UNIVERSIDAD POLITÉCNICA DE VALENCIA

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

APLICACIÓN DEL LENGUAJE DE DESCRIPCIÓN DE ARQUITECTURAS PRISMA A UN SISTEMA ROBÓTICO DE ÁMBITO INDUSTRIAL

PROYECTO FINAL DE CARRERA

RAFAEL CABEDO ARCHER

Agradecimientos

A mi familia, y especialmente a mi padre
por sustentar todas mis decisiones a lo largo de mi vida.

A Maika,
por su cariño incondicional y su comprensión.

A mis directores de proyecto,
Jennifer Pérez e Isidro Ramos
por encaminarme en mi trayectoria profesional.

A José Angel Carsí,
por confiarme el desarrollo del proyecto.

A mi compañero de batallas,
Cristobal Costa,
por sus consejos y enseñanzas.

A Nour Ali,
por todas sus recomendaciones.

Al resto de mis compañeros,
José Antonio, Javi, Elena, Artur, Manolo, Isabel, Carlos, Nelly, Gonzalo
José Manuel, Pepe, Abel, José Miguel, Raquel, Clemente, Leopoldo,
porque juntos formamos un todo.

A mis más especiales Amigos.

"La libertad, Sancho, es uno de los más preciosos dones que a los hombres dieron los cielos; con ella no pueden igualarse los tesoros que encierra la tierra ni el mar encubre; por la libertad, así como por la honra, se puede y debe aventurar la vida, y, por el contrario, el cautiverio es el mayor mal que puede venir a los hombres."

Don Quijote de la Mancha, capítulo LVIII de la 2ª parte.
Miguel de Cervantes y Saavedra.

CONTENIDOS

LISTA DE FIGURAS

LISTA DE TABLAS

Capítulo 1

PLANTEAMIENTO Y OBJETIVOS

CONTENIDOS

PLANTEAMIENTO Y OBJETIVOS

1.1 Introducción

Actualmente, los sistemas *software* son cada vez más difíciles de desarrollar debido a sus complejas estructuras y a la dinámica y competitividad del mercado que implica una evolución constante de los requisitos del sistema. La ingeniería del *software* debe ser capaz de dar respuesta a esta tendencia con la propuesta de nuevas técnicas de desarrollo capaces de tener en cuenta las necesidades específicas de este tipo de sistemas. En este sentido han surgido recientemente dos disciplinas dentro de la ingeniería del *software*: el Desarrollo Basado en Componentes (DSBC) [CBSD] y el Desarrollo Orientado a Aspectos (DSOA) [AOSD]. Las tendencias actuales han conseguido que hoy por hoy los componentes y los aspectos sean la clave para la reutilización de un mayor número de líneas de código. Mediante la reutilización y la encapsulación de propiedades en entidades separadas, es decir en aspectos, se consigue que el proceso de producción de sistemas *software* minimice el tiempo y el coste de desarrollo y mantenimiento. El proceso tradicional de desarrollo *software* no profundiza en la reutilización es decir, no especifica cómo y cuándo se debe hacer uso de la reutilización para el diseño de sistemas complejos. Por ello, el DSBC propone un cambio en el proceso de desarrollo, en el que se incluye una etapa de identificación de elementos potencialmente reutilizables, y de puntos del sistema donde se puedan incorporar componentes ya desarrollados. Por otro lado, el DSOA propugna la utilización del concepto de aspecto en todas las fases del ciclo de vida del desarrollo de *software*. Las técnicas orientadas a aspectos extienden técnicas tradicionales como la orientación a objetos, permitiendo a los desarrolladores de *software* encapsular en módulos separados las propiedades comunes a varios componentes de un sistema. Esta técnica exalta la identificación, separación y modularización de los distintos aspectos que intervienen en un sistema por ejemplo coordinación, distribución, persistencia, seguridad... para componerlos (tejerlos, en su terminología original) con posterioridad para construir el sistema final.

PRISMA (Plataforma Oasis para Modelos Arquitectónicos) es un marco de trabajo (*framework*) que introduce un modelo y un lenguaje para especificar modelos arquitectónicos. Se entiende por modelo arquitectónico, un modelo de arquitecturas *software* compuesto por la estructura de los componentes de un programa o sistema, sus interrelaciones, y los principios y reglas que gobiernan su diseño y evolución a lo largo del tiempo [Gar95]. El modelo PRISMA unifica las dos nuevas tendencias en el desarrollo de sistemas *software*: el diseño basado en componentes y el desarrollo orientado a aspectos, y permite la especificación de sistemas complejos, distribuidos y altamente dinámicos, a través de un lenguaje de descripción de arquitecturas. El lenguaje en el que está basado PRISMA es OASIS [Let98]. OASIS es un lenguaje formal para definir modelos conceptuales de sistemas de información orientados a objetos,

que permite validar y generar las aplicaciones automáticamente a partir de la información capturada en los modelos. PRISMA se basa en OASIS, garantizando la compilación de sus modelos, y lo extiende, para poder definir formalmente la semántica de los modelos de arquitectura, ya que OASIS únicamente es capaz de especificar sistemas de información orientados a objetos.

Los sistemas robóticos difieren de otros tipos de sistemas de información en muchos aspectos. Uno de los principales es la necesidad de alcanzar objetivos complejos y de alto nivel. Generalmente, estos sistemas deben desarrollar su actividad en entornos no estructurados, complejos y dinámicos (ambiente de trabajo de los robots tele-operados), donde el sistema debe asegurar su propio comportamiento dinámico, incluso reaccionando ante cambios inesperados en dicho entorno. Muchos de estos robots incorporan sistemas empotrados en tiempo-real, son sistemas críticos, y en bastantes ocasiones tienen requisitos de seguridad elevados. Como sistemas críticos, en el caso de no cumplir sus tiempos de respuesta, el funcionamiento del sistema no será correcto, y esto, dependiendo de la aplicación para la que esté diseñado, puede tener consecuencias graves para el entorno en el que el robot desarrolla su labor e incluso para la seguridad de las personas. Por todo ello, la seguridad del *software*, su correcta integración con el *hardware*, su rendimiento, tolerancia a fallos, etc., tienen una importancia relevante. Normalmente, estos sistemas son demasiado complejos para ser desarrollados utilizando técnicas convencionales de programación. Para manejar esta complejidad, se hace necesario plantear modelos de desarrollo rigurosos y arquitecturas bien definidas que permitan ser implementadas en estos sistemas complejos. También se busca que dichas arquitecturas se puedan reutilizar en otros sistemas, y que sean fácilmente modificables atendiendo a nuevos objetivos o requisitos, pudiendo además ser analizadas para comprobar que cumplen dichos requisitos.

1.2 Motivación

El presente proyecto forma parte de un conjunto de proyectos enmarcados en un proyecto global denominado Proyecto PRISMA. Éste proyecto tiene como misión desarrollar un compilador para el modelo PRISMA que permita traducir las especificaciones de sistemas complejos en aplicaciones ejecutables sobre la plataforma *.Net*. El proyecto PRISMA sigue la aproximación del Desarrollo de *Software* Dirigido por Modelos para el desarrollo de sistemas *software* basada en la separación entre la especificación de la funcionalidad esencial del sistema y la implementación de dicha funcionalidad, usando plataformas de implementación específicas.

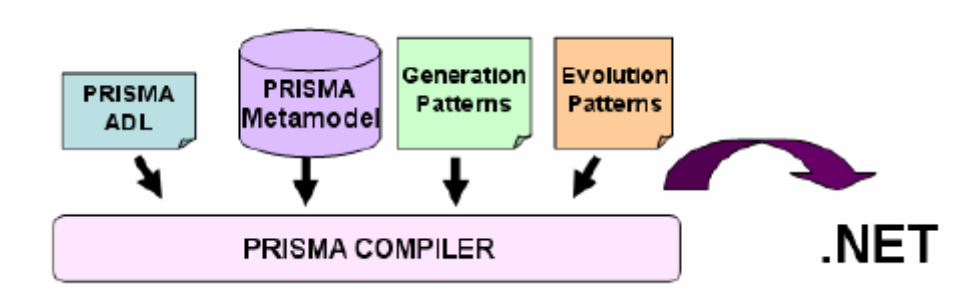


Figura 1 - Compilador PRISMA

Para el desarrollo del compilador de modelos PRISMA ha sido necesario encontrar una correspondencia entre los conceptos que introduce el metamodelo de PRISMA y la plataforma .Net. Esto fue sujeto de un proyecto previo a este proyecto y que se encargó de implementar un *middleware* que permite la carga, ejecución y evolución de modelos arquitectónicos PRISMA. Pero para el desarrollo del compilador final, es necesario realizar la implementación de un modelo arquitectónico de in caso de estudio que haga uso de todos los conceptos definidos en el modelo PRISMA. De forma se pueean identificar las equivalencias entre el Lenguaje de Descripción de Arquitecturas (LDA) PRISMA y el lenguaje de implementación *C#* . Por este motivo, el proyecto presenta una implementación (realizada a mano) de un caso de estudio concreto para identificar los patrones de generación de código que en un futuro empleará el compilador de modelos para emitir el código *C#* equivalente para cualquier especificación de un modelo arquitectónico PRISMA(ver Figura 1).

Entre los distintos tipos de sistemas *software* que tienen arquitecturas complejas, distribuidas, evolutivas y reutilizables, en este proyecto se ha escogido como ámbito de aplicación de PRISMA los sistemas de tele-operación. Los sistemas de tele-operación son sistemas de control que dependen del *software* para realizar sus operaciones. Su diseño es una tarea muy compleja que debe integrar elementos mecánicos, eléctricos y electrónicos y componentes *software* en un mismo sistema. Estos se utilizan para tele-operar mecanismos (robots, vehículos y herramientas) que llevan a cabo actividades de inspección y mantenimiento en entornos cuyas condiciones de habitabilidad o riesgo desaconsejan o impiden la presencia de operarios humanos.

En el marco de el proyecto europeo EFTCoR (*Environmental Friendly and Costeffective Technology for Coating Removal*) [EFTCoR] se está trabajando actualmente en el diseño y construcción de un sistema robótico tele-operado. Dicho sistema esta dedicado a la limpieza de cascos de buques de gran calado con el objetivo de minimizar el impacto medioambiental y corregir los defectos detectados en los prototipos anteriores (ver Figura 2). Dentro de este proyecto, surge con fuerza la necesidad de diseñar arquitecturas para sistemas de tele-operación a un mayor nivel de abstracción, de forma que sea posible reutilizar los diseños para distintos tipos de arquitecturas y que sean fácilmente modificables. Además, aparece la necesidad de la evolución de los diseños de forma correcta y preservando los requisitos de calidad del sistema, tanto en tiempo de compilación como en tiempo de ejecución.

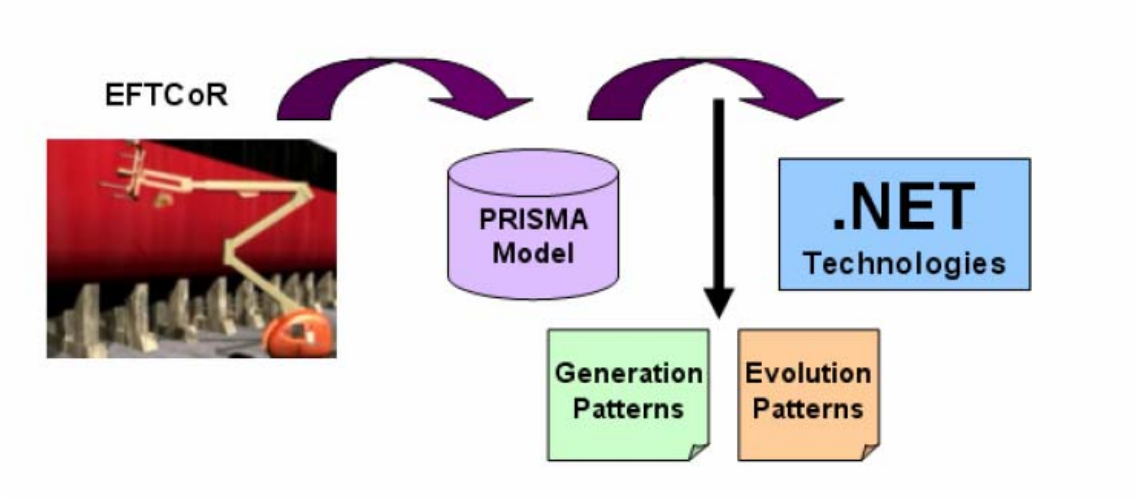


Figura 2 - Desarrollo de EFTCOR

Debido a las dificultades que se presentan a la hora de trabajar con un robot tele-operado de grandes dimensiones y de más de dos toneladas de peso se decidió especificar un sistema robótico tele-operado de características similares al EFTCoR para que sirviera como prototipo. Concretamente, el robot escogido, para utilizarse en este proyecto en las pruebas de especificación y desarrollo en *C#*, es el *MicroBot TeachMover* [TeachMover]. Este robot tradicionalmente se ha empleado con fines docentes para ayudar en el aprendizaje de sistemas robóticos tele-operados, y a pesar de ser más pequeño que el EFTCoR, comparte con éste el marco arquitectónico de referencia denominado ACROSET (Arquitectura de Control para RObots de SErvicios Tele-operados) [Ort05].

1.3 Objetivos

El objetivo principal de este proyecto es realizar una implementación del sistema robótico tele-operado *TeachMover* en *C#* en base al modelo PRISMA. La implementación será guiada por la especificación PRISMA del sistema robótico, realizada con el lenguaje de descripción de arquitecturas, para identificar los patrones de generación de código que empleará el compilador final de modelos PRISMA. Para alcanzar dicho objetivo se establecen una serie de objetivos intermedios que se detallan a continuación:

- Estudio y comprensión del caso de estudio del robot.
- Especificación del caso de estudio mediante el lenguaje de descripción de arquitecturas PRISMA en el marco de referencia ACROSET resaltando el potencial de expresividad y facilidad de uso del lenguaje.
- Traducción del caso de estudio del robot al lenguaje de implementación *C#* empleando el *middleware* PRISMA (PRISMANET).

- Identificación de equivalencias entre el ADL y *C#* y elaboración de los patrones de generación de código.

- Validación del modelo PRISMA y verificación del *middleware* PRISMA

Además, como objetivo secundario se pretende demostrar como implementar la integración de componentes ya fabricados (COTS: *Commercial Off-The-Shelf*) que generalmente acompañan los sistemas robóticos para facilitar la comunicación vía hardware.

1.4 Estructura

El presente proyecto está estructurado en siete capítulos y la bibliografía. A continuación se expone un breve resumen del contenido de cada capítulo:

- **Capítulo 1:** Planteamiento y Objetivos: Presentación de las disciplinas DSBC y DSOA en el modelo PRISMA y los objetivos a alcanzar en el proyecto.

- **Capítulo 2:** PRISMA: Descripción del modelo PRISMA y del lenguaje de descripción de arquitecturas.

- **Capítulo 3:** Caso de Estudio: Robot TEACHMOVER: Descripción y estudio de las características morfológicas del robot *TeachMover*, comunicación y cinemática.

- **Capítulo 4:** Arquitectura PRISMA para el Caso de Estudio: Especificación completa del robot *TeachMover* en el marco de referencia ACROSET.

- **Capítulo 5:** Implementación PRISMA: Implementación en *C#* empleando el *middleware* PRISMANET e identificando los patrones de generación de código.

- **Capítulo 6:** Integración de COTS en PRISMA: Técnica de integración empleada en la implementación del caso de estudio mediante la interoperabilidad que proporciona la plataforma .Net.

- **Capítulo 7:** Conclusiones y Trabajos Futuros: Resultados de los objetivos alcanzados y presentación de PRISMA como lenguaje específico de dominio sobre la plataforma .Net.

Capítulo 2

PRISMA

CONTENIDOS

PRISMA

En un futuro, PRISMA será un marco de trabajo o *framework* que incluirá un modelo, unos lenguajes, una herramienta y una metodología. En este capítulo se presentará tanto el modelo como los lenguajes de este *framework*.

PRISMA se presenta como un enfoque integrado y flexible para describir modelos de arquitectura complejos, distribuidos, evolutivos y reutilizables. Para ello, se basa en componentes y aspectos para la construcción de modelos arquitectónicos y se caracteriza por la integración que realiza del Desarrollo de *Software* Basado en Componentes (DSBC), del Desarrollo de *Software* Orientado a Aspectos (DSOA) y por sus propiedades reflexivas.

PRISMA define una metodología guiada y (semi-)automatizada para obtener el producto *software* final basándose en el paradigma de la prototipación automática de Balzer [Bal85]. En esta metodología, el usuario utilizará una herramienta de modelado (compilador de modelos) que le permitirá validar el modelo mediante la animación de los modelos arquitectónicos definidos en la especificación. PRISMA está basado en un lenguaje de especificación formal Orientado a Objetos llamado OASIS [Let98]. OASIS es un lenguaje formal para definir modelos conceptuales de sistemas de información orientados a objetos, que permite validar y generar las aplicaciones automáticamente a partir de la información capturada en los modelos. PRISMA preserva las ventajas de OASIS, garantizando la compilación de sus modelos, y lo extiende, para poder definir formalmente la semántica de los modelos arquitectónicos, ya que OASIS únicamente es capaz de especificar sistemas de información orientados a objetos.

2.1 El Modelo PRISMA

El modelo arquitectónico PRISMA [Per03] integra dos aproximaciones de desarrollo de *software*, el DSBC y el DSOA. Esta integración se consigue definiendo los tipos mediante la composición de aspectos. La mayoría de modelos arquitectónicos analizan cuáles son las primitivas básicas para la especificación de sus arquitecturas y exponen su sintaxis y semántica. El modelo PRISMA, además de definir los elementos arquitectónicos básicos y especificar su sintaxis y semántica, también especifica los aspectos que cubren las necesidades de cada uno de ellos.

Un elemento arquitectónico de PRISMA puede ser analizado desde dos vistas diferentes: la interna y la externa. La vista interna (ver Figura 3) define un elemento arquitectónico como un prisma con tantas caras como aspectos considere. Dichos aspectos están definidos desde la perspectiva del problema y no de su solución, aumentando el nivel de abstracción y evitando el solapamiento de código que puede sufrir la programación orientada a aspectos.

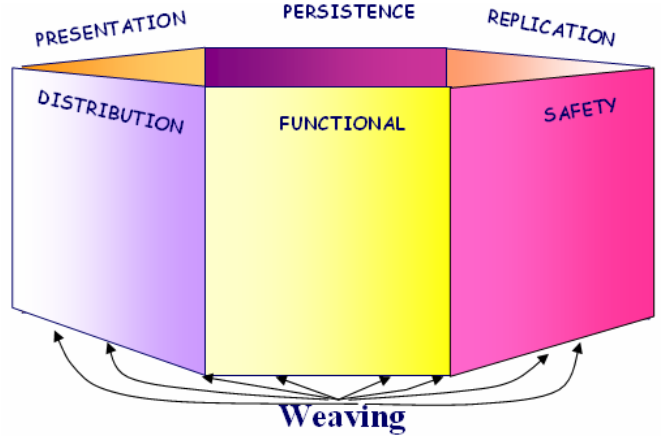


Figura 3 - Vista interna de un elemento arquitectónico PRISMA

Por otro lado, la vista externa de un elemento arquitectónico encapsula la funcionalidad como una caja negra y publica el conjunto de servicios que ofrece al resto de elementos arquitectónicos (ver Figura 4)

Figura 4 - Vista externa de un elemento arquitectónico PRISMA

Por ello, el modelo PRISMA puede analizarse desde dos perspectivas diferentes, la orientada a aspectos (vista interna) y la orientada a componentes (vista externa). A continuación se presenta el modelo desde ambas perspectivas.

2.2 Visión orientada a Aspectos

Las técnicas AOP, como han sido mostradas en los capítulos anteriores, permiten encapsular en un solo elemento (los aspectos) aquella funcionalidad que se repite a lo largo de todo el sistema. Un aspecto puede verse como la unión de un conjunto de interfaces que incluyen servicios y atributos del tipo del aspecto, más la especificación semántica de su estructura y comportamiento. No obstante, el modelo PRISMA va más allá y amplía el concepto de aspecto en el sentido de que cada aspecto representa una vista específica de la arquitectura del sistema (vista funcional, distribución, calidad, etc.). Por ello, cada primitiva de la arquitectura está formada a su vez por varios aspectos que la describen desde diferentes puntos de vista. Algunos tipos de aspectos pueden ser:

- **Aspecto Funcional:** Captura la semántica del sistema de información mediante la definición de sus atributos, sus servicios y su comportamiento
- **Aspecto de Coordinación:** Define la sincronización entre elementos arquitectónicos durante su comunicación.

- **Aspecto de Distribución**: Especifica las características que definen la localización dinámica del elemento arquitectónico en el cual se integra. También define los servicios necesarios para implementar estrategias de distribución de los elementos arquitectónicos (como movilidad, equilibrio de carga, etc.) con el objetivo de optimizar la distribución de la topología del sistema resultante [Ali03].

Existen diferentes tipos de aspecto, definiéndose cada uno de ellos de forma independiente. El número de tipos no está limitado, ya que gracias al metanivel pueden definirse nuevos tipos. Los mostrados son sólo algunos tipos de aspectos, en otros sistemas de información puede haber otros tipos diferentes, que emergerán de los requisitos de los sistemas a los que se aplique el modelo PRISMA.

No obstante, no es suficiente con definir únicamente los tipos de aspectos, sino que también hay que enlazarlos entre sí: esto se realiza mediante los *weavings*. Los *weavings* indican que la ejecución de un servicio de un aspecto puede provocar la invocación de servicios en otros aspectos.

A diferencia de las tecnologías orientadas a aspectos vistas hasta ahora, en la que los aspectos se enlazaban al código base, en PRISMA no existe el código base como tal, sino que la funcionalidad global de una primitiva del modelo arquitectónico se define mediante la unión de los distintos aspectos que la forman. Otra diferencia importante a resaltar, es que en las otras tecnologías (como *AspectJ*) los *weavings* se definen en el mismo aspecto, resultando en una pérdida de reutilización por parte del aspecto. Es por esto que en PRISMA los *weavings* entre los aspectos se definen en el elemento arquitectónico que los integra.

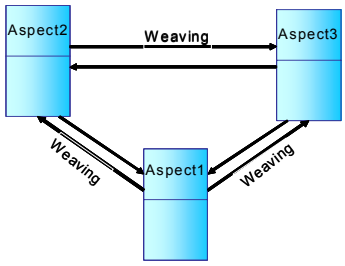


Figura 5 - Weavings entre aspectos

Como se puede observar en la figura, los aspectos se definen independientemente de la arquitectura donde se vayan a integrar, y son los *weavings* los que unen el conjunto de aspectos de un mismo elemento arquitectónico y forman la vista interna mostrada anteriormente. De la misma forma que en las tecnologías vistas anteriormente, PRISMA dispone de varios tipos de *weavings*, que son los siguientes:

- *After:* aspect1.service es ejecutado después de aspect2.service.
- *AfterIf (condition)*: igual que *After*, sólo si se cumple la condición.
- *Before:* aspect1.service es ejecutado antes de aspect2.service.
- *BeforeIf (condition)*: igual que *Before*, sólo si se cumple la condición.
- *Instead*: aspect1.service es ejecutado en lugar de aspect2.service.

Como se puede observar, los aspectos PRISMA son altamente reutilizables y favorecen la mantenibilidad, pues un cambio en un *concern* específico (i.e. el cambio en la estrategia de distribución de elementos arquitectónicos) sólo ha de realizarse en un aspecto y no en todo el conjunto del sistema. Además, la importación de COTS se expresa gracias al carácter opcional de los aspectos, lo que permite ver a las primitivas arquitectónicas como cajas negras en cuyo interior pueden incorporarse componentes COTS.

2.3 Visión orientada a Componentes

La visión externa de los elementos arquitectónicos está orientada hacia el concepto de componentes, a diferencia de la orientación a aspectos de la visión interna. El modelo arquitectónico consta de tres tipos de primitivas: *componentes*, *conectores* y *sistemas*. Y como se ha visto anteriormente, cada uno de estos tipos a su vez está compuesto por tantos aspectos como se consideren relevantes para definir el sistema de información con precisión.

2.3.1 Componentes

Un componente PRISMA se define como un elemento arquitectónico que captura la funcionalidad del sistema de información y no actúa como coordinador entre otros sistemas arquitectónicos. Puede verse como una parte del sistema que no se puede disgregar en partes más simples. Está formado por:

- un identificador,
- un conjunto de aspectos, que le proporcionan su funcionalidad,
- las relaciones de *weaving,* que proporcionan la cohesión entre los aspectos que lo forman,
- y los puertos de entrada y salida, cuyo tipo es una interfaz específica (un conjunto de servicios).

Los puertos son aquellos elementos que permiten la interacción del componente con los demás elementos arquitectónicos. Un puerto puede ofrecer un comportamiento servidor, cliente o cliente/servidor, en función de los servicios que lo definan. El comportamiento servidor especifica los servicios que el componente ofrece al resto de los demás elementos mientras que el comportamiento cliente especifica los servicios que puede requerir de los demás elementos arquitectónicos.

Los componentes PRISMA han de cumplir ciertas restricciones:

- todo componente debe especificar su aspecto funcional, excepto en el caso de componentes externos (COTS)
- un componente nunca puede contener un aspecto de coordinación

- un componente nunca puede contener dos aspectos del mismo tipo
- los tipos de los puertos de un componente sólo podrán ser aquellas interfaces que usen algunos de los aspectos que formen a dicho componente.

2.3.2 Conectores

Un conector PRISMA es un elemento arquitectónico que actúa como coordinador entre otros elementos arquitectónicos. El conector permite describir interacciones complejas entre componentes mediante su aspecto de coordinación. Está formado por:

- un identificador,
- un conjunto de aspectos, que le proporcionan su funcionalidad,
- las relaciones de *weaving,* que proporcionan la cohesión entre los aspectos que lo forman,
- y los roles de entrada y salida, cuyo tipo es una interfaz específica (un conjunto de servicios).

Los conectores sincronizan y conectan componentes, otros conectores o sistemas a través de sus roles, que de la misma forma que los puertos de los componentes, definen un conjunto de servicios que el conector ofrece (comportamiento servidor), que el conector necesita (comportamiento cliente) o un comportamiento mixto (cliente/servidor).

Existen una serie de restricciones que se han de tener en cuenta:

- un conector siempre ha de contener un aspecto de coordinación,
- un conector nunca puede contener un aspecto funcional,
- un conector nunca puede tener dos aspectos del mismo tipo,
- los tipos de los roles de un conecto sólo podrán ser aquellas interfaces que usen alguno de los aspectos que formen a dicho conector,
- una instancia de conector al menos ha de conectar dos elementos arquitectónicos PRISMA

2.3.3 Sistemas

En la mayoría de modelos arquitectónicos, surge la necesidad de tener mecanismos de abstracción que permitan tener elementos de mayor granularidad, aumentando la modularidad, composición y reutilización de elementos arquitectónicos. En PRISMA esto se consigue mediante los sistemas, los cuales permiten encapsular un conjunto de conectores, componentes y otros sistemas correctamente conectados entre sí. En dicha encapsulación pueden surgir propiedades emergentes.

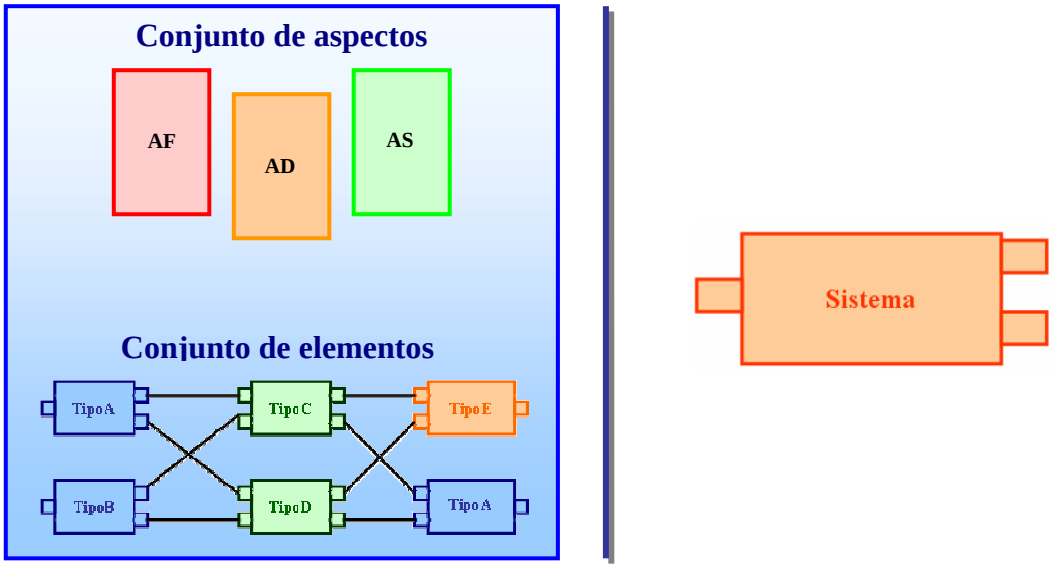


Figura 6 - Vistas de un Sistema PRISMA

Un sistema está formado por los mismos elementos y restricciones que un componente: una función de identificación, un conjunto de aspectos, una serie de *weavings* que los interconecta, y los puertos. No obstante, al ser un tipo compuesto, dispone también de un conjunto de elementos arquitectónicos que pueden o no estar conectados entre sí y realizan una determinada función para el sistema.

La especificación de la conexión entre estos elementos y el propio sistema se realiza mediante los *bindings*, que son los enlaces entre los puertos del sistema y los puertos (o roles en caso de ser conectores) de los elementos que encapsula. Permiten mantener un enlace entre los distintos niveles de granularidad de los elementos arquitectónicos que forman un sistema.

Por otra parte, las conexiones entre los distintos elementos que contiene (componentes, conectores y *attachments*) se realizan mediante los *attachments*, que establecen la conexión entre puertos de componentes y roles de conectores.

2.4 Lenguaje de definición de Arquitecturas

El Lenguaje de Descripción de Arquitecturas de PRISMA (ADL) está basado en OASIS [Let98], como ya se comentó anteriormente. A diferencia de otros modelos, el ADL (*Architecture Definition Language*) de PRISMA está dividido en dos niveles de especificación: el *nivel de definición de tipos* y el *nivel de configuración*.

El nivel de definición de tipos de PRISMA potencia la reutilización y combina el DSBC y DSOA. Este nivel permite definir los tipos necesarios para especificar un sistema arquitectónico. Dichos tipos se guardan en una librería para posteriormente reutilizarlos en la definición de distintos modelos arquitectónicos. Sus ciudadanos de primer orden son: interfaces, aspectos, componentes y conectores.

El nivel de configuración permite definir las instancias y especificar la topología del modelo arquitectónico. Para ello, en primer lugar se han de importar todos aquellos tipos (conectores, componentes y sistemas) definidos mediante el lenguaje de definición de componentes, que se necesiten para un determinado modelo arquitectónico. Después, se ha de definir el conjunto de instancias necesarias de cada uno de los tipos importados. Finalmente, se debe especificar la topología, interconectando adecuadamente las instancias del modelo.

Esta diferenciación de nivel de granularidad proporciona importantes ventajas frente a la mezcla de distintos niveles de granularidad dentro de la especificación. Una de ellas es que permite gestionar de forma independiente los tipos y las topologías específicas de cada sistema, incrementando la reutilización y obteniendo un mejor mantenimiento de las librerías de tipos. Además, permite discernir claramente entre la evolución de tipos (nivel de definición de tipos) y la evolución arquitectónica (nivel de configuración), teniendo meta-eventos diferentes en cada lenguaje para hacer evolucionar a los tipos o a las conexiones existentes entre sus instancias.

2.4.1 Definición de Tipos

A continuación se describirá la sintaxis de los ciudadanos de primer orden de PRISMA: interfaces, aspectos, componentes y conectores. Los sistemas también se definen en este nivel con el objetivo de que se puedan almacenar conjuntamente con los demás elementos y favorecer la reutilización.

2.4.1.1 Interfaces

Las interfaces son el mecanismo usado en PRISMA para establecer canales de comunicación entre los componentes, conectores y sistemas. Una interfaz describe la visibilidad parcial que tienen el resto de elementos, del estado y comportamiento del tipo que define la semántica de dicha interfaz. Por lo tanto, una instancia sólo puede solicitar y consultar aquellas propiedades y servicios publicados en sus interfaces. De este modo, una interfaz actúa como mecanismo de seguridad y permite que los componentes, conectores y sistemas sean vistos como cajas negras, preservando un alto nivel de abstracción.

Las interfaces definen los servicios sin tener en cuenta los puertos, roles o aspectos que las van a utilizar, con el objetivo de favorecer la reutilización. No obstante, se ha de tener en cuenta que todos los servicios que define una misma interfaz han de pertenecer al mismo tipo: funcional, distribución, etc.

pues serán implementadas por completo por un aspecto de un tipo concreto, y no puede haber solapamiento entre aspectos de tipos diferentes.

A continuación, se presenta la interfaz *ICreditCardTransactions*, que define las operaciones típicas para el manejo de cuentas bancarias, e *IMobility*, que define los servicios de movilidad que puede contener un componente.

```
Interface ICreditCardTransactions
   Withdrawal(input quantity: decimal, output newMoney: decimal);
   Balance(output newMoney: decimal);
   ChangeAddress(input newAdd: string);
End_Interface ICreditCardTransactions;

Interface IMobility
  Move(input newLoc: LOC);
End_Interface IMobility;
```

Como se puede observar, los servicios definen los parámetros que son de entrada mediante la palabra reservada *input*, mientras que los de salida se definen mediante la palabra reservada *output*.

2.4.1.2 Aspectos

Para definir los aspectos se ha definido una sintaxis específica basada en OASIS [Let98]. La plantilla genérica se presenta a continuación:

```
1   tipo_aspecto Aspect nombre using interface1, ... interfacen;

2   Attributes
      [Constant | Variable]
        <nombre_atributo1> : <tipo_atributo>;
        ...
        <nombre_atributon> : <tipo_atributo>;
      [Derived]
        <fórmula_derivación>

3   Constraints
      static <restricciones_estáticas>

    Services
4     Begin [(<arg_servicio>)] [as <nombre_servicio> <arg_servicio)];
        Valuations
          <fórmulas_observabilidad_atributo>

      [in | out] <nombre_servicio> (<arg_servicio>)
        [as <nombre_servicio> (<arg_servicio)];
5       Valuations
          <fórmulas_observabilidad_atributo>

      End [(<arg_servicio>)] [as <nombre_servicio> <arg_servicio)];
        Valuations
          <fórmulas_observabilidad_atributo>

6   Preconditions
      <fórmula_precondición_evento>

7   Triggers
      <fórmula_disparo_evento>

8   Operations [transaction]
      <fórmula_transacción>

9   Played Roles
      <fórmula_played_role>

10  Protocols
      <fórmula_protocolo>

    End_ tipo_aspecto Aspect name;
```

Tabla 1 - Plantilla de un aspecto PRISMA

La cabecera de un aspecto (*sección 1*) define el tipo del aspecto (funcional, distribución, etc.), el nombre que lo identifica y las interfaces a las que se da semántica. La *sección 2* define la lista de atributos que almacenan el estado de un aspecto. Para cada atributo se indica el nombre y el tipo de valores que va a contener. Puede ser de tres tipos:

- Constante: su valor se establece en la inicialización y no es modificado
- Variable: su valor puede cambiar cuando ocurra una acción relevante (como la invocación de un servicio o la activación de un disparador)
- Derivado: su valor se especifica en términos de los valores de otros atributos, mediante una fórmula de derivación.

La *sección 3* define las restricciones de integridad que deben cumplirse a lo largo de toda la vida de las instancias del tipo del aspecto. Son fórmulas basadas en el estado del aspecto y se pueden clasificar en estáticas o dinámicas, dependiendo de si se refieren sólo a un estado o relacionan diferentes estados, respectivamente. Las restricciones dinámicas se caracterizan por el uso de operadores temporales como *sometimes* o *always*.

La *sección 4* define los servicios que implementa el aspecto. Todos los servicios de la interfaz o interfaces a las que da semántica el aspecto deben definirse, además de aquellos servicios propios y no públicos del aspecto. Existen diferentes tipos de servicio: el servicio *begin* permite inicializar atributos, el servicio *end* permite liberar la memoria asociada al aspecto y finalmente, los servicios de modificación y consulta. Es importante resaltar que los servicios *begin* y *end* no indican que un aspecto sea instanciable, sino que hacen referencia al servicio de creación y destrucción del elemento arquitectónico al cual el aspecto pertenecerá. Además, cada servicio de modificación o consulta puede tener un comportamiento servidor (*in*), comportamiento cliente (*out*) o un comportamiento servidor y cliente (*in/out*). El comportamiento servidor se caracteriza porque el aspecto proporciona un servicio que puede ser requerido por otros aspectos, mientras que el comportamiento cliente indica que el aspecto va a requerir dicho servicio a otro aspecto/componente.

De la misma forma que se ha visto en la definición de los servicios en la interfaz, los argumentos de los servicios que incluye un aspecto pueden ser de entrada (*input*) o de salida (*output*), pero no devuelven ningún valor.

Por otro lado, el lenguaje ADL también proporciona un mecanismo de renombrado de servicios, mediante el operador *as*, con lo que se consigue que el nombre del servicio definido en una interfaz no tenga que coincidir con el nombre del servicio definido en el aspecto.

La semántica que define el comportamiento de los servicios se especifica mediante las *evaluaciones* (*sección 5*), que permiten definir el cambio de estado de los atributos del aspecto ante la ejecución de un servicio determinado. Se definen mediante fórmulas en lógica dinámica del tipo "$\varphi \rightarrow [a]\phi$" y se interpretan como: "si en un determinado estado del aspecto se

satisface φ y ocurre la acción 'a', en el estado inmediatamente posterior se satisface ϕ". Se puede no declarar la condición de evaluación φ asumiéndose *true*.

Las *secciones 6 y 7* definen respectivamente las precondiciones y los disparadores (*triggers*). Las precondiciones definen las condiciones que se deben cumplir para que un servicio se pueda ejecutar, mientras que los disparos permiten especificar la ejecución de un servicio cuando se cumple una determinada condición.

La *sección 8* define las operaciones y transacciones. Una operación es un servicio no elemental y no atómico, en el que se define una serie de servicios que se van a ejecutar secuencialmente. Para que estos servicios se ejecuten de forma atómica se emplea la palabra reservada *transaction.* Por ejemplo, la definición de una operación transaccional para el envío de noticias a una lista de correo en PRISMA sería:

```
Operations
  transaction EnviarNoticias(input listaCorreo: String, input
                                 listaNoticias: StringList):
    ENVIARNOTICIAS = SeleccionarListaCorreo(listaCorreo).NOTICIAS;
    NOTICIAS =  SeleccionarNoticias(listaNoticias, coleccion).ENVIO;
    ENVIO = envioSimultaneo(coleccion).CONFIRMACIÓN;
    CONFIRMACION = confirmacionLlegada.ENVIARNOTICIAS;
```

En el ejemplo se puede observar cómo se realiza el mapeo de parámetros de la transacción a cada uno de los servicios y cómo se especifica el siguiente estado alcanzado *nombre_servicio.nuevo_estado*.

Por último, en la *sección 9 y 10* se especifican los *Played_Roles* y los Protocolos. El *Protocolo* define un proceso describiendo un conjunto de acciones cuya ocurrencia es posible. Está compuesto por un conjunto de procesos que establecen los servicios y transiciones a otros procesos posibles. Además, también permite especificar las prioridades de ejecución de cada uno de los servicios implicados.

Por otra parte, un *Played_Role* es una proyección del protocolo que define el comportamiento parcial de un rol o papel determinado. Por lo tanto, todo *Played_Role* debe ser compatible en signatura y proceso con el protocolo de forma que, dicho papel o rol se desempeña dentro del comportamiento global del protocolo y sus cálculos son un subconjunto de los cálculos del proceso.

Un *Played_Role* es una vista parcial del protocolo que tiene sentido por sí solo, un comportamiento específico y restrictivo que es posible asociarlo posteriormente a un puerto o rol de un componente o conector. Dicha asociación permite definir de forma exacta el comportamiento que debe ejercer un puerto o rol además de los servicios que publica, establecidos por la interfaz que lo tipa.

Ambos se basan en el *Cálculo Poliádico* [Mil91], el cual permite describir de forma sencilla la ejecución de procesos y servicios susceptibles de ejecutarse concurrentemente. Una parte de su sintaxis es la siguiente:

x,y,z	Nombre (interfaz, servicio, played_role, parámetro, identif.)
x .y	Jerarquía de nombres, tal que x = interfaz e y = servicio
All | * Todos los played_roles	
x(m, n)	Vector de nombres tal que m, n son parámetros
P,Q,R	Proceso
P ::= x?(m,n).P	Prefijo de entrada o acción de recepción
| x!(m,n).P	Prefijo de salida o acción de envío
| P + Q	Selección no determinista
| P ∧ Q	Conjunción
| P || Q	Composición paralela (concurrencia de procesos)
| P → Q	Composición secuencial
| x(m,n):*k*.P	*k* indica la prioridad del servicio x, tal que *k* = 0, 1, ...
| *if* b *then* P *else Q*	Alternativa (derivado)
| *case*(b □ P→Q)	Alternativa múltiple
| |x=y| P	Comparación o *matching*

Tabla 2 - Sintaxis de Pi-Cálculo Poliádico

Para ilustrar estos conceptos, se muestran el conjunto de aspectos (funcional, distribución y coordinación) de un sistema bancario sencillo. El primero de ellos es el aspecto funcional *BankInteraction,* que implementa la funcionalidad básica de una cuenta bancaria.

```
Functional Aspect BankInteraction using ICreditCardTransactions
  Attributes
    Constant
      numberId: decimal;
    Variable
      address: String;
      money: decimal;
  Constraints
    static { money >= 0}
  Services
    Begin(input accountId: decimal);
      Valuations
           [in begin(accountId)] numberId = accountId;
    in/out Withdrawal(input quantity: decimal,
                      output newMoney: decimal);
      Valuations
           [in Withdrawal(quantity, newMoney)] money = money -
quantity;
    in/out Balance(output newMoney: decimal);
      Valuations
           [in Balance(newMoney)] newMoney = money;
    in/out ChangeAddress(input newAdd: String);
      Valuations
           [in ChangeAddress(newAdd)] address = newAdd;
    end;
  Preconditions
    in withdrawal(quantity)
      if quantity <= money;
  Protocols
    BANKINTERACTION = begin.TRANSACTION;
  TRANSACTION = ICreditCardTransactions.Withdrawal?(quantity,
newMoney).TRANSACTION +
```

```
    ICreditCardTransactions.Balance?(newMoney).TRANSACTION +
    ICreditCardTransactions.ChangeAddress?(newAdd).TRANSACTION +
    end;
End_Functional Aspect BankInteraction;
```

Como puede observarse, se han definido una serie de atributos, uno de los cuales es constante (el número de cuenta, que se define en el momento de creación). Se han definido los servicios correspondientes para sacar dinero de la cuenta (*Withdrawal*), para la consulta de saldo (*Balance*), y para cambiar la dirección (*ChangeAddress*). Obsérvese cómo se ha definido la semántica de dichos servicios mediante la lógica dinámica.

Se ha definido una restricción (sección *constraints*) mediante la cual no se permite extraer dinero si la cuenta no dispone de saldo, y una precondición (sección *preconditions*) para el servicio *Withdrawal* por la cual se establece que sólo se podrá sacar dinero si se dispone de dicha cantidad en la cuenta.

En la sección *Protocols* se han definido dos procesos: BANKINTERACTION, que es el proceso inicial, y TRANSACTION. Obsérvese la sintaxis en Pi-Cálculo Poliádico para indicar que la petición (comportamiento servidor, mediante el símbolo ‘ ? ’) de cualquier servicio (*Interfaz.Nombre_Servicio*) volverá al proceso TRANSACTION.

En segundo lugar se muestra el aspecto de distribución y el aspecto de coordinación. Ambos son importantes, pues formarán parte de los componentes que formarán el sistema que se construirá como ejemplo. La finalidad del aspecto de distribución *ExtMbile* no es más que indicar que el componente que lo utilice puede distribuirse a una nueva máquina.

```
Distribution Aspect ExtMbile using IMobility;
  Attributes
    Location: LOC NOT NULL;

  Services
    begin(input initialLOC: LOC);
      Valuations
          [in begin(initialLOC)] Location = initialLOC;
    in Move(input newLoc:LOC);
      Valuations
        [move(newLoc)] Location:= newLoc;
    end;

  Protocols
    CREATION = begin.EXTMBILE;
    EXTMBILE = IMobility.Move?(newLoc).EXTMBILE + end;
End Distribution Aspect ExtMbile;
```

La finalidad del aspecto de coordinación es coordinar las peticiones entre componentes que demanden el servicio *Withdrawal* o *Balance* al componente que actúe de servidor:

```
Coordination Aspect BankCoordination using ICreditCardTransactions
  Services
    in/out Withdrawal(input quantity: decimal, output money: decimal);
    in/out Balance(output money: decimal);
```

```
    begin;
    end;

  Played_Roles
    CLIENT =
      (ICreditCardTransactions.Withdrawal?(quantity, money)
       →
       ICreditCardTransactions.Withdrawal!(quantity, money) )
    +
      (ICreditCardTransactions.Balance?(money)
       →
       ICreditCardTransactions.Balance!(money) )

    SERVER =
      (ICreditCardTransactions.Withdrawal!(quantity, money)
       →
       ICreditCardTransactions.Withdrawal?(quantity, money) )
    +
      (ICreditCardTransactions.Balance!(money)
       →
       ICreditCardTransactions.Balance?(money) )

  Protocol
    BANKCOORDINATION = begin.COORD;
    COORD =
      (CLIENT.Withdrawal?(quantity, money) →
       SERVER.Withdrawal!(quantity, money) →
       SERVER.Withdrawal?(quantity, money) →
       CLIENT.Withdrawal!(quantity, money)).COORD
      +
      (CLIENT.Balance?(money) →
       SERVER.Balance!(money) →
       SERVER.Balance?(money) →
       CLIENT.Balance!(money)).COORD + end;

End_Coordination Aspect BankCoordination;
```

Con este ejemplo, más complejo que los anteriores, puede observarse la potencia expresiva del *Pi-Cálculo Poliádico*. Este aspecto se utilizará en un conector para coordinar dos componentes, uno que se comportará como cliente y el otro como servidor. Para ello, se han definido dos *played_roles*: uno para el comportamiento CLIENT y otro para el comportamiento SERVER. El *played_role* CLIENT describe el flujo de entradas y salidas para el comportamiento cliente. Por ello, cuando se reciba una petición del servicio *Withdrawal* (flujo de entrada, indicado mediante el símbolo '?') la siguiente acción válida sólo podrá ser devolver el resultado de dicha petición (flujo de salida, indicado mediante el símbolo '!'). En cambio, el *played_role* SERVER modela el otro lado de la comunicación: cuando se reciba un flujo de salida (invocar un servicio), la única acción válida sólo podrá ser un flujo de entrada (obtener los resultados de dicho servicio). Los símbolos '!' deben verse como flujos de *salida* (*output*) y los '?' como flujos de *entrada* (*input*).

Por otra parte, el protocolo modela el proceso global: coordina y sincroniza los servicios de los distintos *played_roles*. En la Figura 7 se puede observar el esquema de funcionamiento de un protocolo de forma gráfica. Cuando llegue una petición de *Withdrawal* (?) al aspecto de coordinación, la siguiente acción que se realizará es transmitir dicha petición (!) al componente

que actúa como servidor (el componente *Banco*). Entonces, el aspecto esperará hasta que se reciba la respuesta por parte del servidor (`SERVER.Withdrawal?`), tras lo cual se retransmitirá al componente cliente (`CLIENT.Withdrawal!`). Una vez se enviase al cliente, el siguiente estado válido sería COORD, y podría procesarse otro servicio diferente.

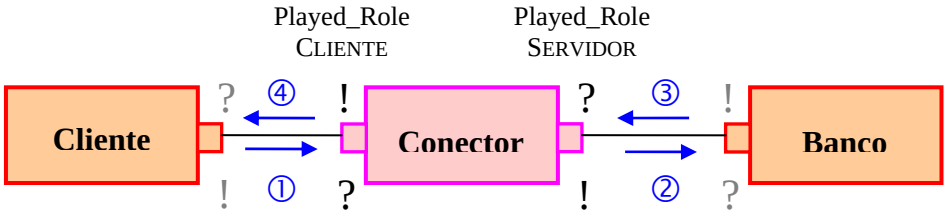


Figura 7 - Esquema de funcionamiento de un protocolo

Si se quisiese permitir la ejecución de otro servicio (como *Balance*) mientras el aspecto de coordinación espera la respuesta del servidor, se indicaría mediante el uso del operador '||' (composición paralela), en lugar del operador '+' (selección no determinista), con lo que se especificaría la ejecución concurrente de servicios.

2.4.1.3 Componentes y Conectores

La plantilla de un componente y de un conector, muy similares entre sí, están formadas por cuatro partes básicas: la cabecera, la definición de puertos o roles de comunicación, los aspectos que lo forman y los *weavings*, que los entretejen entre sí. A continuación se muestra la plantilla genérica, y se describirá cada sección en detalle:

```
1   [Component | Connector]_type <nombre_Componente|Conector>

2   [Port | Role]
      <nombre_i> : <interfaz_i>
3     [Played_Role <nombre_aspecto>.<nom_playedRole_del_aspecto>];
    End_[Port | Role];

4   [Functional | Coordination] Aspect Import <nombre_aspecto_j>;
    <tipo_aspecto_k> Aspect Import <nombre_aspecto_k>;

5   [Weaving
        <nombre_aspecto_k>.<nombre_servicio_k>
              <operador_weaving>
        <nombre_aspecto_n>.<nombre_servicio_n>;
    End_Weaving;]
```

```
6   Initialize
      New(<argumentos>) {
        <nombre_aspecto_i>.begin(<args_constructor_i>);
      }
    End_Initialize;

7   Destruction
      Destroy() {
        <nombre_aspecto_i>.end(<args_destructor_i>);
        <nombre_aspecto_i+1>.end(<args_destructor_i+1>);
      }
    End_Destruction;

    End_[Component | Connector]_type <nombre_Componente|Conector>;
```

Tabla 3 - Plantilla de un componente/conector PRISMA

La cabecera (sección 1) se compone de la palabra reservada *component_type* (para el caso de los componentes) o de *connector_type* (para el caso de los conectores), seguida del nombre que reciba el componente o conector, y que lo identificará unívocamente en la librería PRISMA.

Los puertos de comunicación (o roles en el caso de conectores) se definen en la sección 2, indicando qué interfaz implementan. Además, se puede asociar un played_role al puerto (sección 3) para especificar el comportamiento del conjunto de servicios que forman parte de la interfaz. El played_role pertenece al aspecto indicado mediante la notación punto `<nombre_aspecto>.<nom_playedRole_del_aspecto>`. De esta forma, pueden crearse varios puertos con la misma interfaz pero con played_roles diferentes, con lo que cada puerto con idéntica interfaz tendrá un comportamiento diferente: el especificado por el played_role.

La importación de los aspectos (sección 4) se realiza indicando el tipo y el nombre del aspecto. Al menos debe importarse un aspecto funcional, en el caso de los componentes, o un aspecto de coordinación, en el caso de los conectores. También cabe recordar que no pueden importarse dos aspectos del mismo tipo dentro del mismo componente o conector.

El *weaving* entre los aspectos (sección 5) se define al importar cada aspecto, indicando con qué aspecto van a entretejerse sus servicios. Éstos se declaran especificando el nombre del aspecto en el cual están definidos mediante la notación: `<nombre_aspecto>.<nombre_servicio>`. El *weaving* entre los servicios se realiza mediante `<operador_weaving>`, que indica si la sincronización se realiza *after* (después), *before* (antes) o *instead* (en lugar de), como se mostró en el apartado 2.2.

Por último, las secciones 6 y 7 describen el proceso de creación y destrucción de los componentes/conectores. En la sección *Initialize* se especifica cómo el servicio *New* desencadena la ejecución del servicio *Begin*

de los distintos aspectos que forman el componente, mientras que la sección *Destruction* desencadena la destrucción de los aspectos. Además, en el servicio *New* pueden indicarse los parámetros que necesitan los aspectos para su inicialización, que a su vez pueden ser proporcionados por el elemento arquitectónico que cree el componente/conector.

A continuación, se muestran dos ejemplos de componentes: el componente *ATM*, y el componente *Account*, que importa el aspecto funcional definido anteriormente y realiza el *weaving* del servicio *ChangeAddress* con el servicio *Move* del aspecto de distribución:

```
Component_type ATM
  Port
    VISACreditCard_port: ICreditCardTransactions;
    AccountTrans_port: ICreditCardTransactions;
  End_port;

  Functional Aspect import BankInteraction;
  Distribution Aspect import ExtMbile;
  Initialize
    New(accountId: decimal, initialLocation: LOC) {
      BankInteraction.begin(accountId);
      ExtMbile.begin(initialLocation);
    }
  End Initialize;

  Destruction
    Destroy() {
      BankInteraction.end();
      ExtMbile.end();
    }
  End Destruction;
End_Component_Type ATM;
```

```
Component_type Account
  Port
    Account_port: ICreditCardTransactions;
  End_port;

  Functional Aspect import BankInteraction;
  Distribution Aspect import ExtMbile;

  Weaving
    BankInteraction.ChangeAddress(newAdd: string)
         before ExtMbile.Move(newAdd: string);
  End_Weaving;

  Initialize
    New(accountId: decimal, initialLocation: LOC) {
      BankInteraction.begin(accountId);
      ExtMbile.begin(initialLocation);
    }
  End Initialize;

  Destruction
    Destroy() {
      BankInteraction.end();
      ExtMbile.end();
    }
  End Destruction;
End_Component_Type Account;
```

También se muestra un conector cuya función sería la de coordinar entre sí los dos componentes definidos anteriormente:

```
Connector_type ATMAccount
  Role
    ATM_role: ICreditCardTransactions,
      Played_Role BankCoordination.CLIENT;
    Account_role: ICreditCardTransactions,
      Played_Role BankCoordination.SERVER;
  End_role;

  Coordination Aspect import BankCoordination;
  Distribution Aspect import ExtMbile;

  Initialize
    New(initialLocation: LOC) {
      BankCoordination.begin();
      ExtMbile.begin(initialLocation);
    }
  End Initialize;

  Destruction
    Destroy() {
      BankInteraction.end();
      ExtMbile.end();
    }
  End Destruction;
End_Connector_Type ATMAccount;
```

Como puede observarse, el conector define dos roles con la misma interfaz, pero mediante los played_roles puede distinguir las peticiones de servicio que recibe (comportamiento cliente de los componentes a él conectados) y los servicios ofrecidos (comportamiento servidor). La semántica de coordinación de dichos played_roles vendrá definida por el aspecto de coordinación que lo define.

2.4.1.4 Sistemas, *Attachments* y *Bindings*

La plantilla genérica para definir un sistema es la mostrada en la siguiente tabla:

```
1   System_type <nombre_sistema>

2     Ports
        <nom_puerto_i> : <interfaz_i>;
        <nom_puerto_i+1> : <interfaz_i+1>;
      End_Ports

3     Variables
        <nom_Var_i> : <tipo_componente | tipo_conector>;
        ...
      End_Variables
```

```
4    Attachments
       <nom_Var_i>.<nom_puerto> ←→ <nom_Var_k>.<nom_rol>;
       ...
     End_Attachments;

5    Bindings
       <nom_Var_i>.<nom_puerto> ←→ <nombre_sistema>.<nom_puerto>;
       ...
     End_Bindings;

6    Initialize
       New() {
         <nom_Var_i>= new <(nom_componente | nom_conector)>(<args>);
         ...
       }
     End_Initialize;

7    Destruction
       Destroy() {
         <nom_Var_i>.destroy();
         ...
       }
     End_Destruction;
   End_System_type <nombre_sistema>;
```

Tabla 4 - Plantilla de un sistema PRISMA

En la cabecera de un sistema (sección 1) se indica el nombre identificativo de dicho tipo, que lo identificará dentro de la librería de tipos PRISMA. La definición de puertos (sección 2) permite indicar los puertos de comunicación que tendrá dicho sistema con el resto de elementos arquitectónicos. Mediante los *bindings* se establecerá la relación entre los puertos del sistema y los componentes que les darán semántica a los servicios de dichos puertos.

La definición de variables (sección 3) permite especificar los componentes y/o conectores que formarán parte del sistema. Cada componente importado se asociará con una variable, cuyo tipo será el del componente a importar. La construcción/destrucción de dichos componentes se desencadenará en el momento de creación/destrucción del sistema. Estas operaciones se especifican en la sección *Initialize* y *Destruction* (secciones 6 y 7), respectivamente, y permiten indicar los parámetros que necesitará cada componente para su creación.

La definición de un sistema especifica las relaciones de conexión (*attachments*) y composición (*bindings*) entre los elementos arquitectónicos que contiene. Como se definió anteriormente, los *attachments* establecen la conexión entre los puertos de los componentes y los roles de los conectores, mientras que los *bindings* definen la composición entre el sistema y los componentes o subsistemas que contiene. Esto se define en las secciones 4 y 5. Los *attachments* se especifican mediante una relación entre dos variables, cuyos tipos son un componente y un conector, y a través de la notación punto se indica el puerto y el rol que se va a conectar:

```
<variable: tipo_componente>.<puerto> ←→ <variable:
                tipo_conector>.<rol>
```

Los *bindings* se especifican de la misma forma, sólo que en lugar de definir la relación entre componentes y conectores, se define entre un componente o conector y un sistema.

Como ejemplo, en la siguiente figura se muestra el sistema *BankSystem* que encapsula los componentes y conectores definidos anteriormente, junto con la especificación PRISMA:

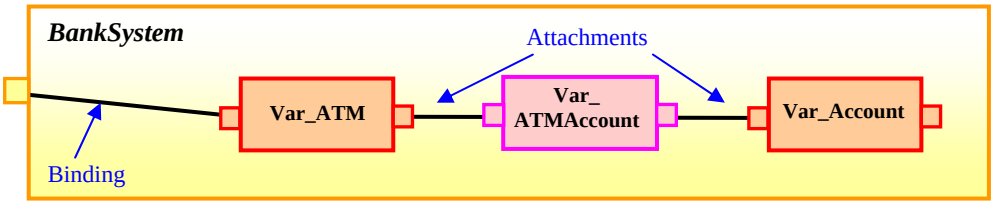


Figura 8 - Ejemplo BankSystem

```
System_type BankSystem
  Ports
    CreditCard_port : ICreditCardTransactions;
  End_Ports

  Variables
    Var ATM : ATM;
    Var Account : Account;
    Var ATMAccount : ATMAccount;
  End Variables

  Attachments
    Var_ATM.AccountTrans_port ←→ Var_ATMAccount.ATM_role;
    Var_Account.Account_port  ←→ Var_ATMAccount.Account_role;
  End Attachments;

  Bindings
    Var_ATM.VISACreditCard_port ←→ BankSystem.CreditCard_port;
  End_Bindings;

  Initialize
    New() {
      Var_ATM = new ATM(accountId: decimal, initialLocation: LOC);
      Var Account=new Account(accountId:decimal,initialLocation:LOC);
      Var ATMAccount = new ATMAccount(initialLocation: LOC);
    }
  End_Initialize;

  Destruction
    Destroy() {
```

```
        Var ATM.destroy();
        Var_Account.destroy();
        Var ATMAccount.destroy();
      }
   End Destruction;
End_System_type BankSystem;
```

Como puede observarse, se ha definido un sistema cuyo único puerto, *CreditCard_port*, está relacionado con el puerto *VisaCreditCard_port* del componente ATM, mediante una relación de *binding*. Además, los componentes que encapsula el sistema están unidos entre sí mediante relaciones de *attachment*, uniendo el puerto *AccountTrans_port* del componente ATM con el rol *ATM_role* del conector ATMAccount, y el puerto *Account_port* del componente *Account* con el rol *Account_role* del conector. Con esto se está especificando que todas las peticiones de servicio que lleguen al sistema se redirijan al componente *ATM*¸ que decidirá cómo procesar dichas peticiones. A su vez, este componente puede requerir comunicarse con el componente *Account*¸ y lo hará a través del conector *ATMAccount*¸ que coordinará las comunicaciones entre ambos.

2.4.2 Configuración Arquitectónica

El nivel de configuración del lenguaje permite definir las instancias y la topología del modelo arquitectónico final. En la tabla siguiente se puede observar la plantilla genérica, muy semejante a la del sistema visto en el punto anterior:

```
1   Architectural Model <nombre_modelo>
2     Variables
        <nom_Var_i>: <tipo_componente | tipo_conector | tipo_sistema>;
      End_Variables

3     Attachments
        <nom_Var_i>.<nom_puerto> ←→ <nom_Var_k>.<nom_rol>;
      End_Attachments;

4     Initialize
        New() {
          <nom_Var_i>= new <(nom_componente | nom_conector)>(<args>);
        }
      End_Initialize;

5     Destruction
        Destroy() {
          <nom_Var_i>.destroy();
        }
      End_Destruction;
    End Architectural Model <nombre_modelo>;
```

Tabla 5 - Plantilla de un modelo arquitectónico PRISMA

Como se puede observar, las secciones que forman parte de un modelo arquitectónico son muy similares a las de los sistemas. La cabecera (sección 1) define el nombre del modelo arquitectónico. La cláusula *Variables* (sección 2) define los componentes, conectores o sistemas que forman el modelo arquitectónico, que a la vez que importa los tipos de la librería de PRISMA, las asocia con un nombre de variable para posteriormente poderles hacer referencia en las siguientes secciones. La especificación de *Attachments* (sección 3) especifica, al igual que en los sistemas, la conexión entre los diferentes elementos arquitectónicos que conforman el modelo.

El bloque *Initialize* (sección 4) especifica cómo instanciar los elementos arquitectónicos que conforman el modelo, proporcionando los valores de inicialización que requieran sus constructores. Pueden seguirse dos aproximaciones. La primera de ellas consiste en especificar en el modelo arquitectónico las instancias que lo forman, proporcionando los correspondientes valores de inicialización. El problema que plantea esta aproximación es que cuando se quieran cambiar los valores de inicialización deberá modificarse la especificación del modelo arquitectónico. La otra aproximación consiste en definir el modelo arquitectónico como un tipo, es decir, no especificar los valores de inicialización que tomarán las instancias, tan

sólo la secuencia de llamadas a los constructores de los elementos arquitectónicos que forman el modelo arquitectónico. Más tarde, cuando se decidiese instanciar el modelo arquitectónico definido, se proporcionarían los valores de inicialización requeridos (ver ejemplo). Con esto, se favorece la reutilización del modelo arquitectónico definido. Además, la inicialización puede ser anidada, es decir, pueden especificarse los valores de inicialización de componentes encapsulados dentro de sistemas, con el objetivo de evitar que los constructores de dichos sistemas acumulen una larga lista de inicialización.

Por último, el bloque de destrucción (sección 5) define qué secuencia de destrucción de los componentes debe seguirse, y si es necesario, indicar servicios adicionales que deban especificarse, como por ejemplo, la persistencia de los datos.

Como ejemplo de modelo arquitectónico, va a tomarse únicamente el sistema definido en el apartado anterior:

```
Architectural Model SimpleBankSystem
   Variables
      Var BankSystem : BankSystem;
   End Variables

   Initialize
      New() {
         Var_BankSystem = new BankSystem();
      }
   End Initialize;

   Destruction
      Destroy() {
         Var_BankSystem.destroy();
      }
   End_Destruction;
End Architectural Model SimpleBankSystem;
```

Como se puede observar en el ejemplo, la definición del modelo arquitectónico está formada únicamente por un único sistema, el definido anteriormente. El bloque de inicialización únicamente define cómo construirlo, pero no lo instancia, con el objetivo de favorecer la reutilización del modelo arquitectónico. La instanciación del modelo arquitectónico se haría de la siguiente forma:

```
BANKSYSTEM = new BankSystem {
   ATM1 = new ATM(123456789, new LOC("tcp://garbi.dsic.upv.es"));
   ACCOUNT1= new Account(987654321,new LOC("tcp://sigil.dsic.upv.es"));
   ATMAccount1 = new ATMAccount(new LOC("tcp://garbi.dsic.upv.es"));
}
```

Obsérvese cómo se han ido anidando los constructores para inicializar los componentes que forman el sistema. El *framework* de PRISMA sería el encargado, mediante reflexión de código, de obtener los componentes que forman el sistema y de pedir al usuario los parámetros de inicialización necesarios, bien de forma gráfica mediante interfaces de usuario, o bien por consola. La otra alternativa consistiría en definir el constructor del sistema final con tantos argumentos como argumentos tuvieran cada uno de los constructores de los componentes/conectores que forman el sistema.

Capítulo 3

CASO DE ESTUDIO: ROBOT *TEACHMOVER*

CONTENIDOS

CASO DE ESTUDIO: ROBOT *TEACHMOVER*

El *MicroBot TeachMover* es un brazo robótico empleado generalmente para la enseñanza de fundamentos robóticos. Este modelo de robot fue diseñado específicamente para simular el comportamiento de robots de índole industrial.

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 9).

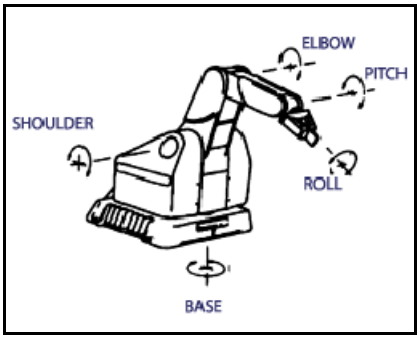


Figura 9 - Brazo Robot *MicroBot TeachMover*

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 10):

- Base: ± 90°

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

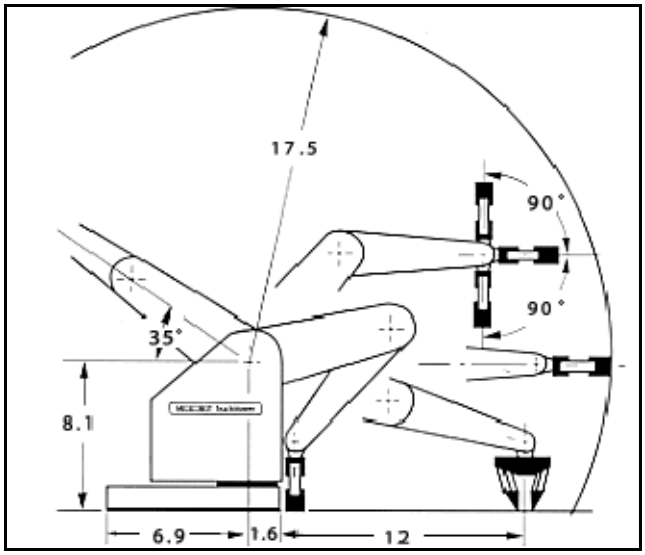


Figura 10 - Movimientos Brazo Robot

Como mencionamos 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 Comunicación

3.2.1 Mando de control

MicroBot TeachMover dispone de un mando de control (ver Figura 11) que permite a través de unas teclas de función mover todas las articulaciones del robot. El mando de control permite rotar la Base, el Hombro y el Codo en el sentido de las agujas del reloj o en el contrario. También permite controlar tanto la inclinación vertical de la muñeca como la rotación de esta y abrir y cerrar la pinza.

Figura 11 - Mando de control

Este mando soporta dos modos de control, el modo entrenamiento y el modo acción. Durante el *modo entrenamiento* se puede realizar cualquier tipo de movimiento libre combinando todos los movimientos previamente mencionados. Por otro lado, el *modo acción* da la posibilidad de ejecutar un programa de instrucciones de movimientos, puesto que el brazo de robot dispone un pequeño modulo de memoria capaz de almacenar hasta 106 instrucciones de movimientos (simples o compuestos). Antes de ejecutar el programa, los movimientos deben ser almacenados en la memoria del robot mediante el mando. Durante la ejecución de movimientos se puede detener de forma inmediata el movimiento del robot mediante la tecla de parada por si ocurriera alguna emergencia.

El mando de control es muy útil para familiarizarse con los tipos de movimiento que permite el brazo de robot aunque con él es complicado realizar movimientos realmente complejos como por ejemplo mover todas las articulaciones simultáneamente. Otra gran utilidad del mando de control es poder posicionar el brazo de robot en una posición inicial previa a una serie de movimientos por ejemplo en el caso que se haya detenido la ejecución mediante la tecla de parada de emergencia y se quiere volver a empezar.

3.2.2 Puerto Serie

Para realizar movimientos complejos o ejecutar una serie de movimientos de forma cíclica, es necesario enviar los comandos de movimiento a través de un ordenador. El robot incorpora un puerto serie RS-232C para poder establecer una comunicación entre el ordenador y el robot, a través del protocolo de enlace a 9600 baudios de velocidad modular, con longitud de 8 bits de trama para datos sin paridad y con un bit de parada. El micro-controlador del robot entiende el código ASCII por lo que todas las instrucciones transmitidas al robot son cadenas de caracteres que siguen con este estándar. Así, después de recibir una instrucción el micro-controlador del robot devuelve un número también en código ASCII en razón de reconocimiento al ordenador.

A continuación detallaremos un subconjunto del juego de instrucciones del robot. Se ha de tener en cuenta que todas las instrucciones que se envían a través del puerto serie comienzan con el símbolo ASCII "@" para que el robot las pueda procesar.

Instrucción STEP

- @STEP <sp>,<J1>,<J2>,<J3>,<J4>,<J5>,<J6><CR>

La instrucción STEP provoca el movimiento simultáneo de todas las articulaciones del brazo de robot, es decir causa la rotación de cada uno de los seis engranajes de los motores eléctricos.

- <sp> parámetro numérico que fija la velocidad del movimiento entre 0 y 245
- <J1> - <J6> parámetros numéricos que indican la cantidad de semi-pasos que cada uno de los seis motores debe rotar.
- <CR> símbolo ASCII equivalente al retorno de carro.

Parámetro	Articulación	Positivo	Negativo
J1	BASE	Izquierda	Derecha
J2	HOMBRO	Abajo	Arriba
J3	CODO	Abajo	Arriba
J4	MUÑECA Derecha	Abajo	Arriba
J5	MUÑECA Izquierda	Abajo	Arriba
J6	HERRAMIENTA	Abrir	Cerrar

Tabla 6 - Parámetros y Dirección del movimiento

Cuando se emplea el comando @STEP es necesario tener en cuenta que los hay ciertos engranajes que están acoplados, lo que significa que existe una dependencia entre sus parámetros. Es el caso del Codo <J3> y la Herramienta <J6>. Además, para la muñeca no se puede fijar directamente la inclinación vertical y la rotación, sino que es necesario fijarlo a través de los semi-pasos del engranaje izquierdo <J4> y derecho <J5> de la Muñeca. Como aclaración, si los semi-pasos para la Base, el Hombro, el Codo, la Inclinación Vertical (Muñeca), la Rotación (Muñeca), y la Pinza vienen dados por B, S, E, P, R, y G respectivamente, entonces el comando de movimiento sería de la siguiente forma:

- @STEP <sp>, B, S, E, (P - R), (P + R), (E+G)<CR>

Por ejemplo el comando,

- @STEP 200,0, 0,500,0, 0,700<CR> que provoca el descenso del codo en 500 semi-pasos y abre la pinza en 200 semi-pasos.

Otro ejemplo sería,

@STEP 200,0,0,0,-300,-300,0<CR> que provoca el movimiento inclinación vertical de la Muñeca en dirección hacia arriba sin rotar sobre sí misma

Y por último,

@STEP 200,0,0,0,-300,+300,0<CR> que provoca únicamente la rotación de la Muñeca hacia la derecha.

Instrucción CLOSE

- @CLOSE <sp>

La instrucción CLOSE provoca el cierre de la pinza hasta que el sensor detecta que se ha cogido algo o hasta que se cierra completamente.

- <sp> parámetro numérico fija la velocidad del movimiento entre 0 y 245

Existe una diferencia significativa entre emplear el comando CLOSE o el comando STEP para cerrar la pinza. El comando STEP cierra la pinza hasta que el motor ha recorrido todos los pasos que se le ha indicado en el comando mientras que con el comando CLOSE cierra la pinza hasta que el sensor detecta que se ha cerrado.

Un ejemplo para agarrar un objeto y ejercitar presión sobre él sería:

- @CLOSE 235
- @STEP 180,0,0,0,0,0,200

La presión máxima que se puede ejercer sobre un objeto es de 14 *Newtons* que equivaldría aproximadamente a 200 semi-pasos. La presión es proporcional al número de semi-pasos que se indiquen y es absolutamente dependiente de la forma del objeto.

3.2.3 Puerto Paralelo

Durante la ejecución de un movimiento del brazo de robot puede existir la necesidad de detener de forma inmediata la acción del brazo. Esto corresponde a una parada de emergencia y puede ser útil cuando el robot este moviéndose a una posición ilegal o que se de el caso de una situación de riesgo.

Por este motivo, al robot se le ha dotado de un mecanismo de interrupción que desconecta vía *software* la alimentación del robot (ver Figura 12). El mecanismo es un circuito que contiene un interruptor que cuando se abre, corta la alimentación eléctrica del robot. El circuito o relé esta conectado a través del puerto paralelo, de esta forma cuando se activa un 1 lógico en el pin D0 del puerto paralelo el relé desconecta la alimentación del robot y enciende un LED de aviso indicando que el robot deja de ser operativo. Para volver a activar la alimentación del robot bastará con fijar un 0 lógico en el mismo pin.

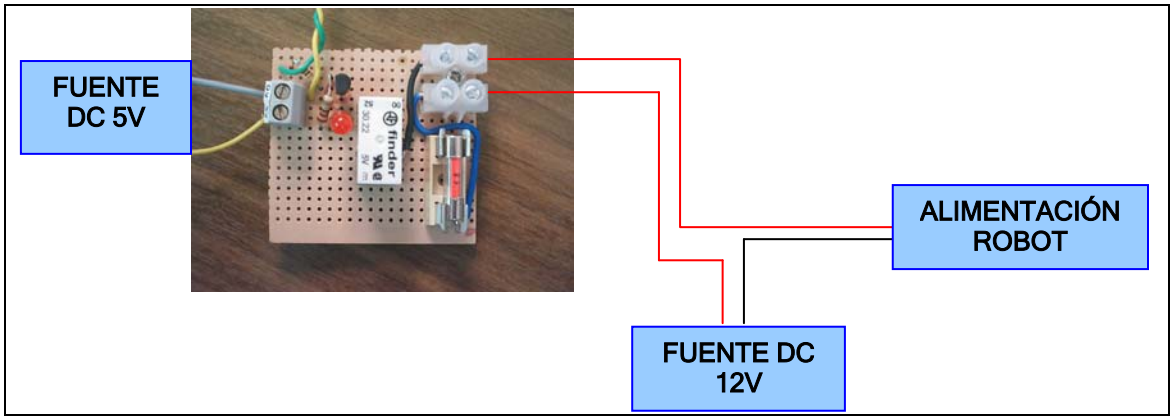


Figura 12 - Esquema de conexión de la placa al Robot y a la fuente de 5vdc.

3.3 Cinemática inversa

La cinemática del robot estudia el movimiento del mismo con respecto a un sistema de referencia. En este caso, la cinemática del brazo del robot trata el estudio analítico de la geometría del movimiento de un robot con respecto a un sistema de coordenadas de referencia fijo.

El objetivo del problema cinemático inverso consiste en encontrar los valores que deben adoptar los ángulos articulares del robot para que su extremo se posicione y se oriente según una determinada localización espacial. La solución a este problema pasa por ser fuertemente dependiente de la configuración del robot.

Segmento	Descripción	Longitud (pulgadas)
H	Desde superficie hasta centro eje HOMBRO	7,68
L	Desde eje HOMBRO hasta eje CODO	7
L	Desde eje CODO hasta eje MUÑECA	7
LL	Desde eje MUÑECA hasta extremo la PINZA abierta en 1,5 pulgadas	3,8

Tabla 7 – Configuración del Brazo de Robot

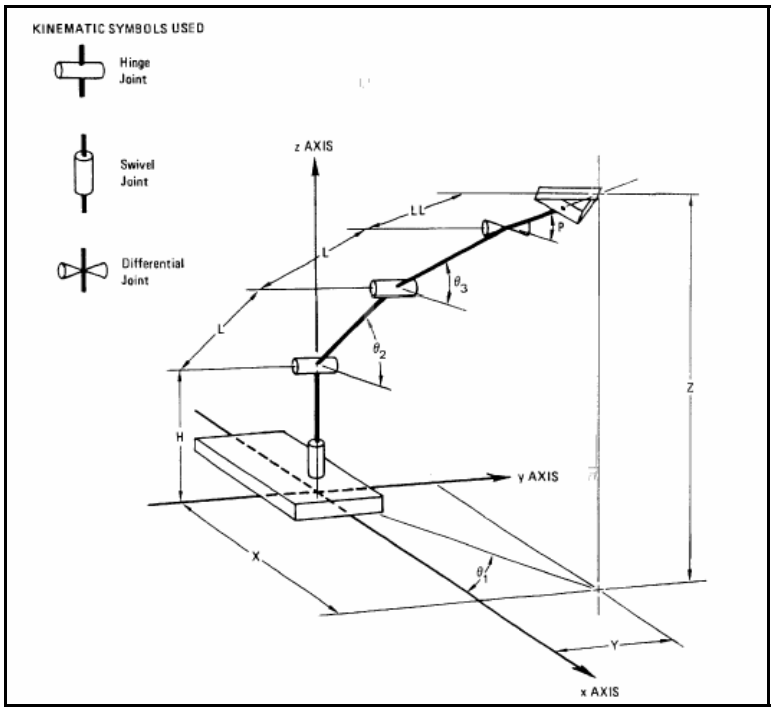


Figura 13 - Geometría de las articulaciones del brazo de robot

La solución a este problema radica en establecer un modelo geométrico basado en sistemas de ecuaciones. El procedimiento en sí se basa en encontrar el suficiente número de relaciones geométricas en las que intervengan las coordenadas del extremo del robot, sus coordenadas articulares y las dimensiones físicas de sus elementos. Se debe considerar el extremo del robot como el punto más alejado de la Pinza. Este punto corresponde al punto medio del extremo de la Pinza.

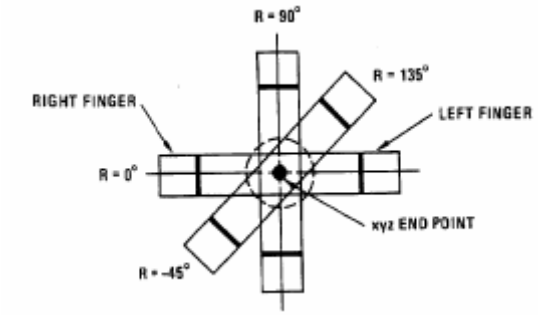


Figura 14 - Diferentes rotaciones sobre el mismo punto final

Para mostrar el procedimiento a seguir se va a aplicar en primer lugar, el método a la resolución del problema cinemático inverso del brazo de robot considerando las cuatro primeras articulaciones: Base, Hombro y Codo y Muñeca. El dato de partida son las coordenadas (X, Y, Z) del punto a donde se quiere posicionar el extremo del robot, y la inclinación vertical de la muñeca (P).

La Base del robot se encuentra en el plano horizontal, por lo que el ángulo de la primera variable articular Θ1está determinado por la distancia del punto extremo del robot sobre los dos ejes X e Y que definen el plano horizontal. El valor del ángulo de la Base Θ1 se obtiene inmediatamente como:

Θ1 = arctan (Y / X),

y utilizando el teorema del coseno establecemos en el mismo plano la distancia ente la base y el punto que se quiere alcanzar como:

RR = $\sqrt{X^2 + Y^2}$

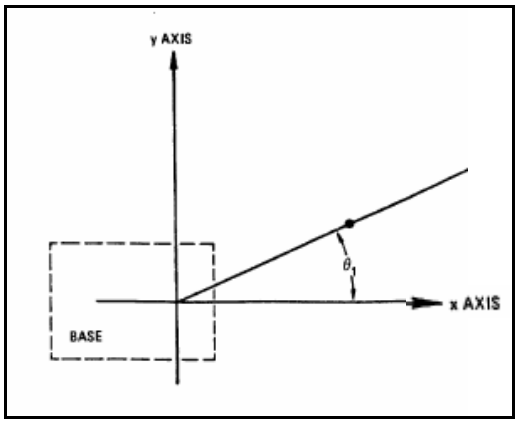


Figura 15 - Geometría parcial de la Base en el plano horizontal

Considerando ahora únicamente el Hombro, el Codo y la Muñeca que están situados en el mismo plano vertical definido por el eje Z, se obtiene que la distancia Ro entre el eje del Hombro y la Muñeca viene determinada como:

Ro = RR - LL * cos(P),

Y de la misma forma, la altura (Zw) de la muñeca respecto al plano horizontal de la Base viene determinada por:

Zw = Z - LL * sin(P), además podemos establecer la altura (Zo) de la muñeca respecto al plano horizontal del Hombro como:

Zo = Z - LL * sin(P) - H

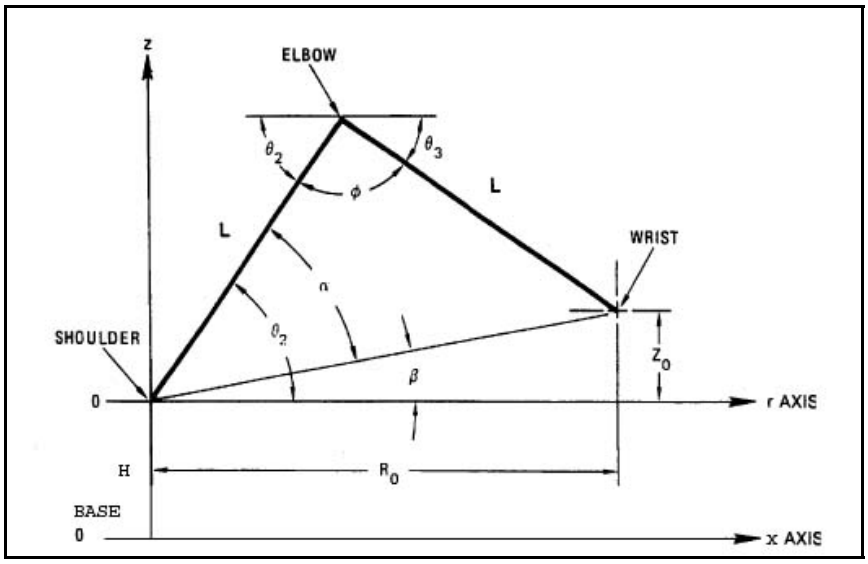


Figura 16 - Geometría parcial del brazo de robot

A partir de la de la distancia (Ro) y la altura (Zo) se puede obtener los valores de los ángulos para la segunda y tercera variable articular. Para establecer el valor de los ángulos del Hombro y del Codo se requiere la ayuda de tres nuevos ángulos (α, β, ϕ) para simplificar las expresiones matemáticas. Los valores de α, β y ϕ vienen determinados por las siguientes igualdades las cuales no se detallan por su alta complejidad.

$\alpha = \arctan\left(\sqrt{(4L^2/(Ro^2 + Zo^2) - 1)}\right)$

$\beta = \arctan(Zo / Ro)$

$\phi = pi - 2\alpha$

A partir de estos nuevos ángulos se puede deducir que el ángulo del Hombro es igual a:

$\Theta 2 = \alpha + \beta$

Y también que para el Codo,

$\Theta 3 + \phi + \Theta 2 = pi,$

por lo que:

$\Theta 3 = \alpha - \beta$

Seguidamente definiremos las igualdades que caracterizan los valores articulares que caracterizan la inclinación vertical (P) y la rotación (R) de la muñeca respecto la posición del brazo de robot. Estos valores vienen determinados por dos ángulos acoplados Θ4 y Θ5 (derecho e izquierdo respectivamente) que definen la orientación con la cual la herramienta se

posicionará en la posición final (X, Y, Z). Por este motivo incorporamos un nuevo parámetro R a los datos de entrada.

Partiendo de que el ángulo de la inclinación vertical y de rotación vienen definidos por:

$$P = 0.5 (\Theta 5 + \Theta 4)$$

$$R = 0.5 (\Theta 5 - \Theta 4)$$

Resolviendo este sistema de dos ecuaciones y dos incógnitas obtenemos finalmente que:

$$\Theta 5 = P + R$$

$$\Theta 4 = P - R$$

De esta forma se establece una relación entre las coordenadas cartesianas X, Y, Z y la orientación de la herramienta, con los ángulos articulares para cada una de las articulaciones. Además, como cada una de las articulaciones está conducida por un motor, los parámetros de pasos del comando STEP (J1, J2, J3, J4, J5, J6) son directamente proporcionales a los valores de cada uno de los ángulos articulares (ver Tabla 8)

Motor	**Articulación**	**Pasos / Radian**
1	BASE	1125
2	HOMBRO	1125
3	CODO	661.2
4	MUÑECA Derecha	244.4
5	MUÑECA Izquierda	244.4
Motor	**Articulación**	**Pasos / Pulgada**
6	PINZA	371

Tabla 8 - Factores de conversión entre pasos y ángulos articulares

Finalmente, se añade a los datos de entrada un parámetro I, que especificará cuan abierta, en pulgadas, va a estar la pinza. Este valor viene determinado por el factor de conversión de la Pinza.

Capítulo 4

ARQUITECTURA PRISMA PARA EL CASO DE ESTUDIO

CONTENIDOS

ARQUITECTURA PRISMA PARA EL CASO DE ESTUDIO

4.1 Introducción

La definición de la arquitectura de un sistema se ha de realizar siguiendo unos criterios específicos para encontrar los elementos arquitectónicos de éste. Los elementos arquitectónicos PRISMA [Per03] se obtienen identificando escenas funcionales del sistema y asignando a cada elemento una escena funcional. Dependiendo de si la escena es simple o compleja, el elemento será un componente o un sistema, respectivamente. El concepto de *escena* aparece en el trabajo de [Nor98]. De esta manera, el enfoque PRISMA considera la escena como un criterio para la detección de componentes en la especificación de requisitos. El criterio consiste en que cada escena que se detecte en la especificación de requisitos se corresponde con una componente del sistema. Para realizar dicha detección de forma adecuada es necesario definir el concepto de escena. El concepto de escena dentro de un sistema de información esta vagamente definido. Es por este motivo, que a continuación se propone una definición.

<<Una escena es una actividad relevante dentro de un sistema de información. Una escena se caracteriza por las tareas que se desempeñan en la actividad siguiendo un determinado protocolo, los actores que las realizan y el espacio virtual o físico donde se desarrolla.>>

Este enfoque para la detección de componentes, es compatible con el *Architecture Based Design Method* (ABD) [Bac00], ya que este método se basa en la descomposición funcional del problema, es decir del sistema de información. ABD es una metodología propuesta por el SEI (*Software Engineering Institute of The Carnegie Mellon University*) para diseñar arquitecturas *software* de un dominio de aplicación o de una familia de productos.

Siguiendo esta metodología se van a identificar los distintos elementos arquitectónicos del Robot *TeachMover* [Pas04]. La identificación funcional se puede realizar de un gránulo funcional mayor a más fino (descendente) o de nivel más simple al más complejo (ascendente). En el caso del Robot *TeachMover* se va a realizar una identificación ascendente, ya que resulta más sencillo para el analista.

La arquitectura del robot va a ser especificada mediante UML 2.0. [UML04]. A pesar de que no existe su versión definitiva y que las herramientas de modelado no incorporan todos sus nuevos conceptos, se va a utilizar, para diseñar los elementos arquitectónicos del robot *TeachMover*, la herramienta Poseidon [Pos04], que ya proporciona componentes y puertos UML 2.0.. Se ha de hacer notar que los puertos en PRISMA son bidireccionales y por lo tanto,

no se va a hacer diferencia entre entrada y salida y se asume que todo puerto PRISMA tiene una interfaz UML 2.0 provista y una requerida.

4.2 Identificación de Componentes Simples

El nivel más bajo del robot tiene como componentes básicos los sensores y los actuadores. Ambos componentes son los que van a hacer de interfaz con el hardware, para el caso específico del robot *TeachMover*. Los actuadores son los encargados en mandar órdenes a las articulaciones o a la herramienta del robot, para que procesen la acción que éstos especifican (ver Figura 17 para el caso de la herramienta). Mientras que los sensores realizan la lectura del resultado de dichas acciones, para saber si estas se han ejecutado de forma correcta. Se ha de tener en cuenta que las órdenes y lecturas de ambos componentes se enviarán y recibirán a través de un puerto de entrada/salida, es decir, un puerto con una interfaz provista y una requerida.

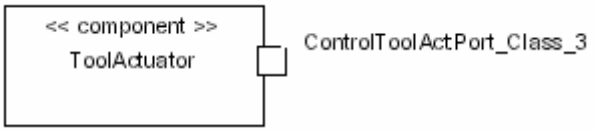


Figura 17 - Actuador

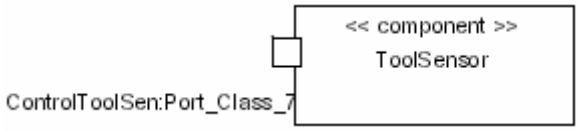


Figura 18 - Sensor

En el sistema tendremos tantos actuadores y sensores como articulaciones tienen el robot más la herramienta. De forma que cada par actuador-sensor ha de estar coordinado por un conector, con el objetivo de separar su interacción de su funcionalidad (ver Figura 19 para el caso de la herramienta). Dicho conector dispondrá de dos roles para comunicarse con cada uno de los componentes que sincroniza y un tercer rol para establecer los *bindings* con el nivel de granularidad superior.

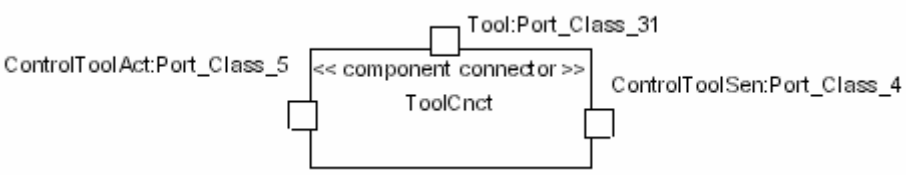


Figura 19 - Conector Sensor-Actuador

4.3 Identificación de Simple Unit Controllers (SUCs)

Los *simple unit controllers* (SUCs) van a formar la segunda capa de la arquitectura (ver Figura 26). Cada SUC va a agrupar el sensor, el actuador y el conector de cada una de las articulaciones del robot o de la herramienta. Se ha de tener en cuenta que los SUCs pueden ser *software* o hardware (por ejemplo: tarjetas), pero en nuestro caso van a ser *software*, ya que se va a implementar la arquitectura completa del robot.

En el modelo arquitectónico del robot *TeachMover*, cada uno de estos SUCs va a ser un sistema PRISMA de primer nivel de composición. Debido a que no interesa acceder directamente al actuador y al sensor del SUC, el puerto del SUC esta conectado con el rol del conector, con el objetivo que este sea el que sincronice la acción de ambos y las órdenes se ejecuten adecuadamente. Los SUCs identificados para el robot son tantos como articulaciones tiene y uno más para manipular la herramienta del robot (ver Figura 21). Estos son los siguientes: *Base, Shoulder , Elbow, Wrist,* y *Tool*.

Figura 20 - Robot *TeachMover*

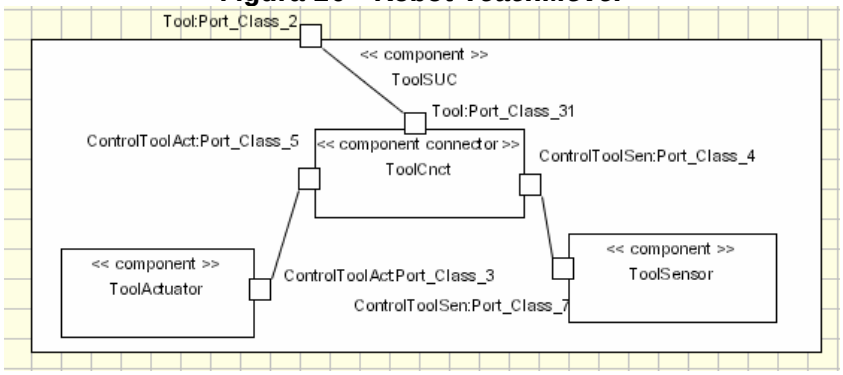


Figura 21 - SUC de la herramienta

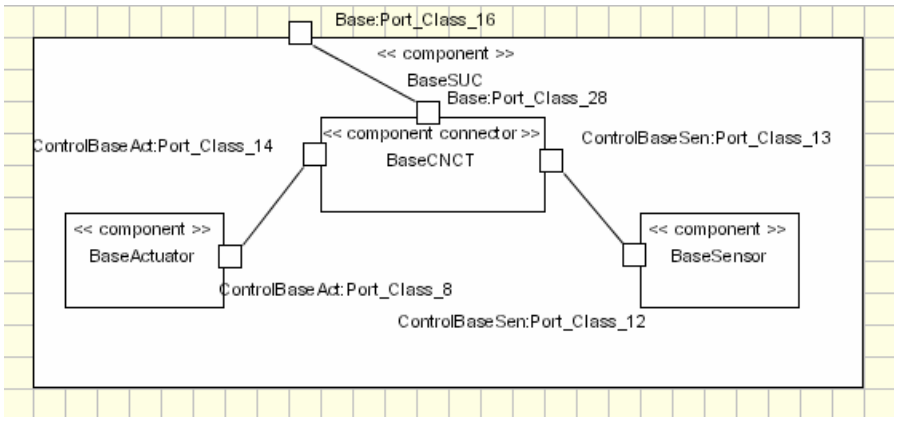


Figura 22 - SUC de la base

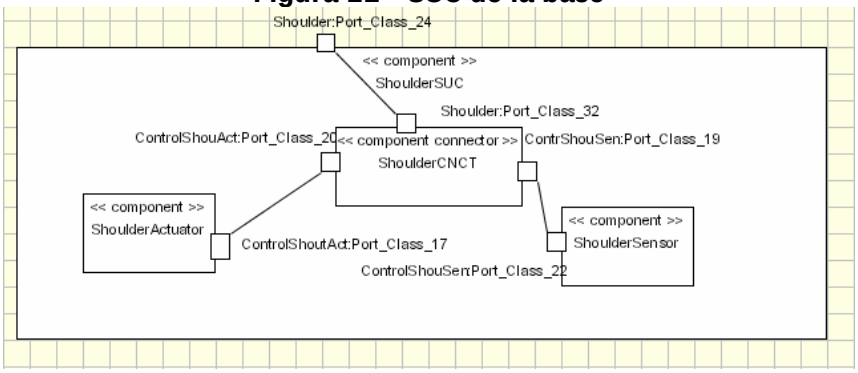


Figura 23 - SUC del shoulder

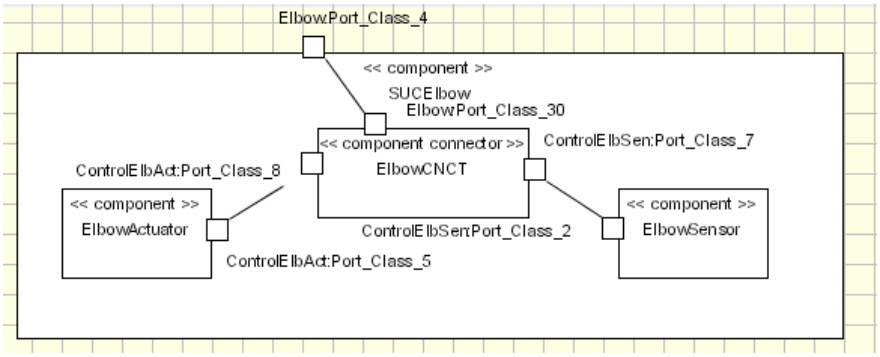


Figura 24 - SUC del elbow

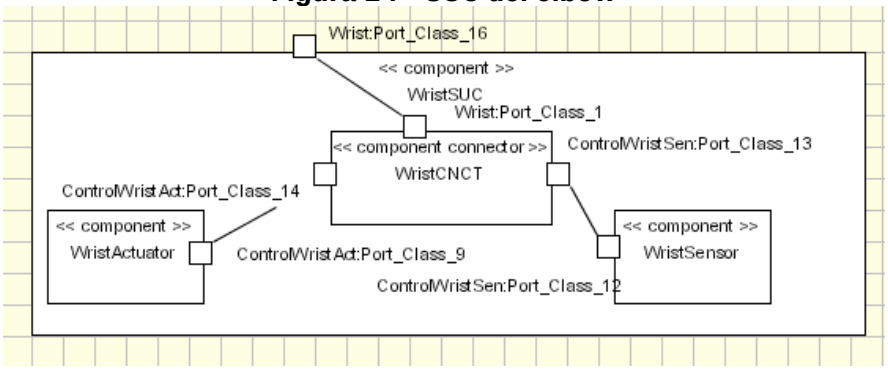


Figura 25 - SUC del wrist

NIVEL 2	**SUCs**
NIVEL 1	**Actuadores y Sensores**

Figura 26 - Dos niveles de abstracción de la arquitectura PRISMA del robot *TeachMover*

4.4 Identificación de Mechanism Unit Controllers (MUCs)

Los MUCs van a formar la tercera capa de la arquitectura (ver Figura 27). Éstos se caracterizan por el hecho de coordinar un conjunto de SUCs a través de uno o más conectores. De esta forma, un MUC permite controlar un mecanismo completo para conseguir un objetivo común (por ejemplo: la coordinación de un conjunto de articulaciones para mover un brazo robot).

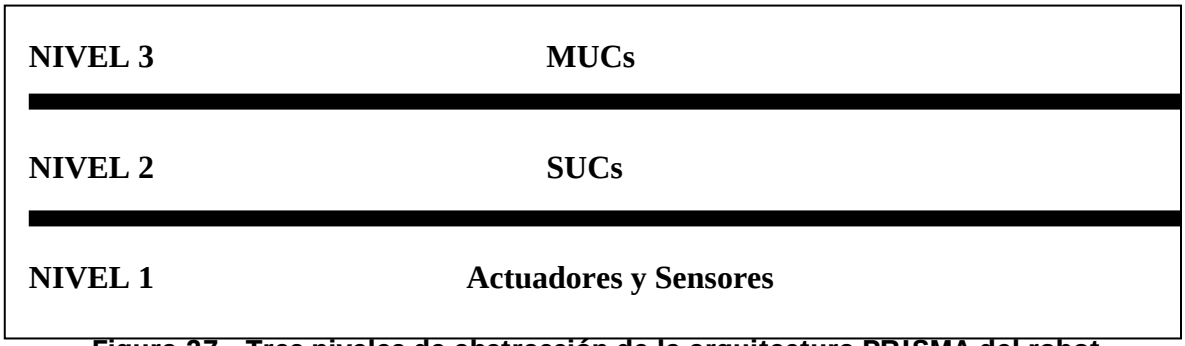


Figura 27 - Tres niveles de abstracción de la arquitectura PRISMA del robot *TeachMover*

Un MUC dentro del modelo PRISMA va a ser un sistema (componente complejo) que integra un conjunto de componentes (simples o complejas) y conectores con un objetivo funcional común (escena).

En el caso del robot *TeachMover*, se va a definir un MUC que incorpore todos los SUCs definidos en el apartado anterior, excepto el SUC herramienta. Esta decisión arquitectónica es una característica del dominio, ya que la herramienta se modela como un elemento arquitectónico independiente del resto del robot. Esto es debido a que al mismo robot se le pueden incorporar distintas herramientas para desempeñar actividades diferentes (limpieza, chorreo, pintura, etc.), incluso a mitad de la ejecución de una tarea, a diferencia de las articulaciones que no pueden sufrir cambios en tiempo de ejecución. Por ejemplo, en el caso del robot *TeachMover*, la herramienta de la que se dispone es de una pinza, sin embargo ésta puede variar por una brocha, manguera, etc. Por lo tanto, se va a describir un MUC que se corresponde con el brazo robot del *TeachMover*. Dicho MUC está compuesto por los SUCs *Base*, *Shoulder*, *Elbow* y *Wrist* y un conector que los va a coordinar con el objetivo común de mover el brazo (ver Figura 28).

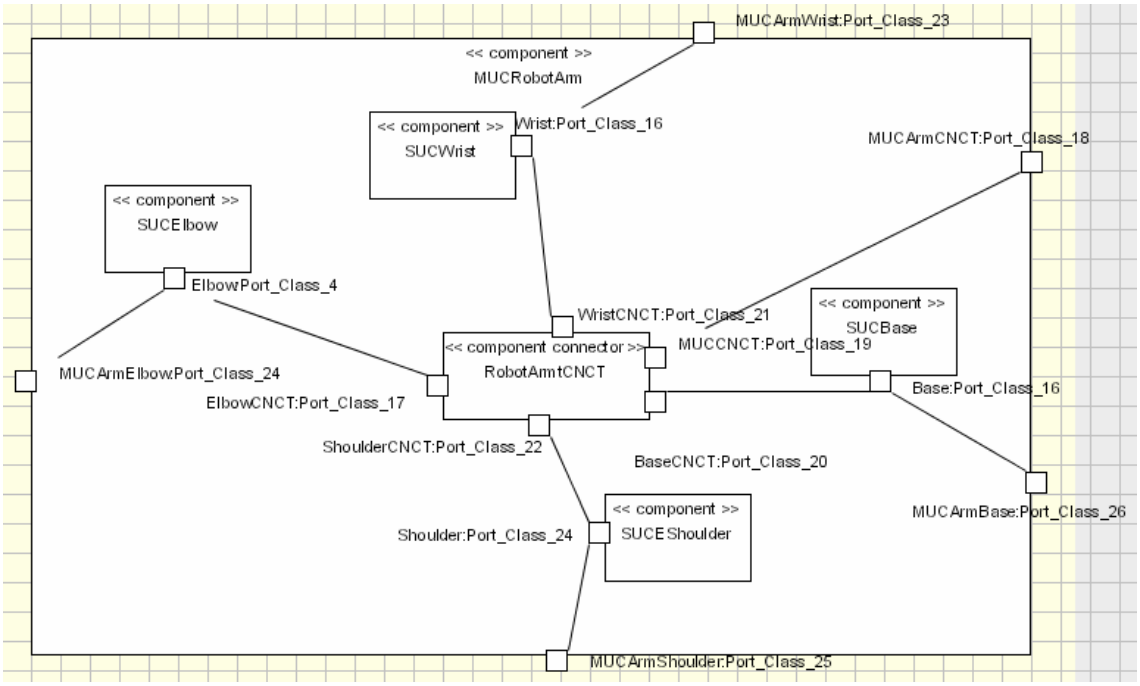


Figura 28 - MUC del brazo robot

Se ha de hacer notar que el MUC se conecta tanto con el conector como con cada uno de los SUCs que lo forman. De esta manera, desde el nivel superior se puede acceder directamente a cada uno de los SUCs sin tener que pasar por el conector (necesario en situaciones de emergencia donde hay que parar la articulación en el mínimo tiempo posible), o bien, se puede acceder a través del conector para ejecutar un comando de movimiento y que éste sincronice a todos los SUCs. Esto implica la necesidad de establecer prioridades de ejecución a los distintos clientes del SUC. Este orden de ejecución se podrá definir en el protocolo asociado al puerto del SUC, el cual concederá prioridad a las órdenes directas del MUC, ya que serán originadas por paradas de emergencia, y posteriormente, se atenderán las órdenes del conector.

4.5 Identificación de Robot Unit Controllers (RUCs)

Los RUCs van a formar la cuarta capa de la arquitectura (verFigura 29), éstos se caracterizan por el hecho de coordinar un robot completo a través de uno o más conectores. Los RUCs son los que nos permiten realizar tareas completas en las que se coordinan todas y cada una de las partes que conforman el dispositivo robot. Es por este motivo que un RUC dentro del modelo PRISMA va a ser un sistema (componente complejo) que integra un conjunto de componentes (simples o complejos) y conectores con el objetivo de que un robot desempeñe sus tareas.

En el caso del robot *TeachMover*, se va a definir un RUC que incorpore el SUC de la herramienta, el MUC del brazo robot y un conector que los sincronice con el objetivo de mover el robot completo. Por lo tanto, se va a describir un RUC que se va corresponder con el robot *TeachMover* (ver Figura 30). El RUC será el encargado de realizar la cinemática inversa del robot y de calcular y coordinar las distintas velocidades de los movimientos de cada SUC en aquellos casos en los que se desee que todas las articulaciones acaben a la misma vez cuando participan en un movimiento común.

NIVEL 4	**RUCs**
NIVEL 3	**MUCs**
NIVEL 2	**SUCs**
NIVEL 1	**Actuadores y Sensores**

Figura 29 - Cuatro niveles de abstracción de la arquitectura PRISMA del robot *TeachMover*

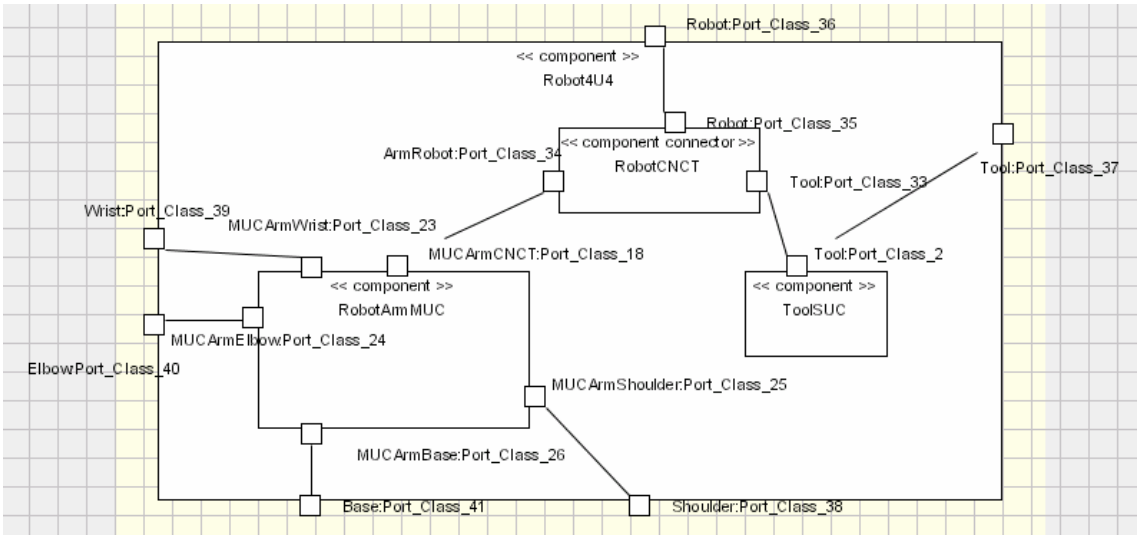


Figura 30 - RUC robot *TeachMover*

Se ha de hacer notar que, al igual que el MUC, el RUC esta conectado tanto con el conector como con el SUC y el MUC, y con este último no sólo lo realiza por el puerto del conector, sino también con cada uno de los puertos que se comunican con los SUCs que lo componen. De esta forma, se puede acceder directamente a cada uno de los SUCs sin tener que pasar por el conector, o bien, se puede acceder a través del conector para ejecutar un comando de movimiento y que éste sincronice a todos los SUCs. Del mismo

modo que en el MUC, en el RUC se ha de establecer un orden de ejecución entre las tareas requeridas por el conector y por él mismo al SUC de la herramienta y al MUC del brazo robot. Esta gestión de prioridades se establecerá en los puertos del SUC y del MUC, dando prioridad a las órdenes requeridas por el RUC, ya que se solicitan en situaciones de emergencia.

4.6 Modelo Arquitectónico Final

Finalmente, nos encontramos con la última capa del modelo arquitectónico del sistema de información (ver Figura 31). Dicha capa modela tanto el robot como el entorno en el que éste se desenvuelve. En robots complejos el número de componentes complejos que formen el entorno será mucho mayor (sistemas de visión, dispositivos inalámbricos, obstáculos, etc.). Sin embargo, el caso del robot *TeachMover* es sencillo, ya que su entorno es una mesa de laboratorio. Pero esto no evita el tener que especificar las variables de entorno aunque sean constantes en todo momento y no sean susceptibles de variación alguna. Es por este motivo, que en el modelo arquitectónico del robot *TeachMover* se han incluido las componentes simples entorno y operario, además del propio robot. Por otro lado, también se ha incluido un conector que controla la ejecución de comandos que ordena el operario y comprueba que éstos sean válidos en base a las características del entorno y a la posición del robot (ver Figura 32).

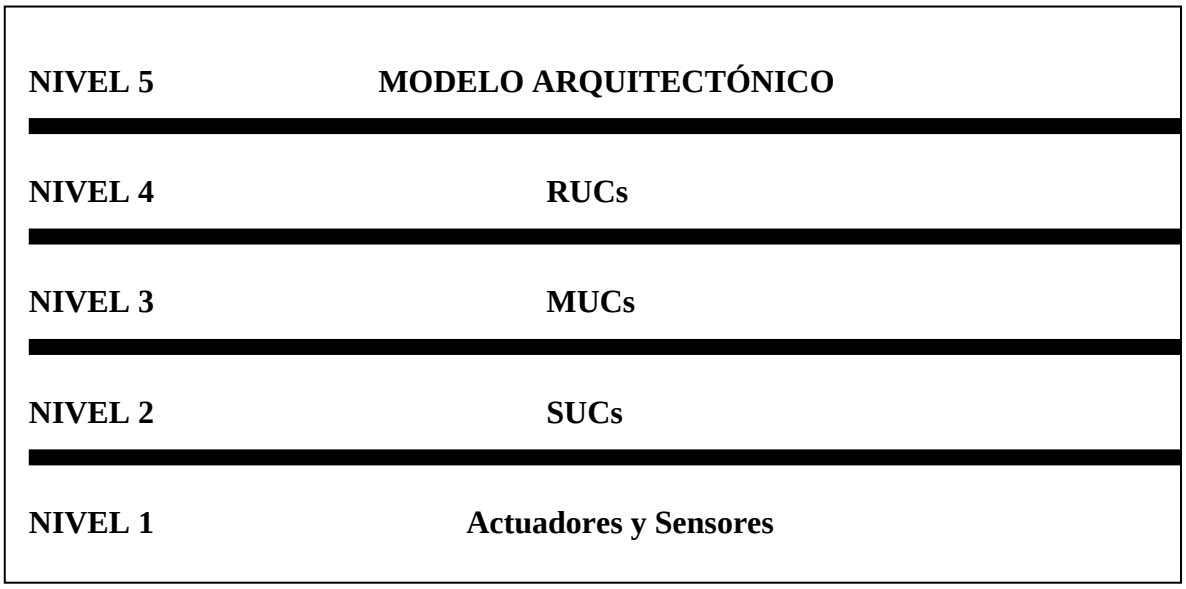


Figura 31 - Cinco niveles de abstracción de la arquitectura PRISMA del robot *TeachMover*

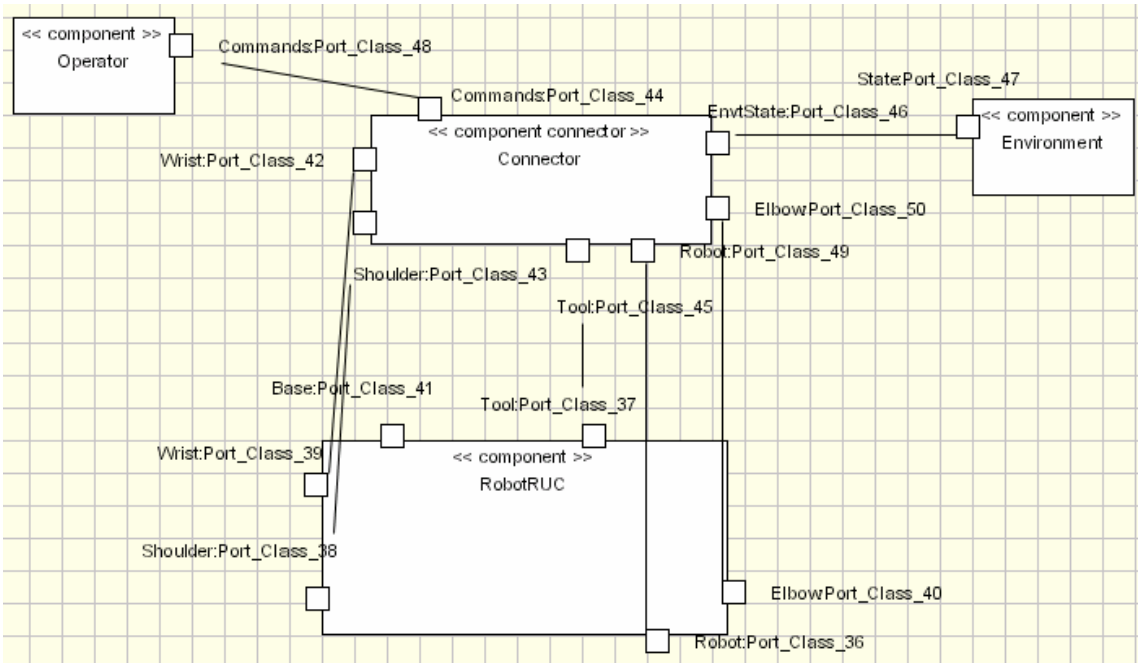


Figura 32 - Modelo arquitectónico para el sistema de información del robot *TeachMover*

4.7 Identificación de Concerns

Una vez ya diseñado el modelo arquitectónico del robot *TeachMover*, es necesario identificar los *concerns* comunes al sistema (*crosscutting concerns*) con el objetivo de separar cada uno de ellos en una entidad reutilizable llamada aspecto. Tras el análisis del sistema del robot *TeachMover*, se puede observar que su función es la de mover el robot, bien sea por pasos simples o mediante el uso de cinemática inversa (moverse a un punto determinado (x, y, z) con unos ángulos de rotación específicos para los movimientos *pitch* y *roll* (p, r) de la muñeca y las pulgadas (*inches*) de apertura de la pinza (i)). Esto hace aparecer en el sistema el *concern* funcional.

Es necesario conocer cómo se realizan los movimientos del robot *TeachMover* para comprender las definiciones de cada uno de los aspectos que se van a especificar en el siguiente apartado. Concretamente, el movimiento más completo y no individualizado del robot, es el movimiento por cinemática inversa. Este movimiento, aunque se solicite a un determinado punto (x,y,z,p,r,i), tal y como se ha explicado anteriormente, dicho punto se ha de traducir a un conjunto de ángulos. Cada uno de ellos para cada una de las articulaciones que conforman el robot. Estos ángulos indican los grados que ha de formar cada articulación respecto al eje de coordenadas de referencia (ver Figura 33) del robot. De forma que, cuando cada articulación forme el ángulo indicado, el robot alcance la posición deseada por el operario.

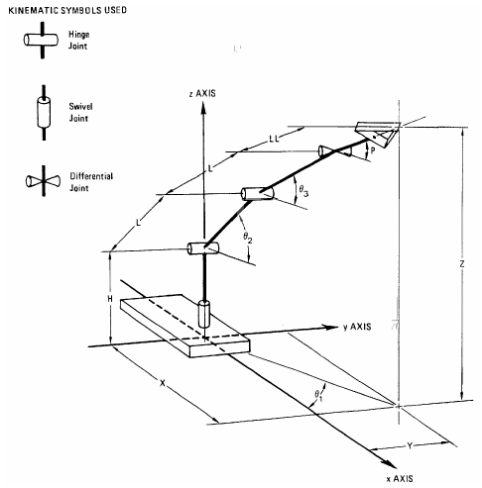


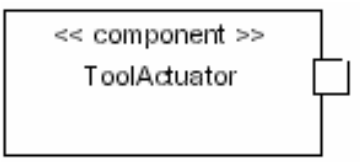
Figura 33 - Eje de coordenadas de referencia del robot

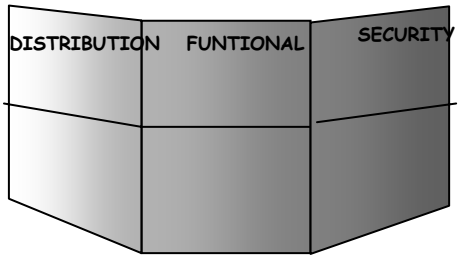
Por otro lado, los movimientos del robot los ordena un operario por control remoto, ya que es necesario que se sitúe en un lugar alejado del robot para evitar situaciones de riesgo. Estas propiedades hacen emerger un *concern* distribución, que cobrará más relevancia y complejidad en sistemas robóticos más grandes como el sistema tele-operado EFTCoR [EFTCoR].

Un *concern* de gran relevancia en este caso de estudio es la seguridad, ya que sus propiedades son de vital importancia para establecer el control seguro de los movimientos del robot. De forma que éstos sean correctos, puedan ejecutarse y ni el robot, ni cualquier otro elemento que forme parte del entorno corra peligro ante sus movimientos.

Finalmente, cabe destacar la necesidad de establecer los movimientos de todas y cada una de las articulaciones de forma coordinada. Esta sincronización de movimientos hace necesaria la aparición de un cuarto *concern* llamado coordinación.

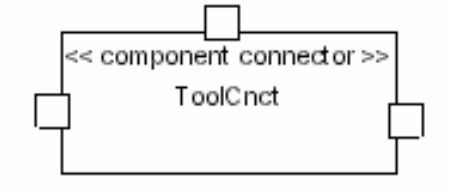
Como resultado de este análisis se han identificado los *concerns*: funcional, distribución, seguridad y coordinación. De forma que se van a separar cada *concern* en distintos aspectos para mejorar su reutilización y mantenibilidad de la arquitectura. Por lo tanto, un componente o sistema PRISMA del modelo arquitectónico del robot *TeachMover* puede estar formado por un aspecto funcional, un aspecto de distribución y uno de seguridad. Así mismo, los conectores que los coordinan pueden estar formados por un aspecto de coordinación, otro de distribución y uno de seguridad. Por lo tanto, la vista interna (*white* box) y externa (*black* box) de componentes (ver Figura 34) y conectores (ver Figura 35) que contengan todos los aspectos se presentan a continuación. Pero se ha de tener en cuenta que no todos los componentes y conectores van a incorporar todos los aspectos posibles, solamente los incorporarán aquellos que los necesiten.

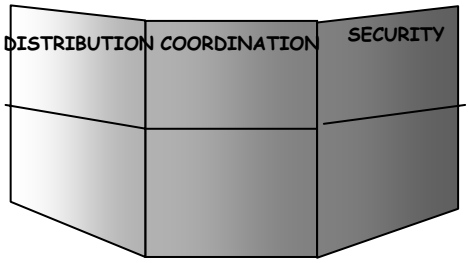




Black Box View	*White Box View*

Figura 34 - Vista interna y externa de los componentes prisma del modelo arquitectónico robot *TeachMover*





Black Box View	*White Box View*

Figura 35 - Vista interna y externa de los conectores prisma del modelo arquitectónico robot *TeachMover*

4.8 Especificación de SUCs

En este apartado se van a especificar los aspectos, identificados en el apartado anterior, de cada una de las componentes y conectores del modelo arquitectónico. Al igual que se ha procedido para el diseño del modelo arquitectónico, los aspectos se han especificado de forma ascendente, partiendo del nivel más bajo de abstracción y acabando en el más alto. Es por este motivo que primero se van a diseñar los aspectos necesarios para definir los SUCs, y posteriormente, se irán definiendo para el resto de componentes de mayor granularidad del sistema.

Los aspectos que son necesarios para especificar un SUC son los aspectos de coordinación, distribución y seguridad del conector y los aspectos de funcionalidad y distribución del actuador y el sensor. Estos se presentan a continuación.

4.8.1 Aspectos SUC

4.8.1.1 Aspecto de Coordinación

El actuador de una articulación es aquel que permite el movimiento de ésta. Cada una de las articulaciones del robot *TeachMover* se pueden mover de

dos formas distintas: por pasos y moviéndose a un punto específico del espacio (cinemática inversa). El movimiento por pasos, consiste en decirle un número que se corresponde con la cantidad de medios pasos (*halfsteps*) que se va a desplazar dicha articulación. Sin embargo, el movimiento por cinemática inversa se le dan los grados del ángulo que ha de formar la articulación para alcanzar el punto (x,y,z) al que se desea mover el robot. Ambos tipos de movimiento se realizan a una velocidad y que puede variar según se desee. El control de los movimientos de la articulación se debe centralizar en la coordinación que realiza el conector de ésta (ver Figura 22 para el caso de la base), el cuál sincroniza los servicios para que los movimientos puedan realizarse de forma correcta. Se ha de tener en cuenta que aunque a la articulación se le puedan solicitar ambos movimientos, el actuador siempre le indica al robot el movimiento solicitado en *halfsteps*. Esto implica que el conector, antes de enviar al actuador el movimiento, ha de realizar el cambio de ángulos a *halfsteps* cuando el movimiento se le ha solicitado mediante cinemática inversa.

Cada articulación tiene almacenado el ángulo en el que se encuentra respecto a la posición del eje de coordenadas de referencia y el número de *halfsteps* equivalente a dicho ángulo. Como el robot se mueve *halfsteps* y su valor equivalente en ángulos es posible obtenerlo mediante una fórmula de conversión, el ángulo se va a especificar como un atributo derivado (derivado) del atributo variable (*variable*) *halfsteps*. De esta forma, dado un movimiento de la articulación su valor en *halfsteps* se deben actualizar y por consiguiente, el ángulo. El hecho de que el valor del atributo *halfSteps* varíe hace que sea de carácter variable (*variable*), sin embargo si su valor no se pudiera modificar sería de carácter constante (*constant*). El hecho de que la posición se gestione en dos formatos diferentes es para facilitar los cálculos de los movimientos del robot. Dentro del aspecto de coordinación del conector de cada uno de los SUCs del modelo, será necesario un atributo para poder calcular el ángulo y otro para almacenar los *halfsteps*. Su especificación se muestra a continuación:

```
  Attributes
    Variable
      halfSteps: integer,
          NOT NULL;
  Derived
      angle:= FtransHalfstepsToAngle(halfSteps);
```

Se ha de hacer notar que el atributo *halfSteps* además del tipo, se le ha dado la propiedad NOT NULL. Esto significa que el atributo es requerido, es decir, que durante la ejecución del aspecto debe tener un valor asociado, nunca puede tener valor nulo.

En el caso de la muñeca serán necesarios dos atributos más para indicar los grados del movimiento *roll* y del movimiento *pitch*. Igualmente, se procede con los *halfsteps*.

```
  Attributes
    Variable
      leftHalfSteps: integer,
                     NOT NULL;
      rightHalfSteps: integer,
                     NOT NULL;
    Derived
```

```
       pitchAngle:= FtransHalfstepsToAngle(leftHalfSteps);
       rollAngle:= FtransHalfstepsToAngle(rightHalfSteps);
```

Los movimientos por pasos y por cinemática inversa se van a realizar mediante dos servicios de moviendo diferentes, *moveJoint* y *cinematicsMoveJoint*, respectivamente. En el caso del *moveJoint*, se le ha de pasar como parámetro el número de *halfsteps* que se desea que se mueva la herramienta y a qué velocidad, mientras que en el caso del *cinematicsMoveJoint* se va a tener que pasar el ángulo exacto a donde se quiere desplazar y la velocidad con que se realice dicho desplazamiento.

```
moveJoint(input NewHalfSteps: integer, input Speed: integer);
cinematicsMoveJoint(input NewAngle: integer, input Speed: integer);
```

Cuando el conector de un SUC invoca unos de estos servicios de movimiento, no implica que la articulación del robot realmente se mueva, a pesar de que el movimiento solicitado sea correcto. El conector solamente tendrá la certeza de que el robot se ha movido cuando el sensor le notifique que el movimiento se ha realizado satisfactoriamente. En ese momento, será cuando el conector modifique el estado de la posición de la articulación correspondiente. La notificación se realiza mediante un servicio que devuelve 0 o 1, dependiendo de si el movimiento se ha realizado o no, respectivamente. La sintaxis del servicio es la siguiente:

```
moveOk(output Success: [0,1]);
```

Esto implica, que la valuación asociada al servicio de movimiento de la articulación (*movejoint*, *cinematicsmovejoint*) no puede actualizar directamente el valor de la posición, y por lo tanto, es necesario tener unos atributos temporales en los que almacenar el movimiento solicitado. Dichos atributos serán asociados posteriormente a los atributos definitivos en la valuación del servicio *moveOK*, que es el que asegura que el movimiento se ha producido.

Además de estos servicios, existe un servicio de parada de emergencia de la articulación. Dicho movimiento es stop y su sintaxis es la siguiente:

```
stop();
```

Se ha de tener en cuenta que los servicios anteriormente expuestos han de publicarse mediante interfaces que se reutilizaran en diferentes puertos y aspectos:

```
Interface ImotionJoint
   moveJoint(input NewHalfSteps: integer, input Speed: integer);
   stop();
End_Interface IMotionJoint;

Interface Iread
    moveOk(output success: [0,1]);
End_Interface IRead;

Interface ISUC
  moveJoint(input NewHalfSteps: integer, input Speed: integer);
  cinematicsMoveJoint( input NewAngle: integer, input Speed: integer);
  stop();
  moveOk(output success: [0,1]);
```

```
End_Interface ISUC;
```

Por otro lado, los servicios *begin* y *end*, propios del modelo PRISMA, son necesarios para la correcta especificación del aspecto. Ambos servicios definen el comienzo y el fin de la ejecución del aspecto al que pertenece. La ejecución del servicio *begin* es desencadenada por el servicio que crea la instancia del elemento arquitectónico que importa el aspecto al que pertenece el servicio *begin*. Así mismo, la ejecución del servicio *end* es desencadenada por el servicio que destruye la instancia del elemento arquitectónico que importa el aspecto al que pertenece el servicio *end*. El servicio *begin*, además de definir el comienzo de la ejecución del aspecto al que pertenece, proporciona el valor que han de tener los atributos requeridos del aspecto. Por ejemplo, el servicio *begin* del aspecto de coordinación pasa como argumento el atributo requerido *HalfSteps*.

```
begin (input InitialHalfSteps: integer);
```

En base al análisis anterior se construye el aspecto de coordinación del conector del sistema SUC. Si bien es cierto, queda por detallar los procesos de coordinación del aspecto. Existen dos tipos de procesos de coordinación, los subprocesos, que describen el comportamiento de cada uno de los componentes que están conectados al conector desde su perspectiva y el protocolo global que ordena estos subprocesos. El aspecto de coordinación define tres subprocesos, el del actuador (ACT), el del sensor (SEN) y el de la articulación (SUC). Por ejemplo, en el caso del subproceso SUC (ver especificación del aspecto que se presenta a continuación), se especifica que a la articulación se le puede solicitar tres tipos de movimiento: moverse por *halfsteps* (*moveJoint*), moverse por ángulos (*cinematicsMoveJoint*) o que se pare (stop). La solicitud de los dos tipos de movimiento no tiene un orden (|), o se ejecuta el uno o el otro, independientemente del orden; mientras que la parada tiene prioridad respecto a los servicios de movimiento. Para tener una gestión de prioridades, se indica la prioridad 0 o 1 seguido de los parámetros, siendo 0 la prioridad máxima y 1 la prioridad normal (stop?():0, resto_de_servicios?():1). Se ha de tener en cuenta, que cuando no se especifica la prioridad se asume que la prioridad es normal, es decir, 1. Finalmente, una vez solicitado cualquiera de estos servicios, el SUC recibirá la contestación de la correcta o incorrecta ejecución del movimiento o la parada (moveok), para posteriormente notificarlo al nivel superior MUC. Ésta ordenación se establece mediante la secuencia siguiente: ((<orden_servicios(stop, movejoint, cinematicsmovejoint)>) → moveok).

Como resultado final, el aspecto de coordinación que se obtiene para los conectores de los SUCs es el siguiente:

```
Coordination Aspect CProcessSUC using IMotionJoint, IRead, ISUC

   Attributes
     Variable
       halfSteps: integer,
```

```
          NOT NULL;
       tempHalfSteps: integer;

     Derived
       angle:= FtransHalfstepsToAngle(halfSteps);


  Services
      begin (input InitialHalfSteps: integer);
          Valuations
[begin (InitialHalfSteps)]
   halfSteps := InitialHalfsteps,

in/out movejoint(input NewHalfsteps: integer, input Speed: integer);
          Valuations
[in movejoint (NewHalfsteps, Speed)]
   tempHalfSteps := NewHalfsteps;


      in cinematicsmovejoint( input NewAngle: integer, input Speed:
integer);
          Valuations
[in cinematicsmovejoint(NewAngle, Speed)]
 tempHalfSteps := FtransAngleToHalfsteps(NewAngle);

in/out stop();

in/out moveok(output Success:[0,1]);
           Valuations
{Success=1}
[in moveok(Success)]
halfSteps:= halfSteps + tempHalSteps;
       end;

    Subprocesses

SUC= (ISUC.stop?():0
     +
     ISUC.moveJoint?(NewHalfsteps, Speed):1
     +
            ISUC.cinematicsMoveJoint?(NewAngle, Speed):1
            )
            →
            ISUC.moveOk!(Success);

ACT= ImotionJoint.stop!():0
     +
     IMotionJoint.moveJoint!(NewHalfsteps, Speed):1;

      SEN= Iread.moveOk(Success);

   End_Subprocesses;

   Protocol
CPROCESSSUC = begin(InitialHalfStep).MOTION;
MOTION = SUC.stop ?()).EMERGENCY
         +
         (SUC.movejoint?(NewHalfsteps, Speed)
   →
         ACT.movejoint!(NewHalfsteps, Speed).ANSWER)
         +
```

```
               (SUC.cinematicsmovejoint ?(NewAngle, Speed)
                →
                ACT.movejoint ! (FTransAngleToHalfSteps(NewAngle),
Speed)
                .ANSWER)
                +
                end;

      ANSWER= SEN.moveok ? (Success) → SUC.moveok!(Success).MOTION

     EMERGENCY= ACT.stop ! ()→ end;

End_Coordination Aspect CProcessSUC;
```

4.8.1.2 Aspecto Funcional

El aspecto funcional del sensor y del actuador, carecen de estado, únicamente envían los servicios que se le solicitan al robot o reciben los servicios del robot. Por lo tanto, sus servicios no tienen valuaciones asociadas, tan sólo participan en un proceso funcional modelado a través de un protocolo. Los aspectos funcionales del actuador y el sensor son los siguientes:

```
Functional Aspect FActuator using IMotionJoint

   Services
       begin;

in movejoint(input NewHalfsteps: integer, input Speed: integer);

in stop();

       end;

   Subprocesses
ROBOT = IMotionJoint.stop ? ():0
  +
               ImotionJoint.movejoint ? (Newhalfsteps, Speed):1;

HW = IMotionJoint.stop ! ():0
     +
            ImotionJoint.movejoint ! (Newhalfsteps, Speed):1;

   End_Subprocesses;

   Protocol
FACTUATOR = begin.MOTION;
MOTION= (ROBOT.stop ? ():0
        →
         HW.stop ! ():0.MOTION)
  +
               (ROBOT.movejoint ?(Newhalfsteps, Speed):1
               →
   HW.movejoint !(Newhalfsteps, Speed):1.MOTION)

               +
        end;

End_Functional Aspect FActuator;
```

```
Functional Aspect FSensor using IRead

   Services
      begin;

      out moveok(output Success: [0,1]);

      end;

   Subprocesses
OK = Iread.moveok ! (Success);
HWOK = Iread.moveok ? (Success);

   End_Subprocesses;


   Protocol
FSENSOR = begin.ANSWER;
ANSWER = (HWOK.moveok ? ()
          →
          OK.moveok ! (Success).ANSWER)
          +
          end;

End_Functional Aspect FSensor;
```

Al igual que en el caso del aspecto de coordinación, los aspectos funcionales del sensor y el actuador del SUC de la muñeca varían por el número de parámetros de los servicios y el número de atributos, sin embargo la funcionalidad especificada es la misma que la de los presentados para el resto de los SUCs.

4.8.1.3 Aspecto Distribución

El aspecto distribución de las articulaciones del robot va a especificar su ubicación en una máquina. Siendo esta máquina la misma en cada una de ellas, ya que las partes no son independientes, físicamente son parte del robot y no se pueden separar. Por lo tanto, el aspecto de distribución del robot y sus partes va a ser estático y no va a proveer servicios de movilidad. Sin embargo si que deberá tener los servicios *begin* y *end* para establecer el comienzo y el fin de la ejecución del aspecto. En el caso del aspecto de distribución, debido a que su atributo requerido es *location*, el servicio *begin* tiene la siguiente sintaxis:

```
begin(input InitialLocation: loc);
```

Finalmente, el aspecto de distribución que se obtiene para los conectores de los SUCs es el siguiente:

```
Distribution Aspect DRobotLocation

   Attributes
     Constant
location: loc,
          NOT NULL;
```

```
   Services
      begin(input InitialLocation: loc);
          Valuations
[begin (InitialLocation)]
   location := InitialLocation;

      end;

   Protocols
DROBOTLOCATION = begin(InitialLocation).FINAL;
      FINAL= end;

End_Distribution Aspect DRobotLocation;
```

4.8.1.4 Aspecto de Seguridad

El aspecto seguridad de las articulaciones del robot va a especificar los rangos máximos y mínimos de movimiento de cada una de ellas y deberá de controlar que sus movimientos sean seguros. Un movimiento es seguro siempre que este dentro de los rangos mínimos y máximos establecidos para dicha articulación. Los rangos máximos y mínimos son dos atributos del aspecto que se les dará el valor predeterminado por el robot en cada una de las instancias que representan sus articulaciones. La comprobación de que el movimiento es seguro se hace mediante un servicio check que comprueba que los grados de desplazamiento (*Degrees*) están dentro de los límites (*Secure := true*).

El aspecto de seguridad deberá tener los servicios *begin* y *end* para establecer el comienzo y el fin de la ejecución del aspecto. En este caso, debido a que tiene como atributos requeridos *minimum* y *maximum*, el servicio *begin* tiene la siguiente sintaxis:

```
begin( input InitialMinimum: integer, input InitialMaximum: integer);
```

Finalmente, el aspecto de seguridad que se obtiene para los conectores de los SUCs es el siguiente:

```
Safety Aspect SMotion

   Attributes
      Constant
   minimum: integer,
            NOT NULL;
   maximum: integer,
            NOT NULL;

   Services
  begin( input InitialMinimum: integer, input InitialMaximum:
integer);
          Valuations
[begin (InitialMinimum, InitialMaximum)]
   minimum := InitialMinimum,
       maximum := InitialMaximum;


in check(input Degrees: integer, output Secure: boolean);
```

```
    Valuations
{(Degrees >= minimum) and (Degrees <= maximum)}
  [check(Degrees, Secure)]
      Secure := true};

      {(Degrees < minimum) or (Degrees > maximum)}
  [check(Degrees, Secure)]
       Secure := false;

   Protocol
SMOTION = begin.CHECKING;
      CHECKING= check (Degrees, Secure) + end;

End_Safety Aspect SMotion;
```

Una característica a resaltar del aspecto de seguridad es que a pesar de tener un servicio, no usa ninguna interfaz, ya que dicho servicio es interno y no público. Por lo tanto, al ser utilizado internamente, y no para la comunicación con otros elementos arquitectónicos, este aspecto tampoco tiene sección subprocesses.

4.8.2 Sistema SUC y sus componentes

Tras haber definido los aspectos necesarios para especificar un SUC, a continuación se van a definir el actuador, el sensor, el conector y el SUC. Una vez definido el tipo SUC, éste será instanciado para crear cada uno de los SUCs de las articulaciones que se han diseñado previamente en el modelo arquitectónico (configuración). A continuación, se presentan los tipos *Actuator*, *Sensor* y *CnctSUC*.

El componente *Actuator* especifica el conjunto de puertos necesarios para comunicarse con el conector y el puerto RS232. Para definir un puerto se ha de detallar tanto el nombre, el tipo, que se corresponde con una de las interfaces declaradas, y el subproceso que especifica el comportamiento del conjunto de servicios que forman la interfaz que tipa al puerto. Este subproceso se encuentra definido en el aspecto funcional que importa el componente *Actuator* (*FActuator*). El subproceso se declara mediante la notación punto con el nombre del aspecto y el nombre del subproceso (*FActuator.ROBOT*). Tal y como se ha explicado anteriormente. El tipo Actuador importa un aspecto de distribución con el objetivo de indicar su ubicación dentro del sistema (*DLocation*). Finalmente, el *Actuator* no tiene aspecto de seguridad ya que se va a tratar desde el conector.

Finalmente, en la sección de *Initialize* se especifica como la ejecución del servicio *begin* de los distintos aspectos que lo forman es desencadenada por el servicio *new* que crea la instancia del elemento arquitectónico. Así mismo, en la sección *Destruction* la ejecución del servicio *end* de los distintos aspectos que lo forman es desencadenada por el servicio *destroy* que destruye la instancia del elemento arquitectónico que importa el aspecto al que pertenece el servicio *end*.

```
Component Actuator
```

```
   Ports
      ControlBaseAct: IMotionJoint,
           Subprocess FActuator.ROBOT;
      HWRS232: IMotionJoint,
Subprocess FActuator.ROBOT;

   End_Ports;

   Functional Aspect Import FActuator;
   Distribution Aspect Import DLocation;

   Initialize
     new (Location: loc)
   {
   FActuator.begin();
   DLocation.begin(Location: loc);
   }
   End_Initialize;
   Destruction
     destroy ()
   {
   FActuator.end();
   DLocation. end();
   }
   End_Destruction;

End_Component Actuator;
```

La componente Sensor, al igual que el *Actuator*, contiene en su especificación un conjunto de puertos que se definen con el nombre, el tipo, que se corresponde con una de las interfaces declaradas (*IRead*), y un subproceso asociado (*FSensor.OK*). Tal y como se ha explicado anteriormente, el componente importa un aspecto de distribución (*DLocation*) y no tiene aspecto de seguridad.

```
Component Sensor
   Ports
      ControlBaseSen: IRead,
           Subprocess FSensor.OK;
      HWRS232: IRead,
           Subprocess FSensor.OK;

   End_Ports;

   Functional Aspect Import FSensor;
   Distribution Aspect DLocation;
   Initialize
     new (Location: loc)
   {
   FSensot.begin();
   DLocation.begin(Location: loc);
   }
   End_Initialize;
   Destruction
     destroy ()
   {
   FSensor.end();
```

```
   DLocation. end();
  }
   End_Destruction;

End_Component Sensor;
```

CnctSUC es un conector con tres roles, que son definidos mediante un nombre, un tipo, que se corresponde con una de las interfaces declaradas, y un subproceso asociado que especifica el comportamiento del conjunto de servicios que forman la interfaz que tipa al rol. Este subproceso se encuentra definido en el aspecto coordinación que importa el conector *CnctSUC* (*CProcessSUC*). El subproceso se declara mediante la notación punto indicado el nombre del aspecto y el nombre del subproceso (*CProcessSUC.SUC*).

El conector, al igual que el resto de componentes va a estar formado por un aspecto de distribución que indica su ubicación dentro del sistema (*DLocation*). Finalmente, el CnctSUC se caracteriza por contener un aspecto de seguridad (*SMotion*) que va a controlar que los movimientos de la articulación sean seguros para el robot y para el entorno. El hecho de que se introduzca este aspecto hace que se deban sincronizar mediante dos *weavings* el aspecto de coordinación y el de seguridad. Ambos *weavings* se realizan mediante el operador *afterif*, el cual, además de establecer una sincronización temporal entre los servicios que participan en él, introduce una semántica de obligación, ya que sólo se ejecutará el segundo evento si y sólo si se satisface la condición asociada al operador. Por ejemplo, dado el siguiente *weaving* que aparece en el aspecto de coordinación:

```
   Coordination Aspect Import CProcessSUC;
     Weavings
         Safety Aspect
            CProcessSUC.movejoint(NewHalfsteps, Speed) afterIf (Secure
= true)
            SMotion.check(FTransHalfstepsToAngle(NewHalfsteps),
Secure);
```

La semántica asociada a esta especificación es la siguiente:

<<El servicio *movejoint* del aspecto de coordinación *CProcessSUC* se ejecutará después del servicio check del aspecto de seguridad *SMotion*, si y sólo si el parámetro *Secure* del servicio check devuelve *true*. >>

```
Connector CnctSUC

   Roles
      SUC: ISUC,
           Subprocess CProcessSUC.SUC;
      ControlActuator: IMotionJoint,
           Subprocess CProcessSUC.ACT;
      ControlSensor:IRead,
```

```
            Subprocess CProcessSUC.SEN;
   End_Roles;

   Coordination Aspect Import CProcessSUC;
   Distribution Aspect Import DLocation;
   Safety Aspect Import Smotion;

  Weavings

       CProcessSUC.movejoint(NewHalfsteps, Speed)
       afterIf (Secure = true)
       SMotion.Check(FTransHalfstepsToAngle(NewHalfsteps), Secure);

       CProcessSUC.cinematicsmovejoint( NewAngle, Speed);
       afterIf (Secure = true) SMotion.Check(NewAngle, Segure);

   End_Weavings;
   Initialize
      new (HalfSteps: integer, Location: loc,
Minimum:integer,Maximum:integer)
      {
         CProcessSUC.begin(HalfSteps: integer);
         DLocation.begin(Location: loc);
         Smotion.begin(Minimum:integer, Maximum:integer);
      }
   End_Initialize;
   Destruction
      destroy ()
      {
         CProcessSUC.end();
         DLocation.end();
         Smotion.end();
      }
   End_Destruction;


End_Connector CnctSUC;
```

Finalmente, se ha de definir el sistema SUC que integre los elementos arquitectónicos anteriores. El SUC tiene un puerto para comunicarse con el resto de elementos. Dicho puerto se define mediante un nombre, un tipo, que se corresponde con una de las interfaces declaradas (ISUC). Sin embargo, este puerto no tiene un subproceso asociado, ya que únicamente se comporta como un repetidor del comportamiento del rol al que está conectado mediante un *binding*. Dentro del sistema es necesario, declarar los elementos arquitectónicos que lo forman y los *attachaments* y *bindings* que los unen.

```
System SUC

   Ports
      SUC: ISUC;
   End_Ports;

   Variables
      VarActuator: Actuator;
      VarSensor: Sensor;
       VarCnctSUC: CnctSUC;
```

```
   End_Variable;


   Attachments
    VarCnctSUC.ControlBaseAct ←→ VarActuator.ControlBaseAct;
    VarCnctSUC.ControlBaseSen ←→ VarSensor.ControlBaseSen;
   End_Attachments;


   Bindings
    VarCnctSUC.SUC ←→ SUC.SUC;
   End_Bindings;

   Initialize
      new ()
      {
        VarActuator = new Actuator(Location: loc);
        VarSensor = new Sensor(Location: loc);
        VarConnector = new CnctSUC(HalfSteps: integer,
               Location: loc,
                                Minimum: integer,
                                Maximum: integer);
      }
   End_Initialize;
   Destruction
      destroy ()
      {
         VarActuator.destroy();
         VarSensor.destroy();
         VarConnector.destroy();
      }
   End_Destruction;

End_System SUC;
```

A partir de este sistema SUC se han de crear las instancias a nivel de configuración de las articulaciones base, *shoulder* y *elbow*. Cuando instanciamos un sistema, se ha de instanciar él y los elementos arquitectónicos que incluye. Esto es en primer lugar nos definimos una instancia para el sistema SUC y a partir de esta instancia indicamos la instanciación de sus partes. Por la tanto el mecanismo de instanciación es en cascada. La instanciación del sistema se hace en el nivel superior de granularidad superior, en este caso el MUC. Pero en este apartado vamos a abordar la instanciación de las partes que componen el sistema. La creación de instancias se hará mediante el servicio *New*, al cual se le pasan como parámetros el conjunto de atributos requeridos para crear la instancia. Los atributos requeridos son aquellos que tienen la propiedad de *NOT NULL* en su definición dentro de cada uno de los aspectos del tipo. La sintaxis del servicio es la siguiente:

```
 Instance_Name = new
PRISMA_Achitectural_Type(ListOfRequiredAttributes: List);
```

La lista de atributos requeridos varía según el tipo al que pertenece la instancia, ya que cada tipo importa un conjunto de aspectos diferente y cada

aspecto tiene una cantidad y un tipo de atributos requeridos distintos. Por ejemplo, para crear las instancias del actuador y el sensor es necesario dar un valor inicial al atributo requerido *location* de su aspecto de distribución y el conector que los une necesita dar valor a este mismo atributo además de los atributos *angle* y *halfsteps* del aspecto funcional y de *minimum* y *maximum* del aspecto de seguridad.

Actuador y Sensor:

```
BaseActuator = new Actuator(Location: loc);
BaseSensor = new Sensor(Location: loc);
```

Conector:

```
BaseConnector = new CnctSUC(HalfSteps: integer,
                                    Location: loc,
                                       Minimum: integer,
                                       Maximum: integer);
```

4.9 Especificación de MUCs

Los aspectos que son necesarios para especificar un MUC son: los aspectos de coordinación y distribución. A diferencia de los elementos arquitectónicos presentados hasta el momento, el brazo robot (MUC) no va a tener aspecto seguridad, ya que todas las comprobaciones de los movimientos globales se realizan en el robot para tener en cuenta la herramienta.

4.9.1 Aspectos del MUC

4.9.1.1 Aspecto de Coordinación

El MUC es el encargado de coordinar las articulaciones del robot que lo forman (SUCs) cuando se solicitan movimientos conjuntos del robot por cinemática inversa. El MUC esta formado por todas las articulaciones del robot excepto la herramienta, tal y como se ha presentado en el apartado 3 (ver Figura 28). Además de coordinar los movimientos conjuntos, el MUC se encarga de redireccionar las peticiones de movimiento individualizadas a cada una de las articulaciones que lo forman.

Los movimientos conjuntos del robot a un punto específico se realizan indicando el ángulo que se ha de desplazar cada una de las articulaciones que conforman el robot. También es posible realizar un movimiento simultáneo de todas las articulaciones por *halfsteps*, indicando los *halfsteps* que se ha de mover cada una de ellas. Finalmente, es posible solicitarle a un MUC el movimiento por *halfsteps* de una sola de las articulaciones que lo forman (movimiento individualizado).

El MUC es el agrupador de la parte hardware estática del robot, ya que a diferencia de la herramienta, las piezas del MUC no se cambian en tiempo de ejecución. El MUC no se corresponde con una pieza hardware del robot y es por este motivo que no tiene una posición asociada. Su posición es la de cada una de las articulaciones que lo forman y por ello, el MUC no tiene atributos propios.

El aspecto de coordinación del conector del MUC es el que se muestra a continuación:

```
Coordination Aspect CProcessMUC using IArmMotion, IMotionSUC,
IMotionSUCWrist

  Services
      begin;
      in moveArmSteps(input BaseHalfSteps:integer,
                  input ShoulderHalfSteps:integer,
                  input ElbowHalfSteps: integer, input
RollHalfSteps:integer,
                  input PitchHalfSteps: integer, input Speed:
integer);

      in moveArmCinematics(input BaseAngle: integer,
                        input ShoulderAngle:integer,
                        input ElbowAngle: integer, input RollAngle:
integer,
                        input PitchAngle: integer, input Speed:
integer);

      in moveSteps(input Joint: String, input NewHalfSteps: integer,
                  inputSpeed: integer);

       in/out wristmovejoint(input NewleftHalfSteps: integer,
                     input NewrightHalfSteps: integer, input Speed:
integer);


       out movejoint(input NewHalfSteps: integer, input Speed:
integer);
       out cinematicsmovejoint( input NewAngle: integer,
                                input Speed: integer);

       out  wristcinematicsmovejoint(input Rolldegrees: integer,
                                     input Pitchdegrees: integer,
                                     input Speed: integer);


      in/out moveok(output success: [0,1]);


      end;

   Subprocesses
      MUC=  (IArmMotion.moveArmSteps ? (BaseHalfSteps,
ShoulderHalfSteps,
                                       ElbowHalfSteps, RollHalfSteps,
                                       PitchHalfSteps, Speed)
             +
             IArmMotion.moveArmCinematics ? (BaseAngle, ShoulderAngle,
```

```
                                          ElbowAngle, RollAngle,
PitchAngle,
                                          Speed)
      +
      IArmMotion.moveSteps ? (Joint, NewHalfSteps, Speed)

      +
IArmMotion.wristmovejoint ?(NewleftHalfSteps,
                            NewrightHalfSteps,Speed)
)
      →
             IArmMotion.moveok ! (Success);


      BASE= (IMotionSUC.movejoint ! (NewHalfSteps, Speed)
      +
             IMotionSUC.cinematicsmovejoint ! (NewAngle, Speed)
             )

      →
             IMotionSUC.moveok ? (Success);

      SHOULDER= (IMotionSUC.movejoint ! (NewHalfSteps, Speed)
          +
                 IMotionSUC.cinematicsmovejoint! (NewAngle, Speed)
                 )

           →
                 IMotionSUC.moveok ? (Success);
      ELBOW= (IMotionSUC.movejoint ! (NewHalfSteps, Speed)
       |
              IMotionSUC.cinematicsmovejoint ! (NewAngle, Speed)
              )

       →
              IMotionSUC.moveok ? (Success);
      WRIST= (IMotionSUCWrist. Wristmovejoint ! (NewleftHalfSteps,
                                                 NewrightHalfSteps,
Speed)
             +
 IMotionSUCWrist.wristcinematicsmovejoint ! (Rolldegrees,
                                             Pitchdegrees,
                                Speed)
              )

      →
             IMotionSUCWrist.moveok ? (Success);
   End_Subprocesses;


   Protocol
      CPROCESSMUC = begin.MOTION;
      MOTION= (
              (MUC.moveArmSteps ? (BaseHalfSteps, ShoulderHalfSteps,
                                   ElbowHalfSteps, LeftHalfSteps,
                                   RightHalfSteps, Speed)
```

```
                →
 (BASE.movejoint ! (BaseHalfSteps, Speed)
 ||
  SHOULDER.movejoint ! (ShoulderHalfSteps,Speed)
 ||
  ELBOW.movejoint ! (ElbowHalfSteps, Speed)
 ||
  WRIST.wristmovejoint ! (LeftHalfSteps, RightHalSteps,Speed)
)
)
       +
       (MUC.moveArmCinematics ? (BaseAngle, ShoulderAngle, ElbowAngle,
                                      RollAngle, PitchAngle, Speed)
       →
      (BASE.cinematicsmovejoint ! (BaseAngle, Speed)
       ||
      SHOULDER.cinematicsmovejoint ! (ShoulderAngle, Speed)
      ||
      ELBOW.cinematicsmovejoint ! (ElbowAngle, Speed)
      ||
            WRIST.wristcinematicsmovejoint ! (RollAngle, PitchAngle,
Speed)
             )
             )
            ).TEAMANSWER
             +
             (
             (MUC.moveSteps?(Joint, NewHalfSteps, Speed)
              →
 (|Joint = “Base”| BASE.movejoint ! (NewHalfSteps, Speed)
  +
  |Joint = “Shoulder”| SHOULDER.movejoint ! (NewHalfSteps, Speed)
  +
  [Joint = “Elbow”] ELBOW.movejoint !(NewHalfSteps,Speed)
 )
 )
             +
       (MUC.wristmovejoint ? (NewleftHalfSteps, NewrightHalfSteps,
                              Speed)
        →
  WRIST.wristmovejoint ! (NewleftHalfSteps, NewrightHalfSteps,
                          Speed)
  )
              )
             ).JOINTANSWER

              +
  end;
JOINTANSWER= ((BASE.moveok ? (Success)
              +
              SHOULDER.moveok ? (Success)
              +
              ELBOW.moveok ? (Success)
              +
              WRIST.moveok?(Success)
              )
              →
              MUC.moveok ! (Success)
              ).MOTION
TEAMANSWER= ((BASE.moveok ? (Success)
             Λ
```

```
            SHOULDER.moveok ? (Success)
            ∧
            ELBOW.moveok ? (Success)
            ∧
            WRIST.moveok ? (Success)
            )
             →
            MUC.moveok ! (Success)
            ).MOTION

End_Coordination Aspect CProcessMUC;
```

4.9.1.2 Aspecto de Distribución

El aspecto distribución del brazo robot va a especificar su ubicación en una máquina, siendo esta máquina la misma de cada una de sus articulaciones, ya que físicamente éstas son las que forman el brazo robot. Por lo tanto, al igual que las articulaciones, el aspecto de distribución del brazo robot va a ser estático y no va a proveer servicios de movilidad.

4.9.2 Sistema MUC y sus componentes

4.9.2.1 Conector del MUC

Tras haber definido los aspectos necesarios para especificar el conector del MUC, a continuación se va a definir éste. Dicho conector será un tipo que se utilizará para definir el tipo MUC. Una vez definido el tipo MUC, éste será instanciado para crear el MUC que forma el brazo robot del modelo arquitectónico (*configuración*). A continuación, se presenta una instancia en lugar del tipo, con el objetivo de que se comprenda mejor la especificación.

El conector del tipo MUC tiene cinco roles para comunicarse, tanto con cada una de las articulaciones que lo forman, como con el nivel de granularidad superior. El nombre que se les ha dado a cada uno de los roles en la especificación, es el mismo que aparece en la Figura 28. Además del nombre, se declara el tipo, que se corresponde con una de las interfaces declaradas, y se le asocia un subproceso que especifica el comportamiento del conjunto de servicios que forman la interfaz que tipa al rol. Este subproceso se encuentra definido en el aspecto de coordinación que importa el conector. Tal y como se ha explicado anteriormente, el tipo conector importa un aspecto de distribución con el objetivo de indicar su ubicación dentro del sistema (*DLocation)*. Finalmente, el conector no tiene aspecto de seguridad ya que se va a tratar en el nivel superior (RUC).

```
Connector RobotArmCnct

   Roles
      MUCCnct: IArmMotion,
           Subprocess CProcessMUC.MUC;
      BaseCnct: IMotionSUC,
           Subprocess CProcessMUC.BASE;
      ShoulderCnct:IMotionSUC,
           Subprocess CProcessMUC.SHOULDER;
```

```
      ElbowCnct:IMotionSUC,
           Subprocess CProcessMUC.ELBOW;
      WristCnct:IMotionSUCWrist,
           Subprocess CProcessMUC.WRIST;
   End_Roles;

   Coordination Aspect Import CProcessMUC;
   Distribution Aspect Import DLocation;



   Initialize
      new (Location: loc)
      {
         CProcessMUC.begin();
         DLocation.begin(Location: loc);
      }
   End_Initialize;
   Destruction
      destroy ()
      {
         CProcessMUC.end();
         DLocation. end();
      }
   End_Destruction;

End_Connector_Instance RobotArmCnct;
```

4.9.2.2 Sistema del MUC

Finalmente, para acabar de especificar el MUC, se ha de definir el sistema que integre conector definido en el apartado anterior y el conjunto de sistemas SUCs que lo forman. El MUC tiene cinco puertos tal y como se muestra en la Figura 28. Además del nombre, se declara el tipo del puerto, que se corresponde con una de las interfaces declaradas. Sin embargo, estos puertos no tienen un subproceso asociado, ya que únicamente se comportan como repetidores del comportamiento del rol o puerto al que están conectados mediante un *binding* (ver Figura 28). Dentro del sistema es necesario, declarar los *attachaments* y *bindings* que unen a los componentes que encapsula.

```
System MUCRobotArm

   Ports
      MUCArmCNCT: IArmMotion;
      MUCArmBase: IEmergency;
      MUCArmShoulder: IEmergency;
      MUCArmElbow: IEmergency;
      MUCArmWrist: IEmergency;
   End_Ports;
   Variables
       VarBase= SUC;
       VarShoulder= SUC;
       VarElbow = SUC;
       VarWrist = SUCWrist;
       VarArmCnct= RobotArmCnct;
   End_Variables;
   Attachments
    VArArmCnct.BaseCNCT ←→ VarBase.Base;
    VarArmCnct.ShoulderCNCT ←→ VarShoulder.Shoulder;
```

```
    VarArmCnct.ElbowCNCT ←→ VarElbow.Elbow;
    VarArmCnct.WristCNCT ←→ VarWrist.Wrist;
   End_Attachments;
   Bindings
      MUCRobotArm.MUCArmCnct ←→ VarArmCnct.MUCCNCT;
      MUCRobotArm.MUCArmElbow ←→ VarElbow.Elbow;
      MUCRobotArm.MUCArmBase ←→ VarBase.Base;
      MUCRobotArm.MUCArmShoulder ←→ VarShoulder.Shoulder;
      MUCRobotArm.MUCArmWrist ←→ VarWrist.Wrist;
   End_Bindings;
   Initialize
      new (){
          VarBase = new SUC();
          VarShoulder = new SUC();
          VarElbow = new SUC();
          VarWrist = new SUCWrist();
          VarArmCnct = new robotArmCnct(Location: loc);
         }
   End_Initialize;

   Destruction
      destroy ()
      {
         VarBase.destroy();
         VarShoulder.destroy();
         VarElbow.destroy();
         VarWrist.destroy();
         VarArmCnct.destroy();

      }
   End_Destruction;


End_System_Instance MUCRobotArm;

MUC= new MUCRobotArm(){
          BaseSUC = new SUC(){
         BaseActuator= new Actuator(localhost);
         BaseSensor= new Sensor(localhost);
         BaseConnector =
new CnctSUC(0,0,localhost,90,-90);
      };

   ShoulderSUC = new SUC(){
         ShoulderActuator= new Actuator(localhost);
         ShoulderSensor= new Sensor(localhost);
         ShoulderConnector =
                  new CnctSUC(0,0,localhost,144,-35);
      };

   ElbowSUC = new SUC(){
         ElbowActuator= new Actuator(localhost);
         ElbowSensor= new Sensor(localhost);
         ElbowConnector =
new CnctSUC(0,0,localhost,0,-149);
      };

   Wrist = new WristSUC(){
   WristAct= new WristActuator(localhost);
         WristSensor= new Sensor(localhost);
         WristCnct = new WristCnctSUC(0, 0, 0, 0,
```

```
                        localhost, 90, -90, 270, -270);
    };
     ARMCNCT = new RobotArmCnct(localhost);
    };
```

4.10 Especificación de RUCs

En este apartado se van a especificar los aspectos y los elementos arquitectónicos necesarios para definir el RUC, es decir, el robot *TeachMover* completo

4.10.1 Aspectos del RUC

Los aspectos necesarios para definir un RUC son los aspectos de coordinación, distribución y seguridad del conector que coordina los movimientos del brazo robot y la herramienta, en este caso la pinza. Los aspectos del conector se presentan a continuación.

4.10.1.1 Aspecto de Coordinación

El RUC es el encargado de coordinar la herramienta y el brazo robot cuando se solicitan movimientos conjuntos del robot por cinemática inversa. El RUC está formado por el MUC, que representa el brazo robot y por el SUC de la herramienta (ver Figura 30). Además de coordinar los movimientos conjuntos, el RUC se encarga de redireccionar las peticiones de movimiento individualizadas a cada una de las articulaciones que lo forman.

Los movimientos conjuntos del robot a un punto específico se realizan indicando el punto (x,y,z) que se desea alcanzar con la herramienta del robot. También es posible realizar un movimiento simultáneo de todas las articulaciones por *halfsteps*, indicando los *halfsteps* que se ha de mover cada una. Finalmente, es posible solicitarle a un RUC el movimiento por *halfsteps* de una sola de las articulaciones que lo forman.

El robot es el que establece el eje de coordenadas de referencia (x0, y0, z0), necesario para calcular que los movimientos que se le solicitan son correctos y son posibles. Dentro del aspecto de coordinación del conector del RUC, serán necesarios tres atributos para poder guardar el eje de coordenadas de referencia. Dicho eje es constante, no se puede modificar y su valor es (5,0,0). Su especificación se muestra a continuación:

```
Attributes
    Constant
       x0: integer (5);
       y0: integer(0);
```

```
          z0: integer(0);
```

Los movimientos conjuntos por pasos y por cinemática inversa se van a realizar mediante dos servicios de movimiento diferentes *stepsmoverobot* y *cinematicsmoverobot*. Mientras que los movimientos individualizados de cada una de las articulaciones del robot, se van a realizar mediante el servicio *moveSteps*. Al servicio *stepsmoverobot* se le han de pasar como parámetros los *steps* que se han de desplazar cada una de las articulaciones y la velocidad con la que han de realizar el movimiento. Sin embargo, el servicio *cinematicsmoverobot* tiene como argumentos el punto (x, y, z) al que se desea mover el robot, los ángulos de rotación específicos para los movimientos *pitch* y *roll* (p,r) de la muñeca, las pulgadas de apertura de la pinza (i) y la velocidad con la que se ha de mover el robot. Finalmente, el servicio *moveSteps*, especificará la articulación que desea mover, el número de *halfsteps* y la velocidad con la que desea realizar el movimiento. Se ha de tener en cuenta que para el caso de la herramienta se utiliza el servicio *movewristjoint* debido a que necesita el número de pasos *left* y *right* y la sintaxis del servicio es distinta a la del resto de articulaciones.

Cuando el conector de un RUC invoca unos de estos servicios de movimiento, no implica que el robot realmente se mueva, a pesar de que el movimiento solicitado sea correcto. El conector solamente tendrá la certeza de que el robot se ha movido cuando se le notifique que el movimiento se ha realizado satisfactoriamente. En ese momento, el robot se lo notificará al operario. La notificación se realiza mediante un servicio que devuelve 0 o 1, dependiendo de si el movimiento se ha realizado o no, respectivamente.

Como resultado final, el aspecto de coordinación que se obtiene para el conector del RUC es el siguiente:

```
Coordination Aspect CProcessRUC using IRobotMotion, IArmMotion,
ISUCTool

   Attributes
     Constant
      x0: integer (5);
      y0: integer(0);
      z0: integer(0);

  Services
      begin;

      in stepMoveRobot( input BaseHalfSteps: integer,
                        input ShoulderHalfSteps: integer,
                        input ElbowHalfSteps: integer,
                        input LeftHalfSteps: integer,
                  input RightHalfSteps: integer, input Speed:
integer);

      in cinematicsMoveRobot( input x: integer, input y: integer,
                              input z: integer, input p: integer,
                              input r: integer, input i: integer,
                              input Speed: integer);
```

```
      in/out moveSteps(input Joint: String, input NewHalfSteps:
integer,
                       input Speed: integer);

      in/out wristMoveJoint(input NewleftHalfSteps: integer,
                            input NewrightHalfSteps: integer,
                            input Speed: integer);

      in/out close();

      in/out moveOk(output Success: [0,1]);

      out moveArmSteps(input BaseHalfSteps: integer,
                       input ShoulderHalfSteps: integer,
                       input ElbowHalfSteps: integer,
                       input LeftHalfSteps: integer,
                       input RightHalfSteps: integer, input Speed:
integer);

      out moveArmCinematics(input BaseAngle: integer,
                            input ShoulderAngle: integer,
                            input ElbowAngle: integer,
                            input RollAngle: integer,
                            input PitchAngle: integer,
                            input Speed: integer);

      out moveInches( input NewInches: integer, input Speed: integer);

      out moveJoint(input NewHalfsteps: integer, input Speed:
integer);


   Subprocesses

      RUC= (IRobotMotion.stepMoveRobot ? (BaseHalfSteps,
ShoulderHalfSteps,
                  ElbowHalfSteps, LeftHalfSteps, RightHalfSteps,
                        HandHalfSteps, Speed)
           +
                  IRobotMotion.cinematicsMoveRobot ? (x, y, z, p, r,
i,Speed)
           +
             IRobotMotion.moveSteps ? (Joint, NewHalfSteps, Speed)
           +
             IRobotMotion.wristMoveJoint ? (NewleftHalfSteps,
                                            NewrightHalfSteps, Speed)
           +
             IRobotMotion.close ? ()
           )
          →
            IRobotMotion.moveOk ! (Success);


      MUC= (IArmMotion.moveSteps ! (Joint, NewHalfSteps, Speed)
           +
            IArmMotion.wristMoveJoint ! (NewleftHalfSteps,
NewrightHalfSteps,
                                         Speed)
           +
            IArmMotion.moveArmSteps !(BaseHalfSteps,
ShoulderHalfSteps,
                                      ElbowHalfSteps, LeftHalfSteps,
```

```
                                        RightHalfSteps, Speed)
          +
           IArmMotion.moveArmCinematics !(BaseAngle, ShoulderAngle,
                                           ElbowAngle,
RollAngle,PitchAngle,
                                           Speed)
           )
           →
           IArmMotion.moveok ? (Success);


      TOOL= (IToolSUC.moveJoint ! (NewHalfsteps, Speed)
       +
      IToolSUC.close ! ()
      )
      →
      IToolSUC.moveOk ? (Success);
   End_Subprocesses;

   Protocol
      CPROCESSRUC = begin.MOTION;
MOTION = (
        (RUC.stepMoveRobot ? (BaseHalfSteps, ShoulderHalfSteps,
                 ElbowHalfSteps, LeftHalfSteps, RightHalfSteps,
                 HandHalfSteps, Speed)
        →
              (MUC.moveArmSteps!(BaseHalfSteps, ShoulderHalfSteps,
                           ElbowHalfSteps, LeftHalfSteps,
RightHalfSteps,
                           Speed)
               ||
               TOOL.moveJoint ! (HandHalfsteps, Speed)
              )

       +

              (RUC.cinematicsMoveRobot ? (x, y, z, p, r, i, Speed)
             →
  (MUC.moveArmCinematics ! (FtranXYZToBaseAngle(x,y,z),
                   FtranXYZToShoulderAngle(x,y,z),
                   FtranXYZToElbowAngle(x,y,z),
                   r, p, Speed)
  ||
              TOOL.moveInches !(i, Speed)
              )
              )
             ).TEAMANSWER

       +

              ((|Joint = “Tool"| RUC.moveSteps ? (Joint,
NewHalfSteps,
                                                  Speed)
             →
               TOOL.moveJoint ! (NewHalfsteps, Speed))

  +

       (|Joint <> “Tool"| RUC.moveSteps ? ( Joint, NewHalfSteps,
                                      Speed)
             →
```

```
                MUC.moveSteps ! ( Joint, NewHalfSteps, Speed)
               )

              +

              (RUC.wristMoveJoint ? (NewleftHalfSteps,
NewrightHalfSteps,
                                     Speed)
              →
  MUC.wristMoveJoint ! (NewleftHalfSteps, NewrightHalfSteps,
                                    Speed)
               )
              +
  RUC.close ? () → TOOL.close ! ()

              ).JOINTANSWER
              +
              end;

    JOINTANSWER= ((MUC.moveOk ? (Success)
                   +
                   TOOL.moveOk ? (Success)
                  )
                  →
                  RUC.moveOk(Success)
                  ).MOTION
    TEAMANSWER= ((TOOL.moveOk ? (Success)
                  ∧
                  MUC.moveOk ? (Success)
                  )
                 →
                 RUC.moveOk ! (Success)
                  ).MOTION

End_Coordination Aspect CProcessRUC;
```

4.10.1.2 Aspecto de Distribución

El aspecto distribución del robot *TeachMover* va a especificar su ubicación en una máquina, siendo esta máquina la misma del brazo robot y la herramienta, ya que físicamente éstos son los que forman el robot. La ubicación del robot no va cambiar en tiempo de ejecución, porque para ello hay que mover el robot desconectándolo del puerto serie y conectándolo al puerto serie de otro ordenador. Si esto se realizara en tiempo de ejecución daría errores. Por este motivo, para el cambio de ubicación del robot, se para la aplicación y se vuelve a lanzar a ejecución el sistema robótico con su nueva ubicación. Por lo tanto, el aspecto de distribución del robot va a ser estático y no va a proveer servicios de movilidad.

4.10.1.3 Aspecto de Seguridad

El aspecto seguridad del robot va a especificar los rangos máximos y mínimos de movimientos conjuntos respecto al eje de coordenadas de referencia y va a controlar que los movimientos conjuntos sean seguros. Un movimiento es seguro siempre que esté dentro de los rangos mínimos y máximos de las restricciones que afectan al movimiento. Las restricciones que se han de comprobar ante un movimiento completo del robot son las siguientes:

<< La herramienta estará demasiado cerca del cuerpo del robot y existirá un riesgo de choque entre ambos, si la distancia del eje de coordenadas de referencia hasta el punto (x,y,z) es menor de 4 y z es menor de 15.>>

Sea RR la distancia del eje de coordenadas de referencia al punto (x,y,z) y pz la coordenada (z). El movimiento del robot será correcto si se cumple la siguiente restricción:

RR >= 4 or pz >= 15

<< La distancia máxima que puede alcanzar el robot es de 17,8. Por lo tanto la distancia del eje de coordenadas al punto (x,y,z) no puede ser mayor de 17.8>>

Sea RR la distancia del eje de coordenadas de referencia al punto (x,y,z). La restricción es la siguiente:

RR < 17.8

<< La herramienta estará muy cerca de la base teniendo un riesgo de choque si x es menor de 2.25, z menor de 1.25, la distancia del centro de coordenadas de referencia a la muñeca es menor de 3.5 y los grados del ángulo *pitch* de la muñeca es menor de -90º>>

Sea R0 la distancia del eje de coordenadas de referencia a la posición de la muñeca. La restricción es la siguiente:

X >= 2,25 or z >= 1.25 or R0 >= 3.5 or p >= -90º

La comprobación de que el movimiento es seguro se hace mediante una transacción *SAFEROBOT* que contiene la ejecución de tres servicios de seguridad que comprueban cada una de estas restricciones. Dicha transacción comprueba que el moviendo es seguro (*Secure := true*).

El aspecto de seguridad deberá tener los servicios *begin* y *end* para establecer el comienzo y el fin de la ejecución del aspecto. En este caso, el servicio *begin* proporciona el valor de los rangos máximos y mínimos de movimiento del robot. Es por este motivo, que la sintaxis del servicio es la que se presenta a continuación:

```
      begin(Initminrrhand, Initminzhand, Initminrr, Initminx,
Initminzwrist,
             Initminr0, Initminp: integer);
```

Finalmente, el aspecto de seguridad que se obtiene para los conectores de los SUCs es el siguiente:

```
Safety Aspect SRobotSafety

   Attributes
```

```
      Variable
         rr: integer;
         rO: integer;
         minrr: integer,
                  NOT NULL;
         minr0: integer,
                  NOT NULL;
         minrrhand: integer,
                  NOT NULL;
         minzhand: integer,
                  NOT NULL;
         minzwrist: integer,
                  NOT NULL;
         minx: integer,
                  NOT NULL;
         minp: integer,
                  NOT NULL;


  Services
      begin(Initminrrhand: integer, Initminzhand: integer, Initminrr:
integer,
             Initminx: integer, Initminzwrist: integer, Initminr0:
integer,
             Initminp: integer);
          Valuations
      [begin (Initminrrhand, Initminzhand, Initminrr, Initminx,
              Initminzwrist, Initminr0, Initminp)]
            minrr := Initminrr;
      minr0:= Initminr0;
      minrrhand:= Initminrrhand;
      minzhand:= Initminzhand;
            minzwrist:= Initminzwrist;
            minx:=Initminx;
            minp:= Initminp;



      in checkHand (input NewX: integer, input NewY: integer,
                    input NewZ: integer; output Secure: boolean);
          Valuations
      [checkHand(NewX, NewY, NewZ, Secure)]
             rr := FHandDistance(NewX,NewY,NewZ,rr),

            {( rr >= minrrhand) or (Newz >= minzhand)}
      [checkHand(NewX, NewY, NewZ, Secure)]
           {Secure := true},

            {(rr < minrrhand) and (Newz < minzhand)}
      [checkHand(NewX, NewY, NewZ, Secure)]
            {Secure := false};


       in checkWrist(input NewX: integer, input NewY: integer,
                     input NewZ: integer, input NewP: integer,
                     input NewR: integer; output Secure: boolean);
          Valuations
      [checkWrist(NewX,NewY,NewZ,NewP, NewR, Secure)]
       r0 := FWristDistance(NewX,NewY,NewZ,NewP, NewR, r0),

      {(NewX >= minx) or (NewZ >= minzwrist) or (R0 >= minr0) or
```

```
       (NewP >= minp)}
      [checkWrist(NewX,NewY,NewZ,NewP, NewR, Secure)]
            {Secure := true},

             { (NewX < minx) and (NewZ < minzwrist) and (R0 < minr0)
and
               (NewP < minp)}
      [checkWrist(NewX,NewY,NewZ,NewP, NewR, Secure)]
             {Secure := false};



      in checkdistance(input NewX: integer, input NewY: integer,
                       input NewZ: integer, output Secure: boolean);
          Valuations
      { RR <= minrr}
      [checkDistance(NewX, NewY, NewZ, Secure)]
            {Secure := true},

             {RR > minrr}
      [checkDistance(NewX, NewY, NewZ, Secure)]
                 {Secure := false};

      end;

  operations
       transaction SAFEROBOT(input NewX: integer, input NewY: integer,
                             input NewZ: integer, NewP, NewR: integer,
                             output Secure: boolean):
            saferobot = checkHand(NewX, NewY, NewZ,
Secure).SAFEDISTANCE;
            SAFEDISTANCE = checkDistance(NewX, NewY, NewZ,
                                         Secure).SAFEWRISTDISTANCE;
            SAFEWRIST = checkWrist(NewX,NewY,NewZ,NewP, NewR,
                                   Secure).SAFEROBOT;
  Protocol
      ROBOTSAFETY = begin.CHECKING;
      CHECKING= SAFEROBOT(NewX, NewY, NewZ, NewP, NewR, Secure) + end;
End_Safety Aspect SRobotSafety;
```

4.10.2 Sistema RUC y sus componentes

4.10.2.1 Conector del RUC

Tras haber definido los aspectos necesarios para especificar el conector del RUC, a continuación se va a definir éste. Dicho conector será un tipo que se utilizará para definir el tipo RUC. Una vez definido el tipo RUC, éste será instanciado para crear el RUC que forma el robot *TeachMover* del modelo arquitectónico (configuración).

El conector *RobotCnct* tiene tres roles para comunicarse, tanto con la herramienta y el brazo robot, como con el nivel de granularidad superior (ver Figura 30). Además del nombre, se declara el tipo, que se corresponde con una de las interfaces declaradas, y se le asocia un subproceso que especifica el comportamiento del conjunto de servicios que forman la interfaz que tipa al rol.

Este subproceso se encuentra definido en el aspecto de coordinación que importa el conector. Tal y como se ha explicado anteriormente, la instancia importa un aspecto de distribución con el objetivo de indicar su ubicación dentro del sistema (*DLocation*). Finalmente, el *RobotCnct* se caracteriza por contener un aspecto de seguridad (*SRobotSafety*) que va a controlar que los movimientos conjuntos del robot sean seguros para el robot y para el entorno. El hecho de que se introduzca este aspecto hace que se deban sincronizar mediante un *weaving* el aspecto de coordinación y el de seguridad. Este *weaving* se realiza mediante el operador *afterif*, el cual, además de establecer una sincronización temporal entre los servicios que participan en él, introduce una semántica de obligación, ya que se sólo se ejecutará el segundo evento si y sólo si se satisface la condición asociada al operador.

```
Connector RobotCNCT

   Roles
      RUCCnct: IRobotMOtion,
           Subprocess CProcessRUC.RUC;
      Tool: ISUCTool,
           Subprocess CProcessRUC.TOOL;
      MUC:IArmMotion,
           Subprocess CProcessRUC.MUC;
   End_Roles;

   Coordination Aspect Import CProcessRUC;
   Distribution Aspect Import DLocation;
   Safety Aspect Import SRobotSafety;

  Weavings
     Coordination Aspect
       CProcessRUC.cinematicsmoverobot(x, y, z, p, r, i, Speed)
       afterIf (Secure = true)
        SRobotSafety.SAFEROBOT(NewX, NewY, NewZ, NewP, NewR, Secure);
   Initialize
      new (Location: loc, Initminrrhand: integer,
Initminzhand:integer,
                  Initminrr: integer, Initminx: integer,
                  Initminzwrist: integer, Initminr0: integer,
                  Initminp: integer)
            {
               CProcessRUC.begin();
               DLocation.begin(Location: loc);
         SRobotSafety.begin(Initminrrhand:integer,
                           Initminzhand: integer,
                                  Initminrr: integer, Initminx:
integer,
                                  Initminzwrist: integer, Initminr0:
integer,
                                  Initminp: integer);
      }
   End_Initialize;
   Destruction
      destroy ()
      {
         CProcessRUC.end();
         DLocation. end();
         SRobotSafety.end();
      }
```

```
   End_Destruction;

End_Connector_Instance RobotTeachMoverCNCT;
```

4.10.2.2 Sistema RUC

Finalmente, para acabar de especificar el RUC, se ha de definir el sistema que integre el conector definido en el apartado anterior, el sistema MUC y el sistema SUC de la herramienta que lo forman. El sistema Robot tiene seis puertos tal y como se muestra en la Figura 30, el nombre que se le da en la especificación es el mismo que aparece en esta figura. Además del nombre, se declara el tipo del puerto, que se corresponde con una de las interfaces declaradas. Sin embargo, estos puertos no tienen un subproceso asociado, ya que únicamente se comportan como repetidores del comportamiento del rol o puerto al que están conectados mediante un *binding*. Dentro del sistema es necesario, declarar las instancias que lo forman y los *attachaments* y *bindings* que los unen.

```
System  Robot

   Ports
      Robot: IRobotMotion;
      Base: IEmergency;
      Shoulder: IEmergency;
      Elbow: IEmergency;
      Wrist: IEmergency;
      Tool: IEmergency;
   End_Ports;

   Variables
       VarTool= ToolSUC;
 VarRobotCnct = RobotCnct;
       VarMUC= MUCRobotArm;
   End_Variables;

   Attachments
    VarRobotCnct.MUCArmCNCT ←→ VarMUC.ArmRobot;
    VarRobotCnct.ToolCNCT ←→ VarTool.Tool;
   End_Attachments;

   Bindings
      Robot.Robot ←→ VarRobotCnct.Robot;
      Robot.Base ←→ VarMUC.MUCArmBase;
      Robot.Shoulder ←→ VarMUC. MUCArmShoulder;
      Robot.Elbow ←→ VarMUC. MUCArmElbow;
      Robot.Wrist ←→ VarMUC. MUCArmWrist;
      Robot.Tool ←→ VarTool.Tool;
   End_Bindings;
   Initialize
      new (){
              VarTool = new ToolSUC(Location:loc);
        VarRobotCnct = new RobotCnct((Location: loc,
                                      Initminrrhand: integer,
                                      Initminzhand:integer,
                                             Initminrr: integer,
                                             Initminx: integer,
                                             Initminzwrist: integer,
                                             Initminr0: integer,
```

```
                                                Initminp: integer);
              VarMUC = new MUCRobotArm();
          }
   End_Initialize;

   Destruction
      destroy ()
      {
         VarTool.destroy();
         VarRobotCnct.destroy();
         VarMUC.destroy();
      }
   End_Destruction;

End_System RobotRUC;



RobotTeachMover = new RUC{
TOOL = new ToolSUC(){
   ToolAct= new WristActuator(localhost);
   ToolSensor= new Sensor(localhost);
   ToolCnct = new WristCnctSUC(0,localhost,0,3);
};
MUC= new MUCRobotArm(){
          BaseSUC = new SUC(){
         BaseActuator= new Actuator(localhost);
         BaseSensor= new Sensor(localhost);
         BaseConnector =
new CnctSUC(0,0,localhost,90,-90);
      };

   ShoulderSUC = new SUC(){
         ShoulderActuator= new Actuator(localhost);
         ShoulderSensor= new Sensor(localhost);
         ShoulderConnector =
                  new CnctSUC(0,0,localhost,144,-35);
      };

   ElbowSUC = new SUC(){
         ElbowActuator= new Actuator(localhost);
         ElbowSensor= new Sensor(localhost);
         ElbowConnector =
new CnctSUC(0,0,localhost,0,-149);
      };

   Wrist = new WristSUC(){
   WristAct= new WristActuator(localhost);
         WristSensor= new Sensor(localhost);
         WristCnct = new WristCnctSUC(0, 0, 0, 0,
                           localhost, 90, -90, 270, -270);
      };
          ARMCNCT = new RobotArmCnct(localhost);
         };
TeachMoverCnct = new RobotCnct(localhost,4,15,17.8,2.25,1.25,3.5,-
90º);
}
```

Capítulo 5

IMPLEMENTACIÓN PRISMA

CONTENIDOS

IMPLEMENTACIÓN PRISMA

En este capítulo se describe el proceso de traducción del un modelo arquitectónico PRISMA al lenguaje de programación *Microsoft C#.* A lo largo del capítulo se detallan todas las decisiones de implementación tomadas para que el proceso de transformación, de una especificación PRISMA al lenguaje imperativo, sea un proceso sencillo, eficiente y que permita la trasformación de cualquier especificación de forma genérica. Los mecanismos de traducción se han definido mediante patrones siguiendo la propuesta de la MDA (*Model Driven Architecture*) [Mellor04].

La propuesta *Model Driven Architecture* (MDA) del *Object Management Group* (OMG) es un estándar de desarrollo de aplicaciones basada en modelos. La MDA promueve el uso de modelos para desarrollar *software* a un alto nivel de abstracción para intentar resolver los problemas actuales de integración del ciclo de vida del *software*, que cada vez es más complejo. Con este fin, el estándar propone el desarrollo de sistemas de información delimitando claramente la especificación de la funcionalidad del sistema, de su implementación y puesta en marcha sobre una plataforma tecnológica específica. En consecuencia, el desarrollo de aplicaciones basadas en MDA permite desarrollar aplicaciones procedentes del mismo modelo sobre distintas plataformas tecnológicas. Para lograr esto, el desarrollo de *software* comienza por la construcción de un Modelo Independiente de Plataforma (PIM). Este modelo representa la funcionalidad y el comportamiento del sistema de forma independiente de la tecnología empleando el Lenguaje Unificado de Modelado (UML). El objetivo de MDA es conseguir transformar los PIM, de más alto nivel de abstracción, en uno o varios Modelos Específicos de Plataforma (PSM), a partir de los cuales sea posible generar de forma automática el código fuente del sistema (ver Figura 36).

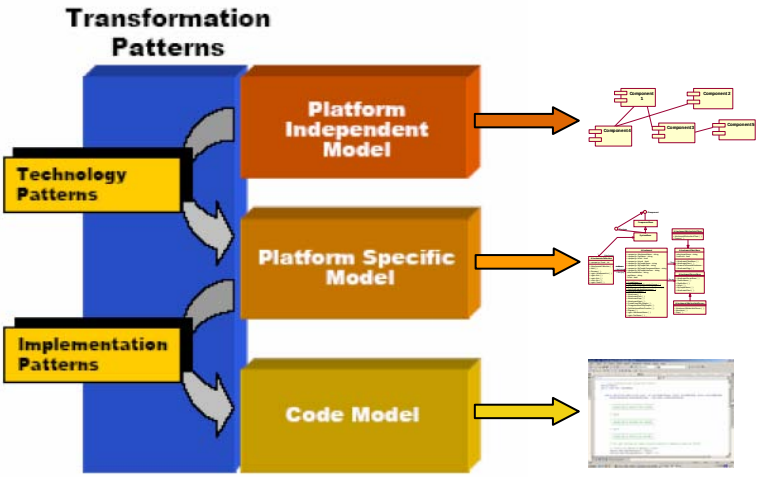


Figura 36 - MDA: Transformación de Modelos

Para realizar las transformaciones entre los modelos, se emplean patrones de transformación. Los patrones de transformación en MDA se clasifican en dos categorías: los Patrones Tecnológicos y los Patrones de

Implementación. Por una parte, los patrones tecnológicos definen las correspondencias entre el PIM y el PSM para una tecnología concreta como *Java* o *.net*. Por otra parte, los patrones de implementación dirigen la traducción de los PSM al código fuente final.

PRISMA es un modelo para definir arquitecturas *software* orientadas a aspectos y basadas en componentes. Por lo que siguiendo la tendencia de la MDA, el modelo PRISMA es un PIM. Asimismo, el *middleware* para la ejecución de modelos arquitectónicos PRISMA sobre la plataforma .NET constituye el modelo PSM del modelo PIM PRISMA. La transformación entre el modelo PRISMA y la plataforma *.net* se define mediante patrones tecnológicos; mientras que la transformación de un modelo arquitectónico determinado PRISMA a código *C#* se realiza mediante los patrones de implementación.

5.1 Patrones

Los patrones se basan en la búsqueda de problemas comunes y recurrentes para extraer la esencia del problema y darle una solución adecuada. Los patrones son soluciones de sentido común que establecen un marco de referencia. Estas soluciones tienen como característica principal que son efectivas, ya que se emplearon y se probaron en ocasiones anteriores, y reusables, ya que se pueden aplicar a diferentes problemas en distintas circunstancias o contextos. Una vez encontradas estas abstracciones se construye un catálogo de patrones, que permita reutilizar el patrón siempre que sea posible.

5.1.1 Definición

El concepto de patrón no queda limitado al área de la ingeniería del *software*, de hecho los primeros patrones fueron diseñados para la construcción. Existen muchas definiciones de patrón y cada una de ellas proporciona unos aspectos o da unas perspectivas diferentes. Entre ellas podemos encontrar las siguientes:

- [Rie96]:

"Un patrón es la abstracción de una forma concreta que se mantiene en contextos específicos y no arbitrarios"

- [App97]:

"Un patrón es una semilla nominada de información instructiva que captura la estructura esencial y la perspicacia de una familia de soluciones probadas a un problema recurrente que surge dentro de un cierto contexto entre intereses o fuerzas contrapuestas"

- [Ale77]:

"Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, y describe la esencia de la solución a ese problema, de tal modo que pueda utilizarse esta solución un millón de veces más, sin siquiera hacerlo de la misma manera dos veces"

- [Ale79]:

"Cada patrón es un regla compuesta por tres partes, que expresan una relación entre un cierto contexto, un problema y una solución"

- [Cop99]:

"Podemos explicar como hacer un vestido especificando la ruta de las tijeras sobre la tela en términos de ángulos y longitudes de corte. O bien, podemos dar un patrón. Leyendo la especificación de corte, nadie tendría idea acerca de que se está construyendo, al menos hasta que esté construido. Sin embargo, el patrón presagia el resultado: es la regla para crear el vestido, pero también, en gran medida, es el vestido en sí mismo"

Finalmente, en el área de informática se pueden destacar varias citas que definen los patrones *software*, como las de [Gam94] que se muestran a continuación:

"La solución no describe un diseño o implementación particular (...) el patrón provee una descripción abstracta del diseño del problema y como los elementos lo solucionan"

"Un patrón de diseño nombra, abstrae e identifica los aspectos clave de una estructura común de diseño que la hace útil para crear un diseño OO reusable"

5.1.2 Patrones *Software*

Los patrones *software* se construyen de forma general en base a dos aspectos importantes, sus cualidades y la clasificación.

5.1.2.1 Cualidades

Cuando se habla de cualidad se hace referencia a una propiedad que un patrón ha de optimizar, ya que lo dota de mejor calidad. Las cualidades de los patrones *software* son:

- **Encapsulación**. Todo patrón ha de estar encapsulado, ya que es una entidad abstracta, y por consiguiente, debe ser capaz de separarse del contexto externo.

- **Granuralidad**. La granularidad del patrón no ha de ser ni tan grande como que uno solo lo solucione todo, ni tan pequeña que pierdan el sentido de entidad independiente.

- **Descomposición** (fragmentación). Todo patrón debe poder ser componente de otros para formar un lenguaje de patrones que solucionen otros más complejos.

Por ello, la solución de un gran problema se debe descomponer en pequeñas abstracciones o patrones que solucionen subproblemas.

- **Dependencias** (interrelaciones). Los patrones dependen unos de otros y se relacionan para solucionar problemas de mayor dimensión.

- **Flexibilidad**. Los patrones deben ser flexibles para aumentar su reutilización y productividad.

- **Escalabilidad y evolución**. Los patrones deben ser capaces de evolucionar para resolver otros problemas.

- **Funcionamiento correcto**. Un patrón debe proporcionar una solución con un funcionamiento correcto.

- **Reutilización**. El patrón es una abstracción que soluciona problemas que suceden frecuentemente. Por lo tanto, debe ser capaz de solucionar el problema todas las veces que suceda en sus diferentes versiones.

5.1.2.2 Clasificación

Existen diferentes ámbitos dentro de la ingeniería del *software* donde se pueden aplicar los patrones:

1. **Patrones de arquitectura:** expresa una organización o esquema estructural fundamental para sistemas *software*. Proporciona un conjunto de subsistemas predefinidos, especifica sus responsabilidades, e incluye una guía para organizar las relaciones entre ellos.

2. **Patrones de diseño:** proporciona un esquema para refinar los subsistemas o componentes de un sistema *software*, o las relaciones entre ellos. Describe estructuras repetitivas para comunicar componentes que resuelven un problema de diseño en un contexto particular.

3. **Patrones de programación (Idioms patterns):** un idioma es un patrón de bajo nivel de un lenguaje de programación específico. Describe como implementar características de componentes o de las relaciones entre ellos utilizando las facilidades del lenguaje de programación.

4. **Patrones de análisis:** describen un conjunto de prácticas que aseguran la obtención de un buen modelo de un problema y su solución.

5. **Patrones organizacionales:** describen la estructura y prácticas de las organizaciones humanas, especialmente en las que producen, utilizan o administran *software*.

Para cada ámbito, los patrones *software* se pueden clasificar según el propósito o según el ámbito de aplicación.

- **Propósito**

- Creación. Tratan procesos de creación de objetos.

- Estructural. Definen estructuras de la composición de las clases u objetos.

- Comportamiento. Definen el comportamiento de los objetos, y describen como los objetos interactúan y se distribuyen las responsabilidades.

- Ámbito de aplicación

- Clase. Solucionan problemas que son fijos en tiempo de diseño.

- Objeto. Solucionan problemas que varían en tiempo de ejecución.

5.2 Patrones Tecnológicos

La implementación de una arquitectura PRISMA determinada se realiza mediante la extensión por herencia de las clases que componen el *middleware* PRISMA para *.net* (PRISMANET) [Cos05]. PRISMANET implementa el metamodelo PRISMA mediante un conjunto de clases que ofrecen la funcionalidad básica de cada uno de los conceptos del modelo. Además, PRISMANET proporciona el soporte a la movilidad y soporta la distribución y evolución de todos los elementos arquitectónicos. Debido a la naturaleza distribuida del modelo, la implementación de las comunicaciones entre los elementos del modelo se ha realizado con la tecnología *.net Remoting* (ver Figura 37). Este *middleware* es el resultado de aplicar los patrones tecnológicos sobre el Modelo Independiente de Plataforma PRISMA, es decir, PRISMANET es el modelo PSM de PRISMA para .NET.

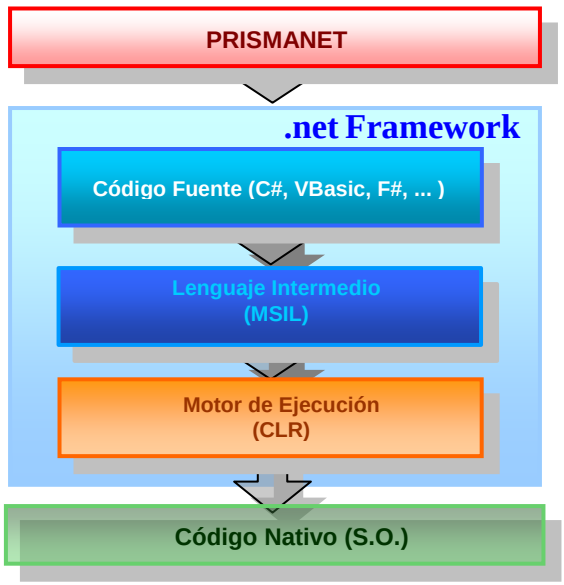


Figura 37 - *Middleware* PRISMA sobre *.net Framework*

El *middleware* PRISMA es una capa adicional de *software* implementada en *CSharp* que se ejecuta sobre el *framework* de la plataforma *.net*. Los diferentes tipos PRISMA se han implementado en una serie de clases, agrupadas según su funcionalidad en diferentes espacios de nombres (ver

Figura 38), y que proporcionan distintos servicios de gestión interna de todos los conceptos del modelo y del modelo de ejecución.

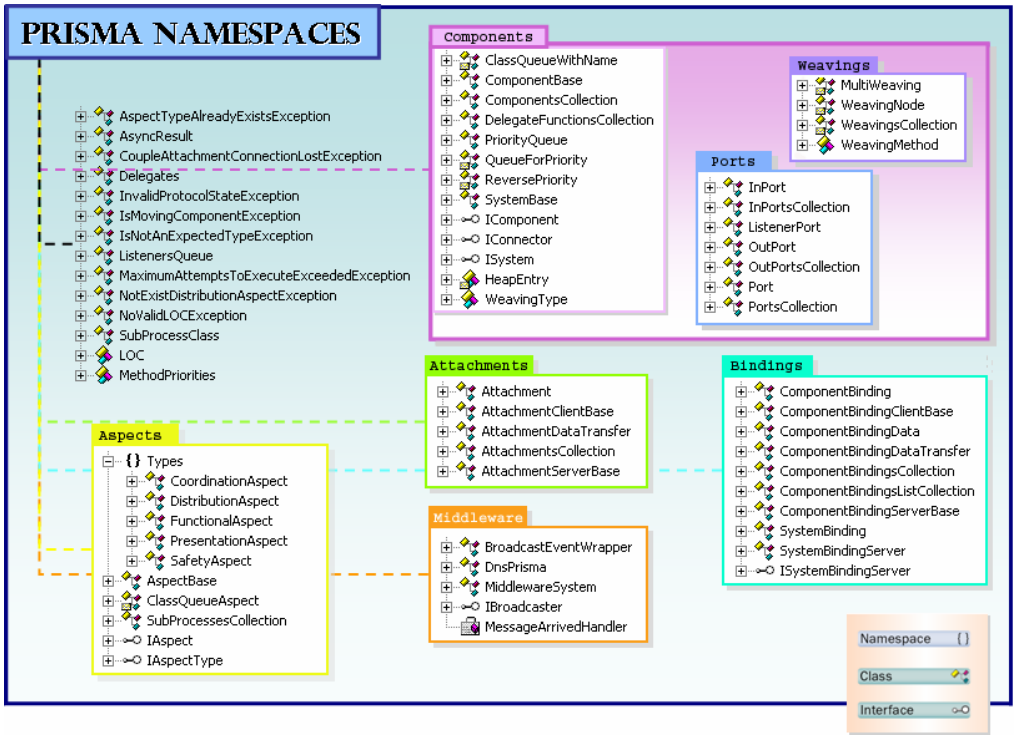


Figura 38 - Clases de PRISMANET

La implementación de un nuevo modelo arquitectónico, correspondiente a un caso de estudio particular, requiere la creación de una nueva "solución en blanco" de Visual Studio .net. La nueva solución incluirá un conjunto de proyectos de bibliotecas de clases *(dll's)* a medida que se vayan implementando. Cada uno de estos proyectos contendrá una clase *CSharp* que corresponderá a la implementación de cada uno de los conceptos del modelo PRISMA especificado en el modelo arquitectónico. Por ejemplo, un proyecto puede contener una clase con la implementación de un aspecto, de un componente, de un conector, de un sistema... Dependiendo del concepto implementado, la clase empleará una o varias clases de PRISMANET para extenderlas según este especificado en el modelo arquitectónico. Así, todos los proyectos de la nueva solución harán uso de las clases definidas en el *middleware* PRISMA mediante una referencia al proyecto donde esta definido PRISMANET (ver Figura 39). Además, la implementación de cada uno de los conceptos en proyectos de bibliotecas de clases independientes permite que estos puedan ser reutilizados en la misma solución o en otras soluciones más adelante preservando la reutilización de los elementos arquitectónicos tal y como define el modelo PRISMA.

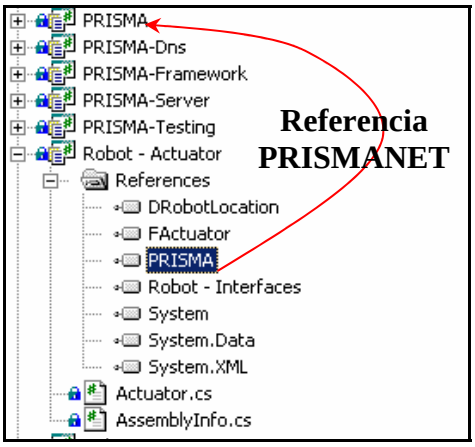


Figura 39 - Proyecto del componente *Actuator*

5.3 Patrones de Generación Automática de Código

Los patrones de generación automática de código se encargan de guiar la traducción de los modelos arquitectónicos PRISMA a código fuente final definiendo los patrones tecnológicos y de implementación de la propuesta MDA. En el caso del modelo especifico de la plataforma *.net* para PRISMA como modelo independiente de plataforma, los patrones de generación de código permiten definir las descripciones de los conceptos del modelo para traducirlas a código fuente *CSharp*.

Las principales fuerzas que se han perseguido durante la elaboración de estos patrones son:

- **Automatización y facilidad de uso**: Se pretende crear reglas genéricas de traducción de modelos arquitectónicos para aplicar de forma automática o semi-automática.

- **Reusabilidad**: La solución propuesta debería ser utilizada para cualquier modelo PRISMA y se pretende especificar una expresión que describa las correspondencias entre los conceptos del modelo y los elementos de implementación del *.net Framework*.

- **Flexibilidad:** La solución es fácilmente adaptable a la traducción de cualquier modelo arquitectónico.

- **Simplicidad:** La traducción del lenguaje de arquitecturas tiene una correspondencia directa con el código fuente emitido.

En el marco de traducción de modelos PRISMA, los patrones de generación de código son patrones que se requieren para transformar un modelo arquitectónico PRISMA a una aplicación ejecutable sobre PRISMANET.

Para ello se han definido un conjunto de patrones para cada uno de los conceptos y elementos que componen el modelo.

5.3.1 Interfaces

Una interfaz es la definición de un conjunto de métodos para los que no se da implementación, al igual que en el modelo PRISMA, una interfaz publica un conjunto de métodos que son implementados por aquellos aspectos que la utilizan.

La implementación de todas la interfaces propias de un modelo arquitectónico se realiza en un mismo proyecto de biblioteca de clases que se denominará Interfaces. Este proyecto contiene un fichero *Interfaces.cs* en el cual se definen las interfaces del modelo PRISMA como interfaces de *.net* mediante el uso de la palabra reservada "`interface`". Como todos los ficheros *CSharp*, este fichero empieza importando los espacios de nombres del *framework* .net que requiera, generalmente *System*. Además, en la definición de las interfaces se emplean elementos definidos en PRISMANET, por lo que también se importa el espacio de nombres *PRISMA* del *middleware.*

```
using System;
using PRISMA;
```

Todos los tipos PRISMA (aspectos, interfaces, etc.) implementados sobre .NET deben ser miembros de un espacio de nombres (*namespace)* que los englobe. Este *namespace* es único para toda la implementación de los conceptos del modelo y se llama de la misma forma que el nombre del modelo arquitéctónico, PRISMA. Debido a esto, la implementación de las interfaces también está incluida dentro de este espacio de nombres.

```
namespace <modelo_arquitectonico>
```

La implementación de las interfaces del modelo PRISMA se realiza mediante las interfaces del lenguaje *CSharp*. Las interfaces conservan el nombre especificado en el modelo arquitectónico.

```
public interface <interfaz>
{
   <miembros>
}
```

Un ejemplo es la interfaz *ISUC* del caso de estudio del robot 4U4:

```
public interface ISUC
{
   <miembros>
}
```

Los servicios PRISMA definidos como miembros de una interfaz se traducen en implementación a la definición de funciones que devuelven como tipo `AsyncResult`. Este tipo de datos está definido en el espacio de nombres PRISMA de PRISMANET y permite incorporar la ejecución asíncrona de los servicios ofrecidos por los aspectos PRISMA. De esta manera, se admite el procesamiento concurrente de múltiple métodos. La asincronía en la ejecución

de un servicio es proporcionada a través de varios mecanismos de PRISMANET.La clase `AsyncResult` es la estructura que permite posteriormente comprobar si la petición ha sido procesada para poder obtener los resultados (ver Figura 40).

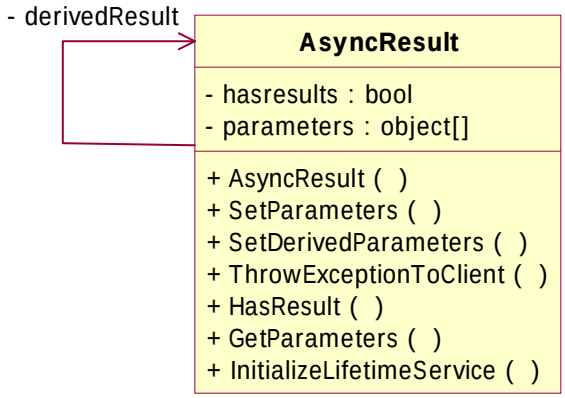


Figura 40 - Clase Asyncresult en PRISMANET

```
AsyncResult <nombreServicio>([<def_parametros>]);
```

Por ejemplo, el servicio `MoveJoint` se define de la siguiente manera:

```
AsyncResult MoveJoint(int newHalfSteps, int speed);
```

La lista de parámetros de los métodos corresponde con los parámetros de los servicios definidos en la especificación. Es posible que un servicio no tenga parámetros por esa razón los `<def_parametros>` son opcionales (representando la opcionalidad entre corchetes "[ ]"), como ilustra la definición del servicio `Stop`

```
AsyncResult Stop();
```

A diferencia del modelo PRISMA, en el que los parámetros de los servicios deben llevar asociado un modificador para indicar si el parámetro es de entrada (input), de salida (output) o de entrada/salida (input / output). En implementación los parámetros sin modificador son considerados como parámetros de entrada y sólo en el caso de que sean de salida o de entrada/salida son dotados del modificador `ref`.

El servicio MoveOK devuelve el resultado mediante el parámetro `success`:

```
AsyncResult MoveOK(ref int success);
```

Por otro lado, por cada servicio que se defina en una interfaz, es necesario definir un tipo que haga referencia al método para soportar el modelo de ejecución de los conceptos PRISMA como veremos más adelante. Una referencia a un método en *.net* se realiza mediante delegados. Un delegado es un tipo especial de clase cuyos objetos pueden almacenar una referencia a un método de tal manera que a través del objeto es posible solicitar su ejecución. En la definición de los delegados se debe tener en cuenta que sólo debe aparecer una vez, es decir si existe un mismo servicio declarado en varias interfaces el tipo delegado para ese servicio se define una únicamente una vez.

El nombre del tipo de cada delegado corresponde con el nombre del método y a continuación `Delegate`.

```
public delegate AsyncResult <nombreServicio>Delegate([<parametros>]);
```

Por ejemplo, la definición de la referencia al metodo MoveJoint

```
public delegate AsyncResult MovejointDelegate(int newHalfSteps, int
speed);
```

Patrón 1 : Interfaces

Contexto

Traducción de interfaces PRISMA al lenguaje de implementación *CSharp*.

Modelo PRISMA

```
Interface name
   service1;
   ... ...
   servicen;
End_Interface name;
```

Patrón tecnológico

```
using System;
using PRISMA;
namespace <nombreModelo>
{
   public interface <nombreInterfaz>
   {
      AsyncResult <nombreServicio>([ [ref] id_par : Tipo ]);
      ... ...
   }
   public delegate AsyncResult
                        <nombreServicio>Delegate([<parametros>]);
   ... ...
}
```

Caso de Estudio

Descripción

La interfaz ISUC del caso de estudio del robot 4U4 define los servicios de movimiento y de confirmación de una articulación del brazo de robot, además del servicio de parada de emergencia.

Modelo Arquitectónico PRISMA del Caso de Estudio

```
Interface ISUC
   moveJoint(input NewHalfSteps: integer, input Speed: integer);
   cinematicsMoveJoint(input NewAngle: integer, input Speed: integer);
   stop();
```

```
  moveOk(output success);
End_Interface ISUC;
```

Patrón Implementación

```
using System;
using PRISMA;
namespace Robot
{
   public interface ISUC
   {
      AsyncResult MoveJoint(int newHalfSteps, int speed);
      AsyncResult CinematicsMoveJoint(double newAngle, int speed);
      AsyncResult Stop();
      AsyncResult MoveOK(ref int success);
   }
   public delegate AsyncResult MovejointDelegate(int newHalfSteps, int
                                                              speed);
   public delegate AsyncResult CinematicsMoveJointDelegate(double
                                                  newAngle, int speed);
   public delegate AsyncResult StopDelegate();
   public delegate AsyncResult MoveOKDelegate(ref int success);
}
```

5.3.2 Aspectos

Un aspecto en PRISMA es la definición completa de la estructura y el comportamiento de un elemento arquitectónico desde el punto de vista de un determinado interés del sistema *software* (c*oncern*).En implementación un aspecto se corresponde con una clase *CSharp* y se implementa de acuerdo con una estructura determinada. Dicha estructura se define mediante una plantilla para proporcionar los conceptos de estado y comportamiento a los objetos de la clase. En tiempo de ejecución, un objeto de la clase aspecto equivale a una instancia de un aspecto que se comporta acorde con la definición de su clase, el modelo de ejecución establecido por el *middleware* PRISMA. La implementación de los aspectos se realiza con el propósito de conservar las propiedades definidas en el modelo PRISMA. Los aspectos tienen un carácter modular y reutilizable, y permiten su ejecución de forma autónoma frente a los demás tipos del modelo arquitectónico. Sin embargo, un aspecto no puede lanzarse a ejecución por si solo, ha de estar importado y referenciado por un componente, tal y como establece el modelo PRISMA.

En PRISMA, una instancia de un aspecto, es un objeto cuya vida está caracterizada por la ocurrencia de acciones, tanto si son solicitadas como si son provistas por ella. Así, una instancia puede actuar como cliente o como servidor según esté solicitando (comportamiento cliente) u ofreciendo servicios

(comportamiento servidor). La estructura de un aspecto está definida en base a dos perspectivas: estática y dinámica. La parte estática se constituye de atributos y son los valores de estos atributos los que caracterizan el estado del aspecto. La parte dinámica está formada por un conjunto de servicios y sus respectivas reglas, expresadas en lógica dinámica [Har84], que determinan la actividad del aspecto. La ocurrencia de dichos servicios genera cambios en los valores de los atributos del aspecto, por tanto produce un cambio del estado del aspecto.

La implementación de cada uno de los aspectos de un modelo arquitectónico se realiza en un nuevo proyecto de biblioteca de clases. Cada proyecto tiene el nombre del aspecto y contiene un fichero de extensión *.cs* en el cual se encuentra la definición de la clase del aspecto. El fichero *CSharp* empieza importando el espacio de nombres del *framework* .net que requiere, es decir *System*. Además, para la definición del tipo del aspecto se emplean los elementos definidos en PRISMANET que dan soporte a los aspectos, por lo que también se importan los espacios de nombres del *middleware* en donde se encuentran definidos estos elementos.

```
using System;

using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
```

La definición de la clase asociada al aspecto se realiza a partir del nombre del tipo del aspecto que está especificado en el modelo. Además, la clase debe derivar de una clase del *middleware* PRISMA que le proporcione la funcionalidad común para todos los aspectos PRISMA. Esta clase, en PRISMANET, está definida dentro del espacio de nombres `PRISMA.Aspects` y se llama `AspectBase`. Esta clase implementa un conjunto de conceptos que forman un aspecto de forma general. Además, en PRISMA existen diferentes tipos de aspecto, definiéndose cada uno de ellos de forma independiente, por lo que en el *middleware* PRISMA existe una colección de clases que representan cada uno de los tipos de aspecto que se pueden definir. Cada una de las clases de la colección se encuentra bajo el espacio de nombres de `PRISMA.Aspects.Types` y heredan de `AspectBase` para extender el comportamiento común de todo aspecto con las características propias de cada tipo. En PRISMANET existen ya una colección de tipos de aspectos predefinidos (aspecto funcional, aspecto de coordinación, aspecto de distribución, aspecto de presentación, aspecto de seguridad, aspecto de replicación…) (ver Figura 41) por lo que la clase del aspecto que se está implementando deriva directamente del tipo de aspecto del que se trate. Si bien es cierto, el número de tipos de aspectos no está limitado y dependiendo del sistema *software* que se implemente serán necesarios unos u otros.

```
using System;

using PRISMA;
using PRISMA.Aspects.Types;

namespace <nombreModelo>
{
   public class <nombreAspecto>: <tipoAspecto>
```

```
    {
      <cuerpoAspecto>
    }
}
```

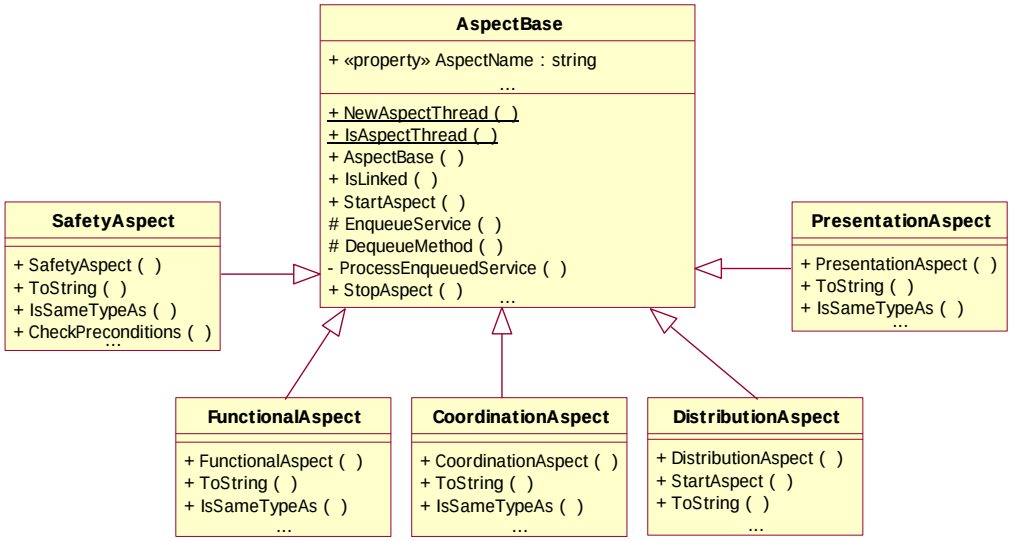


Figura 41 - Superclases para ASPECTOS en PRISMANET

Un aspecto puede verse como la unión de un conjunto de interfaces que publican los servicios del tipo del aspecto. Los servicios de los aspectos están definidos por interfaces para que los aspectos los implementen y así que distintos aspectos puedan implementar los servicios de la misma interfaz. Para implementar los servicios de una interfaz, un aspecto debe derivar de ella, lo que implica que la clase del aspecto debe implementar la interfaz.

```
public class <nombreAspecto>: <tipoAspecto>, <nombreInterfaz>
```

Por ejemplo, en el caso de estudio del brazo de robot, se especifica un aspecto de coordinación (`CProcessSUC`) que implementa la interfaz `ISUC`, `IRead` e `IMotionJoint` (ver Figura 42).

```
public class CProcessSUC: CoordinationAspect,IMotionJoint,IRead,ISUC
```

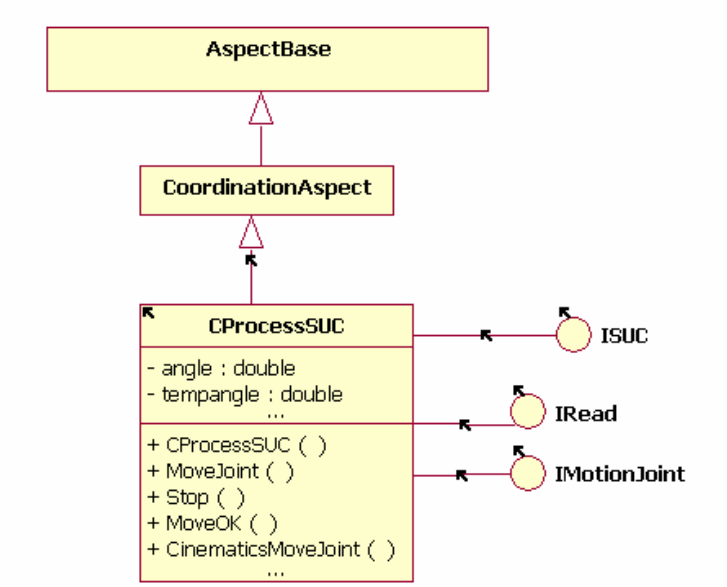


Figura 42 - Implementación CProcessSUC

En la figura se muestra como la clase del aspecto de coordinación `CProcessSUC` hereda de la clase del *middleware* que proporciona la funcionalidad específica para los aspectos de coordinación. Además, la clase `CProcessSUC` implementa todos los servicios publicados por las interfaces `ISUC`, `IRead` e `IMotionJoint`.

El modelo PRISMA permite definir modelos arquitectónicos completamente distribuidos, y además da soporte a la movilidad de todos sus elementos. Esto en implementación implica tener que serializar las instancias de los aspectos para poder enviarlas desde un dominio de aplicación de un nodo, a otro nodo distinto de otro dominio. Por ello se requiere marcar la clase del aspecto con el atributo `Serializable` que proporciona el *.net framework.* Aunque únicamente deberíamos marcar la clase cuando se trate de implementar una arquitectura distribuida y móvil, se va a dotar siempre a la clase del aspecto la propiedad `Serializable` para facilitar su generación automática. Esto es posible ya que no penaliza el código de ninguna manera. Es importante tener en cuenta que el atributo `Serializable` no se puede heredar, de haber sido así hubiera bastado marcar como serializables únicamente a las clases de tipos de aspecto definidas en el *middleware* PRISMA.

```
[Serializable]
public class <nombreAspecto>: <tipoAspecto>, <nombreInterfaz>
```

5.3.2.1 Modelo de ejecución de aspectos

El modelo de ejecución de los distintos aspectos que componen un componente es concurrente. Dicha concurrencia establece que los aspectos se ejecuten de forma concurrente y de forma autónoma entre sí. Los mecanismos que proporcionan este modelo están definidos como parte de la funcionalidad básica que hereda el aspecto de las superclases del *middleware* PRISMA. Para que los aspectos se ejecuten concurrentemente, las superclases dotan al aspecto de un hilo de ejecución propio para ir procesando los servicios que se

le requieran. También, para dar soporte a la concurrencia, cada aspecto dispone de una cola de solicitudes de servicios pendientes en la cual se almacenan las distintas peticiones a medida que son recibidas. El hilo de ejecución se encarga de comprobar si existen solicitudes pendientes en la cola y de ir procesándolas secuencialmente. Gracias a la cola, se garantiza que se ejecuten todos los servicios correctamente y en orden adecuado, ya que se podría dar el caso que mientras el hilo estuviera sirviendo una invocación de un servicio se produjera otra que no podría ser atendida. Además, gracias al asincronismo de los servicios proporcionado por el tipo `AsyncResult`, los clientes dejan sus peticiones en la cola, continúan ejecutándose y posteriormente solicitan los resultados una vez procesado el servicio.(ver Figura 43)

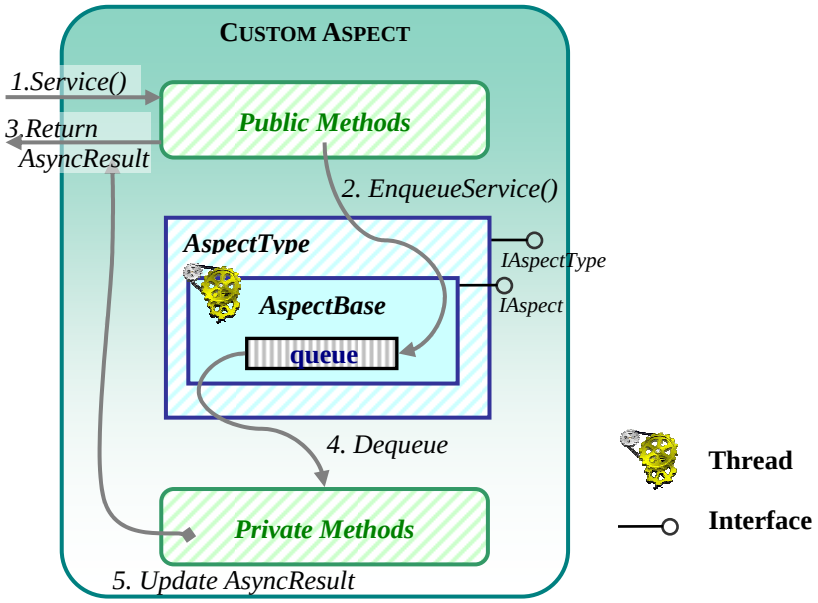


Figura 43 - Esquema de ejecución de un Aspecto

5.3.2.2 Atributos

Los atributos de un aspecto representan el estado de la vida del aspecto de forma análoga que en el modelo orientado a objetos los atributos de una clase. Al igual que en OASIS, en PRISMA se pueden declarar tres tipos distintos de atributos: constantes, variables o derivados.

Los atributos constantes no pueden cambiar su valor. Estos se corresponden en implementación a atributos privados de clase pero precedidos de la palabra reservada `readonly`. De esta forma, las asignaciones a los campos que aparecen en la declaración sólo pueden tener lugar en la propia declaración en tiempo de diseño o en un constructor de la clase para que tome valor al inicio de la vida del aspecto.

```
readonly private <tipoAtributo> <nombreAtributo>
```

Los atributos variables pueden cambiar su valor durante la vida del aspecto, por lo que corresponden con atributos privados normales de clase.

```
private <tipo_atributo> <nombre_atributo>
```

Finalmente, los atributos derivados como su nombre indica, son atributos cuyo valor se deriva a partir los valores de otros atributos. Para dar valor a estos atributos es necesario definir una función de derivación que calcule el valor del atributo a partir del valor de los otros atributos y. Para acceder al valor de los atributos derivados se define una propiedad privada `get` que devuelva el valor de la función de transformación.

```
private <tipoAtributo> <nombreAtributo>

private <tipoAtributo> <nombrePropiedad>
{
    get
    {
         return <invocaciónFunciónTransformación>;
    }
}

private <tipoAtributo> <nombreFunción>
{
      return <formulaDerivación>;
}
```

Además, los atributos constantes y variables se pueden especificar para que sus valores sean requeridos, es decir que no sean nulos. En los atributos constantes basta únicamente con proporcionar al atributo un valor en tiempo de diseño o en su defecto asignándole un valor en el constructor de la clase. Para los atributos variables es necesario asignarles un valor en el constructor de la clase y además, en el caso de que el tipo del atributo no sea un tipo básico (entero, real, booleano…), es necesario declarar una propiedad privada `set` que compruebe que el valor que se le asigna al atributo no es nulo y en el caso de que lo sea, lance una excepción.

```
private <tipoAtributo> <nombrePropiedad>
{
  set
  {
    if (value != null)
      <nombreAtributo> = value;
    else
      throw new Exception("El atributo es requerido")
  }
}
```

En el modelo arquitectónico del robot, el aspecto de coordinación CProcessSUC cuenta con dos atributos variables, uno requerido y otro no (halfSteps, tempHalfSteps) y un atributo derivado (angle). A continuación se muestra la implementación de dichos atributos:

```
int halfsteps, temphalfsteps;
double angle;
private double Angle
{
    get
    {
         return FtransAngle();
    }
}
```

5.3.2.3 Servicios

Los servicios en el modelo PRISMA representan el cambio de estado de los atributos variables de un aspecto y como consecuencia de ello, el cambio de estado de los atributos derivados. Es posible definir tres tipos de servicios dentro un aspecto: comienzo (*begin*), fin (*end*) y el resto de servicios que dan soporte a la funcionalidad del tipo del aspecto. Los servicios ocurren en un instante de tiempo durante la vida del aspecto.

5.3.2.3.1 Servicios de comienzo y fin

En un aspecto, los servicios de comienzo y fin se han implementado como el constructor (*new*) y el destructor (*destroy*) constituyen la factoría del aspecto mediante la cual se inicia y se destruye las instancias del aspecto. Si bien es cierto, dichos servicios únicamente pueden ser invocados desde un componente, ya que en PRISMA no se permite la instanciación de aspectos de forma autónoma sin un componente a la que pertenezcan.

```
public <nombreAspecto>([<parametros>]):base()
{
  <cuerpoConstructor>
}
```

Por ejemplo, el constructor del aspecto CProcessSUC se define:

```
public CProcessSUC(int initialHalfSteps) : base("CProcessSUC")
{
... ...
}
```

En implementación, el servicio de destrucción *destroy* no tiene implementación en *C#*. La destrucción de un aspecto es realizada por el recolector de basura y se inicia automáticamente cuando el recolector detecta que queda poca memoria libre, o cuando se va a finalizar la ejecución de la aplicación y por supuesto no existe ninguna referencia en ejecución al aspecto.

5.3.2.3.2 Servicios de Entrada y Salida

El modelo PRISMA permite definir los servicios de un aspecto como de entrada (in), de salida (out) o de entada/salida (in / out). Cuando un servicio se define de entrada, se está modelando el comportamiento servidor del aspecto, mientras que si un servicio es de salida, se está modelando el aspecto cliente del aspecto.

El carácter cliente del aspecto se puede modelar en los aspectos de dos formas diferentes. En primer lugar y de la misma forma que en OASIS, los disparos (*triggers*) proporcionan al aspecto el comportamiento cliente después de la ejecución de un servicio. Esto obliga que el servicio definido como disparador ocurra necesariamente a continuación de haber servido un servicio. En implementación, la invocación del servicio de salida se realiza en la estrategia de ejecución de los servicios. En segundo lugar, el protocolo permite determinar el orden de ejecución de los servicios, indicando que servicios de entrada desencadenan la invocación de otros servicios de salida. Esto se

traduce en implementación con la invocación de un servicio de salida después de la comprobación de las restricciones de integridad durante la estrategia de ejecución de una petición de servicio.

5.3.2.3.3 Protocolo

Dentro de un aspecto, el concepto de protocolo permite especificar las vidas posibles de las instancias del aspecto. Dentro de un protocolo se definen un conjunto de subprocesos que se corresponde con los estados posibles en la vida de un aspecto y las posibles transiciones válidas entre ellos. Las transiciones entre los subprocesos (estados) vienen determinadas por la ocurrencia de los servicios. Por tanto, el protocolo define la finalización de un proceso y el comienzo de otro (el paso de un estado a otro) a partir de la ejecución de una secuencia determinada de servicios. Los subprocesos que se definen en el protocolo puede ser de tres tipos: creación (contiene el servicio de comienzo de ejecución del aspecto (creación de la instancia) y no existe ningún servicio en el protocolo que transicione hacia él), de destrucción (contiene únicamente el servicio de fin (destrucción de la instancia) y no contiene ningún servicio que transicione hacia otro subproceso) y simples (pueden llegar y salir transiciones). En implementación, el conjunto de subprocesos del protocolo se almacena en un tipo enumeración de *C#* definida en el aspecto y que permite almacenar una lista de constantes. Esta enumeración `protocolStates` define los estados posibles de la ejecución de un aspecto.

```
enum protocolStates
{
   <PRE-CREACION>,
   <ESTADO_1>,
   <ESTADO_2>,
      …
   <ESTADO_N>,
}
```

Además es necesario declarar, como variable privada de la clase del aspecto, un atributo *estado* del tipo de la enumeración. El atributo `estado` almacena el estado del aspecto en función de los servicios que acontezcan.

```
protocolStates estado;
```

La implementación del conjunto de subprocesos que existen dentro del protocolo del aspecto de coordinación de `CProcessSUC` se establece de la siguiente manera:

```
enum protocolStates
     {
        CPROCESSSUC,
        MOTION,
        ANSWER,
        EMERGENCY
     }
```

El inicio de ejecución del protocolo se implementa en el cuerpo del constructor. En primer lugar se inicializa el estado del protocolo, y posteriormente las variables de la clase del aspecto, que se corresponden con atributos constantes, que no se les haya dado valor en tiempo de diseño, y

atributos variables y requeridos, se les dan valor. Por último, se actualiza el estado del protocolo alcanzado.

```
estado = protocolStates.<PRE_CREACION>;
<nombreAtributo_1> = <valor>;
<nombreAtributo_2> = <valor>;
...
estado = protocolStates.<NUEVO_ESTADO>;
```

Por ejemplo, el constructor del aspecto CProcessSUC, inicializa la variable estado al estado de precreación CPROCESSSUC del aspecto, incializa los atributos que reauiere para se ejecución, y finalmente actualiza el estado para indicar que el aspecto se ha puesto en ejecución.

```
public CProcessSUC(int initialHalfSteps) : base("CProcessSUC")
{
   estado = protocolStates.CPROCESSSUC;

   this.halfsteps = initialHalfSteps;

   estado = protocolStates.MOTION;
}
```

Para precisar cómo cambian los valores de los atributos a lo largo de la vida del aspecto, se debe declarar los servicios que ocurrirán durante la ejecución del protocolo del aspecto.

5.3.2.3.4 Implementación de los servicios

En implementación, los servicios de un aspecto se corresponden con las funciones definidas en las interfaces de las que deriva el aspecto y que va a implementar.

```
AsyncResult <nombreServicio>([<parametros>])
{
   <cuerpoServicio>
}
```

Para implementar el cuerpo del servicio se define una estrategia de ejecución. Esta estrategia de ejecución se sigue durante la ocurrencia de un servicio.

1. **Transición válida de estado:** se verifica en el protocolo que exista una transición válida para el servicio requerido.

El cuerpo del servicio debe comprobar antes de llevar a cabo ninguna acción que el estado del protocolo es el correcto, es decir que el servicio se puede ejecutar. En caso de que el estado no sea correcto el servicio lanza una excepción abortando así su ejecución.

```
if (estado != protocolStates.<ESTADO>)
   throw new InvalidProtocolStateException()
```

2. **Satisfacción de precondición:** se comprueba que se cumplan las precondiciones asociadas al servicio requerido.

Una precondición es una condición que ha de satisfacerse para poder ejecutar el servicio que tiene asociada dicha precondición. Dado que *C#* no proporciona ninguna construcción sintáctica para reflejar las precondiciones, estas se corresponden a una comprobación mediante la instrucción *if…then*. En el caso que no se cumpla la precondición el servicio lanza una excepción abortando así su ejecución.

```
if (!<precondicion>)
   throw new InvalidPreconditionException()
```

3. **Satisfacción de del estado previo de la evaluación.**

Un servicio puede ejecutarse en varios estados de protocolo diferentes por lo que las acciones a llevar a cabo en el servicio pueden ser distintas en función del estado.

```
if (!<estadoPrevio>)
   throw new InvalidPreviousStateValuationException()
```

4. **Evaluaciones: se modifican los valores de los atributos afectados por la ocurrencia del servicio.**

Es este punto se lleva acabo el cambio de estado del aspecto. Se ejecutan las acciones que modifican el valor de los atributos del servicio. Puede darse también la invocación de otro servicio si lo define el protocolo.

```
<nombreAtributo> = <valor>;
```

5. **Comprobación de las restricciones de integridad: las evaluaciones del servicio deben dejar al aspecto en un estado válido.**

El cambio de valor de los atributos del aspecto puede no cumplir las restricciones de integridad especificadas sobre el estado del aspecto. En el caso que las restricciones de integridad hayan sido violadas el servicio aborta su ejecución y restaura el valor de los atributos del aspecto que tenían antes del cambio de estado. Para poder restaurar el estado anterior del aspecto se requiere tener una copia de todos los atributos afectados por las evaluaciones. Esto implica definir, dentro del servicio, una variable local del mismo tipo para cada uno de los atributos afectados. Estas variables se caracterizaran por llamarse igual que el atributo seguido de Temp.

```
private <tipoAtributo> <nombreAtributo>Temp
```

A estos atributos temporales se les asigna el valor antes de realizar las acciones o evaluaciones expuestas en el punto anterior, para poder restaurar su estado en caso de violación de una restricción de integridad.

```
<nombreAtributo1>Temp = <nombreAtributo1>
<nombreAtributo2>Temp = <nombreAtributo2>
... ...
```

En consecuencia, las restricciones de integridad se implementan de la siguiente manera:

```
if (!<restricciónIntegridad>)
{
   <nombreAtributo1> = <nombreAtributo1>Temp
```

```
   <nombreAtributo2> = <nombreAtributo2>Temp
   ...
   throw new InvalidConstraintException()
}
```

6. **Comprobación de las relaciones de disparo: después de un cambio de estado válido se verifica el conjunto de disparadores que representa la actividad interna del sistema. Si alguna de ellas se cumple, se activa el servicio correspondiente.**

La ejecución de un servicio puede implicar la ejecución de otro servicio de forma automática. Esto puede ocurrir siempre que se ejecute el servicio o en función de alguna condición. En implementación el carácter de comportamiento cliente que proporciona los disparadores se traduce en una invocación asíncrona de otro servicio del aspecto. Además, si existe una condición asociada al disparador se modela mediante la instrucción if...then.

```
if (<condicion>)
   <invocaciónServicioAspecto>
```

7. **Actualización del nuevo estado del protocolo alcanzado.**

En este punto, todas las precedentes comprobaciones han sido satisfactorias y se han efectuado cambios en los valores de los atributos del aspecto, por lo que el cambio de estado en principio inducido por el servicio se ha producido, alcanzando un nuevo estado final. El cambio de este nuevo estado se refleja actualizando la variable *estado* del aspecto con el nuevo estado.

```
estado = protocolStates.<NUEVO_ESTADO>;
```

Un ejemplo completo de la implementación de un servicio es el servicio `MoveJoint` del aspecto de coordinación `CProcessSUC`. A continuación se muestra el código fuente correspondiente a su implementación:

```
AsyncResult MoveJoint(int newHalfSteps, int speed)
{
   // Transición válida de estado
   if (estado != protocolStates.MOTION)
      throw new InvalidProtocolStateException();


   // Satisfacción de precondición
   //   NO APLICABLE

   // Satisfacción de del estado previo de la evaluación
   //   NO APLICABLE


   // Ejecución de la funcionalidad asociada
   this.temphalfsteps = newHalfSteps;


   // Comprobación de las restricciones de integridad
   //   NO APLICABLE

   // Comprobación de las relaciones de disparo
   //   NO APLICABLE
```

```
  // Nuevo estado alcanzado
  estado = protocolStates.ANSWER;
```

8. Soporte al modelo de ejecución concurrente de los aspectos

Este modelo de ejecución concurrente implica cambios en la estructura de la clase del aspecto. Los servicios del aspecto se dividen en dos partes, una parte pública y otra privada.

La parte pública de los servicios es la que corresponde con la signatura de los servicios de la interfaz en la que se definen, quedando la estrategia de ejecución de los servicios en la parte privada. Cuando se invoca un servicio público este encola la petición en la cola de petición de servicios del aspecto para que sea procesada y devuelve una referencia al cliente del tipo `AsyncResult` para que el cliente pueda recoger los resultados después de que el servicio sea procesado. Este servicio público encola la petición mediante un método genérico del aspecto que se hereda de las superclase `AspectBase`. El método `EnqueueService`, encola una referencia al servicio privado que contiene la implementación del cuerpo del servicio y todos los parámetros de este servicio.

```
public AsyncResult <nombreServicio>([<parametros>])
{
   return EnqueueService(<refServicioPrivado>, [<parametros>]);
}
```

Por ejemplo para el servicio `MoveJoint` del `CProcessSUC`,

```
public AsyncResult MoveJoint(int newHalfSteps, int speed)
{
   return EnqueueService(moveJointdelegate, newHalfSteps, speed);
}
```

Por cada servicio que implemente el aspecto, se debe definir el tipo delegado correspondiente con la signatura a ese servicio al que va a referenciar. Como los servicios que implementa el aspecto están definidos mediante interfaces en otra solución, la definición de los delegados se realiza en el mismo proyecto que las interfaces. Esto facilita la tarea de reutilización, ya que cada par de servicios público y privado tienen la misma signatura, y en el caso de reutilizar las interfaces se dispone de la definición de los delegados para cada uno de los servicios privados que va a implementar el aspecto.

El uso de delegados afecta a la implementación del aspecto ya que por cada interfaz que implementa debe declararse un atributo privado de la clase del tipo de la interfaz. Es este atributo es el que contiene mediante su instanciación la referencia al servicio privado. La instanciación de estos atributos se realiza de forma genérica por el *middleware* cuando el aspecto comienza su ejecución. Para ello se debe seguir la convención de que los nombres de los atributos deben tener el mismo nombre que el tipo del delegado pero con la primera letra en minúscula. Además, estos atributos deben estar marcados como no serializables. Esto es debido a que el *.net Framework* no deserializa correctamente los delegados manteniendo las referencias correctas

a los métodos a los que apuntan. Esto obliga a no serializar el estos atributos e instanciarlo cada vez que el aspecto entra en ejecución.

```
[NonSerialized] <nombreServicio>Delegate <nombreServicio>Delegate;
```

Por ejemplo,

```
[NonSerialized] MoveJointDelegate moveJointdelegate;
```

La parte privada contiene las acciones a realizar por el servicio. Los servicios privados se nombran igual que en la especificación del aspecto añadiendo como prefijo el símbolo "_" para distinguirlos de los servicios públicos. Debido a este modelo de ejecución, los servicios privados no tienen la necesidad de devolver la estructura *AsyncResult* puesto que se devuelve en la parte pública del aspecto pero por aumentar la reutilización de los delegados, se mantiene esta signatura.

```
private AsyncResult _<nombreServicio>([<parametros>]);
{
  <cuerpoServicio>
  return null;
}
```

Por ejemplo para el servicio MoveJoint,

```
private AsyncResult _MoveJoint(int newHalfSteps, int speed)
{
  // Transición válida de estado
  if (estado != protocolStates.MOTION)
    throw new InvalidProtocolStateException();


  // Satisfacción de precondición
  //   NO APLICABLE

  // Satisfacción de del estado previo de la evaluación
  //   NO APLICABLE


  // Ejecución de la funcionalidad asociada
  this.temphalfsteps = newHalfSteps;


  // Comprobación de las restricciones de integridad
  //   NO APLICABLE

  // Comprobación de las relaciones de disparo
  //   NO APLICABLE

  // Nuevo estado alcanzado
  estado = protocolStates.ANSWER;
  return null;
}
```

5.3.2.3.5 Played_Roles

En implementación, la invocación de un servicio se realiza con la ayuda de los *played roles*. En el modelo PRISMA, un *played role* es una vista parcial del protocolo que agrupa un subconjunto de servicios definidos en el aspecto y

que específica una función o rol determinado dentro de la funcionalidad global del aspecto que tiene sentido por sí solo.

La correspondencia de los *played roles* en implementación viene dada por una estructura dinámica definida en las superclase del aspecto (ver Figura 44). Todo aspecto hereda una colección del tipo `PlayedRolesCollection`. Esta colección es una lista dinámica que permite almacenar estructuras del tipo `PlayedRolesClass` que almacena el nombre de los servicios y las prioridades asociadas a cada servicio.

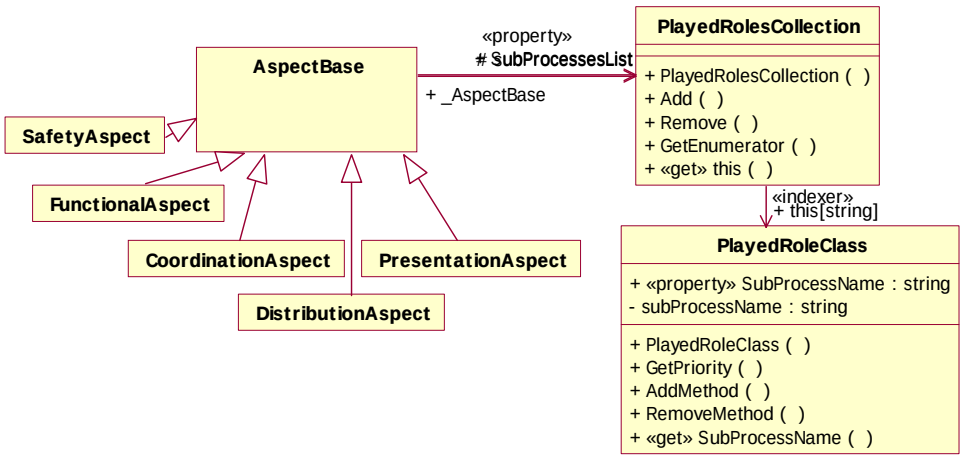


Figura 44 – Clases Played Roles en PRISMANET

Así un objeto de `PlayedRolesClass` se corresponde con un *played role* especificado con el lenguaje de descripción de arquitecturas. Por último la asignación de un *played role* a la colección de *played roles* del aspecto se realiza a través de la propiedad *playedRoleList* que permite añadir los objetos *played roles* a la lista dinámica.

Como el almacenamiento es dinámico y tiene que realizarse al principio de la vida del aspecto, la definición de los *played roles* se realiza en el cuerpo del constructor del aspecto de la siguiente forma.

```
PlayedRoleClass <playedRole> = new PlayedRoleClass("<playedRole>");
<playedRole>.AddMethod("<nombreServicio1>", <prioridad1>);
<playedRole>.AddMethod("<nombreServicio2>", <prioridad2>);
...
this.playedRolesList.Add(<playedRole>);
```

Por ejemplo, para el aspecto `CProcessSUC`, el *played role* ACT se implementa:

```
PlayedRoleClass ACT = new PlayedRoleClass("ACT");
ACT.AddMethod("MoveJoint", 10);
ACT.AddMethod("Stop", 1);
this.playedRolesList.Add(ACT);
```

En el modelo PRISMA, un *played role* de un aspecto está asociado con un puerto o rol del componente o del conector que contiene ese aspecto. En implementación esta relación se establece en el aspecto a través de las interfaces y los *played roles* de tal forma que un par interfaz-*played role*

identifican unívocamente un puerto o rol del componente que contiene el aspecto.

Así pues, una invocación de un servicio out corresponde con la invocación de un servicio a través del puerto del componente identificada por una interfaz y un *played role.* Básicamente, la invocación del servicio se realiza empleando el atributo *link* heredado de la superclase `AspectBase`. Este campo contiene la referencia a la instancia de componente que contiene el aspecto. A partir de la referencia se obtiene el puerto de salida identificado por el par interfaz-*played role,* y se añade una petición del servicio que se quiere invocar.

```
AsyncResult result =
                  link.OutPorts["<interfaz>","<playedRole>"].AddRequest
                                    ("<nombreServicio>",<argumentos>);
```

La invocación del servicio se asigna a una estructura del tipo de `AsyncResult` para poder recuperar los resultados en caso de que sea necesario.

Por ejemplo,

```
AsyncResult result =
link.OutPorts["IMotionJoint","ACT"].AddRequest("MoveJoint",args);
```

5.3.2.3.6 Transacciones

En el modelo PRISMA, las transacciones se definen como una agrupación de servicios que se ejecutan de forma atómica. Además, en la transacción los servicios están agrupados de forma arbitraria indicando el orden de ejecución en el que se deben procesar los servicios. Una transacción, de forma similar al protocolo, permite especificar el estado alcanzado después de la ejecución de cada uno de los servicios que encapsula y así determinar la secuencia de ejecución.

En implementación, una transacción se concibe como un servicio especial dentro de un aspecto. Como los servicios PRISMA, el servicio que corresponde con la transacción se implementa de la misma manera que se impone en el modelo de ejecución, tiene una parte pública y otra privada. La parte pública se implementa exactamente igual que para el resto de los servicios. La parte privada incorpora la estrategia de ejecución para las transacciones que es muy similar a la de los servicios. Esta se determina también en base a los ocho puntos vistos en el apartado 1.3.2.2.3. La única diferencia es que en el procesado de las evaluaciones, en lugar de cambiar el estado, se realiza la invocación secuencial de todas las partes privadas de los servicios. Para asegurar que la ejecución de una transacción se realiza de forma atómica, las invocaciones de los servicios que la constituyen están incluidas dentro de un bloque de tratamiento de excepciones *try…catch* para que si se produce alguna excepción en alguno de los métodos se aborte la ejecución de todos y se pueda recuperar el estado previo a la ejecución. Para recuperar el estado es necesario haberlo guardado antes, para ello se emplean variables locales al servicio privado de la transacción por cada una de los atributos del aspecto que vaya a modificar cualquier servicio encapsulado por la

transacción. En caso de que alguno de los servicios no se pueda ejecutar se recupera el estado en el bloque *catch* de código.

```
<nombreAtributo1>Temp = <nombreAtributo1>
<nombreAtributo2>Temp = <nombreAtributo2>
...
try
{
   this._nombreServicio1(<parametros>);
   this._nombreServicio2(<parametros>);
...
}
catch (Exception)
{
   <nombreAtributo1> = <nombreAtributo1>Temp
   <nombreAtributo2> = <nombreAtributo2>Temp
}
```

5.3.2.4 Patrón de Generación de código de aspectos

A partir de las correspondencias identificadas en los apartados anteriores, se ha definido un patrón de generación de código para los aspectos del modelo PRISMA.

Patrón 2 : Aspectos
Contexto
Traducción de aspectos especificados en un modelo arquitectónico mediante el lenguaje de definición de componentes al lenguaje de implementación *CSharp*.
Modelo PRISMA
tipo_aspecto **Aspect** nombre **using** $interface_1$, ... $interface_n$; **Attributes** **[Constant | Variable]** <nombre_atributo$_1$> : <tipo_atributo>; ... <nombre_atributo$_n$> : <tipo_atributo>; **[Derived]** <fórmula_derivación> **Constraints** **static** <restricciones_estáticas> **Services** **Begin** [(<arg_servicio>)] [**as** <nombre_servicio> <arg_servicio)]; **Valuations** <fórmulas_observabilidad_atributo> **[in | out]** <nombre_servicio> (<arg_servicio>)

```
     [as <nombre_servicio> (<arg_servicio)];
     Valuations
       <fórmulas_observabilidad_atributo>

   End [(<arg_servicio>)] [as <nombre_servicio> <arg_servicio)];
     Valuations
       <fórmulas_observabilidad_atributo>

Preconditions
   <fórmula_precondición_evento>

Triggers
   <fórmula_disparo_evento>

Operations [transaction]
   <fórmula_transacción>

Played Roles
   <fórmula_played_role>

Protocols
   <fórmula_protocolo>

End_ tipo_aspecto Aspect name;
```

Patrón tecnológico

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

namespace <nombreModelo>
{
   [Serializable]
   public class <nombreAspecto>: <tipoAspecto>, <nombreInterfaz>
   {
     readonly private <tipoAtributo> <nombreAtributo>
     private <tipo_atributo> <nombre_atributo>
     private <tipoAtributo> <nombreAtributo>
     private <tipoAtributo> <nombrePropiedad>
     {
```

```
        get
        {
             return <invocaciónFunciónTransformación>;
        }
      }
   enum protocolStates
   {
      <ESTADOS>
   }
   protocolStates estado;

   [NonSerialized] <nombreServicio>Delegate <nombreServicio>Delegate;
   ... ...
   [NonSerialized] <nombreTrans>Delegate <nombreServicio>Delegate;
   ... ...


   public <nombreAspecto>([<parametros>]):base()
   {
      estado = protocolStates.<PRE_CREACION>;
      <nombreAtributo> = <valor>;
      ... ...
      PlayedRoleClass <playedRole> = new
PlayedRoleClass("<playedRole>");
      <playedRole>.AddMethod("<nombreServicio>", <prioridad>);
      ... ...
      this.playedRolesList.Add(<playedRole>);
      estado = protocolStates.<NUEVO_ESTADO>;
   }
   public AsyncResult <nombreServicio>([<parametros>])
   {
      return EnqueueService(<nombreServicio>Delegate>,[<parametros>]);
   }
   ... ...
   private AsyncResult _<nombreServicio>([<parametros>]);
   {
      if (estado != protocolStates.<ESTADO>)
        throw new InvalidProtocolStateException()
      if (!<precondicion>)
        throw new InvalidPreconditionException()
      if (!<estadoPrevio>)
```

```
        throw new InvalidPreviousStateValuationException()
      private <tipoAtributo> <nombreAtributo>Temp
      ... ...
      <nombreAtributo>Temp = <nombreAtributo>
      ... ...
      <nombreAtributo> = <nuevoValor>;
      ... ...
      if (!<restricciónIntegridad>)
        {
             <nombreAtributo> = <nombreAtributo>Temp
             ... ...
             throw new InvalidConstraintException();
        }
      AsyncResult result = link.OutPorts["<interfaz>","<playedRole>"].
                          AddRequest("<nombreServicio>",<argumentos>);
      if (<condicion>)
        <invocaciónServicioAspecto>
      estado = protocolStates.<NUEVO_ESTADO>;
   }
   ... ...
   public AsyncResult <nombreTrans>([<parametros>])
   {
      return EnqueueService(<nombreTrans>Delegate>,[<parametros>]);
   }
   ... ...
private AsyncResult _<nombreTrans>([<parametros>]);
   {
      if (estado != protocolStates.<ESTADO>)
        throw new InvalidProtocolStateException()
      if (!<precondicion>)
        throw new InvalidPreconditionException()

      private <tipoAtributo> <nombreAtributo>Temp
      ... ...
      <nombreAtributo>Temp = <nombreAtributo>
      ... ...
      try
      {
        this._nombreServicio(<parametros>);
        ... ...
```

```
       if (!<restricciónIntegridad>)
            throw new InvalidConstraintException();
     }
     catch (Exception)
     {
     <nombreAtributo> = <nombreAtributo>Temp
     ... ...
     }
     AsyncResult result =
                        link.OutPorts["<interfaz>","<playedRole>"].
                        AddRequest("<nombreServicio>",<argumentos>);
     if (<condicion>)
       <invocaciónServicioAspecto>
     estado = protocolStates.<NUEVO_ESTADO>;
   }
   ... ...
   }
}
```

Caso de Estudio

Descripción

Aspecto de coordinación CProcessSUC del caso de estudio del robot 4U4 que define la coordinación entre los servicios de movimiento y los servicios de confirmación de una articulación del brazo de robot, además del servicio de parada de emergencia.

Modelo Arquitectónico PRISMA del Caso de Estudio

```
Coordination Aspect CProcessSUC using IMotionJoint, IRead, ISUC

   Attributes
     Variable
halfSteps: integer,
           NOT NULL;
       tempHalfSteps: integer;

     Derived
angle:= FtransHalfstepsToAngle();


  Services
      begin (input InitialHalfSteps: integer);
          Valuations
[begin (InitialHalfSteps)]
   halfSteps := InitialHalfsteps,
```

```
in/out movejoint(input NewHalfsteps: integer, input Speed: integer);
          Valuations
[in movejoint (NewHalfsteps, Speed)]
   tempHalfSteps := NewHalfsteps;


      in cinematicsmovejoint( input NewAngle: integer, input Speed:
integer);
          Valuations
[in cinematicsmovejoint(NewAngle, Speed)]
 tempHalfSteps := FtransAngleToHalfsteps(NewAngle);

in/out stop();

in/out moveok(output Success:[0,1]);
           Valuations
{Success=1}
[in moveok(Success)]
halfSteps:= halfSteps + tempHalSteps;
       end;

    Subprocesses

SUC= (ISUC.stop?():0
     +
     ISUC.moveJoint?(NewHalfsteps, Speed):1
     +
            ISUC.cinematicsMoveJoint?(NewAngle, Speed):1
            )
            →
            ISUC.moveOk!(Success);

ACT= ImotionJoint.stop!():0
     +
     IMotionJoint.moveJoint!(NewHalfsteps, Speed):1;


      SEN= Iread.moveOk(Success);
```

```
   End_Subprocesses;

   Protocol
CPROCESSSUC = begin(InitialHalfStep).MOTION;
MOTION = SUC.stop ?()).EMERGENCY
         +
         (SUC.movejoint?(NewHalfsteps, Speed)
   →
         ACT.movejoint!(NewHalfsteps, Speed).ANSWER)
         +
               (SUC.cinematicsmovejoint ?(NewAngle, Speed)
                →
                ACT.movejoint ! (FTransAngleToHalfSteps(NewAngle),
Speed)
                .ANSWER)
                +
                end;

      ANSWER= SEN.moveok ? (Success) → SUC.moveok!(Success).MOTION

     EMERGENCY= ACT.stop ! ()→ end;

End_Coordination Aspect CProcessSUC;
```

Patrón Implementación

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

namespace Robot {
  [Serializable]
  public class CProcessSUC : CoordinationAspect , IMotionJoint ,
IRead , ISUC
  {
    double angle;
    int halfsteps, temphalfsteps;

    enum protocolStates
    {
      CPROCESSSUC,
```

```
        MOTION,
        ANSWER,
        EMERGENCY,
        END
    }
    protocolStates estado;

    MoveJointDelegate moveJointdelegate;
    ... ...
    public CProcessSUC(int halfsteps) : base("CProcessSUC")
    {
        estado = protocolStates.CPROCESSSUC;

        this.halfsteps = halfsteps;

        PlayedRoleClass ACT = new PlayedRoleClass("ACT");
        ACT.AddMethod("MoveJoint", 10);
        ACT.AddMethod("Stop", 1);
        this.playedRolesList.Add(ACT);
        ... ...

        estado = protocolStates.MOTION;
    }

    public AsyncResult MoveJoint(int newHalfSteps, int speed)
    {
        return EnqueueService(moveJointdelegate, newHalfSteps, speed);
    }
    ... ...
    private AsyncResult _MoveJoint(int newHalfSteps, int speed)
    {
        // Transición válida de estado
        if (estado != protocolStates.MOTION)
            throw new InvalidProtocolStateException();

        // Satisfacción de precondición
        //         NO APLICABLE

        // Satisfacción de del estado previo de la evaluación
        //         NO APLICABLE
```

```
        // Ejecución de la funcionalidad asociada
        this.temphalfsteps = newHalfSteps;

        // Comprobación de las restricciones de integridad
        //         NO APLICABLE
        AsyncResult result =
link.OutPorts["IMotionJoint","ACT"].AddRequest("MoveJoint",args);

        // Comprobación de las relaciones de disparo
        //         NO APLICABLE

        // Nuevo estado alcanzado
        estado = protocolStates.ANSWER;
        return null;
     }
     ... ...
   }
}
```

5.3.3 Componentes y Conectores

Los componentes y los conectores son elementos arquitectónicos básicos del modelo PRISMA. Tanto los componentes como los conectores son tipos dentro del modelo que cumplen una función distinta. Los componentes tienen como objetivo capturar la funcionalidad del sistema, mientras que los conectores especifican la interacción entre los componentes actuando como coordinadores. En implementación se ha considerado a los componentes y a los conectores como elementos de primer orden del modelo PRISMA que tienen propiedades y características muy similares. Esto implica que la forma de implementar cualquier elemento arquitectónico básico del modelo es prácticamente idéntica, independientemente de que sea componente o conector, lo cual es una ventaja y simplifica la implementación.

Un elemento arquitectónico PRISMA integra en su interior un conjunto de tipos de aspectos distintos entre sí, y ofrece una interfaz externa que es la agregación de las interfaces de sus aspectos. Esta interfaz es publicada al resto de elementos arquitectónicos a través de los puertos (o roles)[1] que

[1] En el modelo PRISMA, los puertos están asociados a los componentes y los roles a los conectores. En implementación no existe diferencia ya que el comportamiento de los puertos para los componentes o de los roles para los conectores es idéntico.

proporciona el componente para que así puedan comunicarse e interactuar. La comunicación entre los distintos elementos arquitectónicos se realiza a través de la invocación en el elemento arquitectónico de los servicios publicados mediante los puertos. Una invocación de un servicio determinado se procesa en el interior del elemento arquitectónico gracias a la semántica definida en algún aspecto de entre todos los que se encuentran en el interior del elemento. En ocasiones, el procesado de un servicio requiere la ejecución de otros servicios definidos en otros tipos de aspecto en el interior del elemento arquitectónico. En este caso, el elemento arquitectónico se encarga de la sincronización entre los servicios de los distintos aspectos, este mecanismo se conoce como *weaving*.

En implementación, un componente o un conector se han traducido por una clase que además de integrar dinámicamente en su interior los aspectos y los puertos de comunicación, también realiza dinámicamente los *weavings* entre los aspectos y redirecciona adecuadamente las peticiones de servicio entre puertos y aspectos.

La implementación de cada uno de los tipos básicos de un modelo arquitectónico se realiza en un nuevo proyecto de biblioteca de clases para poder reutilizarlo de forma sencilla. Cada proyecto tiene el nombre del tipo especificado en el modelo arquitectónico y contiene un fichero de extensión *.cs* en el cual se encuentra la definición del elemento arquitectónico. El fichero *CSharp* empieza importando el espacio de nombres del *framework .net* que requiere, es decir *System*. Además, para la definición del tipo del elemento arquitectónico se emplean los elementos definidos en PRISMANET que dan soporte a los componentes y a los conectores, por lo que también se importan los espacios de nombres del *middleware* en donde se encuentran definidos estos elementos.

```
using System;

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

La definición de la clase del elemento arquitectónico debe estar incluida en el espacio de nombres denominado de la misma forma que el modelo arquitectónico. La definición de la clase se realiza a partir del nombre del tipo del elemento que está especificado en el modelo. Además, la clase debe derivar de una superclase del *middleware* PRISMA que le proporciona la funcionalidad genérica común para todos los componentes y conectores PRISMA. Esta clase, en PRISMANET, está definida dentro del espacio de nombres `PRISMA.Components` y se llama `ComponentBase`. Esta clase da soporte al modelo de ejecución de los elementos arquitectónicos e implementa de forma genérica los métodos para dar soporte a los distintos conceptos de un elemento arquitectónico PRISMA como los puertos, la gestión de aspectos o la gestión de *weavings*.

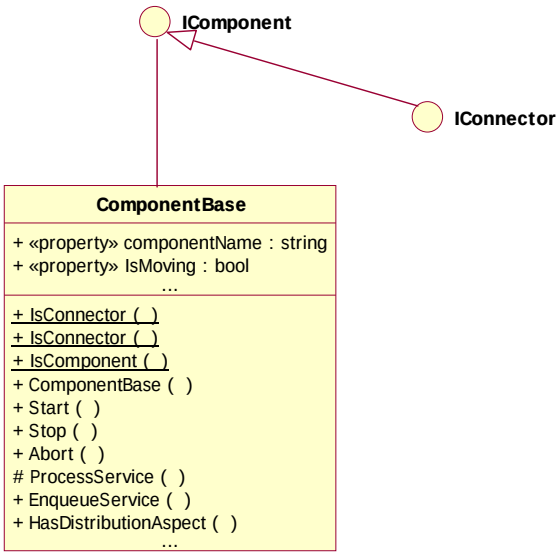


Figura 45 - ComponentBase en PRISMANET

Como se aprecia en la Figura 45, la clase `ComponentBase` implementa la interfaz `IComponent`. Esta interfaz define todos los miembros de clase que se tienen que implementar en `ComponentBase` para proporcionar toda la funcionalidad básica de los elementos arquitectónicos. Además, existe otra nueva interfaz `IConnector`, que hereda de `IComponent`. Esta interfaz no añade ningún miembro nuevo a la interfaz `IComponent` pero se emplea para diferenciar la definición de un componente o de un conector, es decir cuando se define la clase que se corresponde con un conector, se le indica a esta que implementa la interfaz `IConnector`. El objetivo de distinguir entre un componente de un conector es el de controlar que se cumplen ciertas características que los distinguen. Por ejemplo, que un conector tenga imperativamente un aspecto de coordinación y que un componente tenga el aspecto funcional, o que la comunicación entre elementos arquitectónicos se realice siempre entre un componente y un conector o viceversa.

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

namespace <nombreModelo>
{
```

- Componente:

```
  public class <nombreComponente>: ComponentBase
  {
    <cuerpoComponente>
  }
}
```

- Conector:

```
  public class <nombreConector>: ComponentBase , IConnector
  {
    <cuerpoConector>
```

```
   }
}
```

Todos elementos del modelo PRISMA pueden ser distribuidos. Por este motivo, las clases de los elementos arquitectónicos componente y conector están marcadas con el atributo `[Serializable]` que proporciona el *.net Framework*.

```
[Serializable]
public class <nombreComponente>: ComponentBase
```

En el caso de estudio del brazo de robot se especifica un conector `CnctSUC` que integra el aspecto de coordinación visto en el apartado de aspectos, además de un aspecto de Distribución y otro de Seguridad.

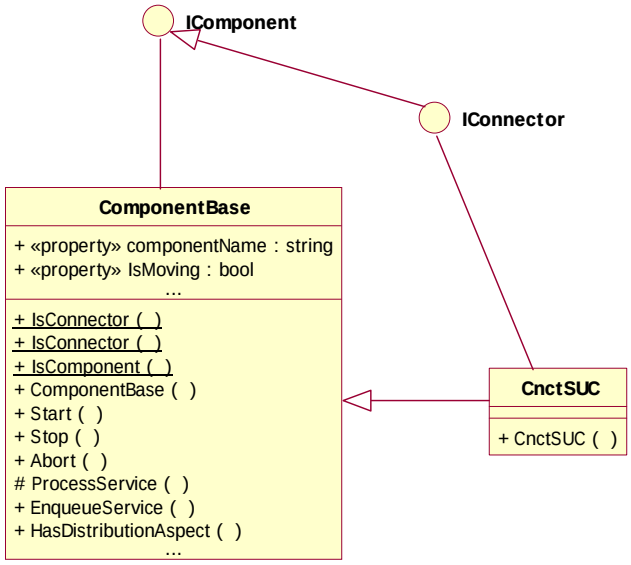


Figura 46 - Diagrama de clases de CnctSUC

En la Figura 46, se muestra como la clase del conector `CnctSUC` hereda de la superclase del *middleware* `ComponentBase` que proporciona la funcionalidad específica para elementos arquitectónicos. Además, la clase `CnctSUC` implementa la interfaz `IConnector` porque es un conector.

```
public class CnctSUC : ComponentBase , IConnector
```

En tiempo de ejecución, un objeto de una clase componente o conector representa una instancia del modelo PRISMA. Para poder instanciar un elemento arquitectónico, la clase implementa un constructor en donde se construyen o se agregan todos los conceptos PRISMA necesarios que van a constituir el elemento, es decir, se incorporan al componente todos los aspectos que integra, los *weavings* entre los servicios de estos aspecto (*weavings*) y por último, los puertos que publican los servicios.

```
public <nombreComponente>([<parametros>,] MiddlewareSystem
                          middlewareSystem):base(middlewareSystem)
{
   <cuerpoConstructor>
}
```

El constructor de la clase toma como parámetros todos los argumentos necesarios para instanciar los aspectos que van a componer el elemento arquitectónico. Además, el constructor necesita una referencia al *middleware* en donde se va a crear y se va ejecutar. Esta referencia es necesaria para invocar al constructor de la clase base `ComponentBase`, y se emplea para almacenar la localización en dónde se encuentra el elemento, pues este puede ser distribuido [Cos05].

5.3.3.1 Instanciación y gestión dinámica de aspectos de un elemento arquitectónico

Tanto los componentes como los conectores están compuestos por aspectos. Estos aspectos se agregan al componente durante su construcción. En PRISMA no estan limitados los distintos tipos de aspectos que pueden utilizarse en el desarrollo de un sistema sofware. Por lo que un elemento arquitectónico puede componerse de un número indefinido de aspectos, siempre y cuando sean de distinto tipo. Los elementos arquitectónicos almacenan sus aspectos de forma dinámica, mediante una lista de aspectos `aspectList` definida en `ComponentBase`. De esta forma, los aspectos pueden ser agregados mediante el método `AddAspect` o eliminados mediante el método `RemoveAspect`, en tiempo de ejecución sin la necesidad de tener que recompilar la definición del componente.

Antes de añadir los aspectos al elemento arquitectónico es necesario haberlos instanciado previamente. Para instanciar cada uno de los aspectos, en el cuerpo del constructor del elemento arquitectónico se invoca a los constructores de los aspectos pasándoles como argumentos todos los parámetros que necesitan.

```
new <nombreAspecto>([<parametros>])
```

Finalmente, una vez instanciados los aspectos se añaden a la lista dinámica de aspectos del elemento arquitectónico.

```
AddAspect(new <nombreAspecto>([<parametros>]));
```

Además, para poder hacer referencia a los aspectos, es necesario agregar una referencia de proyecto entre el proyecto de biblioteca de clases del elemento arquitectónico y cada uno de los proyectos de los aspectos que lo vayan a componer, ya que están definidos en proyectos diferentes.

En el caso de estudio del robot, el conector `CnctSUC` está compuesto por tres aspectos, uno de coordinación `CProcessSUC`, otro de distribución `DRobotLocation` y por último uno de seguridad `SMotion`. Para cada uno de estos aspectos se establece una referencia desde el proyecto donde está definido el conector.

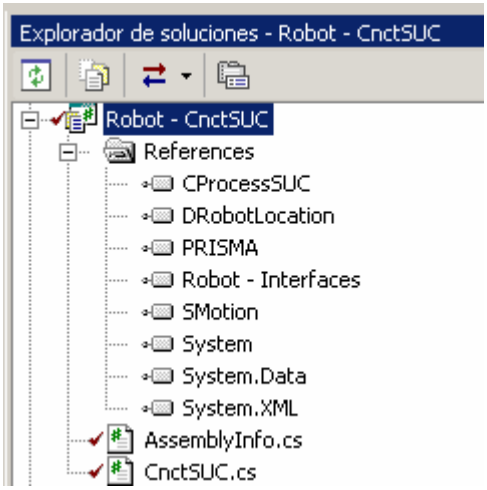


Figura 47 - Referencias de Aspectos en CnctSUC

En el constructor del conector se instancian y se añaden a su lista de aspectos cada uno de los aspectos que lo componen.

```
public CnctSUC(string name, string joint, int initialHalfsteps, double
                  initalMinimum, double initialMaximum,MiddlewareSystem
                        middlewareSystem) : base(name,middlewareSystem)
{
   AddAspect(new CProcessSUC(joint,initialHalfsteps));
   AddAspect(new DRobotLocation());
   AddAspect(new SMotion(initalMinimum,initialMaximum));
}
```

5.3.3.2 Weavings

Los *weavings* son los mecanismos de sincronización que permiten que la ejecución de un servicio de un aspecto concreto se asocie antes, después o en lugar de un servicio ofrecido por otro aspecto. Esta funcionalidad se especifica en los elementos arquitectónicos en lugar de en los aspectos. De forma que los servicios de los aspectos desconocen con qué otro servicio de otro aspecto pueden ser enlazados. De este modo se facilita la reutilización de aspectos ya que los aspectos no tienen dependencias entre sus servicios y se permite realizar sincronizaciones diferentes entre los mismos aspectos en elementos arquitectónicos distintos.

En PRISMANET existe un conjunto de clases para dar soporte a los *weavings*. Un *weaving* se asocia a un servicio de algún aspecto del elemento arquitectónico. A este servicio se le denominará servicio origen pues su invocación desencadenará la invocación de otros servicios asociados a través del *weaving*. El servicio original puede tener asociados hasta tres tipos de weavings distintos: *after*, *before* e *instead*. Se incluyen los tipos de *weaving* condicionales *afterIf* y *beforeIf* como un subtipo del *weaving after* y *before*, respectivamente. Al servicio que se ejecuta como consecuencia de la aplicación de un weaving se le denominará servicio destino. Este puede reemplazar al servicio original (*instead*), modificar los parámetros de entrada (*before*) del servicio original o modificar los parámetros de salida (*after*). Cuando se invoca el servicio de un aspecto (servicio origen), se le proporcionan los argumentos necesarios para su ejecución. Si éste tiene *weaving* asociado,

sus argumentos también serán proporcionados al servicio cuya invocación desencadene el *weaving* (servicio destino).

Todos los elementos arquitectónicos disponen de una lista dinámica para añadir o quitar *weavings*. Esta lista es del tipo `WeavingCollections` (ver Figura 48) y está definida en la clase `ComponentBase`. A partir de esta lista y con el método `AddWeaving` un elemento arquitectónico puede asociar un *weaving* a un servicio de un aspecto con otro servicio de otro aspecto diferente. De la misma forma, el componente o el conector pueden eliminar un *weaving* con el método *RemoveWeaving*.

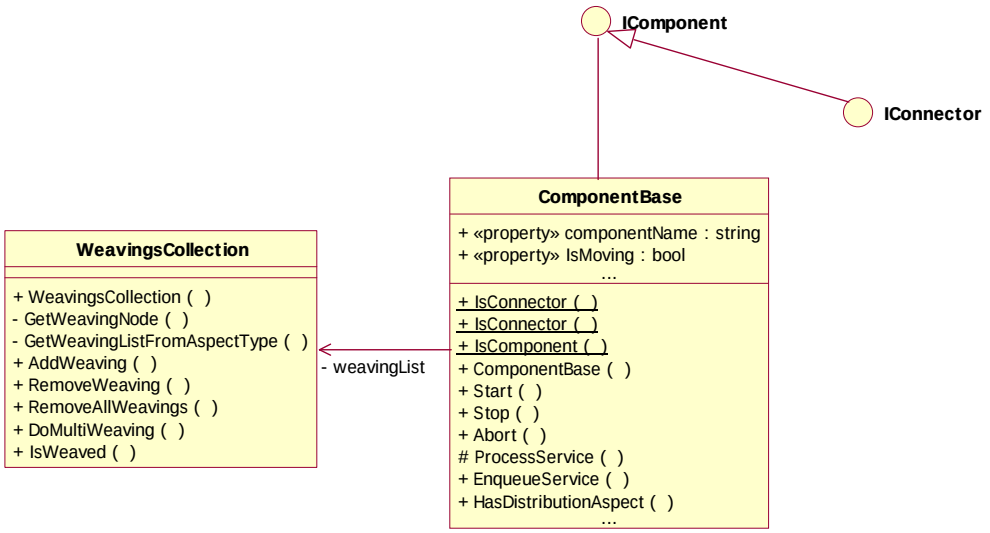


Figura 48 - WeavingsCollection en PRISMANET

La definición de un *weaving* se realiza en el cuerpo de constructor del elemento arquitectónico. A la hora de definir un *weaving* es necesario proporcionar al método `AddWeaving`: el nombre del servicio original y la instancia del aspecto donde está definido el servicio original, el nombre del servicio destino y la instancia del aspecto donde está definido el servicio destino, así como el tipo de *weaving*. La estructura definida en PRISMANET `WeavingType` permite seleccionar el tipo de *weaving* a aplicar seleccionando el tipo de *weaving*, por ejemplo en el caso de asociar un *weaving* del tipo *Before* seleccionaríamos el tipo `WeavingType.BEFORE`. Para los tipos de *weaving* condicionales *BeforeIF* y *AfterIF* también es necesario indicar la condición que se debe cumplir para procesar el *weaving*. La condición se expresa indicando entre paréntesis el nombre de un parámetro de salida del servicio y el valor que debe tener para que el *weaving* se procese, por ejemplo `WeavingType.AFTERIF("x",1)`que indica que se procesaría el *weaving* si el valor del parámetro de salida es igual a "1". Además, como las instancias de los aspectos agregados al elemento arquitectónico se encuentran en la lista dinámica del componente `aspectList`, se emplea el método `GetAspect`, definido en ComponentBase, para obtener la referencia de un aspecto de un tipo determinado (Funcional, Coordinación...). Por ejemplo, si se quisiera obtener la referencia del aspecto de coordinación de un conector se emplearía `GetAspect(typeof(CoordinationAspect))`.

```
AddWeaving(GetAspect(typeof(<tipoAspectoOrigen>),"<servicioOrigen>",
```

```
       WeavingType.<tipoWeaving>[("parametro",valor)],
       GetAspect(typeof(<tipoAspectoDestino>),"<servicioDestino>”);
```

En el caso de estudio del brazo de robot se especifica un *weaving* dentro del conector CnctSUC entre el servicio origen *CinematicsMoveJoint* del aspecto de coordinación y el servicio destino *Check* del aspecto de seguridad. Este *weaving* es de tipo AFTERIF y tiene como condición que el parámetro de salida *secure* del servicio *check* sea igual a *true*. En caso de que la condición se cumpla una vez el servicio *check* sea ejecutado, se procesará el *weaving*, por lo que se ejecutará el servicio *CinematicsMoveJoint*. El propósito de este *weaving* es el de comprobar si el ángulo en el que se quiere posicionar la articulación del robot es un ángulo válido que el robot puede alcanzar.

```
AddWeaving(GetAspect(typeof(CoordinationAspect),"CinematicsMoveJoint",
     WeavingType.AFTERIF("secure",true),GetAspect(typeof(SafetyAspect),
     "Check");
```

En un *weaving* los parámetros del servicio original se pasan al servicio destino para que pueda ejecutarse. Pero existe la posibilidad de que los servicios no tengan en su signatura ni el mismo número de parámetros ni del mismo tipo. Este problema se ha resuelto en PRISMANET a través de funciones de transformación de parámetros. Las funciones de transformación de parámetros permiten asociar a cada parámetro de un método destino una función. Dicha función toma como entrada uno o varios parámetros definidos para el método origen y devuelve un objeto del mismo tipo que el parámetro del método destino al cual está asociada. Las funciones de transformación están almacenadas en un repositorio donde se pueden añadir nuevas funciones. En caso de que sea necesario aplicar una función en un *weaving*, el elemento arquitectónico almacenará una referencia (delegado) a la función de transformación y lo asignará a un parámetro del servicio destino por medio de una instancia de la clase `DelegateFunctionsCollections` implementada en PRISMANET. A la hora de crear una instancia de esta clase se proporciona al constructor qué servicio destino y de qué aspecto va a emplear funciones de transformación.

```
DelegateFunctionsCollection <instancia> = new
DelegateFunctionsCollection(GetAspect(typeof(<tipoAspectoDestino>),
"<servicioDestino>");
```

Una vez creada la instancia se le indica qué parámetro del servicio destino va a emplear y se le asocia la función de transformación.

```
<instancia>["parametroDestino"] = new <nombreFuncion>Delegate
(<nombreFuncion>);
... ...
```

Por último, en la definición del *weaving* se añade al método `AddWeaving`, la instancia de `DelegateFunctionsCollections` para que las funciones se puedan aplicar en la ejecución.

```
AddWeaving(GetAspect(typeof(<tipoAspectoOrigen>),"<servicioOrigen>",
       WeavingType.<tipoWeaving>[("parametro",valor)],
  GetAspect(typeof(<tipoAspectoDestino>),"<servicioDestino>",
                                                  <instancia>);
```

Para el conector `CnctSUC` del caso de estudio del robot también se define un *weaving* entre el servicio origen `MoveJoint` del aspecto de coordinación y el servicio destino `Check` del aspecto de seguridad. Este *weaving* es de tipo AFTERIF y tiene como condición que el parámetro de salida `secure` del servicio `Check` sea igual a *true*. En este *weaving* se aplica una función de transformación que convierte el argumento `haflsteps` de `MoveJoint` a un ángulo y se le asocia al parámetro `newAngle` de `Check`, para que se pueda comprobar en el servicio `Check` que el movimiento es seguro.

```
DelegateFunctionsCollection functions = new
       DelegateFunctionsCollection(safetyAspect, "Check");

functions["newAngle"] = new CProcessSUC.FtransAngleDelegate(
           ((CProcessSUC)coordinationAspect).FtransAngle);

AddWeaving(coordinationAspect,"MoveJoint",
  WeavingType.AFTERIF("secure",true),safetyAspect,"Check",functions);
```

5.3.3.3 Puertos y Roles

Los puertos y roles son entidades agregadas a los componentes y a los conectores respectivamente. Estas entidades publican al exterior del elemento arquitectónico cada una de las interfaces que implementan los aspectos del componente o del conector para que puedan comunicarse e interaccionar entre ellos. En el modelo PRISMA existe una correspondencia directa entre una interfaz y un puerto (o rol), es decir cada elemento arquitectónico tendrá un puerto (o rol) por cada una de las interfaces que implementan sus aspectos agregados.

En implementación, el concepto de puerto y rol es el mismo debido a que su diferencia en el modelo PRISMA es únicamente conceptual. De esta forma tanto un puerto como un rol tienen la misma funcionalidad por lo que todo las características de los puertos se pueden aplicar a los roles.

Cada uno de los puertos del modelo se corresponde en PRISMANET con dos puertos, un puerto de entrada y un puerto de salida. Esta división se ha realizado para separar la funcionalidad cliente de la de servidor implícita en un sólo puerto del modelo. Un puerto de entrada publica los servicios de una interfaz que ofrece el componente y recibe las peticiones de otros elementos arquitectónicos. Mientras que un puerto de salida se encarga de entregar las peticiones generadas por el componente a los otros elementos externos con los cuales quiere comunicarse.

5.3.3.3.1 Implementación de puertos y roles

La implementación de un puerto (o rol) del modelo PRISMA se efectúa definiendo dos nuevas clases, una para definir la funcionalidad del puerto de entrada y otra para definir el comportamiento del puerto de salida. Puesto que un puerto tiene una relación de dependencia con una interfaz, puesto cada puerto de entrada y de salida publican los servicios de una única interfaz. Por este motivo, la definición de las clases de los puertos se realiza en el mismo proyecto que las interfaces pero en ficheros distintos. Así, se define la clase del puerto de entrada en un fichero y la clase del puerto de salida en otro fichero.

Cada fichero *CSharp* empieza importando el espacio de nombres del *framework* de *.net* que requiere, es decir *System*. Además, para la definición de la clase del puerto se emplean los elementos definidos en PRISMANET que dan soporte a los puertos, por lo que también se importan los espacios de nombres del *middleware* en donde se encuentran definidos estos elementos.

La clase base `Ports` definida en el *middleware* (ver Figura 49) proporciona la funcionalidad genérica común para los puertos de entrada y los puertos de salida. La clase `Ports` principalmente contiene el nombre que en ejecución tendrá una instancia del puerto, la interfaz que implementa y un mecanismo de identificación del puerto basado en el nombre de la interfaz que implementa y el nombre del *played rol* que va jugar cuando se esté ejecutándose.

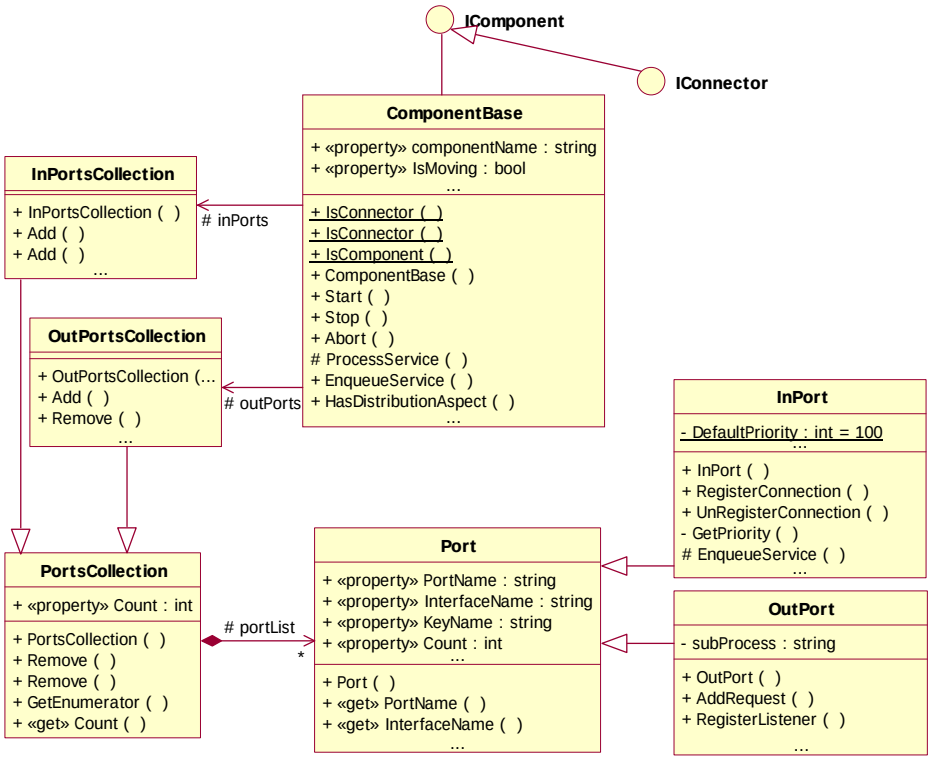


Figura 49 - Superclases para Puertos (Roles) en PRISMANET

5.3.3.3.1.1 Puertos de entrada

El comportamiento genérico de los puertos de entrada se define en la clase `InPort` de PRISMANET, que a su vez hereda de la clase `Port`. Un puerto de entrada, en el momento de su creación, se asocia al aspecto el cual implementa los servicios que él publica. Estos servicios pueden tener una prioridad asignada, que vendrá determinada por los *played roles* del aspecto.

Para definir un tipo de puerto de entrada es necesario crear una nueva clase que herede su comportamiento de la clase `InPort`. En esta clase se implementa la especificación de los servicios de la interfaz que publica el puerto. Por este motivo el nombre de la clase es `InPort<interfaz>`. Además la clase debe ser serializable para que en caso de que el componente sea distribuido o sea movido el puerto también lo pueda ser.

```
[Serializable]
public class InPort<interfaz> : InPort, <interfaz>
{
  <cuerpoPuertoEntrada>
}
```

Para poder instanciar un puerto de entrada, la clase implementa un constructor que recoge toda la información necesaria para la construcción del puerto de entrada como el nombre, la interfaz que publica, el *played role* con la información de los servicios de la interfaz, el aspecto donde se implementan los servicios, y el componente al que va a pertenecer.

```
public InPortIMotionJoint(string inPortName, string interfaceName,
       PlayedRole methods,IAspect aspect, IComponent component)
       : base(inPortName, interfaceName, methods, aspect, component){}
```

Además, en el cuerpo del puerto de entrada se definen los métodos que se encargan de redireccionar las invocaciones de los servicios publicados hacia el interior del elemento arquitectónico, es decir hacia los aspectos que implementan los servicios. Estos métodos tiene la misma signatura que los servicios definido en la interfaz por lo que habrá un método por cada servicio.

```
public AsyncResult <nombreServicio>([<parametros>])
{
  lock(this)
  {
    <nombreServicio>Delegate <nombreServicio>Delegate = null;

    if (aspect!=null)
    <nombreServicio>Delegate = new <nombreServicio>Delegate((
                (<interfaz>)this.aspect).<nombreServicio>);

    AsyncResult result = this.EnqueueService
    ("<nombreServicio>",<nombreServicio>Delegate,<parametros>);

    return result;
  }
}
```

Para el conector `CnctSUC` del caso de estudio de robot se especifica un rol, entre otros, que publica los servicios de la interfaz `IMotionJoint`. La implementación del servicio `MoveJoint` definido en esta interfaz se implementaria de la siguiente manera.

```
public AsyncResult MoveJoint(int newHalfSteps, int speed)
{
  lock(this)
  {
    MoveJointDelegate moveJointDelegate = null;

    if (aspect!=null)
    moveJointDelegate = new MoveJointDelegate(((IMotionJoint)
      this.aspect).MoveJoint);

    AsyncResult result = this.EnqueueService
      ("MoveJoint",moveJointDelegate,newHalfSteps,speed);

    return result;
  }
```

```
}
```

Finalmente, el patrón generación de código para los patrones quedaría en base a las correspondencias definidas anteriormente, de la siguiente forma:

Patrón 3 : Puertos de entrada

Contexto

Traducción de los puertos del modelo PRISMA al lenguaje de programación CSharp.

Modelo PRISMA

```
[Port | Role]
  <nombre_i> : <interfaz_i>
  [Played_Role <nombre_aspecto>.<nom_playedRole_del_aspecto>];
End_[Port | Role];
```

Patrón tecnológico

```
using System;

using PRISMA;
using PRISMA.Aspects;
using PRISMA.Components;
using PRISMA.Components.Ports;
namespace <nombreModelo>
{
[Serializable]
public class InPort<interfaz> : InPort, <interfaz>
{
  public InPortIMotionJoint(string inPortName, string interfaceName,
           PlayedRole methods,IAspect aspect, IComponent component)
           : base(inPortName, interfaceName, methods, aspect,
                                                    component){}
  public AsyncResult <nombreServicio>([<parametros>])
  {
    lock(this)
    {
      <nombreServicio>Delegate <nombreServicio>Delegate = null;

      if (aspect!=null)
      <nombreServicio>Delegate = new    <nombreServicio>Delegate((
                    (<interfaz>)this.aspect).<nombreServicio>);

      AsyncResult result = this.EnqueueService
      ("<nombreServicio>",<nombreServicio>Delegate,<parametros>);

      return result;
```

```
    }
  }
}
}
```

Caso de Estudio

Descripción

Puerto de entrada que publica los servicios de la interfaz IMotioJoint

Modelo Arquitectónico PRISMA del Caso de Estudio

```
Connector SUCconnector

   Roles
      ControlActuator: IMotionJoint,
           Played_Roles CProcessSUC.ACT;
   End_Roles;
```

Patrón Implementación

```
using System;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Components;
using PRISMA.Components.Ports;

namespace Robot
{
  [Serializable]
  public class InPortIMotionJoint : InPort, IMotionJoint
  {
    public InPortIMotionJoint(string inPortName,string interfaceName
                      ,SubProcessClass methods,IAspect aspect,
                      IComponent component) : base(inPortName,
                      interfaceName,methods,aspect,component) {}
    public AsyncResult MoveJoint(int newHalfSteps, int speed)
    {
           lock(this)
           {
                 MoveJointDelegate moveJointDelegate = null;
                 object[] args = new object[2];
                 args[0] = newHalfSteps;
                 args[1] = speed;
                 if (aspect!=null)
                       moveJointDelegate = new
MoveJointDelegate(((IMotionJoint) this.aspect).MoveJoint);
```

```
                AsyncResult result =
this.EnqueueService("MoveJoint",moveJointDelegate, args);
                return result;
            }
    }
    public AsyncResult Stop()
    {
            lock(this)
            {
                StopDelegate stopDelegate = null;
                object[] args = new object[0];
                if (aspect!=null)
                      stopDelegate = new
StopDelegate(((IMotionJoint) this.aspect).Stop);
                AsyncResult result =
this.EnqueueService("Stop",stopDelegate, args);
                return result;
            }
    }
  }
}
```

5.3.3.3.1.2 Puertos de salida

Los puertos de salida se han diseñado para permitir la invocación de servicios por parte de los aspectos de un elemento arquitectónico a los demás elementos arquitectónicos, sin que para ello los aspectos deban conocer el elemento arquitectónico concreto con el que se están comunicando, y así mantener la reutilización de los aspectos. En PRISMANET, un puerto de salida se encuentra implementado en la clase `OutPort`, y al igual que los puertos de entrada se identifica por el nombre del puerto o bien por la interfaz que implementa y el `playedRole` que representa.

Al contrario que para los puertos de entrada, en implementación no es necesario crear una clase nueva que herede de la clase `OutPorts` para cada interfaz. Gracias a los mecanismos de reflexión que proporciona el *.net Framework*, cuando se le asocia un puerto de salida a un componente, este ya conoce los servicios que pública la interfaz asignada al puerto, por lo que podrá invocarlos desde los aspectos.

Para realizar la invocación de un servicio desde un aspecto hacia otro elemento arquitectónico, el aspecto el atributo *link* heredado de la superclase `AspectBase`. El atributo `link` de los aspectos es la referencia que éstos tienen hacia el componente que los agrega, mediante la cual pueden acceder al atributo `OutPorts` que contiene la lista de puertos de salida que forman parte

del componente. El aspecto obtiene la referencia del puerto en el que depositar la petición de un servicio a través de la interfaz que publica y el *played_role* que representa.

```
link.OutPorts["<interfaz>","<playedRole>"].AddRequest("<nombreServicio
>",<argumentos>);
```

En el modelo PRISMA, el protocolo que se especifica en el aspecto utilizando π-cálculo poliádico [Mil91] indica cuando un servicio de un aspecto debe invocar a otro servicio a través del puerto de salida.

En el ejemplo del caso de estudio del brazo de robot, en el servicio MoveJoint del aspecto de coordinación CProcessSUC, se efectua una invocación a través de un puerto de salida identificado por la interfaz IMotionJoint y el subproceso ACT que invoca el servicio MoveJoint también pero del componente Actuador.

```
link.OutPorts["IMotionJoint","ACT"].AddRequest("MoveJoint",args);
```

5.3.3.3.2 Gestión dinámica de puertos y roles

De igual forma que para los aspectos y los *weaving*, los puertos de entrada y de salida se agregan dinámicamente en el constructor del elemento arquitectónico. Todos los elementos arquitectónicos disponen de dos listas dinámicas para añadir o quitar puertos de entrada `InPorts` y salida `OutPorts` respectivamente. De esta forma, los puertos pueden ser agregados mediante el método `Add` o eliminados mediante el método `Remove`, en tiempo de ejecución sin la necesidad de tener que recompilar la definición del elemento arquitectónico.

Para agregar puerto de entrada el método Add requiere como parámetros el nombre del puerto, el nombre de la interfaz que publica, el aspecto de donde están implementados los servicios de la interfaz y por último el nombre del *played role* asociado al puerto.

```
InPorts.Add("<Puerto>", "<Interfaz>", <aspecto>, "<played_role>");
```

Sin embargo, para añadir un puerto de salida a la lista dinámica del elemento arquitectónico no es necesario indicar el aspecto.

```
OutPorts.Add("<Puerto>", "<Interfaz>", "<played_role>");
```

El conector CnctSUC incorpora tres tipos de puertos distintos para interaccionar con otros elementos arquitectónicos. El puerto `ImotionJointPort` publica los servicios de la interfaz `IMotionJoint` y que se encuentran implementados en el aspecto de coordinación `CProcessuc`. Además este puerto está asociado con el subproceso `ACT` del aspecto.

```
InPorts.Add("IMotionJointPort",   "IMotionJoint",   coordinationAspect,
"ACT");

OutPorts.Add("IReadPort", "IRead", "SEN");
```

5.3.3.3.3 Modelo de ejecución

5.3.3.3.3.1 Puertos de entrada

En tiempo de ejecución, los puertos de entrada se encargan de redireccionar las solicitudes de servicio, que le llegan procedentes de otros elementos arquitectónicos, hacia el interior del componente al que pertenece. En el puerto, una petición de servicio es una invocación de un servicio implementado en la clase `InPort<interfaz>`.

```
public AsyncResult <nombreServicio>([<parametros>])
{
   lock(this)
... ...
```

Este servicio está protegido con un monitor (*lock*) para evitar que dos peticiones de un mismo puerto se efectúen simultáneamente. Seguidamente, el servicio construye la petición del servicio que, como en el caso de los aspectos, es un delegado al servicio implementado por el aspecto y que el puerto de entrada conoce.

```
<nombreServicio>Delegate = new <nombreServicio>Delegate((
                  (<interfaz>)this.aspect).<nombreServicio>);
```

Por último, el servicio introduce en la cola de peticiones pendientes de l componente mediante el método `EnqueueService` definido en la clase `InPort` de PRISMANET.

```
AsyncResult result = this.EnqueueService
     ("<nombreServicio>",<nombreServicio>Delegate,<parametros>);
```

Además `EnqueService` devuelve una referencia de tipo AsyncResult para poder consultar, si es necesario, los resultados de la invocación del servicio. Esta referencia la devuelve el servicio del puerto al elemento que ha efectuado la solicitud.

```
return result;
```

5.3.3.3.3.2 Puertos de salida

En tiempo de ejecución, los puertos de salida recogen las peticiones de servicios, que efectúan los aspectos desde el interior de un elemento arquitectónico, para solicitar el servicio a otro elemento arquitectónico. Cuando los aspectos desean solicitar un servicio de otro componente, depositan sus peticiones en el puerto de salida.

```
AsyncResult result =
                  link.OutPorts["<interfaz>","<playedRole>"].AddRequest
                                    ("<nombreServicio>",<argumentos>);
```

Las peticiones se almacenan en una cola, con la finalidad de que sean procesadas en el mismo orden que fueron enviadas. Estas peticiones son dirigidas hacia el puerto de entrada de entrada de otro elemento arquitectónico que publique el servicio de la petición. Finalmente, la invocación del método `AddRequest` devolverá un objeto `AsyncResult` para procesar posteriormente los resultados obtenidos.

5.3.3.4 Modelo de ejecución

El modelo de ejecución de los elementos arquitectónicos es concurrente. Esto implica que los elementos arquitectónicos se ejecuten de forma concurrente y de forma autónoma entre sí. Los mecanismos que proporcionan este modelo están definidos como parte de la funcionalidad básica, que hereda un alimento arquitectónico de las superclases del *middleware* PRISMA. Para que los elementos arquitectónicos se ejecuten concurrentemente, las superclases los dotan de un hilo de ejecución propio para ir procesando los servicios que se le requieran. También, para salvaguardar el orden de las solicitudes que entran por los puertos de entrada, en función de su prioridad, cada elemento dispone de una cola de solicitudes de servicios pendientes en la cual se almacenan las distintas peticiones a medida que son recibidas por los puertos. El hilo de ejecución, se encarga de comprobar si existen solicitudes pendientes en la cola y de ir distribuyéndolas a los aspectos que contiene secuencialmente. Gracias a la cola, se garantiza que se ejecuten todos los servicios correctamente y en orden adecuado (ver Figura 50).

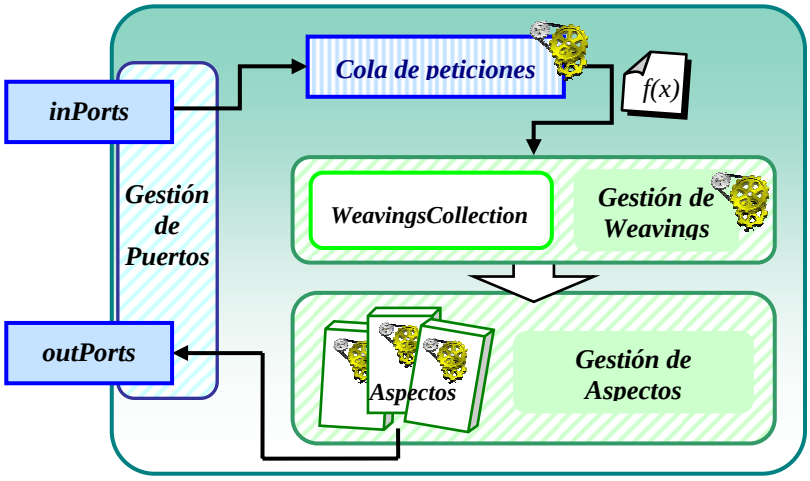


Figura 50 - Estructura interna de un componente

En el momento en el que el hilo de un elemento arquitectónico recoge una petición de servicio para enviarla a un aspecto, comprueba si el servicio tiene *weaving*. De ser así, los distintos servicios, es decir, el método origen y los distintos métodos destino implicados deben ser ejecutados. Esta ejecución es secuencial y síncrona debido a que la propia definición de *weaving* establece un orden de ejecución de los distintos servicios, de lo que se deriva que hasta que un servicio no se haya procesado completamente no podrá procesarse el siguiente. Debido a esto, la ejecución de los servicios se efectúa en un nuevo hilo de ejecución exclusivo para el *weaving* y así liberar el hilo del componente. De esta forma, el hilo del componente puede atender a otras peticiones de servicio encoladas.

5.3.3.5 Patrón Implementación

Patrón 3 : Componentes y Conectores
Contexto
Traducción de componentes PRISMA al lenguaje de implementación *CSharp*.

Modelo PRISMA

```
[Component | Connector]_type <nombre_Componente|Conector>

[Port | Role]
  <nombre_i> : <interfaz_i>
  [Played_Role <nombre_aspecto>.<nom_playedRole_del_aspecto>];
End_[Port | Role];

[Functional | Coordination] Aspect Import <nombre_aspecto_j>;
<tipo_aspecto_k> Aspect Import <nombre_aspecto_k>;

[Weaving
    <nombre_aspecto_k>.<nombre_servicio_k>
          <operador_weaving>
    <nombre_aspecto_n>.<nombre_servicio_n>;
End_Weaving;]

Initialize
  New(<argumentos>) {
    <nombre_aspecto_i>.begin(<args_constructor_i>);
  }
End_Initialize;
```

Patrón tecnológico

```
using System;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;
namespace <nombreModelo>
{
[Serializable]
public class <Componente> | <Conector> : ComponentBase , IConnector
{
  public <Componente>([<parametros>,] MiddlewareSystem
                      middlewareSystem):base(middlewareSystem)
  {
    AddAspect(new <nombreAspecto>([<parametros>]));
    ... ...
```

```
    [DelegateFunctionsCollection <instancia> = new
           DelegateFunctionsCollection(GetAspect(typeof(
           <tipoAspectoDestino>), "<servicioDestino>");]
    <instancia>["parametroDestino"] = new<nombreFuncion>Delegate(
           <nombreFuncion>);
    ... ...]
    AddWeaving(GetAspect(typeof(<tipoAspectoOrigen>),
           "<servicioOrigen>",WeavingType.<tipoWeaving>
           [("parametro",valor)],  GetAspect(typeof(
           <tipoAspectoDestino>),"<servicioDestino>”[,<instancia>]);
    ... ...
    InPorts.Add("<Puerto>", "<Interfaz>", <aspecto>,
           "<played_role>");
    OutPorts.Add("<Puerto>", "<Interfaz>", "<played_role>");
  }
}
}
```

Caso de Estudio

Descripción

Componente CnctSUC del caso de estudio del robot 4U4 que integra el aspecto de coordinación CProcessSUC, el aspecto de Distribución DRobotLocation y el aspecto de Seguridad SMotion.

Modelo Arquitectónico PRISMA del Caso de Estudio

```
Connector SUCconnector

   Roles
      SUC: ISUC,
           Played_Roles CProcessSUC.SUC;
      ControlActuator: IMotionJoint,
           Played_Roles CProcessSUC.ACT;
      ControlSensor:IRead,
           Played_Roles CProcessSUC.SEN;
   End_Roles;

   Coordination Aspect Import CProcessSUC;
   Distribution Aspect Import DLocation;
   Safety Aspect Import Smotion;

  Weavings
```

```
 CProcessSUC.movejoint(NewHalfsteps, Speed)
 afterIf (Secure = true)
 SMotion.Check(FTransHalfstepsToAngle(NewHalfsteps), Secure);

       CProcessSUC.cinematicsmovejoint( NewAngle, Speed);
       afterIf (Secure = true) SMotion.Check(NewAngle, Segure);

   End_Weavings;
   Initialize
     new (HalfSteps: integer, Location: loc,
Minimum:integer,Maximum:integer)
   {
   CProcessSUC.begin(HalfSteps: integer);
   DLocation.begin(Location: loc);
   Smotion.begin(Minimum:integer, Maximum:integer);
   }
   End_Initialize;
   Destruction
     destroy ()
   {
   CProcessSUC.end();
   DLocation.end();
   Smotion.end();
   }
   End_Destruction;


End_Connector SUCconnector;
```

Patrón Implementación

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

namespace Robot {
   [Serializable]
   public class CnctSUC : ComponentBase , IConnector {
```

```
    public CnctSUC(string name,string joint,int initialHalfsteps,
      double initalMinimum,double initialMaximum,
      MiddlewareSystem middlewareSystem) :
      base(name,middlewareSystem)
    {
      AddAspect(new CProcessSUC(joint,initialHalfsteps));
      AddAspect(new DRobotLocation());
      AddAspect(new SMotion(initalMinimum,initialMaximum));
      DelegateFunctionsCollection functions = new
           DelegateFunctionsCollection(
           GetAspect(typeof(SafetyAspect)),"Check");
      functions["newAngle"] = new CProcessSUC.FtransAngleDelegate(
           ((CProcessSUC)GetAspect(
           typeof(CoordinationAspect))).FtransAngle);
      AddWeaving(GetAspect(typeof(CoordinationAspect)),"MoveJoint",
           WeavingType.AFTERIF("secure",true),
           GetAspect(typeof(SafetyAspect)),"Check",functions);
      AddWeaving(GetAspect(typeof(CoordinationAspect)),
           "CinematicsMoveJoint",WeavingType.AFTERIF("secure",true),
           GetAspect(typeof(SafetyAspect)),"Check");
      InPorts.Add("ISUCPort", "ISUC",
           GetAspect(typeof(CoordinationAspect)), "SUC");
      InPorts.Add("IMotionJointPort", "IMotionJoint",
           GetAspect(typeof(CoordinationAspect)), "ACT");
      InPorts.Add("IReadPort", "IRead",
            GetAspect(typeof(CoordinationAspect)), "SEN");
      OutPorts.Add("ISUCPort", "ISUC", "SUC");
      OutPorts.Add("IMotionJointPort", "IMotionJoint", "ACT");
      OutPorts.Add("IReadPort", "IRead", "SEN");
    }
  }
}
```

5.3.4 Sistemas

En el modelo PRISMA un sistema es un componente complejo que permite la composición de los elementos arquitetónicos básicos del modelo. De esta forma un sistema permite aumentar el nivel de abstracción del modelado. Desde el punto de vista arquitectónico, un sistema aporta toda la funcionalidad

de un componente básico pero además proporciona propiedades adicionales. Al igual que los componentes simples, los sistemas capturan la funcionalidad de los sistemas de información a través del conjunto de aspectos que lo forman y de los elementos arquitectónicos que contiene y sus respectivas conexiones. Por este motivo un sistema PRISMA es un componente complejo formado por aspectos, weaving y puertos; y también por componentes, conectores y sus respectivas conexiones. Las relaciones entre los elementos arquitectónicos se definen a a nivel de sistema. Estas conexiones permiten la comunicación entre los elementos y pueden ser de dos tipos. En primer lugar existen conexiones entre elementos del mismo nivel de granularidad: *attachments*. Estas conexiones son el canal de comunicación entre los componentes y conectores. En segundo lugar conexiones entre elementos arquitectónicos de distinto nivel de granularidad: *bindings*. Estos establecen la comunicación entre los elementos arquitectónicos del sistema y el propio sistema.

Al igual que para la implementación de otros conceptos del modelo PRISMA, en PRISMANET existe un conjunto de clases que proporcionan la funcionalidad genérica de los sistemas. La clase principal es `SystemBase` (ver Figura 51). Esta clase hereda de `ComponentBase` para así extender el comportamiento de los elementos arquitectónicos básicos añadiendo las propiedades adicionales que caracterizan a los sistemas. Los métodos que que dan soporte a estas propiedades están definidas en la interfaz `ISystem` que implementa `SystemBase`. Dichos servicios permiten añadir o eliminar dinamicamente componentes (`AddComponent`, `RemoveComponent`), conectores (`AddConnector`, `RemoveConnector`), *attachments* (`AddAttachment`, `RemoveAttachment`) o *bindings* (`AddBinding`, `RemoveBinding`). Así, la clase `SystemBase` mantiene todas las entidades PRISMA que componen el sistema en listas dinámicas `componentsList`, `connectorsList`, `attachmentsList` y `bindingsList`.

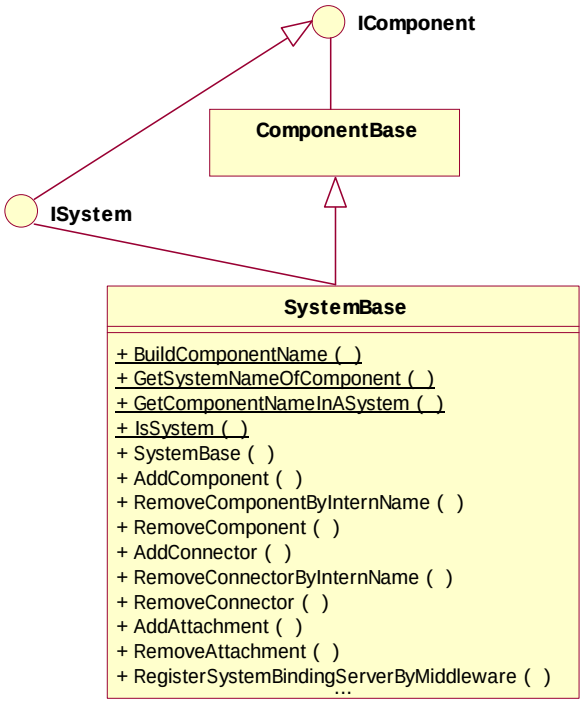


Figura 51 - SystemBase en PRISMANET

Al igual que los componentes, la implementación de cada uno de los sistemas de un modelo arquitectónico se realiza en un nuevo proyecto de biblioteca de clases. Cada proyecto se denomina igual que el tipo especificado en el modelo arquitectónico y que se va a implementar, y contiene un fichero de extensión *.cs* en el cual se encuentra la clase que corresponde con el tipo del sistema. El fichero *CSharp* empieza importando el espacio de nombres del *framework* de *.net* que requiere, es decir *System*. Además, para la definición del tipo se emplean los elementos definidos en PRISMANET que dan soporte a los sistemas, por lo que también se importan los espacios de nombres del *middleware* en donde se encuentran definidos estos elementos.

```
using System;
using PRISMA;
using PRISMA.Components;
using PRISMA.Middleware;
```

Como en la implementación de los componentes, la definición de la clase del sistema debe estar incluida en el espacio de nombres denominado de la misma forma que el modelo arquitectónico. La definición de la clase se realiza a partir del nombre del tipo del sistema que está especificado en el modelo. Además, la clase debe derivar de la superclase `SystemBase`. Esta clase, en PRISMANET, está definida dentro del espacio de nombres `PRISMA.Components`. De la misma forma que los componentes, un sistema puede ser distribuido o móvido de emplazamiento físico, por lo que la clase también es serializable.

```
namespace <nombreModelo>
{
  public class <nombreSistema>: SystemBase
```

```
  {
    <cuerpoSistema>
  }
}
```

En tiempo de ejecución, un objeto de una clase del sistema representa una instancia de un sistema PRISMA. Para poder instanciarla, la clase implementa un constructor en donde se construyen o se agregan todos los conceptos PRISMA necesarios, vistos anteriormente, que van a constituir el sistema.

```
public <nombreSistema>([<parametros>,] MiddlewareSystem
                              middlewareSystem):base(middlewareSystem)
{
  <cuerpoConstructor>
}
```

El constructor de la clase toma como parámetros todos los argumentos necesarios para instanciar los elementos arquitectónicos y los aspectos que van al sistema. Además, como para los componentes, el constructor necesita una referencia al *middleware* en dónde se va a crear y se va ejecutar. Además, el constructor debe proporcionar el nombre de la instancia del sistema que se va a crear. Este nombre permite a PRISMANET identificar cada sistema del modelo arquitectónico de de forma unívoca independientemente de su ubicación física.

```
public <nombreSistema>( string name,[<parametros>,] MiddlewareSystem
                        middlewareSystem):base(name,middlewareSystem)
{
  <cuerpoConstructor>
}
```

En el cuerpo del constructor todas las entidades que forman parte de un sistema (componentes, conectores, *attachments* y *bindings*) son instanciadas. Esto es posible gracias a los servicios que proporciona PRISMANET para crear, mantener y eliminar todos los conceptos PRISMA. El *middleware* se encarga de buscar el tipo necesario, cargarlo y ponerlo en ejecución (para los servicios de creación), de mantener las distintas entidades en un estado consistente (para los servicios de movilidad), y de eliminarlas. El sistema mantiene una serie de listas dinámicas con los nombres identificativos y únicos de las distintas entidades que lo forman, y que proporciona en el momento de su creación. Por esta razón, la clase SystemBase define los atributos componentsList, connectorsList, attachmentsList y bindingsList como listas dinámicas de cadenas, en las que se almacenan los nombres identificadores de cada uno de los elementos que forman parte del sistema.

5.3.4.1 Instanciación de aspectos

Los sistemas son elementos arquitectónicos compuestos, por lo que de igual forma que los conectores y los componentes, están compuestos por aspectos. Estos aspectos se agregan al componente durante su construcción. Así, como para el resto de elemetos arquitectónicos, en el constructor del sistema se instancian y se añaden a su lista de aspectos cada uno de los aspectos que lo componen.

```
AddAspect(new <nombreAspecto>([<parametros>]));
```

De esta forma, el sistema puede tener aspectos propios, además de los aspectos contenidos en los elementos arquitectónicos básicos que lo componen, y por tanto también publica los servicios implementados por sus aspectos mediante puertos.

Por otro lado, y del mismo modo que los componentes y conectores, el sistema también puede interconectar los servicios de los distintos aspectos que lo forman a través de la instrucción `AddWeaving`.

5.3.4.2 Instaciación arquitectónica

Además de aspectos, un sistema está compuesto por componentes y conectores. De forma análoga a la instanciación de aspectos, en el constructor del sistema se emplean los métodos AddComponent y AddConnector para agregar las instancias de los componentes y conectores al sistema. La diferencia principal, es que estos métodos no agregan directamente el objeto componente o conector al sistema, sino que almacenan en su lista dinámica el nombre que los identifica de forma unívoca en el modelo arquitectónico (ver Figura 52).

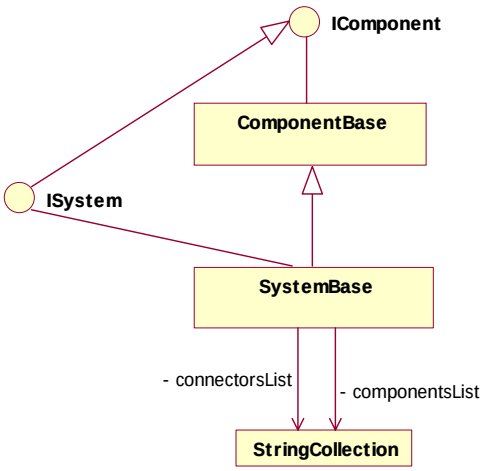


Figura 52 - Listas dinámicas de un sistema para componentes y conectores

Esto es necesario para que el *middleware* pueda gestionar cada elemento de forma individual y, además de su creación y destrucción, pueda moverlos a otras máquinas independientemente de la máquina donde se encuentre el sistema. En caso contrario, los componentes y conectores no podrían ser reubicados a distintas máquinas, pues el *middleware* no tendría acceso a ellos.

La asociación entre un sistema y un elemento arquitectónico se realiza durante la instanciación del sistema. En el cuerpo del constructor del sistema se asocian los componentes con el método definido en SystemBase AddComponent y AddConnector. Estos métodos requieren los mismos tipos de parámetros, pues su implementación es idéntica. Los parámetros requeridos son el nombre definido en la especificación, el tipo del componente/conector a

agregar al sistema, la URL donde se ubicará dicho componente/conector inicialmente, y un vector con los parámetros adicionales que pueda requerir.

```
AddComponent(<nombreComponente>,typeof(<componente>),<URL>,
                                                    [<parametros>]);



AddConnector(<nombreConector>,typeof(<Conector>),<URL>,
                                                    [<parametros>]);
```

En general, estos métodos comprueban en primer lugar si se violan algunas restricciones, como que el elemento a crear no existe previamente o que los tipos proporcionados como parámetros son los correctos. Después, invocan los servicios equivalentes del *middleware* para instanciarlos. Y finalmente, si dicha invocación ha tenido éxito, los nombres identificativos de los elementos creados son añadidos a las listas componentsList o connectorList del sistema.

5.3.4.3 Instaciación de puertos

De la misma forma que los otros elementos arquitectónicos, los sistemas también tienes asociados puertos. La implementación de los puertos de los sistemas se realiza exactemente igual que para los componentes y los conectores. Es necesario definir un nuevo tipo de puerto de entrada por cada interfaz que implementa los aspectos del sistema, y tenga que publicar. Y posteriormente instanciar cada puerto de entrada y de salida cuando se instancie el sistema.

```
InPorts.Add("<Puerto>", "<Interfaz>", GetAspect(typeof(
            <tipoAspectoDestino>,"<played_role>");
OutPorts.Add("<Puerto>", "<Interfaz>", "<played_role>");
```

De esta forma, los sistemas publican los servicios que implementan sus aspectos, además deben publicar todos los servicios de los elementos arquitectónicos que lo componen y que puedan ser invocados desde el exterior del sistema.

5.3.4.4 *Attachments*

En el modelo PRISMA, la comunicación entre componentes y conectores se realiza a través de los *attachments*. Cada attachment conecta dos elementos arquitectónicos (componente y conector) y se encarga de llevar las peticiones de servicio que solicitan entre ellos, actuando como si fuera un canal de comunicación. Los *attachments* añaden una capa de abstracción adicional, permitiendo que las comunicaciones distribuidas sean transparentes para componentes y conectores. La comunicación es bidireccional, ya que por una parte se encargan de hacer accesible al componente desde otras ubicaciones remotas (comportamiento servidor), y por otra, se encargan de establecer la comunicación hacia los componentes remotos (comportamiento cliente). Además, también permiten que los distintos elementos arquitectónicos interconectados funcionen de forma independiente, sin que se conozcan entre sí.

De forma general, un attachment comunica un puerto de salida de un componente con el puerto de entrada de un conector y viceversa, proporcionando una comunicación bidireccional. Además, puede conectar un puerto y un rol con distinta interfaz, siempre y cuando una de ellas sea un subconjunto de la otra. Los elementos arquitectónicos a conectar pueden encontrarse en la misma máquina o en distintas máquinas, en cuyo caso la comunicación es distribuida.

A nivel de implementación, la comunicación distribuida entre objetos se ha realizado mediante la tecnología *.Net Remoting*, ya que dispone de la mayor parte de los mecanismos para acceder remotamente a objetos, así como para serializarlos y moverlos de un dominio de aplicación a otro. Por otro lado, la implementación de los *attachments* se realiza a través de la herencia, de forma similar a los puertos de entrada. Por una parte, el comportamiento genérico del attachment se ha definido en un conjunto de clases de PRISMANET y por otra parte, el comportamiento específico definido en un modelo arquitectónico concreto, es generado en tiempo de diseño a la vez que se implementan las interfaces .NET, correspondientes a las interfaces PRISMA, heredando el comportamiento básico de las clases de PRISMANET.

5.3.4.4.1 Modelo de ejecución

La comunicación entre un componente y un conector se realiza creando una pareja de *attachments* (Client y Server) correspondientes a la interfaz de los puertos que se desea conectar, instanciándolos e iniciando su ejecución. La parte cliente de los *attachments* se ejecuta en un hilo de ejecución propio que periódicamente comprueba si se han depositado peticiones de servicio en los puertos de salida de los elementos arquitectónicos. En el caso de haber peticiones, el cliente las procesa para redireccionarlas hacia la parte servidora invocando el servicio en cuestión. El servidor redirige el servicio al puerto de entrada del elemento arquitectónico destino para que posteriormente, la petición se añada a la cola de peticiones de servicio del elemento arquitectónico (ver Figura 53).

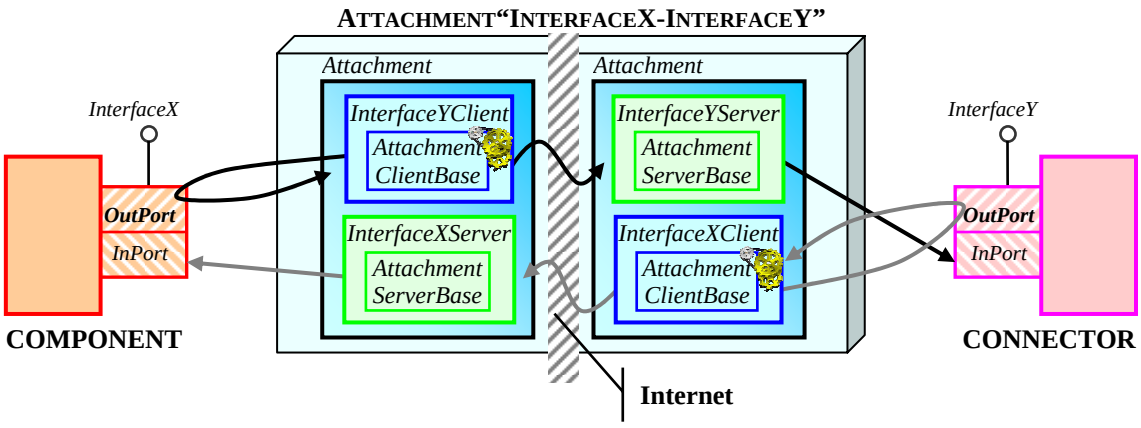


Figura 53 - Funcionamiento de los *Attachments*

5.3.4.4.2 Implementación

Un attachment tiene dos comportamientos claramente diferenciados: cliente y servidor. El comportamiento cliente consiste en escuchar periódicamente las peticiones depositadas por los aspectos de un elemento arquitectónico en el puerto de salida, actuándo como un *listenner*. El comportamiento servidor consiste en reenviar las peticiones recogidas, al puerto de entrada del elemento arquitectónico al cual va dirigida la petición. Como los elementos arquitectónicos que conecta el attachment pueden estar en máquinas distintas, en PRISMANET se ha subdividido al attachment en dos entidades, cada una de las cuales representa a un elemento arquitectónico de la comunicación y que se ubican en la misma máquina del elemento arquitectónico al que representan.

La implementación de cada una de las partes del attachment se efectúa extendiendo una clase base del *middleware* que les proporciona el comportamiento genérico y común de todos los *attachments*. Esta clase se denominan dentro de PRISMANET como AttachmentClientBase y attachmentClientServer. Para el caso de estudio del brazo de robot, hay un attachment que conecta el conector *CnctSUC* y la componente *Actuador* que envía las instrucciones de movimineto al robot. Este attachment soporta la comunicación entre el rol del connector y el puerto del componente que publican los servicios de la interfaz IMotionJoint de cada uno de los elementos arquitectónicos.

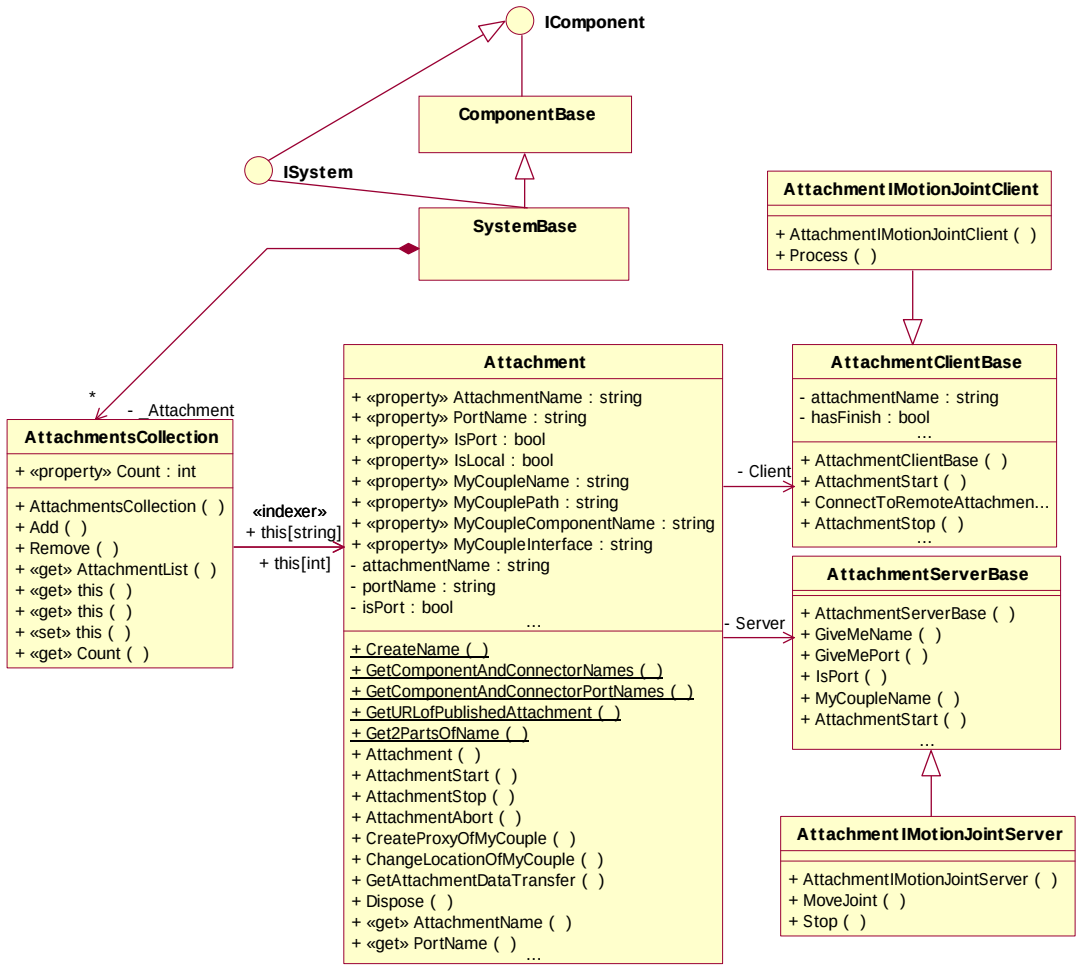


Figura 54 – Implementación del attachment asociado a la interfaz IMotionJoint

La implementación del comportamiento cliente de un attachment se realiza extendiendo por herencia la clase AttachmentClientBase. La definición de esta nueva clase se genera en el mismo proyecto de biblioteca de clases que las interfaces y los puertos debido a que un tipo de attachment está vinculado a un tipo de puerto y a una interfaz. Para instanciar la parte cliente de un attachment el constructor toma como argumentos el nombre del attachment, la referencia al componente al que se le va a asociar y el nombre del puerto en concreto en el que va escuchar las petiones de servicio

```
using System;

using PRISMA;
using PRISMA.Components;
using PRISMA.Attachments;

namespace <nombreModelo>
{

  public class Attachment<interfaz>Client : AttachmentClientBase
  {
    public Attachment<interfaz>Client(IComponent component,
                  string portName, string attachmentName)
                 : base(component, portName, attachmentName) {}
```

El comportamiento específico de cada attachment se define sobreescribiendo el método abstracto *Process* definido en la clase padre. Este método es el encargado de redigir las peticiones de servicio a la parte servidora del attachment. Dependiendo de la petición de servicio depositada en la cola de salida del componente, el cliente invocará el mismo servicio al attachment servidor que ya se encarga de redirigir la petición hacia el interior del elemento arquitectónico al que esta asociado.

```
public override void Process(ListenersQueue queue)
{
  try
  {
    switch(queue.NombreServicio)
    {
      case "<servicio>":
      {
           AsyncResult result = ((Attachment<interfaz>Server)remote)
                                          .<servicio>(queue.Args);
           queue.Result.SetDerivedParameters(result);
           break;
      }
      ... ...
      catch (Exception e)
      {
           throw e;
      }
  }
```

La implementación del comportamiento servidor de un attachment se realiza extendiendo por herencia la clase AttachmentServerBase. En general, esta clase recoge las peticiones de servicios que le envía la parte cliente del attachment y las reenvía al puerto de entrada del elemento arquitectónico destino que las debe ejecutar. Para ello, cuando el attachment servidor se instancia, obtiene una referencia al puerto de entrada del elemento arquitectónico destino.

```
using System;

using PRISMA;
using PRISMA.Components;
using PRISMA.Attachments;

namespace <nombreModelo>
{
  public class Attachment<interfaz>Server : AttachmentServerBase,
                                                    <interfaz>
  {

    private <interfaz> InPortOfComponent;

    public Attachment<interfaz>Server(Attachment attach)
                                                    : base (attach)
    {
      InPortOfComponent = (<interfaz>)
                                  Component.InPorts[attach.PortName];

      if (InPortOfComponent == null)
      {
           throw new Exception("InPort does not exists in " +
```

```
                    Component.componentName + " component.");
            }
        }
```

Para realizar la redirección de los puertos, la clase implementa los métodos de la interfaz que publica el puerto para redireccionarlos hacia él. Por lo que la redirección consiste en una invocación del mismo servicio en el puerto de entrada del elemento arquitectónico.

```
public AsyncResult <servicio>(<parametros>)
{
  return InPortOfComponent.<servicio>(<parametros>);
}
... ...
```

5.3.4.4.3 Patrón *Attachments*

Patrón 4 : *Attachments*

Contexto

Traducción de *attachments* PRISMA al lenguaje de implementación *CSharp*.

Modelo PRISMA

Attachments

<nom_Var$_i$>.<nom_puerto> ←→ <nom_Var$_k$>.<nom_rol>;

... ...

End_Attachments;

Patrón tecnológico

```
using System;
using PRISMA;
using PRISMA.Components;
using PRISMA.Attachments;
namespace <nombreModelo>
{
public class Attachment<interfaz>Client : AttachmentClientBase
{
  public Attachment<interfaz>Client(IComponent component,
            string portName, string attachmentName)
          : base(component, portName, attachmentName) {}
  public override void Process(ListenersQueue queue)
  {
    try
    {
      switch(queue.NombreServicio)
      {
           case "<servicio>":
           {
                AsyncResult result =
```

```
((Attachment<interfaz>Server)remote)
                                          .<servicio>(queue.Args);
                queue.Result.SetDerivedParameters(result);
                break;
           }
           ... ...
      }
      catch (Exception e)
      {
           throw e;
      }
    }
  }
}

public class Attachment<interfaz>Server : AttachmentServerBase,
                                                    <interfaz>
{
  private <interfaz> InPortOfComponent;
  public Attachment<interfaz>Server(Attachment attach)
                                              : base (attach)
  {
    InPortOfComponent =(<interfaz>)
                          Component.InPorts[attach.PortName];

    if (InPortOfComponent == null)
    {
      throw new Exception("InPort does not exists in " +
                Component.componentName + " component.");
    }
  }
  public AsyncResult <servicio>(<parametros>)
  {
    return InPortOfComponent.<servicio>(<parametros>);
  }
... ...
}
}
```

Caso de Estudio

Descripción

Attachment del caso de estudio del robot 4U4 que soporta la comunicación entre el rol del conector CnctSUC y el puerto de la componente Actuator que publican los servicios de la interfaz IMotionJoint.

Modelo Arquitectónico PRISMA del Caso de Estudio

```
Attachments
   VarSUCconnector.IMotionJointPort ←→ VarActuator. IMotionJointPort;
End_Attachments;
```

Patrón Implementación

```
using System;
using PRISMA;
using PRISMA.Components;
using PRISMA.Attachments;

namespace Robot {

public class AttachmentIMotionJointClient : AttachmentClientBase {

   public AttachmentIMotionJointClient(IComponent component,
                  string portName, string attachmentName)
                  : base(component, portName, attachmentName) {}

   public override void Process(ListenersQueue queue)
   {
      try
      {
        switch(queue.NombreServicio)
        {
             case "MoveJoint":
             {
                   int arg0 =(int)queue.Args[0];
                   int arg1 =(int)queue.Args[1];
                   AsyncResult result =((AttachmentIMotionJointServer)
                                    remote).MoveJoint(arg0,arg1);
                   queue.Result.SetDerivedParameters(result);
                   break;
             }
             case "Stop":
             {
```

```
                    AsyncResult result =((AttachmentIMotionJointServer)
                                                          remote).Stop();
                    queue.Result.SetDerivedParameters(result);
                    break;
                }
            }
        }
        catch (Exception e)
        {
            throw e;
        }
    }
}

public class AttachmentIMotionJointServer : AttachmentServerBase,
                                            IMotionJoint
{
    private IMotionJoint InPortOfComponent;

    public AttachmentIMotionJointServer(Attachment attach)
                                            : base (attach)
    {
        InPortOfComponent = (IMotionJoint)
                    Component.InPorts[attach.PortName];
        if (InPortOfComponent == null)
        {
            throw new Exception("IMotionJoint's InPort does not
        exists in " + Component.componentName + "
    component.");
        }
    }

    public AsyncResult MoveJoint(int newHalfSteps, int speed)
    {
        return InPortOfComponent.MoveJoint(newHalfSteps,speed);
    }
    public AsyncResult Stop()
    {
        return InPortOfComponent.Stop();
    }
}
```

```
}
```

5.3.4.4.4 Instanciación de *Attachments*

En un sistema, los *attachments* se almacenan en la colección AttachmentList. Para añadir instancias de *attachments* que comuniquen distintos elementos arquitectónicos pertenecientes al sistema, es necesario emplear el método AddAttachment definido en la superclase SystemBase. Este método, indica al *middleware* que realice la construccion de las partes cliente y servidora del attachment, y posteriormente las añade a la lista dinámica. El método AddAttachment requiere, para poder crear el canal de comunicación, el nombre de la componente, el nombre del puerto, el nombre del conector y el nombre del rol que van a ser participes de la comunicación. Además, también requiere la localización (URL) de los dos elementos arquitectónicos.

```
this.AddAttachment("<componente>","<puerto>","<url>",
                                "<conector>","<rol>","url");
```

En el caso de estudio del robot, el attachment que actua como canal de comunicación entre el componente CnctSUC y el Actuador se añade al sistema de la siguiente forma:

```
this.AddAttachment("Actuator", "IMotionJointPort","tcp://localhost"
                  ,"CnctSUC", "IMotionJointPort","tcp://localhost");
```

5.3.4.5 *Bindings*

En PRISMA, un binding es la entidad encargada de establecer la comunicación entre un elemento arquitectónico y el sistema que lo encapsula. Un binding asocia un puerto de un sistema con un puerto o rol de un elemento arquitectónico, actuando como un redirector de peticiones, de forma transparente para los componentes externos al sistema. Los *bindings* como los *attachments* son canales de comunicación, la principal diferencia que los distingue es que los binding permiten comunicar a elementos arquitectónicos de un nivel de granulidad con los elementos arquitectónico de más alto nivel de composición, es decir con sistemas de mayor granularidad.

5.3.4.5.1 Modelo de ejecución

El modelo de ejecución de los *bindings* es semejante al de los *attachments*. Un binding también está formado por dos entidades, cada una de las cuales reside en la misma máquina que la entidad a la cual representa. La diferencia entre un attachment y un binding radica en que la entidad del binding que representa al sistema se ha diseñado de forma específica para que sea común para todos los *bindings* de dicho sistema [Cos05] (ver Figura 55).

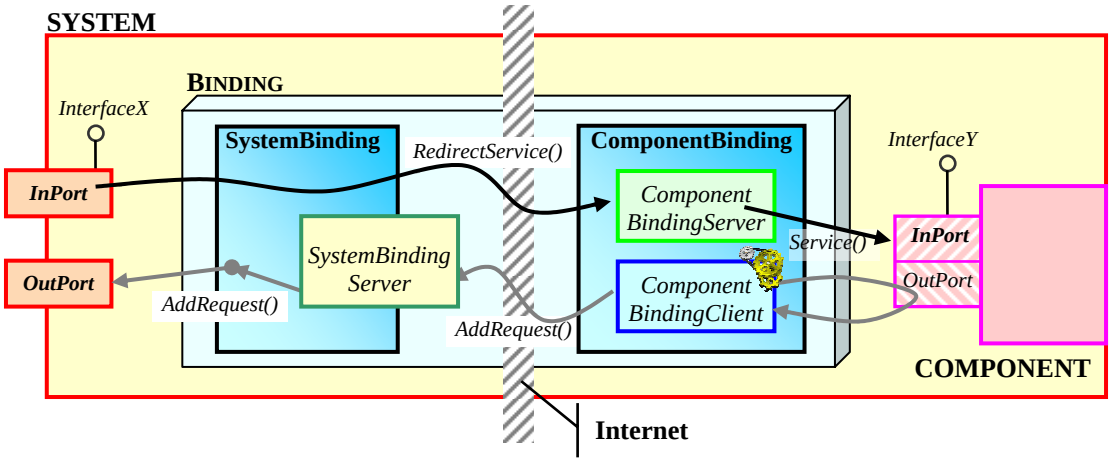


Figura 55 – Funcionamiento de los *bindings*

La parte que reside en el lado del componente o conector, a la que se ha denominado *ComponentBinding*, conserva el mismo diseño de los *attachments*: está formada por un objeto que implementa el comportamiento cliente (*ComponentBindingClient*) y otro objeto con el comportamiento servidor (*ComponentBindingServer*). El objeto cliente se encarga de recibir las peticiones de servicio del puerto de salida del componente y redirigirlas hacia el otro extremo de la comunicación, la parte residente en el lado del sistema. El objeto servidor se encarga de recibir las peticiones enviadas por la parte del binding residente en el sistema y redirigirlas al puerto de entrada del componente o conector.

La parte que reside en el lado del sistema, *SystemBinding*, se encarga de recoger todas las peticiones de servicio salientes de los componentes internos, enviadas a través de los *ComponentBindings,* y las reenvía al puerto de salida del sistema. Por otra parte, inicializa adecuadamente el puerto de entrada del sistema para que redirija las peticiones de servicio entrantes hacia los componentes internos.

5.3.4.5.2 Implementación

A diferencia de los *attachments*, toda la funcionalidad de los *bindings* está definida dentro de PRISMANET. Los *bindings* son creados dinámicamente sin necesidad de extender ninguna clase base, gracias a la reflexión y al uso de los delegados proporcionados por .NET. Esto implica que no es necesario implementar ninguna nueva clase adicional para definir el comportamiento de los *bindings*.

5.3.4.5.3 Instanciación de *Bindings*

En un sistema, los *bindings* se almacenan en la colección *BindingList*. Para añadir instancias de *bindings* que comuniquen los elementos arquitectónicos pertenecientes al sistema con el exterior, es necesario emplear el método *AddBinding* definido en la superclase *SystemBase*. Este método, indica al *middleware* que realice la construcción de las partes cliente y servidora asociadas al elemento arquitectónico para vincularlas con el puerto del sistema, y posteriormente las añade a la lista dinámica. El método

AddAttachment requiere, para poder crear el canal de comunicación, el nombre del puerto del sistema, el nombre del componente o conector y el nombre del puerto o rol que van a ser participes de la comunicación. Además, también requiere la localización (URL) del elemento arquitectónico interno al sistema.

```
this.AddBinding("<puerto>","<elementoArq.>","<puerto/rol>","url");
```

En el caso de estudio del robot, el binding que actúa como canal de comunicación entre el sistema *SUC* y el conector *CnctSUC* se añade al sistema de la siguiente forma:

```
this.AddBinding("SystemISUCPort", "CnctSUC", "ISUCPort",
                                                "tcp://localhost");
```

5.3.4.6 Patrón Implementación

Patrón 4 : Sistemas
Contexto
Traducción de sistemas PRISMA al lenguaje de implementación *CSharp*.
Modelo PRISMA

```
System_type <nombre_sistema>

  Ports
    <nom_puerto_i> : <interfaz_i>;
    <nom_puerto_i+1> : <interfaz_i+1>;
  End_Ports

  Variables
    <nom_Var_i> : <tipo_componente | tipo_conector>;
    ...
  End_Variables

  Attachments
    <nom_Var_i>.<nom_puerto> ←→ <nom_Var_k>.<nom_rol>;
    ...
  End_Attachments;

  Bindings
    <nom_Var_i>.<nom_puerto> ←→ <nombre_sistema>.<nom_puerto>;
    ...
  End_Bindings;

  Initialize
```

```
    New() {
      <nom_Var_i>= new <(nom_componente | nom_conector)>(<args>);
      ...
    }
  End_Initialize;

  Destruction
    Destroy() {
      <nom_Var_i>.destroy();
      ...
    }
  End_Destruction;
End_System_type <nombre_sistema>;
```

Patrón tecnológico

```
using System;
using PRISMA;
using PRISMA.Components;
using PRISMA.Middleware;


namespace <nombreModelo>
{
  public class <nombreSistema>: SystemBase
  {
    public <nombreSistema>( string name,[<parametros>,]
                MiddlewareSystem  middlewareSystem):
                base(name,middlewareSystem)
    {
    AddComponent(<nombreComponente>,typeof(<componente>),
                                              <URL>,[<parametros>]);
    ... ...
    AddConnector(<nombreConector>,typeof(<Conector>),
                                              <URL>,[<parametros>]);
    ... ...
    InPorts.Add("<Puerto>", "<Interfaz>", GetAspect(typeof(
                            <tipoAspectoDestino>,"<played_role>");
    OutPorts.Add("<Puerto>", "<Interfaz>", "<played_role>");
    ... ...
    AddAttachment("<componente>","<puerto>","<url>",
```

```
                                        "<conector>","<rol>","url");
    ... ...
    AddBinding("<puerto>","<elementoArq.>","<puerto/rol>","url");
    }
  }
}
```

Caso de Estudio

Descripción

Sistema SUC del caso de estudio del robot 4U4 compuesto por el conector CnctSUC y la componente Actuator

Modelo Arquitectónico PRISMA del Caso de Estudio

```
System SUC

   Ports
      SUC: ISUC;
   End_Ports;

   Variables
     VarActuator: Actuator;
     VarSUCConnector: SUCconnector;
   End_Variable;

   Attachments
    VarSUCconnector.ControlBaseAct ←→ VarActuator.ControlBaseAct;
   End_Attachments;

   Bindings
    VarSUCconnector.SUC ←→ SUC.SUC;
   End_Bindings;

   Initialize
     new () {
        VarActuator = new Actuator(Location: loc);
        VarConnector = new SUCconnector(HalfSteps: integer,
               Location: loc, inimum: integer, Maximum: integer);}
   End_Initialize;
   Destruction
     destroy (){
```

```
   VarActuator.destroy();
   VarConnector.destroy();}
   End_Destruction;
End_System SUC;
```

Patrón Implementación

```
using System;
using PRISMA;
using PRISMA.Components;
using PRISMA.Middleware;

namespace Robot
{
  [Serializable]
  public class SUC: SystemBase
  {
    public SUC(string name,string joint, int initialHalfsteps,
double initalMinimum, double initialMaximum, MiddlewareSystem
middlewareSystem) : base(name, middlewareSystem)
    {
      InPorts.Add("SystemISUCPort", "ISUC");
      OutPorts.Add("SystemISUCPort", "ISUC", "");

      AddComponent("Actuator", typeof(Robot.Actuator),
                                              "tcp://localhost");

      AddConnector("CnctSUC", typeof(Robot.CnctSUC),
                                  "tcp://localhost", joint, 0,
                                   initalMinimum, initialMaximum);

      AddAttachment("Actuator","IMotionJointPort","tcp://localhost",
                 "CnctSUC", "IMotionJointPort","tcp://localhost");

      AddBinding("SystemISUCPort", "CnctSUC", "ISUCPort",
                                              "tcp://localhost");
    }
  }
}
```

Capítulo 6

INTEGRACION DE COTS EN PRISMA

CONTENIDOS

INTEGRACIÓN DE COTS EN PRISMA

En los últimos años, los tiempos de entrega de productos *software* han disminuido notoriamente. El descenso de los tiempos de entrega ha implicado que las compañías de desarrollo se vean obligadas a emplear componentes de otros productores que sean fiables, eficientes y económicos para reducir el tiempo de su proceso de desarrollo. Concretamente, el uso de los productos comerciales disponibles, *Commercial Off-The-Shelf* (COTS), como elementos de sistemas de información grandes está siendo cada vez una práctica más común. El uso de COTS disminuye los costes del proceso de desarrollo ya que permiten la adaptación los nuevos requisitos que pueden surgir durante el desarrollo del sistema de forma más rápida. La utilización de COTS supone a los diseñadores menos esfuerzo a la hora de diseñar el sistema y de escribir menos código durante su implementación. Sin embargo, durante el proceso de desarrollo se ha de hacer frente a otros problemas. El diseñador debe asegurarse de que el producto que se va a emplear realiza la funcionalidad que se ajusta a las necesidades del sistema, y que su funcionalidad es correcta y que no perjudica ni al sistema ni al proceso de desarrollo. Pero sobretodo, el mayor inconveniente que implica el uso de los COTS en el desarrollo del *software* es que puede requerir un esfuerzo enorme de integración con los demás elementos del sistema. Esto puede suponer que el empleo de COTS para producir *software* sea contraproducente reduciendo de forma significativa el potencial para ahorrar tiempo y dinero que tienen dichos elementos.

En este capítulo se va a presentar la técnica para incorporar COTS en la implementación de un modelo arquitectónico PRISMA, haciendo de la integración de COTS en PRISMA, un proceso sencillo gracias a las posibilidades de interoperabilidad que ofrece el *.Net Framework*.

6.1 COTS

El acrónimo COTS significa *Commercial Off-The-Shelf* que se refiere a productos comerciales disponibles para su uso. Para dar una definición adecuada de qué es un COT se debe entender que se entiende por comercial y que se entiende por disponible. Un elemento comercial es un elemento que debe cumplir con unas características concretas [Obe97].

- Debe ser vendido, alquilado, o licenciado al público en general y usado con propósitos no gubernamentales.

- Debe estar disponible comercialmente únicamente si satisface todos los requisitos para los cuales ha sido creado.

Por otro lado, el término *off-the-shelf*, se refiere a que el elemento a emplear no ha sido desarrollado por los usuarios y que está disponible para su uso.

En definitiva un COTS en el marco de la ingeniería del *software* es un componente completamente funcional, comercial y disponible para su uso en la creación de nuevos sistemas de *software*. Las principales propiedades de un COTS es que existe a priori, está disponible para todo el público en general, y puede ser comprado.

Un COTS puede ser clasificado en función de algunos factores que lo caracterizan como su proveniencia, coste, modificabilidad, ensamblado, capacidad de integración y tipo de funcionalidad [Car00].

1. **Origen:** El origen de un COTS generalmente se divide en dos categorías (comercial o de libre uso). El origen de un COTS tiene una estrecha relación con el coste ya que el precio de un COTS comercial puede variar entre distintos fabricantes y los de libre uso son gratuitos pero pueden llevar restricciones de uso asociados.

2. **Modificabilidad:** Un COTS puede ser modificable por el usuario para ajustarlo a sus necesidades. En ocasiones, además de la documentación, pueden proporcionar algún tipo de mecanismo como APIs, SDKs,... para modificar la funcionalidad.

3. **Ensamblado:** Un COTS puede estar ensamblado mediante:

   - código fuente
   - librería de enlace binario
   - librería de enlace dinámico
   - fichero ejecutable

4. **Integración:** Un COTS puede tener asociado un módulo para su integración dentro de un sistema heterogéneo como por ejemplo un compilador o un mecanismo de interoperabilidad con otros lenguajes.

5. **Tipo de funcionalidad:** La funcionalidad de un COTS puede clasificarse en función de dos categorías, horizontal y vertical.

   - Horizontal: La funcionalidad no es específica a un dominio, sino que se puede reutilizar en diversas aplicaciones. Un ejemplo de este tipo de COTS son DBMSs, GUIs, protocolos de establecimiento de una red, browsers,...
   - Vertical: La funcionalidad es específica a un dominio, y se puede reutilizar solamente en el dominio. Un ejemplo de este tipo son COTS de usos financieros, contabilidad, planeamiento del recurso de ERP o de la empresa, fabricación, control de sistemas robóticos...

El uso de COTS para desarrollar sistemas *software* tiene ciertas ventajas fácilmente identificables. La principal ventaja es que la funcionalidad que se

necesita está implementada y disponible para su uso por lo que el de desarrollo se ve reducido, ya que generalmente, el coste del COTS es menor que el del desarrollo propio de la misma funcionalidad. Además, es muy probable que la COTS esté siendo usada por más grupos de desarrollo lo que garantiza su buen funcionamiento. Sin embargo, el uso de COTS también conlleva ciertos inconvenientes. Habitualmente, los COTS no se entregan con el código fuente por lo que las tareas de mantenimiento y de ajuste a las necesidades del usuario son imposibles de realizar.

6.2 Interoperabilidad .net

6.2.1 Introducción

.Net *Framework* representa un cambio importante en el modo en el que los desarrolladores crean y ejecutan aplicaciones al introducir el concepto de código manejado destinado a Common Language Runtime. El código manejado ofrece numerosas ventajas, incluidas la administración automática de la memoria, la programación de atributos y un sistema de tipos común. Sin embargo, las mismas características que confieren tanta eficacia al código manejado son las que lo diferencian básicamente del código no manejado, como es el caso de los COTS. Aunque .Net facilita relativamente el uso y la creación de objetos no manejados, estas tareas se pueden complicar. El proceso por el que se consigue que los objetos manejados y los no manejados funcionen conjuntamente se denomina interoperabilidad.

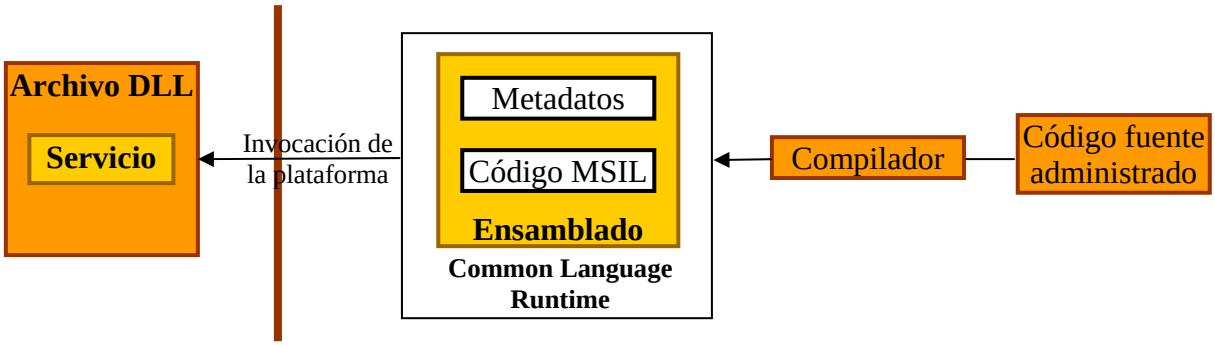


Figura 56 - Invocación de plataforma

La interoperabilidad permite comunicar distintos tipos de objetos no manejados como pueden ser componentes COM, *ActiveX*, librerías de enlace dinámico (*dll's*). En este proyecto se ha acotado el ámbito de la utilización de la interoperabilidad considerando únicamente las librerías de enlace dinámico como COTS, por lo que se van a presentar solamente los mecanismos de consumo de servicios que se encuentren implementados dentro de una *dll*. En la Figura 56, se puede observar como los objetos manejados pueden acceder a los servicios de una *dll* mediante la invocación de plataforma de .Net que realiza el cálculo de referencias entre los entornos manejados y no manejados

a partir de la información que se encuentra en los metadatos de los ensamblados que ya están compilados.

6.2.2 Interoperabilidad con servicios de *dll* no administradas

La invocación de la plataforma del .Net *Framework* es un servicio que permite que el código manejado llame a funciones no administradas implementadas en bibliotecas de vínculos dinámicos (*dll*) como los COTS. Además, este servicio localiza e invoca un servicio exportado y calcula las referencias a sus argumentos (enteros, cadenas, matrices, estructuras, clase,...) y valores de retorno dentro de los límites de la interoperabilidad, según sea necesario.

Para interoperar con servicios exportados en una *dll* se debe realizar las siguientes tareas:

1. Identificar servicios en archivos *DLL*

La identidad de un servicio de un archivo *dll* está formada por los siguientes elementos:

- Nombre del servicio
- Nombre del archivo *dll* donde se puede encontrar la implementación

Es posible cambiar el nombre de un servicio no administrada en el código, siempre y cuando se asigne el nuevo nombre al punto de entrada original del archivo *dll*. Un punto de entrada identifica la ubicación de un servicio en un archivo *dll*. En un proyecto manejado, el nombre original de un servicio de destino identifica dicha función dentro de los límites de la interoperabilidad. Además, puede asignarle otro nombre al punto de entrada, lo que supone en realidad un cambio de nombre de la función.

2. Crear una clase para contener los servicios (*wrapper*)

Empaquetar un servicio de un archivo *dll* en una clase administrada es un buen sistema para encapsular la funcionalidad del COTS. Aunque no es obligatorio hacerlo en todos los casos, es muy cómodo proporcionar un contenedor de clase porque facilita la invocación de los servicios del COTS de forma transparente,. Al empaquetar un archivo dll dentro de una clase, se define un método estático para cada servicio de la *dll* a la que se desea llamar. Esta forma, permite invocar a los servicios del mismo modo que se llama a los métodos de cualquier otro servicio estático. La invocación de la plataforma controla el servicio de la *dll* de forma automática.

A la hora de diseñar los contenedores para los servicios *dll*, hay que tener en cuenta las relaciones que existen entre las clases y los servicios de archivos *dll*. Por ejemplo, se puede:

- Declarar servicios de archivos *dll* dentro de una clase existente.

- Crear una clase individual para cada servicio de un archivo *dll*, de este modo las funciones se mantienen aisladas y resultan fáciles de encontrar.

- Crear una clase para un conjunto de servicios relacionados de un archivo *dll* para formar grupos lógicos y reducir la sobrecarga.

3. Implantar el servicio dentro de la clase

Para implantar un servicio definido dentro de una *dll* en el *wrapper*, se ha de emplear la clase `DllImport` definido en el espacio de nombres `System.Runtime.InteropServices` del .Net *Framework*. Esta clase actúa como un atributo .Net y permite declarar funciones externas pasándole como atributo el nombre del archivo *dll* que contiene la definición de la función y el punto de entrada o nombre de la función que se quiere manejar.

```
using System.Runtime.InteropServices;

... ...
[DllImport("dll", EntryPoint="función")]
private static extern <signatura>;
```

4. Invocar un servicio de un archivo *dll*

La invocación de un servicio no manejado de archivos *dll* es prácticamente idéntica a la llamada a cualquier servicio de código manejado. La forma de invocar un servicio desde otra clase es haciendo referencia a la clase wrapper donde se ha importando el servicio e invocarlo pasándole los parámetros que necesita. La única diferencia que lo distingue de una invocación en código manejado es cuando las referencias de los tipos de los parámetros del servicio no se pueden calcular automáticamente debido a su complejidad. En estos casos, el cálculo de referencias entre tipos de datos manejados y no manejado se debe especificar antes de invocar la función [MSDN].

6.3 COTS en PRISMA

El modelo PRISMA permite modelar sistemas incorporando COTS. A nivel de modelo, PRISMA permite importar COTS como ciudadanos de primer orden. PRISMA trata los COTS como componentes externos que pueden ser importados en un modelo arquitectónico. Esto es posible gracias a que un componente en PRISMA tiene dos posibles vistas diferentes: una visión interna, en la que un componente se ve como un prisma con tantas caras como aspectos considere, y una visión externa que es una caja negra fuertemente encapsulada dentro de un modelo de arquitectura y que interactúa con el resto del sistema a través de sus puertos. Por lo que, desde la visión externa, un COTS en PRISMA se considera un componente del mismo grado de importancia que los del resto del modelo arquitectónico. Se ha de tener en cuenta que en PRISMA, la importación de COTS se expresa gracias al carácter

opcional de los aspectos. Esta opcionalidad permite ver a los tipos como cajas negras y con ello la posibilidad de incorporar componentes COTS.

La incorporación de los componentes COTS permite a PRISMA definir sistemas integrados (*Integrated Systems*) [Car97] que incluyen varios componentes COTS que se comunican entre sí y en donde la integración es la clave para construir el sistema.

6.3.1 Técnicas de integración

La integración de componentes COTS en un sistema funcional se presenta generalmente como un hándicap para la mayoría de desarrolladores. Los problemas que se presentan con la integración de COTS son muy similares a los considerados en general con la reutilización del *software* [Sha95]. Debido a que los diversos componentes están escritos por diferentes vendedores con distintas necesidades es necesario adaptarlos para trabajar con ellos. En general, la integración de COTS es un problema complejo ya que se deben considerar múltiples factores para adaptar los COTS con un sistema como por ejemplo, la representación, comunicación, empaquetado, sincronización, semántica, control, y otras características.

Actualmente, se emplean tres técnicas diferentes para adaptar e integrar COTS en los sistemas. La primera de ellas consiste en integrar el COTS envolviéndolo en un elemento *software* (*wrapper*) propio del sistema. La segunda se denomina *glueware* y pretende el uso de un intermediador entre los distintos componentes. Y por último, la técnica que emplea adaptadores o puentes (*proxies*) para esconder las incompatibilidades entre las distintas interfaces de los componentes. Estos tres mecanismo son técnicas de caja negra que integran los componentes sin acceder al código fuente de los COTS.

Los *wrappers* son envolturas *software* que incluyen el COTS con el fin de de poder utilizarlos y adaptarlos a una determinada arquitectura de programación. Además permiten definir una interfaz estándar para acceder a la funcionalidad de la COTS y extender o restringir dicha funcionalidad según sea conveniente [Vid97].

La técnica de *glueware* consiste en definir una capa intermedia (*middleware*) ente los componentes que se encarga de resolver las incompatibilidades técnicas entre COTS y así que se puedan conectar de forma transparente. Además, este *middleware* puede ser utilizado para gestionar el flujo del control y para gestionar el control de errores.

Los adaptadores o puentes posibilitan la intercomunicación de distintos componentes unificando las interfaces de los COTS para que puedan invocar los distintos servicios definidos en cada uno de ellos ocultando su complejidad.

6.3.2 Integración en PRISMA

Para realizar la integración de COTS en PRISMA, se emplea la técnica que emplea un elemento *software* (*wrapper*) para envolver el COTS. Para elegir el mejor contenedor que se adaptará de forma uniforme con los modelos arquitetónicos, se estudiaron dos posibilidades. La primera empleaba como *wrapper* un componente y la segunda empleaba un aspecto. Finalmente se optó por utilizar el aspecto como contenedor de los servicios de la *dll* por se la más ventajosa. A continuación se presentan ambas aproximaciones.

6.3.2.1 Componente como *wrapper*

Una primera aproximación para solventar el problema fue considerar que un componente actuará como *wrapper* del COTS. De esta forma, el componente que envolvía la COTS publicaba su interfaz a través de un puerto para poder invocar los servicios definidos.

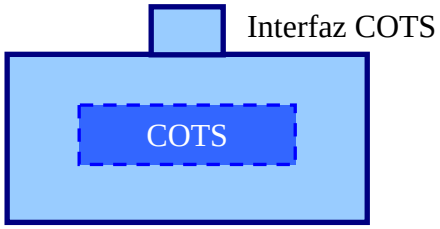


Figura 57 - Componente como wrapper

Con esta aproximación se integra una COTS dentro del modelo PRISMA y se permite interactuar con ella a través del puerto de la componente que publica a través de la interfaz los servicios implementados en el COTS. Además, incluso se puede restringir la funcionalidad del COTS no publicando todos sus servicios en el puerto. El problema que reside con esta aproximación es que no es muy uniforme con el modelo ya que generalmente los servicios de una componente PRISMA están definidos en aspectos, de forma que puedan interactuar entre ellos a través del weaving y se puedan reutilizar fácilmente. Además, tiene una restricción añadida y es que no se puede extender el comportamiento del COTS.

6.3.2.2 Aspecto como *wrapper*

La aproximación definitiva pretende resolver los inconvenientes presentados. Esta aproximación emplea 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.

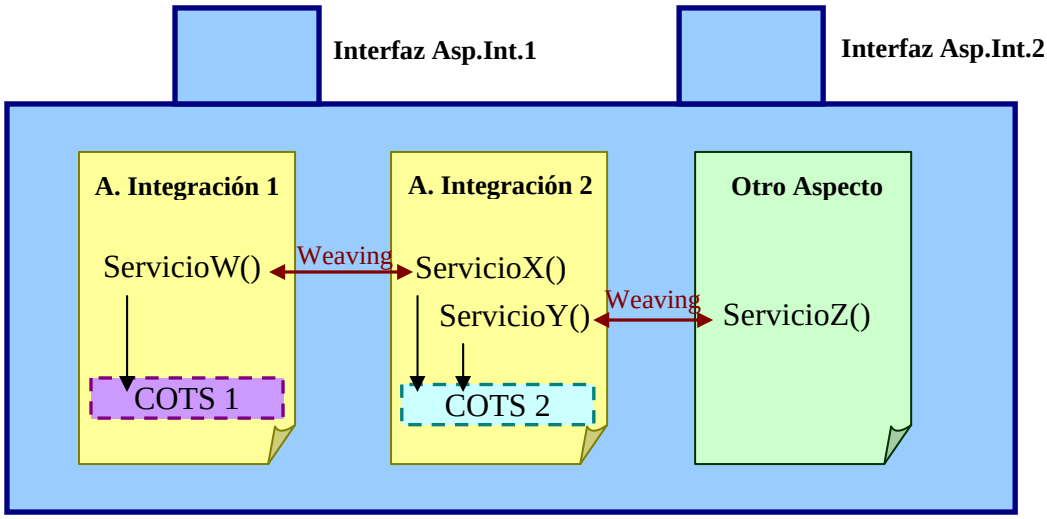


Figura 58 – Aspecto como wrapper

En implementación, se hace uso de los mecanismos de interoperabilidad que proporciona la plataforma *.net* para poder hacer uso de *dll*'s no administradas. Para integrar una COTS dentro de un aspecto de integración se debe de crear una clase que herede `IntegrationAspect`. En primer lugar, esta clase se encarga de importar todos los servicios externos de la *dll* que se quieren importar empleando para ello el atributo `DllImport`. En segundo lugar y para mantener la uniformidad con el modelo de ejecución que establece PRISMA para los aspectos, el aspecto define por cada servicio externo dos métodos, uno público y uno privado, con la misma signatura que la del servicio externo. De la misma forma que para la implementación del resto de aspectos PRISMA, el método público es el encargado de encolar las peticiones se servicio que se efectúen desde el exterior, y al método privado se le encomienda la invocación del servicio externo de la COTS y la devolución de los resultados de la ejecución del servicio. Además, es necesario publicar todos los métodos públicos, para que puedan ser invocados, mediante una interfaz y la definición de los delegados de sus métodos.

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 9 - 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 aspecto 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")]
```

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

  enum protocolStates
  {
       FCOTS,
       SEND
  }
  protocolStates estado;

  public FCOTS() : base()
  {
    estado = protocolStates.FCOTS;

    PlayedRoleClass playedRoleICOTS = new PlayedRoleClass("COTS");
    playedRoleICOTS.AddMethod("Send", 10);
    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;
```

```
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 7

CONCLUSIONES Y TRABAJOS FUTUROS

CONTENIDOS

CONCLUSIONES Y TRABAJOS FUTUROS

7.1 Conclusiones

En este proyecto se ha presentado, la aplicación de la metodología PRISMA para analizar, especificar e implementar un sistema arquitectónico complejo, distribuido y reutilizable como es el caso de los sistemas robóticos tele-operados empleados a nivel industrial. Como prototipo se ha escogido un robot *TeachMover* con el fin de suscitar el aprendizaje y comprensión de este tipo de sistemas.

Para cumplir con el objetivo de implementación del sistema robótico tele-operado *TeachMover* en *C#* en base al modelo PRISMA, en primer lugar, este trabajo presenta el análisis y la especificación del robot mediante el lenguaje de descripción de arquitecturas. De este modo se muestra como PRISMA reúne todos los requisitos necesarios para la construcción de arquitecturas complejas, distribuidas y reutilizables de forma sencilla a través de su lenguaje de descripción de arquitecturas que combina la reutilización proporcionada por la orientación a aspectos y la configuración de los elementos arquitectónicos de los modelos basados en componentes.

La implementación de la especificación del caso de estudio sobre la tecnología .Net muestra como las equivalencias entre los conceptos del modelo PRISMA y la implementación de dicho conceptos ha sido posible mediante el soporte de un *middleware* sobre la plataforma .Net, permitiendo validar la implementación de arquitecturas *software* concurrentes y altamente reutilizables. Además, la implementación ha servido de retroalimentación para refinar el *middleware*, uno de los aportes fundamentales ha sido el soporte a la integración de componentes comerciales (COTS) para aumentar las capacidades de reutilización en la implementación del modelo PRISMA.

La propuesta de implementación realizada se ha materializado en un prototipo desarrollado con la tecnología .NET, del que se han identificado los patrones de generación de código que deberán ser utilizadas por el futuro compilador para generar código *C#* a partir de especificaciones PRISMA. Este es el primer paso para que PRISMA se sume a la innovadora tendencia de Desarrollo de *Software* Dirigido por Modelos (DSDM), y más concretamente la propuesta MDA (Model Driven Architecture) de OMG que constituyen una aproximación para el desarrollo de sistemas *software* basada en la separación entre la especificación de la funcionalidad esencial del sistema y la implementación de dicha funcionalidad, usando plataformas de implementación específicas como por ejemplo .Net.

7.2 Trabajos futuros

Una vez cumplidos con los objetivos de este proyecto, la siguiente fase consiste en desarrollar un compilador PRISMA que permitirá generar automáticamente código ejecutable. El compilador transformará las especificaciones PRISMA a .Net en base a los patrones de generación de código que han sido identificados en este proyecto para cada uno de los conceptos PRISMA que expresa el metamodelo.

A corto plazo, se pretende definir PRISMA como un lenguaje específico de dominio sobre la plataforma .Net. Para ello se estudiará la aplicación de las posibles metodologías innovadoras que confluyen actualmente como por ejemplo MDA o *Software Factories* y que tienen el objetivo de establecer unos mecanismos de desarrollo de *software* dirigido por modelos.

Además, a medio plazo se pretende abordar la especificación del sistema de tele-operación EFTCoR, una vez esté terminado, en base al modelo arquitectónico del robot *Teachmover* para validar las ventajas del modelo arquitectónico de referencia ACROSET. También, y en relación con ésta, en un futuro, es necesario abordar la aplicación en la etapa de requisitos la metodología ATRIUM [Nav04] y cómo integrar la aplicación del modelo arquitectónico ACROSET dentro de ésta.

BIBLIOGRAFÍA

[Ale77] Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press. 1977.

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press. 1979.

[Ali03] Ali N.H., Silva J., Jaen J., Ramos I., Carsi J.A., Perez J. *Mobility and Replicability Patterns in Aspect-oriented Component-Based Software Architectures*. In Proc. of 15th IASTED, Parallel and Distributed Systems, Acta Press, ISBN: 0-88986-392-X, ISSN: 1027-2658, pp. 820-826. Marina del Rey, C.A., USA, Noviembre 2003.

[AOSD] "*Aspect Oriented Software Development*", http://aosd.net

[App97] Brad Appleton. Patterns and Software: *Essential Concepts and Terminology*. http://www.enteract.com/~bradapp/docs/patterns-intro.html. 1997.

[Bac00] Bachman F., Bass et al., *The Architecture Based Design Method,* Technical Report CMU/SEI-200-TR-001, Carnegie Mellon University, USA, January 2000.

[Bal85] Balzer R., *A 15 Year Perspective on Automatic Programming*. IEEE Transactions on Software Engineering, vol.11, num.11, págs. 1257-1268, Noviembre 1985.

[Car97] Carney, D. Assembling Large Systems from COTS Components: Opportunities, Cautions, and Complexities. *SEI Monographs on Use of Commercial Software in Government Systems*, Software Engineering Institute, Pittsburgh, USA, June 1997.

[Ca00] Carney, D., Long, F., What Do You Mean by COTS?, IEEE Software, March/April 2000, pp. 83-86.

[CBSD] D. F. D'Souza and A. Cameron. "*Objects, Components and Frameworks with UML: The Catalysis Approach",* Addison-Wesley, 1999.

[Cos05] Costa C., Estudio e Implementación de un Modelo de Arquitecturas Orientado a Aspectos y basado en Componentes sobre Tecnología .NET, Universidad Politecnica de Valencia, 2005.

[Cop99] Jim Coplein. *Software Patterns*. 1999.

[EFTCoR] EFTCoR Project: Friendly and Cost-Effective Technology for Coating Removal. V Programa Marco, Subprograma Growth, G3RD-CT-2002-00794. 2002

[Gam94] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Dessign Patterns:Elements of Reusable Object-Oriented Software*. Addyson-Wesley.1994.

[Gar95] D. Garland y D.E. Perry, *Introduction to the Special Issue on Software Architecture*, IEEE Transactions on Software Engineering, 21(4), abril 1995.

[Har84] Harel D.: Dynamic Logic; in Handbook of Philosphical Logic II, editors D.M. Gabbay, F. Guenthner; pp.497-694, Reidel (1984).

[Let98] Letelier P., Sánchez P., Ramos I., Pastor O. OASIS 3.0: "*Un enfoque formal para el modelado conceptual orientado a objeto*". Universidad Politécnica de Valencia, SPUPV -98.4011, ISBN 84-7721-663-0, 1998.

[Mel04]: Stephen J. Mellor, Kendall Scott, Axel Uhl, Dirk Weise, *MDA Distilled: Principles of Model-Driven Architecture,* ISBN: 0201788918, *Addison-Wesley*

[Mil91] Robin Milner, π- Cálculo Poliadico: Un tutorial,1991.

[MSDN]
http://msdn.microsoft.com/library/SPA/cpguide/html/cpconmarshalingdatawithplatforminvoke.asp

[Nav04] Elena Navarro, Patricio Letelier, Isidro Ramos, Goals and Quality Characteristics: Separating Concerns, Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, collocated to OOPSLA 2004, Monday, October 25, 2004, Vancouver, Canada.

[Nor98] P. Noriega, C. Sierra, Subastas y Sistemas Multiagente, Revista Iberoamericana de Inteligencia Artificial, Number 6, pp. 68-84, 1998.

[Obe97] Oberndorf, T., COTS and Open Systems - An Overview, 1997, http://www.sei.cmu.edu/str/descriptions/cots.html#ndi

[Ort05] Ortiz Francisco José, *"Arquitectura de Referencia para Unidades de Control de Robots de Servicio Teleoperados"*. Universidad Politécnica de Cartagena.

[Pas04] Juan A. Pastor, Bárbara Álvarez, Pedro Sánchez, Francisco Ortiz, *An Architectural Framework for Modeling Teleoperated Service Robots,* IEEE Transactions on Computers.

[Per03] Pérez, J., Ramos, I., *Oasis como Soporte Formal para la Definición de Modelos Hipermedia Dinámicos, Distribuidos y Evolutivos*. Informe Técnico DSIC-II/22/03, Universidad Politécnica de Valencia, octubre 2003.

[Pos04] Poseidon, 2004, http://www.gentleware.com

[Rie96]Dirk Riehle. "*Describing and Composing Patterns Using Role Diagrams*." In *Proceedings of the 1996 Ubilab Conference, Zürich.* Edited by Kai-Uwe Mätzel and Hans-Peter Frei. Konstanz, Germany: Universitätsverlag Konstanz, 1996. Page 137-152.

[Sha95] Shaw, M., Architectural Issues in Software Reuse: It's Not Just the Functionality, It's Packaging, Proceedings of the Symposium on Software Reusability, 1995, Seattle, WA, USA, pp. 3-6.

[TeachMover] TeachMover Robot, http://www.questechzone.com/microbot/teachmover.htm

[UML04] UML, 2004, http://www.uml.org/

[Vid97] Vidger, M.R., Dean, J., An Architectural Approach to Building Systems from COTS Software Components, The 22nd Software Engineering Workshop, NASA/Goddard Space Flight Center Software Engineering Laboratory (SEL), Greenbelt, MD, December 1997, pp. 99-131