ARQUITECTURA PRISMA PARA EL CASO DE ESTUDIO: ROBOT 4U4

Jennifer Pérez Benedí, Nour Ali Israid, Jose Ángel Carsí Cubel, Isidro Ramos Salavert
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
C/Camino de Vera s/n
E-46071 Valencia- España
jeperez@dsic.upv.es

Abstract

El desarrollo de software basado en componentes y aspectos esta despertando mucho interés, no sólo en el ámbito académico, sino también en el industrial. En este trabajo se aplica el modelo PRISMA a un caso de estudio de ámbito industrial en el marco del proyecto europeo EFTCoR (Environmental Friendly and Cost-effective Technology for Coating Removal). El caso de estudio consiste en el diseño y construcción de un sistema robótico teleoperado dedicado a la limpieza de cascos de buques. El diseño de la arquitectura de estos sistemas se debe abordar desde un alto nivel de abstracción de forma que sea posible reutilizar los diseños para distintos tipos de arquitecturas y que sean fácilmente modificables. Concretamente, este trabajo presenta el diseño arquitectónico y aspectual del robot 4U4. Este robot ha sido cedido por parte de miembros de proyecto EFTCoR para utilizarse en este proyecto en las pruebas de especificación y desarrollo en C# de las componentes software del robot. Este robot, a pesar de ser más pequeño que el EFTCoR, comparte con aquél el marco arquitectónico de referencia definido para el EFTCoR (llamado ACROSET) e instanciado a las peculiaridades del 4U4. El desarrollo del caso de estudio muestra como PRISMA reúne todos los requisitos necesarios para la construcción de arquitecturas complejas, distribuidas y reutilizables, como los sistemas de tele-operación, de forma sencilla a través de su lenguaje de descripción de arquitecturas orientado a aspectos.

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 4U4 [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 4U4 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 4U4, 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.

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 4U4. 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 1 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 (ver Figura 2 para el caso de la herramienta). 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.

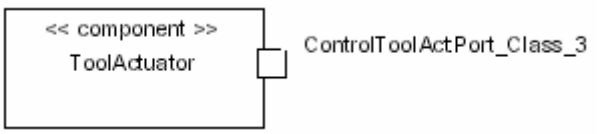


Figura 1: Actuador

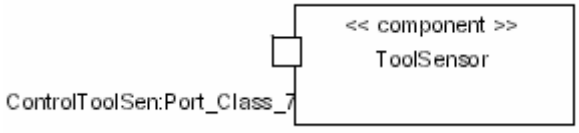


Figura 2: Sensor

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 3 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.

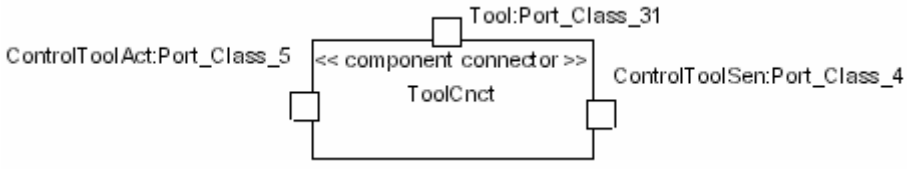


Figura 3: Conector Sensor-Actuador

3. IDENTIFICACIÓN DE SIMPLE UNIT CONTROLLERS (SUCs)

Los *simple unit controllers* (SUCs) van a formar la segunda capa de la arquitectura (ver Figura 10). 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 4U4, 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 4). Estos son los siguientes: *Base, Shoulder , Elbow, Wrist,* y *Tool*.

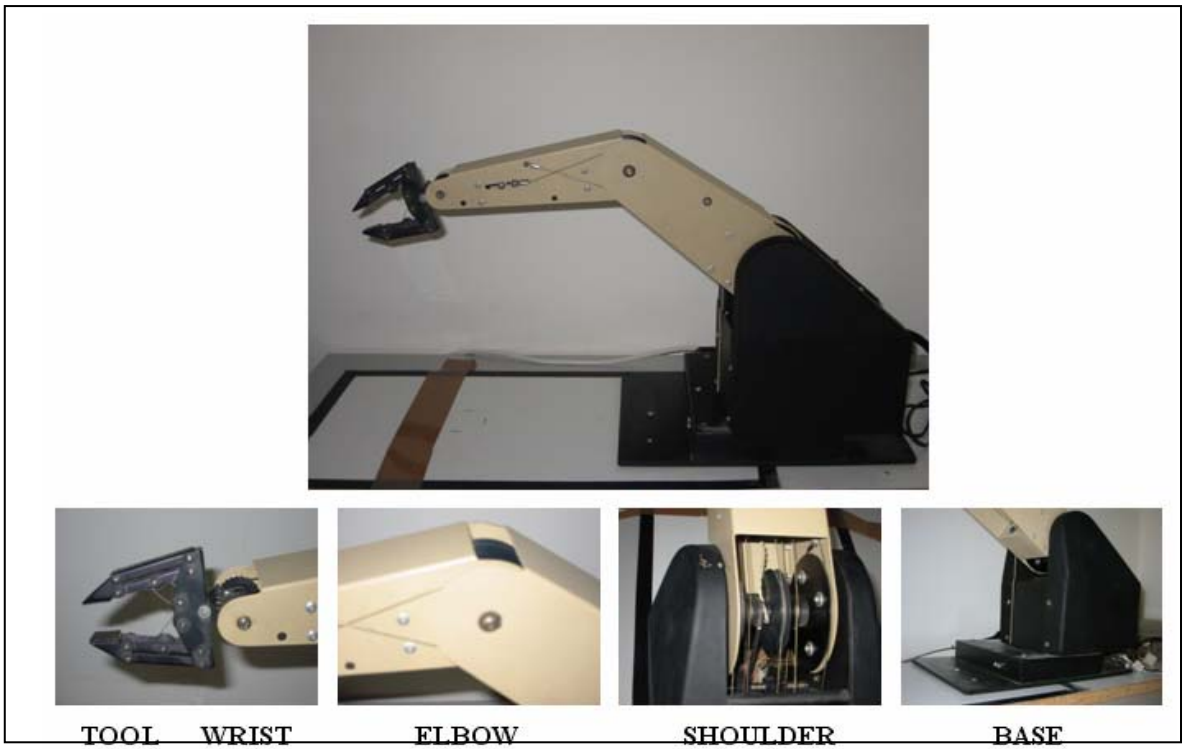


Figura 4: Robot 4U4

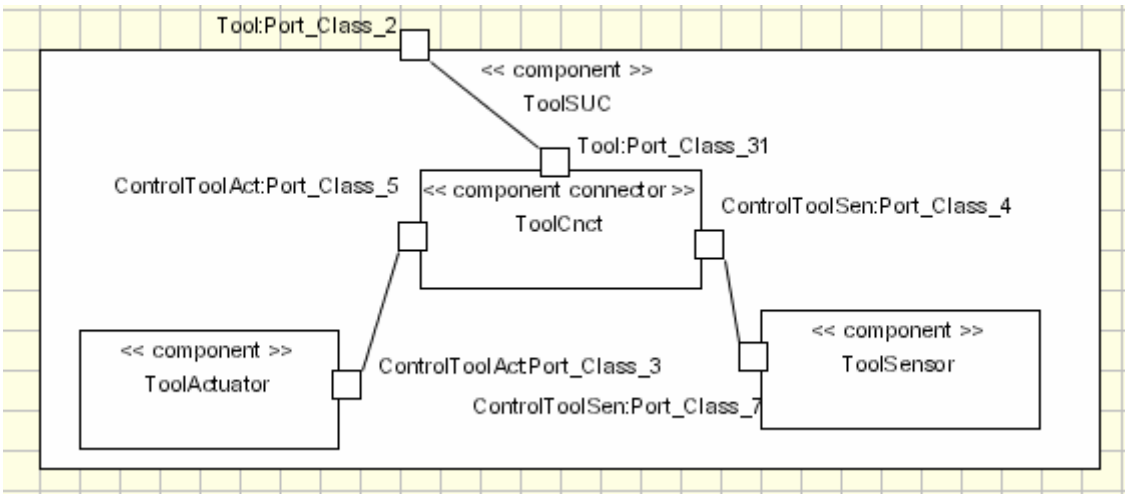


Figura 5: SUC de la herramienta

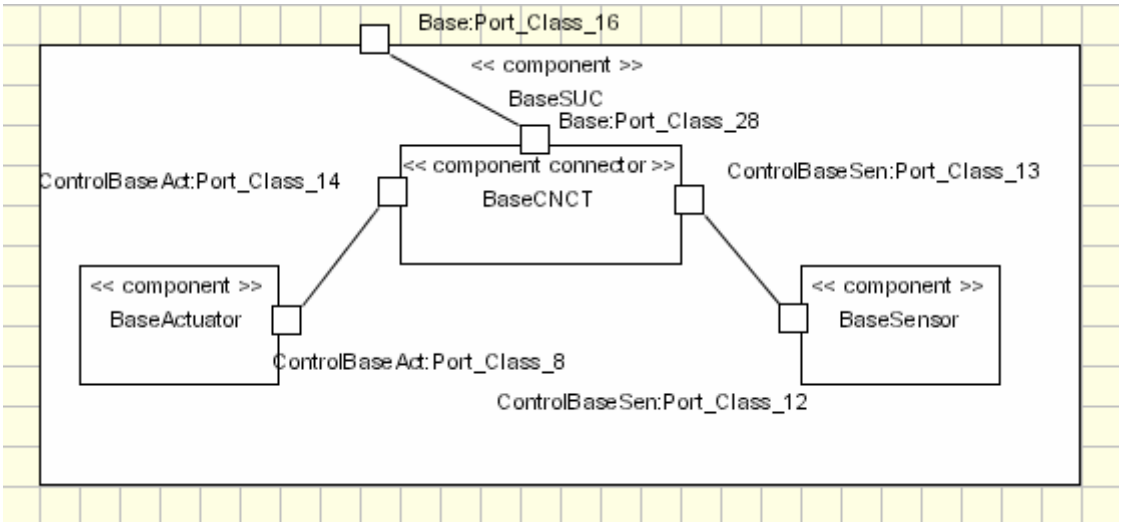


Figura 6: SUC de la base

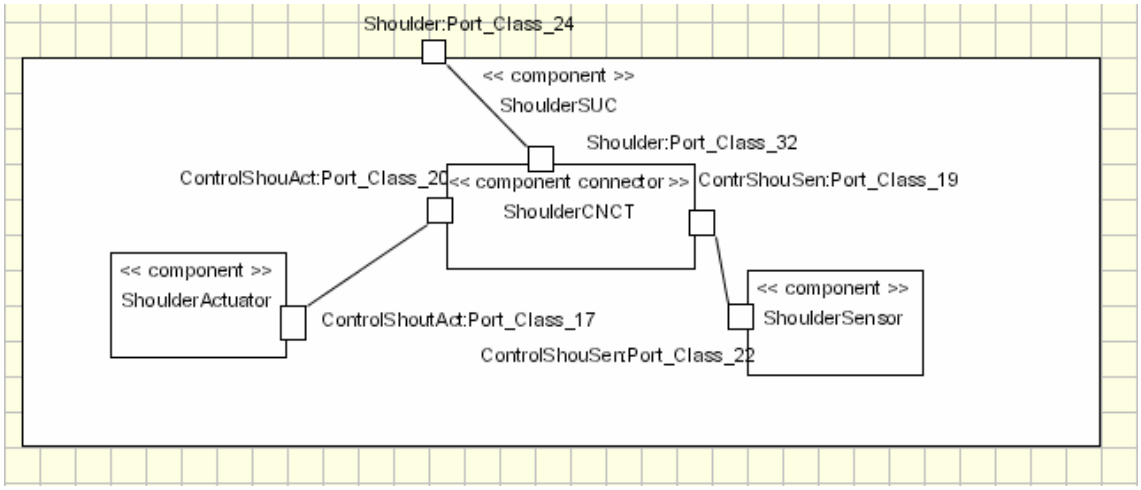


Figura 7: SUC del shoulder

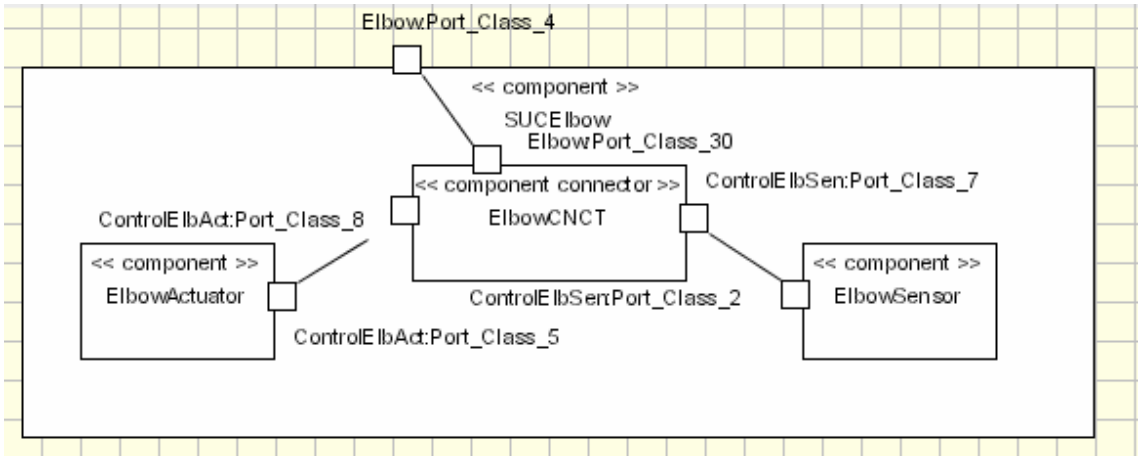


Figura 8: SUC del elbow

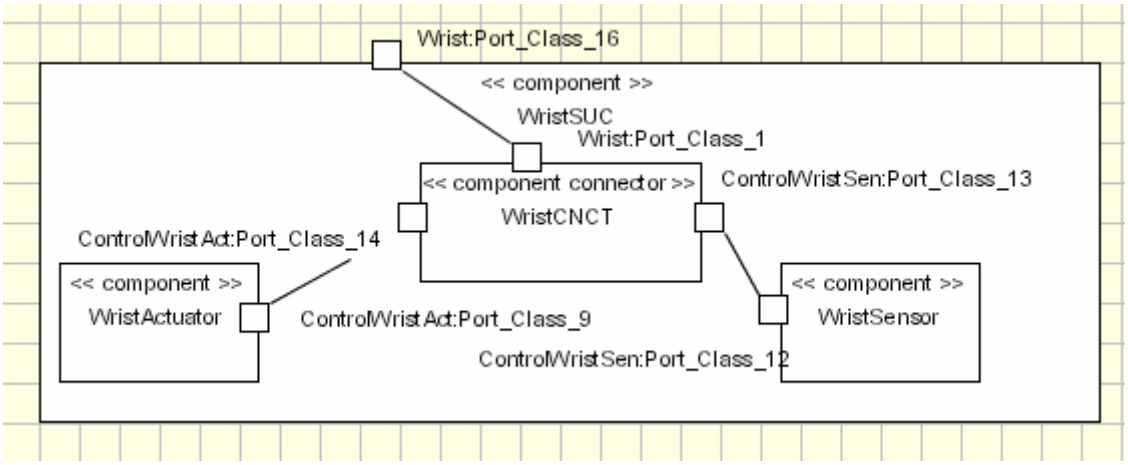


Figura 9: SUC del wrist

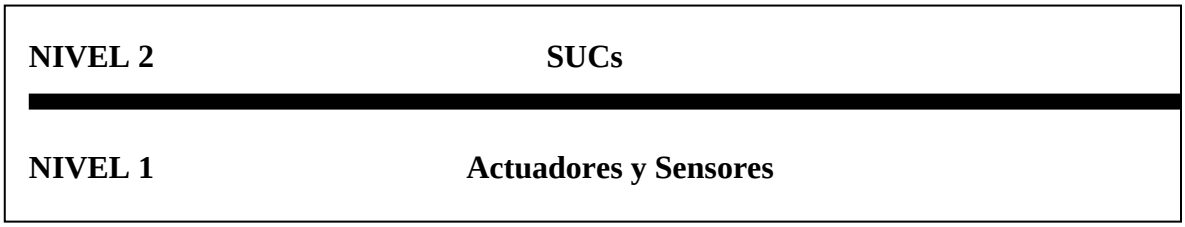


Figura 10: Dos niveles de abstracción de la arquitectura PRISMA del robot 4U4

4. IDENTIFICACIÓN DE Mechanism Unit Controller (MUCs)

Los MUCs van a formar la tercera capa de la arquitectura (ver Figura 11). 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).

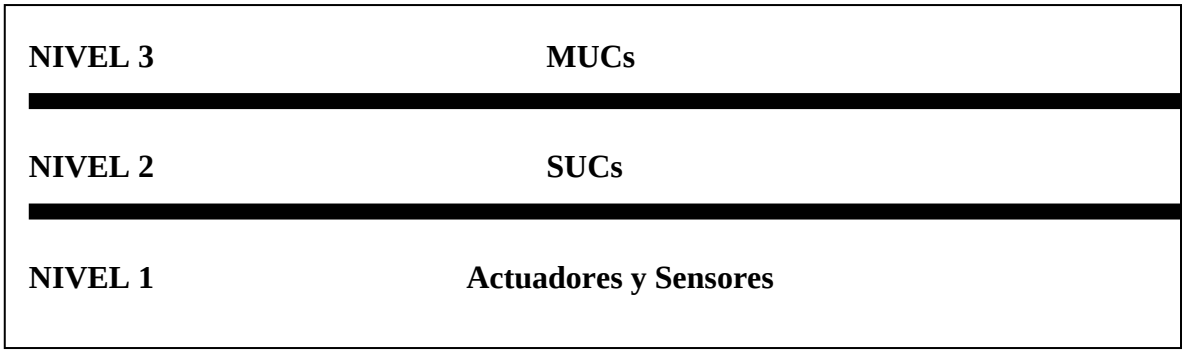


Figura 11: Tres niveles de abstracción de la arquitectura PRISMA del robot 4U4

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 4U4, 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 4U4, 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 4U4. Dicho MUC está compuesto por los SUCs *Base, Shoulder, Elbow* y *Wrist* y un connector que los va a coordinar con el objetivo común de mover el brazo (ver Figura 12).

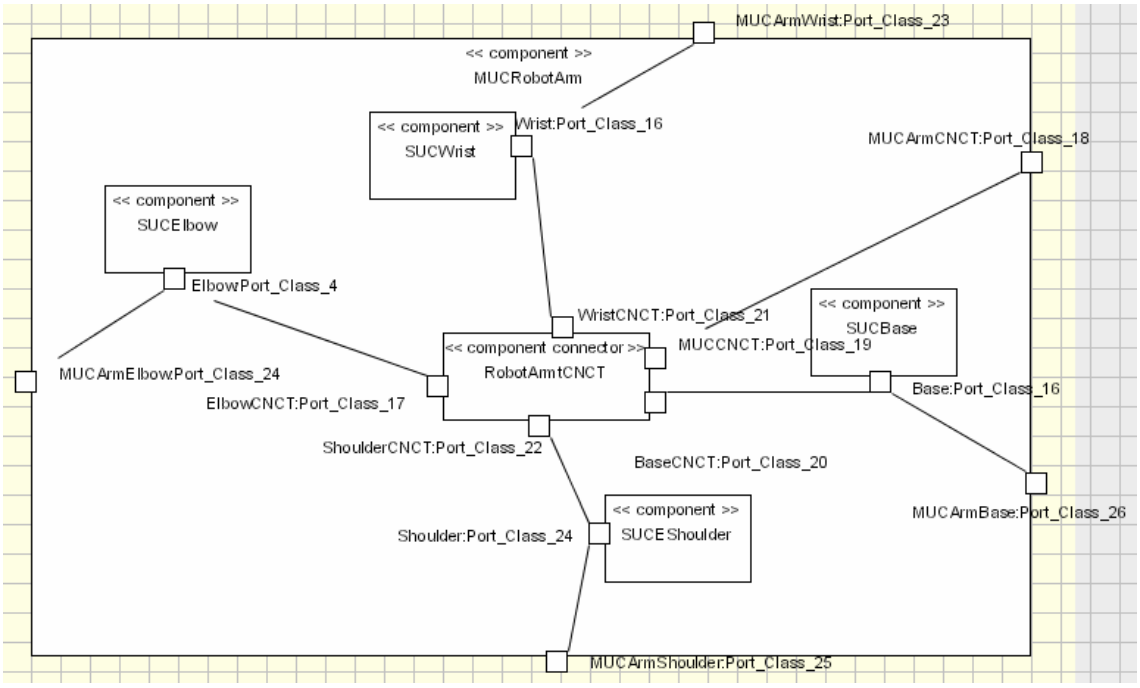


Figura 12: MUC del brazo robot

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.

5. IDENTIFICACIÓN DE Robot Unit Controller (RUCs)

Los RUCs van a formar la cuarta capa de la arquitectura (ver figura 13), 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 4U4, 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 4U4 (ver Figura 14). 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 13: Cuatro niveles de abstracción de la arquitectura PRISMA del robot 4U4

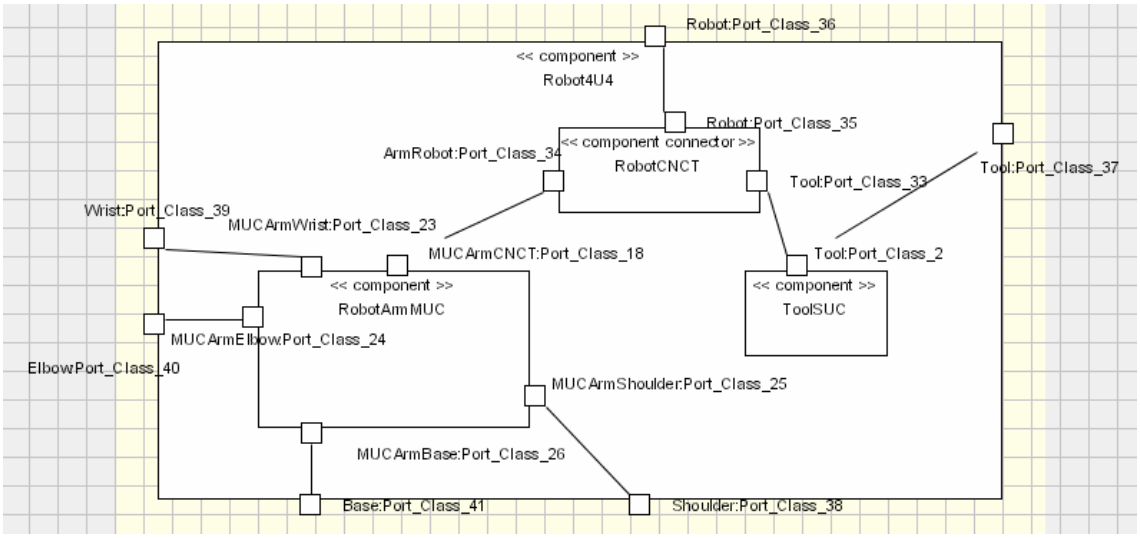


Figura 14: RUC robot 4U4

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.

6. MODELO ARQUITECTÓNICO FINAL

Finalmente, nos encontramos con la última capa del modelo arquitectónico del sistema de información (ver figura 15). 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 4U4 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 4U4 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 figura16).

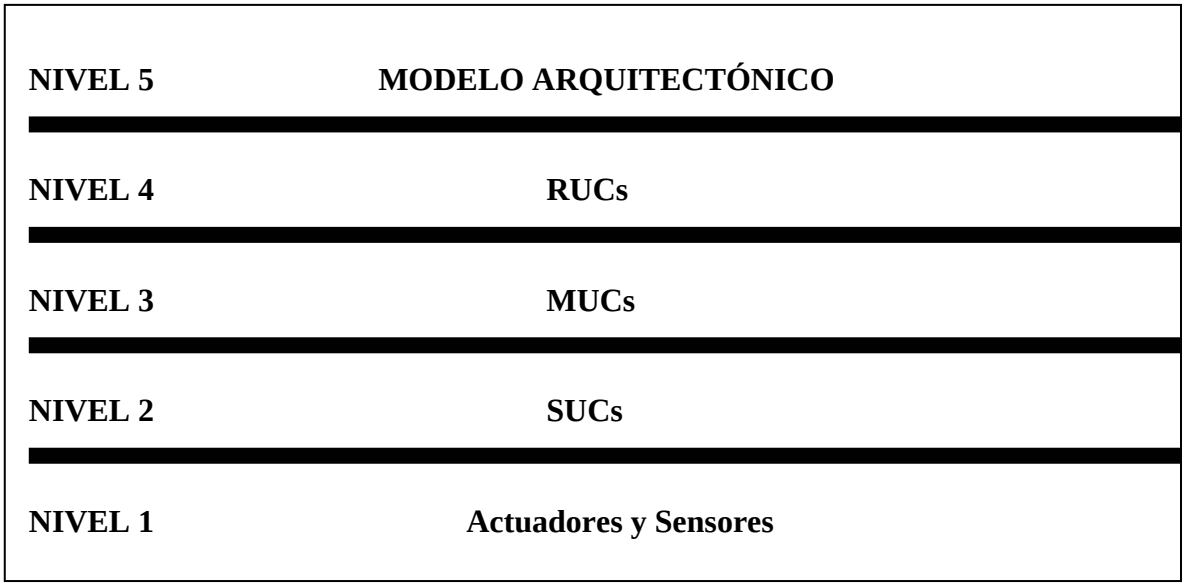


Figura 15: Cinco niveles de abstracción de la arquitectura PRISMA del robot 4U4

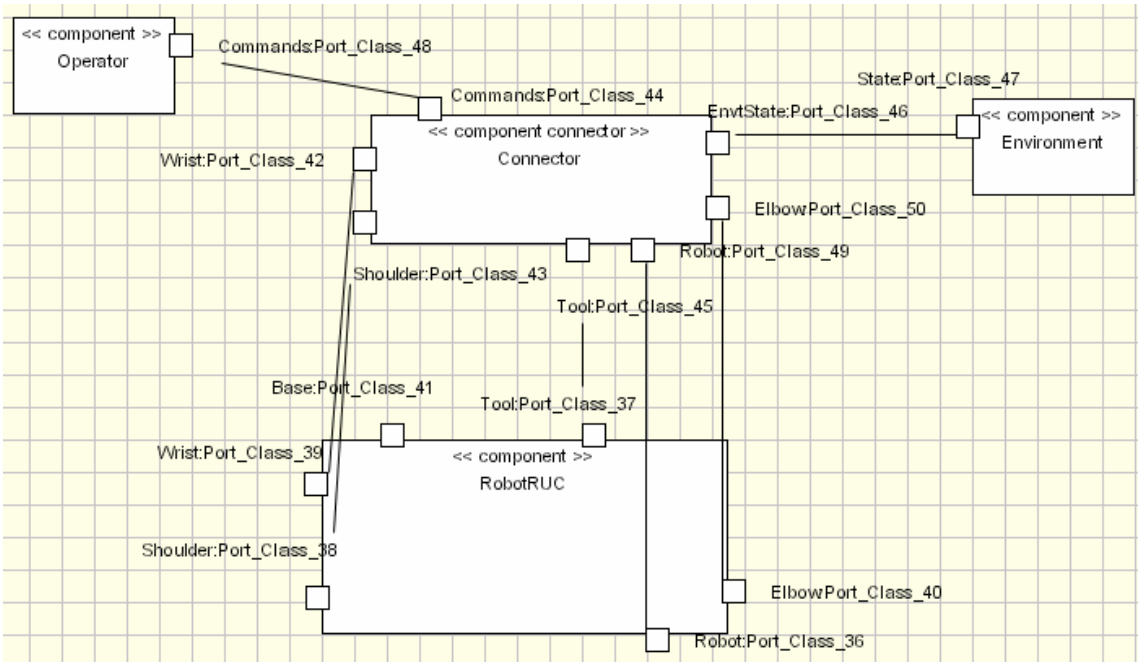


Figura 16: Modelo arquitectónico para el sistema de información del robot 4U4

7. IDENTIFICACIÓN DE CONCERNS

Una vez ya diseñado el modelo arquitectónico del robot 4U4, 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 4U4, 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 4U4 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 17) del robot. De forma que, cuando cada articulación forme el ángulo indicado, el robot alcance la posición deseada por el operario.

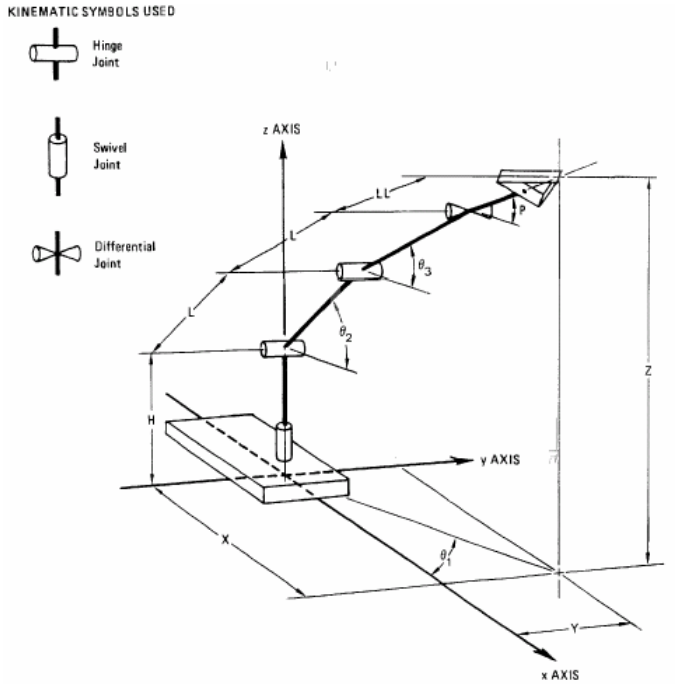


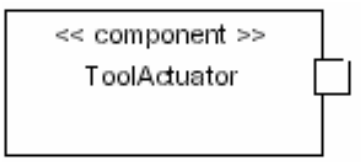
Figura 17: Eje de coordenadas de referencia del robot

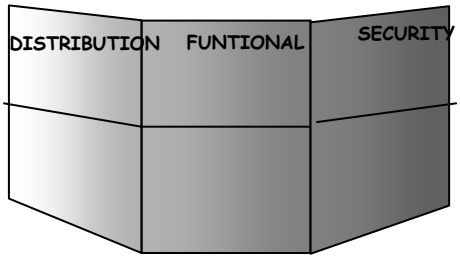
Por otro lado, los movimientos del robot los ordena un operario por control remoto, ya que es necesario que se sitúe en un lugar alejado del robot para evitar situaciones de riesgo. Estas propiedades hacen emerger un *concern* **distribución**, que cobrará más relevancia y complejidad en sistemas robóticos más grandes como el sistema teleoperado EFTCoR [EFT02].

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**.

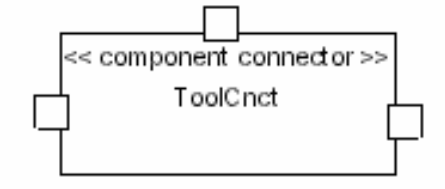
Como resultado de este análisis se han identificado los *concerns*: **funcional, distribución, seguridad y coordinación**. De forma que se van a separar cada *concern* en distintos aspectos para mejorar su reutilización y mantenibilidad de la arquitectura. Por lo tanto, un componente o sistema PRISMA del modelo arquitectónico del robot 4U4 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 18) y conectores (ver Figura 19) 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.

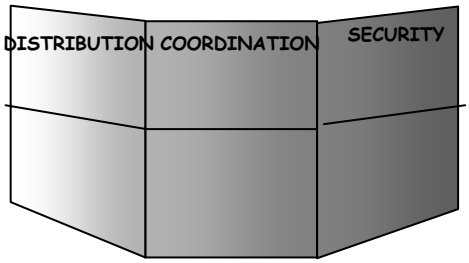




Black Box View	White Box View

Figura 18: Vista interna y externa de los componentes prisma del modelo arquitectónico robot 4U4





Black Box View	White Box View

Figura 19: Vista interna y externa de los conectores prisma del modelo arquitectónico robot 4U4

8. ESPECIFICACIÓN DE SUCS

8.1. Aspectos 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.

8.1.1. Base, Shoulder, Elbow

Los aspectos que son necesarios para especificar un SUC *Base, Shoulder o Elbow* 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.

8.1.1.1. Aspecto de Coordinación

El actuador de una articulación es aquel que permite el movimiento de ésta. Cada una de las articulaciones del robot 4U4 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 (*half-steps)* 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 20 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.

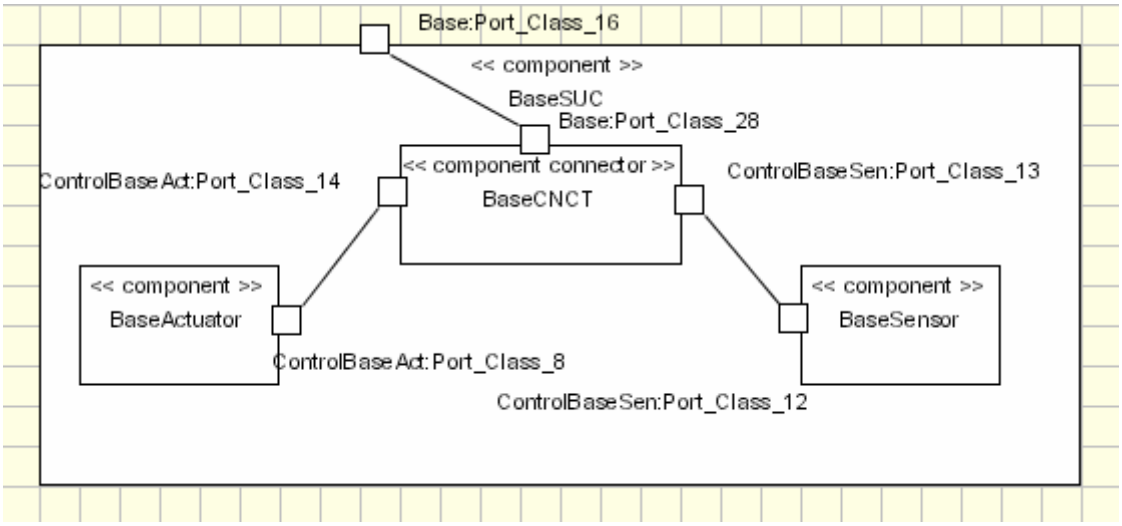


Figura 20: Actuador, conector y sensor de la base

Cada articulación tiene almacenado el ángulo en el que se encuentra respecto a la posición del eje de coordenadas de referencia (ver Figura 17) y el número de *half-steps* equivalente a dicho ángulo. Como el robot se mueve *half-steps* 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) half-steps*. De esta forma, dado un movimiento de la articulación su valor en *half-steps* 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 gesione 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 *half-steps*. 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 *half-steps*.

```
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 *half-steps* 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;
```

8.1.1.2. 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;


   Protocols
       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 (ver anexo A apartado A.5.1.1.).

8.1.1.3. Aspecto de 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;
```

8.1.1.4. 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*.

8.1.2. Wrist

Los aspectos del SUC Wrist son diferentes al resto ya que sus servicios y atributos de movimiento varían. Esto es debido a que no sólo se han de especificar el punto a desplazarte o el número de halfsteps, también se ha de especificar los grados de giro correspondientes para la rotación sobre un mismo punto de la muñeca (*roll*) y para desplazarse arriba y abajo (*pitch*). A continuación se presentan los aspectos necesarios para la muñeca.

8.1.2.1. Aspecto de Coordinación

El actuador de una articulación es aquel que permite el movimiento de ésta. Cada una de las articulaciones del robot 4U4 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 (*half-steps)* 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 y los grados de giro correspondientes para la rotación sobre un mismo

punto de la muñeca (*roll*) y para desplazarse arriba y abajo (*pitch*). 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, 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 *half-steps*. 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.

La muñeca tiene almacenado el ángulo en el que se encuentra respecto a la posición del eje de coordenadas de referencia y el número de *half-steps* equivalente a dicho ángulo. Asi como, los ángulos de los movimientos *pitch* y *roll* y su correpondiente valor en *half-steps*. Como el robot se mueve *half-steps* 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) half-steps*. Dentro del aspecto de coordinación de la muñeca, será necesarios tres atributos para poder calcular los ángulos y otros tre para almacenar los *half-steps*. Su especificación se muestra a continuación:

```
Attributes
  Variable
    leftHalfSteps: integer,
                   NOT NULL;
    rightHalfSteps: integer,
                    NOT NULL;
    tempLeftHalfSteps: integer;
    tempRightHalfSteps: integer;


  Derived
    angle:= FtransHalfstepsToAngle(halfSteps);
    pitchAngle:= FtransHalfstepsToAngle(leftHalfSteps);
    rollAngle:= FtransHalfstepsToAngle(rightHalfSteps);
```

Los movimientos por pasos y por cinemática inversa se van a realizar mediante dos servicios de moviento diferentes, *wristMoveJoint* y *wristCinematicsMoveJoint*, respectivamente. En el caso del *wristMoveJoint*, se le ha de pasar como parámetro el número de *half-steps* que se desea que se mueva la muñeca, tanto a la izquierda como a la derecha, y a qué velocidad, mientras que en el caso del *wristCinematicsMoveJoint* se va a tener que pasar el ángulo exacto a donde se quiere desplazar la muñeca, tanto el *pitch* como el *roll,* y la velocidad con que se realice dicho desplazamiento. Además de estos servicios habrá un servicio *stop* de parada de emergencia. Por lo tanto, los servicios necesarios para mover la muñeca son los siguientes:

```
wristMoveJoint(input NewleftHalfSteps: integer, input NewrightHalfSteps,
               input Speed: integer);

wristCinematicsMoveJoint(input Rolldegrees: integer, input Pitchdegrees:
                         integer, input Speed: integer);

stop();
```

Al igual que en el aspecto de coordinación del resto de SUCs, cuando el conector del SUC de la muñeca invoca unos de estos servicios de movimiento, no implica que la muñeca del robot realmente se mueva, a pesar de que el movimiento solicitado sea correcto. También en este caso, es el sensor el que le notifica que el movimiento se ha realizado satisfactoriamente mediante el servicio *moveok*.

```
moveok(output success: [0,1]);
```

Igualmente, esto implica, que la valuación asociada a los servicios *wristMovejoint* y *wristCinematicsMoveJoitn* no puede actualizar directamente el valor de la posición, y por lo tanto, es necesario tener atributos temporales en los que almacenar el movimiento solicitado. Dichos atributos serán asociados posteriormente a los atributos definitivos en la valuación del servicio *moveok.*

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 ImotionWrist

   wristMoveJoint(input NewleftHalfSteps: integer, input NewrightHalfSteps:
                  integer, input Speed: integer);

   stop();

End_Interface IMotionWrist;

Interface Iread

  moveOk(output success: [0,1]);

End_Interface IRead;

Interface ISUCWrist

  wristMoveJoint(input NewLeftHalfSteps: integer, input NewRightHalfSteps:
                 integer, input Speed: integer);

  wristCinematicsMoveJoint(input Rolldegrees: integer, input Pitchdegrees:
                           integer, input Speed: integer);

  stop();

  moveOk(output success: [0,1]);

End_Interface ISUCWrist;
```

En el caso de la coordinación de la muñeca, debido a que los atributos requeridos que tiene son los *half-steps* de la izquierda y la derecha, el servicio *begin* del aspecto de coordinación pasa los siguientes argumentos:

```
begin (InitialLeftHalfSteps: integer, InitialRightHalfSteps: integer);
```

En base al análisis anterior se construye el aspecto de coordinación del conector de la pinza. Éste es el que se presenta a continuación:

```
Coordination Aspect CProcessWristSUC using IMotionWrist, IRead, ISUCWrist

Attributes
     Variable
       leftHalfSteps: integer,
                     NOT NULL;
       rightHalfSteps: integer,
                      NOT NULL;
       tempLeftHalfSteps: integer;
       tempRightHalfSteps: integer;

     Derived
       angle:= FtransHalfstepsToAngle(halfSteps);
       pitchAngle:= FtransHalfstepsToAngle(leftHalfSteps);
       rollAngle:= FtransHalfstepsToAngle(rightHalfSteps);

  Services
      begin (InitialLeftHalfSteps: integer, InitialRightHalfSteps: integer);
          Valuations
              [begin (InitialLeftHalfSteps, InitialRightHalfSteps)]
               leftHalfSteps := InitialLeftHalfsteps,
               rightHalfSteps := InitialRightHalfSteps;

     in/out wristMoveJoint(input NewLeftHalfSteps: integer,
                           input NewRightHalfSteps:integer,
                           input Speed: integer);
          Valuations
              [in movejoint (NewLeftHalfSteps, NewRightHalfSteps,Speed)]
              tempLeftHalfSteps:= NewLeftHalfSteps,
              tempRightHalfSteps:= NewRightHalfSteps;

    in wristCinematicsMoveJoint(input Rolldegrees: integer,
                                input Pitchdegrees: integer,
                                input Speed: integer);
          Valuations
              [in cinematicsmovejoint(Rolldegrees, Pitchdegrees, Speed)]
              tempLeftHalfSteps:= FtransAngleToHalfsteps(PitchDegrees),
              tempRightHalfSteps:= FtransAngleToHalfsteps(RollDegrees);


   in/out stop();


   in/out moveok(output Success:[0,1]);
       Valuations
              {Success=1}
              [in moveok(Success)]
               leftHalfSteps:= leftHalfSteps + tempLeftHalfSteps,
                rightHalfSteps:= rightHalfSteps + tempRightHalfSteps;
       end;

    Subprocesses

       SUC= (ISUCWrist.stop?():0
            +
            ISUCWrist.wristMoveJoint ? (NewLeftHalfSteps, NewRightHalfSteps,
                                        Speed):1
            +
            ISUCWrist.wristCinematicsMoveJoint?(RollDegrees, PitchDegrees,
                                                Speed):1
            )
            →
            ISUCWrist.moveOk!(Success);
```

```
        ACT= ImotionWrist.stop!():0
             +
             IMotionWrist.wristMoveJoint!(NewLeftHalfsteps, NewRightHalfSteps,
                                          Speed):1;

        SEN= Iread.moveOk(Success);

    End_Subprocesses;

    Protocol
        CPROCESSSUC = begin(InitialHalfStep).MOTION;
        MOTION = SUC.stop ?()).EMERGENCY
                 +
                 (SUC.wristMoveJoint?(NewLeftHalfsteps, NewRightHalfSteps,Speed)
                 →
                 ACT.wristMoveJoint!(NewLeftHalfsteps, NewRightHalfSteps,
                                     Speed).ANSWER)
                 +
                (SUC.wristCinematicsMoveJoint ?(RollDegrees, PitchDegrees,
                                                Speed)
                 →
                 ACT.movejoint ! (FTransAngleToHalfSteps(RollDegrees),
                                  FTransAngleToHalfSteps(PitchDegrees),
                                  Speed)
                 .ANSWER)
                 +
                 end;

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

        EMERGENCY= ACT.stop ! ()→ end;

End_Coordination Aspect CProcessWristSUC;
```

8.1.2.2. Aspecto Funcional

El aspecto funcional del sensor y el 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 FWristAct using IMotionWrist

    Services
        begin;
        in/out wristmovejoint(input NewLeftHalfsteps: integer,
                              input NewRightHalfsteps: integer,
                              input Speed: integer);
        in/out stop();
        end;

    Subprocesses
        ROBOT = IMotionWrist.stop?():0
                +
                IMotionWrist.wristMoveJoint ? (NewLeftHalfsteps,
                                               NewRightHalfSteps, Speed):1;
        HW = IMotionWrist.stop!():0
             +
             IMotionWrist. wristMoveJoint ! (NewLeftHalfsteps,
                                             NewRightHalfSteps, Speed):1;
    End_Subprocesses;
```

```
   Protocol
       FTOOLACT = begin.MOTION;
       MOTION= (ROBOT.stop?():0
                →
                HW.stop!():0.MOTION)
               +
               (ROBOT.wristMoveJoint ? (NewLeftHalfsteps, NewRightHalfSteps,
                                        Speed):1
                →
                HW.wristMoveJoint ! (NewLeftHalfsteps, NewRightHalfSteps,
                                     Speed):1.MOTION;
               +
               end;

End_Functional Aspect FWristAct;


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

End_Functional Aspect FSensor;
```

8.1.2.3. Aspecto de Distribución

Como se ha comentado anteriormente, el aspecto distribución de las articulaciones del robot va a especificar su ubicación en una máquina. Por lo tanto, el aspecto de distribución que se va a utilizar para el SUC de la muñeca es el que se presenta a continuación:

```
Distribution Aspect DRobotLocation

   Attributes
      Constant
          location: loc;
   Services
      Begin(InitialLocation: loc);
          Valuations
              [begin (InitialLocation)]
               location := InitialLocation;

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

End_Distribution Aspect DRobotLocation;
```

8.1.2.4. Aspecto de Seguridad

El aspecto seguridad de la muñeca va a especificar los rangos máximos y mínimos de movimiento hacia su izquierda y su derecha 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 representen la articulación. La comprobación de que el movimiento es seguro se hace mediante un servicio *check* que comprueba que los grados de desplazamiento (*RollDegrees, PitchDegrees*) 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, el servicio *begin* tiene la siguiente sintaxis:

```
begin( input InitialMinimumRoll: integer, input InitialMaximumRoll: integer,
       input InitialMinimumPitch: integer,
       input InitialMaximumPitch: integer);
```

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

```
Safety Aspect SMotionWrist

   Attributes
      Constant
          minimumRoll: integer,
                       NOT NULL;
          maximumRoll: integer,
                       NOT NULL;
          minimumPitch: integer,
                        NOT NULL;
          maximumPitch: integer,
                        NOT NULL;

   Services
      begin( input InitialMinimumRoll: integer,
             input InitialMaximumRoll: integer,
             input InitialMinimumPitch: integer,
             input InitialMaximumPitch: integer);
                Valuations
                    [begin (InitialMinimumRoll, InitialMaximumRoll,
                            InitialMinimumPitch, InitialMaximumPitch)]
                       minimumRoll:= InitialMinimumRoll,
                       maximumRoll:= InitialMaximumRoll,
                       minimumPitch:= InitialMinimumPitch,
                       maximumPitch:= InitialMaximumPitch;


       in check(input RollDegrees: integer, input PitchDegrees: integer,
              output Secure: boolean);
            Valuations
              {(DegreesRoll >= minimumRoll) and (DegreesRoll <= maximumRoll)
              and
              (DegreesPitch >= minimumPitch) and (DegreesPitch <=maximumPitch)}
                [check(RollDegrees, PitchDegrees, Secure)]
               Secure := true};
```

```
            {(DegreesRoll < minimumRoll) or (DegreesRoll > maximumRoll) or
             (DegreesPitch < minimumPitch) or (DegreesPitch > maximumPitch)}
               [check(RollDegrees, PitchDegrees, Secure)]
             Secure := false;

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

End_Safety Aspect SMotionWrist;
```

8.1.3. Tool

Como ya se ha dicho en este documento previamente, la herramienta se va a tratar a parte. En este caso la herramienta del robot se corresponde con una pinza. A continuación se presentan los aspectos necesarios para la pinza.

8.1.3.1. Aspecto de Coordinación

El actuador de la pinza permite que ésta se abra y se cierre. La apertura o cierre de la pinza se puede indicar por pasos o por pulgadas. Sin embargo, el actuador solicita al roboy el movimiento en pasos, haciendo previamente el conector la conversión pulgadas a pasos en el segundo caso. En el primer caso se indica el número de medios pasos (*half-steps)* que se va a abrir o cerrar la pinza y en el segundo la cantidad de pulgadas.

La pinza tiene almacenada la posición en la que se encuentra en cada momento. De forma que, dado un movimiento de la articulación, su posición se debe actualizar. Dicho control se centraliza en la coordinación que realiza el conector de la pinza. Dentro del aspecto de coordinación del conector del SUCs de la pinza hay un atributo para poder especificar el número de pulgadas y el número de *half-steps*. Su especificación se muestra a continuación:

```
   Attributes
     Variable
       halfSteps: integer,
                  NOT NULL;
     Derived
       inches:= FtransHalfstepsToInches(halSteps);
```

Los movimientos por pasos y por pulgadas se van a realizar mediante dos servicios de movimiento diferentes, *movejoint* y *moveinches*, respectivamente. En el caso del *movejoint*, se le ha de pasar como parámetro el número de *half-steps* que se desea que se mueva la herramienta y a qué velocidad, mientras que en el caso del *moveinches* se va a pasar el número de pulgadas exacto a donde se quiere abrir o cerrar la pinza (si es una apertura o un cierre depende del número de pulgadas del estado anterior, si es mayor o menor) y la velocidad. Además de estos servicios habrá un servicio *close* para cerrar la pinza y el servicio *stop* de parada de emergencia. Por lo tanto, los servicios necesarios para mover la pinza son los siguientes:

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

moveinches(input NewInches: integer, input Speed: integer);
```

```
close();
stop();
```

Al igual que en el aspecto de coordinación del resto de SUCs, cuando el conector del SUC de la pinza invoca unos de estos servicios de movimiento, no implica que la pinza del robot realmente se mueva, a pesar de que el movimiento solicitado sea correcto. También en este caso, es el sensor el que le notifica que el movimiento se ha realizado satisfactoriamente mediante el servicio *moveok*.

```
moveok(output success: [0,1]);
```

Igualmente, esto implica, que la valuación asociada a los servicios *movejoint, moveinches* y *close* no puede actualizar directamente el valor de la posición, y por lo tanto, es necesario tener atributos temporales en los que almacenar el movimiento solicitado. Dichos atributos serán asociados posteriormente a los atributos definitivos en la valuación del servicio *moveok.*

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 IToolJoint
       movejoint(input NewHalfsteps: integer, input Speed: integer);
       close();
       stop();
End_Interface IToolJoint;

Interface IRead
        moveok(output Success: [0,1]);
End_Interface IRead;

Interface ISUCTool
   movejoint(input NewHalfsteps: integer, input Speed: integer);
   moveinches( input NewInches: integer, input Speed: integer);
   close();
   stop();
   moveok(output Success: [0,1]);
End_Interface ISUCTool;
```

En el caso de la coordinación de la herramienta, debido a que los atributos requeridos que tiene son *inches* y *halfSteps,* el servicio *begin* del aspecto de coordinación pasa los siguientes argumentos:

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

En base al análisis anterior se construye el aspecto de coordinación del conector de la pinza. Éste es el que se presenta a continuación:

```
Coordination Aspect CProcessTool using IToolJoint, IRead

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

     Derived
      inches:= FtransHalfstepsToInches(halSteps);

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 moveinches(input NewInches: integer, input Speed: integer);
           Valuations
            [in moveinches(NewInches, Speed)]
             tempHalfSteps:= FtransInchesToHalfsteps(NewInches);

       in/out close();
           Valuations
            [in close()]
             tempHalfSteps:= FtransInchesToHalfSteps(0);

      in/out stop();

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

       end;

   Subprocesses

       SUC=  (ISUCTool.stop ? ():0
              + ISUCTool.moveinches ? (NewInches, Speed):1
              + ISUCTool.movejoint ? (halfstep, Speed):1
              + ISUCTool.close ? ():1)
              →
              ISUCTool.moveOk !(Success);


       ACT= ItoolJoint.stop ! ():0
            + ItoolJoint.movejoint ! (halfstep, Speed):1
            + ItoolJoint.close ! ():1;


       SEN= Iread.moveOk?(Success);

   End_Subprocesses;
```

```
   Protocol
       CPROCESSTOOL = begin(InitialInches, InitialHalfSteps).MOTION;
       MOTION= SUC.stop ? ():0.EMERGENCY
               +
              (SUC.moveinches?(NewInches, Speed):1
               →
               ACT.movejoint !(FTransInchesToHalfsteps(NewInches), Speed):1
               .ANSWER)
               +
              (SUC.movejoint ? (Halfstep, Speed):1
               →
               ACT.movejoint ! (Halfstep, Speed).1.ANSWER)
               +
               (SUC.close ? ():1
               →
               ACT.close ! ().ANSWER)
               +
               end;

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

      EMERGENCY= ACT.stop ! ()→ end;

End_Coordination Aspect CProcessTool;
```

8.1.3.2. Aspecto Funcional

El aspecto funcional del sensor y el 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 FToolAct using IToolJoint

   Services
       begin;
       in/out movejoint(input NewHalfsteps: integer, input Speed: integer);
       in/out close();
       in/out stop();
       end;

   Subprocesses
       ROBOT = ItoolJoint.stop?():0
               +
               ItoolJoint.movejoint ? (NewHalfsteps, Speed):1
               +
               ItoolJoint.close ? ():1;
       HW = ItoolJoint.stop!():0
            +
            ItoolJoint.movejoint ! (NewHalfsteps, Speed):1
            +
            ItoolJoint.close ! ():1;

   End_Subprocesses;
```

```
   Protocol
       FTOOLACT = begin.MOTION;
       MOTION= (ROBOT.stop?():0
                →
                HW.stop!():0.MOTION)
               +
               (ROBOT.movejoint ? (NewHalfsteps, Speed):1
                →
                HW.movejoint ! (NewHalfsteps, Speed):1.MOTION)
                +
                (ROBOT.close ? ():1
                →
                 HW.close ? ():1.MOTION)
               +
               end;

End_Functional Aspect FToolAct;


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;

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

End_Functional Aspect FSensor;
```

8.1.3.3. Aspecto de Distribución

Como se ha comentado anteriormente, el aspecto distribución de las articulaciones del robot va a especificar su ubicación en una máquina. Por lo tanto, el aspecto de distribución que se va a utilizar para el SUC de la pinza es el que se presenta a continuación:

```
Distribution Aspect DRobotLocation

   Attributes
      Constant
          location: loc;
   Services
      Begin(InitialLocation: loc);
          Valuations
              [begin (InitialLocation)]
               location := InitialLocation;

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

End_Distribution Aspect DRobotLocation;
```

8.1.3.4. Aspecto de Seguridad

El aspecto seguridad de la pinza va a especificar los rangos máximos y mínimos de pulgadas y deberá de controlar que sus movimientos sean seguros. Un movimiento es seguro siempre que este dentro de los rangos mínimos y máximos de pulgadas. Los rangos máximos y mínimos son dos atributos del aspecto que se les dará el valor predeterminado por el robot cuando se instancie la pinza. La comprobación de que el movimiento es seguro se hace mediante un servicio *checkInches* que comprueba que el número de pulgadas (*NewInches*) están dentro de los límites (Secure := true).

```
Safety Aspect SToolMotion

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

   Services

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

       in checkInches(input NewInches: integer, output Secure: boolean);
           Valuations
              {(NewInches >= minimum) and (NewInches <= maximum)}
                [checkInches(NewInches, Secure)]
              {Secure := true},

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

   Protocol
       SMOTION = begin.CHECKING;
      CHECKING= checkInches ? (NewInches, Secure) +
                end;

End_Safety Aspect SToolMotion;
```

8.2. Sistemas SUCs y sus componentes

8.2.1. Base, Shoulder, Elbow

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 que forman para el caso general de la Base, el shoulder y el elbow. Cada uno de ellos será un tipo, de forma que el tipo actuador, el tipo sensor y el tipo conector del SUC formarán el tipo 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 SUCConnector.*

El componente *Actuator* especifica el conjunto de puertos necesarios para comunicarse con el connector 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 (*IMotionJoint),* 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 Import DLocation;
   Initialize
       new (Location: loc)
       {
          FSensot.begin();
          DLocation.begin(Location: loc);
       }
   End_Initialize;
   Destruction
       destroy ()
       {
          FSensor.end();
          DLocation. end();
       }
   End_Destruction;

End_Component Sensor;
```

SUCconnector es un connector con tres roles, que son definidos mediante un nombre, un tipo, que se corresponde con una de las interfaces declaradas (*IMotionJoint, IRead e ISUC),* 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 *SUCconnector (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 *SUCConnector* 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 SUCconnector

   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 SUCconnector;
```

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;
        VarSUCConnector: SUCconnector;
   End_Variable;


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


   Bindings
    VarSUCconnector.SUC ←→ SUC.SUC;
   End_Bindings;

   Initialize
       new ()
       {
         VarActuator = new Actuator(Location: loc);
          VarSensor = new Sensor(Location: loc);
         VarConnector = new SUCconnector(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 SUCconnector(HalfSteps: integer,
                                 Location: loc,
                                 Minimum: integer,
                                 Maximum: integer);
```

Estos servicios se invocan y dan valor en la creación de instancias del sistema, tal y como se muestra a continuación:

```
BaseSUC = new SUC(){
   BaseActuator= new Actuator(localhost);
   BaseSensor= new Sensor(localhost);
   BaseConnector = new SUCconnector(0,0,localhost,90,-90);
};

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

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

8.2.2. Wrist

Al igual que para el caso de las articulaciones Base, Shoulder y Elbow, es necesario definir un tipo SUC para la muñeca (Wrist) y un tipo actuador, sensor y conector que lo formen. Estos tipos son diferentes a los especificados en la sección anterior porque los aspectos que los forman son distintos, excepto para el caso del sensor. A continuación se presentan los tipos y la instancia asociada para formar la articulación Wrist del robot.

```
Component WristActuator
   Ports
      ControlBaseAct: IMotionWrist,
           Subprocess FWristAct.ROBOT;
      HWRS232: IMotionWrist,
             Subprocess FWristAct.ROBOT;
   End_Ports;

   Functional Aspect Import FWristAct;
   Distribution Aspect Import DLocation;

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


Connector Wristconnector

   Roles
      Wrist: ISUCWrist,
           Subprocess CProcessWristSUC.SUC;
      ControlWristAct: IMotionWrist,
           Subprocess CProcessWristSUC.ACT;
      ControlWristSensor: IRead,
           Subprocess CProcessWristSUC.SEN;
   End_Roles;

   Coordination Aspect Import CProcessWristSUC;
   Distribution Aspect Import DLocation;
   Safety Aspect Import SMotionWrist;

  Weavings

        CProcessWristSUC. wristMoveJoint(NewLeftHalfSteps, NewRightHalfSteps,
                                         Speed: integer);
        afterIf (Secure = true)
        SMotionWrist.check(FtransHalfStepsToAngle(RollDegrees),
                       FtransHalfStepsToAngle(PitchDegrees), Secure);

       CProcessWristSUC. wristCinematicsMoveJoint(Rolldegrees, Pitchdegrees,
                                                  Speed);
       afterIf (Secure = true) SMotionWrist.Check(Rolldegrees, Pitchdegrees,
                                           Segure);

   End_Weavings;
   Initialize
       new (InitialLeftHalfSteps: integer, InitialRightHalfSteps: integer,
            Location: loc, InitialMinimumRoll: integer,
            InitialMaximumRoll: integer, InitialMinimumPitch: integer,
            InitialMaximumPitch: integer)
       {
         CProcessWristSUC.begin(InitialLeftHalfSteps: integer,
                                InitialRightHalfSteps: integer);
```

```
            DLocation.begin(Location: loc);
            SmotionWrist.begin(InitialMinimumRoll: integer,
                               InitialMaximumRoll: integer,
                               InitialMinimumPitch: integer,
                              InitialMaximumPitch: integer);
                }
    End_Initialize;
    Destruction
        destroy ()
        {
           CProcessWristSUC.end();
           DLocation.end();
           SmotionWrist.end();
        }
    End_Destruction;


End_Connector WristConnector;

System WristSUC

    Ports
       SUC: ISUCWrist;
    End_Ports;

    Variables
        VarActuator: WristActuator;
        VarSensor: Sensor;
         VarSUCConnector: Wristconnector;
    End_Variable;


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


    Bindings
     VarSUCconnector.SUC ←→ SUC.SUC;
    End_Bindings;

    Initialize
        new ()
        {
          VarActuator = new WristActuator(Location: loc);
           VarSensor = new Sensor(Location: loc);
          VarSUCConnector = new Wristconnector(InitialLeftHalfSteps: integer,
                                               InitialRightHalfSteps: integer,
                                               Location: loc,
                                               InitialMinimumRoll: integer,
                                               InitialMaximumRoll: integer,
                                               InitialMinimumPitch: integer,
                                               InitialMaximumPitch: integer)
        }
    End_Initialize;

    Destruction
        destroy ()
        {
           VarActuator.destroy();
           VarSensor.destroy();
           VarConnector.destroy();
        }
    End_Destruction;
```

```
End_System WristSUC;


Wrist = new WristSUC(){
   WristAct= new WristActuator(localhost);
   WristSensor= new Sensor(localhost);
   WristCnct = new WristSUCconnector(0,0,0,0,localhost,90,-90,270,-270);
};
```

8.2.3.Tool

También, el SUC de la pinza varía sustancialmente, utilizando aspectos diferentes. La especificación del SUC completo de la pinza es el que se presenta a continuación:

```
Component ToolActuator

   Ports
      ControlToolAct: IToolJoint,
           Subprocess FToolAct.ROBOT;
      HWRS232: IToolJoint,
             Subprocess FToolAct.ROBOT;

   End_Ports;

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

End_Component ToolActuator;

Connector ToolConnector

   Roles
      Tool: ISUCTool,
           Subprocess CProcessTool.SUC;
      ControlBaseAct: IToolJoint,
           Subprocess CProcessTool.ACT;
      ControlBaseSen:IRead,
           Subprocess CProcessTool.SEN;
   End_Roles;

   Coordination Aspect Import CProcessTool;
   Distribution Aspect Import DLocation;
   Safety Aspect Import SToolmotion;
```

```
  Weavings

     Coordination Aspect
        CProcessTool.movejoint(NewHalfsteps, Speed)
        afterIf (Secure = true)
        SToolMotion.checkInches(FTransHalfStepsToInches(NewHalfSteps), Secure);

       CProcessTool.moveinches(NewInches, Speed)
       afterIf (Secure = true) SToolMotion.checkInches(NewInches, Segure);

   End_Weavings;
   Initialize
       new (InitialHalfSteps: integer, Location: loc, InitialMinimum: integer,
            InitialMaximum: integer)
       {
          CProcessTool.begin(InitialHalfSteps: integer);
          DLocation.begin(Location: loc);
          StoolMotion.begin(InitialMinimum: integer, InitialMaximum: integer);
       }
   End_Initialize;
   Destruction
       destroy ()
       {
          CProcessTool.end();
          DLocation. end();
          StoolMotion.end();
       }
   End_Destruction;

End_Connector ToolConnector;


System ToolSUC

   Ports
      Tool: ISUCTool;
   End_Ports;

   Variables
       VarActuator: WristActuator;
       VarSensor: Sensor;
        VarSUCConnector: Wristconnector;
   End_Variable;


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


   Bindings
    VarSUCconnector.SUC ←→ SUC.SUC;
   End_Bindings;

   Initialize
       new (){
           VarActuator = new ToolActuator(Location: loc);
           VarSensor = new Sensor(Location: los);
           VarConnector = new ToolConnector(InitialHalfSteps: integer,
                                            Location: loc,
                                            InitialMinimum: integer,
                                            InitialMaximum: integer);
         }
   End_Initialize;
```

```
   Destruction
       destroy ()
       {
          VarActuator.destroy();
          VarSensor.destroy();
          VarConnector.destroy();
       }
   End_Destruction;


System ToolSUC;


Tool = new ToolSUC(){
   ToolAct= new WristActuator(localhost);
   ToolSensor= new Sensor(localhost);
   ToolCnct = new WristSUCconnector(0,localhost,0,3);
};
```

9. MUC

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.

9.1. Aspectos del MUC

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 21). 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 *half-steps*, indicando los *half-steps* que se ha de mover cada una de ellas. Finalmente, es posible solicitarle a un MUC el movimiento por *half-steps* 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.

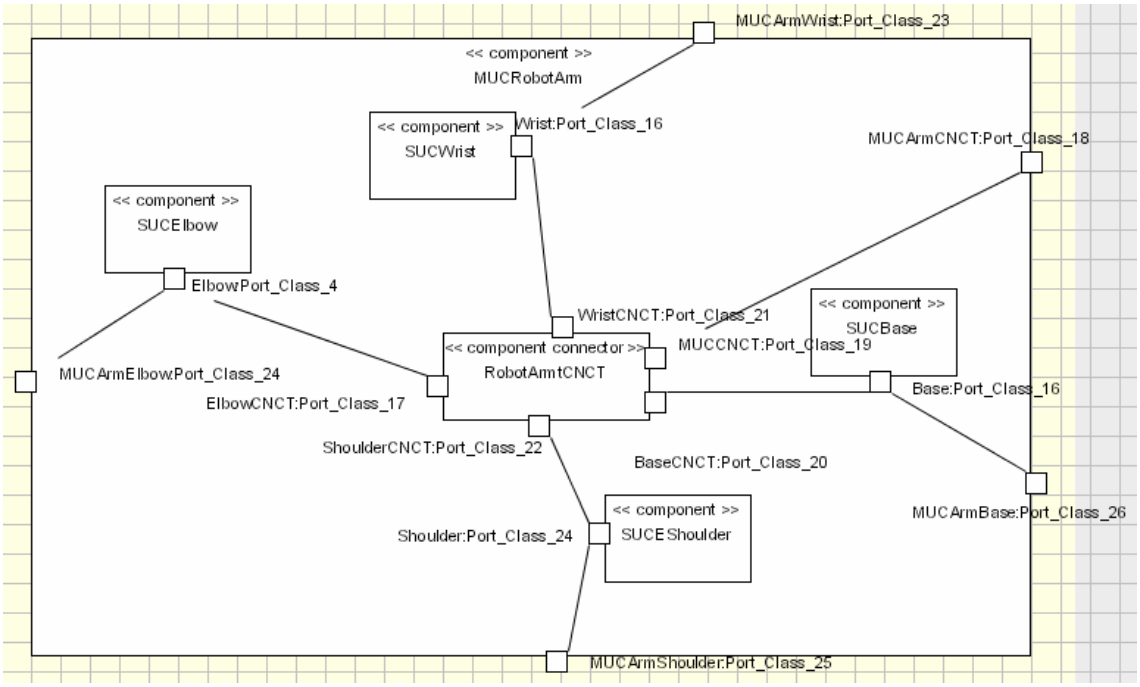


Figura 21: MUC del brazo robot

Los servicios de movimiento del MUC son publicados por una serie de interfaces que se muestran a continuación:

```
Interface IArmMotion

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

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

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


      wristmovejoint(input NewleftHalfSteps: integer,
                     input NewrightHalfSteps: integer, input Speed: integer);

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

End_Interface IArmMotion;


Interface IMotionSUC
  movejoint(input NewHalfSteps: integer, input Speed: integer);

  cinematicsmovejoint( input NewAngle: integer, input Speed: integer);

   moveok(output success: [0,1]);
End_Interface IMotionSUC;
```

```
Interface IEmergency
    stop();
End_Interface IEmergency;


Interface IMotionSUCWrist

  wristmovejoint(input NewleftHalfSteps: integer,
                 input NewrightHalfSteps: integer, input Speed: integer);

  wristcinematicsmovejoint(input Rolldegrees: integer,
                           input Pitchdegrees: integer, input Speed: integer);
  moveok(output success: [0,1]);

End_Interface IMotionSUCWrist;
```

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

End_Coordination Aspect CProcessMUC;

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.

```
Distribution Aspect DRobotLocation

   Attributes
       location: loc;
   Services
      Begin(InitialLocation: loc);
          Valuations
              [begin (InitialLocation)]
               location := InitialLocation;

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

End_Distribution Aspect DRobotLocation;
```

9.2. Sistema MUC y sus componentes

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 connector 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 (ver Figura 21). El nombre que se les ha dado a cada uno de los roles en la especificación, es el mismo que aparece en la Figura 21. Además del nombre, se declara el tipo, que se corresponde con una de las interfaces declaradas (*IMotionJoint, IArmMotion),* 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;
```

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 21. 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 21). 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 SUCconnector(0,0,localhost,90,-90);
                    };

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

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

          Wrist = new WristSUC(){
                              WristAct= new WristActuator(localhost);
                              WristSensor= new Sensor(localhost);
                              WristCnct = new WristSUCconnector(0, 0, 0, 0,
                                                  localhost, 90, -90, 270, -270);

                    };
          ARMCNCT = new RobotArmCnct(localhost);
         };
```

10. RUC

En este apartado se van a especificar los aspectos y los elementos arquitectónicos necesarios para definir el RUC, es decir, el robot 4U4 completo.

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:

10.1.1. Aspectos 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 22). 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 *half-steps*, indicando los *half-steps* que se ha de mover cada una. Finalmente, es posible solicitarle a un RUC el movimiento por *half-steps* de una sola de las articulaciones que lo forman.

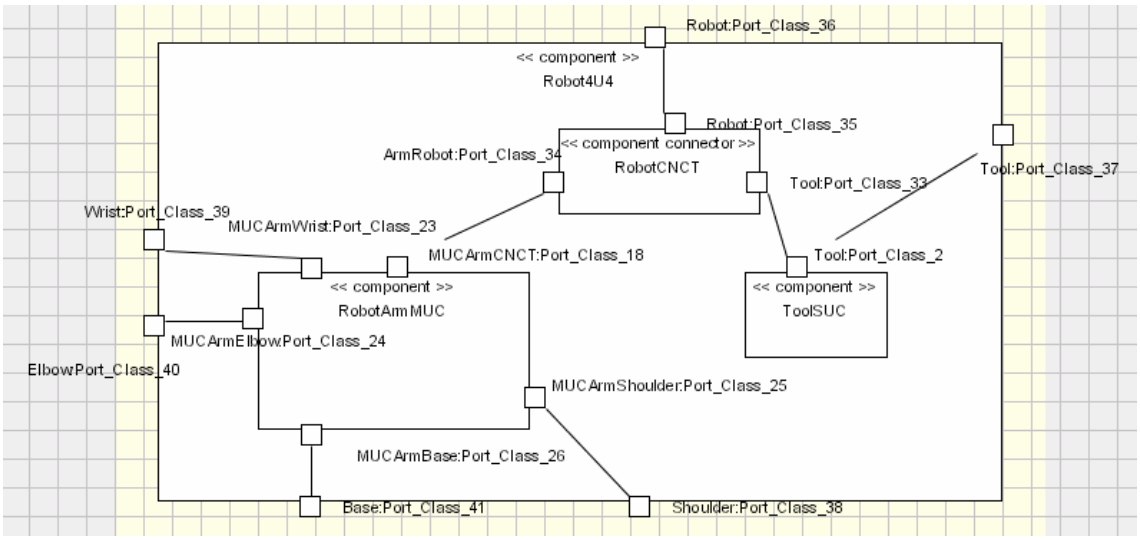


Figura 22: RUC robot 4U4

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 necesariso 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 *half-steps* 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.

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

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

moveSteps(input Joint: String, input NewHalfSteps: integer,
          input Speed: integer);

wristMoveJoint(input NewleftHalfSteps: integer,
               input NewrightHalfSteps: integer, input Speed: integer);
```

Además de estos servicios, existe un servicio para el cierre de la pinza. Éste se especifica a continuación.

```
close();
```

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. La sintaxis del servicio es la siguiente:

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

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 IRobotMotion

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

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


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

```
   wristMoveJoint(input NewleftHalfSteps: integer,
                  input NewrightHalfSteps: integer, input Speed: integer);

   close();

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

End_Interface IRobotMotion;
```

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;
```

10.1.2. Aspecto de Distribución

El aspecto distribución del robot 4U4 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 ésto 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.

```
Distribution Aspect DRobotLocation

   Attributes
     Constant
       location: loc,
                 NOT NULL;
   Services
      begin(InitialLocation: loc);
          Valuations
             [begin (InitialLocation)]
              location := InitialLocation;

      end;

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

End_Distribution Aspect DRobotLocation;
```

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;
```

10.2. Sistema RUC y sus componentes

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 4U4 del modelo arquitectónico (*configuración*).

El connector *RobotCnct* tiene tres roles para comunicarse, tanto con la herramienta y el brazo robot, como con el nivel de granularidad superior (ver Figura 22). Además del nombre, se declara el tipo, que se corresponde con una de las interfaces declaradas (*IRobotMotion, IArmMotion, ISUCTool),* 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 Robot4U4CNCT;
```

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 22, 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* (ver Figura 22). 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;
```

Robot4U4 = **new** RUC{

```
TOOL = new ToolSUC(){
   ToolAct= new WristActuator(localhost);
   ToolSensor= new Sensor(localhost);
   ToolCnct = new WristSUCconnector(0,localhost,0,3);
```

};

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

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

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

          Wrist = new WristSUC(){
                              WristAct= new WristActuator(localhost);
                              WristSensor= new Sensor(localhost);
                              WristCnct = new WristSUCconnector(0, 0, 0, 0,
                                                localhost, 90, -90, 270, -270);
                        };
          ARMCNCT = new RobotArmCnct(localhost);
```

```
        };
    4U4Cnct = new RobotCnct(localhost,4,15,17.8,2.25,1.25,3.5,-90º);
}
```

11. SISTEMA FINAL

En este apartado se van a especificar los aspectos y lo elementos arquitectónicos necesarios para definir el sistema final, en el que el operador manipula el ordenador en un entorno determinado (ver Figura 23). Si bien es cierto, como el entorno es variable no lo vamos a definir en esta especificación con el objetivo de no complicarla con este elemento arquitectónico. Por lo tanto, se van a especificar el operario y el conector que lo comunica con el robot 4U4.

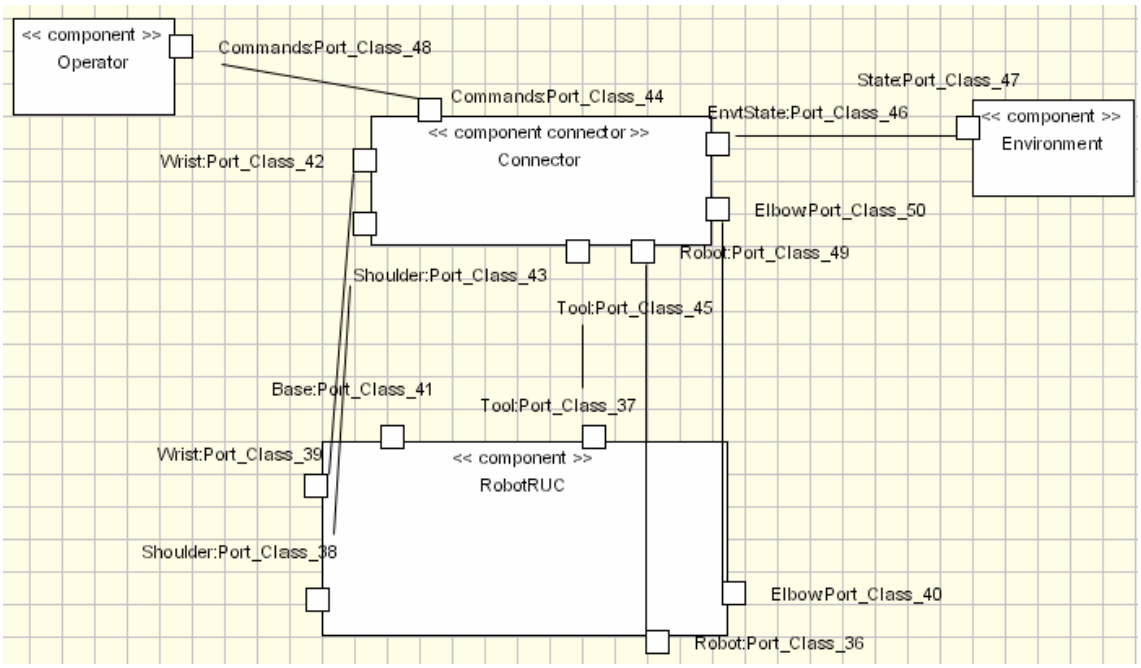


Figura 23: Modelo arquitectónico para el sistema de información del robot 4U4

11.1. Operario

Los aspectos necesarios para definir un operario son los aspectos de funcionalidad, presentación, distribución y replicación. Éstos se presentan a continuación, pero en un futuro se considerará el aspecto autenticación de los operarios mediante un login y pasword, para tener control de los operarios que manipulan el robot de forma remota.

11.1.1. Aspectos del Operario

11.1.1.1. Aspecto de presentación

El aspecto va albergar el conjunto de componentes gráficos que van a definir la interfaz de usuario con la que va a trabajar el operario.

Los componentes gráficos se sincronizarán mediante el weaving con el aspecto funcional. Si bien es cierto, debido a lo extenso que puede ser este aspecto, únicamente se definirá, a modo de ejemplo, la interfaz gráfica necesaria para un solo servicio. El servicio va a ser

cinematicsmoverobot(x, y, z, p, r, i, Speed). Supongamos la siguiente interfaz grafica (ver figura 24):

Figura 24: Ejemplo de interfaz gráfica asociada al movimiento cinemático del robot

El aspecto de presentación para dicha interfaz gráfica sería la siguiente:

```
Presentation Aspect PRobot using IGraphics

   Attributes
     Variable
       formcinematicsrobot: window;
       labelX, labelY, labelZ, labelInches: label;
       labelPitch, labelRoll, labelSpeed: label;
       textX, textY, textZ, textInches: text;
       textPitch, textRoll, textSpeed: text;
       buttonMovement: button;

       … …

  Services
       begin();

       in onclickmoverobot(output textX: text, output textY: text,
                           output textZ: text, output textInches: text, output
                           textPitch: text, output textRoll: text,
                           output textSpeed: text);
       in showWindow (input Startform: window);
       in closeWindow (input Startform: window);
       in showMessage (input Message: string);

       … …
       end;

   Subprocesses

       APPLICATION= IGraphics.onclickButton ? (textX, textY, textZ, textInches,
                                               textPitch, textRoll, textSpeed)
                    +
                    IGraphics.showWindow ?(Startform)
                    +
                    IGraphics.closeWindow ? (Startform)
                    +
                    IGraphics.showMessage ? (Message);
              … …
   End_Subprocesses;
   Protocol
       PROBOT = begin.DRAW;
       DRAW = showWindow (Startform).MANAGEMENT + …;
       MANAGEMENT= onclickmoverobot(textX, textY, textZ, textInches, textPitch,
                                    textRoll, textSpeed).MANAGEMENT +
                    closeWindow(Startform).DRAW + end;


End_Presentation Aspect PRobot;
```

11.1.1.2. Aspectos de Distribución y Replicación

El aspecto distribución del operario va a especificar la ubicación de la máquina desde la cual éste manipula al robot, así como los servicios de movilidad y replicación; tanto para que el operario pueda gestionar el robot desde distintas máquinas, como para que haya más de un operario lanzando peticiones al robot.

```
Distribution Aspect DManagement

   Attributes
         location: loc NOT NULL;
         locMax: loc NOT NULL;
         locMin: loc NOT NULL;

   Services
      begin(Location:loc, LocMAx:loc, LocMin:loc)
       Valutions
          [begin(Location,LocMax, LocMin)]
           location:=Location,
           locMax:=LocMax,
           locMin:=LocMin;


      checkLocation( input Location:loc, output checkC:bool)
         Valutions
             {Location>locMin & Location<locMax}
             [checkLocation(Location,checkC)]
             checkC:=true;


             {Location<locMin & Location>locMax}
              [checkLocation(Location,checkC)]
              checkC:=false;


      move(input Newlocation:loc)
          Valuations
            {NewLocation>locMin & NewLocation<locMax}
            [move(input NewLocation)]
            location = NewLocation;

      end;

    Constraints
           always{ location >locMin & location< locMax};


    Protocols
           mobility  ≡  begin.MOBILITY2;
            MOBILITY2≡ end + checkLocation + move.MOBILITY2;

End_Distribution Aspect DManagement;
```

```
Replication Aspect ROperator

      Services
           begin;
           in replicate(input NewLocation:loc);
           end;

      Protocols
                replication  ≡  begin.REPLICATION1;
                REPLICATION1 ≡ end + replicate.REPLICATION1;

End_Replication Aspect ROperator;
```

11.1.1.3. Aspecto Funcional

El aspecto funcional del operario va a consistir en la solicitud de los servicios de movimiento al robot.

```
Functional Aspect FAskMovement using IOperator, IEmergency
   Services
       begin;

       out stepMoveRobot( input BaseHalfSteps: integer,
                          input ShoulderHalfSteps: integer,
                          input ElbowHalfSteps: integer,
                          input LeftHalfSteps, RightHalfSteps: integer,
                          input HandHalfSteps: integer,
                          input Speed: integer);
      out cinematicsMoveRobot( input x: integer, input y: integer,
                               input z: integer, input p: integer,
                               input r: integer, input i: integer,
                               input Speed: integer);

      out moveSteps(input Joint: String, input NewHalfSteps: integer,
                    input Speed: integer);
      out wristMoveJoint(input NewleftHalfSteps: integer,
                         input NewrightHalfSteps: integer,
                         input Speed: integer);
      out stop();
      out close();

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

      end;

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

```
   Protocol
       FASKMOVEMENT = begin.MOTION;
       MOTION= ASK.MOTION +
               end;

End_Functional Aspect FAskMovement;
```

Se ha de hacer notar que la interfaz *Ioperator* coincide con la de *IrobotMotion,* excepto por el hecho de que incorpora un servicio más, el servicio *stop.*

11.1.2. Componente Operario

Tras haber definido los aspectos necesarios para especificar el componente *Operator*, a continuación se va a definir éste. Una vez definido el operario (*RobotOperator)*, éste será instanciado para crear el operario que maneja el robot 4U4 del modelo arquitectónico (*configuración*). A continuación, se presenta una instancia del tipo *Operator:*

```
Componet RobotOperator
   Ports
      Commands: IOperator,
           Subprocess FAskMovement.ASK;
   End_Ports;

   Functional Aspect Import FAskMovement;
   Presentation Aspect Import PRobot;
   Distribution Aspect Import DManagement;
   Replication Aspect Import ROperator;

  Weavings
       PRobot.onClickButton(textX, textY, textZ, textInches, textPitch,
                            textRoll, textSpeed)
        before
        FAskMovement. cinematicsMoveRobot(x, y, z, p, r, i, Speed);

        FAskMovement. moveOk(Success)
         beforeif(Success = 1)
        PRobot. showMessage(" The robot has been moved succesfully");

   Initialize
      new (Location:loc, LocMAx:loc, LocMin:loc){
                FAskMovement.begin();
                PRobot.begin();
                DManagement. begin(Location:loc, LocMAx:loc, LocMin:loc);
                ROperator.begin();
              }
   End_Initialize;

   Destruction
       destroy ()
       {
         FAskMovement.end();
         PRobot.end();
         DManagement.end();
         ROperator.end();
       }
   End_Destruction;

End_Component RobotOperator;
```

```
OPERATOR = new RobotOperator (localhost, 0, 100);
```

11.2. Conector del Modelo Arquitectónico

Los aspectos necesarios para definir el conector son los aspectos de coordinación y distribución. Si bien es cierto, cuando el entorno se considere, deberá incluirse un aspecto de seguridad para tener en cuenta las restricciones propias del entorno en el que se está ejecutando el robot.

11.2.1. Aspectos del Conector

11.2.1.1 Aspecto de Coordinación

El conector entre el operario y el robot hace la función de repetidor entre ambos, simplemente reenvia al robot los servicios solicitados por el operario. El aspecto de coordinación del conector es el que se muestra a continuación:

```
Coordination Aspect CRelayStation using IOperator, IEmergency

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

  Services
      begin;

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

      in/out 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 stop();

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

      end;

   Subprocesses

       REQUESTMOVEMENT= IOperator.stop ? ()
                        +
                       (IOperator.stepMoveRobot ? (BaseHalfSteps,
                                                   ShoulderHalfSteps,
                                                   ElbowHalfSteps,
```

```
                                        LeftHalfSteps,
                                        RightHalfSteps,
                                        HandHalfSteps, Speed)
                    +
                    IOperator.cinematicsMoveRobot ? (x, y, z, p, r, i,
                                          Speed)
                   +
                    IOperator.moveSteps ? (Joint, NewHalfSteps, Speed)
                    +
                    IOperator.wristMoveJoint ? (NewleftHalfSteps,
                                        NewrightHalfSteps,
                                        Speed)
                    +
                    IOperator.close ? ()
                     )
                     →
                     IOperator.moveOk ! (Success);

    INVOKEMOVEMENT= (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);

    EMERGENCY= IEmergency.stop ! ();
End_Subprocesses;

Protocol
    CRELAYSTATION = begin.RELAY;
    RELAY = (REQUESTMOVEMENT.stop ? ()
              →
             EMERGENCY.stop !()
             +
             (
             (REQUESTMOVEMENT.stepMoveRobot ?(BaseHalfSteps,
                                       ShoulderHalfSteps,
                                       ElbowHalfSteps, LeftHalfSteps,
                                       RightHalfSteps, HandHalfSteps,
                                       Speed)
              →
              INVOKEMOVEMENT.stepMoveRobot ! (BaseHalfSteps,
                                       ShoulderHalfSteps,
                                       ElbowHalfSteps, LeftHalfSteps,
                                       RightHalfSteps, HandHalfSteps,
                                       Speed)
          )
          ||
          (REQUESTMOVEMENT.cinematicsMoveRobot ? (x, y, z, p, r, i, Speed)
           →
           INVOKEMOVEMENT.cinematicsMoveRobot ! (x, y, z, p, r, i, Speed)
           )
          ||
          (REQUESTMOVEMENT.moveSteps ? (Joint, NewHalfSteps, Speed)
```

```
            →
            INVOKEMOVEMENT.moveSteps ! (Joint, NewHalfSteps, Speed)
            )
           ||
           (REQUESTMOVEMENT.wristMoveJoint ? (NewleftHalfSteps,
                                              NewrightHalfSteps, Speed)
            →
            INVOKEMOVEMENT.wristMoveJoint ! (NewleftHalfSteps,
                                             NewrightHalfSteps, Speed)
           )
            ||
            (REQUESTMOVEMENT.close ? ()
             →
             INVOKEMOVEMENT.close ! ()
            )
            ).ANSWER
            +
            end;

         ANSWER= INVOKEMOVEMENT.moveOk ? (Success)
                 →
                 REQUESTMOVEMENTmoveOk ! (Success).RELAY

End_Coordination Aspect CRelayStation;
```

11.2.1.2. Aspecto de Distribución

El aspecto distribución del conector va a especificar la ubicación de la máquina en la que se encuentra, así como los servicios de movilidad para desplazarse a otra máquina en caso de que se considere oportuno.

```
Distribution Aspect DOpeRobot

   Attributes
         location: loc NOT NULL;
         locMax: loc NOT NULL;
         locMin: loc NOT NULL;

   Services
      begin(Location:loc, LocMAx:loc, LocMin:loc)
       Valutions
          [begin(Location,LocMax, LocMin)]
           location:=Location,
           locMax:=LocMax,
           locMin:=LocMin;


       checkLocation( input Location:loc, output checkC:bool)
          Valutions
              {Location>locMin & Location<locMax}
              [checkLocation(Location,checkC)]
              checkC:=true;

              {Location<locMin & Location>locMax}
               [checkLocation(Location,checkC)]
```

```
            checkC:=false;


      move( input Newlocation:loc)
         Valuations
           {NewLocation > locMin & NewLocation < locMax}
           [move(input NewLocation)]
           location = NewLocation;

      end;

   Constraints
           always{ location  > locMin & location < locMax};

   Protocols
           mobility  ≡  begin.MOBILITY2;
           MOBILITY2≡ end + checkLocation + move.MOBILITY2;

End_Distribution Aspect DOpeRobot;
```

11.2.2. Conector

Tras haber definido los aspectos necesarios para especificar el conector *ConnectMovement*, a continuación se va a definir éste.

```
Connector ConnectMovement

   Roles
      Commands: IOperator,
           Subprocess CRelayStation.REQUESTMOVEMENT;
      Robot: IRobotMotion,
           Subprocess CRelayStation.INVOKEMOVEMENT;
      Base: IEmergency,
           Subprocess CRelayStation.EMERGENCY;
      Shoulder: IEmergency,
           Subprocess CRelayStation.EMERGENCY;
      Elbow: IEmergency,
           Subprocess CRelayStation.EMERGENCY;
      Wrist: IEmergency,
           Subprocess CRelayStation.EMERGENCY;
      Tool: IEmergency,
           Subprocess CRelayStation.EMERGENCY;
   End_Roles;

   Coordination Aspect Import CRelayStation;
   Distribution Aspect Import DOpeRobot;

   Initialize
      new (Location:loc, LocMAx:loc, LocMin:loc){
                CRelayStation.begin();
                DOpeRobot.begin(Location:loc, LocMAx:loc, LocMin:loc);
              }
   End_Initialize;


   Destruction
       destroy ()
       {
        CRelayStation.end();
        DOpeRobot.end();

       }
   End_Destruction;

End_Connector ConnectMovement;
```

```
RobotCnct = new ConnectMovement (localhost, 0, 100);
```

11.2.3. Modelo Arquitectónico

Finalmente, para acabar la arquitectura completa del robot 4U4, falta definir el modelo arquitectónico del sistema, con el operario, el robot y los attachments que los unen. Dicho modelo arquitectónico se presenta a continuación:

```
Architectural_Model  TeachMover

   Variables
       VarRobotOperator: RobotOperator;
       VarCnct: ConnectMovement;
        VarEnvironment: Environment;
        VarRobot: RUC;
   End_Variables;

   Attachments
    VarCnct.Commands ←→ VarRobotOperator.Commands;
    VarCnct.EnvState ←→ VarEnvironment.State;
    VarCnct.Base ←→ VarRobotOperator.Base;
    VarCnct.Shoulder ←→ VarRobotOperator.Shoulder;
    VarCnct.Elbow ←→ VarRobotOperator.Elbow;
    VarCnct.Wrist ←→ VarRobotOperator.Wrist;
    VarCnct.Tool ←→ VarRobotOperator.Tool;
    VarCnct.Robot ←→ VarRobotOperator.Robot;
   End_Attachments;

Initialize
  New (){
       VarRobotOperator = new RobotOperator();
       VarCnct = new ConnectMovement ();
        VarEnvironment = new Environment();
        VarRobot = new RUC();
}
End_Initialize

Destruction
  destroy (){
       VarRobotOperator.destroy();
       VarCnct.destroy();
        VarEnvironment.destroy();
        VarRobot.destroy();
}
End_Destruction

End_Architectural_Model TeachMover;
```

CONFIGURACIÓN FINAL

```
 ROBOTCNCT = new ConnectMovement (localhost, 0, 100);
 OPERATOR = new RobotOperator (localhost, 0, 100);

 ROBOT4U4 = new RUC{

            TOOL = new ToolSUC(){
                       ToolAct= new WristActuator(localhost);
                       ToolSensor= new Sensor(localhost);
                       ToolCnct = new WristSUCconnector(0,localhost,0,3);

                   };
             MUC= new MUCRobotArm(){
                       BaseSUC = new SUC(){
                                    BaseActuator= new Actuator(localhost);
                                    BaseSensor= new Sensor(localhost);
                                    BaseConnector = new SUCconnector(0, 0,
                                                          localhost, 90, -90);
                                 };
                       ShoulderSUC = new SUC(){
                                    ShoulderActuator= new Actuator(localhost);
                                    ShoulderSensor= new Sensor(localhost);
                                    ShoulderConnector =
                                       new SUCconnector(0,0,localhost,144,-35);
                                 };

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

                       Wrist = new WristSUC(){
                                    WristAct= new WristActuator(localhost);
                                    WristSensor= new Sensor(localhost);
                                    WristCnct = new WristSUCconnector(0, 0, 0,
                                             0, localhost, 90, -90, 270, -270);

                                 };
                       ARMCNCT = new RobotArmCnct(localhost);
                     };
             4U4Cnct = new RobotCnct(localhost,4,15,17.8,2.25,1.25,3.5,-90º);
  }
}

End_Inizialize;
Destruction
       destroy ()
       {
        OPERATOR.destroy();
        ROBOTCNCT.destroy();
        ROBOT4U4.destroy();

       }
   End_Destruction;
```

BIBLIOGRAFÍA

[Bac00] Bachman F., Bass et al., *The Architecture Based Design Method,* Technical Report CMU/SEI-200-TR-001, Carnegie Mellon University, USA, January 2000.

[EFT02] EFTCoR Project: Friendly and Cost-Effective Technology for Coating Removal. V Programa Marco, Subprograma Growth, G3RD-CT-2002-00794. 2002

[Nor98] P. Noriega, C. Sierra, *Subastas y Sistemas Multiagente*, *Revista Iberoamericana de* Inteligencia Artificial, Number 6, pp. 68-84, 1998.

[Pas04] Juan A. Pastor, Bárbara Álvarez, Pedro Sánchez, Francisco Ortiz, *An Architectural Framework for Modeling Teleoperated Service Robots,* IEEE Transactions on Computers (pendiente de notificación)

[Per03] Perez J., Ramos I., Jaén J., Letelier P., Navarro E. (2003a); "PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures";. In proceedings of 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November, © IEEE Computer Society Press ISBN 0-7695-2015-4, pp. 59-66.

[Pos04] Poseidon, 2004, http://www.gentleware.com

[UML04] UML, 2004, http://www.uml.org/