforma de desarrollo, con el objetivo de no tener que adaptar la implementación a nuevas versiones del CLR, MSIL, etc.

En la Figura 9, se muestra gráficamente la ubicación de PRISMA en la arquitectura del framework .NET. Se puede afirmar por tanto que PRISMANET no solo extiende la tecnología .NET incorporándole los conceptos del modelo PRISMA, dando soporte a la reconfiguración dinámica de arquitecturas locales y distribuidas, sino que además esta funcionalidad se ha conseguido haciendo uso de los mecanismos que proporciona .NET.

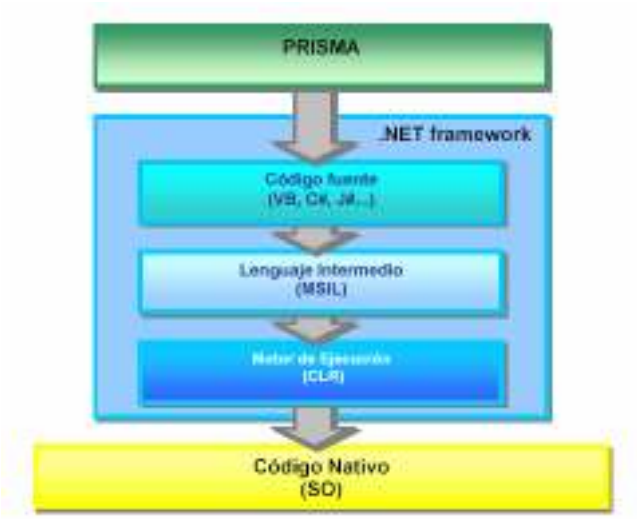


Figura 9. PRISMA sobre .NET

Para poder desarrollar un modelo sobre PRISMA, en el proceso de compilación, se deben realizar una serie de transformaciones desde el modelo hacia el código ejecutable (en este caso C#), extendiendo los constructores proporcionados por PRISMANET. Por ejemplo, para representar un aspecto del modelo PRISMA el compilador de modelos extiende la clase AspectBase que el PRISMANET proporciona.

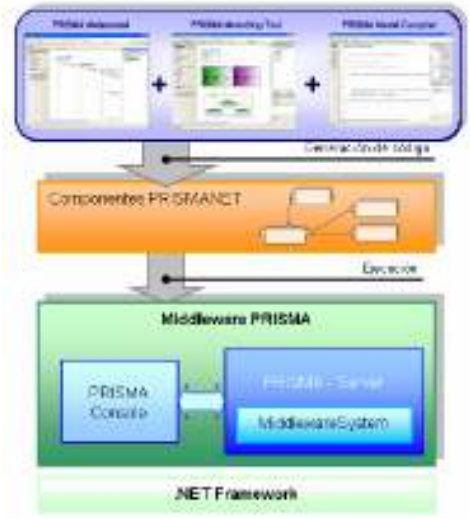


Figura 10. Arquitectura PRISMANET

El middleware es una capa abstracta que se ejecuta sobre la plataforma .NET, formada a su vez por dos módulos:

• **PRISMA Server**: Módulo que contiene el núcleo del Middleware, implementado en la clase MiddlewareSystem, y proporciona los servicios de gestión, distribución y evolución de los componentes PRISMA.

• **PRISMA Console**: Consola de administración del Middleware que ofrece al usuario una interfaz genérica para poner en ejecución configuraciones PRISMA, así como gestionar las instancias en ejecución. Además muestra información sobre el entorno de

ejecución, y los diferentes mensajes que son lanzados por los diferentes middlewares que lo conforman.

Los dos módulos son independientes entre sí, para que en caso de fallar uno de ellos, el sistema pueda recuperarse fácilmente sin más que volver a cargar el módulo. Para la ejecución de configuraciones PRISMA es necesario que PRISMA Server esté en ejecución, pues es quien proporciona los mecanismos propios de la arquitectura PRISMA necesarios para la ejecución. La ejecución de PRISMA Console es optativa, y sólo necesaria en caso que el usuario quiera lanzar a ejecución una configuración o ver información sobre el estado de una configuración en funcionamiento. Esto implica que en cada nodo donde se quiera ejecutar PRISMA debe estar instalado y cargado el PRISMA Server, ya que además de gestionar los componentes PRISMA también proporciona los servicios de distribución y mantenimiento a las instancias.

Los distintos middleware se encargan de gestionar los elementos que están ejecutándose en su nodo y de establecer las comunicaciones con sus middlewares vecinos. Los elementos compilados del modelo PRISMA se encuentran distribuidos por los distintos nodos, de forma que si uno de ellos falla, el sistema formado por el conjunto de middlewares puede redistribuirse las tareas para mantener estable al sistema.

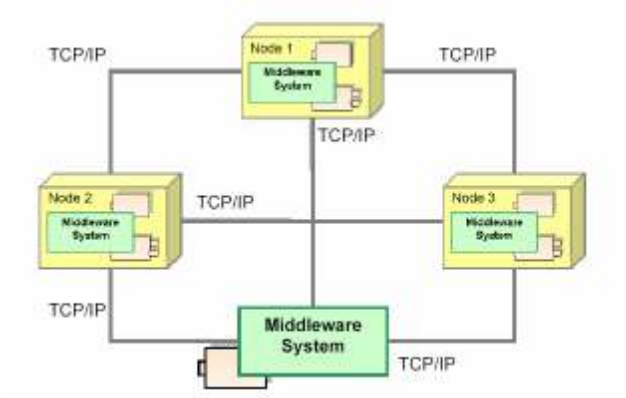


Figura 11. Middlewares ejecutándose de forma distribuida

El hecho que todos los middlewares que forman el entorno de ejecución estén comunicados entre si, permite que, pese a que las instancias de los elementos arquitectónicos se encuentren distribuidas entre diferentes nodos, todas las instancias sean accesibles desde cualquier middleware. En cierto modo se puede considerar que el conjunto de nodos con un middleware PRISMANET en ejecución son una parte de un middleware virtual en ejecución.

Como se ha visto, la funcionalidad del middleware PRISMANET está contenida en la clase *MiddlewareSystem*. Dicha clase es instanciada cuando se pone en ejecución el PRISMA Server en el nodo. La clase *MiddlewareSystem* está factorizada en un conjunto de capas que encapsulan las diferentes facetas de su funcionalidad.

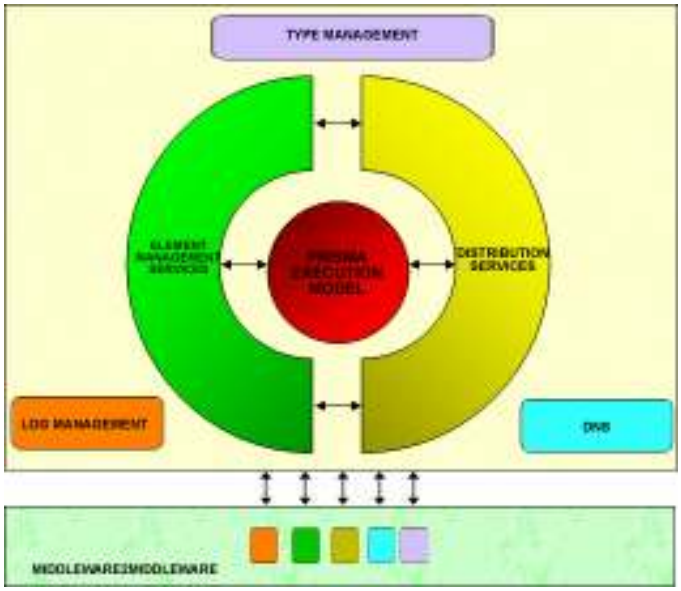


Figura 12. Capas de Middleware System

Las capas en las que se divide MiddlewareSystem se explican a continuación:

- Element Management Services: Esta es la capa encargada de la instanciación de los elementos arquitectónicos, de su configuración e interconexión. Guarda la información de los elementos arquitectónicos que se crean, así como de los *attachments* y *bindings*. En la Figura 13 se muestra un esquema con las diferentes clases que forman el Element Management Services.

- Distribution Services: Es la encargada de la comunicación distribuida.

- Type Management: Capa encargada de gestionar los tipos PRISMA. Se encarga de localizar los ensamblados en que están definidos los tipos PRISMA entre los middlewares y transferirlos a los middlewares que los puedan necesitar. Así mismo mantiene una lista con los tipos con los que está trabajando el middleware.

- DNS: Realiza la función de DNS y es útil para el resto de capas a la hora de localizar la ubicación en la que se están ejecutando los elementos arquitectónicos.

- Log Management: La finalidad de esta capa es proporcionar un información al usuario de lo que está ocurriendo en el middleware. Mediante ella el resto de capas muestran mensajes por consola.

- Middleware2Middleware: Capa encargada de establecer la comunicación entre los diferentes middlewares distribuidos que componen un entorno de ejecución PRISMA. La capa actúa como un proxy en el cual solo se muestran los servicios de las diferentes capas cuya invocación tenga sentido hacerla desde otro middleware.

Como se puede ver, cada una de las capas tiene una finalidad específica que la diferencia del resto. Con todo esto, dada una aplicación PRISMA, se pueden distinguir dos clases de comunicaciones:

• Las comunicaciones entre los distintos elementos arquitectónicos, según lo requieran las especificaciones del modelo (solicitud de servicios unos a otros). Este tipo de comunicaciones se realiza mediante *attachments* y *bindings*.

• Las realizadas entre los distintos Middlewares para averiguar las localizaciones de los distintos elementos. Estas comunicaciones se hacen mediante la capa Middleware2Middleware.

Debido a la naturaleza distribuida del modelo, la implementación de las comunicaciones se ha realizado con la tecnología *.Net Remoting* ya que proporciona implementados la mayor parte de los mecanismos para acceder remotamente a objetos así como para serializarlos y moverlos de una máquina a otra.

En la siguiente figura se pueden ver los distintos namespaces que conforman la implementación de PRISMA en .NET. Como se puede observar se han agrupado por cada uno de los conceptos del modelo las diversas clases que implementan la funcionalidad necesaria para dar soporte al modelo de ejecución que PRISMA propone y que .NET no proveía por si mismo.

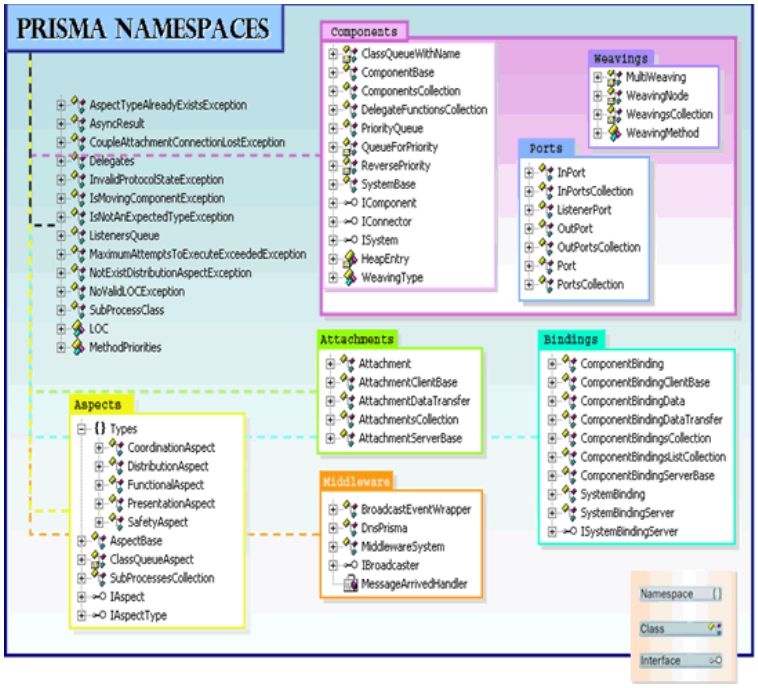


Figura 13. Prisma Namespaces

2.2.2.2 PRISMADSL

PRISMADSL es una herramienta que permite realizar modelado de arquitecturas PRISMA de una forma gráfica.

La implementación de la herramienta PRISMADSL ha sido realizada utilizando las posibilidades que ofrece DSL Tools de Microsoft para dar soporte a las *softwares factories*, siguiendo el enfoque de MDD. Al inicio de este capítulo se ha podido ver en que consiste la herramienta DSL Tools, que proporciona y cuales son sus posibilidades. En esta sección se puede ver como han sido utilizadas cada unas de estas características

para desarrollar una nueva aplicación que de soporte al modelado de arquitecturas PRISMA.

En primer lugar se introdujo el metamodelo PRISMA en DSL Tools. Para ello es necesario definir el metamodelo en el cual se basarán los modelos PRISMA. Para la creación de la herramienta, utilizando DSL Tools en Visual Studio 2005, se ha creado un nuevo proyecto seleccionando la opción *minimal language*, ya que se pretende introducir un nuevo metamodelo.

Una vez creado el proyecto, y utilizando la Tool Box, se pueden ir creando el metamodelo PRISMA. A continuación se puede observar una parte de la definición del metamodelo en DSL Tools.

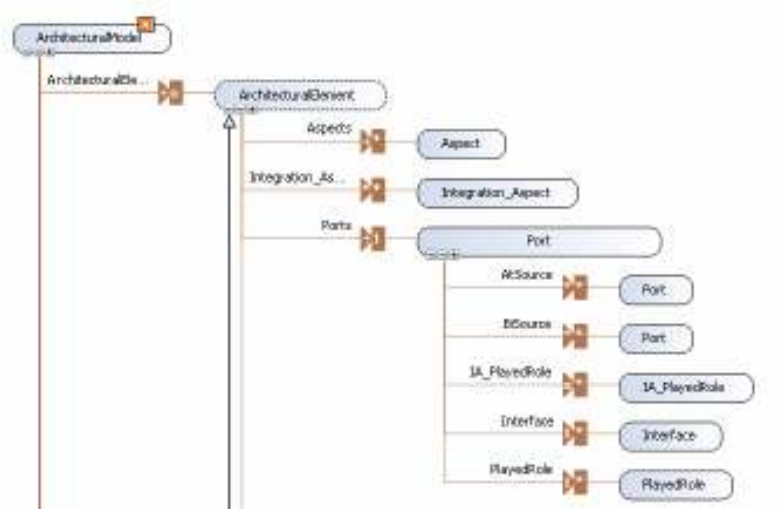


Figura 14. Metamodelo PRISMA en DSL Tools

La definición del metamodelo se realizó utilizando las primitivas de modelado de DSL Tools. Cada uno de los conceptos del metamodelo se definen utilizando clases, relaciones entre ellas, y atributos en ambas. Tambien se puede configuar la navegavilidad en las relaciones definidas, así como su cardinalidad. Toda esta definición es importante, dado que proporciona los mecanismos para poder realizar el posterior modelado de arquitecturas PRISMA, así como el recorrido de dicho modelo para realizar el proceso de compilación.

Después de definir el metamodelo, se le asocia una representación gráfica a los elementos arquitectónicos, aspectos, puertos, *attachments*, *bindings*, *weavings*, interfaces, definición del diagrama de estados para definir el protocolo y las relaciones entre todos estos elementos. Esta representación gráfica facilita al usuario la tarea de modelado, dado que siempre ayuda definir una especificación de forma gráfica en lugar de textual.

Junto a la definición del metamodelo y sus correspondientes representaciones gráficas, se definieron una serie de reglas de verificación. Estas reglas pueden verse desde dos puntos de vista distintos. Hay reglas que impiden realizar ciertas acciones, como puede

ser añadir un puerto a un aspecto, dado que es algo que no tiene sentido en el modelo PRISMA. Por otro lado se tienen una reglas algo menos prohibitivas, ya que pueden no cumplirse porque el modelo no esta acabado. Por lo tanto, se permite su violación durante el modelado, pero se han de satisfacer una vez acabado. Estas últimas reglas, permiten verificar cada uno de los elementos del modelo por separado o todos juntos, y comprueban desde que cada elemento definido tenga todo lo necesario según el metamodelo, como que se haya definido bien sintácticamente.

Una vez definido en DSL Tools el metamodelo PRISMA y su representación. La ejecución de un proyecto de este tipo permite modelar arquitecturas software PRISMA.

Una vez el usuario abre el papel tapiz, empieza a definir el modelo, utilizando los elementos definidos en la ToolBox DSL de conceptos PRISMA. A continuación se puede observar dicha ToolBox, (ver Figura 15).

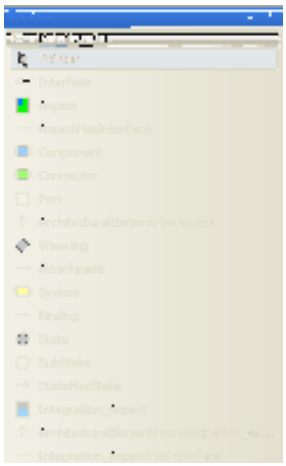


Figura 15. PRISMA DSL Tool Box

Otra parte importante en la herramienta es el *DSL Explorer* (ver Figura 16), que permite ver en forma de árbol el modelo que se va generando. A través del mismo se pueden ir añadiendo las opciones del modelado que no tienen una representación visual. Esto se realiza navegando por dicho árbol de elementos e interactuando con el sistema a través de la ventana de propiedades (ver Figura 17),que puede visualizarse de cada elemento seleccionado.

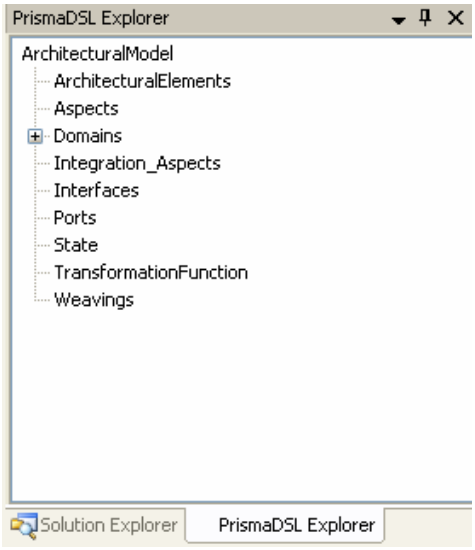


Figura 16. Prisma DSL Explorer

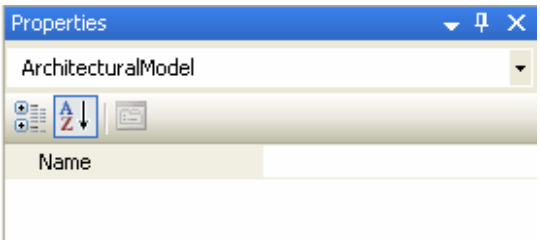


Figura 17. Ventana de propiedades

Como conjunción de las partes antes descritas, PRISMA DSL se conforma de la definición del metamodelo PRISMA y por otro lado la herramienda de modelado de arquitecturas PRISMA, (ver Figura 18).

Figura 18. Prisma DSL

2.2.2.3 PRISMAConfiguration

Una vez vistas las herramientas existentes y la carencia de la unión entre las mismas, que implica el principal objetivo de este proyecto, a continuación se muestra PRISMAConfiguration. Con lo visto hasta el momento en PRISMADSL se tiene definido el metamodelo PRISMA y el usuario puede definir sus propios modelos. Estos dos niveles corresponderían con los niveles M2 y M1 de la torre de reflexión de MOF.

A partir del modelo se generá código correspondiente al nivel de tipos, que es compilado para poder lanzarse a ejecución.

En PRISMA se ha dado un paso más, se proporcionará al usuario soporte para realizar la configuración de las instancias a lanzar a ejecución. Con este objetivo se ha realizado PRISMAConfiguration.

Por tanto, PRISMAConfiguration soporta un nivel más dentro de MOF, es decir, a través del modelo definido previamente por el usuario, se puede modelar la configuración de las instancias que serán lanzadas a ejecución. Esto correspondería con el paso del nivel M1 al nivel M0 de MOF.

PRISMA Configuration ha sido implementado en el mismo entorno que PRISMADSL, por lo que se obtienen las ventajas del uso de las DSL Tools. PRISMA Configuration es un nuevo proyecto que se ha definido con una estructura similar a PRISMADSL, es decir. Consta en primer lugar de un metamodelo, el cual tiene una representación gráfica para algunos de sus elementos, lanzando a ejecución el proyecto PRISMA Configuration, el usuario dispone de un papel tapiz sobre el cual podrá ir modelando la configuración inicial de las instancias correspondientes al modelo definido en el paso anterior. Una vez llegado a este paso hay que proveer al usuario un mecanismo para poder lanzar a ejecución el modelo de instancias. Estas son, a grandes rasgos, las carácterísticas de esta nueva herramienta. (ver Figura 19).

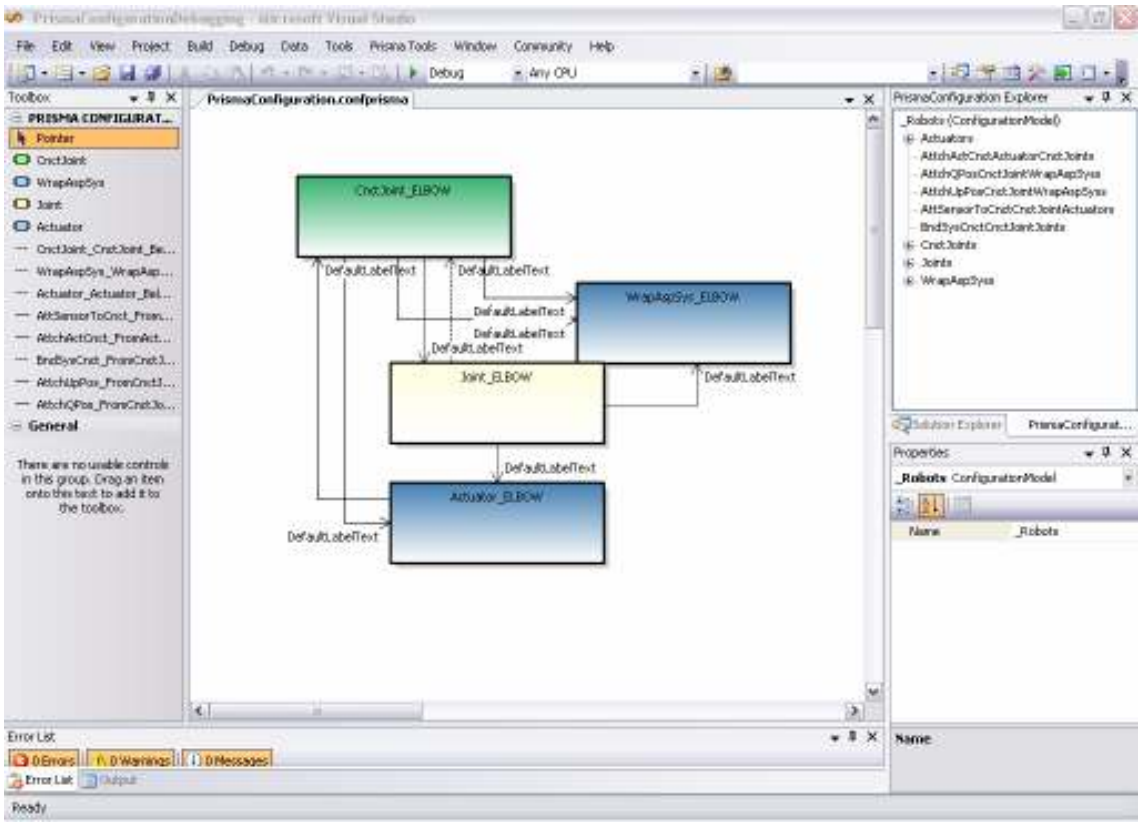


Figura 19. PrismaConfiguration

Una vez visto en que consiste PRISMA Configuration, se muestra como ha sido creada esta herramienta. Una primera gran diferencia respecto a PRISMADSL, radica en que el metamodelo a utilizar es dinámico, es decir, cambia para cada modelo definido por el usuario, dado que este metamodelo es obtenido a partir del mismo. Por esta razón se ha tenido que definir un mecanismo para obtener el metamodelo para PRISMA Configuration a partir del modelo definido en PRISMADSL. Para realizar esta tarea se han utilizado las plantillas de transformación disponibles en DSL Tools. Se han tenido que definir tantas plantillas como ficheros son necesarios para definir el metamodelo en PRISMA Configuration. Las dos plantillas más importantes son las que generan el fichero del metamodelo "DomainModel.dsldm", así como el fichero que contiene la configuración de la representación gráfica de los elementos del metamodelo, "Designer.dsldd". Básicamente la información que se indica en estos ficheros es la correspondiente a los los distintos elementos arquitéctónicos y sus relaciones de bindings y weavings, así como los atributos que son necesarios para la creación de dichos elementos arquitectónicos.

Otras plantillas que se han definido son para facilitar el modelado de las instancias al usuario, teniendo en cuenta algunas restricciones para impedir realizar acciones en el modelo que esten restringidas. Esta es una primera versión de PRISMA Configuration que tendrá que ser evolucionada para proporcionarle mayor expresividad para ayudar al máximo al usuario de la herramienta.

Hasta la realización de este proyecto se tiene disponible la herramienta PRISMA Configuration a nivel de modelado, pero aun no se puede lanzar a ejecución el modelo de instancias definido por el usuario. Esta es una de las tareas del presente proyecto. Como introducción a la tarea a realizar, se debe facilitar al usuario el lanzamiento a ejecución del modelo de instancias que se ha definido, para ello hay que crear una nueva plantilla de transformación que recorra el modelo de instancias definido y obtenga un fichero con dicha configuración, el cual tendrá que ser leido desde PRISMANET para poder ejercutarse.

Desde la propia herramienta PRISMAConfiguration se ha habilitado en el menu el lanzamiento de PRISMANET, que tras la realización de este proyecto permitirá al usuario ejecutar su configuración arquitectónica desde este punto. (ver Figura 20).

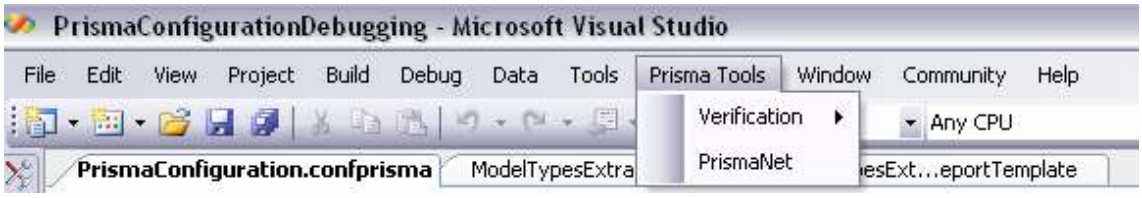


Figura 20. Ejecutar PrismaNet

Tras la realización de este proyecto y tras ejecutar prismanet podremos interactuar con el sistema software desarrollado como podremos ver en la Figura 21.

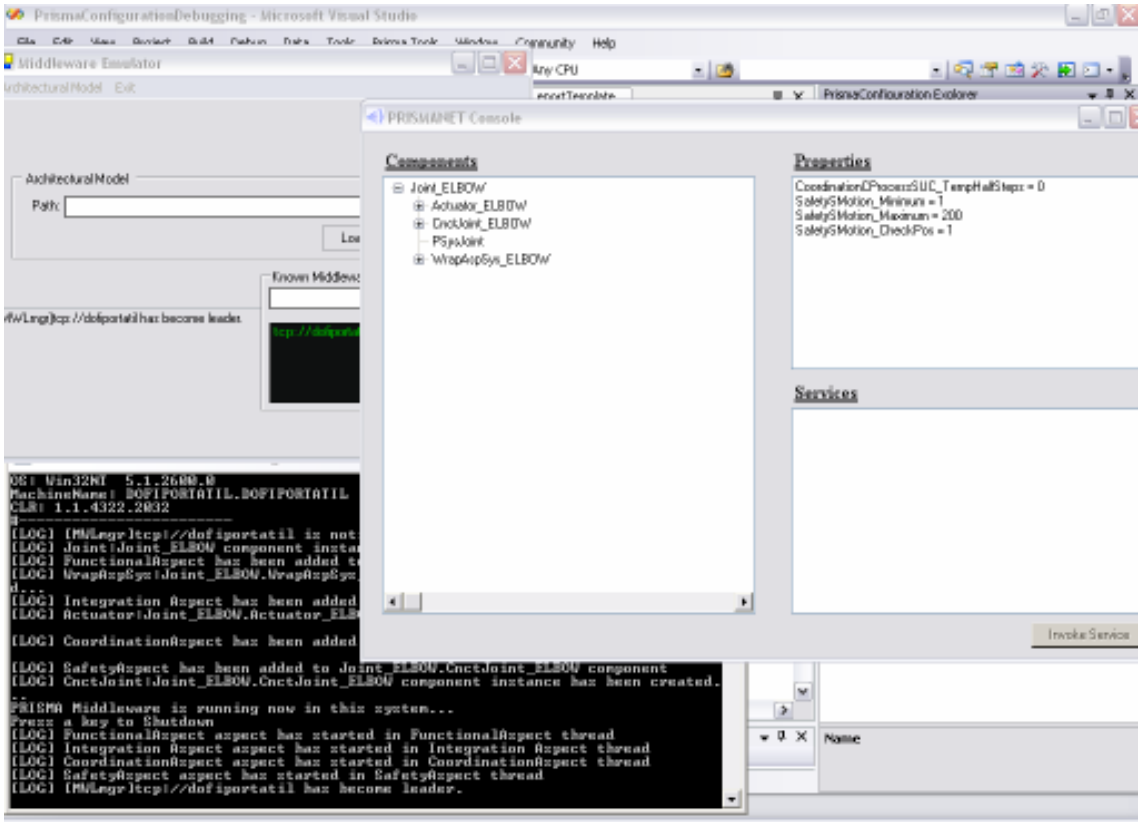


Figura 21. PrismaNet en ejecución.

Capítulo 3

Caso de estudio: Teach Mover

Contenido del capítulo

3 Caso de estudio: El robot *TeachMover*

El caso de estudio que se ha utilizado para obtener un *feedback* de la herramienta desarrollada ha sido el TeachMover. Se trata del desarrollo de una arquitectura software que de soporte a un sistema tele-operado como es el caso del robot *TeachMover*. Los robot tele-operados son sistemas de control que dependen del software para realizar actividades de inspección y mantenimiento en entornos muy peligrosas. Por lo tanto estos, son controlados por operarios desde zonas seguras para estos.

El *TeachMover* es una simplificación de los robots tele-operados utilizados en la industria náutica para el mantenimiento e inspección de los cascos de las embarcaciones sin generar polución medio-ambiental.

En este capitulo se muestra la morfología del caso de estudio y la arquitectura software que se ha detectado en base a dicho estudio.

3.1 Morfología

El robot está constituido por un conjunto de cuatro articulaciones: Base (Base), Hombro (Shoulder), Codo (Elbow), Muñeca (Wrist), y una herramienta que le permite realizar tareas (Tool). En base a estas articulaciones, los posibles movimientos del brazo de robot incluyen la rotación sobre sí mismo a través de la Base, la articulación tanto del Hombro como del Codo, la inclinación vertical de la muñeca (Pitch) y la rotación de la muñeca (Roll). En este caso, la herramienta es una pinza que permite tanto acciones de apertura como de cierre de sí misma y que posibilitan la recogida y deposición de objetos sobre una superficie (ver Figura 22).

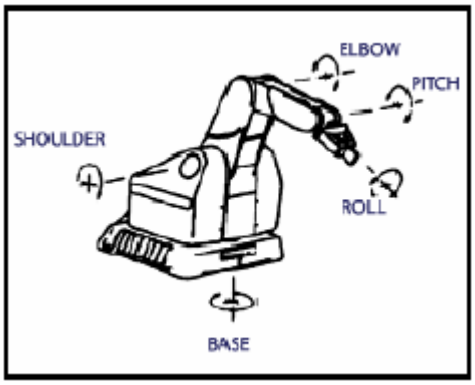


Figura 22. Brazo Robot *TeachMover*

El robot dispone de seis motores de pasos eléctricos que se encargan de impulsar la dirección de los movimientos de cada una de las articulaciones en función de los pasos que se le envíen. Los motores dirigen a sus respectivos miembros a través de engranajes y de un sistema de cable que los une para realizar los movimientos de forma exacta.

A causa de la disposición de los ejes de cada una de las articulaciones y la distancia que las separa entre sí, los movimientos para cada una de éstas están sometidos a unas

ciertas restricciones en cuanto a valores máximos y mínimos. Los rangos son los siguientes (ver Figura 23):

- Base: ± 90°
- Hombro: + 144°, - 35°
- Codo: + 0° , -149°
- Inclinación Muñeca: ± 90°
- Rotación Muñeca: ± 180°
- Apertura Pinza: 0 pulgadas, + 3 pulgadas (7,62 cm.)

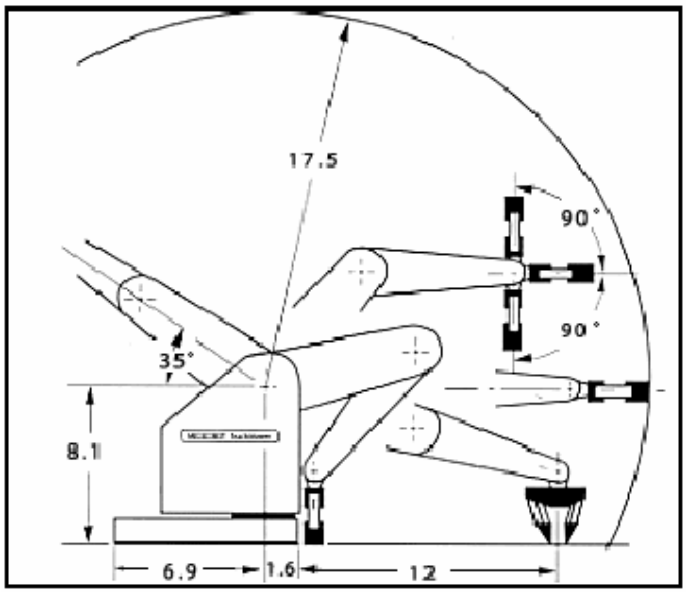


Figura 23. Movimientos del Brazo Robot

Como se mencionó anteriormente, el brazo de robot tiene la posibilidad de cargar objetos empleando la pinza. Para coger objetos sin romperlos dispone de un sensor que indica cuando lo sostiene. La capacidad de carga máxima que llega a soportar, con el brazo totalmente extendido, es de 16 onzas, es decir, aproximadamente unos 450 gramos. La pinza también permite ejercer una presión sobre los objetos que aguanta como máximo de 14 Newtons.

Por último, la velocidad de desplazamiento del robot está comprendida entre 0 y 7 pulgadas por segundo (178 mm/s) en función de la carga. De este modo, el MicroBot TeachMover puede simular los movimientos de la mayoría de las unidades industriales en situaciones de producción.

3.2 Modelo arquitectónico

La arquitectura del TeachMover tiene diferentes niveles de abstracción para sus componentes, conectores y la interacción entre ellos. El mínimo nivel de abstracción de la arquitectura del robot reside en sus sensores y actuadores, los cuales son componentes básicos, que se comunican con los elementos hardware del robot. La funcionalidad de los actuadores y sensores es la siguiente:

- Actutador: Un actuador envía comandos a una articulación o herramienta física del robot. Estos comandos son procesados y ejecutados por la articulación del robot o la herramienta seleccionada.

- Sensor: Un sensor se encarga de obtener una lectura de los comandos que emite el robot para conocer si un comando enviado a una articulación o herramienta ha sido realizado satisfactoriamente.

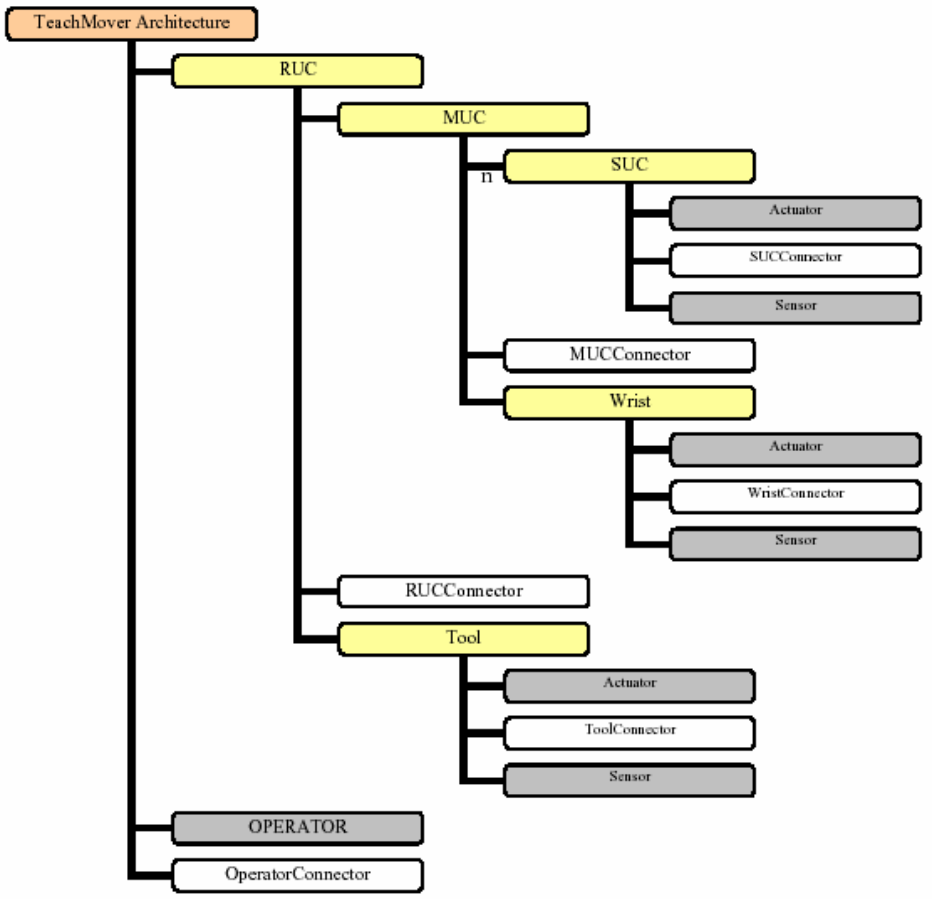


Figura 24. Elementos arquitectónicos de la Arquitectura Software del *TeachMover*.

Un actuador y un sensor son coordinados mediante un conector (SUCConnector). Estos tres elementos arquitectónicos (actuador, sensor y SUCConnector) están encapsulados dentro de un componente complejo llamado Simple Unit Controller (SUC). Sin embargo, dos SUC especiales han sido identificados para definir ciertas características de la articulación de la muñeca (WRIST) y las herramientas. Los SUC, el WristSUC y el ToolSUC deben componerse y coordinarse para poder formar una estructura completa y la funcionalidad necesaria para un robot tele-operado. Esta composición genera los diferentes niveles de granularidad en la arquitectura.

- ♦ Mechanism Unit Controllers (MUCs): Este elemento arquitectónico representa el brazo del robot (ARM), que esta compuesto por un SUC y el WRIST SUC coordinados mediante un conector.

- ♦ Robot Unit Controller (RUCs): Este elemento arquitectónico representa el robot, el cual esta compuesto de un MUC y de un Tool SUC coordinados mediante un conector.

- El modelo arquitectónico: Este nivel representa las interacciones entre el operario y el robot a través de un conector.

Este proyecto se ha centrado en el nivel de SUC para ilustrar de forma adecuada y no compleja todo el trabajo realizado.

3.3 Asuntos comunes (*Crosscutting-Concerns)*

Una vez ya diseñado el modelo arquitectónico del robot TeachMover, es necesario identificar los aspectos comunes al sistema (*crosscutting-concerns*) con el objetivo de separar cada uno de ellos y modelarlos posteriormente como aspectos.

Los *crosscutting-concerns* que han sido detectados son los siguientes:

- Aspecto funcional: Define la funcionalidad del robot TeachMover.
- Aspecto de coordinación: Define como son coordinados las peticiones de servicios entre los componentes del sistema.
- Aspecto de seguridad: Permite establecer servicios que establezcan políticas de seguridad en la ejecución de los movimientos del robot, de forma que los movimientos sean seguros para el robot, el operario y el entorno que les rodea.
- Aspecto de integración: Este aspecto permite la utilización en el modelo arquitectónico elementos software implementados sin seguir el modelo PRISMA, es decir, COTS.

Capítulo 4

INTRODUCCIÓN

Contenido del capítulo

4 PRISMANET (Refactoring)

Este proyecto parte de una implementación de PRISMANET que necesita una serie de mejoras. Dichas mejoras se han realizado en este proyecto. En este capítulo se enumeran y explican cada una de las mejoras que se han realizado, de forma que se ha obtenido una implementación más completa de PRISMANET, permitiendo así la ejecución de modelos PRISMA con mayor funcionalidad, flexibilidad y reutilización.

Estas mejoras tienen como fin dar soporte a necesidades que aun no habían sido contempladas previamente, así como una serie de modificaciones para permitir la generación de código de forma automática más eficientemente.

A continuación se muestran cada una de las mejoras que se han desarrollado:

- Gestión de una cola de prioridad para la ejecución de servicios en los aspectos. Permite introducir la definición de prioridades entre los distintos servicios que pueden ser ejecutados dentro de un aspecto, hasta el momento se disponía de una cola FIFO (First In First Out). En esta tarea se puede ver como se gestiona la cola de prioridad, así como las condiciones que establecen las prioridades entre los distintos servicios.

- Ejecución de servicios en modo cliente (OUT) o en modo servidor (IN). El poder definir los mismos servicios con un comportamiento distinto, según se comporten como cliente o como servidor. En la definición del protocolo del aspecto se define el modo en el cual se irán ejecutando los servicios dentro de un aspecto, es decir, en unos casos se puede ejecutar en modo servidor atendiendo a una petición y en otros casos este servicio será solicitado hacia el exterior del aspecto para que otro componente sirva dicha petición.

- Ampliación de la funcionalidad de las relaciones de Weavings entre los servicios de los aspectos.
    - Mejora en la identificación de los parámetros de los servicios que se relacionan mediante un Weaving.
    - Ampliación de los operaciones lógicos en los Weavings condicionales.
    - Modificación en el modo en el cual se identifican los parámetros entre la especificación de una función de transformación y su utilización en la definición de los weavings.
- Introducción del uso de funciones externas para su uso en la implementación de los servicios de los aspectos. Las funciones externas son funciones implementadas fuera de la arquitectura que se está desarrollando y que son utilizadas en la definición de los aspectos.

- Uso de estructuras abstractas de datos, para ampliar los tipos de datos a utilizar dentro de los atributos de los aspectos.

- Propagación del playedRole que se encuentra relacionado con el puerto de entrada por el cual se solicita a un componente un servicio, para que el aspecto conozca que playedRole ha solicitado dicha ejecución. Esta información es de

especial importancia en relación a las prioridades de los servicios, ya que dependiendo de cada playedRole un servicio puede tener prioridades diferentes.

4.1 Cola de prioridad en los Aspectos

En PRISMA se permite dotar de una prioridad a cada uno de los servicios de un aspecto. Esta prioridad indica si un servicio debe ejecutarse antes que otro, aun habiendo sido solicitado posteriormente. En PRISMA cuanto menor es el valor que se indique, mayor es la prioridad asociada al servicio indicado.

Hasta el momento los aspectos tenían una Cola que se regía por un proceso FIFO (First In First Out). Se iban atendiendo cada una de las peticiones recibidas en el orden en el cual llegaban al aspecto. Lo que se pretende es cambiar esta Cola por una cola de prioridades y dar soporte a las necesidades del modelo PRISMA.

Antes de explicar el funcionamiento de la gestión de la cola de prioridad, hay que hacer hincapié en como se definen dichas prioridades. En PRISMA se definen las prioridades en el protocolo de los aspectos. Los aspectos importan un conjunto de interfaces y por tanto dan semántica a la ejecución de los servicios que son publicados en dichas interfaces. En el protocolo, para cada uno de estos servicios se define en qué estados participan, con qué playedRole, con que prioridad y el modo en el cual se ejecuta (cliente o servidor). Por ello las prioridades dependen en primer lugar del estado en el cual se encuentre un aspecto, posteriormente del playedRole que solicita el servicio y por último del servicio. De este modo se puede tener para un servicio diferentes prioridades dependiendo del playedRole y del estado en el cual esta un aspecto en el momento de su solicitud de ejecución.

Una vez visto como se definen las prioridades en el modelo PRISMA, hay que buscar como almacenar esta información para su posterior uso en la gestión de la cola de prioridad. Hasta ahora ya se tenía una estructura en los aspectos que almacenaba los distintos servicios que cada uno de los diferentes playedRole podía ejecutar (*PlayedRoleCollection* y *PlayedRoleClass,* ver Figura 25). Dada la nueva información que se necesita almacenar en un aspecto, se ha reutilizado gran parte de esta estructura de clases para ampliarla con una nueva clase llamada ***ClassStatePriority***. Gracias a esta estructura podemos almacenar para cada uno de los diferentes playedRoles de un aspecto, todos los servicios que pueden ser invocados a través de ese playedRole, y asociados a dichos servicios se pueden definir diferentes prioridades para cada uno de los estados del aspecto (*methodpriorities)*, con lo que ya soportamos la definición de la información relevante a las prioridades de los aspectos (verFigura 25).

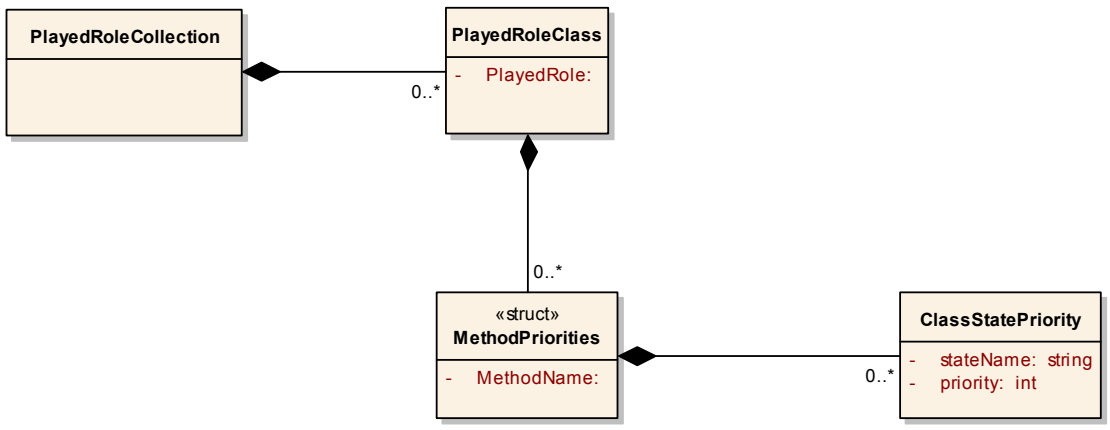
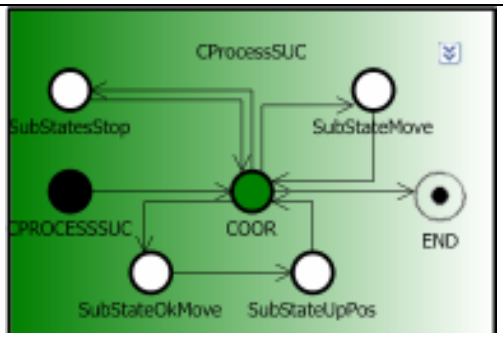


Figura 25. Diagrama de clases de soporte a la cola de prioridad de los aspectos

Después de haber mostrado la estructura donde se almacenará la información de las prioridades de los servicios, se muestra en que momento y como es almacenada esta información. Dado que al construir la instancia de un aspecto se definen los diferentes playedRoles, ese es el momento idóneo para almacenar toda la información referente a las prioridades de los servicios. Al reutilizar parte de la estructura que ya se disponía para los diferentes playedRoles de los aspectos, es más sencillo el código que se debe generar para almacenar la nueva información de las prioridades. Para ello se ha definido un nuevo servicio dentro de los aspectos ***AddPriorityService***, en el cual se indica el nombre del estado, el del playedRole, el servicio y la prioridad asociada. El código que se debe generar en el constructor del aspecto es la invocación de este método para cada una de las prioridades definidas en el modelo PRISMA realizado por el analísta. Este código se generá tras el código necesario para la definición de los playedRoles dentro del constructor del aspecto. A continuación se muestra un patrón que indica el paso de la definición de las prioridades mediante PRISMA DSL y su posterior transformación a código C#.

Patrón Prioridades del Aspecto usando el protocolo
Contexto
Traducción de la definición de los servicios mediante el protocolo. -Definición de PlayedRoles. -Definicion de prioridades. Paso de la especificación en PRISMA a la implementación en C#.
PRISMA DSL
Definición del protocolo, indicando para cada estado las transiciones entre ellos a través de la ejecución de servicios, sobre los que se indica el playedRole y la prioridad asociada.



Properties

StateHasState

AspectPlayedRole	ACT
AspectService	moveJoint
AspectTransaction	
Color	Black
Condition	
DashStyle	Solid
IA_PlayedRole	(none)
IntAsp_PlayedRole	(none)
Integration_AServi	(none)
Modifier	Out
PlayedRole	ACT
Priority	1
Service	moveJoint
Transaction	(none)
TransactionModifie	In

Properties

StateHasState

AspectPlayedRole	JOINT
AspectService	stop
AspectTransaction	
Color	Black
Condition	
DashStyle	Solid
IA_PlayedRole	(none)
IntAsp_PlayedRole	(none)
Integration_AServi	(none)
Modifier	In
PlayedRole	JOINT
Priority	0
Service	stop
Transaction	(none)
TransactionModifie	In

TRANSFORMACIÓN A C#

Paso 1: Definir los posibles estados que puede tener un aspecto, así como una variable que almacenará el estado actual de un aspecto. Para ello dentro de un aspecto se definirán los estados utilizando un tipo **enum.** Junto a la variable que almacena el estado del aspecto se ha definido una propiedad que se encarga de modificar una variable llamada "StateName", definida en la clase *AspectBase*. Esto es necesario, dado que toda la lógica de ejecución de los aspectos viene implementada en la clase *AspectBase*, y desde ella se debe tener acceso al estado actual del aspecto en todo momento. Los aspectos que el usuario define, extienden el funcionamiento de *AspectBase* utilizando el mecanismo de herencia. El problema es que hasta que el usuario no define un aspecto no se definen los estados que tiene. El mecanismo que se ha utilizado para poder informar a la superclase *AspectBase*, de la cual heredará el nuevo aspecto ha sido la definición de la propiedad *State*, la cual actualiza el valor de la variable "StateName" heredada de AspectBase, por lo que se dispone de este valor en la lógica de ejecución de la cola de prioridad.

En la lógica de ejecución de la cola de prioridad es necesario obtener el nombre del estado en cada momento que se atiende una solicitud de ejecución de un servicio, por este motivo se ha definido esta forma de comunicación utilizando esta variable, dado que hasta la definición de un aspecto concreto no se conocen los posibles estados del protocolo del mismo.

A continuación se muestra el código resultante de la definción del protocolo de un aspecto, se puede observar en primer lugar una variable de tipo enumerado, que

contiene los valores de los estados definidos. Justo a continuación se define la propiedad State, cuya justificación de uso ya se ha explicado anteriormente.

```
enum protocolStates
{
  CPROCESSSUC, COOR, END, SubStatesStop, SubStateMove, SubStateOkMove,
  SubStateUpPos
}
protocolStates state;
private protocolStates State{
      get { return state;}
      set { state=value;
            this.StateName=state.ToString();
          }
}
```

Paso 2: En el constructor del aspecto, definir los distintos playedroles, indicando para cada uno de los servicios que puede ejecutar, si se van a ejecutarse en modo servidor (true) o en modo cliente (false). Finalmente se añaden dichos playedroles a una colección que los almacena.

En este ejemplo se puede observar como se han definido los distintos playedRoles, indicando para cada uno de ellos los servicios que pueden ejecutarse con ellos y su modo de ejecución. Finalmente se van añadiendo estos playedRoles a una colección que tiene el aspecto para almacenar dichas instancias de playedRoles.

```
public CProcessSUC() : base("CProcessSUC")
{
// Creación de los playedRoles
   indicando si son IN (true) o OUT (false)

PlayedRoleClass ACT = new PlayedRoleClass("ACT");
                  ACT.AddMethod("moveJoint", false);
                  ACT.AddMethod("stop", true);
                  this.playedRoleList.Add(ACT);


                  PlayedRoleClass SEN = new PlayedRoleClass("SEN");
                  SEN.AddMethod("moveOk", true);
                  this.playedRoleList.Add(SEN);


                  PlayedRoleClass JOINT = new PlayedRoleClass("JOINT");
                  JOINT.AddMethod("moveJoint", true);
                  JOINT.AddMethod("stop", true);
                  JOINT.AddMethod("moveOk", false);
                  this.playedRoleList.Add(JOINT);


                  PlayedRoleClass UPDATEPOS = new
PlayedRoleClass("UPDATEPOS");
                  UPDATEPOS.AddMethod("newPosition", false);
                  this.playedRoleList.Add(UPDATEPOS);
...
}
```

Paso 3: En este momento ya se tienen definidos los playedroles y los servicios que pueden ejecutarse en cada uno de los mismos. Entonces se añaden las prioridades para todos los servicios, indicando: Estado, PlayedRole, Servicio y Prioridad. El uso del método AddPriorityService solo es necesario para aquellos servicios que puedan ser solicitados al Aspecto.

En el siguiente ejemplo, se puede observar como se han añadido para cada uno de los servicios del aspecto sus correspondientes prioridades. Podemos ver como se indica el estado, el playedRole, el servicio y finalmente la prioridad.

```
AddPriorityService(protocolStates.SubStatesStop.ToString(),
ACT.PlayedRoleName, "stop", 0);

AddPriorityService(protocolStates.SubStateMove.ToString(),
ACT.PlayedRoleName, "moveJoint", 1);

AddPriorityService(protocolStates.COOR.ToString(), SEN.PlayedRoleName,
"moveOk", 1);

AddPriorityService(protocolStates.COOR.ToString(),
JOINT.PlayedRoleName, "stop", 0);

AddPriorityService(protocolStates.COOR.ToString(),
JOINT.PlayedRoleName, "moveJoint", 1);

AddPriorityService(protocolStates.SubStateUpPos.ToString(),
JOINT.PlayedRoleName, "moveOk", 1);

AddPriorityService(protocolStates.SubStateOkMove.ToString(),
UPDATEPOS.PlayedRoleName, "newPosition", 1);
```

Paso 4: Por último se indica el estado en el cual inicia el aspecto su ejecución.

```
      State = protocolStates.COOR;
```

Una vez vistos los pasos necesarios para obtener y almacenar la información sobre las prioridades para los diversos servicios dependiendo de cada PlayedRole y Estado posibles, se muestra como se ha llevado a cabo la implementación de la Cola de prioridad y los cambios necesarios dentro del modelo de ejecución de los aspectos.

En el diseño de la cola de prioridad, había que tener en cuenta que las prioridades de los servicios dependen en primer lugar del Estado en el cual se encuentra el Aspecto en cada momento. Esta restricción hace surgir la problemática de tener que dar soporte a tener las distintas peticiones encoladas con la información relativa a su prioridad adecuada al estado actual del Aspecto. Para realizar esto de una forma más eficiente la implementación se ha realizado basándose en la siguiente estructura de clases. (ver Figura 26).

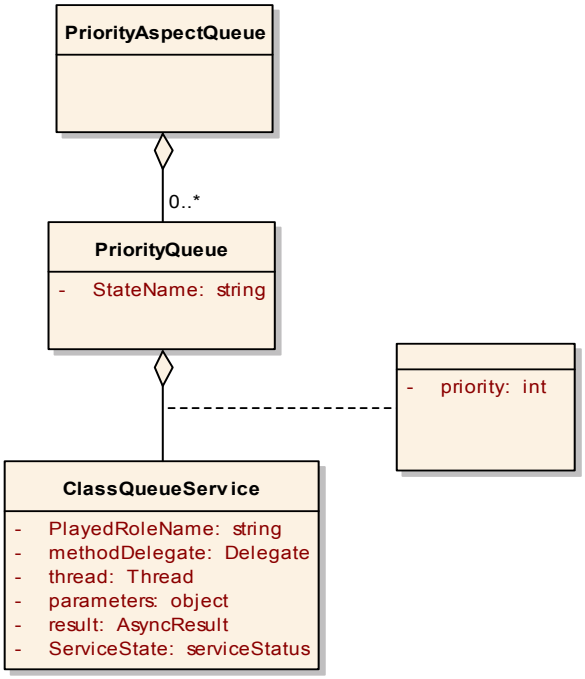


Figura 26. Diagrama de clases de la cola de prioridad del aspecto.

El aspecto dispone de una referencia a un gestor de colas de prioridad (PriorityAspectQueue). Este gestor se compone a su vez de una cola de prioridad por cada estado posible en el aspecto. Esta cola de prioridad viene soportada por la clase PriorityQueue.

Esta cola de prioridad esta implementada como una tabla Hash. Cada entrada de la tabla es el valor de las posibles prioridades, y hace referencia a una Cola que contiene todas las peticiones de una misma prioridad. Esto hace muy eficiente la inserción de nuevas peticiones en el gestor de prioridades y a la hora de recuperar la petición a ejecutar, simplemente va obteniendo por orden las solicitudes con mas prioridad hasta llegar a las de menos prioridad.

Después de ver la estructura del gestor de prioridades y las colas de prioridad asociadas, se muestra el modelo de ejecución de los aspectos con las modificaciones necesarias para dar soporte a la gestión de prioridades.

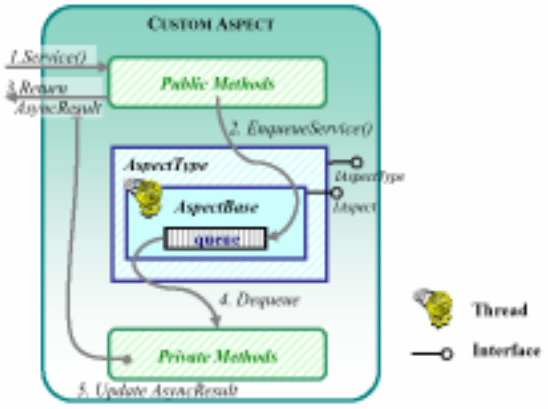


Figura 27. Esquema de ejecución de una petición de servicio en un aspecto.

Paso 1.- Petición de ejecución de un servicio. Cuando llega una petición de servicio a un aspecto, se crea un nodo para insertar en la cola de prioridad. Este nodo será un elemento de la clase *ClassQueueService*, que tiene toda la información del servicio a ejecutar. Cuando se crea un nodo, se indica que esta pendiente de ser servido en una de sus propiedades internas, que será de utilidad para cuando se desencolen las peticiones de las colas de prioridad.

Paso 2.- Encolar la petición de servicio en el gestor de prioridades.

Se obtienen todos los estados para los cuales se ha definido una prioridad para el servicio a encolar. En cada una de las colas de prioridad asociadas a dichos estados, se encola el nodo antes definido indicando la prioridad que le corresponde.

Puede darse el caso de que un servicio no tenga definidas prioridades para todos los estados, por lo que hay que encolar el nodo en las colas de prioridad de estos estados con la prioridad por defecto, el valor del mismo es un 10.

Paso 3.- Se devuelve una objeto *AsyncResult* para no detener la ejecución del componente que ha solicitado la ejecución del servicio. El tipo AsyncResult se ha definido en para posibilitar la ejecución de servicios de un modo asincrono y poder obtener información del estado de dicha ejecución cuando obtiene resultados, como pueden ser captura de excepciones o de obtención de parametros de salida. La clase AsyncResult ya existe en .NET pero ha sido extendida para obtener mayor funcionalidad.

Paso 4.- Desencolar una petición para ejecutarla.

El aspecto en su modo de ejecución, se encuentra a la espera de tener que procesar solicitudes de servicios. Una vez se dispone de alguna petición en el gestor de prioridades, se van procesando en el orden que establezcan las distintas prioridades y orden de solicitud de los servicios.

Para ello se utilizan los métodos Peek y Dequeue. En ambos hay que indicar el estado concreto en el que se encuentra el aspecto, para desencolar de la cola de prioridad apropiada.

Cuando se desencola un nodo y es procesado correctamente, se saca el nodo de la cola de prioridad y se indica en su propiedad de estado que ha sido procesado. De este modo si este nodo estuviera en otra cola de prioridad de otro estado del aspecto, cuando haya un cambio de estado y se desencolen peticiones de otra cola, al obtener el nodo de la cola, se comprueba si este ha sido servido, si es así se elimina y se obtiene otro elemento.

Esta implementación obtiene de una forma rápida las peticiones de servicios, manteniendo el orden de encolado en base a las distintas prioridades, y cuando hay un cambio de estado en el aspecto, no supone un coste elevado el obtener nuevas peticiones de la cola de prioridad.

4.2 Ejecución de servicios en modo cliente (OUT) o en modo Servicor (In)

El poder definir los servicios con un comportamiento distinto según se comporten como cliente o como servidor permite definir el comportamiento de los servicios en el protocolo del aspecto. Este comportamiento cuando es en modo cliente, se realiza una petición a otro componente para que gestione dicha solicitud de ejecución, en el modo servidor es el propio aspecto el que atiende la petición.

En la definición de un servicio de un aspecto, se puede definir su comportamiento de tres formas distintas. Servicios solo de cliente, solo de servidor o de cliente-servidor.

En primer lugar hay que diferenciar los servicios que tiene sólo un comportamiento cliente del resto, para no permitir que en los puertos de entrada del componente publiquen la interfaz de dicho servicio hacia el exterior. Para definir el modo de ejecución se ha ampliado la defínción de los playedRoles, para contemplar este nuevo parámtetro de configuración.

Este nuevo parámetro se ha definido como una variable booleana, cuyo significado es el siguiente:

True .- Servidor o Cliente-Servidor
False.- Cliente

A partir de esta información cuando se crean los puertos de entrada se puede obtener dinámicamente los servicios que son publicados en dichos puertos.

A continuación podemos ver un ejemplo de como se definiría un PlayedRole y los servicios asociados al mismo. Este código esta en la definición del constructor del aspecto asociado.

Se puede observar como en la definición del playedRole JOINT, al añadir cada uno de los servicios se indica el modo de ejecución (true o false).

```
PlayedRoleClass JOINT = new PlayedRoleClass("JOINT");
                JOINT.AddMethod("moveJoint", true);
                JOINT.AddMethod("stop", true);
                JOINT.AddMethod("moveOk", false);
                this.playedRoleList.Add(JOINT);
```

Una vez definidos los modos de ejecución de los servicios, hay que definir como se genera el cuerpo de dichos servicios. Es necesario tener dos partes bien diferenciadas dentro de la implementación de cada servicio. Por un lado la parte que se ejecutará cuando el servicio se procese en modo Servidor y por otro lado el modo Cliente.

Para llevar a cabo esta diferenciación, se ha definido en primer lugar una variable en *AspectBase* llamada *ServiceIn*. Se trata de una variable booleana.

True .- Modo Servidor
False.- Modo Cliente

Cuando llega una petición a través de los puertos de entrada, siempre se trata de peticiones a ejecutar en modo Servidor, por lo que cuando se desencola una petición de servicio, se pone a *true* el valor de la variable *ServiceIn*.

En la ejecución de un servicio en modo Servidor, puede que según la definición del protocolo, deba realizar una solicitud en modo Cliente a otro componente. Para ello, se utiliza el método *CallOutService*, implementado en *AspectBase*. Este método se encarga de cambiar el valor de *ServiceIn* a False, ejecuta el servicio asociado para que realice la ejecución en modo Cliente hacia otro componente utilizando los puertos de salida, y finalmente vuelve a poner el valor de la variable *ServiceIn* a True.

A continuación podemos ver la estructura que tendría un servicio, con ambas partes bien diferenciadas.

```
if(ServiceIn)
{
...

}
else {
Tratamiento antes de encolar en los outports.

CallOutService(interfaceName,subProcessName,methodName,args);

Tratamiento despues de procesar el servicio por otro componente.
}
```

4.3 Weavings

Los weavings permiten la ejecución de servicios de otros aspectos como consecuencia de la ejecución de un servicio en un aspecto. La ampliación de la funcionalidad de las relaciones de Weavings entre los servicios de los aspectos han sido las siguientes:

- Mejora en la identificación de parámetros de los servicios que se relacionan en un Weaving.
- Ampliación de los operaciones lógicos en los Weavings condicionales.
- Funciones externas. Uso de funciones que no han sido definidas en los aspectos. Las funciones externas son funciones implementadas fuera de la arquitectura que se está desarrollando y que son utilizadas en la definición de los aspectos.Por ejemplo, el uso de funciones matemáticas.
- Modificación en el modo en el cual se identifican los parámetros entre la especificación de una función de transformación y su utilización en la definición de los weavings.
- Análisis del soporte para el uso de estructuras abstractas de datos.

4.3.1 Mejora de la identificación de parámetros

Cuando un analista modela en PRISMA un weaving no tiene porque saber los nombres exactos que tienen los parámetros de los servicios que intenta asociar. Lo único que se pretende identificar, son los valores de un servicio que serán procesados anterior o posteriormente por otro.

Hasta el momento la identificación de los parámetros dentro de un weaving se realizaba en base a la definición de los servicios, lo cual conlleva una dependencia entre los servicios. Si se deseaba realizar un weaving entre dos servicios, y se quería pasar un parámetro de un servicio al otro en la ejecución del weaving, se debía definir dicho parámetro con el mismo nombre en los dos servicios. Para evitar esta restricción tan fuerte, se ha modificado la forma en la cual se realiza la identificación de parámetros en la ejecución de los weavings.

El analista define los parámetros de los servicios con los nombres que el desea, sin importarle los posteriores weavings que se puedan establecer entre ellos. Esto posibilita tener mayor reutilización de los mismos.

Posteriormente se definen los weavings y el usuario debe indicar el nombre de los parámetros de los servicios, con el único objetivo de nombrar igual a los parámetros que se desee compartir entre los servicios implicados en el weaving.

Por ejemplo, la definición de los servicios en los aspectos podría ser del siguiente modo:

```
SafetyAspect Smotion : check(Steps, Secure)

CoordinationAspect CProcessSUC : moveJoint(NewSteps, Speed)
```

Para realizar el weaving, se pueden nombrar los parámetros como uno desee. Los parámetros que se llamen igual en los dos servicios serán pasados de un servicio a otro.

```
AddWeaving(GetAspect(typeof(SafetyAspect)), "check","Pasos,Secure", weavingType,
GetAspect(typeof(CoordinationAspect)), "moveJoint", "Pasos,Speed");
```

A continuación se muestra un ejemplo de la transformación de un weaving en el ADL PRISMA hacia su correspondencia en código en C#.

Definición Prisma

```
Coordination Aspect Import CProcessSUC;
Weavings
      Safety Aspect
      CProcessSUC.movejoint(NewHalfsteps, Speed)
      after
      SMotion.check(NewHalfsteps);
```

Implementación C#

```
AddWeaving(Coordination, "movejoint",
           new string []{"NewHalfSteps","Speed"},
           WeavingType.AFTER,
           SafetyAspect, "check",
```

```
            new string []{"NewHalfSteps"} );
```

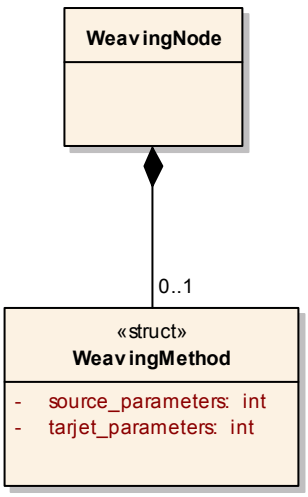
En un weaving se tienen dos servicios. Un servicio origen (source) y otro destino (target). Cada vez que se añada un weaving en el código a generar en C# se deberá indicar:

Aspecto origen y destino, servicio origen y destino, y los parámetros de los servicios origen y destino. Los parámetros contendrán solo el nombre de los mismos. A continuación se puede ver una plantilla del código que se generaría para dar soporte a estos conceptos.

```
AddWeaving(<target aspect>, <target service>,
           new string [] { <target parameters> },
           <WeavingType> ,
           <source aspect>, <source service>
           new string [] { <service parameters> },
           <function_name>);
```

Los nombres indicados en los parámetros del método destino del weaving deben ser un subconjunto de los parámetros del método que provoca el weaving, pudiendo haber algunos parámetros que no estén definidos porque se obtiene a través de funciones de transformación. En el caso de un servicio destino que no tenga parámetros, se pondrá null.

Una vez vistos los cambios a realizar en la definición de un weaving, se pueden observar los cambios necesarios para dar soporte a esta nueva forma de identificación. Se ha añadido una nueva estructura para almacenar la información referente tanto a los parámetros del servicio origen como destino del weaving, Por lo que en la clase WeavingNode, que es donde se almacena la información referente a la definición de un weaving, se le asocia esta estructura, para poder ser consultada posteriormente y comprobar la correspondencia de valores de los parámetros.



En la ejecución de los weavings, como paso previo se obtienen los parámetros necesarios para cada servicio. Esta funcionalidad se facilita con añadiendo la función

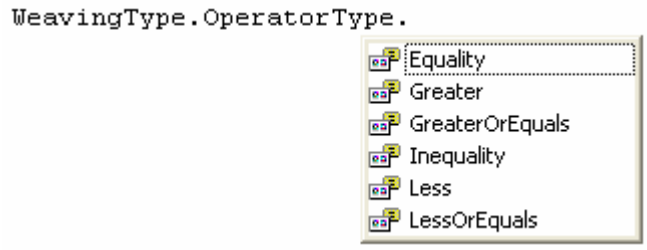
ParameterMatchingString en la clase *Multiweaving.cs*, para obtener antes de la ejecución de cada servicio dentro de un weaving, los argumentos necesarios. Esta función identifica la correspondencia entre los nombres de los parámetros origen y destino y con la posición que tienen, se realiza el paso de parámetros de un servicio a otro.

4.3.2 Ampliación de los operadores lógicos en Weavings condicionales

Los posibles weavings condicionales son AFTERIF, INSTEADIF, BEFOREIF. Hasta ahora solo se disponía de un operador lógico, la igualdad. Ahora se ha extendido la funcionalidad para que se puedan utilizar los siguientes operadores:

<,<=,<>,=,>,>=

Para identificar cada uno de estos operadores se ha creado el siguiente atributo dentro de la clase "WeavingType" para ser usado en la definición de los weavings y en su posterior ejecución.

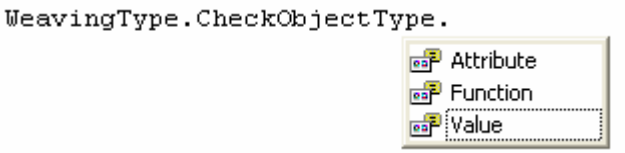


En todos los weavings condicionales se pretende realizar una comparación de la cual depende la ejecución de un servicio dependiendo de cada uno de los diversos tipos de weavings. Estas condiciones siempre tienen como fin comparar dos valores, un valor **A** contra otro valor **B**.

El valor **A** siempre es uno de los parámetros del servicio que desencadena el weaving condicional, y el valor **B** se puede obtener de las siguientes formas.

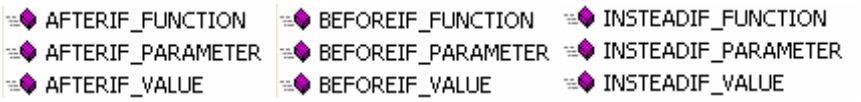
- A través de otro de los argumentos implicados en los servicios del weaving. (**Attribute**)
- Con el resultado de una función de transformación sobre uno de esos argumentos. (**Function**)
- Contra un valor estático definido en la especificación. (**Value**)

Para dar soporte a estas diferentes formas de obtener el dato sobre el cual realizar la condición del weaving se ha definido en la clase "WeavingType" la propiedad "CheckObjectType", en la cual se indica el origen del valor **B** de la condición del weaving.



Cuando se crea un weaving condicional en el que el valor contra el cual se realiza la comparación es obtenido en base a una función de transformación, hay que indicar en la construcción del weaving, tanto el nombre de la misma, como el de sus argumentos.

Para la creación de los posibles tipos de weaving se han definido las siguientes propiedades estáticas en la clase "WeavingType". Como se puede observar hay una propiedad por cada tipo de forma de obtener los valores de la condición, así como de que tipo de weaving se trata.



A continuación se muestran los diversos argumentos necesarios para poder invocar cada una de las propiedades:

Function

(<Parameter to compare> , < Operator type> ,
<Function name> , new string [] {<function parameters>})

Los nombres de los parámetros de la función deben coincidir con los nombres de los parámetros del servicio que desencadena el weaving, para poder obtener los valores correctos. Esta forma de identificación se basa en la forma en la cual se obtenía la correspondencia entre los parámetros de los weavings, por lo que reutilizando la parte ya implementada, se puede realizar el paso de parámetros entre el servicio que desencadena el weaving y la función de transformación.

Parameter

(<Parameter to compare> , < OperatorType> , <Parameter to apply comparation>)

En este caso se indica el parámetro del servicio origen, el operador de comparación y el parámetro del método destino sobre el cual se aplica la comparación.

Value

(<Parameter to compare> , < OperatorType> , <Value>)

Para el weaving condicional de tipo valor, simplemente se tiene que indicar el nombre el parámetro que deseamos comparar, el operador de comparación y el valor contra el cual se realizará esta comparación en todo momento de la ejecución del sistema.

Una vez vistos los diferentes tipos de weavings condicionales que se pueden definir, con todas sus posibles variantes, se muestran unos ejemplos de cada tipo para ver la implementación de los mismos.

Ejemplo weaving de tipo Attribute:

```
WeavingType weavingType = WeavingType.AFTERIF_PARAMETER
    ("money",
      WeavingType.OperatorType.LessOrEquals,
     "quantity");
```

En este ejemplo, se puede observar como se indica la comparación entre dos parámetros del servicio del weaving. En concreto se define la condición *money* <= *quantity*.

Ejemplo weaving de tipo Función:

Este ejemplo difiere del anterior en la utilización de una función, ***funcion1*****,** que transformará el parámetro *quantity*, para posteriormente compararlo en el weaving con el parámetro *money*.

```
WeavingType weavingType = WeavingType.AFTERIF_FUNCTION
   ("money",
     WeavingType.OperatorType.LessOrEquals,
    "Funcion1",  new string [] {"quantity"});
```

Ejemplo weaving de tipo Valor:

Este tipo de weaving compará el valor que tenga un parámetro de un servicio del weaving con un valor definido por el analísta.

```
WeavingType weavingType = WeavingType.AFTERIF_VALUE
   ("money",
    WeavingType.OperatorType.Equality,
    (decimal) 0);
```

4.3.3 Cambios en el uso de Funciones de transformación

El uso de las funciones de transformación se extiende mas allá del uso que se les da en los weavings condicionales. Esta necesidad viene dada por la incompatibilidad de tipos entre distintos parámetros del servicio que desencadena el weaving y el servicio que se ve afectado por esta ejecución.

La forma de identificación de parámetros para las funciones de transformación ha sido cambiada para corresponderse con la identificación de parámetros de los weavings. El modo de definir las funciones de transformación es el siguiente.

En primer lugar se crea una colección que almacena las distintas funciones de transformación,"DelegateFunctionsCollection" para el weaving correspondiente. Se indican los parámetros del servicio destino del weaving y los parámetros del weaving origen. Posteriormente, hay que asociar un delegado para cada parámetro y su función correspondiente de transformación. De esta forma el delegado posibilita la ejecución de la función de transformación. Esos delegados son punteros a servicios, que permiten la posterior ejecución de las funciones de transformación asociadas.

Una vez definidas todas las funciones de transformación para todos los parámetros necesarios, se crea el Weaving, indicando en este caso como último parámetro el nombre de la variable del tipo "DelegateFunctionsCollection".

Seguidamente podemos ver la estructura que sigue la implementación de un weaving en el cual hay definidas funciones de transformación para algunos de sus parámetros.

Paso 1: Crear una objeto de la clase DelegateFunctionsCollection para cada almacenar las funciones de transformación a utilizar en un weaving, indicando los parámetros del servicio origen y del servicio destino.

```
DelegateFunctionsCollection <functions_name> =
      new DelegateFunctionsCollection (
            new string [] { <target_parameters> },
            new string [] { <source_parameters> });
```

Paso 2: Para cada parámetro a transformar, hay que indicar su nombre, crear un delegado de la función que ejecutará la transformación y los nombres de los parámetros de dicha función de transformación. Los parámetros indicados para la función de transformación deben ser un subconjunto de los parámetros del método origen del weaving. Al añadir una función de transformación a un parámetro del weaving destino, se indica el parámetro que obtendrá dicho valor. Por lo que formara parte de los parámetros del método destino.

```
<functions_name>.AddFunctionDelegate( <param_name>,
      new <Transformation Function delegate > ,
      new string [] { <transformation function parameters> });
```

Paso 3: Indicar el nombre del objeto que almacena todas las funciones de transformación a aplicar en un weaving en la definición del mismo.

```
AddWeaving(<target aspect>, <target service>,
           new string [] { <target parameters> },
           <WeavingType> ,
           <source aspcet>, <source service>
           new string [] { <service parameters> },
           <function_name> );
```

4.4 Funciones de externas

En la especificación de un aspecto el desarrollador puede querer utilizar funciones que no van a ser implementadas en el mismo aspecto, sino que ya están implementadas.

Esto puede suceder al querer utilizar funciones matemáticas, funciones de transformación, etc.

La implementación de estas funciones deberán estar realizadas en un proyecto en C# llamado PRISMA_ExternFunctions. En el mismo estará la implementación de dichas funciones o bien la redirección hacia librerías ya definidas previamente.

En el proyecto debe adjuntarse la clase FunctionTypeAttribute.cs, necesaria para definir un atributo en las clases del proyecto indicando que contienen funciones. Esto es necesario dado que en la especificación de un aspecto en prisma solo se indica el nombre de una función, pero no se sabe en que clase esta su implementación. Indicando este atributo dentro de cada una de las clases que definan funciones y posteriormente utilizado los mecanismos de reflexión que posee .NET se puede encontrar la clase que realmente tiene la especificación de la misma.

De esta forma el desarrollador puede tener el proyecto, con todos los tipos de funciones a utilizar, de una forma más estructurada posibilitando un mayor mantenimiento del mismo.

A continuación se puede ver una plantilla de la definición de una función externa, donde se puede apreciar como se indica la propiedad *FunctionType*.

```
namespace PRISMA_TransformationFuntions
{
      [FunctionType("Function")]

      public class <Nombre de la clase>
      {
            // Definición de la función
            public static <tipo retorno> <Nombre>(<parametros>)
            {
                  ...

            }
      }
}
```

El proporcionar una propiedad a la clase que la diferencie de cualquier otra como una clase de funciones sirve para mejorar la eficiencia a la hora de que el Middleware se encargue de realizar la búsqueda en el ensamblado que contiene la implementación de las funciones de transformación.

La implementación de estas funciones podría estar en una clase llamada Functions dentro de ese proyecto. Para el uso de las mismas solo sería necesario invocarlas como métodos estáticos.

Ejemplo:

Functions.Servicio_1(arg_1,…arg_n)

4.5 Estructuras Abstractas de Datos

En todo desarrollo es necesario poder definir nuevas estructuras abstractas de datos. Estas son desarrolladas dependiendo de cada sistema que se vaya a desarrollar.

Esto es algo necesario en el desarrollo de cualquier sistema, y más en sistemas complejos como son los desarrollados mediante PRISMA, por lo que es necesario poder utilizar este tipo de representación de la información.

Las estructuras abstractas de datos que sean utilizadas desde un aspecto, se implementarán en un proyecto de biblioteca de clases aparte. El la implementación de los aspectos será necesaria la incorporación del "using" indicado a continuación para que se pueda acceder al uso de estas estructuras de datos.

using PRISMA.AbstractDataTypes

Una vez añadido el uso de la librería, se puede utilizar cualquier estructura abstracta de datos que allí haya sido definida.

Posteriormente en el proceso de compilación debería añadirse la librería que contiene la definición de las estructuras de datos utilizadas, para un correcto funcionamiento.

Capítulo 5

PRISMACase

Contenido del capítulo

5 PRISMACase

5.1 Compilador PRISMA

El compilador de modelos PRISMA permite generar automáticamente código C# a partir de los modelos arquitectónicos definidos utilizando PRISMA CASE. La realización del compilador de modelos se ha desarrollado utilizado la herramienta PRISMADSL.

El desarrollo del proceso de compilación puede subdividirse en varias tareas. La primera tarea consistió en la definición de los patrones de transformación que definen las correspondencias entre el lenguaje visual de modelado y C#, para generar las clases C# que dan soporte al nivel de tipos de PRISMA. La segunda tarea consistió en implementar dichos patrones en las plantillas de transformación que proporciona PRISMADSL al efecto. Las plantillas de transformación vienen dadas por el entorno de desarrollo Visual Studio, a través de los ficheros de tipo "report templates" que vienen con las DSL Tools. Dado que PRISMADSL ha sido desarrollado con esta tecnología, se dispone de este mecanismo para obtener información en base a los modelos que se definen por los usuarios.

Una vez desarrollada esta segunda etapa, es posible ejecutar las plantillas de transformación y así, obtener como resultado la generación de código C# de los tipos de la arquitectura PRISMA especificada mediante el correspondiente modelado. Partiendo de estos resultados la siguiente tarea, consistió en implementar las plantillas de transformación que posibilitan a partir del modelado de instancias de un modelo PRISMA instanciar el código C# generado en la tarea anterior.

En último lugar y como extensión al modelo PRISMA se ha ampliado los tipos basicos del metamodelo PRISMA, para dar soporte a la integración de COTS en un modelo PRISMA.

5.1.1 Patrones de transformación PRISMADSL → C#

En esta sección se muestra con detalle como se realizan todas las transformaciones desde el modelo PRISMA hacia código en C# que este soportado por el Middleware PRISMANET. Para ello, el conjunto de patrones de transformación que describen qué debe contener cada plantilla DSLPRISMA para generar dicho código C# se detallan en esta sección.

Los patrones de transformación han sido agrupados en base a diferentes elementos del modelo PRISMA. En concreto se tiene un patrón para las interfaces, otro para los aspectos y en último lugar el correspondiente a los elementos arquitectónicos (componentes, conectores y sistemas).

El desarrollo de estos patrones de transformación han sido desarrollados en la herramienta PRISMADSL, a través de las plantillas de tranformación de diversos ficheros “report template”. A continuación se muestra la estructura de estos ficheros.

Al inicio de cualquier fichero “report template” se define el tipo de extensión del fichero de salida, así como el fichero origen del modelo sobre el cual se va ha realizar la transformación.

```
<#@ output extension=".cs" #>
<#@ ArchitecturalModel processor="PrismaDSLDirectiveProcessor"
requires="fileName='..\PrismaDSL.prisma'"
```

A continuación se pueden hacer referencias a otras plantillas o a librerias de .NET.

```
<#@ include file = "Utils.ReportTemplate" #>
<#@ import namespace="System.Collections" #>
```

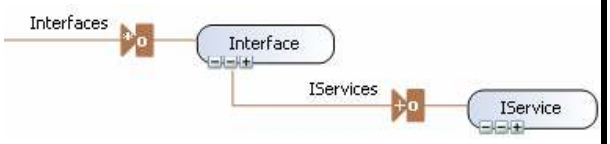
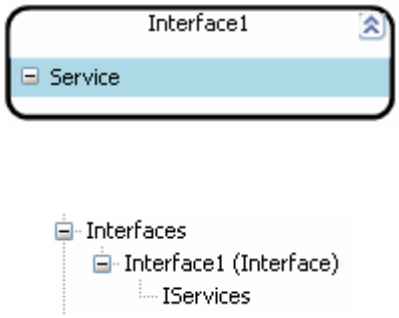
Una vez se han definido estos parámetros se comienza con la implementación de la plantilla de transformación. En este caso el lenguaje para recorrer el modelo es C#. Se pueden observar los simbolos: <#, #>, <#=. Estos simbolos son utilizados para diferenciar el código de la plantilla que recorre el modelo y el código o información que se generá en el fichero de salida. Cabe destacar que la forma en la cual se utilizan estas plantillas hace que el código no sea muy legible, más si cabe en el caso de este proyecto, dado que se utiliza codigo C# para recorrer el modelo y se genera código de este mismo lenguaje, es una limitación de la herramienta que hace un poco más costoso el desarrollo de las transformaciones.

Posteriormente estas plantillas son ejecutadas por el usuario, recorriendo el modelo definido y obteniendo el fichero con la información correspondiente a la implementación de cada una de las plantillas.

Cabe destacar que se puede generar el código que uno desee a través de la información del modelo definido.

5.1.1.1 Interfaces

Patrón 1.

Modelo PRISMA en DSL	Representación Gráfica
	

PATRÓN

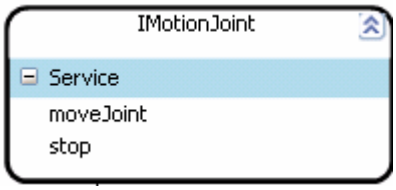
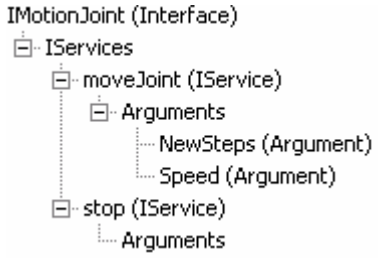
Descripción

En este patrón se describe la plantilla para generar código en C# a partir de las interfaces definidas en el modelo.

Plantilla

```
using System;
using PRISMA;
namespace <#=this.Model.Name#>
{
<#
  SortedList serviceList = new SortedList();
  foreach (Interface interfaz in this.Model.Interfaces)
  {
#>
  public interface <#=interfaz.Name#>
  {
<#
   foreach (Service servicio in interfaz.IServices)
   {
#>
   AsyncResult <#=servicio.Name#>(<#=CommaSeparatedArguments(servicio.Arguments)#>);

<#      if(!serviceList.Contains(servicio.Name))
            serviceList.Add(servicio.Name,servicio);

   }
#>
  }

<#
  }
  foreach(Service servicio in serviceList.Values)
  {
#>
  public delegate AsyncResult
  <#=servicio.Name#>Delegate(<#=CommaSeparatedArguments(servicio.Arguments)#>);

<#
  }
#>
}
```

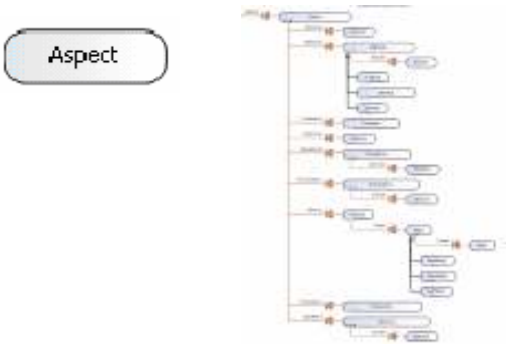
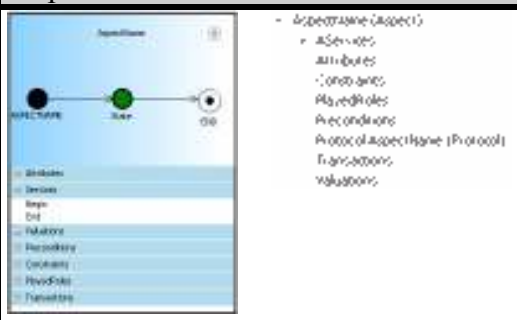
Caso de estudio

Descripción
La interfaz "*ImotionJoint"* describe los servicios necesarios para solicitar las ordenes de movimiento y parada sobre un componente del robot *TeachMover*..
Representación
 
Resultado patrón

`using System;`
`using PRISMA;`
`namespace RobotJoint`
`{`
`public interface IMotionJoint`
`{`
`AsyncResult moveJoint(int NewSteps, int Speed);`
`AsyncResult stop();`
`}`
`public delegate AsyncResult moveJointDelegate(int NewSteps, int Speed);`
`public delegate AsyncResult stopDelegate();`
`}`

Patrones relacionados
No existen patrones relacionados.

5.1.1.2 Aspectos

Patrón 2.	
Modelo PRISMA en DSL	Representación Gráfica
	
PATRÓN	
Descripción	
En este patrón se describe la plantilla para generar código en C# a partir de los aspectos	

definidos en el modelo. Dada la complejidad de los aspectos, en este patrón solo se muestra la generación de código para el aspecto como entidad pero sin describir su contenido. Para su mejor entendimiento, cada una de las partes que componen la especificación de un aspecto han sido definidas en patrones posteriores.

Plantilla

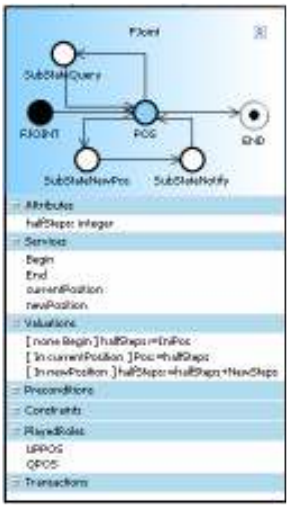
```
using System;
using PRISMA;
using PRISMA.Aspects.Types;
using PRISMA.Exceptions;
using System.Collections;

namespace <#=this.Model.Name#>
{
<#
        foreach (Aspect aspect in this.Model.Aspects)
        {
#>
        [Serializable]
        public class <#=aspect.Name#> :
         <#=aspect.Concern#>Aspect<#=CommaSeparatedNames3(aspect.Interfaces)#>
        {
<#
             /* Detalle generación código del aspecto
                        ...
             */
        }/* endforeach (Aspect aspect in this.Model.Aspects) */
#>
}
```

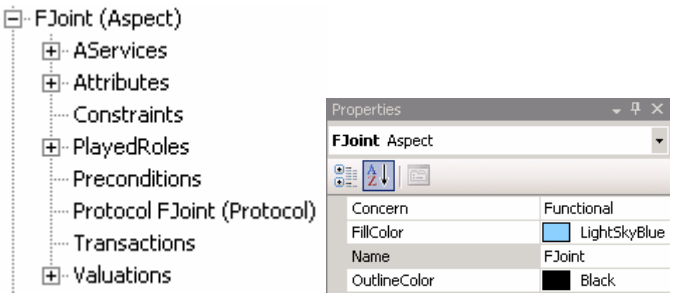
Caso de estudio

Descripción

El aspecto *FJoint* representa el aspecto funcional del modelo arquitectónico del TeachMover.

Representación





Resultado patrón

```
using System;
using PRISMA;
using PRISMA.Aspects.Types;
using PRISMA.Exceptions;
using System.Collections;
```

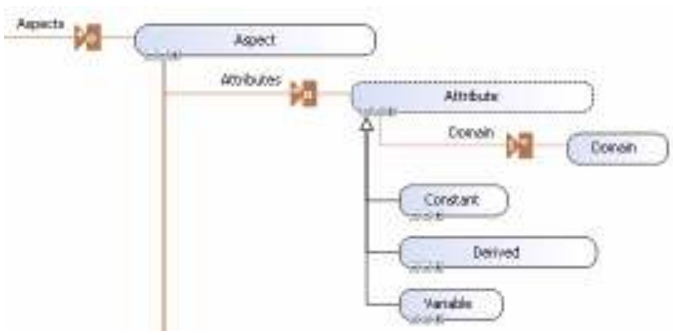
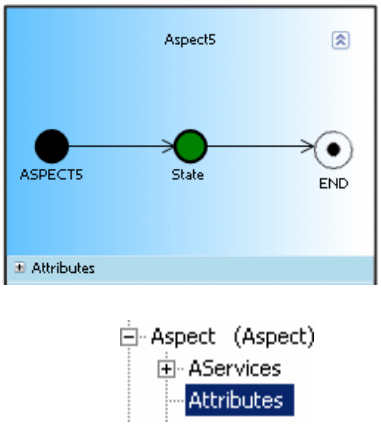
```
namespace RobotJoint
{
       [Serializable]
        public class FJoint : FunctionalAspect, IQueryPos, IUpdatePos
        {
             ...
        }
}
```

Patrones relacionados

Los siguientes patrones reflejan las transformaciones que generan el contenido del aspecto. Patrón 3,Patrón 4, Patrón 5, Patrón 6, Patrón 7, Patrón 8, Patrón 9, Patrón 10, Patrón 11, Patrón 12 y Patrón 13.

5.1.1.3 Atributos

Patrón 3.

Modelo PRISMA en DSL	Representación Gráfica
	

PATRÓN

Descripción

En este patrón se da soporte a la generación de código para los distintos atributos de un aspecto. En el modelo se pueden ver los distintos tipos de atributos que se pueden generar (Constantes o variables). En el caso de la generación de código no hay ninguna diferencia entre unos y otros. Se definen como variables que pueden ser accedidas desde la ejecución de los servicios de los aspectos. El modelo será quien restringe la modificación de las variables constantes, por lo que no es necesario realizar ningún tipo de control desde la generación de código.

Plantilla

```
...
<#
        SortedList attributes=new SortedList();
        foreach (DSIC.ISSI.PrismaDSL.DomainModel.Attribute atribute in aspect.Attributes)
        {

                attributes.Add(atribute.Name,null);
#>
        <#=DomainToType(atribute.Domain)#> <#=atribute.Name#>;
```

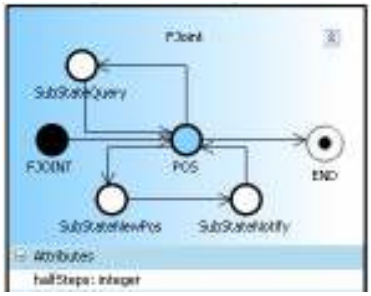
```
        public <#=DomainToType(atribute.Domain)#>
        <#=atribute.Name.Substring(0,1).ToUpper()#><#=atribute.Name.Substring(1)#>
        {
          get { return <#=atribute.Name#>; }
        }
<#
        }
#>
```

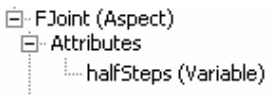
...

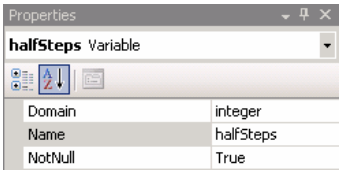
Caso de estudio
Descripción

En el aspecto "*Fjoint*" se ha definido un atributo "*halfSteps*" de tipo Integer para almacenar la información necesaria para enviar el movimiento al *TeachMover*.

Representación







Resultado patrón

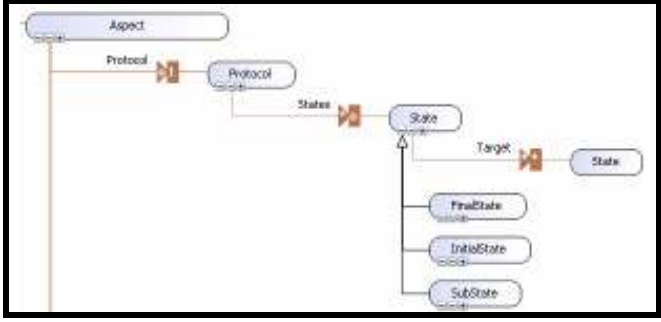
```
            ...
  int halfSteps;
  public int HalfSteps
  {
    get { return halfSteps; }
  }
            ...
```

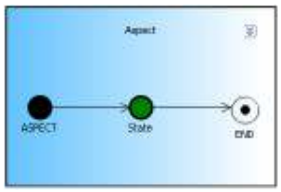
Patrones relacionados

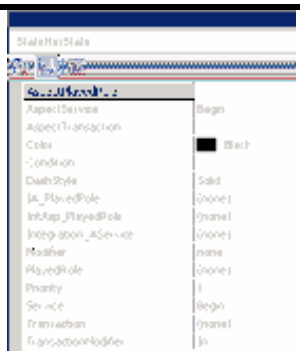
Este patrón se relaciona con el Patrón 2.

5.1.1.4 Protocolo

Patrón 4.	
Modelo PRISMA en DSL	Representación Gráfica







PATRÓN

Descripción

Este patrón presenta como se genera el protocolo de los aspectos. En este patrón sólo se detalla la generación de los estados dentro del aspecto, no la generación que determina cuando un servicio puede ejecutarse, así como operaciones de actualización del estado de dicho protocolo. Se ha de tener en cuenta que los subestados (estados de fondo blanco, ver representación) de un protocolo son los que definen este orden de ejecución y no representan cambio de estado. Por lo tanto, no son tratados en este patrón. En este patrón sólo se tratan los estados que representan un cambio de estado: estado inicial, estado final y estados (estados sin fondo blanco, ver representación).

Plantilla

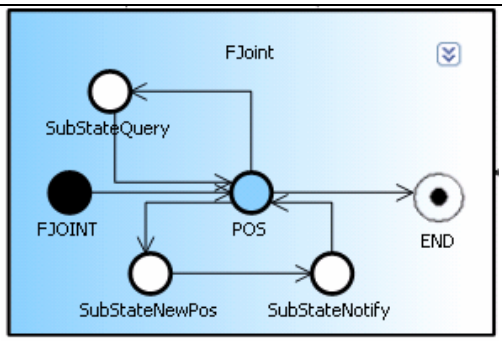
```
                                                    ...
#>
                enum protocolStates
                {
                <#=CommaSeparatedNames(aspect.Protocol.States)#>
                }
                protocolStates state;
                private protocolStates State{
                        get { return state;}
                        set { state=value;
                                this.StateName=state.ToString();
                              }
                }
<#
                                                    ...
```

Caso de estudio

Descripción

El protocolo del aspecto "*Fjoint*" define en que estados pueden ejecutarse cada uno de los distintos servicios. Esos estados son FJOINT, POS, y END.

Representación



Resultado patrón

```
                                                    ...

enum protocolStates
{
        FJOINT, POS, END, SubStateNewPos, SubStateNotify, SubStateQuery
}
protocolStates state;
private protocolStates State{
        get { return state;}
        set { state=value;
              this.StateName=state.ToString();
            }
}
                                                    ...
```

Patrones relacionados

Este patrón se relaciona con el Patrón 2.

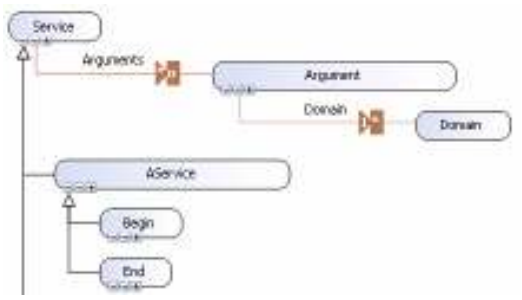
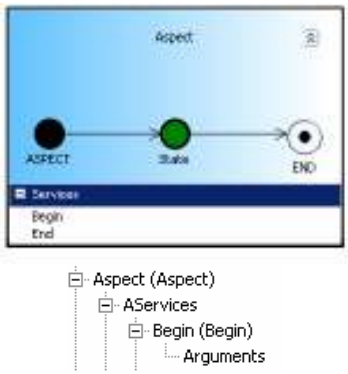
5.1.1.5 Servicios

Dentro de la definición de un aspecto existen tres tipos de servicios. En primer lugar el servicio *Begin*, que será el constructor de dicho aspecto. Luego se tienen servicios privados o públicos del aspecto y un tipo especial de servicios que son las transacciones, en las cuales se tiene que generar código específico para que se realicen todas las comprobaciones necesarias en dichas transacciones.

En esta sección se muestran los patrones de los tres tipos.

5.1.1.5.1 Servicio Begin

Patrón 5.

Modelo PRISMA en DSL	Representación Gráfica
	

PATRÓN

Descripción

Este patrón presenta la generación de código para el servicio *Begin* de un aspecto. Este servicio actúa como constructor del aspecto. En la generación del servicio *Begin* se tienen que realizar varios pasos. En primer lugar se indica el estado inicial del aspecto. Seguidamente se indican las valuaciones asociadas a dicho servicio si existieran. El siguiente dato a generar es la información relevante a los diferentes *PlayedRoles* del aspecto, una vez definidos se añade la prioridad para cada servicio definido en el protocolo. Por último se indica el estado en el cual se queda el estado del protocolo tras la ejecución del servicio *Begin*. Como la definición de las valuaciones es común a todos los tipos de servicio, la generación de código para ellas se ha definico en otro patrón que se presenta posteriormente.

Plantilla

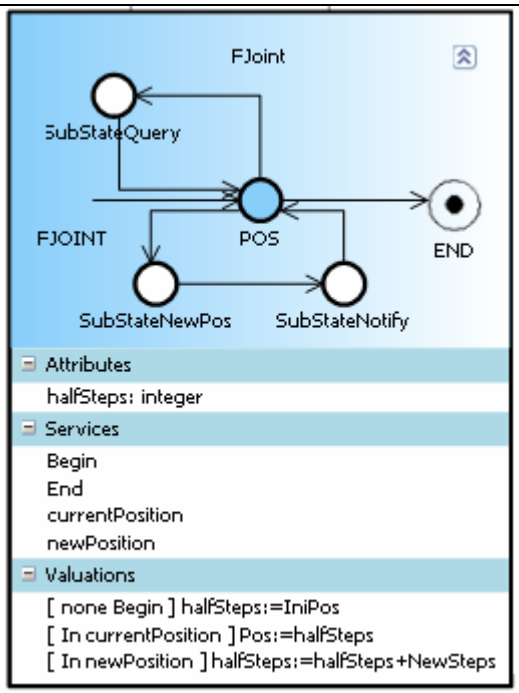
. . .

```
<#
foreach (AService service in aspect.AServices)
{
   if (service is Begin)
   {
#>
   public <#=aspect.Name#>(<#=CommaSeparatedArguments(service.Arguments)#>) :
    base("<#=aspect.Name#>")
{

   State = protocolStates.<#=service.StateHasState[0].Source.Name#>;

<#
   /* Valuaciones, se muestra en patrones posteriores. */

   /* PlayedRoles */

   foreach (PlayedRole playedRole in aspect.PlayedRoles)
   {
#>
   PlayedRoleClass <#=playedRole.Name#> = new PlayedRoleClass("<#=playedRole.Name#>");

<#
   bool ServiceIn;
   foreach (IService servic in playedRole.Interface.IServices)
```

```
   {
      ServiceIn=false;
      foreach (StateHasState stateHasState in playedRole.StateHasState)
      {
       if(stateHasState.Service.Name == servic.Name)
       {
         if(stateHasState.Modifier == TransitionModifier.In)
         {
           ServiceIn=true;
           break;
         }
        }/* End if(stateHasState.Modifier == TransitionModifier.In)*/
      } /* End foreach (StateHasState stateHasState in playedRole.StateHasState) */
   }/* End foreach (IService servic in playedRole.Interface.IServices)*/
#>

 <#=playedRole.Name#>.AddMethod("<#=servic.Name#>",<#=ServiceIn.ToString().ToLower()#>);

<#
   }
#>
   this.playedRoleList.Add(<#=playedRole.Name#>);

<#
   }/* End foreach (PlayedRole playedRole in aspect.PlayedRoles)*/
#>
   this.stateList=new ArrayList();

<#
foreach(Microsoft.VisualStudio.Modeling.NamedElement element in aspect.Protocol.States)
{
#>
   this.stateList.Add("<#=element.Name#>");

<#
}
foreach (PlayedRole playedRole in aspect.PlayedRoles)
{
       foreach(StateHasState stateHasState in playedRole.StateHasState)
       {
#>
          AddPriorityService(protocolStates.<#=stateHasState.Source.Name#>.ToString(),
          <#=playedRole.Name#>.PlayedRoleName,"<#=stateHasState.Service.Name#>",
          <#=stateHasState.Priority#>);

<#
       }
}
#>
       State = protocolStates.<#=service.StateHasState[0].Target.Name#>;
}
<#
}/* endif (service is Begin) */
```

...

Caso de estudio
Descripción
El servicio *"Begin"* del aspecto "*Fjoint"* se muestra a continuación.
Representación



Resultado patrón

...

```
public FJoint(int IniPos) : base("FJoint")
{

  State = protocolStates.FJOINT;
  halfSteps=IniPos;

  PlayedRoleClass UPPOS = new PlayedRoleClass("UPPOS");
  UPPOS.AddMethod("newPosition", true);
  this.playedRoleList.Add(UPPOS);

  PlayedRoleClass QPOS = new PlayedRoleClass("QPOS");
  QPOS.AddMethod("currentPosition", true);
  this.playedRoleList.Add(QPOS);

  this.stateList=new ArrayList();
  this.stateList.Add("FJOINT");
  this.stateList.Add("POS");
  this.stateList.Add("END");
  this.stateList.Add("SubStateNewPos");
  this.stateList.Add("SubStateNotify");
  this.stateList.Add("SubStateQuery");

  AddPriorityService(protocolStates.POS.ToString(), UPPOS.PlayedRoleName, "newPosition",
1);
  AddPriorityService(protocolStates.SubStateNewPos.ToString(),       QPOS.PlayedRoleName,
"currentPosition", 1);
  AddPriorityService(protocolStates.SubStateNotify.ToString(),       QPOS.PlayedRoleName,
"currentPosition", 1);
  AddPriorityService(protocolStates.POS.ToString(),                  QPOS.PlayedRoleName,
"currentPosition", 1);
  AddPriorityService(protocolStates.SubStateQuery.ToString(),        QPOS.PlayedRoleName,
"currentPosition", 1);
  State = protocolStates.POS;
}
```

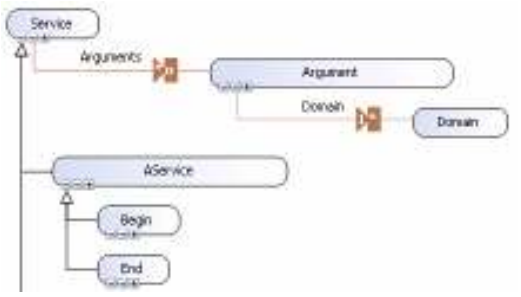
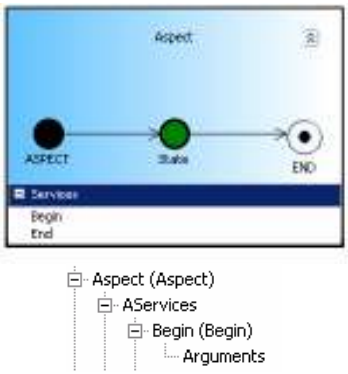
...

Patrones relacionados

Este patrón se relaciona con Patrón 2, Patrón 9, Patrón 10.

5.1.1.5.2 Servicios públicos

Patrón 6.

Modelo PRISMA en DSL	Representación Gráfica
	

PATRÓN

Descripción

Este patrón presenta la generación de código para los servicios de un aspecto que son publicados por alguna de las interfaces que el aspecto importa. En este patrón sólo se presenta la plantilla para la generación de la cabecera del método y la parte que distingue el modo de comportamiento IN del modo OUT de dichos servicios. El resto de componentes del servicio se ven con detalle en los patrones relacionados con este.

Plantilla

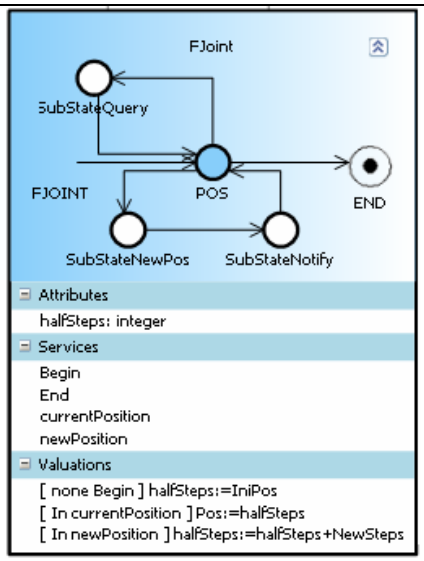
...

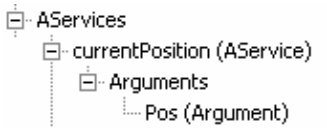
```
<#
   foreach (AService service in aspect.AServices)
   {
    if (!(service is Begin))
    {
       ...
    }/* endif (service is Begin) */
    else
    {
        if (!(service is End))
        {
#>

public AsyncResult <#=service.Name#>(<#=CommaSeparatedArguments(service.Arguments)#>)
{
<#
        SortedList parameters=new SortedList();
        foreach (Argument element in service.Arguments)
        {
                parameters.Add(element.Name,null);
        }


        if(service.Modifier != AServiceModifier.none)
        {
#>
        // Modo In
        if(ServiceIn)
        {
<#
        }

         /* Tratamiento de estado correcto para los servicios*/
```

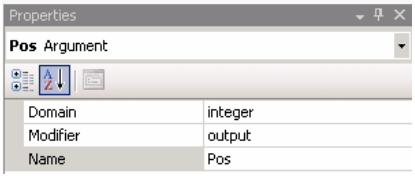
```
        /* Comprobación de las Precondiciones*/

        /* Valuaciones */

        /* Comprobación de las Restricciones de Integridad*/

        /* Tratamiento de servicios a ejecutar dentro de una secuencia del protocolo*/

        /* Actualizacion del estado del protocolo */

      if(service.Modifier != AServiceModifier.none)
      {
#>
      } //End modo IN
      // Modo Out
      else
      {
<#
      }


       if(service.Modifier == AServiceModifier.Out ||
          service.Modifier == AServiceModifier.InOut )
       {

        /* Valuaciones */

#>
         return
CallOutService(this.interfaceName_ServiceOut,this.playedRoleName_ServiceOut,
               "<#=service.Name#>",this.aspectStateCareTaker.ActiveTransaction,
                <#=CommaSeparatedNames(service.Arguments)#>);

<#
       }
       else if(service.Modifier == AServiceModifier.In)
       {
#>
       throw new Exception("This method doesn't have Service mode Out");

<#     }

       }/* end if (!(service is End))*/
   }/* endelse (service is Begin) */
}/*endforeach (AService service in aspect.AServices)*/
#>
```

...

Caso de estudio
Descripción
A continuación se puede ver el resultado de la generación del código expresado en la plantilla anterior para el servicio "*currentPosition*" del aspecto *"FJoint"*.
Representación







Resultado patrón

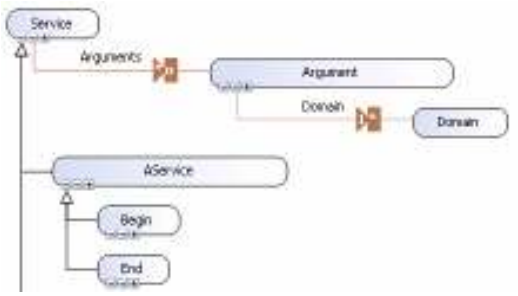
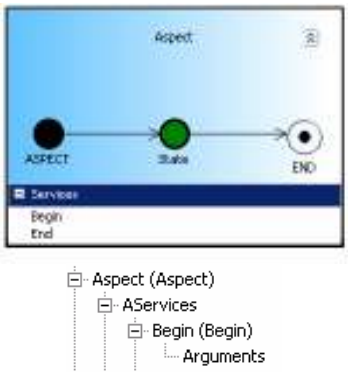
```
                                                    ...
public AsyncResult currentPosition(ref int Pos)
{
  // Modo In
  if(ServiceIn)
  {
        /* Tratamiento de estado correcto para los servicios*/

        /* Comprobación de las Precondiciones*/

        /* Valuaciones */

        /* Comprobación de las Restricciones de Integridad*/

        /* Tratamiento de servicios a ejecutar dentro de una secuencia del protocolo*/

        /* Actualizacion del estado del protocolo */

  }
  // Modo Out
  else
  {
 return CallOutService(this.interfaceName_ServiceOut,this.playedRoleName_ServiceOut,
                "currentPosition",this.aspectStateCareTaker.ActiveTransaction,Pos);
  }

}
                                                    ...
```

Patrones relacionados

Patrón 2, Patrón 8, Patrón 9, Patrón 10, Patrón 11, Patrón 12, Patrón 13

5.1.1.5.3 Servicios privados

Patrón 7.

Modelo PRISMA en DSL	Representación Gráfica
	

PATRÓN

Descripción

Este patrón presenta la generación de código para los servicios de un aspecto que son privados.

Plantilla

```
...
<#
   foreach (AService service in aspect.AServices)
   {
    if (!(service is Begin))
    {
       ...
    }/* endif (service is Begin) */
    else
    {
       if (!(service is End))
       {
            if(service.Modifier == AServiceModifier.none)
            {
#>
       public delegate AsyncResult
         <#=service.Name#>Delegate(<#=CommaSeparatedArguments(service.Arguments)#>);

<#
             }//End if(service.Modifier != AServiceModifier.none)
#>

public AsyncResult <#=service.Name#>(<#=CommaSeparatedArguments(service.Arguments)#>)
{
<#
       SortedList parameters=new SortedList();
       foreach (Argument element in service.Arguments)
       {
              parameters.Add(element.Name,null);
       }

       /* Tratamiento de estado correcto para los servicios*/

       /* Comprobación de las Precondiciones*/

       /* Valuaciones */

       /* Comprobación de las Restricciones de Integridad*/
```

```
        /* Tratamiento de servicios a ejecutar dentro de una secuencia del protocolo*/

        /* Actualizacion del estado del protocolo */

        if(service.Modifier == AServiceModifier.none)
        {
#>
          return null;
<#
        }
      }/* end if (!(service is End))*/
   }/* endelse (service is Begin) */
}/*endforeach (AService service in aspect.AServices)*/
#>
```

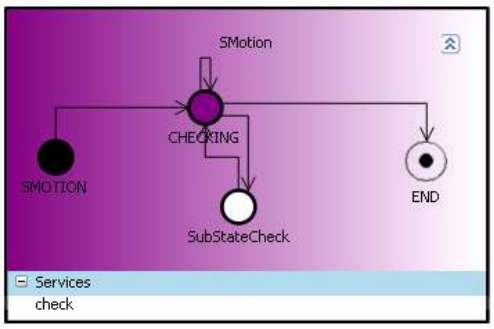
...

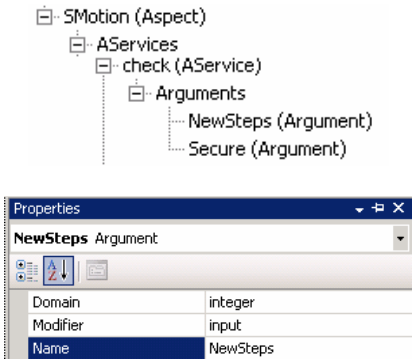
Caso de estudio

Descripción

A continuación se puede ver el resultado de la generación del código expresado en la plantilla anterior para el servicio privado "*Check*" del aspecto *"SMotion"*.

Representación





Resultado patrón

...

```
public delegate AsyncResult checkDelegate(int NewSteps, ref bool Secure);

public AsyncResult check(int NewSteps, ref bool Secure)
{
        /* Tratamiento de estado correcto para los servicios*/

        /* Comprobación de las Precondiciones*/

        /* Valuaciones */

        /* Comprobación de las Restricciones de Integridad*/

        /* Tratamiento de servicios a ejecutar dentro de una secuencia del protocolo*/

        /* Actualizacion del estado del protocolo */

       return null;
}
```

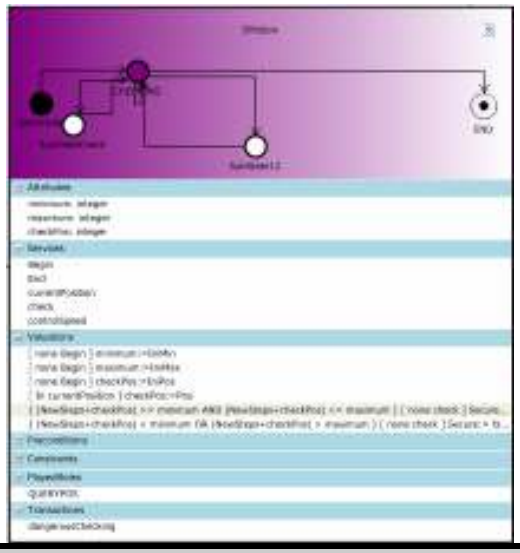
...

Patrones relacionados

Patrón 2, Patrón 8, Patrón 9, Patrón 10, Patrón 11, Patrón 12, Patrón 13

5.1.1.5.4 Transacciones

Patrón 8.

Modelo PRISMA en DSL	Representación Gráfica
	

PATRÓN

Descripción

Este patrón presenta la generación de código para los servicios de tipo transacción de un aspecto. Dado el metamodelo PRISMA las transacciones son un tipo especial de servicios. En el modelado de una transacción se añade como un nuevo servicio en el apartado Transacciones del aspecto, pero en el protocolo cuando se define las transiciones entre los diferentes estados, se indica el servicio que hace posible la transición, así como si esta incluido dentro de una transacción. Todo lo anterior hace que el proceso de obtener la secuencia de servicios que hay que ejecutar dentro de la transacción sea más complejo, por lo que esta plantilla es más extensa que cualquier otro tipo de servicio. Entre el código generado se pueden observar las distintas instrucciones que son necesarias utilizar para que el Middleware PRISMANET sea capaz de ejecutar correctamente dichas transacciones.

Plantilla

```
...
<# foreach(DSIC.ISSI.PrismaDSL.DomainModel.Transaction_ transaction
             in aspect.Transactions)
   {
#>
  public AsyncResult <#=transaction.Name#>
    (<#=CommaSeparatedArguments(transaction.Arguments)#>)
  {

   // Modo In
   if(ServiceIn)
   {
<#
        /* Tratamiento de estado correcto para los servicios*/

        /* Comprobación de las Precondiciones*/

        /* Valuaciones */

        /* Comprobación de las Restricciones de Integridad*/
```

```
/* invocacion secuencia transaccion **/
foreach (AService serviceTrans in aspect.AServices)
{
   foreach(StateHasState stateHasState in serviceTrans.StateHasState)
   {
      if(stateHasState.Transaction   !=   null   &&   stateHasState.Transaction.Name   ==
transaction.Name)
      {
         if (!(stateHasState.Source is SubState))
         {
            if (stateHasState.Target is SubState)
            {
#>
               if (state == protocolStates.<#=stateHasState.Source.Name#>)
               {
               aspectStateCareTaker.StartTransaction();
               try
               {
<#                      /* Tratamiento del primer servicio de la transacción */
                        if(stateHasState.Condition != String.Empty)
                        {
#>
                        if(<#=stateHasState.Condition#>)
                        {
<#
                        }
                        /* Invoca a un servicio OUT */
                        if(stateHasState.Modifier == TransitionModifier.Out)
                        {
#>

InvokeOutService("<#=stateHasState.PlayedRole.Interface.Name#>","<#=stateHasState.Played
Role.Name#>",

"<#=stateHasState.Service.Name#>",this.aspectStateCareTaker.ActiveTransaction,
                        <#=CommaSeparatedNames(stateHasState.Service.Arguments)#>);
<#
                        }

                        /* Invoca a un servicio privado del Aspecto */
                        if(stateHasState.PlayedRole==null)
                        {
#>

<#=stateHasState.Service.Name#>(<#=CommaSeparatedNames(stateHasState.Service.Arguments)#
>);
<#
                         }
                        /* Ejecuta en servicio IN como servicio privado dentro del aspecto
por estar en una transición */
                        if(stateHasState.Modifier == TransitionModifier.In)
                        {
#>

<#=stateHasState.Service.Name#>(<#=CommaSeparatedNames(stateHasState.Service.Arguments)#
>);

<#
                         }
#>
                        aspectStateCareTaker.CheckConsistence;
<#
                        if(stateHasState.Condition!="")
                        {
#>
                        }
<#
                        }
#>
                        /* Actualización del estado */
                        state = protocolStates.<#=stateHasState.Target.Name#>;
<#
        /* Tratamiento del resto de servicios de la transacción */
        SubState subState = stateHasState.Target as SubState;
        System.Collections.IList                                        substateLinks=
```

```
subState.GetElementLinks(subState.Source.TargetRole.Id);
        StateHasState subStateHasState=null;
        int i=0;
        while( i<substateLinks.Count)
        {
                if(substateLinks[i] is StateHasState)
                {
                   subStateHasState=(StateHasState)substateLinks[i];
                   if(subStateHasState.Condition != "")
                   {
#>
                        if(<#=subStateHasState.Condition#>)
                        {
<#
                   }
                   /* Invoca a un servicio OUT */
                   if(subStateHasState.Modifier == TransitionModifier.Out)
                   {
#>
InvokeOutService("<#=subStateHasState.PlayedRole.Interface.Name#>",

"<#=subStateHasState.PlayedRole.Name#>","<#=subStateHasState.Service.Name#>",
                  this.aspectStateCareTaker.ActiveTransaction,
                  <#=CommaSeparatedNames(subStateHasState.Service.Arguments)#>);
<#
                   }
                   /* Invoca a un servicio privado del Aspecto */
                   if(subStateHasState.PlayedRole==null)
                   {
#>

<#=subStateHasState.Service.Name#>(<#=CommaSeparatedNames(subStateHasState.Service.Argum
ents)#>);

<#
                   }
                  /* Ejecuta en servicio IN como servicio privado dentro del aspecto por
estar en una transición */

                  if(subStateHasState.Modifier == TransitionModifier.In)
                  {
#>

<#=subStateHasState.Service.Name#>(<#=CommaSeparatedNames(subStateHasState.Service.Argum
ents)#>);

<#
                   }
#>
                aspectStateCareTaker.CheckConsistence;
<#
                  if(subStateHasState.Condition!="")
                  {
#>              }

<#
                  }
#>
                state = protocolStates.<#=subStateHasState.Target.Name#>;

<#
                  i++;

                  if(subStateHasState.Target is SubState)
                  {
                        subState = subStateHasState.Target as SubState;
                        substateLinks=
subState.GetElementLinks(subState.Source.TargetRole.Id);
                        i=0;
                  }
               } /*End if(substateLinks[i] is StateHasState)*/
        }/*End While(i<substateLinks.Count) */
        }/* End stateHasState.Target is SubState */
#>
        }
        catch
```

```
        {
                aspectStateCareTaker.SetConsistecy(false);
        }
        finally
        {
                aspectStateCareTaker.EndTransaction();
        }
    }
}
<#
            }
        }
    }
}
availableStates.Clear();
}/*endforeach transacction*/
#>
return null;
}
```

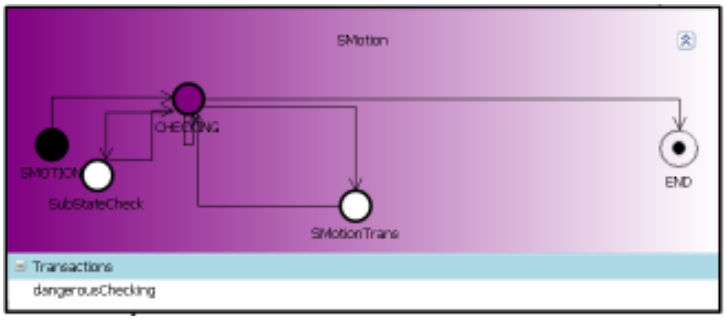
...

Caso de estudio

Descripción

Como ejemplo de la generación de código de una transacción se ha definido en el modelo una transacción para el control de la velocidad con la que se mueve el robot del caso de estudio.

Representación



AspectPlayedRole	QUERYPOS
AspectService	controlSpeed
AspectTransaction	dangerousChecking
Color	Black
Condition	
DashStyle	Solid
IA_PlayedRole	(none)
IntAsp_PlayedRole	(none)
Integration_AServic	(none)
Modifier	In
PlayedRole	QUERYPOS
Priority	1
Service	controlSpeed
Transaction	dangerousChecking
TransactionModifier	In

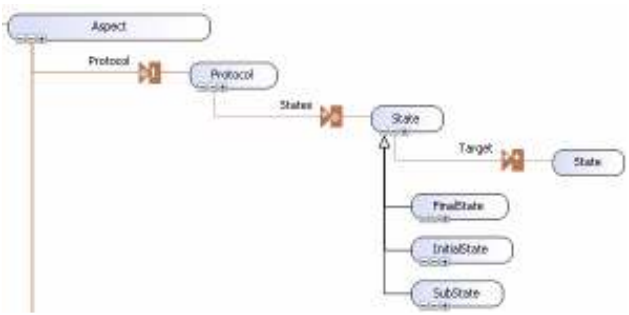
Resultado patrón

```
public AsyncResult dangerousChecking()
{

  // Modo In
  if(ServiceIn)
  {
        if( state != protocolStates.CHECKING
        ) throw new InvalidProtocolStateException("SMotion","dangerousChecking");

        if (state == protocolStates.CHECKING)
        {
                aspectStateCareTaker.StartTransaction();
                try
                {
                controlSpeed(Steps, CurrentSpeed, Secure);
                aspectStateCareTaker.CheckConsistence;
                state = protocolStates.SubState12;

        InvokeOutService("IQueryPos","QUERYPOS","controlSpeed",this.aspectStateCareTaker.
ActiveTransaction,Steps, CurrentSpeed, Secure);
                aspectStateCareTaker.CheckConsistence;
```

```
                state = protocolStates.CHECKING;
        }
        catch
        {
                aspectStateCareTaker.SetConsistecy(false);
        }
        finally
        {
                aspectStateCareTaker.EndTransaction();
        }
}
}
}
```

Patrones relacionados
Patrón 2, Patrón 9, Patrón 10, Patrón 11, Patrón 12, Patrón 13

5.1.1.6 Comprobación protocolo

Patrón 9.	
Modelo PRISMA en DSL	Representación Gráfica
	

PATRÓN
Descripción
Este patrón presenta como se define la comprobación del estado para saber si el estado es correcto y es posible ejecutar un servicio del protocolo.
Plantilla

. . .

```
<#
   bool isFirst = true;
   System.Collections.SortedList availableStates=new System.Collections.SortedList();
   /* Tratamiento de estado correcto para los servicios*/
   foreach(StateHasState stateHasState in service.StateHasState)
   {
        if (isFirst)
        {
```

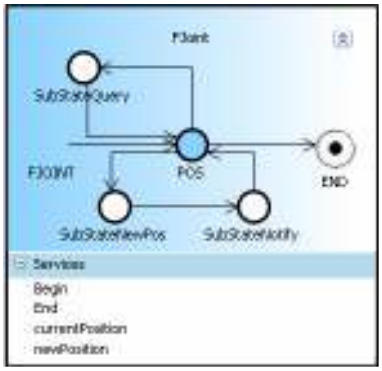
```
                isFirst = false;
                availableStates.Add(stateHasState.Source.Name,null);
#>
              if( state != protocolStates.<#=stateHasState.Source.Name#>

<#      }
        else if(!availableStates.Contains(stateHasState.Source.Name))
                {
#>
                   && state != protocolStates.<#=stateHasState.Source.Name#>

<#
                   availableStates.Add(stateHasState.Source.Name,null);
                }
        }
#>
        )                                          throw                                    new
InvalidProtocolStateException("<#=aspect.Name#>","<#=service.Name#>");

<#
                                          ...
```

Caso de estudio

Descripción

Como ejemplo del uso de la comprobación del estado en un servicio, se muestra la definición del servico "*currentPosition*" del aspecto "*FMotion*".

Representación



Resultado patrón

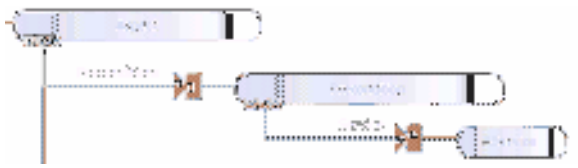
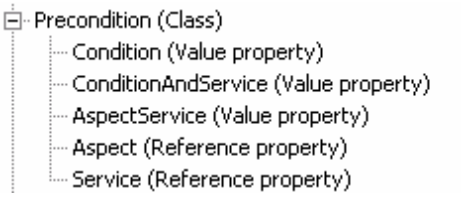
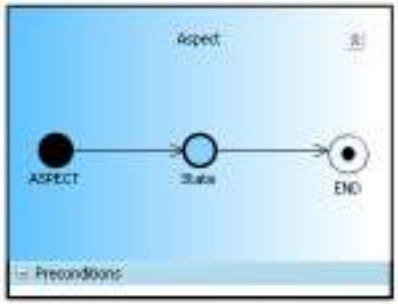
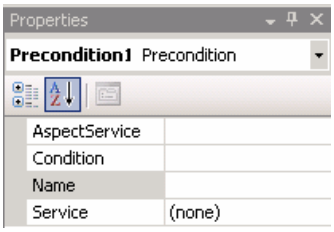
```
                                          ...

public AsyncResult currentPosition(ref int Pos)
{
                                          ...

 if( state != protocolStates.SubStateNewPos
    && state != protocolStates.SubStateNotify
    && state != protocolStates.POS
    && state != protocolStates.SubStateQuery
  ) throw new InvalidProtocolStateException("FJoint","currentPosition");

                                          ...
}
                                          ...
```

Patrones relacionados

Patrón 6, Patrón 7, Patrón 8 y Patrón 14

5.1.1.7 Precondiciones

Patrón 10.

Modelo PRISMA en DSL	Representación Gráfica
 	  

PATRÓN

Descripción

Este patrón muestra la plantilla para la generación de código asociada a las precondiciones.

Plantilla

. . .

```
/* Comprobación de las Precondiciones*/
foreach(Precondition precondition in  service.Precondition)
{
#>
   if (!(<#=precondition.Condition.Replace("=","==").Replace("<>","!=").
         Replace("<==","<=").Replace(">==",">=").Replace("and","&&").
         Replace("or","||").Replace("AND","&&").Replace("OR","||")#>))
   throw new InvalidPreconditionException("<#=aspect.Name#>","<#=service.Name#>");

<#
}
```

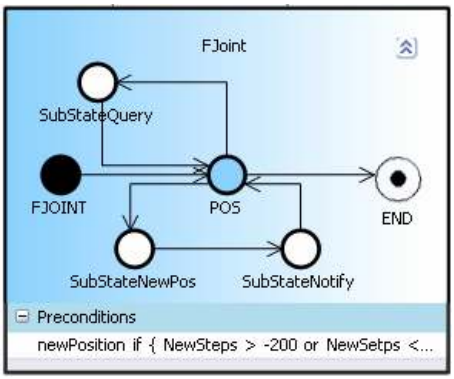
. . .

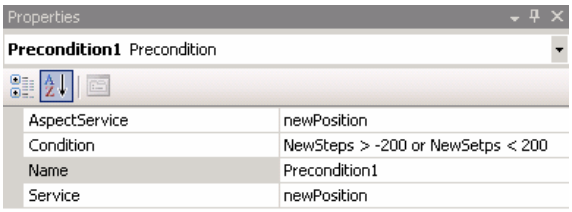
Caso de estudio

Descripción

Un ejemplo de aplicación de una precondición sería la definida para el servicio *"newPosition"* del aspecto "*FJoint*".

Representación





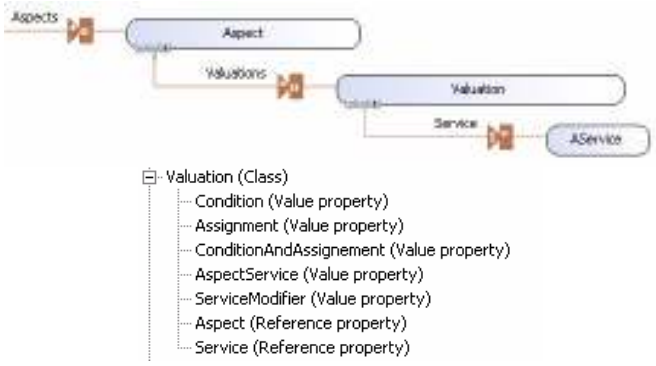
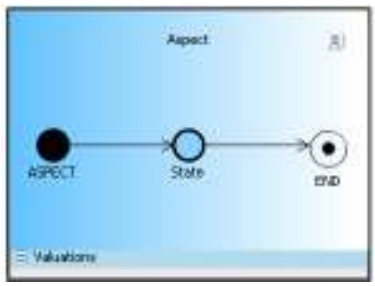
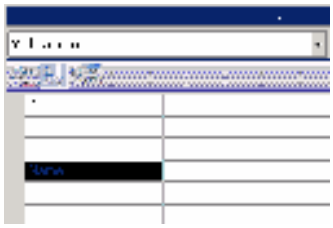
Resultado patrón

```
                                                ...
public AsyncResult newPosition(int NewSteps)
{
                                                ...

if (!(NewSteps > -200 || NewSetps < 200))
        throw new InvalidPreconditionException("FJoint","newPosition");

                                                ...
}
                                                ...
```

Patrones relacionados

Patrón 2, Patrón 5, Patrón 6, Patrón 7.

5.1.1.8 Valuaciones

Patrón 11.	
Modelo PRISMA en DSL	Representación Gráfica
	 
PATRÓN	
Descripción	

Este patrón presenta como se generan las valuaciones dentro de un aspecto.

Plantilla

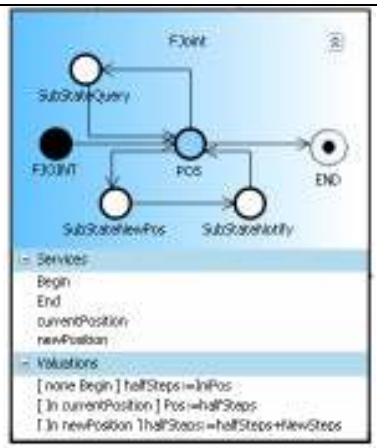
```
...
/* Valuaciones */
foreach(Valuation valuation in service.Valuations)
{
        if(valuation.ServiceModifier != AServiceModifier.Out)
        {
                if (valuation.Condition == String.Empty)
                {
                        foreach(string                    valuationItem                    in
valuation.Assignment.Replace(":=","=").Split(','))
                        {
                          string Item=valuationItem;
                          if(Item.StartsWith(" "))
                                Item=Item.Substring(1);
#>
                        <#=Item#>;
<#
                        }
                }
                else
                {
#>
                         if (<#=valuation.Condition.Replace("=","==").Replace("<>","!=").
                             Replace("<==","<=").Replace(">==",">=").Replace("and","&&").
                             Replace("or","||").Replace("AND","&&").Replace("OR","||")#>)
                        {
<#
                foreach(string valuationItem in
                         valuation.Assignment.Replace(":=","=").Split(','))
                {
                        string Item=valuationItem;
                        if(Item.StartsWith(" "))
                                Item=Item.Substring(1);
#>
                                <#=Item#>;
<#
                                                        }
                        if(valuation.Condition != String.Empty)
                        {
#>
                        }

<#                      }
                }
        }
}
...
```

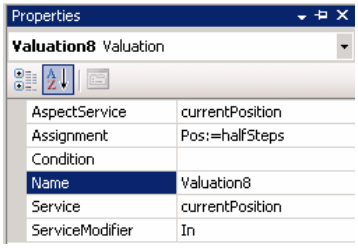
Caso de estudio

Descripción

Como ejemplo del uso valuación, se muestra la asociada al servicio *"currentPosition"* del aspecto *"Fmotion"*.

Representación





Resultado patrón

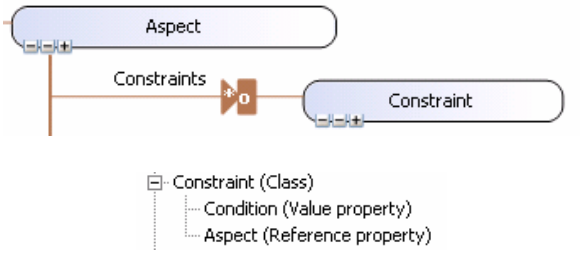
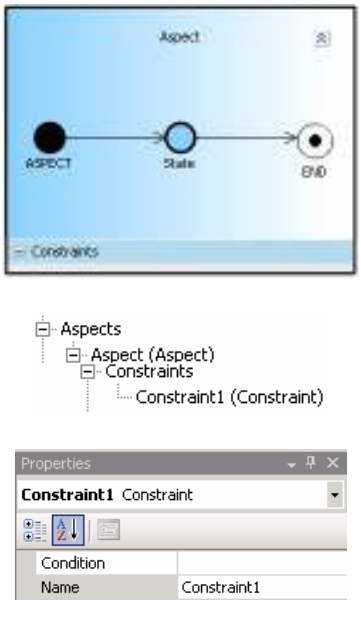
```
                                                ...
public AsyncResult currentPosition(ref int Pos)
{
                                                ...
   Pos=halfSteps;
                                                ...
}
                                                ...
```

Patrones relacionados

Patrón 2, Patrón 5, Patrón 6, Patrón 7.

5.1.1.9 Restricciones (*Constraints*)

Patrón 12.

Modelo PRISMA en DSL	Representación Gráfica
	

PATRÓN

Descripción

Este patrón presenta como se generan las restricciones de un aspecto.

Plantilla

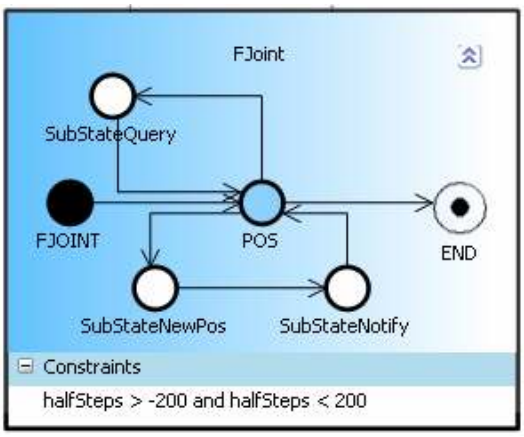
```
...
        /* Comprobación de las Restricciones de Integridad*/
        foreach(Constraint constraint in  aspect.Constraints)
        {
#>
        if (!(<#=constraint.Condition.Replace("=","==").Replace("<>","!=").
              Replace("<==","<=").Replace(">==",">=").Replace("and","&&").
              Replace("or","||").Replace("AND","&&").Replace("OR","||")#>))
  throw new InvalidIntegrityConstraintException("<#=aspect.Name#>","<#=service.Name#>");

<#
        }
...
```

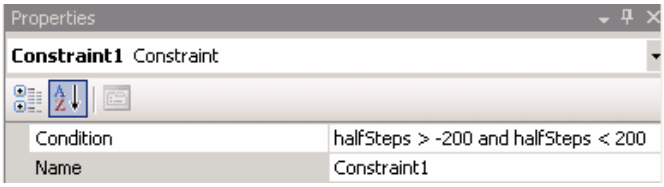
Caso de estudio

Descripción

Como ejemplo de generación de código en C# para una constraint se puede observar el resultado obtenido para la correspondiente a la restricción (*halfSteps > -200 and halfSteps < 200*) del aspecto *"FJoint"*

Representación





Resultado patrón

```
...
public AsyncResult newPosition(int NewSteps)
{
                                        ...
  if (!(halfSteps > -200 && halfSteps < 200))
     throw new InvalidIntegrityConstraintException("FJoint","newPosition");
```

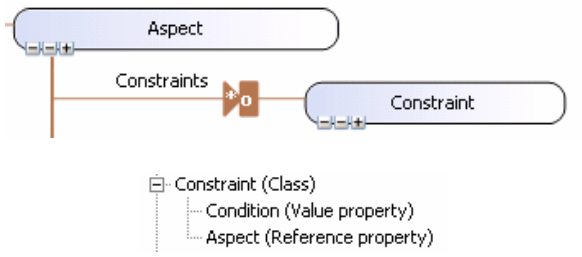
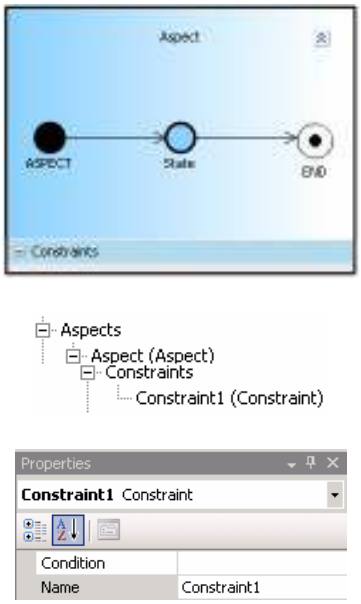
```
...
}

...
```

Patrones relacionados

Patrón 2, Patrón 6, Patrón 7, Patrón 8.

5.1.1.10 Actualización del estado del protocolo

Patrón 13.

Modelo PRISMA en DSL	Representación Gráfica
	

PATRÓN

Descripción

Este patrón muestra como se realiza el cambio del estado del protocolo tras la ejecución de cada uno de los servicios.

Plantilla

```
...
/* Actualizacion del estado del protocolo */
foreach(StateHasState stateHasState in service.StateHasState)
{
  if (!(stateHasState.Source is SubState) &&
       (stateHasState.Transaction ==null && (IsSequence > 0) ) )
  {
       if (stateHasState.Target is SubState)
       {
          SubState subState = stateHasState.Target as SubState;
          System.Collections.IList substateLinks=
           subState.GetElementLinks(subState.Source.TargetRole.Id);
```

```
            StateHasState subStateHasStateProtocol=null;
            int i=0;
            while( i<substateLinks.Count)
            {
             if(substateLinks[i] is StateHasState)
             {
              subStateHasStateProtocol=(StateHasState)substateLinks[i];
              i++;

              if(subStateHasStateProtocol.Target is SubState)
              {
               subState = subStateHasStateProtocol.Target as SubState;
               substateLinks=  subState.GetElementLinks(subState.Source.TargetRole.Id);
               i=0;
              }
             } /*End if(substateLinks[i] is StateHasState)*/
            }/*End While(i<substateLinks.Count) */
#>
          if (state == protocolStates.<#=stateHasState.Source.Name#>

<#
          if (stateHasState.Condition != "")
          {
#>
              && <#=stateHasState.Condition#>

<#        }

#>        )
          state = protocolStates.<#=subStateHasStateProtocol.Target.Name#>;

<#
          }
          else {
#>
          if (state == protocolStates.<#=stateHasState.Source.Name#>

<#
            if (stateHasState.Condition != "")
            {
#>
          && <#=stateHasState.Condition#>

<#          }
#>        )
          state = protocolStates.<#=stateHasState.Target.Name#>;
<#
        }
    }
}
```

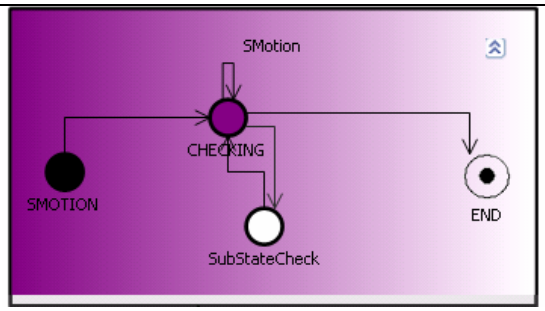
. . .

Caso de estudio

Descripción

Como ejemplo de código generado para actualizar el estado de un aspecto tras ejecutar un servicio, se puede observar el resultado obtenido en el servicio "*currentPosition*" del aspecto "*SMotion*".

Representación



Resultado patrón

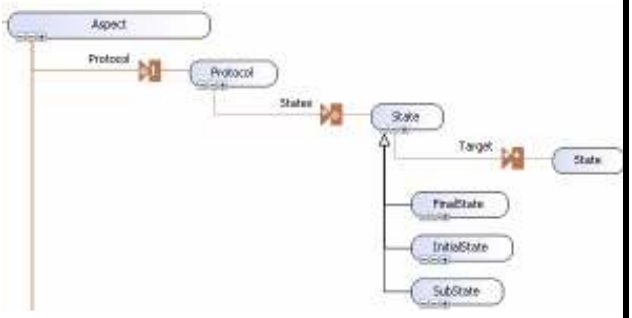
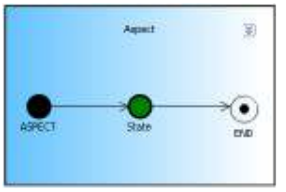
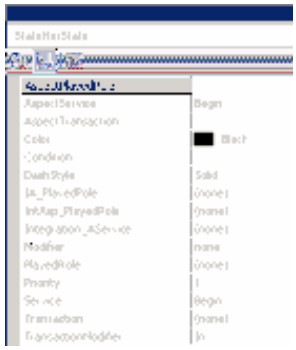
```
                                                ...
public AsyncResult currentPosition(ref int Pos)
{
                                                ...
   if (state == protocolStates.CHECKING )
      state = protocolStates.CHECKING;
                                                ...
}
                                                ...
```

Patrones relacionados

Patrón 2, Patrón 6, Patrón 7, Patrón 8

5.1.1.11 Tratamiento de una secuencia de servicios

Patrón 14.

Modelo PRISMA en DSL	Representación Gráfica
	 

PATRÓN

Descripción

En la definición del protocolo de un aspecto se puede definir una secuencia de servicios

que debe ejecutarse tras la invocación a un servicio dado. Por tanto, en el cuerpo de un servicio pueden aparecer solicitudes de ejecución a otros servicios.

Plantilla

```
                                                                ...
/* Tratamiento de servicios a ejecutar dentro de una secuencia del protocolo*/
int IsSequence=0;
foreach(StateHasState stateHasState in service.StateHasState)
{
   if(stateHasState.Transaction == null )
   {
        if (!(stateHasState.Source is SubState))
        {
                IsSequence++;
                if (stateHasState.Target is SubState)
                {
                        IsSequence--;
                        SubState subState = stateHasState.Target as SubState;
#>
                if (state == protocolStates.<#=stateHasState.Source.Name#>)
                {
                        state = protocolStates.<#=stateHasState.Target.Name#>;
<#
                if(stateHasState.Condition != String.Empty)
                {
#>
                if(<#=stateHasState.Condition#>)
                {
<#
                }
        System.Collections.IList                                                substateLinks=
subState.GetElementLinks(subState.Source.TargetRole.Id);
                StateHasState subStateHasState=null;
               int i=0;
                while( i<substateLinks.Count)
               {
                if(substateLinks[i] is StateHasState)
                {
                        subStateHasState=(StateHasState)substateLinks[i];
                       if(subStateHasState.Condition != String.Empty)
                      {
#>                  if(<#=subStateHasState.Condition#>)
                    {
<#
                      }
                       foreach(Argument element in subStateHasState.Service.Arguments)
                       {
                               if(!attributes.Contains(element.Name)                    &&
!parameters.Contains(element.Name))
                               {
                                       parameters.Add(element.Name,null);
#>
        <#=DomainToType(element.Domain)#> <#=element.Name#>=0;

<#
                               }
                       }
                    /* Invoca a un servicio OUT */
                    if(subStateHasState.Modifier == TransitionModifier.Out)
                    {
#>
    InvokeOutService("<#=subStateHasState.PlayedRole.Interface.Name#>",
                        "<#=subStateHasState.PlayedRole.Name#>",
                        "<#=subStateHasState.Service.Name#>",
                        this.aspectStateCareTaker.ActiveTransaction,
                        <#=CommaSeparatedNames(subStateHasState.Service.Arguments)#>);
<#
                    }
                    /* Invoca a un servicio privado del Aspecto */
                    if(subStateHasState.PlayedRole==null)
                    {
#>
    <#=subStateHasState.Service.Name#>(
        <#=CommaSeparatedNames4(subStateHasState.Service.Arguments)#>);
```

```
<#
                    }
                    /* Ejecuta en servicio IN como servicio privado dentro del
                       aspecto por estar en una transición */
                    if(subStateHasState.Modifier == TransitionModifier.In)
                    {
#>                <#=subStateHasState.Service.Name#>(
                          <#=CommaSeparatedNames4(subStateHasState.Service.Arguments)#>);
<#                  }

                    if(subStateHasState.Condition != String.Empty)
                    {
#>
                  }
<#
                    }
#>
                state=protocolStates.<#=subStateHasState.Target.Name#>;
<#
                    i++;
                    if(subStateHasState.Target is SubState)
                    {
                        subState = subStateHasState.Target as SubState;
                        substateLinks=
                           subState.GetElementLinks(subState.Source.TargetRole.Id);
                        i=0;
                    }
                  } /*End if(substateLinks[i] is StateHasState)*/
                }/*End While(i<substateLinks.Count) */

               if(stateHasState.Condition != String.Empty)
               {
#>             }
<#
          }
#>
        }
<#
        }
      }
    }
}
```

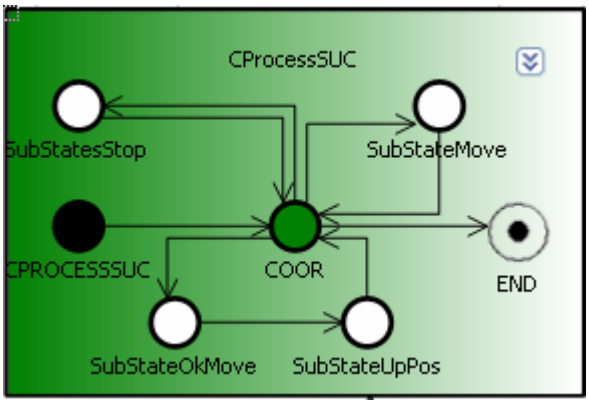
...

Caso de estudio

Descripcion

En el caso de estudio en el aspecto CProcessSUC se tiene una secuencia de ejecución de servicios tras la ejecución del servicio MoveOk.

Representación



Resultado patrón

...

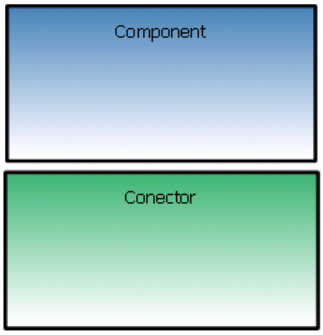
```
public AsyncResult moveOk()
{
  // Modo In
  if(ServiceIn)
  {
        if( state != protocolStates.COOR
            && state != protocolStates.SubStateUpPos
        ) throw new InvalidProtocolStateException("CProcessSUC","moveOk");
        if (state == protocolStates.COOR)
        {
                state = protocolStates.SubStateOkMove;
                int NewSteps=0;

        InvokeOutService("IUpdatePos","UPDATEPOS","newPosition",this.aspectStateCareTaker
.ActiveTransaction,NewSteps);
                state=protocolStates.SubStateUpPos;

        InvokeOutService("IJoint","JOINT","moveOk",this.aspectStateCareTaker.ActiveTransa
ction,null);
                state=protocolStates.COOR;
        }
        return null;

  }
  // Modo Out
  else
  {
        return
CallOutService(this.interfaceName_ServiceOut,this.playedRoleName_ServiceOut,"moveOk",thi
s.aspectStateCareTaker.ActiveTransaction,null);
  }

}
```

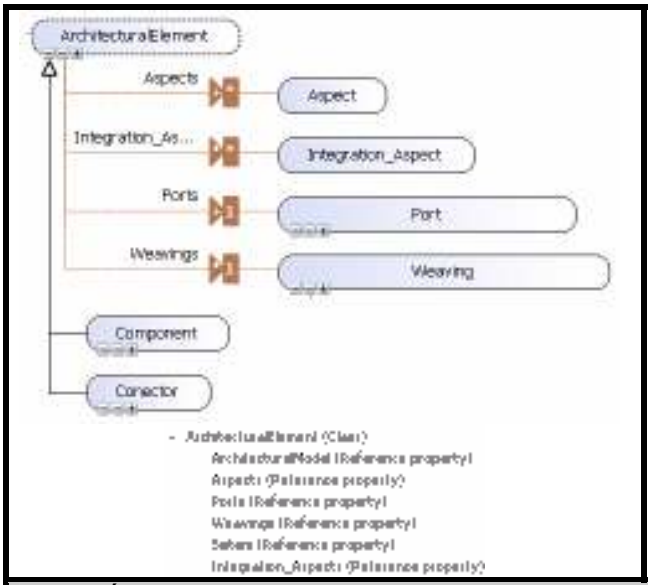
...

Patrones relacionados

Patrón 2, Patrón 5, Patrón 6, Patrón 7.

5.1.1.12 Elementos arquitectónicos simples (Componentes, Conectores)

Patrón 15.	
Modelo PRISMA en DSL	Representación Gráfica
	



PATRÓN

Descripción

Este patrón presenta indica como se genera un elemento arquitectónico simple. En el metamodelo se pueden observar las partes que conforman a todo elemento arquitectónico simple. En este patrón sólo demuestra como se define la transformación a código C# del encabezado de todo elemento arquitectónico. En posteriores patrones se podrá definirá como se obtiene la implementación del resto de elementos que conforman dichos elementos arquitectónicos.

Plantilla

```
                                                    ...
using System;
using System.Reflection;

using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace <#=this.Model.Name#>
{
<#
foreach (ArchitecturalElement architecturalElement in this.Model.ArchitecturalElements)
{
  if (architecturalElement is Component || architecturalElement is Conector)
  {
#>
  [Serializable]
  public class <#=architecturalElement.Name#> : ComponentBase
<#
    if (architecturalElement is Conector)
    {
#>
    , IConnector
<#
    }
#>
  {
    public <#=architecturalElement.Name#>
     (string name<#=ArchitecturalElementArguments(architecturalElement)#> ) : base(name)
    {
<#
                                        /* Aspects */
```

```
                                        /* Weavings */
                                         /* Ports */
#>
    }
  }
<#
  }/* endif (architecturalElement is Component || architecturalElement is Conector)*/
```

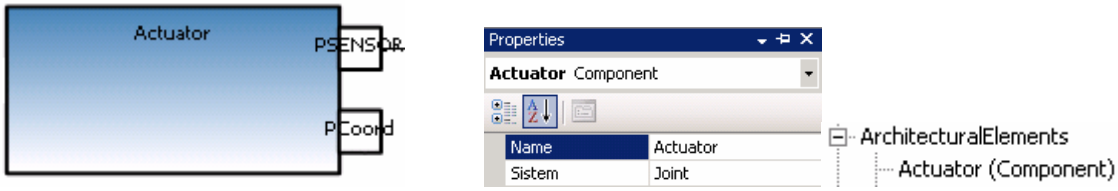
...

Caso de estudio

Descripción

Como ejemplo de generación de código para un elemento arquitectónico, se ha tomado el componente *"Actuator"* del caso de estudio. En este ejemplo se puede ver la representación gráfica así como el resultado en C# tras ejecutar la plantilla de transformación.

Representación



Resultado patrón

...

```
using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace RobotJoint
{
        [Serializable]
        public class Actuator : ComponentBase
        {
                public Actuator(string name ) : base(name)
                {
                  /* Aspects */
                  /* Weavings */
                  /* Ports */
                }
        }
}
```

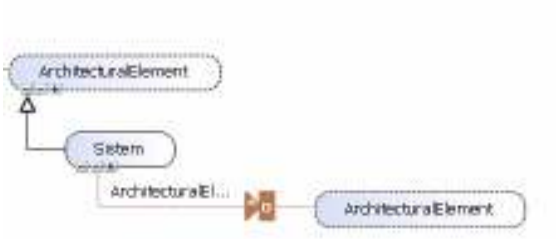
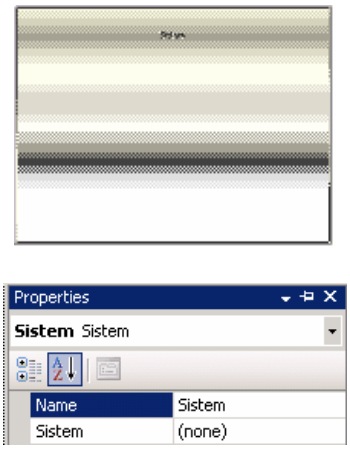
...

Patrones relacionados

Patrón 17, Patrón 18 y Patrón 19.

5.1.1.13 Elementos arquitectónicos complejos (Sistemas)

Patrón 16.

Modelo PRISMA en DSL	Representación Gráfica
	

PATRÓN

Descripción

Este patrón muestra como se genera un elemento arquitectónico complejo. La diferencia principal es que un sistema puede componerse de más elementos arquitectónicos. En la versión actual no se añaden dichos elementos arquitectónicos en este paso del proceso de compilación, ya que cuando se instancien dichos sistemas el usuario indicará las diferentes instancias de los elementos arquitectónicos que pueden conformar dicho sistema.

Plantilla

```
                                                    ...
using System;
using System.Reflection;

using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace <#=this.Model.Name#>
{
<#
foreach (ArchitecturalElement architecturalElement in this.Model.ArchitecturalElements)
{
  if (architecturalElement is Component || architecturalElement is Conector)
  {
...
  }
  else
  {
    if (architecturalElement is Sistem)
    {
      Sistem system = architecturalElement as Sistem;
#>
  [Serializable]
```

```
   public class <#=architecturalElement.Name#> : SystemBase
   {
   public                                          <#=architecturalElement.Name#>(string
name<#=ArchitecturalElementArguments(architecturalElement)#>) : base(name)
   {
<#
                                        /* Aspects */
                                       /* Weavings */
                                        /* Ports */

#>
    }
   }
<#
                        }/*if (architecturalElement is Sistem)*/
                }
        }/*      endforeach      (ArchitecturalElement      architecturalElement      in
this.Model.ArchitecturalElements) */
#>
}
```

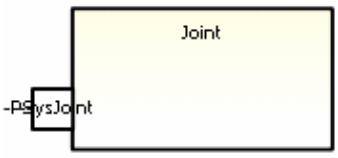
. . .

Caso de estudio

Descripción

Como ejemplo de generación de código para un sistema, se ha tomado el elemento arquitectónico "*Joint*" del caso de estudio. En este ejemplo se puede ver la representación gráfica así como el resultado en C# tras ejecutar la plantilla de transformación.

Representación



Resultado patrón

. . .

```
using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace RobotJoint
{
        [Serializable]
        public class Joint : SystemBase
        {

                public Joint(string name) : base(name)
                {
                  /* Aspects */
                  /* Weavings */
                  /* Ports */
                }
        }
}
```

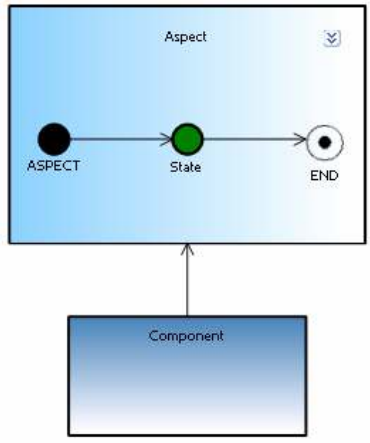
. . .

Patrones relacionados

Patrón 17, Patrón 18 y Patrón 19.

5.1.1.14 Importación de aspectos de un elementos arquitectónico

Patrón 17.

Modelo PRISMA en DSL	Representación Gráfica
	

PATRÓN

Descripción

Este patrón muestra como se genera la importación de los aspectos del elemento arquitectónico, así como el código resultante en C# tras aplicar la plantilla de transformación.

Plantilla

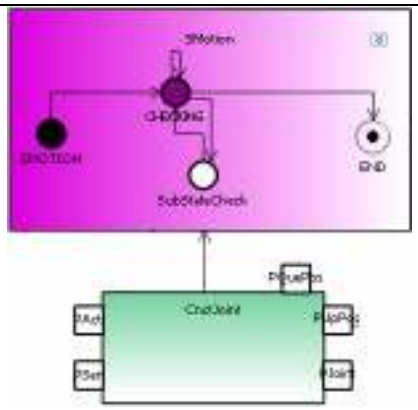
```
                                                    ...
<#
        foreach (Aspect aspect in architecturalElement.Aspects)
        {
#>
        AddAspect(new <#=aspect.Name#>(<#=AspectArguments(aspect)#>));
<#
        } /* endforeach (Aspect aspect in this.Model.Aspects)*/
#>
                                                    ...
```

Caso de estudio

Descripción

Como ejemplo de generación de código para un sistema, se ha tomado el elemento arquitectónico *"CntJoint"* y el aspecto *"Smotion"* del caso de estudio. En este ejemplo se puede ver la representación gráfica así como el resultado en C# tras ejecutar la plantilla de transformación.

Representación



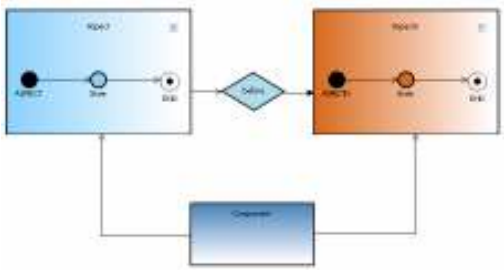
Resultado patrón

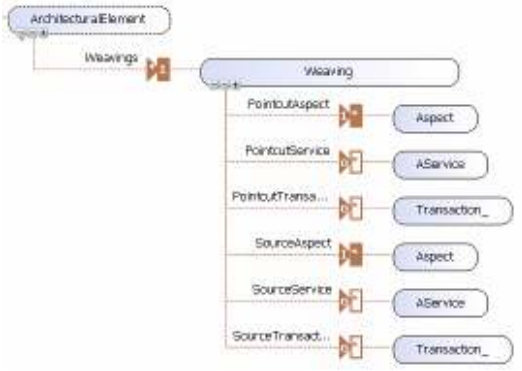
```
                                                    ...
using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace RobotJoint
{
  [Serializable]
  public class CnctJoint : ComponentBase , IConnector
  {

        public CnctJoint(string name, int IniMin, int IniMax, int IniPos ) : base(name)
        {
          AddAspect(new CProcessSUC());
          AddAspect(new SMotion(IniMin, IniMax, IniPos));
                                                    ...
        }
  }
}
                                                    ...
```

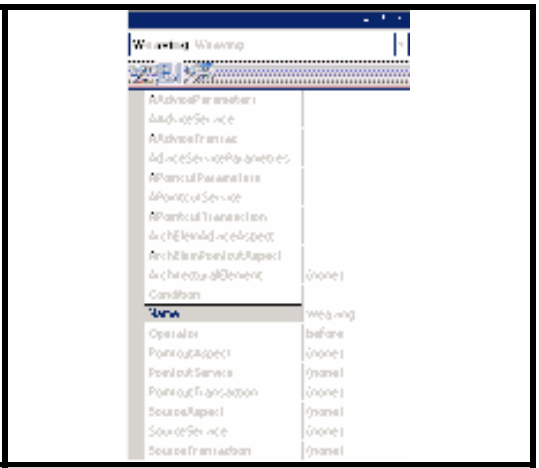
Patrones relacionados

Patrón 14, Patrón 15

5.1.1.15 Weavings

Patrón 18.	
Modelo PRISMA en DSL	Representación Gráfica
	





PATRÓN

Descripción

Este patrón presenta como se obtiene el código necesario para dar soporte a las relaciones de *weavings* entre los distintos servicios de los aspectos que importa un elemento arquitectónico. Los distintos tipos de weavings los podemos agrupar basicamente en dos, los condicionales y lo que no precisan de una condición para ejecutarse. Esto simplifica la generación de código ya que el resto de variables a tener en cuenta en la definición del weaving ya se encuentran parametrizadas y las indica el usuario.

Plantilla

```
...
<#
foreach (Weaving weaving in architecturalElement.Weavings)
{
        if( weaving.Operator.Equals(WeavingOperator.before) ||
            weaving.Operator.Equals(WeavingOperator.after) ||
            weaving.Operator.Equals(WeavingOperator.insteadof) )
        {
#>
        AddWeaving(GetAspect(typeof(<#=weaving.SourceAspect.Concern#>Aspect)),
                    "<#=weaving.SourceService.Name#>","<#=weaving.APoincutParameters#>",
                    WeavingType.<#=weaving.Operator.ToString().ToUpper()#>,
                    GetAspect(typeof(<#=weaving.PointcutAspect.Concern#>Aspect)),
                    "<#=weaving.PointcutService.Name#>",
                    "<#=weaving.AAdviceParameters#>");

<#
        }
        else
        {
                //El separador dentro de la condición debe ser un espacio.
                string parametro_condicion=weaving.Condition.Split(' ')[0];
                string operador=weaving.Condition.Split(' ')[1];
                string valor_condicion=weaving.Condition.Split(' ')[2];
#>
  WeavingType weavingType =
                WeavingType.<#=ChangeWeavingType(weaving.Operator.ToString())#>
                            ("<#=parametro_condicion#>",
                             WeavingType.OperatorType.<#=ChangeOperator(operador)#>,
                             <#=valor_condicion#>);

        AddWeaving(GetAspect(typeof(<#=weaving.SourceAspect.Concern#>Aspect)),
                    "<#=weaving.SourceService.Name#>","<#=weaving.APoincutParameters#>",
                    weavingType,
                    GetAspect(typeof(<#=weaving.PointcutAspect.Concern#>Aspect)),
                    "<#=weaving.PointcutService.Name#>",
                    "<#=weaving.AAdviceParameters#>");
```

```
<#
        }
}/* endforeach (Weaving in architecturalElement.Weavings) */
#>
```

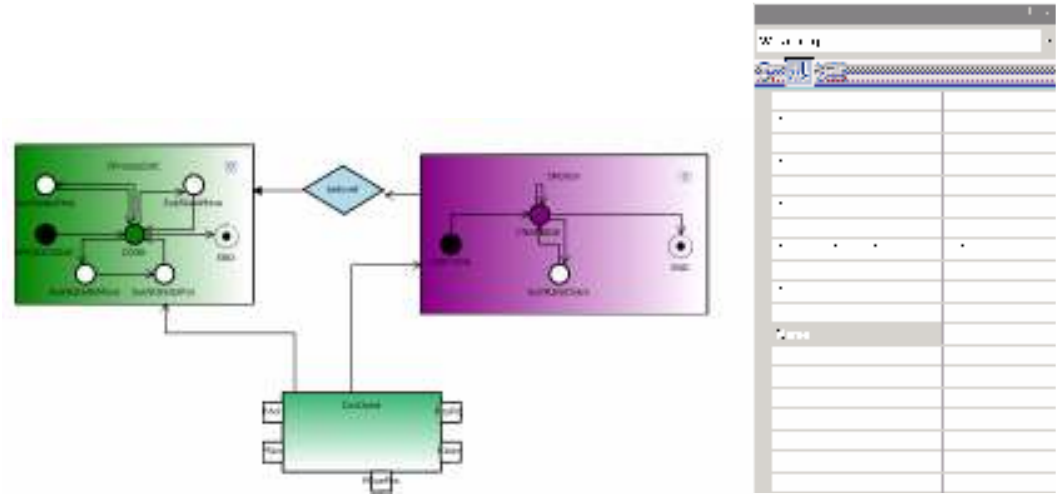
...

Caso de estudio

Descripción

Como ejemplo de generación de código para un weaving, se ha escogido el weaving *BefoteIf* dentro del conector "*CnctJoint*" entre los aspectos "*CProcessSUC*" y "*Smotion*". Los servicios que relaciona son el servicio "*moveJoint*" con el "*Check*" asociados respectivamente a los aspectos antes mencionados.

Representación



Resultado patrón

...

```
using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace RobotJoint
{
  [Serializable]
  public class CnctJoint : ComponentBase , IConnector
  {

        public CnctJoint(string name, int IniMin, int IniMax, int IniPos ) : base(name)
        {
```

...

```
WeavingType                                                              weavingType=
WeavingType.BEFOREIF_VALUE("Secure",WeavingType.OperatorType.Equality,true);

AddWeaving(GetAspect(typeof(SafetyAspect)), "check","NewSteps,Secure", weavingType,
           GetAspect(typeof(CoordinationAspect)), "moveJoint", "NewSteps,Speed");
```

...

```
        }
  }
}
```

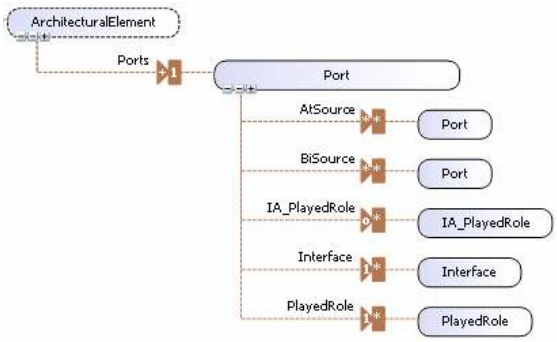
...

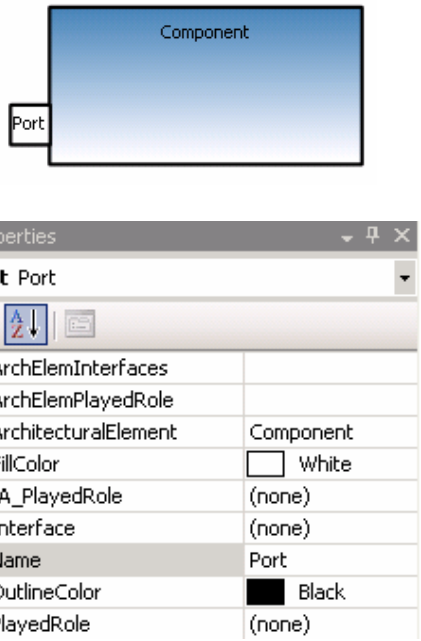
Patrones relacionados
Patrón 14, Patrón 15

5.1.1.16 Elementos arquitectónicos: Puertos

Patrón 19.

Modelo PRISMA en DSL	Representación Gráfica





PATRÓN

Descripción

Este patrón indica como se obtiene el código asociado a los puertos de los elementos arquitectónicos.

Plantilla

```
                                                    . . .
<#
foreach (Port port in architecturalElement.Ports)
{
        if(port.PlayedRole !=null)
        {
#>

InPorts.Add("<#=port.Name#>", "<#=port.Interface.Name#>", "<#=port.PlayedRole.Name#>");

OutPorts.Add("<#=port.Name#>", "<#=port.Interface.Name#>", "<#=port.PlayedRole.Name#>");

<#
        }
        else if(port.IA_PlayedRole !=null)
        {
#>
InPorts.Add("<#=port.Name#>", "<#=port.Interface.Name#>",
```

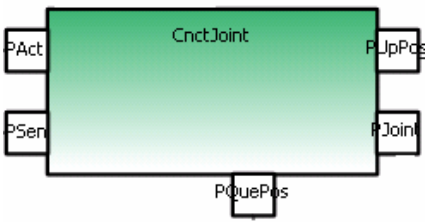
```
            "<#=port.IA_PlayedRole.Name#>");

OutPorts.Add("<#=port.Name#>", "<#=port.Interface.Name#>",
             "<#=port.IA_PlayedRole.Name#>");

<#      }
 }/* endforeach (Port port in architecturalElement.Ports) */
#>
                                                    ...
```

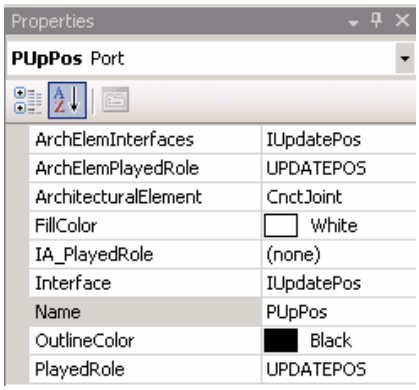
Caso de estudio

Descripción

Siguiendo con el conector utilizado en las plantillas anteriores, "*CnctJoint*", se muestra la definición de uno de sus puertos *(PUpPos)* y el resultado tras ejecutar la plantilla antes descrita.

Representación





Resultado patrón

```
                                                    ...
using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace RobotJoint
{
  [Serializable]
  public class CnctJoint : ComponentBase , IConnector
  {

        public CnctJoint(string name, int IniMin, int IniMax, int IniPos ) : base(name)
        {
                                                    ...

         InPorts.Add("PUpPos", "IUpdatePos", "UPDATEPOS");
         OutPorts.Add("PUpPos", "IUpdatePos", "UPDATEPOS");
                                                    ...
        }
  }
}
                                                    ...
```

Patrones relacionados

Patrón 14, Patrón 15

5.1.2 Plantillas de Transformación PRISMAConfiguration → XML

Una vez definida una arquitectura PRISMA utilizando la herramienta PRISMADSL, hay que definir una plantilla que pueda recorrer los elementos arquitectónicos que el usuario desee instanciar.

Para ello hay que crear una plantilla en la herramienta PRISMADSL, que como resultado de su ejecución obtenga una plantilla que se ejecutará en la herramienta PRISMA-CONFIGURATION.

Esta plantilla tiene como finalidad conocer los elementos arquitectónicos que forman el modelo definido por el usuario, es decir, los tipos que se han definido. Esto se hace necesario dado que el metamodelo que conforma la herramienta PRISMA CONFIGURATION es el modelo definido por el usuario en PRISMA DSL, y esta plantilla debe conocer los elementos de ese metamodelo para ser capaz de recorrer dicho modelo.

De esta forma cuando el usuario utilice la aplicación que proporciona los tipos del modelo arquitectónico definido en la herramienta PRISMADSL para definir la configuración inicial (PRISMA CONFIGURATION) del modelo, dispondrá de una plantilla que recorrerá los tipos definidos en el modelo, para generar el código de las instancias que defina el usuario para así, disponer de la información necesaria para que el middleware PRISMANET pueda cargar una arquitectura y lanzarla a ejecución.

En el caso de las plantillas de C# es más sencillo ya que tenemos definido el metamodelo PRISMA, y conocemos a priori que elementos hemos de recorrer para obtener la información necesaria en cada momento. Sin embargo, para la configuración se debe recorrer el modelo que defina el usuario, y como es obvio, será distinto en cada momento. Por todo ello se proporciona el mecanismo necesario para conseguir el resultado esperado, sin necesidad de que el usuario tenga que añadir nada en el código de las plantillas de transformación.

La complejidad de esta plantilla reside en su interpretación y legibilidad. Dado que ya es confuso ver una plantilla normal en la que se entrelaza el código que permite recorrer el modelo para extraer la información necesaria para generar el código, con el código a inyectar en el fichero destino, que será el resultado del patrón. La complejidad de esta plantilla es mayor, ya que es una meta-plantilla. Una plantilla que va a generar otra plantilla para que luego se comporte como una plantilla normal en la herramienta PRISMA CONFIGURATION.

Por lo tanto, se ha definido una simbología para poder generar el léxico necesario para las plantillas, por ejemplo <# por ABIERTO. Al finalizar la transformación de las plantillas hay que sustituir estos símbolos por los correctos según el léxico de las plantillas *ReportTemplate* de DSL Tools.

Los datos necesarios en cualquier elemento arquitectónico son, su tipo y los atributos necesarios para su creación, así como el tipo de cada uno de los mismos.

Toda esta información se utiliza, para proporcionar al usuario una interfaz gráfica que le permite modelar la arquitectura inicial, así como proporcionar todos los datos necesarios para su posterior instanciación y lanzamiento a ejecución.

Una vez el usuario defina la configuración inicial de la arquitectura y, tras su posterior verificación, se podrá lanzar a ejecución la arquitectura y el analista podrá ver el correcto funcionamiento de su modelo, sin haber tenido que implementar ni una sola línea de código.

Una vez definidos los valores necesarios de lo que contendrá la arquitectura a lanzar a ejecución, desde la aplicación PRISMA-Configuration, se genera un XML con todos estos valores.

La estructura de este XML es la siguiente.

```
<?xml version="1.0" encoding="utf-8"?>
<ConfigurationModel name = "_operation">
  <Systems>
    <System name = "" type = "">
    </System>
  </Systems>

  <Components>
    <Component name = "" type = "">
      <Properties>
        <Property name = ""
                  type = "" value = "">
        </Property>
      </Properties>

      <SystemRef name = "">
      </SystemRef>

    </Component>
  </Components>

  <Connectors>
    <Connector name = "" type = "">
      <Properties>
        <Property name = ""
                  type = "" value = "">
        </Property>
      </Properties>

      <SystemRef name = "">
      </SystemRef>
    </Connector>
  </Connectors>

  <Attachments>
     <Attachment type = "">
        <source name = "" port = "">
        </source>
        <target name = "" port = "">
```

```
        </target>
        <SystemRef name = "">
        </SystemRef>
     </Attachment>
  </Attachments>

  <Bindings>
     <Binding type = "">
        <source name = "" port = "">
        </source>
        <target name = "" port = "">
        </target>
        <SystemRef name = "">
        </SystemRef>
     </Binding>
  </Bindings>

</ConfigurationModel>
```

La información del XML generado es utilizada por la herramienta PRISMANET para poder lanzar a ejecución la configuración realizada por el usuario.Los datos necesarios para poder realizar esta tarea son:

Elementos arquitectónicos
Atributos de los constructores de los aspectos.
Relaciones de *Attachments* y *Bindings* entros los elementos arquitectónicos.
Relación de inclusión de elementos arquitectónicos en sistemas.

Con toda esta información, como se muestra en la siguiente sección, se puede lanzar a ejecución la configuración establecida.

5.2 Integración PRISMADSL con PRISMANET

Una vez generado el código en C# asociado al modelo definido por el usuario, así como la definición de la configuración inicial de la arquitectura, el siguiente paso es lanzar a ejecución sobre el Middleware PRISMANET la arquitectura definida en base a la configuración realizada.

Esta sección se presenta como se han realizado las diversas tareas en el Middleware PRISMANET, para dar soporte a las necesidades antes mencionadas.

En primer lugar se establecen los datos necesarios de entrada para que el Middleware sea capaz de lanzar a ejecución una arquitectura PRISMA.

Estos datos de entrada son:

- Código en C# compilado del modelo definido por el usuario.
  - ArquitecturalElement.dll
  - Aspect.dll
  - Interface.dll
  - IAspect.dll
- XML con la información de la configuración inicial del modelo definido por el usuario.

Una vez se han mostrado los datos de entrada necesarios se puede ver el diagrama de clases necesarias para realizar el proceso de lanzar a ejecución una arquitectura PRISMA bajo el middleware PRISMANET. (Ver Figura 28)

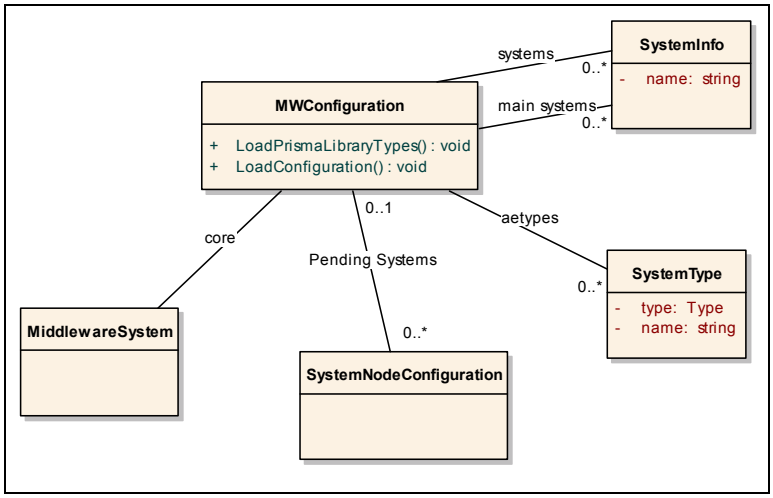


Figura 28. Diagrama clases Middleware Configuration

La clase *MWConfiguration* tiene toda la lógica necesaria para leer toda la información tanto del XML como del código compilado con los tipos de los elementos arquitectónicos, para posteriormente, utilizando servicios del Middleware, cargarlos y lanzarlos a ejecución.

En primer lugar se deben obtener los tipos definidos en el ensamblado *"ArchitecturalElement.dll"* que se ha definido desde la herramienta PRISMA-DSL. Los únicos tipos necesarios son los de componentes, conectores y sistemas. Una vez leída esta información, se almacena para su posterior uso.

Después de haber obtenido los tipos de los elementos arquitectónicos se procede a leer la información del XML.

El orden en el cual se leen los datos es el siguiente.

1. Sistemas.
2. Componentes.
3. Conectores.
4. Attachments.
5. Bindings.

Este orden viene dado por al forma en la cual esta implementada la creación de elementos arquitectónicos en el middleware. En primer lugar hay que cargar los sistemas que no pertenecen a otros en si. Y una vez se han creado estos, el resto de elementos se les van añadiendo.

En la carga de los sistemas, dado que un sistema puede estar formando parte de otro, si al leer el XML no se hace en el orden adecuado, hay que almacenar la información necesaria para su creación en un momento posterior. Para ello se ha definido la clase *SystemNodeConfiguration*.

Una vez creados todos los elementos arquitectónicos y sus relaciones (*Attachments* y *Bindings*), hay que lanzarlos a ejecución. Para ello, simplemente hay que ejecutar el servicio "*Start"* de cada uno de los sistemas principales (Los que no se encuentran incluidos en otros), y esto desencadena el inicio de la ejecución de toda la arquitectura.

5.3 Middleware Console

Hasta ahora cualquier arquitectura definida en Prisma necesitaba de un Aspecto de presentación para poder interactuar el usuario con el modelo implementado. Esto complica la generación automática de código dada la necesidad de proporcionar una herramienta que permita la definición de interfaces gráficas de usuario que a su vez interactúen con un Aspecto de presentación asociado. Dado la envergadura que podría suponer realizar un herramienta de este tipo, se deja al propio analista la función de implementar a mano las interfaces de usuario y enlazarlas con el aspecto de presentación del modelo arquitectónico que se desarrolle.

Como consecuencia, y dado que la finalidad de la herramienta PRISMA-DSL, es la utilización de la misma, para la posterior validación y mejora de los modelos arquitectónicos, así como su uso en medios académicos o de investigación a priori, surge la necesidad de proporcionar un mecanismo que posibilite la interacción gráfica con la arquitectura en ejecución sin necesidad de tener que crear ninguna interfaz por parte del analista.

Por todas estas razones se ha definido la "Consola del Middleware" la cual proporciona un medio de interacción entre el usuario y los elementos arquitectónicos definidos. Esta comunicación debe permitir ver el estado de los componentes en ejecución, así como poder ejecutar los servicios que cada unos de los mismos sirve al exterior a través de sus puertos.

De esta forma, simplemente modelando mediante la herramienta PRISMA-DSL y generando el código se puede lanzar a ejecución el sistema y poder comprobar si su funcionamiento es correcto.

Una vez modelada una arquitectura PRISMA y verificado su correcto comportamiento, se podría pasar a realizar, ya de forma manual, la implementación de la interfaz gráfica de usuario.

El siguiente diagrama de clases muestra la estructura necesaria para dar soporte a la consola.

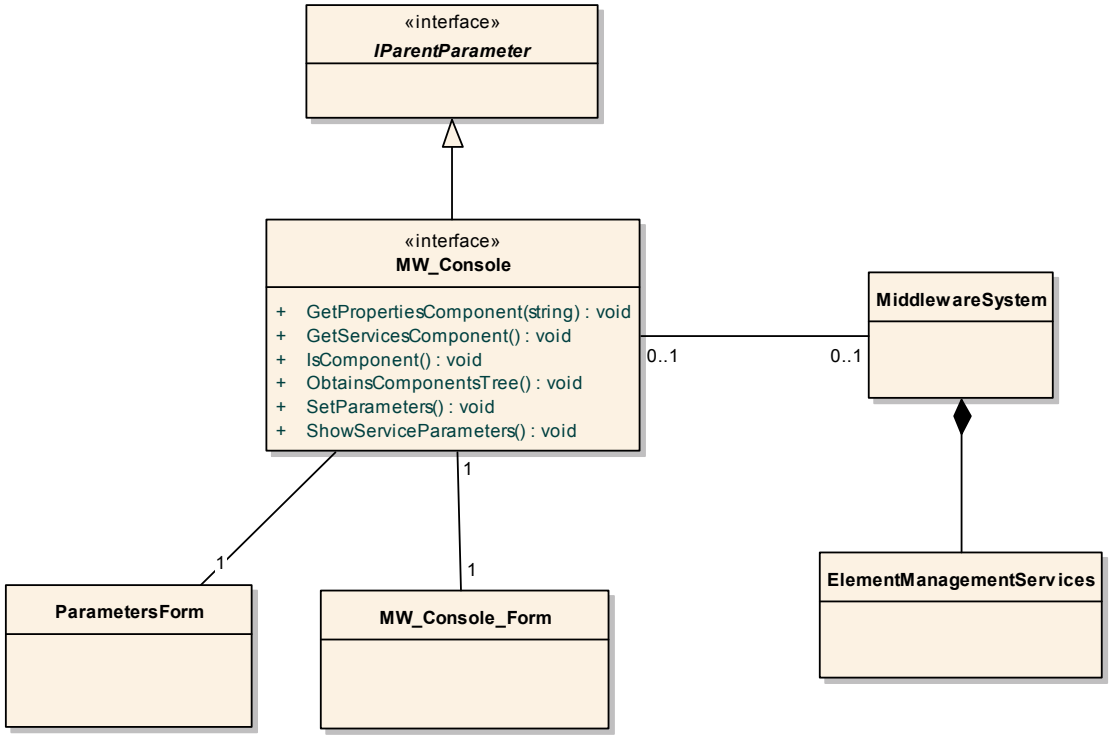


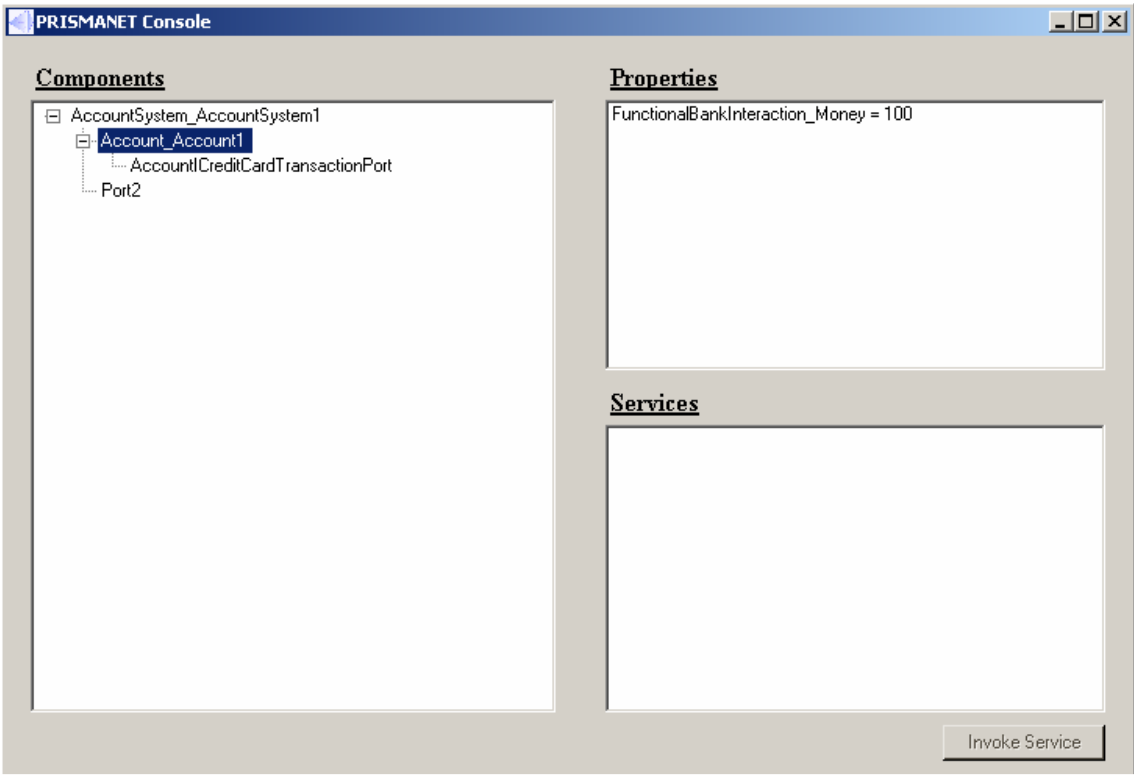
Figura 29. Diagrama de clases de soporte a *Middleware Console*

La forma en la que esta diseñado se puede ver como una distribución entre varias capas. Por una parte la capa de presentación sería provista por *MW_Console_Form*, que permite al usuario interactuar con los elementos instanciados en cada momento.La clase *MW_Console* hace un papel de mediador entre la capa de presentación y el middleware. De esta forma, la información necesaria desde el formulario que ve el usuario es obtenida a través de esta clase, la cual se encarga de obtenerla del middleware y procesarla. Posteriormente entraremos en algunos detalles sobre ella para conseguir una mejor eficiencia en la ejecución de la aplicación. Por último estaría la capa de middleware ElementManagementServices, que se encarga de proveer toda la información necesaria.

Otras clases necesarias son la interfaz *IParentParameter* y *ParametersForm*. Esta interfaz y este formulario son utilizados desde distintitas partes de la aplicación para obtener parámetros del usuario de una forma dinámica, por lo que el formulario se define cada vez con los parámetros necesarios y la interfaz permite comunicar el formulario o clase que necesita estos parámetros, para poder ser obtenidos una vez indicados por el usuario.

El usuario necesita poder consultar el estado de cada uno de los componentes lanzados a ejecución. Este estado viene dado por los aspectos que componen cada unos de estos elementos arquitectónicos, pero para mantener la encapsulación definida en PRISMA, no se ha entrado en detalle de seleccionar cada uno de los diferentes aspectos. Otra necesidad a cubrir sería poder visualizar los diferentes servicios que sirven cada uno de los componentes por cada uno de los puertos y poder ser ejecutados.

A continuación vemos una captura de la interfaz de usuario resultante, donde se puede ver con más detalle cada una de estas partes.



En la parte izquierda se ven, en estructura de árbol, los diversos componentes que se encuentran en ejecución. De esta forma se puede observar la composición de componentes, ya que en la raíz estarían los sistemas que contienen al resto de componentes.

En la parte superior derecha se ven las propiedades de los componentes, y en la parte inferior derecha se muestran los distintos servicios para poder ser ejecutados.

Una vez vistos los distintos requisitos, vamos a profundizar en lo necesario para proporcionar toda la información y servicios necesarios.

En primer lugar, el middleware a través de su capa ElementManagementServices, tiene que proporcionar diversos servicios para poder obtener tanto información de que elementos arquitectónicos hay en ejecución, como de su estado y de los distintos servicios ofrecidos por sus puertos de entrada.

Por supuesto, toda esta información debe ser obtenida de todos los middlewares en ejecución, dado que la consola proporciona una interacción con todo el sistema en ejecución, este donde este. Dado que la consola estará en ejecución en un middleware, accederá a toda la información necesaria a través del mismo, obteniendo si es necesario información de otros posibles middlewares.

Al inicio hay que obtener los diversos componentes y sus puertos de entrada. Dado que en el middleware, los componentes se identifican por su nombre, y si un componente esta incluido en un sistema, mediante el nombre y con la notación punto podemos conocer esta relación, se obtienen todos estos datos en forma de cadenas de

caracteres para construir el árbol que se le muestra al usuario y poder acceder posteriormente a consultar la información relevante de cada uno de los mismos.

Este primer paso, en principio, solo se dará al inicio de la ejecución, ya que si no hay una reconfiguración dinámica sobre la arquitectura no podría haber nuevos componentes en ejecución.

Una vez obtenida esta información, la clase MW_Console, se encarga de crear una estructura de árbol para posteriormente ser tratada por la capa de presentación y mostrarla de una forma correcta.

En segundo lugar hay que proporcionar al usuario la posibilidad de consultar el estado de los diversos componentes.

Para ello, en primer lugar el middleware debe obtener en cada momento el estado de los diversos aspectos que componen el componente que se desea consultar. Esto se ha realizado a través de los mecanismos de reflexión que proporciona .NET. De esta forma se obtienen solo las propiedades que son definidas por el usuario, no las que tienen que ver con la ejecución propia de los aspectos.

Esta tarea, debe ser solicitada siempre al middleware, dado que el estado es algo que puede estar siempre cambiando, y debe solicitarse al middleware concreto en el cual se encuentra en ejecución el componente solicitado. Para que el acceso sea más rápido hay ya definida una cache en el middleware que almacena la localización en la cual se encuentra un componente.

En último lugar, esta la parte de invocación de los servicios. Dado que los servicios vienen asociados a los diversos puertos de entrada, y es algo que de momento no cambia durante la ejecución, se ha definido una cache que almacena toda la información referente a los mismos, tanto los distintos servicios de un puerto como la información relevante sobre los parámetros que necesita un servicio para ser ejecutado.

La primera vez que se consulta un puerto de entrada de un componente se solicitan al middleware los diversos servicios que se ofrecen por el mismo, y se añade la información a la cache para ser accedida posteriormente. Una vez seleccionado por el usuario un servicio a ejecutar y tras lanzar a ejecución el mismo, se obtienen los diversos parámetros requeridos por el servicio. Si el servicio tiene parámetros de entrada se piden al usuario a través de un formulario que se crea automáticamente para cada servicio. Si no tiene parámetros el servicio o son todos de salida, se invoca el servicio directamente.

Para poder invocar un servicio es necesario en primer lugar obtener los parámetros y hacer un casting de tipos para poder ejecutar correctamente el servicio. Cuando ya se dispone de toda la información necesaria el middleware se encarga de encolar dicha petición de servicio en el puerto de entrada indicado. Este encolado se realiza en el middleware que tenga la instancia del componente sobre el que estamos trabajando.

La estructura de clases necesaria para mantener esta cache sobre los servicios es la siguiente.

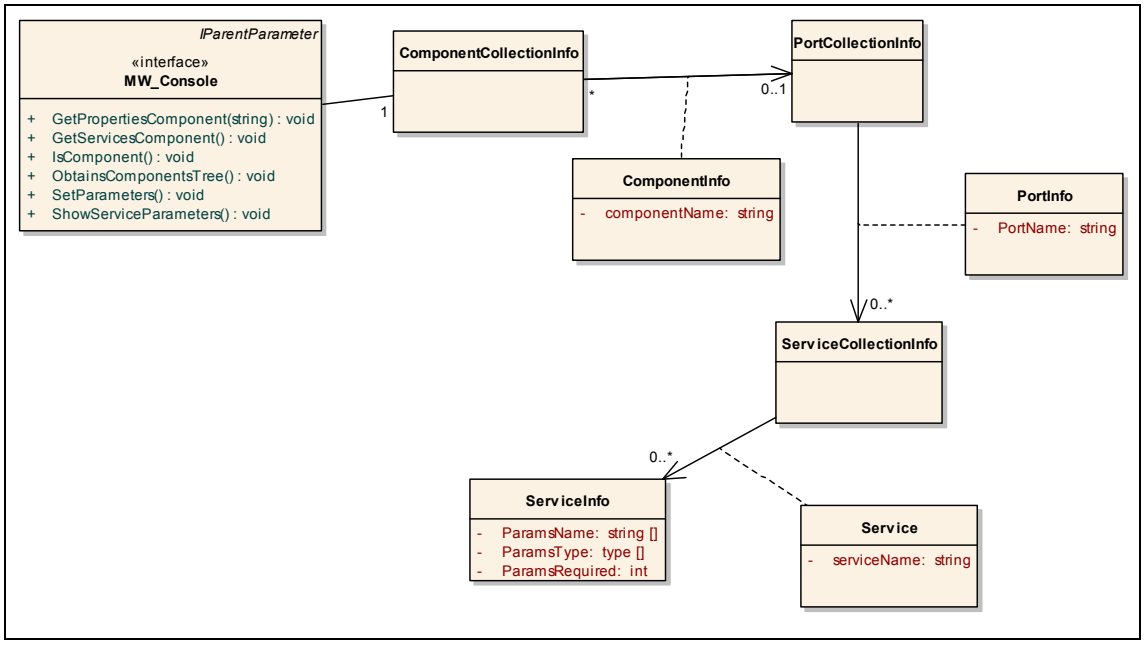


Figura 30. Diagrama de clases MiddlewareConsole

- A continuación se muestra un diagrama de secuencia en el cual podemos ver el orden se servicios a ejecutar para la creación de la consola.

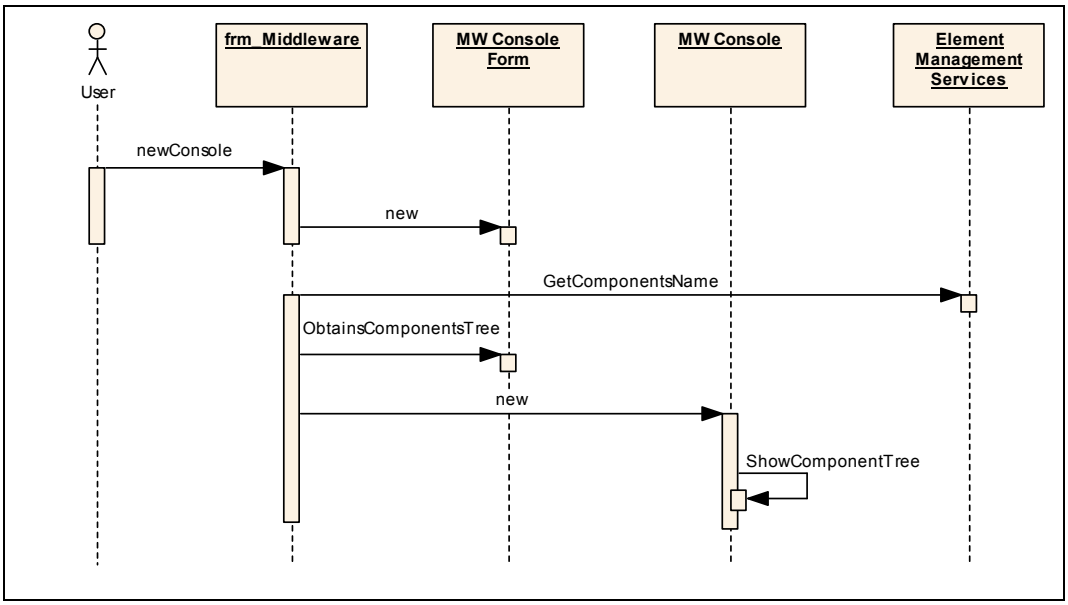


Figura 31. Diagrama secuencia creación consola

- Diagrama de secuencia para la consulta de propiedades de un componente

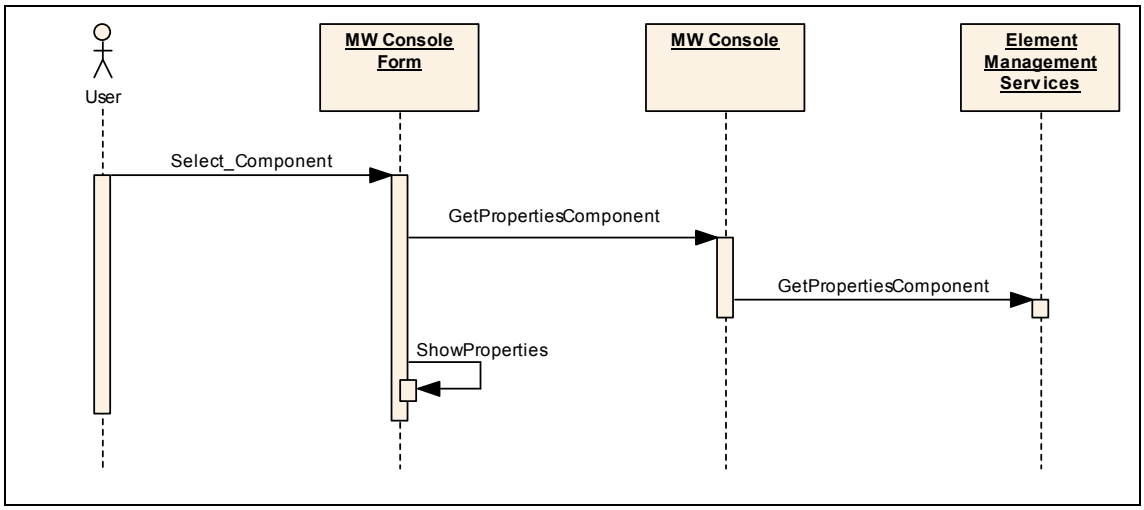


Figura 32. Diagrama de secuencia consulta propiedades componentes

- Diagrama secuencia consulta y ejecución servicios de un puerto de un componente.

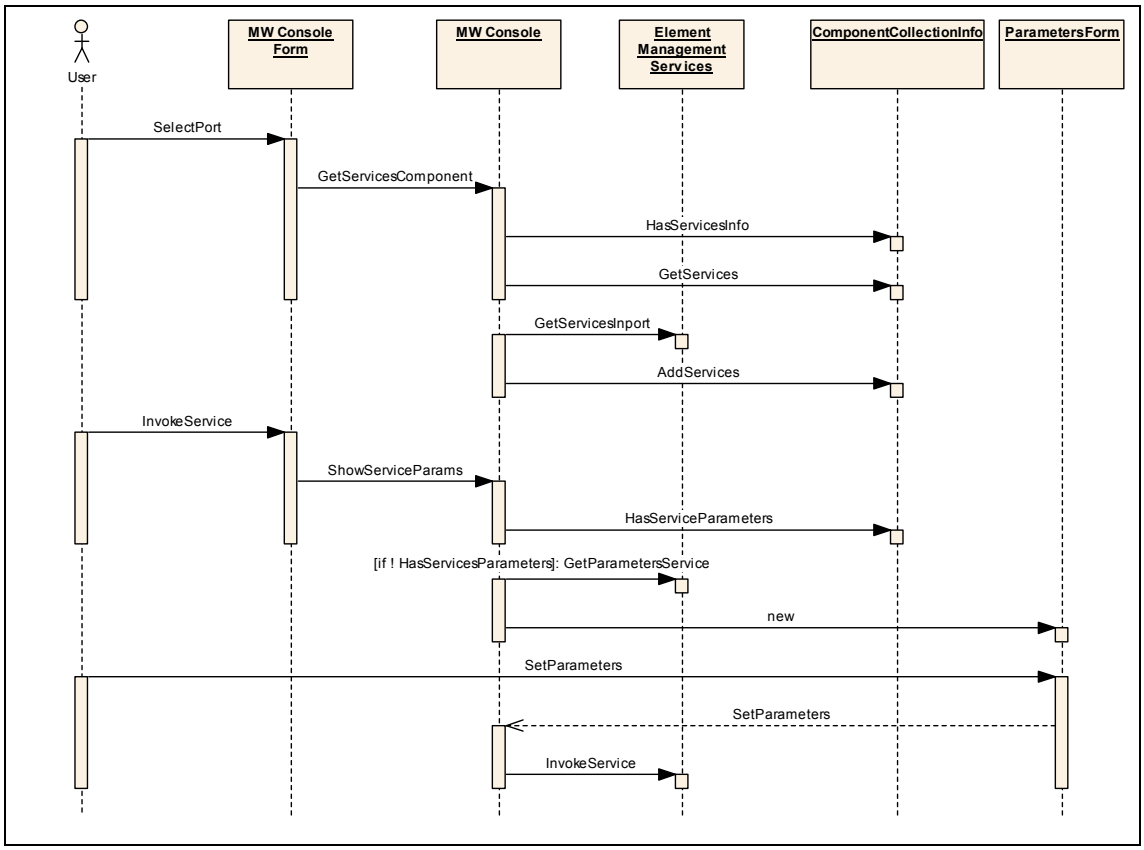


Figura 33. Diagrama de secuencia ejecución servicios

5.4 Aspecto de Integración. COTS.

La aproximación elegida para resolver la integración del COTS en PRISMA es un nuevo tipo de aspecto para que actúe como wrapper del COTS, de tal forma que se define una nueva interfaz, que implementará el aspecto de integración a través del COTS. Así, en el caso de querer extender el comportamiento del COTS, sólo es necesario definir nuevos servicios para que los implemente el aspecto de la misma forma que en cualquier otro aspecto. Por otro lado, para que el resto de elementos arquitectónicos puedan interactuar con el COTS es necesario que el aspecto de integración se incluya dentro de un componente. Este componente será el encargado de publicar los servicios que implementa el aspecto de integración a través de un puerto. Además, emplear un aspecto como wrapper, ofrece la posibilidad de conectar sus servicios con otros aspectos. Es decir, si en el componente donde se incluye el aspecto de integración, también se incluyen otros tipos de aspecto, se pueden crear dependencias entre los servicios del COTS y los servicios de los demás aspectos mediante el weaving, de la misma forma que para el resto de aspectos. Incluso, se podría interconectar servicios de distintas COTS si se incluyeran tantos aspectos de integración como COTS y se definiera weavings entre sus servicios.

Para llevar a cabo la comunicación entre el Aspecto de Integración de las Cots, habría que crear un wrapper entre ambos. Esto es necesario dada la necesidad de homogeneizar el acceso a librerías nativas como librerías realizadas desde la plantaforma .NET. Por ello, en el Aspecto de Integración se debe agregar la referencia a la librería que contenga el wrapper, así como la redirección de cada uno de los servicios del aspecto hacia la clase estática que tenga la implementación de los mismos.

Ejemplo.-

Librería: cots.dll
Servicio: send(int joint,int halfsteps,int speed);

```
using cots;

...

Public AsyncResult send (int joint,int halfsteps,int speed)
{
      cots.send( joint, halfsteps, speed);
}
```

El usuario por su parte debe proporcionar esta librería que contiene el wrapper entre el Aspecto de Integración y las COTS. Para ello tiene dos formas de realizarlo.

- Comunicación directa con COTS.
- Crear una capa para gestionar el acceso del COTS.

El caso más sencillo, es aquel en el cual se quiere utilizar los servicios de las COTS directamente. Simplemente se tiene que utilizar el atributo DllImport, en la cual se indica el nombre de la librería así como el nombre del servicio. De esta forma se crea una redirección hacia el servicio de las COTS. Habría que realizar esto con cada uno de

los servicios que se quiera trabajar. Esto solo se puede realizar con librerías nativas, ya que si se trata de una biblioteca de clases realizada desde la plataforma .NET no se pueden definir los puntos de entrada dentro del ensamblado.

```
[DllImport("TeachMover.dll", EntryPoint="send")]

private static extern int SendCOTS(int joint,int halfsteps,int speed);
```

La otra opción que tiene un usuario viene dada por la necesidad de querer proporcionar una envoltura sobre los servicios definidos en las COTS. En este caso el usuario definirá una biblioteca de clases y creara los servicios que incluyan toda la funcionalidad que se desee. En el caso concreto del Robot, el acceso hacia el puerto serie para manejar sus movimientos, requiere invocar una serie de servicios así como la inicialización de algunos parámetros que permitan una correcta comunicación. Por ello se ha definido un proyecto de biblioteca de clases en C# en el cual se proporciona toda esta funcionalidad. De este modo cuando

En el caso de estudio del brazo del robot, se integra un COTS que mantiene la comunicación por el puerto serie del ordenador con el robot. Esta COTS es una dll que dispone de un único servicio para mandarle la velocidad y la cantidad de pasos que debe moverse una articulación. Este servicio después de su ejecución devuelve como resultado si el movimiento del robot ha sido correcto o no.

A partir de la documentación del COTS se puede identificar el nombre del COTS, el nombre del servicio, el número y el tipo de los parámetros que necesita para su invocación y por último el tipo del valor de retorno del servicio. La definición del servicio es la siguiente:

```
send(int joint,int halfsteps,int speed) : int
```

El servicio toma como primer argumento el tipo de articulación a la que va dirigida el movimiento. Este es un número entero entre 1 y 6.

Parámetro	Articulación
1	BASE
2	HOMBRO
3	CODO
4	MUÑECA Derecha
5	MUÑECA Izquierda
6	HERRAMIENTA

Tabla ¿? - Tipo de articulación

El segundo parámetro es un entero que indica a la articulación el número de pasos que debe desplazarse, y por último, el tercer argumento es un entero entre 1 y 220 que fija la velocidad con la que se va a efectuar el movimiento.
El valor de retorno es un entero que puede tomar el valor de 1 si el movimiento se ha realizado correctamente y el valor de 0 si habido algún problema durante la ejecución del servicio.

Para integrar el COTS en e modelo arquitectónico se implementa una interfaz que define el método público que se encargará de encolar las peticiones en la cola del aspecto cuando se reciba una petición de servicio de send. Además se define el delegado para este método.

```
public interface ICOTS
{
AsyncResult Send(int joint,int halfSteps,int speed);
}
public delegate AsyncResult SendDelegate(int joint,int halfSteps,
int speed);
```

La definición del aspecto de integración se define exactamente igual que los demás aspectos salvo por que hay que importar el servicio del COTS mediante el atributo DLLImport. Además, la definición del método privado se realiza la invocación al método importado y se devuelva el resultado de la llamada.

```
using System;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
namespace Robot {
[Serializable]
public class FCOTS : IntegrationAspect , ICOTS
{
[DllImport("TeachMover.dll", EntryPoint="send")] HERRAMIENTA

      private static extern int SendCOTS(int joint,int halfsteps,int
speed);
enum protocolStates
{
FCOTS,
SEND
}
      protocolStates estado;
{
estado = protocolStates.FCOTS;
public FCOTS() : base()
```

```
    PlayedRoleClass playedRoleICOTS = new
          PlayedRoleClass("COTS");
```

```
this.playedRoleList.Add(playedRoleICOTS);
estado = protocolStates.SEND;
}
public AsyncResult Send(int joint,int halfsteps,int speed)
{
return EnqueueService(sendDelegate,joint,halfSteps,speed);
}
private AsyncResult _Send(int joint,int halfsteps,int speed)
{
if (estado != protocolStates.SEND) {
throw new InvalidProtocolStateException();
}
// Comprobacion de las precondiciones asociadas
// NO APLICABLE
// Comprobación del estado anterior a la valuación
```

```
// NO APLICABLE
// Invocación servicio COTS
int response = SendCOTS(joint,halfSteps,speed);
      link.OutPorts["ICOTS","COTS"].AddRequest("Send",response);
      // Comprobación de la activación de Triggers
      // NO APLICABLE
estado = protocolStates.SEND;
return null;
}
}
```

Por último, el aspecto se agrega en un componente como si se tratará de un aspecto cualquiera, pudiendo definir weavings entre sus servicios, y publicando sus servicios a través de la interfaz que implementa el aspecto.

```
using System;
using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware; playedRoleICOTS.AddMethod("Send", 10);

      namespace Robot
      {
      [Serializable]
      public class CCOTS : ComponentBase
      {
public CCOTS(string name,MiddlewareSystem middlewareSystem) :
base (name,middlewareSystem)
{
AddAspect(new FCOTS);
InPorts.Add("ICOTSPort", "ICOTS",
      GetAspect(typeof(IntegrationAspect), "COTS");
      // Creación de PUERTOS de SALIDA
      OutPorts.Add("ICOTSPort", "ICOTS", "COTS");
}
      }
}
```

Capítulo 6

CONCLUSIONES

Contenido del capítulo

6 Conclusiones

En este capítulo se presentan las conclusiones y trabajos futuros del proyecto. En las conclusiones se pueden observar los resultados obtenidos en la herramienta, y seguidamente en los trabajos futuros se muestra las tareas que han quedado pendientes para realizar a partir del actual trabajo realizado.

6.1 Conclusiones

El principal objetivo de este proyecto era poder obtener una herramienta que proporcionada un soporte global e integrado a los arquitectos que quisieran desarrollar arquitecturas software complejas orientadas a aspectos basándose en el enfoque DDM..

Hasta la realización de este proyecto un analista que deseaba utilizar el modelo PRISMA para definir su arquitectura software, disponía de un lenguaje de modelado junto a una herramienta como es PRISMADSL, podía implementar manualmente dicho modelo en código ejecutable en el middleware PRISMANET, pero no podía realizarlo de forma automática.

Este proyecto ha permitido completar este desarrollo, con lo que actualmente se dispone de una aplicación que facilita en gran medida la definición de arquitecturas software orientadas a aspectos siguiendo el modelo PRISMA siguiendo MDD.

Para poder llegar a este resultado se ha tenido que realizar un compilador que a partir de un modelo dado generará código compilado para su posterior ejecución. Como en este

proyecto se ha definido una herramienta para modelar la configuración de la arquitectura, el usuario puede crear a través de la herramienta sus instancias y lanzarlas a ejecución. Estas tareas han sido realizadas en PRISMADSL y en PRISMANET, con la complejidad que ello ha conllevado.

Ahora es posible asegurar que el usuario de esta herramienta puede definir una arquitectura software sin necesidad de generar código de forma manual.

El principal problema a la hora de definir la herramienta ha residido en la complejidad de definir una compilación del modelo de forma que se puedan soportar el mayor número de posibilidades de especificación que ofrece del modelo PRISMA, ya que este es un modelo con una gran expresividad y flexibilidad.

La implementación de PRISMADSL ha sido en una versión Beta de DSL Tools, lo que ha supuesto deficiencias en la documentación del entorno de desarrollo. Esto ha, incrementando la carga de trabajo para desarrollar la herramienta. Esto ha venido dado por la utilización de una tecnología tan novedosa. Junto con el problema de la poca documentación, se tiene la complejidad que comprende la definición de las plantillas de transformación, que aunque permiten recorrer el modelo definido por el usuario para poder obtener el código que uno desea fácilmente, el entorno de implementación es poco intuitivo y el soporte de depuración es bastante limitado.

Otro punto crítico en el desarrollo de PRISMACase ha estado en el soporte del nivel de configuración, a través del cual se permite la creación de instancias a través de la herramienta de modelado. Desarrollar los elementos necesarios para automatizar esta tarea, simplificando el trabajo del usuario ha sido de gran complejidad. La razón de ello reside en la dificultad de generar desde el proceso de compilación el soporte para poder obtener posteriormente en la herramienta de configuración, un mecanismo que pudiera recorrer el modelo de instancias y generar la información necesaria para PRISMANET. En posteriores versiones de las DSL Tools, puede que esta tarea hubiera sido más sencilla, pero estas limitaciones no han impedido el desarrollo por completo de PRISMACase.

Por último, el poder ofrecer a los usuarios de la herramienta una consola con la cual pueden interactuar con sistemaza aplicación arquitectónica orientada a aspectos, aumenta aun más la funcionalidad de PRISMACase, hasta que se integre o desarrolle una aplicación para poder diseñar interfaces gráficas de usuario e integrarlas con el código generado.

6.2 Trabajos futuros

Los trabajos futuros son de gran diversidad, dado el carácter de PRISMACase.

Estas tareas futuras comienzan con dar soporte a todo el modelo PRISMA. Esta herramienta es la primera versión completa de la misma, por lo que no contempla en su totalidad las posibilidades que el modelo PRISMA ofrece. Por ejemplo, el soporte a atributos derivados, utilización de estructuras abstractas de datos, ampliación de tipos básicos, etc.

Todas estas tareas, requieren en primer lugar tener su correspondiente soporte en el Middleware PRISMANET, seguidamente se deberá añadir en el metamodelo de PRISMA su definición así como su representación gráfica y finalmente su generación automática de código.

Una vez completadas las necesidades del modelo PRISMA, se podrían aumentar las comprobaciones de restricciones sobre el modelado. Ello permitiría al usuario un uso de la herramienta más guiado o al menos más seguro, con lo que el tiempo de desarrollo sería inferior al actual.

Hasta el momento no se tiene un repositorio con los elementos definidos en un modelo, una nueva tarea sería proporcionar un repositorio de elementos arquitectónicos, aspectos, etc. Con ello se podría aumentar la reutilización de los modelos, ampliando la potencia de desarrollo para el usuario de la herramienta.

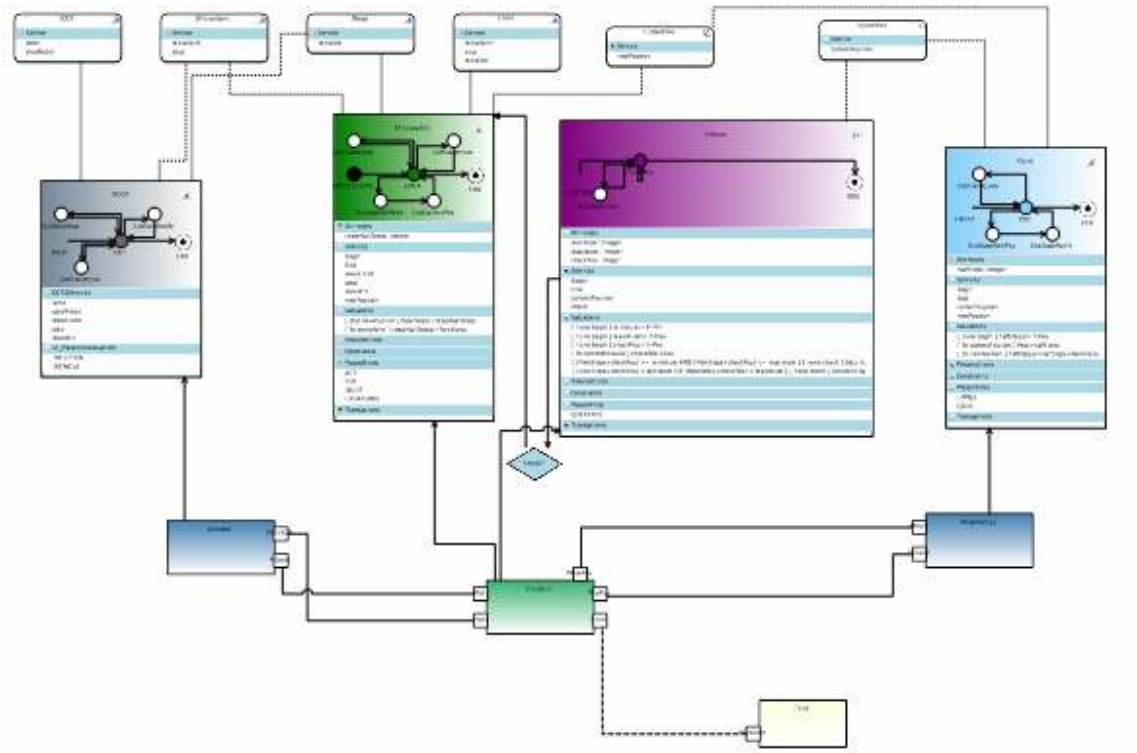
Otro punto importante a tener en cuenta es la versión de las DSL Tools utilizada en este proyecto, ya que se trata de una versión Beta. Actualmente ya se dispone de una versión estable, con lo que migrar el proyecto supondría disponer de una herramienta más estable y con mayor funcionalidad, así como capacidad de desarrollo de nuevas tareas.

Relacionado con el metamodelo de PRISMA, actualmente se esta desarrollando en el marco de una Tesis Doctoral, una ampliación de PRISMA permitiendo las posibilidades de movilidad y distribución de los elementos arquitectónicos del modelo. Estas nuevas funcionalidades, suponen cambios en el metamodelo, con sus correspondientes modificaciones a lo largo de todo el desarrollo de la actual herramienta. Tras su desarrollo se obtendría una herramienta con mayor expresividad para los usuarios de la misma.

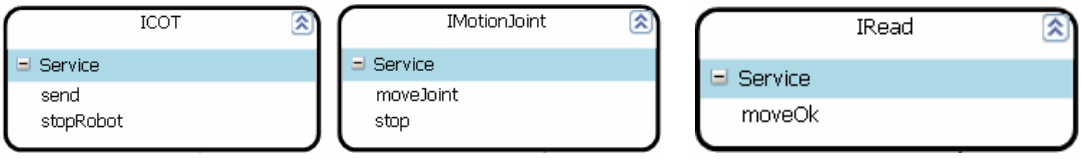
ANEXOS

ANEXO A Modelos arquitectura caso de estudio

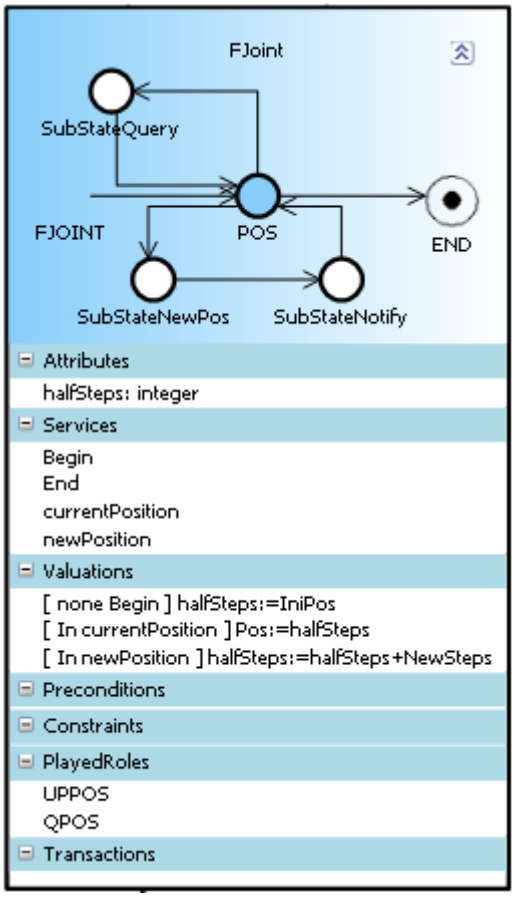
A.1 PrismaDSL

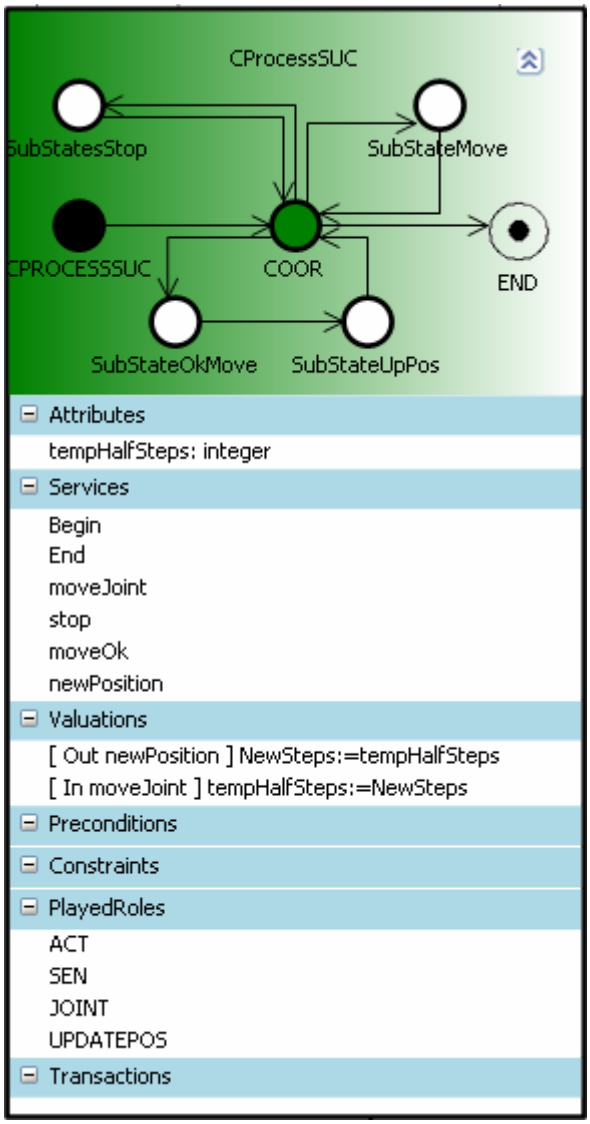


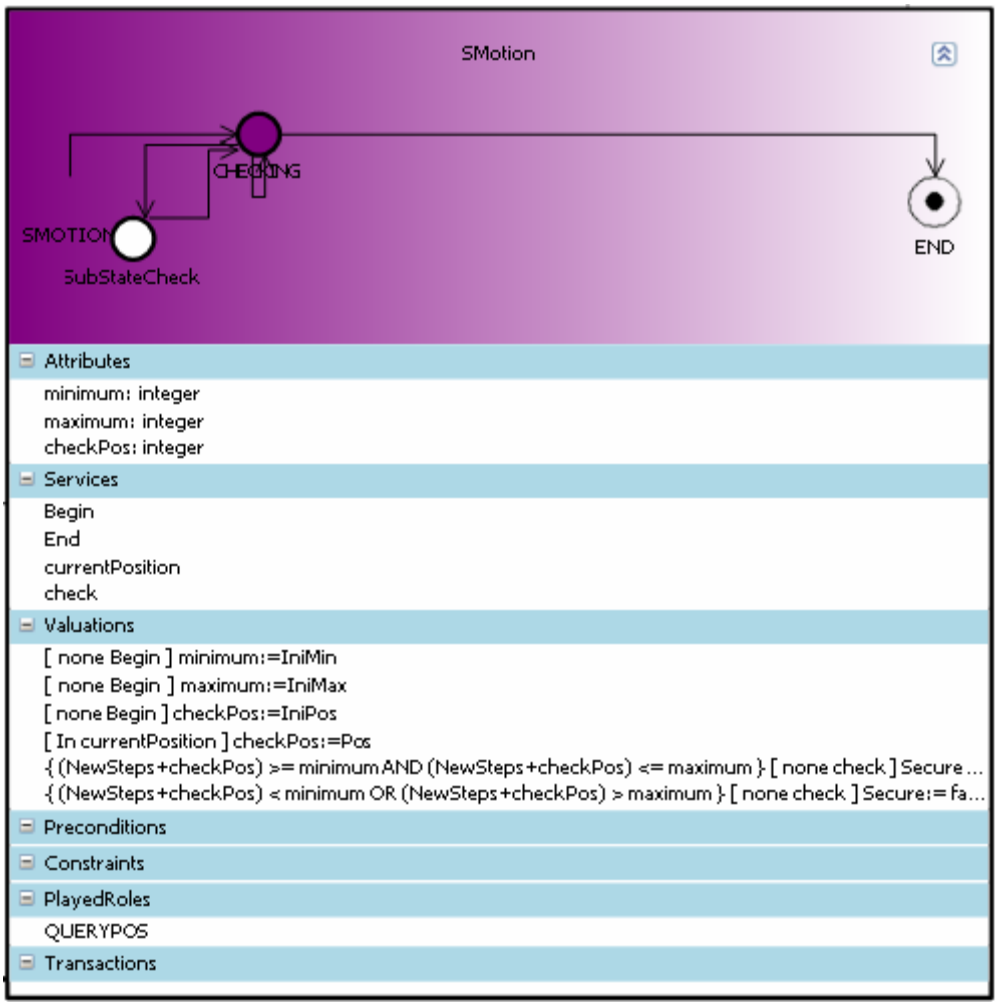
A.1.1 Interfaces



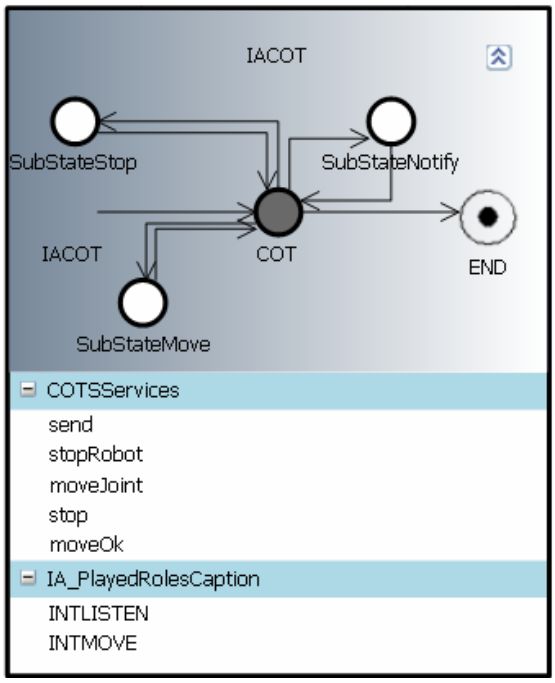
A.1.2 Aspectos



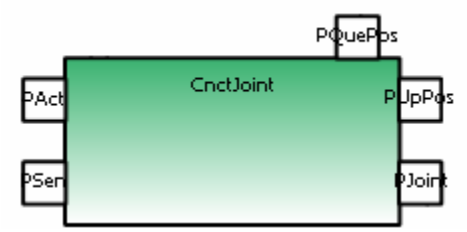


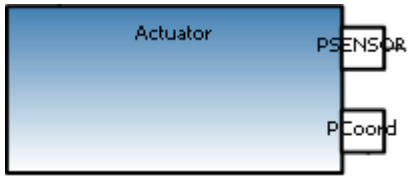

SMotion
CHECKING
SMOTION
SubStateCheck
END
Attributes
minimum: integer
maximum: integer
checkPos: integer
Services
Begin
End
currentPosition
check
Valuations
[none Begin] minimum:=IniMin
[none Begin] maximum:=IniMax
[none Begin] checkPos:=IniPos
[In currentPosition] checkPos:=Pos
{ (NewSteps+checkPos) >= minimum AND (NewSteps+checkPos) <= maximum } [none check] Secure ...
{ (NewSteps+checkPos) < minimum OR (NewSteps+checkPos) > maximum } [none check] Secure:= fa...
Preconditions
Constraints
PlayedRoles
QUERYPOS
Transactions

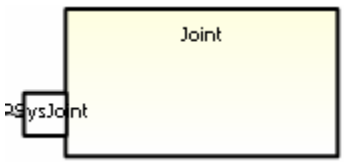
A.1.3 Aspecto Integración



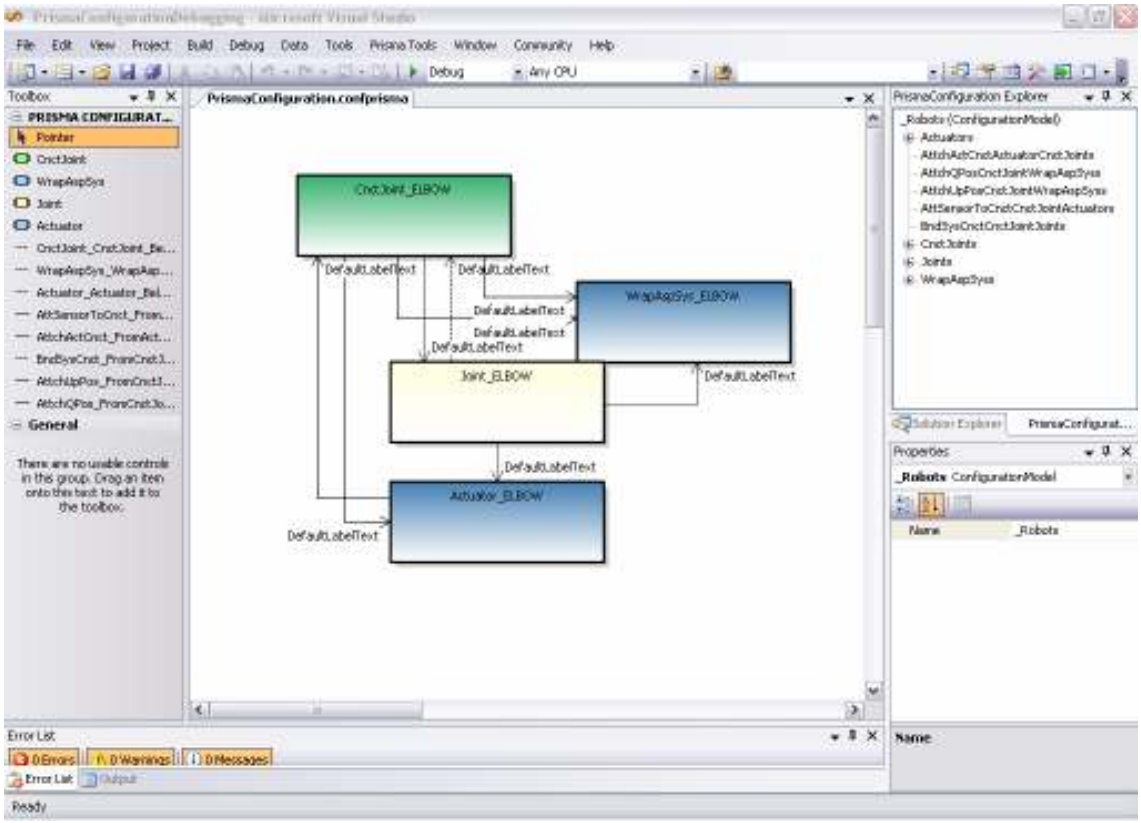
A.1.4 Elementos arquitectónicos



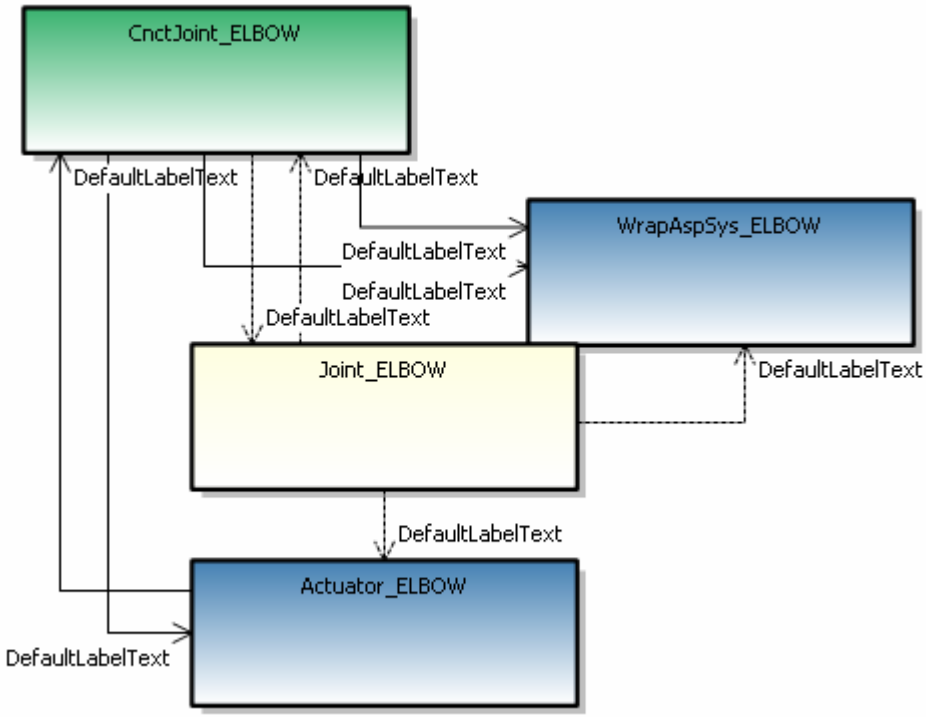




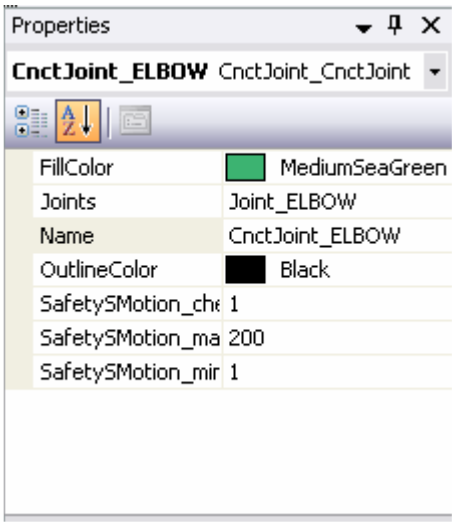
A.2 PrismaConfiguration



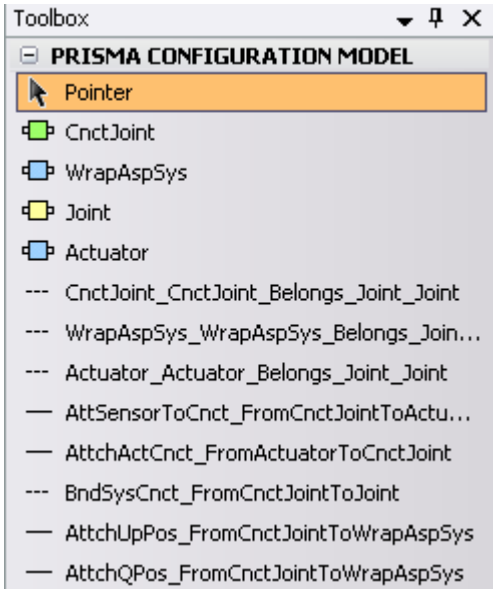
A.2.1 Modelo de configuración



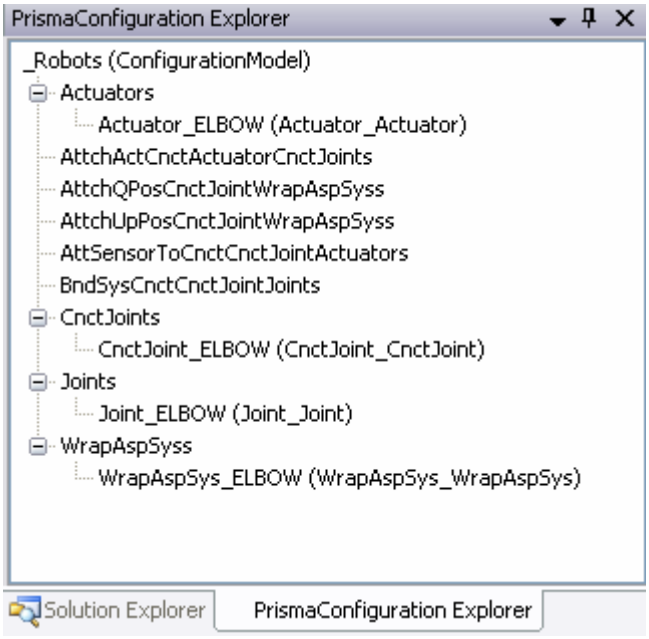
A.2.2 Ventana de propieades



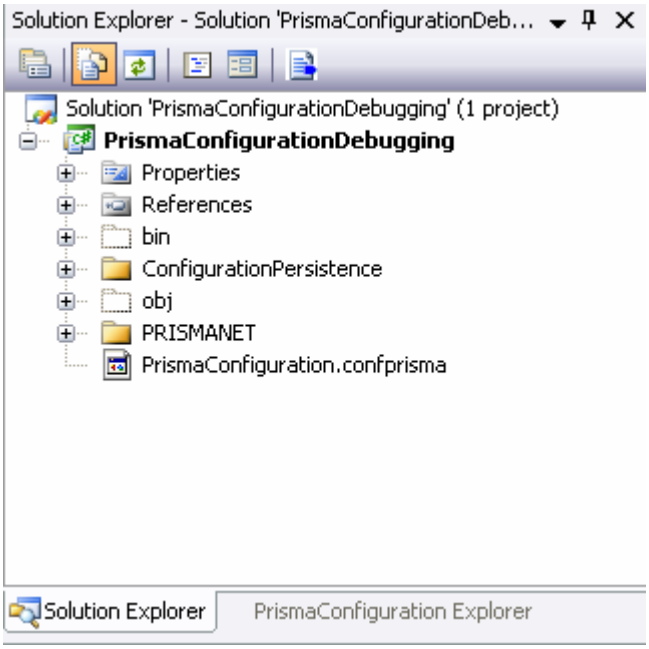
A.2.3 ToolBox



A.2.4 PrismaConfiguration Explorer



A.2.5 Solution Explorer

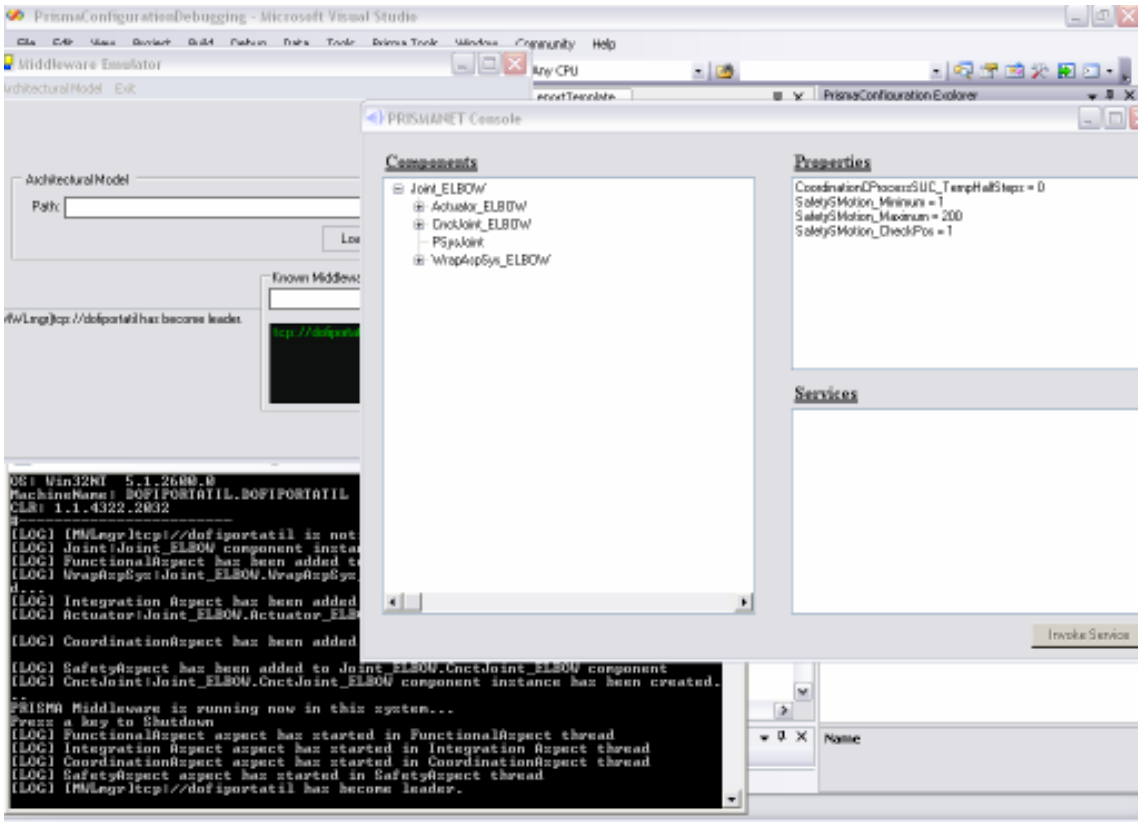


A.2.6 XML Configuración

```
<?xml version="1.0" encoding="utf-8"?>
<ConfigurationModel name = " Robots">
 <Systems>
       <System name = "Joint_ELBOW" type = "Joint">
       </System>
 </Systems>
 <Components>
       <Component name = "WrapAspSys_ELBOW" type = "WrapAspSys">
         <Properties>
          <Property name = "FunctionalFJoint_halfSteps" type = "integer" value = "1">
          </Property>
         </Properties>
         <SystemRef name = "Joint ELBOW">
         </SystemRef>
       </Component>
       <Component name = "Actuator_ELBOW" type = "Actuator">
         <SystemRef name = "Joint ELBOW">
         </SystemRef>
       </Component>
  </Components>
  <Connectors>
       <Connector name = "CnctJoint_ELBOW" type = "CnctJoint">
         <Properties>
               <Property name = "SafetySMotion minimum" type = "integer" value = "1">
               </Property>
               <Property name = "SafetySMotion_maximum" type = "integer" value = "200">
               </Property>
               <Property name = "SafetySMotion_checkPos" type = "integer" value = "1">
               </Property>
         </Properties>
         <SystemRef name = "Joint_ELBOW">
         </SystemRef>
       </Connector>
  </Connectors>
  <Attachments>
       <Attachment type = "AttSensorToCnctCnctJointActuator">
               <source name = "Actuator_ELBOW" port = "PSENSOR">
               </source>
               <target name = "CnctJoint_ELBOW" port = "PSen">
               </target>
               <SystemRef name = "Joint ELBOW">
               </SystemRef>
       </Attachment>
       <Attachment type = "AttchActCnctActuatorCnctJoint">
               <source name = "CnctJoint_ELBOW" port = "PAct">
               </source>
               <target name = "Actuator ELBOW" port = "PCoord">
               </target>
               <SystemRef name = "Joint_ELBOW">
               </SystemRef>
       </Attachment>
       <Attachment type = "AttchUpPosCnctJointWrapAspSys">
               <source name = "WrapAspSys ELBOW" port = "PPosUp">
               </source>
               <target name = "CnctJoint_ELBOW" port = "PUpPos">
               </target>
               <SystemRef name = "Joint ELBOW">
               </SystemRef>
       </Attachment>
       <Attachment type = "AttchQPosCnctJointWrapAspSys">
               <source name = "WrapAspSys_ELBOW" port = "PPosQ">
               </source>
               <target name = "CnctJoint ELBOW" port = "PQuePos">
               </target>
               <SystemRef name = "Joint_ELBOW">
               </SystemRef>
       </Attachment>
  </Attachments>
  <Bindings>
       <Binding type = "BndSysCnctCnctJointJoint">
               <source name = "Joint_ELBOW" port = "PSysJoint">
```

```
                </source>
                <target name = "CnctJoint ELBOW" port = "PJoint">
                </target>
                <SystemRef name = "Joint ELBOW">
                </SystemRef>
        </Binding>
  </Bindings>
</ConfigurationModel>
```

A.2.7 PRISMANET Ejecución modelo



ANEXO B Código generado del caso de estudio

B.1 Interfaces

```
using System;
using PRISMA;
namespace RobotJoint
{
	public interface IMotionJoint
	{
		AsyncResult moveJoint(int NewSteps, int Speed);
		AsyncResult stop();
	}


	public interface IRead
	{
		AsyncResult moveOk();
	}


	public interface IUpdatePos
	{
		AsyncResult newPosition(int NewSteps);
	}


	public interface IQueryPos
	{
		AsyncResult currentPosition(ref int Pos);
	}


	public interface IJoint
	{
		AsyncResult moveJoint(int NewSteps, int Speed);
		AsyncResult stop();
		AsyncResult moveOk();
	}


	public interface ICOT
	{
		AsyncResult send(int Speed, int HalfSteps, string Joint);
		AsyncResult stopRobot();
	}


	public delegate AsyncResult currentPositionDelegate(ref int Pos);
	public delegate AsyncResult moveJointDelegate(int NewSteps, int Speed);
	public delegate AsyncResult moveOkDelegate();
	public delegate AsyncResult newPositionDelegate(int NewSteps);
	public delegate AsyncResult sendDelegate(int Speed, int HalfSteps, string Joint);
	public delegate AsyncResult stopDelegate();
	public delegate AsyncResult stopRobotDelegate();

}
```

B.2 Componentes

```
using System;
using System.Reflection;

using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace RobotJoint
{
  [Serializable]
  public class CnctJoint : ComponentBase , IConnector
  {
        public CnctJoint(string name, int IniMin, int IniMax, int IniPos ) : base(name)
        {
                AddAspect(new CProcessSUC());
                AddAspect(new SMotion(IniMin, IniMax, IniPos));

                WeavingType weavingType=

WeavingType.BEFOREIF_VALUE("Secure",WeavingType.OperatorType.Equality,true);
                AddWeaving(GetAspect(typeof(SafetyAspect)), "check","NewSteps,Secure",
                           weavingType, GetAspect(typeof(CoordinationAspect)),
"moveJoint",
                          "NewSteps,Speed");


                InPorts.Add("PSen", "IRead", "SEN");
                OutPorts.Add("PSen", "IRead", "SEN");

                InPorts.Add("PAct", "IMotionJoint", "ACT");
                OutPorts.Add("PAct", "IMotionJoint", "ACT");

                InPorts.Add("PJoint", "IJoint", "JOINT");
                OutPorts.Add("PJoint", "IJoint", "JOINT");

                InPorts.Add("PUpPos", "IUpdatePos", "UPDATEPOS");
                OutPorts.Add("PUpPos", "IUpdatePos", "UPDATEPOS");

                InPorts.Add("PQuePos", "IQueryPos", "QUERYPOS");
                OutPorts.Add("PQuePos", "IQueryPos", "QUERYPOS");

        }
  }
  [Serializable]
  public class WrapAspSys : ComponentBase
  {
        public WrapAspSys(string name, int IniPos ) : base(name)
        {
                AddAspect(new FJoint(IniPos));


                InPorts.Add("PPosUp", "IUpdatePos", "UPPOS");
                OutPorts.Add("PPosUp", "IUpdatePos", "UPPOS");

                InPorts.Add("PPosQ", "IQueryPos", "QPOS");
                OutPorts.Add("PPosQ", "IQueryPos", "QPOS");

        }
  }
  [Serializable]
  public class Joint : SystemBase
  {

        public Joint(string name) : base(name)
        {

                InPorts.Add("PSysJoint", "IJoint", "JOINT");
                OutPorts.Add("PSysJoint", "IJoint", "JOINT");

        }
```

```
  }
  [Serializable]
  public class Actuator : ComponentBase
  {
        public Actuator(string name ) : base(name)
        {
                AddAspect(new IACOT(name.Split('_')[1].Split('.')[0]));


                InPorts.Add("PCoord", "IMotionJoint", "INTMOVE");
                OutPorts.Add("PCoord", "IMotionJoint", "INTMOVE");

                InPorts.Add("PSENSOR", "IRead", "INTLISTEN");
                OutPorts.Add("PSENSOR", "IRead", "INTLISTEN");

        }
  }
}
```

B.3 Aspectos

```
using System;
using PRISMA;
using PRISMA.Aspects.Types;
using PRISMA.Exceptions;
using System.Collections;

namespace RobotJoint
{
  [Serializable]
  public class CProcessSUC : CoordinationAspect, IMotionJoint, IRead, IUpdatePos, IJoint
  {
        int tempHalfSteps;
        public int TempHalfSteps
        {
                get { return tempHalfSteps; }
        }

        enum protocolStates
        {
        CPROCESSSUC, COOR, END, SubStatesStop, SubStateMove, SubStateOkMove,
SubStateUpPos
        }
        protocolStates state;
        private protocolStates State{
                get { return state;}
                set { state=value;
                      this.StateName=state.ToString();
                }
        }

        public CProcessSUC() : base("CProcessSUC")
        {

                State = protocolStates.CPROCESSSUC;

                PlayedRoleClass ACT = new PlayedRoleClass("ACT");
                ACT.AddMethod("moveJoint", false);
                ACT.AddMethod("stop", true);
                this.playedRoleList.Add(ACT);


                PlayedRoleClass SEN = new PlayedRoleClass("SEN");
                SEN.AddMethod("moveOk", true);
                this.playedRoleList.Add(SEN);


                PlayedRoleClass JOINT = new PlayedRoleClass("JOINT");
                JOINT.AddMethod("moveJoint", true);
                JOINT.AddMethod("stop", true);
```

```
                JOINT.AddMethod("moveOk", false);
                this.playedRoleList.Add(JOINT);


                PlayedRoleClass UPDATEPOS = new PlayedRoleClass("UPDATEPOS");
                UPDATEPOS.AddMethod("newPosition", false);
                this.playedRoleList.Add(UPDATEPOS);

                this.stateList=new ArrayList();
                this.stateList.Add("CPROCESSSUC");
                this.stateList.Add("COOR");
                this.stateList.Add("END");
                this.stateList.Add("SubStatesStop");
                this.stateList.Add("SubStateMove");
                this.stateList.Add("SubStateOkMove");
                this.stateList.Add("SubStateUpPos");
                AddPriorityService(protocolStates.SubStatesStop.ToString(),
                                   ACT.PlayedRoleName, "stop", 0);
                AddPriorityService(protocolStates.SubStateMove.ToString(),
                                   ACT.PlayedRoleName, "moveJoint", 1);
                AddPriorityService(protocolStates.COOR.ToString(), SEN.PlayedRoleName,
                                   "moveOk", 1);
                AddPriorityService(protocolStates.COOR.ToString(), JOINT.PlayedRoleName,
                                   "stop", 0);
                AddPriorityService(protocolStates.COOR.ToString(), JOINT.PlayedRoleName,
                                   "moveJoint", 1);
                AddPriorityService(protocolStates.SubStateUpPos.ToString(),
                                   JOINT.PlayedRoleName, "moveOk", 1);
                AddPriorityService(protocolStates.SubStateOkMove.ToString(),
                                  UPDATEPOS.PlayedRoleName, "newPosition", 1);
                State = protocolStates.COOR;
        }
        public AsyncResult moveJoint(int NewSteps, int Speed)
        {
          // Modo In
          if(ServiceIn)
          {
                if( state != protocolStates.COOR
                    && state != protocolStates.SubStateMove
                ) throw new InvalidProtocolStateException("CProcessSUC","moveJoint");

                tempHalfSteps=NewSteps;
                if (state == protocolStates.COOR)
                {
                   state = protocolStates.SubStateMove;

                   InvokeOutService("IMotionJoint","ACT","moveJoint",
                           this.aspectStateCareTaker.ActiveTransaction,NewSteps, Speed);
                   state=protocolStates.COOR;
                }
                return null;

           }
           // Modo Out
           else
           {
                return CallOutService(this.interfaceName_ServiceOut,
                             this.playedRoleName_ServiceOut,"moveJoint",
                             this.aspectStateCareTaker.ActiveTransaction,NewSteps,
Speed);
           }

        }
        public AsyncResult stop()
        {
          // Modo In
          if(ServiceIn)
          {
                if( state != protocolStates.COOR
                    && state != protocolStates.SubStatesStop
                ) throw new InvalidProtocolStateException("CProcessSUC","stop");

                if (state == protocolStates.COOR)
                {
                        state = protocolStates.SubStatesStop;

                        stop();
```

```
                        state=protocolStates.COOR;
                }
                        return null;

            }
            // Modo Out
            else
            {
                return CallOutService(this.interfaceName ServiceOut,
                            this.playedRoleName ServiceOut,"stop",
                            this.aspectStateCareTaker.ActiveTransaction,null);
            }

        }
        public AsyncResult moveOk()
        {
          // Modo In
          if(ServiceIn)
          {
                if( state != protocolStates.COOR
                    && state != protocolStates.SubStateUpPos
                ) throw new InvalidProtocolStateException("CProcessSUC","moveOk");

                if (state == protocolStates.COOR)
                {
                        state = protocolStates.SubStateOkMove;
                        int NewSteps=0;

                        InvokeOutService("IUpdatePos","UPDATEPOS","newPosition",
                        this.aspectStateCareTaker.ActiveTransaction,NewSteps);
                        state=protocolStates.SubStateUpPos;

                        InvokeOutService("IJoint","JOINT","moveOk",
                           this.aspectStateCareTaker.ActiveTransaction,null);
                        state=protocolStates.COOR;
                }
                        return null;

           }
           // Modo Out
           else
           {
                return CallOutService(this.interfaceName ServiceOut,
                        this.playedRoleName ServiceOut,"moveOk",
                        this.aspectStateCareTaker.ActiveTransaction,null);
           }

        }
        public AsyncResult newPosition(int NewSteps)
        {
          // Modo In
          if(ServiceIn)
          {
                throw new Exception("This method doesn't have Service mode In");

          }
          // Modo Out
          else
          {
                NewSteps=tempHalfSteps;
                return CallOutService(this.interfaceName ServiceOut,
                        this.playedRoleName ServiceOut,"newPosition",
                        this.aspectStateCareTaker.ActiveTransaction,NewSteps);
          }

        }

}
[Serializable]
public class SMotion : SafetyAspect, IQueryPos
{
        int minimum;
        public int Minimum
        {
                get { return minimum; }
        }
```

```
        int maximum;
        public int Maximum
        {
                get { return maximum; }
        }

        int checkPos;
        public int CheckPos
        {
                get { return checkPos; }
        }

        enum protocolStates
        {
                SMOTION, CHECKING, END, SubStateCheck
        }
        protocolStates state;
        private protocolStates State{
                get { return state;}
                set { state=value;
                        this.StateName=state.ToString();
                }
        }

        public SMotion(int IniMin, int IniMax, int IniPos) : base("SMotion")
        {

                State = protocolStates.SMOTION;

                minimum=IniMin;
                maximum=IniMax;
                checkPos=IniPos;
                PlayedRoleClass QUERYPOS = new PlayedRoleClass("QUERYPOS");
                QUERYPOS.AddMethod("currentPosition", true);
                this.playedRoleList.Add(QUERYPOS);


                this.stateList=new ArrayList();
                this.stateList.Add("SMOTION");
                this.stateList.Add("CHECKING");
                this.stateList.Add("END");
                this.stateList.Add("SubStateCheck");
                AddPriorityService(protocolStates.SubStateCheck.ToString(),
                                   QUERYPOS.PlayedRoleName, "currentPosition", 1);
                AddPriorityService(protocolStates.CHECKING.ToString(),
                                   QUERYPOS.PlayedRoleName, "currentPosition", 1);
                State = protocolStates.CHECKING;
        }
        public AsyncResult currentPosition(ref int Pos)
        {
          // Modo In
          if(ServiceIn)
          {
                if( state != protocolStates.SubStateCheck
                          && state != protocolStates.CHECKING
                ) throw new InvalidProtocolStateException("SMotion","currentPosition");

                checkPos=Pos;
                if (state == protocolStates.CHECKING )
                        state = protocolStates.CHECKING;
                return null;

           }
           // Modo Out
           else
           {
                return CallOutService(this.interfaceName ServiceOut,
                          this.playedRoleName ServiceOut,"currentPosition",
                          this.aspectStateCareTaker.ActiveTransaction,Pos);
           }

        }
        public delegate AsyncResult checkDelegate(int NewSteps, ref bool Secure);
        public AsyncResult check(int NewSteps, ref bool Secure)
        {
          if( state != protocolStates.CHECKING
          ) throw new InvalidProtocolStateException("SMotion","check");
```

```
        if ((NewSteps+checkPos) >= minimum && (NewSteps+checkPos) <= maximum)
        {
              Secure=true;
        }

        if ((NewSteps+checkPos) < minimum || (NewSteps+checkPos) > maximum)
        {
              Secure= false;
        }

        if (state == protocolStates.CHECKING)
        {
              state = protocolStates.SubStateCheck;
              int Pos=0;

              InvokeOutService("IQueryPos","QUERYPOS","currentPosition",
                               this.aspectStateCareTaker.ActiveTransaction,Pos);
              state=protocolStates.CHECKING;
        }
              return null;

      }

}
[Serializable]
public class FJoint : FunctionalAspect, IQueryPos, IUpdatePos
{
      int halfSteps;
      public int HalfSteps
      {
              get { return halfSteps; }
      }

      enum protocolStates
      {
              FJOINT, POS, END, SubStateNewPos, SubStateNotify, SubStateQuery
      }
      protocolStates state;
      private protocolStates State{
              get { return state;}
              set { state=value;
                        this.StateName=state.ToString();
              }
      }

      public FJoint(int IniPos) : base("FJoint")
      {

              State = protocolStates.FJOINT;

              halfSteps=IniPos;
              PlayedRoleClass UPPOS = new PlayedRoleClass("UPPOS");
              UPPOS.AddMethod("newPosition", true);
              this.playedRoleList.Add(UPPOS);


              PlayedRoleClass QPOS = new PlayedRoleClass("QPOS");
              QPOS.AddMethod("currentPosition", true);
              this.playedRoleList.Add(QPOS);


              this.stateList=new ArrayList();
              this.stateList.Add("FJOINT");
              this.stateList.Add("POS");
              this.stateList.Add("END");
              this.stateList.Add("SubStateNewPos");
              this.stateList.Add("SubStateNotify");
              this.stateList.Add("SubStateQuery");
              AddPriorityService(protocolStates.POS.ToString(), UPPOS.PlayedRoleName,
                                 "newPosition", 1);
              AddPriorityService(protocolStates.SubStateNewPos.ToString(),
                                 QPOS.PlayedRoleName, "currentPosition", 1);
              AddPriorityService(protocolStates.SubStateNotify.ToString(),
                                 QPOS.PlayedRoleName, "currentPosition", 1);
              AddPriorityService(protocolStates.POS.ToString(), QPOS.PlayedRoleName,
                                 "currentPosition", 1);
```

```
                AddPriorityService(protocolStates.SubStateQuery.ToString(),
                                   QPOS.PlayedRoleName, "currentPosition", 1);
                State = protocolStates.POS;
        }
        public AsyncResult currentPosition(ref int Pos)
        {
          // Modo In
          if(ServiceIn)
          {
                if( state != protocolStates.SubStateNewPos
                    && state != protocolStates.SubStateNotify
                    && state != protocolStates.POS
                    && state != protocolStates.SubStateQuery
                ) throw new InvalidProtocolStateException("FJoint","currentPosition");

                Pos=halfSteps;
                if (state == protocolStates.POS)
                {
                    state = protocolStates.SubStateQuery;

                    InvokeOutService("IQueryPos","QPOS","currentPosition",
                                     this.aspectStateCareTaker.ActiveTransaction,Pos);
                    state=protocolStates.POS;
                }
                return null;

           }
           // Modo Out
           else
           {
                return CallOutService(this.interfaceName ServiceOut,
                             this.playedRoleName ServiceOut,"currentPosition",
                             this.aspectStateCareTaker.ActiveTransaction,Pos);
           }

        }
        public AsyncResult newPosition(int NewSteps)
        {
          // Modo In
          if(ServiceIn)
          {
                if( state != protocolStates.POS
                ) throw new InvalidProtocolStateException("FJoint","newPosition");

                halfSteps=halfSteps+NewSteps;
                if (state == protocolStates.POS)
                {
                        state = protocolStates.SubStateNewPos;
                        int Pos=0;

                        currentPosition(ref Pos);
                        state=protocolStates.SubStateNotify;

                        InvokeOutService("IQueryPos","QPOS","currentPosition",
                                  this.aspectStateCareTaker.ActiveTransaction,Pos);
                        state=protocolStates.POS;
                }
                        return null;

           }
           // Modo Out
           else
           {
                throw new Exception("This method doesn't have Service mode Out");
           }

        }

   }
}
```

B.4 Aspecto integración: COTS

```
using TeachMover;
using System.Collections;

using System;
using PRISMA;
using PRISMA.Aspects.Types;
using PRISMA.Exceptions;
namespace RobotJoint
{

        public class IACOT : IntegrationAspect , IMotionJoint, ICOT, IRead
        {

                //TODO: Añadir estados del protocolo.
                enum protocolStates
                {
                                SubStateNotify,IACOT, COT, END


                }
                protocolStates state;
                private protocolStates State{
                        get { return state;}
                        set { state=value;
                                  this.StateName=state.ToString();
                                 }
                }

                public IACOT(string name) : base(name)
                {

                        PlayedRoleClass INTLISTEN = new PlayedRoleClass("INTLISTEN");
                        INTLISTEN.AddMethod("moveOk", true);
                        this.playedRoleList.Add(INTLISTEN);
                        this.stateList=new ArrayList();

                        PlayedRoleClass INTMOVE = new PlayedRoleClass("INTMOVE");
                        INTMOVE.AddMethod("moveJoint", true);
                        INTMOVE.AddMethod("stop", true);
                        this.playedRoleList.Add(INTMOVE);
                        this.stateList=new ArrayList();

                        AddPriorityService(protocolStates.SubStateNotify.ToString(),
                           INTLISTEN.PlayedRoleName, "moveOk", 1);
                        AddPriorityService(protocolStates.COT.ToString(),
                           INTMOVE.PlayedRoleName, "stop", 0);
                        AddPriorityService(protocolStates.COT.ToString(),
                           INTMOVE.PlayedRoleName, "moveJoint", 1);
                        AddPriorityService(protocolStates.COT.ToString(),
                           INTMOVE.PlayedRoleName, "moveJoint", 1);
                        State = protocolStates.COT;

                }
                public AsyncResult moveJoint (int NewSteps, int Speed)
                {
                        send(NewSteps, Speed,this.aspectName);
                        CallOutService("IRead","INTLISTEN","moveOk",
                            this.aspectStateCareTaker.ActiveTransaction,null);
                        return null;
                }
                public AsyncResult stop ()
                {
                        stopRobot();
                        return null;
                }

                public AsyncResult send (int Speed, int HalfSteps, string Joint)
                {

                        TeachMover.TeachMover.send(Speed, HalfSteps, Joint);
                        return null;
                }
```

```
                public AsyncResult stopRobot ()
                {

                        TeachMover.TeachMover.stopRobot();
                        return null;
                }

                public AsyncResult moveOk ()
                {
                        return null;
                }

        }
}
```

Bibliografia

Desarrollo Dirigido por Modelos

Greenfield, J., et al. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. Wiley, 2004.

Object Management Group. Model Driven Architecture Guide v1.0.1, 2003. http://www.omg.org/

Arquitecturas Software

Perry, D., Wolf A., *Foundations for the Study of Software Architecture*. ACM Software Engineering Notes, Vol. 17, No. 4, pp. 40-52, October 1992.

DSOA

Kiczales G., Hilsdale E., Huguin J., Kersten M., Palm J., Griswold W.G., *An Overview of AspectJ*. The 15th European Conference on Object-Oriented Programming, Lecture Notes in Computer Science (LNCS), Springer-Verlag, Vol.2072, Budapest, Hungary, June 18-22, 2001.

Kizcales G., Lamping J., Mendhekar A., Maeda C., *Aspect-Oriented Programming*. The 11th European Conference on Object-Oriented Programming *(ECOOP)*, Lecture Notes in Computer Science (LNCS), Springer-Verlag, Vol. 1241, Jyväskylä, Finland, June 9-13, 1997.

PRISMA

Enfoque

Pérez, J. PRISMA: Aspect-Oriented Software Architectures. Tesis Doctoral, Universidad Politécnica de Valencia, 2006.

Lenguaje
Pérez, J., et al., Designing Software Architectures with an Aspect-Oriented Architecture Description Language. 9th Int. Symp. on Component-Based Software Engineering (CBSE), LNCS vol. 4063, Springer, 2006.

Modelo
Pérez, J., et al. Dynamic Evolution in Aspect-Oriented Architectural Models. 2nd Eur. Wks. on Software Architecture (EWSA), LNCS 3527 Pisa, Italy, 2005.

PRISMANET
Pérez, J., et al. Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology. 3rd Int. Conf. on .NET Technologies. Pilsen, Czech Republic 2005.
PRISMA web page, http://prisma.dsic.upv.es

Cabedo Archer, Rafael. Aplicación del lenguaje de descripción de arquitecturas PRISMA a un sistema robótico de ámbito industrial.

Costa Soria, Cristobal. Estudio e implementación de un modelo de arquitecturas orientado a aspectos y basado en componentes sobre tecnología .net. Proyecto Final de Carrera. Febrero 2005

Millán Belda, Carlos. Desarrollo de aplicaciones distribuidas y móviles desde un modelo arquitectónico orientado a aspectos. Proyecto Final de Carrera. Septiembre 2006