UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Departamento de Tecnología Electrónica

Arquitectura de Referencia para Unidades de Control de Robots de Servicio Teleoperados

TESIS DOCTORAL

Francisco José Ortiz Zaragoza

Ingeniero Industrial

Directores

Dra. Bárbara Álvarez Torres

Dr. Juan Ángel Pastor Franco

2005

Título: Arquitectura de Referencia para Unidades de Control de Robots de Servicio Teleoperados

Autor: Francisco José Ortiz Zaragoza
Directores: Dra. Bárbara Álvarez Torres
Dr. Juan Ángel Pastor Franco

Conclusión: Cartagena, 3 de diciembre de 2004
Defensa: UPCT, Cartagena, 29 marzo de 2005

Tribunal: Dr. José María Fernández Meroño
Dr. Juan Antonio de la Puente Alfaro
Dr. Carlos Balaguer Bernaldo de Quirós
Dr. Pablo González de Santos
Dr. Alejandro Alonso Muñoz

Calificación: Sobresaliente Cum Laude

Agradecimientos

Necesitaría varias páginas para expresar todo el agradecimiento que siento por Juan Ángel Pastor. Tengo muy claro que esta tesis no la habría podido realizar sin su constante ayuda, apoyo, alegría contagiosa y paciencia infinita, sin sus consejos y también los quebraderos de cabeza provocados al hacerme ver las dificultades o incluso incongruencias de algunas de mis propuestas; del mismo modo ha sabido comprender mis puntos de vista y soluciones a sus preguntas para que al final, después de muchas discusiones, tener la amplitud de miras suficiente como para llegar a un acuerdo. Por todo ello, y después de incontables horas de trabajo y diálogo, ACROSET es tan suya como mía.

La otra persona a quien debo la realización de esta tesis es Bárbara Álvarez. Al igual que Juan Ángel ha impulsado mi espíritu crítico, Bárbara me ha puesto los pies en la tierra gracias a su visión práctica del conocimiento y a su capacidad de concreción y síntesis. Le debo, entre muchas cosas, la génesis de la propia tesis; ella me cedió el testigo para seguir investigando con su mismo interés en la teleoperación de robots. Me ha guiado enfocando mi trabajo hacia los aspectos más útiles, innovadores e interesantes de nuestro campo de investigación común.

Además de todo ello, la calidad como personas de mis dos directores de tesis, su amistad, buen humor y cariño han hecho posible crear un ambiente de trabajo y de relación personal inmejorable. Todo ello me hace sentir que he tenido los mejores directores de tesis que jamás podría haber deseado. Gracias por todo.

Siendo persona tan agradecida, es muy difícil incluir en pocas líneas a todas las personas que de una forma u otra, me han ayudado a realizar esta tesis. A Andrés Iborra, director de mi Departamento de Tecnología Electrónica, le debo la confianza puesta en mí, su habilidad para saber impulsar al máximo mi carrera profesional en la Universidad y su continuo apoyo e incansable trabajo y esfuerzo por conseguir una investigación de calidad. Junto a él, debo agradecer también a José María Fernández Meroño y a Roque Torres que me animaran a empezar mi trabajo en la Universidad, primero como becario y luego como profesor ayudante; con ellos empecé a vivir la investigación como algo enriquecedor.

Desde esos comienzos, he tenido la suerte de rodearme de unos compañeros de trabajo excelentes, tanto en mi labor docente, con todos los miembros del Departamento de Tecnología Electrónica, como en mi labor investigadora, con todos los integrantes del DSIE, cuya profesionalidad y compañerismo han influido sin duda en mi crecimiento profesional. De entre ellos, debo mencionar especialmente a aquellos que me han enriquecido tanto en lo laboral como en lo personal gracias a su amistad y colaboración desinteresada: desde el principio, a Pedro Navarro, por su amistad, compañía, buen humor y espíritu de trabajo, a Alejandro Martínez, por un tanto de lo mismo, a Conchi Jiménez por su compañerismo y apoyo sinceros, a Juan Suardíaz, por su generosidad sin límites. También desde el comienzo han estado conmigo los compañeros y amigos que han trabajado en el DSIE y han seguido después su vida laboral por otros caminos, en especial Maria José, por estar ahí siempre, incluso cuando no está cerca…, Alfonso, Gabi, Javier.

Debo también dar las gracias a Pedro Sánchez, que me guió por los rigurosos mundos del UML y los métodos de desarrollo más o menos formales. A los llegados más recientemente: Diego Alonso, al que le debo, no sólo su apoyo y puntos de vista prácticos en las últimas instanciaciones de ACROSET, sino también su trabajo codo a codo conmigo que espero continúe en lo que, en un futuro, sea su propia tesis. Fernando Losilla, su concienzudo trabajo en el EFTCoR me ha ayudado a probar que ACROSET puede ser una realidad, tan grande como su capacidad de trabajo es su calidad como persona. Todos los compañeros de trabajo en el GOYA y en el EFTCoR, por la UPCT: Carlos Fernández, Esther de Jódar y los compañeros ya mencionados, en IZAR Carenas: Noelia Ortega, Jesús Carrión, David Rodríguez, Raúl Borraz y el resto de los integrantes del proyecto. Las personas cuyo Proyecto Fin de Carrera he dirigido y cuyo trabajo ha aportado su grano de arena a la tesis y a otros trabajos de investigación: Pedro Joaquín, Jorge, los dos Pacos, Manolo y Pablo, Javi, Javier, José María, Ramón. En la recta final: Ángel y Juan Álvaro por sus consejos de encuadernación, Ana Belén por su revisión lingüística, Lola y Luis Pedro del SAIT por su ayuda en la impresión y por su simpatía. Fuera del trabajo, también debo agradecer el apoyo puntual o continuo de aquellos que han sido y siguen siendo mis amigos y amigas…y a todos aquellos que me he dejado en el tintero, pero que también tienen mi agradecimiento sincero.

A nivel personal, cómo no agradecer la comprensión, cariño y amor de mis padres. También a mis hermanos, mis sobrinos y el resto de mi familia.

… y por último, pero en primer lugar, a mi Lali, por ella, por todo.

A la mujer de mi vida

"… nunca sabré qué espero de él
ni qué conjuro deja en mis tobillos,
pero cuando estos ojos se hartan de baldosas
y esperan entre el llano y las colinas
o en calles que se cierran en más calles,
entonces sí me siento un náufrago
y sólo el mar puede salvarme."

M. Benedetti

Resumen

Las unidades de control de robots son sistemas intensivos en *software*, por lo que el diseño del mismo es esencial para el desarrollo y evolución del sistema global. El *hardware* es también un elemento primordial en estos sistemas, no sólo por su papel como plataforma de ejecución del *software* de control, sino porque es habitual encontrar componentes COTS o diseñados especialmente para la aplicación, que realizan parte de la funcionalidad del sistema. Por ello, *hardware* y *software* se deben poder integrar adecuadamente y combinar de distintas formas de acuerdo con el diseño del sistema.

Para manejar la complejidad intrínseca de estos sistemas, se hace necesario plantear modelos de desarrollo rigurosos y arquitecturas bien definidas que se puedan reutilizar en otros productos. Disponer de tales arquitecturas facilita enormemente el desarrollo de nuevas aplicaciones fiables y de calidad, pues permite, por un lado la reutilización de modelos y componentes y por otro, ofrece un marco para el desarrollo de los mismos. De este modo se pueden reducir costes dado que se reduce el tiempo de desarrollo y se reutilizan componentes ya probados.

En esta tesis se propone ACROSET como una arquitectura de referencia para el dominio de las unidades de control de robots de servicio teleoperados que define los principales subsistemas que deben o pueden aparecer en cualquier arquitectura concreta, sus responsabilidades y relaciones. En ACROSET se propone un modelo de componentes conceptual en el que se definen los componentes que pueden aparecer en cualquier sistema del dominio considerado y los patrones de interacción entre componentes al mismo nivel que éstos, gracias al uso de puertos y conectores.

Una de las principales características de ACROSET es su flexibilidad tanto para la definición de arquitecturas de sistemas concretos como para la evolución de las mismas. Precisamente el éxito de la arquitectura residirá en su habilidad para adaptarse a la variabilidad entre los sistemas del dominio para el que ha sido definida. Para el nivel de abstracción que exige el manejo de tal variabilidad, las metodologías de desarrollo dirigidas por casos de uso no son apropiadas, por lo que se adopta una metodología orientada específicamente hacia el diseño de arquitecturas como el ABD (*Architecture Based Design Method)*, que parte de requisitos funcionales y de calidad lo suficientemente amplios y abstractos como para abarcar todo el dominio. Dicha metodología se completa con el modelo de 4 vistas propuesto por Hofmeister con el fin de expresar el modelo conceptual en UML y tratar a componentes y conectores como entidades de primera clase.

Se ha validado ACROSET con su instanciación para las unidades de control de diversos robots de limpieza de cascos de buques. En concreto, en esta tesis se presenta su instanciación para el prototipo GOYA y la familia de robots del proyecto EFTCoR. También se ha utilizado un robot didáctico 4U4 para probar diferentes combinaciones *hardware/software* a partir de ACROSET.

Abstract

Robots control units are *software* intensive systems, so that the design of that *software* is essential for the development and evolution of the overall system. *Hardware* is also a primordial element in these systems, not only for its role as an execution platform for the control *software*, but because it is habitual to find COTS components or especially designed components that carry out part of the functionality of the system. For that reason, *hardware* and *software* should can be appropriately integrated and combined in different ways according to the design of the system.

To deal with the intrinsic complexity of these systems is necessary to use rigorous development methods and architectural frameworks and tools that embody well defined concepts to enable effective realization of systems to meet high level goals. Such methods and architectural frameworks allow rapid development of systems and reuse of a large variety of components, with concomitant savings in time and money.

In this thesis ACROSET is proposed as reference architecture for the domain of control units for teleoperated service robots. This architecture defines the main subsystems that should or could appear in any concrete architecture, their responsibilities and relationships. ACROSET proposes a conceptual framework to define the components and their interaction patterns that could appear in any system of the considered domain. This architectural framework relies on the abstract concepts of component, port and connector.

One of the main characteristics of ACROSET is its flexibility as much for the definition of architectures for concrete systems as for the evolution of the same ones. The success of the architecture depends on its ability to deal with the variability among the systems of the considered domain. Furthermore, at the level of abstraction required to manage the variability of the systems, the development methodologies directed by use cases are not appropriate. For this reason, an architecture oriented approach is adopted: the ABD (*Architecture Based Design Method)*, which starts from functional and quality requirements wide and abstract enough to include the entire domain. Such method is completed with the 4 views proposed by Hofmeister to express the conceptual view of the architecture with UML and to treat components and connectors as first class entities.

ACROSET has been validated with different instantiations for the control units of various ship hull cleaning robots. Specifically, in this thesis the instantiation of ACROSET for the GOYA prototype and the family of robots in the EFTCoR project is presented. A didactic robot (4U4) has also been used to test different *hardware/software* combinations.

Contenidos

Lista de Figuras

CAPÍTULO 1

CAPÍTULO 2

CAPÍTULO 3

CAPÍTULO 4

CAPÍTULO 5

CAPÍTULO 6

CAPÍTULO 7

ANEXOS

Lista de Tablas

Capítulo 1

Planteamiento y Objetivos de la Tesis

1.1. Introducción

A medida que el tamaño y la complejidad de un sistema *intensivo en software* aumentan, la especificación, análisis y diseño de su estructura juegan un papel cada vez más importante. Según la definición ofrecida en [IEEE Std-1471-2000], un sistema *intensivo en software*, es aquél donde el *software* influye y contribuye de una manera esencial al diseño, construcción, desarrollo y evolución del sistema global. Los sistemas **robóticos** (o los sistemas **mecatrónicos** en general – sistemas compuestos de una parte mecánica, unos accionamientos eléctricos, hidráulicos, etc., una electrónica y un *software*), son sistemas en los que el *software* es una parte fundamental, pero no la única ni, a menudo, la más importante. Son sistemas compuestos por un *hardware* que es gobernado por un determinado *software* o una combinación de ambos. El *software* es una parte más que debe integrarse en el sistema global. Ignorar este hecho sería catastrófico, pues se originarían aplicaciones que no se ejecutarían ni podrían integrarse correctamente con el *hardware* seleccionado.

Los sistemas **robóticos** difieren de otros tipos de sistemas intensivos en *software* en muchos aspectos. Uno de los principales es la necesidad de alcanzar objetivos complejos y de alto nivel; estos sistemas deben desarrollar su actividad en un entorno estructurado, en el caso más favorable, o en entornos no-estructurados, complejos y dinámicos (típico ambiente de trabajo de los robots teleoperados), donde el sistema debe asegurar su propio comportamiento dinámico, incluso reaccionando ante cambios inesperados en dicho entorno. Muchos de estos robots incorporan sistemas empotrados en tiempo-real, son **sistemas críticos,** y en bastantes ocasiones tienen requisitos de seguridad elevados. Como sistemas críticos, en el caso de no cumplir sus tiempos de respuesta (*deadlines*), el funcionamiento del sistema no será correcto, y esto, dependiendo de la aplicación para la que esté diseñado, puede tener consecuencias graves para el entorno en el que el robot desarrolla su labor e incluso para la seguridad de las personas. Por todo ello, la seguridad del *software*, su correcta integración con el hardware, su rendimiento, tolerancia a fallos, etc., tienen una importancia relevante.

Normalmente, estos sistemas son demasiado complejos para ser desarrollados utilizando técnicas convencionales de programación. Para manejar esta complejidad, se hace necesario plantear modelos

de desarrollo rigurosos y **arquitecturas** bien definidas que permitan ser implementadas en estos sistemas complejos. También se busca que dichas arquitecturas se puedan **reutilizar** en otros sistemas, y que sean fácilmente modificables atendiendo a nuevos objetivos o requisitos, pudiendo además ser analizadas para comprobar que cumplen dichos requisitos, bien sean temporales o de otra índole.

En el artículo *"Architecture, the Backbone of Robotic Systems"* [Coste00], se considera a las arquitecturas de referencia como la *columna vertebral* de un sistema robótico, siendo su selección o diseño una de las decisiones más importantes a la hora de abordar el diseño de tal sistema, puesto que su correcta elección puede facilitar la especificación, implementación y validación del sistema robótico en cuestión. De forma general, se puede definir una arquitectura con una *descripción de qué clase de elementos componen dicha arquitectura (software, lenguajes, modelos de ejecución, modelos de control, modelos de comunicación, hardware, maquinaria, etc.), cómo están conectados (lógica y físicamente), y cómo interactúan* [Kramer93].

El "*peso"* del componente *software* en los sistemas robóticos depende del tipo de unidad de control que se emplee, por ejemplo, un PC industrial, un PC empotrado, un PLC (*Programmable Logic Controller*), un controlador basado en una FPGA (*Field Programmable Gate Array*) ó un ASIC (*Application Specific Integrated Circuit*). De aquí, la importancia de plantear una arquitectura para este tipo de sistemas que posibilite la **integración del *software* y el *hardware*** que componen el sistema, haciendo posible además, construir **distintas implementaciones según sean las necesidades**: desde una configuración puramente *software* a una combinación *hardware-software*, o en un caso extremo, a una solución puramente *hardware*.

Por lo tanto, el diseño de este tipo de sistemas requiere conocer y seguir disciplinadamente una serie de reglas bien definidas, que indiquen cómo ensamblar y conectar los diferentes elementos del sistema y cómo gestionar el desarrollo y evolución del mismo. Sin dichas reglas, el diseño se convierte en un proceso caótico, que puede agravarse si el sistema debe estar sujeto a modificaciones a lo largo de su ciclo de vida. Así pues, además de definir una arquitectura, se debería proponer una notación formal o semi-formal para describirla y una metodología sistemática de desarrollo que permita en el futuro reutilizar la arquitectura minimizando los esfuerzos necesarios para ello, aunque la configuración *hardware-software* sea distinta. De esta forma se conseguiría reducir los costes y los plazos de desarrollo del sistema.

1.2. Antecedentes, definiciones y motivación

El trabajo de tesis que aquí se expone se enmarca dentro de los trabajos del grupo de investigación *División de Sistemas e Ingeniería Electrónica* - DSIE y el *Departamento de Tecnología Electrónica* – DTE, de la *Universidad Politécnica de Cartagena* - UPCT.

El DSIE ha acumulado una considerable experiencia durante estos últimos años en el desarrollo de diferentes sistemas teleoperados. En estos sistemas, un operador está encargado de monitorizar y operar el robot de acuerdo a la información que le provee el sistema de teleoperación (**Fig. 1.1**). Para ello, el sistema de teleoperación recibe comandos del operador y se comunica con la unidad de control, que a su vez realiza las acciones necesarias sobre los elementos actuadores o accionadores de los motores, supervisando mediante realimentación el correcto funcionamiento de los mismos. Un sistema sensorial captura la información referente a la posición, velocidad, etc. La unidad de control del robot envía el estado actual del mismo al sistema de teleoperación, que será mostrado al operador. La frontera entre las tareas que realiza la unidad de control y la estación de teleoperación es variable, así como los modos de control que podemos encontrar; entonces, según sea el grado de actuación del operador humano y su relación con el sistema de control podrá diferenciarse entre control teleoperado, supervisado y compartido [Granot01]. Aunque en el Capítulo 4 se profundizará más en la explicación de estos conceptos es conveniente presentar ya algunas definiciones propias del dominio de estudio que serán utilizadas a lo largo de toda la tesis.

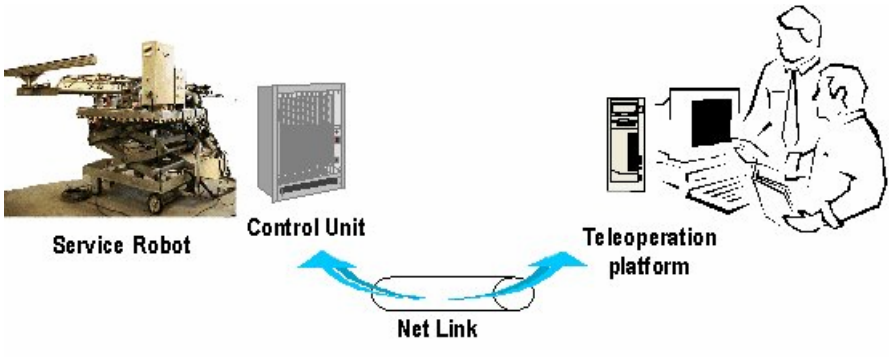


Fig. 1.1.- Esquema general de un sistema de teleoperación

1.2.1. Definiciones

En primer lugar, hay que diferenciar entre manipulador y robot:

- **Manipulador**: mecanismo formado generalmente por elementos en serie o en paralelo, articulados entre sí, destinado al agarre y desplazamiento de objetos. Es multifuncional y puede ser gobernado directamente por un operador humano o por un dispositivo lógico.
- **Robot** (ISO)**:** manipulador multifuncional reprogramable con varios grados de libertad, capaz de manipular materias, piezas, herramientas o dispositivos especiales según trayectorias variables programadas para realizar tareas diversas.

Una vez hecha la distinción, se pueden definir los siguientes términos:

- **Teleoperación:** conjunto de tecnologías que comprenden la operación o gobierno a distancia de un dispositivo por un ser humano. Por tanto, teleoperar es la acción que realiza un ser humano de operar o gobernar a distancia un dispositivo; mientras que un sistema de teleoperación será aquel que permita teleoperar un dispositivo, que se denominará dispositivo teleoperado.
- **Telemanipulación:** conjunto de tecnologías que comprenden la operación o gobierno a distancia por un ser humano de un manipulador. Por tanto, telemanipular es la acción que realiza un ser humano de operar o gobernar a distancia un manipulador; mientras que un sistema de telemanipulación será aquel que permita teleoperar un manipulador, que se denominará manipulador teleoperado.

Se puede usar el término teleoperación o telemanipulación de forma indistinta para hacer referencia a la teleoperación de un manipulador. Si además el manipulador que está siendo teleoperado posee las características de un robot, en el sentido de que posee un grado de reprogramación y de adaptación autónoma al entorno, de forma que el operador humano únicamente tenga que comunicarse con él de forma intermitente, monitorizando su funcionamiento y cambiando objetivos o planes, se hablará de tele-robótica.

- **Tele-robótica:** conjunto de tecnologías que comprenden la monitorización y reprogramación a distancia de un robot por un ser humano. Se hablará entonces de la teleoperación de un robot, que se denominará tele-robot o robot teleoperado.

Otro grupo de conceptos relacionados con la teleoperación son aquellos que tienen que ver con la realimentación sensorial al operador. Debido a que el entorno de trabajo está alejado del lugar donde se encuentra el operador, éste ha de recibir algún tipo de información que le permita seguir la evolución de la tarea, es decir, es necesario que el operador se sienta de alguna manera presente en el lugar donde se desarrolla la misma. Así, surge el concepto de telepresencia como objetivo primario

que se quiere conseguir y sobre el que se pueden encontrar algunas reflexiones en [Held92] y [Colon99].

- **Telepresencia:** situación o circunstancia que se da cuando un ser humano tiene la sensación de encontrarse físicamente en el lugar remoto. La telepresencia se consigue realimentando coherentemente al ser humano suficiente cantidad de información sobre el entorno remoto.

En el caso de la telemanipulación, la manera más básica de conseguir algo de telepresencia es contar con un número adecuado de cámaras en la zona remota. Si, además, se puede realimentar al operador con las fuerzas de contacto con el entorno o con los sonidos que se producen durante el transcurso de la tarea, se tendrá un mayor grado de telepresencia. Como se ve, existen distintos grados de telepresencia, aunque la telepresencia absoluta se podría decir que hoy por hoy es imposible.

Hay otra serie de términos relacionados con la telepresencia, pero con sus matices particulares, que se utilizan y son por sí mismos, materia de estudio para otras tesis doctorales ([Ferre97-td], [Arcara02-td]), como son **realidad virtual**, **realidad aumentada**, **propiocepción** y **realimentación háptica**.

1.2.2. Antecedentes

Aplicando el esquema general de control teleoperado, el grupo de investigación DSIE ha participado en el desarrollo de diferentes sistemas: Aplicaciones Automáticas [AAA95], TRON [TRON96] y GOYA [GOYA98a]. La realización de estos proyectos ha sido posible gracias a la financiación del Ministerio de Ciencia y Tecnología con fondos de la Unión Europea (MINER en los primeros y FEDER en el caso de GOYA). La realización de los mismos, ha permitido comprobar el interés de desarrollar *software* de calidad que facilite la modificación y mantenimiento de los diferentes sistemas. En todos estos sistemas teleoperados se ha trabajado tomando como base una arquitectura de referencia para plataformas de teleoperación que fue desarrollada en el primero de los proyectos, en el año 96 y que ha sido difundida en diversas publicaciones de prestigio, [Álvarez98], [Iborra00], [Álvarez00a], [Álvarez00b], [Álvarez01a]. En concreto, la implementación realizada en los siguientes proyectos, destinados a servicios de mantenimiento en centrales nucleares, permitió validar la arquitectura de teleoperación:

- Desarrollo de una implementación para el sistema ROSA (*Remotely Operated Service Arm*) [AAA95]. La arquitectura [Álvarez97-td] fue reutilizada para el control de cada una de las herramientas para operaciones de mantenimiento de la placa de tubos de los generadores de vapor de una central nuclear.
- Implementación de la arquitectura para el sistema IRV (*Inspection Retrieving Vehicle*) [Pastor98], [AAA95], que permitió validar la arquitectura para la teleoperación de un vehículo dedicado a la recogida de objetos caídos desde la caja de aguas a la tobera del primario.
- Implementación de la arquitectura para el sistema TRON (*Teleoperated and Robotized System for Maintenance Operation in Nuclear Power Plants Vessels*) [TRON96], destinado a la recuperación de objetos en el fondo de la vasija del reactor. Permitió el traslado de la arquitectura a otras plataformas y la posibilidad de reutilizar los componentes genéricos desarrollados en Ada 83 para el sistema ROSA.

Las experiencias mencionadas anteriormente han permitido constatar las ventajas de utilizar una arquitectura de referencia para un dominio concreto (en su momento "plataformas remotas de teleoperación"). Dicha arquitectura [Álvarez97-td], fue implementada y reutilizada con un esfuerzo relativamente bajo. Sin embargo, en todos los casos la ***Unidad de Control Local*** (*Control Unit* en **Fig. 1.1**) fue desarrollada a medida de la aplicación, y el coste de desarrollo fue muy elevado. Esto llevó a pensar en el interés de seguir profundizando en esta línea para conseguir una **arquitectura de referencia para la unidad de control** específica para el dominio **de robots de servicio teleoperados**.

El hecho de centrarse en la implementación del *software* de la estación de teleoperación en los sistemas mencionados hasta ahora, no ha hecho necesario el estudio detallado de la integración del *software* con el *hardware* del sistema o la inclusión del *hardware* en la propia arquitectura, puesto que

en el caso de la estación de teleoperación, la implementación de la arquitectura de referencia solía tratarse de un conjunto de procesos *software* ejecutándose de forma concurrente en una determinada plataforma (PC o Estación de Trabajo).

Sin embargo, en el caso de la **unidad de control**, el problema es mucho más complejo, puesto que normalmente intervendrá un *software* muy integrado y dependiente de un determinado *hardware* (bien sea comercial o diseñado especialmente para el sistema en cuestión), y además, habrá que tener en cuenta consideraciones de tiempo real, concurrencia, etc., y habrá que diseñar interfaces bien definidos entre los distintos componentes del sistema, protocolos de comunicación, etc.

En cuanto a los antecedentes más próximos y el actual **marco de trabajo**, hay que destacar que diversos proyectos motivan y proporcionan las bases para la realización de esta Tesis Doctoral:

- **GOYA**. *"Robot escalador para la limpieza de cascos de buques, respetuoso con el medio ambiente"*. Se obtuvo un prototipo de plataforma para la limpieza de grandes superficies de un barco en dique seco de forma teleoperada [Iborra01]. Proyecto FEDER [GOYA98a] finalizado en diciembre de 2001 (**Fig. 1.2**). En este proyecto se iniciaron los estudios para proponer una arquitectura de referencia para el desarrollo de la unidad de control local que, en este caso, fue implementada en un PC industrial [Álvarez02], [Ortiz02b].

En la actualidad, el DSIE está desarrollando, entre otros, el proyecto europeo:

- **EFTCoR**. *"Environmental friendly and cost-effective technology for coating removal"*. Proyecto GROWTH del *V Programa Marco* de la Unión Europea [EFTCOR02] que tiene como objetivo desarrollar un sistema de limpieza automatizada de la superficie de un buque, así como el sistema de recogida y reciclaje para asegurar una tecnología respetuosa con el medio ambiente.

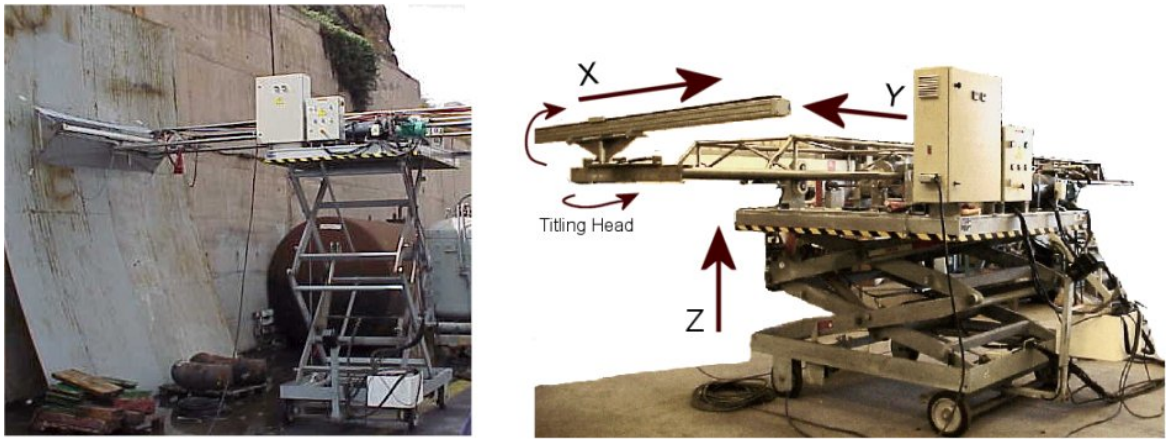


Fig. 1.2.- Prototipo de robot de limpieza para superficies de buques GOYA

En el marco de estos proyectos se ha validado la arquitectura propuesta en esta Tesis (el prototipo GOYA ha finalizado y el proyecto EFTCoR está todavía en desarrollo), implementando el sistema de control de los distintos sistemas de limpieza de superficies. El proyecto EFTCoR se gesta como continuación del proyecto GOYA, que obtuvo como resultado un primer prototipo de robot de limpieza. Con este proyecto se persigue diseñar un sistema integral de limpieza de las superficies verticales y *finos* de buques, teniendo en cuenta las dificultades especiales que entraña el caso particular del *spotting* (limpieza de pequeñas áreas aisladas del barco), y el reciclado de los residuos producidos, lo que conduce al diseño de una familia de robots especializados en los distintos entornos operativos.

También hay que destacar que los principales objetivos marcados en esta Tesis Doctoral están avalados por la concesión de un proyecto de investigación financiado por la Fundación Séneca de la Comunidad Autónoma de la Región de Murcia:

- **ROVA**. *"Arquitectura de Referencia para Unidades de Control de Sistemas Teleoperados"*. Ayudas para la realización de proyectos de investigación de la CARM [ROVA01].

En relación con este tema, la convocatoria de "Ayudas para la realización de proyectos de investigación", dentro de las acciones financiadas por la Consejería de Ciencia, Tecnología, Industria y Comercio del Programa Séneca 2002, ha permitido la evaluación de la arquitectura de referencia empleada para la plataforma de teleoperación y su rediseño en base a un modelo de componentes [Seneca02].

Además, el grupo de investigación DSIE cuenta también con financiación nacional del Ministerio de Ciencia y Tecnología para el desarrollo de los proyectos:

- **EFTCOR**. *"Environmental Friendly And Cost-Effective Technology For Coating Removal"*. Convocatoria de acciones especiales del Ministerio de Ciencia y Tecnología. Ref. DPI2002-11583-E [EFTCOR03].

- **ANCLA**. "Arquitecturas dinámicas para sistemas de teleoperación". TIC2003-07804-C05-02. [ANCLA03].

Finalmente, con el fin de divulgar la labor investigadora realizada en el marco de los proyectos mencionados anteriormente (GOYA, EFTCoR y ANCLA), y en concreto, en el desarrollo de una arquitectura de referencia para unidades de control de sistemas teleoperados, se ha publicado en varias revistas reconocidas internacionalmente y se han realizado diversas ponencias en congresos nacionales e internacionales, como se puede ver en la sección de divulgación de resultados en el Capítulo 8 y en la bibliografía: [Ortiz00], [Molina01], [Iborra01], [Ortiz02a], [Ortiz02b], [Álvarez02], [Ortiz03].

1.2.3. Importancia de los robots de servicio en la industria

La diferencia fundamental entre un robot industrial y un robot de servicio es que este último está específicamente diseñado para realizar tareas muy concretas de mantenimiento, reparación e inspección de instalaciones, sistemas o componentes (inspeccionar y limpiar conducciones de aire acondicionado, inspeccionar y reparar componentes principales de una planta industrial, etc.), mientras que el diseño de los robots industriales se orienta a realizar tareas muy repetitivas, normalmente en entornos estructurados.

Desde el punto de vista físico ambos están conformados por una serie de elementos interconectados entre sí y que componen su anatomía. Estos elementos van desde su unidad de control hasta sus elementos terminales (pinzas o herramientas). Entre ambos extremos están los accionadores, actuadores, sistema de transmisión y la propia estructura mecánica del robot. Sin embargo, existen algunas diferencias. En el caso de los robots de servicios, el diseño mecánico global del mismo está orientado a la aplicación final. Mientras que en los robots industriales, se parte de un robot de una tipología más o menos general (robot cartesiano, polar, antropomórfico, *scara*, etc.) y es la herramienta final la que se adapta al proceso.

En la mayor parte de las ocasiones, los robots industriales son programados una sola vez para realizar una tarea repetitiva. En el caso de los robots de servicios la situación es muy diferente; los entornos no suelen estar claramente determinados y, en muchas ocasiones, pueden estar cambiando durante el servicio en cuestión, debiendo realizar, en algunas ocasiones, distintos tipos de tareas. Debido a estas condiciones de operación, los robots de servicio suelen ser sistemas **teleoperados** con un determinado grado de automatización y diseñados específicamente para la aplicación.

Los sistemas de teleoperación

Desde los primeros desarrollos de la teleoperación, la industria nuclear ha sido el principal consumidor de dichos sistemas, debido a la peligrosidad intrínseca que llevan los trabajos de mantenimiento en centrales nucleares [Larcombe84]. Sin embargo, con el paso de los años se fue abriendo su

aplicabilidad a otros sectores, especialmente relacionados con las industrias de servicio, por ejemplo, el de la reparación, limpieza y mantenimiento de las industrias petroquímicas, eléctricas y navales.

La importancia de la teleoperación en estas industrias radica en que hay muchas operaciones peligrosas, con condiciones de trabajo bastante duras, que son realizadas por operadores humanos y que son susceptibles de ser ejecutadas por un robot que elimine el riesgo para el operador. Dichas operaciones suelen realizarse en ambientes cambiantes, no estructurados, que hacen muy difícil el funcionamiento de un robot autónomo. Por ello es idóneo el empleo de la teleoperación en esta clase de entornos.

En [Sheridan92], [EURON01] y [EURON04] se puede encontrar una descripción muy pormenorizada de la mayoría de estas aplicaciones, en concreto, en [EURON01] y [EURON04] se presentan el mercado europeo y las oportunidades de la robótica, en general y la robótica de servicio, en particular, en este mercado, como se describe en el siguiente apartado. En cuanto a las futuras aplicaciones, éstas irán en aumento a medida que las tecnologías avancen y se vaya investigando en la mejora de cada uno de los elementos que componen un sistema de teleoperación. El futuro pasa por alcanzar una madurez en las tecnologías, que en los últimos años, se están incorporando dentro del campo de la teleoperación.

Por todo ello, el estudio y desarrollo de métodos genéricos que faciliten el avance de dichas tecnologías es de enorme interés para la industria. No sólo se trata de avanzar en tecnologías propias de fabricación de los distintos componentes de un sistema teleoperado (manipuladores, herramientas, sistemas de transporte, etc.), sino que es necesario introducir nuevas metodologías para el desarrollo de *software* de calidad y de integración con el *hardware*, dado que a medida que el tamaño y la complejidad de un sistema *software* aumenta, el diseño y la especificación de su estructura juega un papel más importante.

Oportunidades de la robótica en el mercado europeo

La producción de robots, sobre todo para la industria, ha ido aumentando de forma considerable en los últimos años, como se puede observar en la **Fig. 1.3**, donde destaca el hecho de que Europa no ha parado en su ritmo de producción ascendente, llegando a superar a Japón. Entre los países europeos, en el 2003 España ocupa el quinto lugar en número de robots industriales por cada 10.000 trabajadores, prácticamente empatada con Francia (**Fig. 1.4**). A pesar de que la industria automovilística sigue siendo la principal consumidora de robots para incorporarlos en su sistema productivo (**Fig. 1.5**), la importancia de los robots de servicio en la industria europea ha seguido también un ritmo ascendente; su mayor influencia en el futuro está respaldada por un informe de la Red Europea de Investigación Robótica EURON. Este organismo financiado por la Unión Europea conjunta las principales empresas, universidades, y organismos públicos dedicados a la robótica en Europa [EURON01]. El DSIE es miembro de este organismo, y es representado por el autor de esta Tesis Doctoral, tanto en su primera versión, como en la constitución de EURON II, como red de excelencia en el VI Programa Marco.

En este informe, se expone que las industrias de servicio (limpieza, mantenimiento, vigilancia, salud, etc.) dependen cada vez más de ayuda inteligente para mejorar las condiciones de trabajo, calidad y costes. No sólo la industria sino también la ayuda en el ámbito personal como futuro inmediato. Para mantener el nivel de desarrollo económico actual o mejorarlo se necesitan métodos de producción más eficientes. Por tanto, se observa el dominio de los robots de servicio como una línea de trabajo con futuro y se marca como una de las vías de expansión y desarrollo de la industria robótica europea. Así, se preveía que de los 6.600 robots de servicio en el mercado en el año 1999 se pasaría a unos 50.000 en el año 2004.

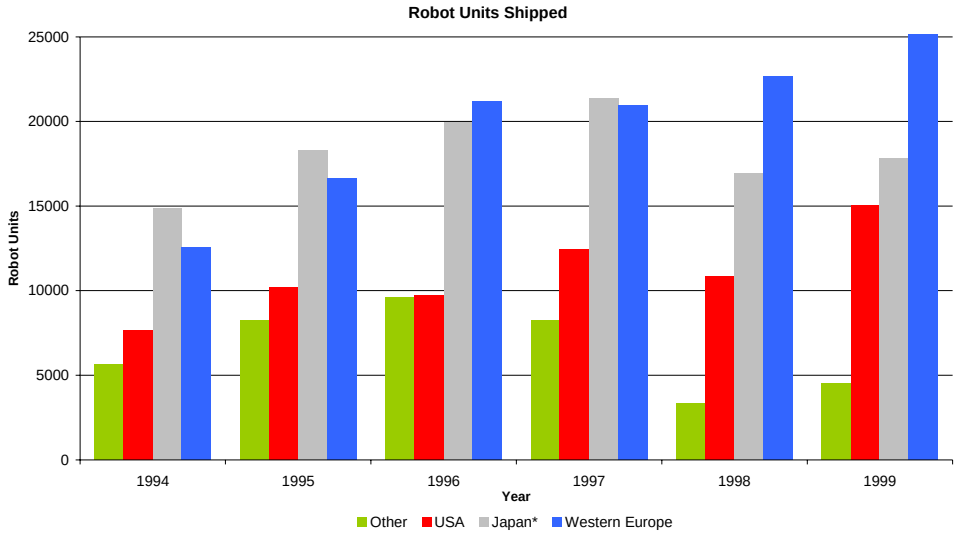


Fig. 1.3.- Evolución en número de unidades de robots producidas por año desde 1994 hasta 1999 [EURON01]

	2003
Japón a/	322
Rep. de Corea b/	138
Estados Unidos	**63**
Unión Europea	**93**
Alemania	148
Italia	116
Suecia	99
Finlandia	78
España	72
Francia	71
Austria	54
Benelux	53
Dinamarca	50
Reino Unido	39
Australia	36
Noruega	24
Portugal	15
Rep. Checa	12

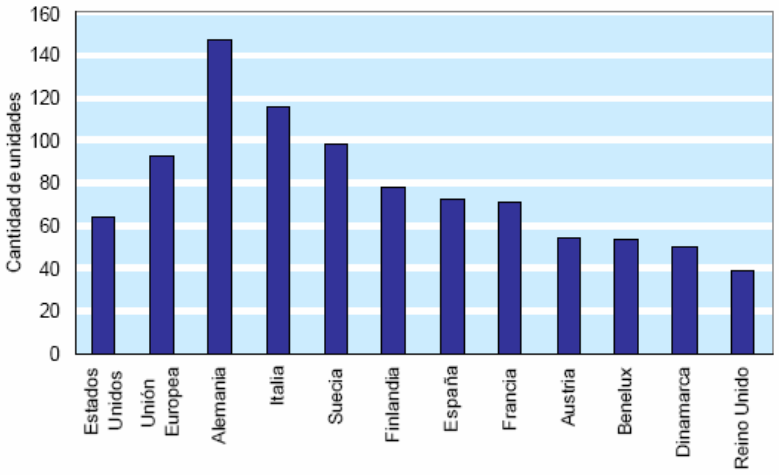


Fig. 1.4.- Número de robots por 10.000 personas empleadas en la industria en el año 2003 [ONU04]

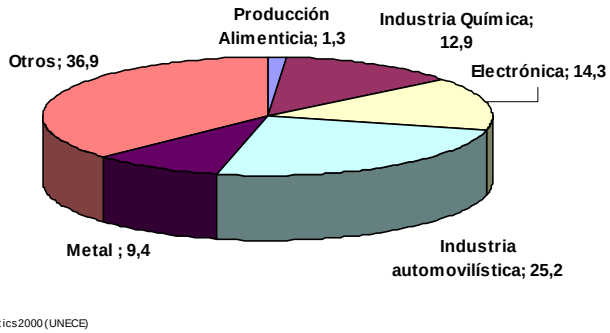


Fig. 1.5. Uso de robots industriales en diferentes sectores del mercado en el año 2000

Aunque los robots de servicio pueden ser tan diversos como las aplicaciones para las que están diseñados, se pueden distinguir tres categorías:

- Robots para servicios de limpieza, inspección, seguridad, etc, que surgen de la especialización de máquinas existentes o la creación de sistemas mecatrónicos especializados. (**Fig. 1.5-a**)
- Robots especialmente diseñados para mercados de gran volumen, como por ejemplo robots para entretenimiento, para tareas caseras (cortador de cesped, aspirador "inteligente", etc) (**Fig. 1.5-b**)
- Robots clásicos modificados para trabajar fuera del entorno de la fabricación, como por ejemplo, robots para medicina, robots para automatización de servicios, como llenado de gasolina, almacenes automatizados, expendedores de video automáticos, etc. (**Fig. 1.5-c**)

a) Robots de servicio para mercados de poco volumen de mercado: limpieza de suelos, inspección de alcantarillas, ocio y entretenimiento

b) Robots de servicio para aplicaciones domésticas con un gran volumen de mercado: cortacésped, aspirador y recolector de pelotas de tenis

c) Robots de servicio basados en brazos robots convencionales: repostaje, cirugía ortopédica, llenado de bebidas automático

Fig. 1.6. Aplicaciones de los robots de servicio

En la mayoría de los casos, las empresas deben adoptar la robótica para satisfacer los requisitos de la aplicación concernientes a funcionalidad, reducción de costes y aseguramiento de la calidad en la ejecución de tareas. Para conseguir un sistema fiable y competitivo hay que tener en cuenta los siguientes factores:

- ***Costes de desarrollo y del sistema***. Al contrario de lo que sucede con los robots industriales, los robots de servicio están diseñados especialmente para la aplicación, normalmente para realizar tareas específicas, que generalmente tienen lugar en sistemas no estructurados, con la posibilidad de interacción directa hombre máquina. Se pueden minimizar los costes de desarrollo usando componentes o subsistemas independientes de la aplicación, tanto *hardware* como *software*.
- ***Requisitos técnicos.*** Muchos de los componentes de los robots de servicio se pueden adoptar de la tecnología de robots industriales. Sin embargo, la necesidad de ampliaciones significativas de estas tecnologías, así como de las prestaciones de ciertos componentes o subsistemas, se hace patente en muchas aplicaciones:
    - *Percepción y modelado del entorno*
    - *Navegación*
    - *Planificación de tareas*
    - *Interacción y comunicación con el ser humano y con su entorno*
- ***Seguridad.*** En muchos casos, las tareas que ejecutan los robots de servicio tienen lugar en espacios públicos, lo que requiere la definición adecuada de la interacción máquina-ser humano. Las normas de seguridad existentes son difíciles de aplicar en muchos casos y pueden llegar a limitar considerablemente el área de trabajo.

Finalmente, cabría resaltar la importancia de la investigación coordinada entre las distintas organizaciones y empresas productoras de los sistemas necesarios para desarrollar estos robots, ahora que parece que las aplicaciones de los robots de servicio tienden a ser cada vez más numerosas.

1.3. Objetivos y aportaciones de la Tesis

El objetivo principal de esta Tesis Doctoral es plantear una **arquitectura de referencia para unidades de control de robot de servicios teleoperados**. A lo largo de la tesis se hará referencia a esta arquitectura de referencia mediante el acrónimo ***ACROSET*** (**A**rquitectura de **C**ontrol para **RO**bots de **SE**rvicios **T**eleoperados).

Propósito principal

La idea que se persigue es la propuesta de una arquitectura de referencia compuesta por elementos reutilizables que soporten interoperabilidad entre ellos y sea aplicable al desarrollo de unidades de control de sistemas de servicio. Tal arquitectura debe reducir costes y tiempo de desarrollo, así como permitir implementaciones en sistemas fiables y de calidad. El punto de partida para obtener la arquitectura es un análisis del dominio (modelo de dominio), consistente en representar y organizar la información relativa a una familia de sistemas, a través del estudio de las aplicaciones existentes y del estudio de modelos genéricos (patrones o modelos bien reconocidos). En el Capítulo 4 se abordará dicho estudio revisando el estado actual de las arquitecturas de control para robots.

Del esquema general de un sistema teleoperado presentado en la **Fig. 1.1.**, hay que resaltar que la frontera entre la estación de teleoperación y la unidad de control local es cambiante, lo cual no tiene por qué afectar a la acotación del dominio de estudio de la tesis. A medida que el estado de la técnica lo va permitiendo, o la aplicación así lo exige, labores típicas del sistema de teleoperación que requieren la inteligencia del operador, pueden asignarse a la unidad de control local, aumentando su autonomía y liberando al sistema de teleoperación, que podrá dedicarse a otras actividades.

ACROSET debería ser aplicable tanto a teleoperación de manipuladores como de robots de servicio con un nivel limitado de autonomía (extender el dominio a la teleoperación de robots semi-autónomos sería generalizar demasiado, no garantizando la reutilización).

Objetivos

Para la consecución del propósito principal que se plantea con esta Tesis Doctoral, se proponen los siguientes objetivos, que al ser alcanzados, constituirán las principales aportaciones de la tesis:

1. Obtener una arquitectura de referencia para la unidad de control, que pueda ser reutilizada para este tipo de aplicaciones. Dicha arquitectura se podrá aplicar sobre diferentes robots de servicios teleoperados.

2. Realizar un estudio del estado de la técnica en arquitecturas de control de robots, acotando el dominio hasta los robots de servicio teleoperados.

3. Lograr una especificación genérica de requisitos para la unidad de control de este tipo de sistemas, a partir de un estudio riguroso de sus características. Describir dicha especificación mediante el uso del lenguaje unificado de modelado UML (*Unified Modelling Language)* y sus extensiones para expresar arquitecturas.

4. Demostrar que la arquitectura diseñada posibilita la implementación de sistemas que integren *software* y *hardware*, haciendo posible su interrelación, y permitiendo la construcción de distintos sistemas de control según sean las necesidades de la aplicación: desde una configuración puramente *software* a una combinación *hardware-software*, o en un caso extremo, una solución puramente *hardware*.

5. Aplicar la arquitectura en el control de diversos robots de limpieza para cascos de buques. La instanciaciones de la arquitectura deberán permitir diferentes implementaciones, incluyendo la combinación de componentes *software* con otros componentes *hardware* que pueden ser necesarios según sean los requisitos del sistema en cuanto a temporización, recursos, etc., y a las plataformas empleadas. Con ello se validará la arquitectura propuesta en la presente tesis.

En los siguientes párrafos se comentan algunos aspectos que completan la definición de objetivos realizada.

Especificación genérica de requisitos

La captura de requisitos funcionales y no funcionales del dominio de aplicación de la tesis es fundamental para el posterior desarrollo de un sistema basado en la arquitectura propuesta. El diseño de un sistema se basa en sus requisitos. Los errores u omisiones que puedan cometerse durante su definición son, sin duda, los más graves de todos, puesto que se arrastran durante todo el diseño y su corrección puede implicar un total replanteamiento del sistema. Por ello, una definición correcta y tan precisa como sea posible de los requisitos y restricciones que debe cumplir el sistema es clave para el posterior desarrollo del mismo.

Arquitectura y notación

La arquitectura es un modelo de muy alto nivel, que puede ser entendido tanto por los técnicos, ingenieros, informáticos, analistas o diseñadores, como por los clientes y usuarios finales. Como dice Garlan [Garlan94]:

"La arquitectura software simplifica la comprensión de grandes sistemas mediante su presentación en un nivel de abstracción en el cual es posible entender el sistema como un todo."

Al ser un modelo inteligible por todos, proporciona un lenguaje común en el que las necesidades y prioridades de cada parte pueden ser expresadas, negociadas y resueltas. Si la arquitectura está convenientemente descrita, constituye un excelente medio de comunicación para llegar a un consenso acerca de las propiedades y estructuras del sistema.

Para poder ser entendida por todos, es conveniente que el lenguaje utilizado para describir la arquitectura sea conocido también por todos. A ese respecto, las ventajas de utilizar un lenguaje de

modelado de reconocido prestigio y divulgación como es el Lenguaje de Modelado Unificado [UML99a], [UML99b], [OMG00], servirá para garantizar dicha comunicación y conocimiento de la arquitectura. Entre las causas que han hecho de UML un lenguaje muy utilizado, también destacan la facilidad de comunicación entre los integrantes del equipo de desarrollo del sistema y la posibilidad de dejar descrito el diseño para un futuro (modificaciones, ampliaciones, etc). Se trata de formalizar el proceso de creación en todas sus etapas, y normalizar la nomenclatura utilizada en el diseño, evitando los descuidos que se suelen cometer cuando el diseño formal consiste en un dibujo sobre un papel al que se le van añadiendo nuevas líneas y otras modificaciones.

No basta con utilizar una nomenclatura o un lenguaje semi-formal, también es necesario seguir un proceso de desarrollo riguroso en el diseño de sistemas en general y de arquitecturas en particular, para que cada etapa del análisis y diseño del sistema se cubra con garantías. Es importante que ese proceso permita volver atrás para modificar y afinar el diseño en vista de los resultados y que, además, permita no sólo generar una solución concreta, sino proponer una arquitectura de referencia para un dominio. Justamente por proponer una arquitectura de referencia, hay que estudiar si los métodos de desarrollo rigurosos orientados a objetos, como USDP [Jacobson00], COMET [Gomaa00] y ROPES [Douglass00] son adecuados para plantear arquitecturas de referencia, o bien es necesario contar con un método de diseño de arquitecturas como ABD [Bachmann00a] propuesto por el *Software Engineering Institute* (SEI) de la *Carnegie Mellon University* en Estados Unidos, o las *4 vistas de Hofmeister* [Hofmeister00], que utiliza una notación basada en extensiones de UML inspiradas en ROOM [Selic94]. Precisamente este último método plantea la utilización de componentes, puertos y conectores como entidades de primera clase para expresar arquitecturas.

Como consecuencia de lo anteriormente expuesto, otro objetivo importante de la tesis es el estudio y utilización de métodos semi-formales para el análisis y diseño de sistemas intensivos en *software*. Puesto que el objetivo principal es obtener una arquitectura de referencia, deben estudiarse los modelos de desarrollo de sistemas posibles (ABD, USDP, COMET, ROPES, 4 vistas de Hofmeister) y adoptar uno o varios. Si uno solo no cubriera las necesidades de análisis y diseño para obtener la arquitectura, se planteará una combinación de procesos de desarrollo.

Comportamiento temporal

Ya se ha mencionado que los sistemas robóticos actuales incorporan en muchos casos sistemas empotrados de tiempo-real y concurrentes. Lo que tienen de especial estos sistemas, además de necesitar que su funcionamiento sea estable, fiable y tolerante a fallos, es que para que el funcionamiento del sistema sea correcto deben cumplirse los tiempos de respuesta marcados para cada tarea o cada acción. Puesto que el comportamiento temporal del *software* diseñado para un sistema de tiempo-real depende de las prestaciones del *hardware* sobre el que se ejecute, se debe tener muy en cuenta las arquitecturas y capacidades del *hardware* disponible.

En muchos sistemas de tiempo-real, las consecuencias de no llegar a cumplir un tiempo de respuesta no hacen que el sistema funcione erróneamente. En estos casos, se dice que es un sistema *acríticos*[1], para diferenciarlos de los llamados sistemas *críticos*, donde un fallo en un determinado tiempo de respuesta ocasionaría que el sistema funcionara incorrectamente.

El funcionamiento correcto o incorrecto de un sistema de tiempo-real, dependiendo del tipo de aplicación, puede ser muy importante, incluso puede tener consecuencias catastróficas (por ejemplo, fallos en el controlador de vuelo de un avión, el sistema de control de un tren, etc.)

La arquitectura que se busca proponer en esta tesis será implementada normalmente en sistemas críticos (o por lo menos con una parte crítica) y con requisitos de seguridad por ser robots de servicio que normalmente están en contacto con personas, o cuyas operaciones pueden tener efectos secundarios sobre la seguridad de las personas. Por lo tanto será necesario que las instanciaciones de la

[1] En inglés, se utiliza la terminología *soft real time* (acríticos) y *hard real-time* (críticos).

arquitectura puedan ser evaluadas para comprobar que cumplen los requisitos temporales establecidos para un sistema concreto.

Integración software/hardware

Los términos *codiseño*, sistemas empotrados (*embedded systems*), sistemas de tiempo-real y sistemas críticos introducen una tendencia actual importante en el desarrollo combinado de *software* y *hardware* no sólo para robots, sino para cualquier sistema electrónico de los que hoy en día podemos encontrar desde en nuestro bolsillo (PDAs, teléfonos móviles, consolas de video-juegos, etc), hasta en la sala de control de una central nuclear. Dichas tendencias en el ámbito de la mecatrónica y la robótica también tienen y tendrán un papel importante, puesto que las mismas tarjetas controladoras de motores o ciertas unidades de control son sistemas empotrados que deben integrarse perfectamente en la arquitectura global del sistema.

No es objetivo de la tesis plantear un estudio de codiseño para ver qué parte de ACROSET podría ser mejor implementada en *hardware* y qué parte en *software*, puesto que este trabajo de tesis no aborda una solución concreta, sino que pretende dar una arquitectura de referencia que se pueda utilizar para muchos sistemas. Pero si que es un objetivo el dotarla de las características que garanticen suficientemente su reutilización en distintas configuraciones. En cualquier caso, su instanciación deberá satisfacer los requisitos particulares, implementando una parte en *hardware* y otra en *software* según estos requisitos. Por ello es importante plantear una arquitectura que permita la integración del *software* y el *hardware* que componen el sistema, posibilitando construir distintas implementaciones según sean las necesidades del sistema: desde una configuración puramente *software* a una combinación *hardware-software*, o en un caso extremo, a una solución puramente *hardware*. A modo de ejemplo se expondrá la implementación de la arquitectura para el sistema GOYA y su ampliación para el sistema EFTCoR, que permitirá mostrar la capacidad de la arquitectura de integrar *hardware* y *software.*

Aplicación y validación de la arquitectura

Las propuestas y conclusiones de una investigación no tendrían suficiente interés si no se validaran con su implementación en un sistema real. Para validar la arquitectura propuesta se cuenta con varios robots de limpieza de barcos desarrollados por el grupo de investigación DSIE, como son los sistemas GOYA y EFTCoR. Estos entornos de validación son ideales en tanto que introducen la mayoría de requisitos y características propias del dominio de estudio de esta tesis y ofrecen la posibilidad de validar la arquitectura en diferentes tipos de implementaciones y combinaciones hardware/software.

1.4. Estructura y contenidos de la Tesis

La tesis ha sido estructurada en ocho capítulos, tres anexos y la bibliografía.

Después de este primer capítulo introductorio, donde se revisan los antecedentes y se proponen los objetivos de la Tesis Doctoral, se plantearán en el siguiente capítulo las definiciones necesarias para la comprensión de la misma.

En el Capítulo 3 se expondrá un breve estado del arte de las notaciones y metodologías de desarrollo de arquitecturas, justificando las elecciones hechas en la tesis para expresar sus contenidos y diseñar la arquitectura propuesta. En el siguiente capítulo se expone el estado de la técnica en el dominio de los robots de servicio teleoperados, que es el marco de trabajo de la tesis.

En los dos capítulos posteriores (5 y 6) se explica todo el análisis de requisitos y proceso de desarrollo que se ha llevado a cabo para alcanzar el principal objetivo marcado en la tesis: la obtención de una arquitectura de referencia para unidades de control de robots de servicios teleoperados y su validación con algunos robots de limpieza para barcos.

En el Capítulo 7 se presenta la validación de la arquitectura mediante su instanciación para diversos robots de limpieza de la superficie de barcos.

El contenido fundamental de la tesis finaliza con el capítulo de conclusiones y futuros trabajos, complementándose la información presentada con los anexos correspondientes, así como la bibliografía utilizada para confeccionar esta tesis.

A continuación se describe la composición de los diferentes capítulos y un breve resumen de los mismos.

Capítulo 1. Planteamiento y objetivos de la tesis.

Breve introducción a la tesis y planteamiento de los objetivos de la misma.

Capitulo 2. Arquitecturas de referencia: definición y tendencias.

Se define arquitectura de referencia y las distintas vistas que la describen, su contribución en el desarrollo de un sistema y la utilización de patrones y estilos arquitectónicos en su planteamiento. También se muestran las tendencias actuales en el diseño de arquitecturas, modelos de componentes y *frameworks*.

Capítulo 3. Notaciones y metodologías de desarrollo de arquitecturas

Aquí se comentarán las distintas técnicas y lenguajes de descripción formal o semi-formal que existen, explicando las diferencias entre ellos, y justificando la utilización de *UML* como notación semi-formal para esta tesis. Asimismo, se describirán brevemente los principales mecanismos del lenguaje y herramientas CASE utilizadas.

En este capítulo se describen también diversos métodos de desarrollo riguroso orientados a objetos y a la arquitectura (ABD, USDP, COMET, ROPES). A partir de estos métodos o una combinación de ellos, se explicará cual es el enfoque metodológico utilizado en la tesis para desarrollar la arquitectura.

Capítulo 4. Estudio del dominio de los robots de servicio teleoperados.

En este capítulo se abordará el estudio del dominio de aplicación de la arquitectura propuesta en la tesis. Se expondrán las nuevas tendencias y soluciones aplicables al dominio de los robots de servicio teleoperados, particularizando el estudio en las diferentes arquitecturas de control propuestas. Se ofrecerá una comparativa entre las arquitecturas existentes, así como aportaciones de esta tesis frente al estado de la técnica actual.

Capítulo 5. Análisis de requisitos y estrategias para obtener ACROSET.

El capítulo plasmará la aplicación, en sus primeros pasos, de los métodos de desarrollo más adecuados para obtener la arquitectura de referencia ACROSET. Como fruto de la aplicación de los mismos, se establecerán los requisitos y se ofrecerán las distintas estrategias u opciones arquitectónicas para obtener la arquitectura.

Capítulo 6. Una arquitectura de referencia para unidades de control de robots de servicio teleoperados.

En este capítulo se describirá la arquitectura propuesta en la tesis: ACROSET.

Capítulo 7. Aplicación de ACROSET en robots de limpieza de barcos.

Para validar la arquitectura se contará con su implementación en la unidad de control de varios robots de limpieza de barcos que exigen distintas instanciaciones de la arquitectura y distintas configuraciones *hardware/software.*

Capítulo 8. Conclusiones y futuros trabajos.

Se exponen los resultados obtenidos en esta Tesis Doctoral y se presentan las líneas de investigación a seguir como extensión de la misma.

Anexo I. Conceptos fundamentales de ABD y 4 vistas de Hofmeister.

Se ofrece un resumen de los aspectos fundamentales de la notación UML, así como una descripción básica de los pasos a seguir en los procesos de desarrollo asociados a ABD y 4 vistas de Hofmeister.

Anexo II. Plantillas de Atributo

Se describen con detalle las plantillas de atributo presentadas en el Capítulo 5.

Anexo III. Complementos al modelo de análisis del sistema GOYA

Se ofrecen algunos diagramas complementarios al modelo de análisis del sistema GOYA realizado en el Capítulo 7: Modelo de datos de las actividades de mantenimiento y desarrollo detallado de casos de uso.

Bibliografía.

Se enumera la bibliografía utilizada para la elaboración de la tesis.

Capítulo 2

Arquitecturas de Referencia: Definición y Tendencias

2.1. Necesidad de arquitecturas de referencia

En el capítulo anterior, ya se mencionó la diferencia de desarrollar sistemas robóticos, en particular, la unidad de control de un robot, respecto al desarrollo de otros sistemas intensivos en *software*. Se puso especial atención en resaltar las características particulares que tiene el diseño de controladores para robots, en tanto que son sistemas complejos, muchas veces implementados en sistemas *empotrados* con restricciones de tiempo-real y requisitos de seguridad elevados tanto en su diseño como en su ejecución. En este sentido, suelen encontrarse con las típicas restricciones de los sistemas críticos.

A la hora de desarrollar un sistema tan complejo como un robot, no basta tener como objetivo que el sistema funcione bien. Con esta única meta, el desarrollo del sistema se convertiría en un continuo "parcheo" de una solución inicial, haciéndose el diseño cada vez más confuso y difícil de mantener. Por ello, debe plantearse un diseño basado en una **arquitectura** del sistema (o de una familia de sistemas) bien definida que permita controlar el desarrollo del mismo. Desafortunadamente, muchos desarrolladores juntan arquitectura e implementación de tal forma que el sistema se diseña con el único criterio de: *"si funciona, vale"*. Con esta forma de actuar, se pierden las ventajas de una arquitectura bien concebida.

Como recogen Coste-Manière y Simmons en [Coste00], una forma de abordar la complejidad de los sistemas es a través de la **modularidad** dentro de una estructura dada; la complejidad del sistema global se puede reducir descomponiéndolo en componentes más pequeños, con niveles de abstracción bien definidos e interfaces claros entre ellos. Esta estructura modular se puede crear mediante modelos de desarrollo rigurosos, dando lugar a **arquitecturas** bien definidas que permitan ser implementadas en estos complejos sistemas.

Una arquitectura debería facilitar el desarrollo de los sistemas robóticos proporcionando restricciones beneficiosas en el diseño e implementación de la aplicación deseada, sin ser demasiado restrictivas [Coste00]. Aunque este criterio es fácil de expresar, es mucho más difícil de llevar a la práctica. En

particular, aplicaciones diferentes tienen diferentes necesidades, que pueden ser satisfechas con diferentes arquitecturas. La clave consistirá en la adopción de un compromiso entre los *stakeholders*[1] que influyen en la arquitectura, para escoger la que mejor se aproxime al mayor número de posibles necesidades del sistema (ver punto *2.2.2.1. Atributos de calidad*).

Las tres piezas clave que ayudan en el diseño de una arquitectura son: **notaciones, metodologías** y **herramientas** para automatizar el proceso de diseño y simular comportamientos.

En los puntos siguientes se profundiza en la definición de arquitectura de referencia y las distintas vistas que la describen, su contribución en el desarrollo de un sistema y la utilización de patrones y estilos arquitectónicos en su planteamiento. También se muestran las tendencias actuales en el diseño de arquitecturas y *frameworks*. En el capítulo siguiente se estudiarán los medios, metodologías y herramientas para poder diseñar, expresar y describir una arquitectura.

2.1.1. Definiciones de arquitectura

En la literatura técnica se pueden encontrar diversos estudios que describen lo que se entiende por arquitectura, tanto en el marco general de la *Ingeniería del Software* [Bachmann00a], [Bass98], [Shaw01], [Glass02], como en su aplicación concreta en el campo de la robótica [Coste00], [Kramer93]. Puesto que el campo de estudio de la *arquitectura software* es bastante importante para la *Ingeniería del Software*, la mayoría de las definiciones se refieren al diseño y organización del *software*, pero se pueden generalizar para sistemas *hardware-software* y arquitecturas de sistemas intensivos en *software*. De esta manera, cuando se hable de componentes o subsistemas, con sus respectivos interfaces, en la mayoría de los casos, estos componentes podrán ser tanto *software*, como *hardware*.

En el documento del SEI *"How do you define Software Architecture"* [SEI03], se puede encontrar una recopilación de definiciones extraídas de las más importantes fuentes bibliográficas. De entre las definiciones *clásicas*, destacamos aquí la de Booch, Rumbaugh y Jacobson, los creadores de UML [UML99b]:

*"Una **arquitectura** se define por una serie de decisiones significativas acerca de la organización de un sistema software, la selección de elementos estructurales y sus interfaces que conforman un sistema, junto con su comportamiento especificado como colaboraciones entre esos elementos. Describe también la composición de estos elementos estructurales y de comportamiento en subsistemas progresivamente más grandes, y los estilos arquitectónicos que guían esta organización, estos elementos y sus interfaces, sus colaboraciones y su composición"*.

Una definición más simple, pero también acertada, la ofrece D. Garlan en [Garlan95]:

"La arquitectura de un sistema software es la estructura de sus componentes, sus relaciones y los principios que gobiernan su evolución a lo largo del tiempo".

Existen otras muchas definiciones de *arquitectura*, la mayoría muy parecidas, diferenciándose unas de otras en el énfasis que ponen sobre los aspectos que desean resaltar. De todas ellas nos quedaremos con la propuesta por Bass [Bass98]:

"La arquitectura de un sistema software es la estructura o estructuras del sistema, incluyendo sus componentes software, las propiedades externamente visibles de dichos componentes y las relaciones entre ellos".

---

[1] En la terminología de Bass [Bass98] se definen así los *agentes* de la organización que desarrolla la arquitectura (desarrolladores, responsables, publicistas, etc.), los clientes, los usuarios finales, etc. Cada uno de estas personas o grupos de personas establecen requisitos diferentes y en muchos casos, contradictorios, que debe cumplir la arquitectura o el sistema que se desarrolla. La labor del *arquitecto* a este respecto consistirá en intentar conciliar las restricciones impuestas por unos y otros, llegando a un compromiso que permita desarrollar el sistema o arquitectura.

Esta definición es la adoptada en el *Software Engineering Institute* (SEI) de la *Carnegie Mellon University* en EE.UU. por los creadores del método de desarrollo basado en arquitectura ABD (*Architecture Based Design Method*) [Bachmann00a] y de los métodos de evaluación de arquitecturas SAAM (*Software Architecture Analysis Method*) [Kazman94] y ATAM (*Architecture Tradeoff Analysis Method*) [Kazman00]. Esta definición está ampliamente justificada y documentada en el libro "*Software Architecture in Practice*" de L. Bass [Bass98].

Se podría matizar no obstante, el hecho de que se nombre explícitamente "sistema *software*", "componente *software*", puesto que la definición podría ser más general, ya que dichas estructuras y componentes pueden ser *hardware*, *software* o distintos subsistemas [IEEE-610.12]. En este sentido, la definición dada en la recomendación estándar IEEE Std 1471-2000, *"IEEE Recommended Practice for Architectural Description of Software-Intensive Systems"*, está generalizada para sistemas intensivos en *software*, no sólo puramente *software* [IEEE Std-1471-2000]:

"Se define arquitectura como la organización fundamental de un sistema, plasmada en sus componentes, las relaciones entre los mismos y con su entorno, y los principios que gobiernan su diseño y evolución".

Esta definición intenta abarcar una variedad de usos del término *arquitectura* reconociendo sus elementos comunes. El concepto principal que subyace en estas definiciones es la necesidad de comprender y controlar esos elementos del sistema que capturan la utilidad, coste y riesgo del mismo. En algunos casos, estos elementos son componentes físicos y sus relaciones. En otros casos, estos componentes no son físicos, sino lógicos. En un caso adicional, estos elementos son los principios o patrones que crean las estructuras organizativas perdurables.

Se puede observar fácilmente la gran similitud entre las tres últimas definiciones, que se han escogido por ser las más significativas, influyentes y acertadas en opinión del autor de esta Tesis Doctoral. Teniendo en cuenta las matizaciones anteriormente expuestas, se podría adoptar una definición conciliadora entre ellas, que incluya los conceptos importantes de cada una de las definiciones. De este modo se considera la siguiente definición como la más adecuada para los propósitos de esta tesis:

"Se define arquitectura como la estructura o estructuras en las que se organiza un sistema, lo cual incluye, sus componentes con las propiedades externamente visibles de los mismos, las relaciones entre estos componentes y con su entorno, y los principios que gobiernan su diseño y evolución".

Esta definición de arquitectura asume las siguientes consideraciones:

- Asume que un sistema puede incluir más de una estructura *("...es la estructura o estructuras del sistema...")*. En consecuencia, la descripción de la arquitectura debe incluir la descripción de todas estas estructuras sin que pueda primarse en principio una de ellas sobre las demás. Puede haber más de una clase de estructura (módulos, procesos, etc.) que defina unas relaciones entre componentes (subdivisión, sincronización) en más de un contexto (tiempo de desarrollo, tiempo de ejecución,...). Con objeto de no imponer restricciones innecesarias, la definición no especifica qué tipo de componentes deben ser los componentes arquitectónicos y qué relaciones habrá entre ellos. Puede ser un proceso, una librería, una memoria, una tarjeta comercial o en general, un componente *hardware* o *software*, etc.
- La arquitectura incluye las propiedades *externamente visibles* de los componentes del sistema. Se expresa así el hecho de que la arquitectura debe abstraer alguna información del sistema, pero proporcionando suficiente información para el análisis, toma de decisiones y reducción de riesgos. Esto significa dos cosas:
  - ✓ El comportamiento de un componente es parte de la arquitectura sólo si dicho comportamiento puede ser observado por otro componente del sistema o por un actor externo que interacciona con él.
  - ✓ Debe proporcionarse una descripción clara de las interfaces que cada componente ofrece al resto y al exterior (qué servicios proporciona y cómo los proporciona).

- Una arquitectura puede ser mala o buena si permite que el sistema alcance su comportamiento, rendimiento y cualidades expresadas en los requisitos del sistema para todo su ciclo de vida. También son parte de la arquitectura los principios que gobiernan su diseño y evolución en el tiempo. Esto revela la importancia de los métodos de desarrollo de arquitecturas a la hora de definir bien una arquitectura, asegurándose así el tiempo de vida, calidad y mantenibilidad del sistema.
- La definición también asume que todo sistema tiene una arquitectura, buena o mala, pero la tiene, ya que todo sistema puede ser descrito en función de sus componentes y de las relaciones entre ellos. Incluso en el caso de un sistema monolítico hay una arquitectura compuesta por un solo componente. La definición revela la diferencia entre la **arquitectura de un sistema**, que existe porque existe el sistema, y la **descripción de dicha arquitectura** que puede o no existir[2]. Esto revela la importancia de la **representación de una arquitectura**.

2.1.2. Arquitectura del sistema vs. arquitectura *software*

En marzo de 2003, el SEI *(Software Engineering Institute)* de la *Carnegie Mellon University* en EE.UU. junto con el Departamento de Defensa de EE.UU. y algunos organismos más, organizaron el *DoD Architecture Framework (DoDAF[3]) and Software Architecture Workshop* [SEI03-TN6]., donde se discutieron las similitudes y diferencias entre la representación de la arquitectura *software* y la arquitectura del sistema *(system and software architectures)* en cualquier sistema intensivo en *software* y los retos involucrados en su diseño y representación. Dicha discusión está fundamentada en la recomendación estándar de IEEE 1471-2000 [IEEE Std-1471-2000], y de ella se pueden extraer las siguientes conclusiones:

Aunque todo el mundo parece estar de acuerdo en que *"la arquitectura"* es un ingrediente esencial en el desarrollo de sistemas más o menos complejos, hay una profusión de términos para nombrar conceptos arquitectónicos que, en la mayoría de los casos, solapan su significado y ámbito de estudio. Así nos encontramos en la literatura multitud de referencias a *"arquitectura computacional"*, *"arquitectura de sistema"*, *"arquitectura software"*, *"arquitectura del sistema de sistemas"*, *"arquitectura hardware"*, *"arquitectura de comunicaciones"*, *"arquitectura de datos"* y muchas otras, cuyas definiciones son normalmente demasiado generales, o incluso ambiguas, como para evitar ese solapamiento (de hecho, probablemente no se pueda evitar).

Ciertamente, si quisiéramos dividir el estudio, diseño y representación de la estructura de un sistema en diferentes arquitecturas (*software*, *hardware*, de datos, etc.) nos encontraríamos que son dependientes unas de otras y están relacionadas.

Por ejemplo, si hablamos de sistemas de tiempo-real, el comportamiento temporal del *software* depende de las prestaciones del *hardware*, no sólo de las prestaciones individuales de cada componente (procesadores, tarjetas de adquisición, memorias, ASICs, etc), sino también de cómo estén interconectados esos componentes, del tipo de acceso a memoria se utiliza, del algoritmo de procesamiento y un largo etcétera. Por tanto, se puede concluir que los métodos de diseño de *software* y arquitecturas *software* para sistemas de tiempo-real estarán fuertemente influenciados por el *hardware* que se elija para ellos, así como la arquitectura en la que se organicen dichos componentes *hardware*. A modo de ejemplo, se puede consultar [Redell98], donde se hace un detallado estudio de modelado del controlador para una de las "patas" de un robot cuadrúpedo basándose en dos posibles arquitecturas *hardware*, una con dos microcontroladores y otra con cinco, interconectados por un bus

---

[2] Esta es la posición de [Bas98]. Bachmann en [Bachmann00a] afirma que la arquitectura sólo existe si existe una descripción de la misma. Ambas posiciones son razonables, simplemente ven el problema desde diferentes puntos de vista.

[3] El DoDAF es una revisión que actualmente está en marcha, del C4ISR (Command, Control, Communications, Computer, Intelligence, Surveillance, and Reconnaissance) Architecture Framework.

CAN, donde se puede apreciar la importancia de la arquitectura *hardware* adoptada para implementar el *software* de un controlador.

En [SEI03-TN6] se reconoce la dificultad de encontrar definiciones exactas para cada término. Se defiende la interrelación entre arquitectura del sistema y arquitectura *software*, diferenciando una aproximación sistemista frente a una aproximación *software* y estableciendo no obstante la distinción siguiente:

La "arquitectura del sistema" está particularmente asociada a la funcionalidad, mientras que la arquitectura software está asociada más a cómo se alcanza dicha funcionalidad, muestra cómo se puede llegar a realizar determinadas funciones como resultado de la cooperación de ciertos elementos estructurales.

La comunidad de **ingenieros de sistemas** parece sentirse más cómoda a la hora de detallar una arquitectura con la aproximación que empieza diseñando los elementos *hardware* con una funcionalidad asociada. Los **ingenieros de *software*** abandonaron hace mucho tiempo esta aproximación a favor de una aproximación **orientada a objetos (OO)** que asigna el *software* a un *hardware* en una fase más tardía del ciclo de desarrollo. La mayoría de los componentes *software* que se distribuyen por todo el sistema están pobremente representados en la primera aproximación, pero están muy bien representados en la aproximación OO. Las infraestructuras *software*, como sistemas operativos, protocolos de comunicación y *middleware* también son difícilmente representables en la aproximación puramente sistemista. Las "tensiones" entre ambas comunidades hacen que sea desafiante el reto de resolver estos problemas.

Se suele diferenciar entre el arquitecto del sistema, que configura la arquitectura *hardware*, y el arquitecto *software*, y se dice que sus trabajos están ligados entre sí [Hofmeister00]. La arquitectura *hardware* es otra entrada fundamental para el diseño de la arquitectura *software* si se toma como condicionante, pero también puede adaptarse, cambiar o sustituir partes del *software* según sean las necesidades del sistema. El arquitecto *software* da los requisitos al arquitecto del sistema, y puede sugerir modificaciones conforme el proceso de diseño de la arquitectura *software* progresa. Hoy en día la arquitectura *hardware* está íntimamente ligada a la *software*, se condicionan y se complementan, más aún con el empuje del *hardware* reconfigurable (programable, pero *hardware* al fin y al cabo), sobre todo las FPGAs. También cabe destacar, aunque queda fuera del ámbito de esta tesis, el campo de estudio del **codiseño**, que aboga por el diseño complementario, desde las fases iniciales, de *hardware* y *software*, realizando el particionado[4] del sistema más adelante según los resultados de co-simulación (para más detalles ver [Staunstrup97]).

Vistas de una arquitectura

Lo que sí está claro es que, independientemente de si se proyecta una arquitectura del sistema global, *hardware* o bien una arquitectura *software*, es necesario que las vistas de la arquitectura sean lo suficientemente explicativas para el propósito que se utilizan, que no es otro que el de **describir** la arquitectura.

La arquitectura *software* y la arquitectura de sistema consideradas como vistas, tienen diferentes propósitos, pero también se solapan de alguna forma.

Los arquitectos *software* usan la palabra "vista" para describir una serie de elementos *software* y las relaciones entre ellos. Por ejemplo, una vista lógica describe clases y relaciones entre clases, una vista de proceso describe procesos y sus usos. Las definiciones de vistas se complican por el hecho de que es posible más de una representación para una sola vista (por ejemplo, diagramas de estado y diagramas de actividad pueden ser utilizados para representar aspectos de comportamiento de procesos).

---

[4] División entre las partes del sistema que se implementan en *hardware* y las que lo hacen en *software*.

Debido al enorme número de vistas que se podrían construir, los desarrolladores de la arquitectura de un sistema, deben seleccionar qué vistas (tanto a nivel de sistema como de *software*) son las más importantes para documentar la arquitectura. También se debería especificar el orden y la secuencia temporal necesarios para desarrollar esas vistas. La selección de algunas de las múltiples vistas posibles depende del propósito para el que se construye la arquitectura, las personas que la revisarán, y otros muchos factores [IEEE Std 1471- 2000]. En el punto *2.2. Vistas de la arquitectura de un sistema*, se profundiza en la utilización de las vistas para representar la arquitectura.

2.1.3. Patrones, estilos de arquitectura, modelos de referencia, arquitecturas de referencia y específicas de dominio

Asociados al concepto de arquitectura (sobre todo a nivel de *arquitectura software*), hay una serie de términos de uso común. Como ocurre con la propia arquitectura no hay un consenso definitivo sobre su alcance. Sin embargo, aunque estos términos carezcan de definiciones precisas, permiten a los diseñadores razonar sobre sistemas complejos usando abstracciones que los hacen inteligibles. Los patrones y estilos arquitectónicos y los modelos y arquitecturas de referencia proporcionan el contenido semántico necesario para describir las propiedades del sistema, su evolución en el tiempo, sus paradigmas computacionales y sus relaciones con otros sistemas similares. Las descripciones que aparecen en esta sección se han confeccionado a partir de las dadas en [Gamma95], [Buschmann96], [Klein99], [Bass98] y [Hofmeister00].

Patrones y estilos arquitectónicos

Un patrón es una solución recurrente a un problema estándar. Los patrones de diseño capturan las estructuras estáticas y dinámicas de soluciones que funcionan de forma satisfactoria dentro de ciertos contextos o aplicaciones. Así, cada patrón define ciertos componentes y las reglas mediante las cuales pueden relacionarse con objeto de resolver un problema *software* específico.

Obsérvese que los patrones son definidos por inducción. Los patrones *software* describen soluciones que han sido probadas con éxito al aplicarlas una y otra sobre los mismos problemas. Los patrones se documentan usando un formato en el que se describe no sólo el patrón, sino su nivel de abstracción, su ámbito de aplicación y las consecuencias de su uso, entre ellas su relación con ciertos atributos de calidad. La documentación de patrones pone a disposición de los diseñadores soluciones probadas para problemas comunes, favorece el uso de esquemas de reconocida eficacia y constituye un paso más hacia la reutilización del *software*.

El estudio y documentación de patrones es un campo en auge desde mediados de los 90, [Gamma95], [Buschmann96], sin embargo aún queda mucho trabajo por hacer, en especial en lo referente a la asociación de patrones con atributos de calidad [Klein99], a la clasificación de los patrones y a la definición de lenguajes de descripción de patrones [Coplien95][5].

Un estilo de arquitectura es una descripción de un conjunto de componentes y un patrón o patrones que describen la transferencia de control y datos entre los mismos. Un estilo define una serie de restricciones respecto al tipo de componentes que pueden utilizarse y a las formas en que dichos componentes pueden interaccionar. La frontera entre patrón y estilo no está clara en la literatura, ni tampoco en las descripciones aquí ofrecidas (diferenciando patrón arquitectónico de otros patrones, por ejemplo, de diseño, de análisis, etc). La construcción cliente/servidor ¿es patrón o estilo? Desde luego es una solución probada a un problema concreto. Sin embargo pueden adoptarse diferentes patrones para su adecuación a un problema particular. Los patrones pueden ser vistos como una *micro-arquitectura* que define un estilo *puro* o primario. Un estilo de arquitectura viene generalmente definido por una combinación de patrones. Puede así entenderse el estilo como una abstracción de nivel más alto que los patrones, que define una codificación particular de elementos de diseño y

[5] El lector puede encontrar una excelente base de datos sobre patrones en [Brad00], en donde se ofrecen gran cantidad de enlaces a publicaciones sobre el tema, incluyendo todos los clásicos.

combinaciones formales de los mismos, limitando su número y el tipo de sus posibles combinaciones y mecanismos de interacción. Esta distinción no es seguida por todos los autores, y es habitual que estilos arquitectónicos como la descomposición del sistema en niveles, los sistemas cliente/servidor, los diseños *pipe-filter*, las máquinas virtuales, las pizarras, etc., sean definidos como patrones en la literatura referente al tema.

Modelos de Referencia, Arquitecturas de Referencia y Arquitecturas Específicas de Dominio

Un modelo de referencia es una descomposición estándar de un problema conocido en unidades funcionales que trabajan cooperativamente para resolverlo. Un modelo de referencia no es una arquitectura, sino una descripción de la división de la funcionalidad de un sistema *software* en diferentes piezas y de la forma en que fluyen los datos a través de ellas. Los modelos de referencia son típicos de dominios muy maduros, en los que existe un consenso entre los diseñadores acerca de la descomposición funcional del sistema.

Ejemplos de modelos de referencia son la torre OSI (*Open System Interconnection*) y los modelos de referencia para el diseño de compiladores**.**

Una arquitectura de referencia es una correspondencia entre un modelo de referencia y los componentes *software* que implementan la funcionalidad definida en el mismo [Bass98]. La correspondencia entre las unidades funcionales definidas en el modelo y los componentes de la arquitectura de referencia no es necesariamente uno a uno.

Una arquitectura de referencia define la infraestructura común a todos los sistemas del dominio considerado, los componentes o subsistemas que incluyen y las interfaces que deben ofrecer dichos componentes o subsistemas. Disponer de tal arquitectura facilita enormemente el desarrollo de nuevas aplicaciones, pues permite por un lado la reutilización de modelos y componentes y por otro ofrece un marco para el desarrollo de los mismos.

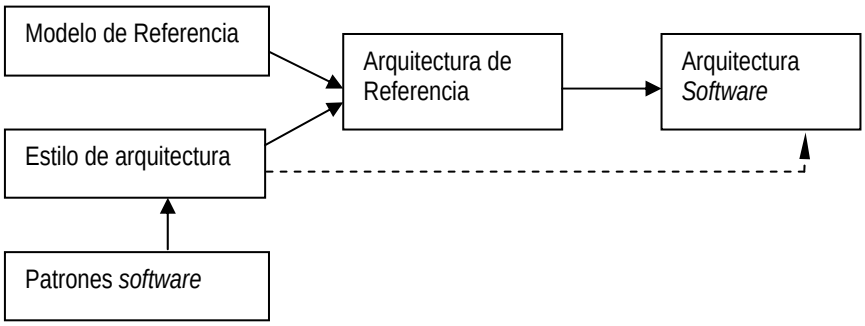


Fig. 2.1.- Relación entre patrones, estilos, modelos de referencia y arquitecturas

Un ejemplo de arquitectura de referencia es un compilador. Hay una notación general de los elementos básicos de un compilador (por ejemplo, *front end, back end, symbol table*), qué deberían ser, y como están interconectados. Alguien que diseñe un nuevo compilador no empezaría desde el principio, sino que empezaría con esta arquitectura de referencia básica en mente al definir la arquitectura *software* del nuevo compilador.

Las arquitecturas de referencia se construyen a partir de un modelo de referencia en el que las unidades funcionales y sus relaciones se definen siguiendo un determinado estilo o patrón (**Fig. 2.1**). La arquitectura de referencia, aunque ya define pautas claras de implementación, sigue siendo un artefacto bastante general que tendrá que especializarse para definir la arquitectura de un sistema concreto.

La clase de situaciones para las que una arquitectura se piensa usar es su **dominio**. Por ejemplo, una arquitectura podría ser aplicada al dominio de los robots de servicio teleoperados, podrían definirse así una **arquitecturas específica de dominio**. Una **aplicación** es un subconjunto de una o más situaciones en el dominio de una arquitectura que tienen similares características. Un robot de limpieza

de barcos, sería una aplicación en el dominio de robots de servicio teleoperados. La realización de una arquitectura en *hardware* y *software* para una aplicación es la **implementación** de la arquitectura.

Los términos **arquitectura de referencia** y **arquitectura específica de dominio**, a veces son usados de una manera similar. Como los estilos arquitectónicos, definen tipos de elementos e interacciones permitidas, pero en este caso se aplican a un dominio en particular. Por tanto, definen como la funcionalidad del dominio es trasladada[6] a los elementos de la arquitectura [Hofmeister00].

Una **arquitectura tipo línea de producto** se aplica generalmente a una serie de productos dentro de una organización o compañía. Esta idea ha crecido desde la comprensión de que, cuando una compañía tiene una familia de productos similares, son más consistentes y pueden ser desarrollados a menor coste si comparten un diseño común y quizás parte de la implementación.

En la **Tabla 2.1** se recoge una recopilación de todos estos términos especificando qué elementos aparecen en cada término.

Término	*Define tipos de elementos y como interactúan*	*Define un mapping de funcionalidad en elementos de la arquitectura*	*Define instancias de elementos de la arquitectura*
Un estilo o un patrón arquitectónico (normalmente no específico de dominio)	Si	Algunas veces	No
Una arquitectura de referencia o arquitectura específica de dominio (se aplica en un dominio particular)	Si	Si	No
Una arquitectura de línea-de-producto (se aplica a una serie de productos dentro de una organización)	Si	Si	Algunas veces
Una arquitectura sofware o de un sistema (se aplica a un sistema o producto)	Si	Si	Si

Tabla 2.1. Comparación de términos relacionados con la arquitectura [Hofmeister00]

2.1.4. Aportaciones de la arquitectura en el proceso de desarrollo de un sistema

La arquitectura de un sistema constituye un modelo inteligible de la estructura del sistema y de las relaciones entre sus componentes. El nivel de abstracción necesario para describirla se sitúa aún muy lejos de los detalles de implementación, pero exhibe ya toda una serie de características que van a condicionar tanto el producto final como su proceso de desarrollo. Los principales intereses a la hora de diseñar una arquitectura son mejorar el **tiempo de vida, calidad** y **mantenimiento** de un producto. Si la arquitectura está correctamente descrita, favorece los siguientes aspectos [Garlan94]:

Documentación de la estructura y de las propiedades del sistema.

Al constituir la arquitectura un modelo *fácil* de entender, proporciona un lenguaje común en el que las necesidades y prioridades de cada parte (*stakeholders*) pueden ser expresadas, negociadas y resueltas. Si la arquitectura está convenientemente descrita, constituye un excelente medio de comunicación para llegar a un consenso acerca de las propiedades y estructuras del sistema.

---

[6] En inglés *mapped.*

Evaluación precoz del sistema.

Al mostrar todas las características importantes que tendrá el sistema una vez construido, la arquitectura es la primera representación completa de las decisiones de diseño y, por tanto, la primera oportunidad seria de evaluar dichas decisiones. Oportunidad que, además, se ofrece antes de que el sistema haya empezado a construirse, y en un formato que, como se acaba de explicar, puede ser entendido por clientes y usuarios finales que de esta manera pueden sumarse a la discusión y al consenso.

Reutilización de modelos.

Al alejarse de los detalles de implementación, la arquitectura se convierte en un artefacto transferible que puede ser aplicado a otros sistemas que compartan requisitos similares. Reutilizar una arquitectura es una decisión crítica y de muy largo alcance, pues supone adoptar unas decisiones de diseño que van a afectar a todo el proceso de desarrollo. Los beneficios de la reutilización son tanto mayores cuanto antes se apliquen en el ciclo de desarrollo del sistema, pero también los riesgos. Puesto que la arquitectura define las propiedades externamente visibles de los componentes, todos los sistemas que compartan una misma arquitectura pueden, en principio, utilizar componentes que exhiban un mismo comportamiento, abriéndose una puerta para el uso de componentes comerciales.

Ejemplos de reutilización de arquitecturas son las líneas de producto [Gannod00], [Bosch00] y las arquitecturas de referencia específicas de un dominio. En las líneas de producto, aunque cada uno de los productos posea ciertas características propias, todos comparten una misma arquitectura y por tanto utilizan en gran medida los mismos componentes. En los dominios en los que se ha alcanzado una gran madurez y existe un consenso acerca de la división funcional del sistema es posible la creación de un mercado de componentes y la construcción de aplicaciones como mecanos, a base de ensamblar componentes. El mercado de componentes *hardware* es muy maduro, no así el de componentes *software*. En el punto 2.3.1 se reflexiona sobre las tendencias en desarrollo de componentes *software*.

El prototipado evolutivo y el crecimiento incremental.

Una vez que la arquitectura ha sido definida, puede ser analizada y prototipada como un modelo ejecutable. Esto tiene dos benéficas consecuencias. En primer lugar permite la simulación del sistema y la detección de errores de diseño antes de iniciarse la fase de implementación. En segundo lugar, los componentes simulados pueden ir sustituyéndose por versiones cada vez más refinadas de los componentes finales, favoreciendo de esta manera el crecimiento incremental y la adopción de ciclos de vida en espiral dirigidos por riesgos. Como consecuencia, se ayuda a reducir costes y tiempo de desarrollo. En el Capítulo 3 se abordará el estudio de diferentes métodos de desarrollo evolutivo e incremental.

La arquitectura guía finalmente las tareas de implementación, incluyendo el diseño detallado, desarrollo de código, integración y pruebas. Aunque *artefactos* como los patrones de diseño de código y plantillas de implementación son considerados por algunos autores como parte de la arquitectura, muchos otros los consideran mecanismos de implementación, que no forman parte de la arquitectura. Aunque idealmente todas las restricciones técnicas, que deben ser acomodadas durante la implementación, habrían sido anticipadas y propuestas en la arquitectura, esto no es realista; inevitablemente habrá algunas consideraciones de implementación imprevistas que causen cambios a la arquitectura.

2.2. *Vistas* de la arquitectura de un sistema

Ya se ha mencionado que para que una arquitectura sea realmente útil y pueda reutilizarse, se debe disponer de una buena descripción de la misma, un conjunto de herramientas adecuado y una metodología que guíe el proceso. Una buena descripción de la arquitectura debe representar a todas las estructuras del sistema. Pero, *¿Cuáles son esas estructuras?* y *¿cómo representarlas?*

2.2.1. Estructuras de un sistema

Existen diversas clasificaciones de las estructuras de un sistema, que dependen de los *puntos de vista* desde los cuales se contempla. Según este enfoque, cada estructura de la arquitectura representa una *vista* de la misma.

Se podría definir una *vista* como una representación del sistema global desde la perspectiva de un conjunto de conceptos relacionados y *punto de vista* como una especificación de las convenciones utilizadas para construir las vistas. Aun cuando existe un consenso general acerca de la necesidad de representar la arquitectura utilizando diferentes vistas, tal consenso desaparece cuando hay que definir cuales son esas vistas. El estándar IEEE-1471-2000 [IEEE Std 1471- 2000] define de alguna manera los puntos de vista que deben tenerse en consideración para describir la arquitectura, pero a un nivel de abstracción tan alto que las vistas pueden seleccionarse o definirse siguiendo criterios muy diferentes. Ejemplos de modelos que definen puntos de vista explícitos son el modelo de 4+1 vistas de Kruchten [Kruchten95], los modelos de Sowa y Zachman [Sowa92] y el modelo propuesto por Hofmeister, Soni y Nord en [Hofmeister00]. De todos estos modelos, el que ha conseguido mayor aceptación es el modelo de 4 + 1 vistas de Krutchen. UML por su parte no define puntos de vista explícitos, pero éstos pueden deducirse de la semántica de sus diagramas, coincidiendo básicamente con las 4+1 vistas de Krutchen (en UML la vista física es referida como *vista de despliegue*, la de desarrollo como *vista de componentes* y la de escenarios es la de *vista de casos de uso*).

El modelo 4+1 de Krutchen

El modelo 4+1 (figura 2.1) define cuatro estructuras o *vistas* que deben ser consideradas en la documentación de todo sistema:

Vista Física. Describe la relación entre el *software* y el *hardware* y sus aspectos de distribución. Básicamente es la descripción de cómo el *software* se ha *trasladado* al *hardware* del sistema. En esta vista se identifican los procesadores del sistema, su interconexión y los componentes *software* que debe ejecutar cada uno de ellos.

Vista Lógica. Describe los requisitos funcionales del sistema. En un sistema orientado a objetos esta vista sería la descomposición del sistema en clases y objetos, la descripción de sus relaciones y la definición de las interfaces proporcionadas por los mismos. De forma más general, esta vista describe los servicios que deben proporcionar los componentes del sistema, bien a otros componentes, bien a un usuario externo.

Vista de Procesos. Describe el comportamiento dinámico del sistema y sus procesos. Cubre aspectos tales como el paralelismo, la concurrencia y la sincronización de tareas. Krutchen enfatiza que estos aspectos capturan los requisitos no funcionales del sistema y menciona explícitamente el rendimiento, que considera no funcional, la disponibilidad, la integridad y la tolerancia a fallos. Este enfoque no es del todo compartido por otros autores [Bass98], que argumentan que dichos requisitos están presentes en todas las vistas y que ciertos requisitos no funcionales están más directamente asociados a otras vistas. De hecho, Krutchen en el mismo artículo [Krutchen95], asocia la reusabilidad y la portabilidad a la vista de desarrollo.

Vista de Desarrollo. Describe la organización estática del *software* en su entorno de desarrollo. El *software* se agrupa en módulos o subsistemas, organizados en diferentes librerías *software*, y cada módulo ofrece a los demás una interfaz bien definida. En esta vista se definen las relaciones de uso (importación y exportación de servicios) entre los diferentes módulos del sistema. La vista de desarrollo tiene una extraordinaria importancia en la planificación del proyecto, ya que es la más habitualmente usada para la asignación de trabajo a los diferentes equipos de desarrollo.

Vista de escenarios. Los escenarios, representan casos de uso significativos del sistema y son la argamasa que une a las diferentes vistas del sistema, ya que muestran al sistema como un todo. La vista de escenarios es redundante o, si se prefiere complementaria, respecto del resto (de ahí el “+1”) y tiene dos propósitos:

- ✓ Servir de herramienta para descubrir elementos arquitectónicos (componentes y relaciones).
- ✓ Validar la arquitectura. Los mismos casos de uso que se han usado para el diseño, aunque más elaborados, sirven de trazas para validar el sistema en sus diferentes fases de desarrollo hasta llegar al producto final.

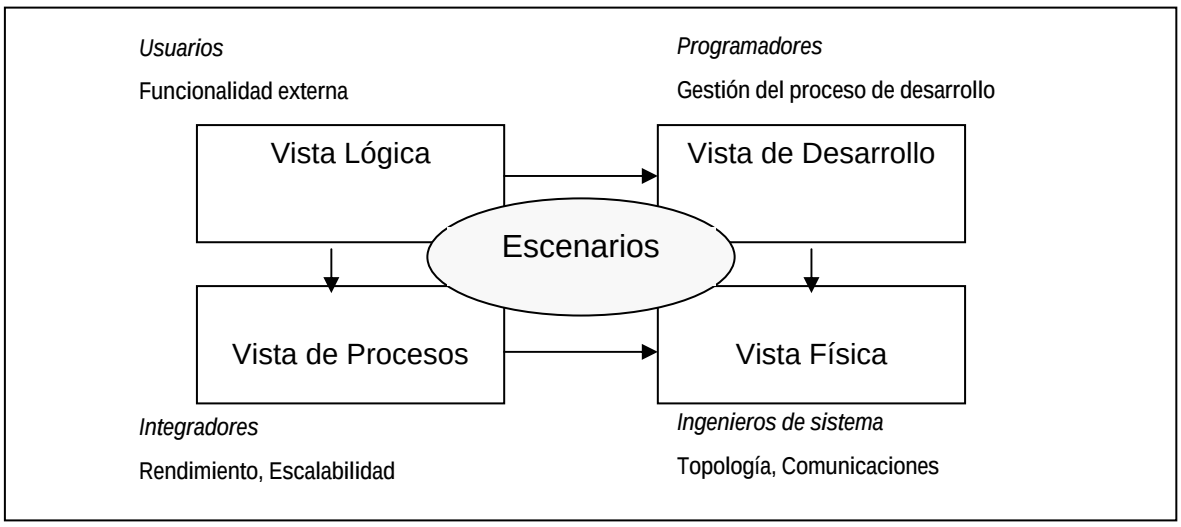


Fig. 2.2.- El modelo 4+1 [Krutchen95]

Las 4 vistas de Hofmeister

Hofmeister, Soni y Nord proponen cuatro vistas de una arquitectura: *conceptual*, de *módulos*, de *ejecución* y de *código*. Esta separación está más cerca de un proceso de desarrollo (ver Capítulo 3) que del concepto de vistas que se acaba de explicar, debido a que cada vista es más bien una fase del desarrollo *top-down* de la arquitectura de un sistema. En el Anexo I se puede ver con detalle en qué consisten estas vistas y su proceso de desarrollo asociado. Se resumen aquí los aspectos básicos:

Vista conceptual. Describe la arquitectura en términos de sus mayores elementos de diseño y las relaciones entre ellos. Los elementos básicos utilizados son los componentes, con puertos que definen los servicios requeridos y proporcionados por los componentes, y los conectores entre puertos, que se encargan de *transportar* los mensajes que definen dichas interacciones.

Vista de módulos. Define la descomposición del sistema y su organización en capas o subsistemas. Una importante consideración en esta vista es limitar el impacto del cambio en el *software* o *hardware*.

Vista de ejecución. Es la vista en tiempo de ejecución del sistema, que define la traducción de módulos en entidades de ejecución y sus atributos, como uso de memoria y asignación de *hardware*.

Vista de código. Es la encargada de capturar cómo los módulos e interfaces se traducen a código fuente, y cómo las entidades de ejecución se convierten en ficheros ejecutables.

Independientemente del modelo adoptado, la descripción del sistema mediante diferentes vistas facilita su diseño y análisis, ya que proporciona una forma muy útil de abordar su complejidad. Sin embargo introduce un nuevo problema: puesto que cada vista es una representación del sistema global, las vistas no son completamente independientes. Las modificaciones realizadas sobre una vista pueden implicar modificaciones en el resto de las vistas. Este problema se complica a medida que aumenta la complejidad del sistema.La integración de las diferentes vistas es un problema no resuelto por la comunidad *software* y es objeto de múltiples trabajos de investigación. El lector interesado puede consultar los trabajos de Egyed y Medvidovic [Egyed99], centrados en la integración de diferentes vistas utilizando UML.

2.2.2. Calidad de la arquitectura. Necesidad de evaluación

Las decisiones arquitectónicas se suelen tomar en las primeras etapas del diseño y afectan a la estructura misma del sistema, condicionando todo el posterior proceso de desarrollo; también son las más difíciles de corregir en caso de error. Así:

- **La arquitectura no fuerza una implementación, pero la condiciona**, ya que deberá adaptarse al tipo de componentes y a las relaciones que entre ellos se hayan definido en la arquitectura.
- **La arquitectura condiciona la organización y gestión del proyecto**. La arquitectura define una descomposición estructural del sistema en subsistemas con interfaces bien definidas. Esto hace posible la división del proyecto en paquetes de trabajo, que se asignarán a diferentes equipos de desarrollo, facilitando, pero también condicionando, la planificación y la asignación de recursos. El reverso de la moneda es que una modificación de la arquitectura puede producir un enorme impacto sobre la gestión del proyecto, pues supone la reestructuración de los grupos de trabajo y la reasignación de recursos y responsabilidades.
- **La arquitectura influye en los atributos de calidad del sistema, mejorando unos a costa de empeorar otros**. Los atributos de calidad no existen de forma independiente, sino que interaccionan entre sí [Boehm96], [Kazman00]. La arquitectura adoptada puede verse como el consenso final sobre los atributos de calidad del sistema al que llegan todas las partes implicadas.
- **Las decisiones arquitecturales ocurren en las primeras etapas de la vida de un proyecto y afectan a la estructura misma del sistema, condicionando todo el posterior proceso de desarrollo**. La detección precoz de errores es uno de los caballos de batalla de la informática. El coste de corregir un error se hace tanto mayor cuanto más tarde se detecta en el ciclo de vida de un proyecto. Por ello es necesario disponer de métodos fiables para evaluar la arquitectura y aplicarlos tan pronto como sea posible. En este sentido han ido encaminados los trabajos de, entre otros, Boehm (WinWin y QARCC [Boehm96]) y del SEI (Métodos de análisis de arquitecturas SAAM [Bass98] y ATAM [Kazman00]).

Obviamente, la consecución de los atributos de calidad de un sistema no depende sólo de la arquitectura. Es también función de otras muchas variables. La gestión del proyecto, la elección de lenguajes de programación y de algoritmos apropiados y la calidad de la implementación influyen también de forma decisiva en el resultado final. Pero aunque una buena arquitectura no sea condición suficiente para asegurar la calidad del sistema, es sin ninguna duda una condición necesaria.

Las arquitecturas deben evaluarse, confrontándolas con los atributos de calidad que dependen de ellas. Pero *¿cómo medir la calidad de una arquitectura?* Los atributos de calidad se han de tener muy en cuenta para responder a esta pregunta.

2.2.2.1. Atributos de calidad

Las arquitecturas se evalúan con relación a los atributos de calidad que debe poseer el sistema y los requisitos impuestos a estos atributos. Aunque existen definiciones normalizadas de cada uno de estos atributos[7], tales definiciones son demasiado abstractas como para servir de guía en el análisis de *una* arquitectura. Los atributos de calidad no existen como conceptos aislados, sino que tienen sentido dentro de un contexto dado ([Boehm96], [Bass98], [Kazman00]).

Por otro lado, los atributos de calidad no son independientes entre ellos, sino que interaccionan entre sí a través de las relaciones estructurales impuestas por la arquitectura. La mejora de unos, frecuentemente solo puede lograrse a costa de empeorar otros. Por ello es necesario llegar a

[7] [IEEE-610.12]

compromisos de diseño (del inglés *tradeoffs*). Hay que optar por unos atributos a costa de otros. Como dice Boehm [Boehm96]

"...muchos proyectos, a pesar de tener la funcionalidad e interfaces bien especificadas, fracasan por no tener bien definidos los requisitos de sus atributos de calidad. Encontrar el balance adecuado entre todos los atributos de calidad es un paso importante para lograr el éxito del producto. Para ello, es necesario identificar los conflictos entre los atributos deseados y alcanzar un balance satisfactorio...".

El consenso acerca de los atributos de un sistema involucra a todas las partes implicadas (estas partes implicadas se denominan *stakeholders* en la terminología de Bass [Bass98] y de IEEE [IEEE Std-1471-2000]). Clientes, usuarios, ingenieros, todos ellos deben participar en la especificación de los requisitos del sistema. Obviamente, los intereses de cada parte son diferentes, a menudo contradictorios, e incluyen factores sociales y de negocio que se unen a los técnicos (ver **Fig. 2.3**).

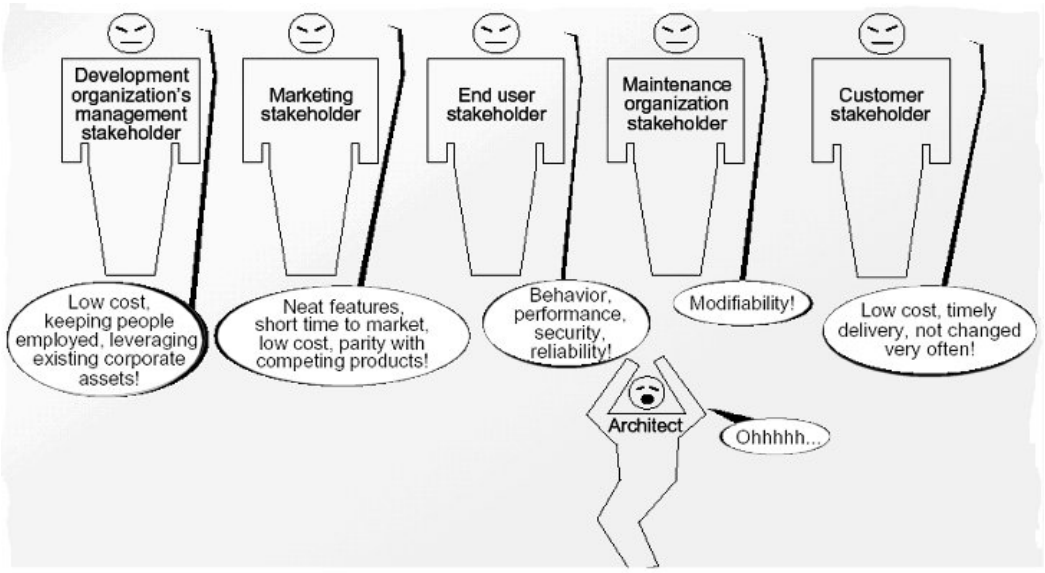


Fig. 2.3.- Influencia de los *stakeholders* en el arquitecto [Bass98]

Así, será necesario considerar factores como el coste, los tiempos de vida y de comercialización del producto, el uso de patentes, las estrategias empresariales, etc. Según [Bass98] y [Boehm96] estos atributos tienen al menos la misma importancia que los técnicos a la hora de alcanzar el *balance* que mencionaba Boehm. Las preguntas que debemos hacernos ahora son del estilo: *¿qué es más deseable, un sistema plenamente portable o comercializar el producto antes de 6 meses?* Las relaciones entre los factores técnicos, sociales y de negocio son complejas y el consenso no es una meta fácil. En el proceso de negociación cada parte implicada tendrá que renunciar a una parte de sus requisitos en aras del éxito del sistema como un todo. En [Bass98] se define el ABC (*Architecture Business Cycle*), para explicar estas interacciones. Boehm propone el sistema QARCC [Boehm96] como una herramienta basada en conocimiento (un sistema experto) para ayudar a usuarios y desarrolladores a expresar sus requisitos, identificar los conflictos y llegar a un consenso final sobre las *calidades* del sistema. Por último, ATAM [Kazman00] trata de establecer un marco de razonamiento en el que dichas interacciones puedan ser descubiertas, puestas en relación con los componentes y conectores definidos en la arquitectura y finalmente resueltas.

2.2.2.2. Clasificación de los atributos de calidad

Los requisitos o atributos de calidad pueden clasificarse según la siguiente taxonomía:

- **Requisitos técnicos del sistema**, impuestos por las especificaciones que debe cumplir el sistema final.

- **Requisitos de negocio**, impuestos por la estrategia empresarial en relación con el producto que se pretende desarrollar.
- **Requisitos propios de la arquitectura** o requisitos que debe cumplir toda arquitectura, independientemente de los requisitos técnicos y de negocio. Los requisitos propios de la arquitectura son la *integridad conceptual* (coherencia del diseño), la *corrección* (capacidad de la arquitectura para cumplir sus requisitos), la *compleción* (la capacidad de cubrir todos los requisitos) y la *capacidad de realización* (se refiere a la dificultad de implementación del sistema que supone utilizar una determinada arquitectura).

Tradicionalmente, los atributos o requisitos técnicos de un sistema se han clasificado en dos categorías:

- Requisitos funcionales
- Requisitos no funcionales o de calidad

Aunque ésta es la clasificación que sigue este trabajo de tesis, algunos autores la rechazan por dos razones. En primer lugar porque es ambigua (¿dónde está la frontera entre lo funcional y lo no funcional?). Y en segundo lugar, porque asume implícitamente que ambos tipos de atributos pueden abordarse y alcanzarse por separado, relegando el cumplimiento de los requisitos no funcionales a etapas demasiado tardías del ciclo de desarrollo.

En [Bass98] se propone una clasificación alternativa de los atributos del sistema entre:

- Atributos observables en tiempo de ejecución y
- Atributos no observables en tiempo de ejecución

Los atributos observables en tiempo de ejecución se corresponden aproximadamente con los requisitos funcionales y los no observables con los requisitos no funcionales, pero ahora el argumento de clasificación no asume que ambos puedan abordarse por separado, sino simplemente diferentes métodos de medida. El cumplimiento de los atributos observables en tiempo de ejecución no implica que se cumplan los no observables, y viceversa, pero están estrechamente relacionados. Los atributos de calidad, funcionales o no, dependen de las relaciones estructurales impuestas por la arquitectura. La elección de un determinado patrón o estilo arquitectónico favorece o perjudica la obtención de ciertos atributos. Es decir, los atributos interaccionan entre sí a través de las estructuras del sistema, de forma que la optimización de uno a menudo sólo se logra a costa de empeorar otros.

Los atributos de calidad deben considerarse tanto en las fases de diseño como en las de implementación del sistema, sin embargo no todas las calidades se manifiestan ni se consiguen de igual manera en cada una de estas fases. La arquitectura es crítica para la obtención de la mayoría de estos atributos, que por tanto deben ser evaluados en este nivel, mientras que otros dependen más de los algoritmos utilizados y del estilo y calidad de la implementación. Por supuesto, muchos de los atributos dependen de ambas cosas.

2.3. Tendencias

Bass, en 1998 hacía la siguiente reflexión [Bass98]:

"Las arquitecturas ponen especial énfasis sobre los componentes individuales y se concentran en la organización de los componentes y sus interacciones; y es esta abstracción, lejos de la profundización en los componentes individuales, lo que hace posible tal interoperabilidad. Se podría decir que éste es el punto en el que nos encontramos actualmente. No hay razón para pensar que esta tendencia hacia las abstracciones más grandes y poderosas no continúe en el futuro".

No se puede decir que se haya avanzado mucho en la consecución de abstracciones más grades y poderosas. Las principales tendencias siguen siendo el desarrollo de *software* basado en componentes y los *frameworks* orientados a objeto.

2.3.1. Desarrollo de *software* basado en componentes

A medida que el tamaño y la complejidad de los sistemas intensivos en *software* han ido aumentando, ha surgido la necesidad de poner cada vez más atención en su diseño y especificación. Por otro lado, el desarrollo de sistemas industriales fiables y que se adapten a las especificaciones supone costes elevados, siendo muchas veces el *software* repetitivo. Por ello, actualmente, muchas organizaciones implementan tales sistemas *software* basándose en **componentes reutilizables**. De esta forma, se reducen costes, se logra acortar el ciclo de desarrollo de los productos, y los sistemas desarrollados requieren menos tiempo de especificación, diseño, pruebas y mantenimiento del producto final. De aquí, que la ingeniería de *software* basada en componentes (CBSE) esté ganando tremendo interés en la industria [Brown98][García02].

El desarrollo basado en componentes o CBD (de sus siglas en inglés: *Component Based Development*) ofrece una forma flexible para el desarrollo eficiente de soluciones *software* a partir de componentes reutilizables. En el marco de trabajo que ofrece el CBD las aplicaciones se diseñan a partir de unos componentes que cumplen una serie de especificaciones predefinidas que pueden ensamblarse para dar lugar a aplicaciones completas.

El CBD toma muchas de los conceptos de la programación orientada a objetos, pero carece del grado de madurez que ha alcanzado ésta. Para empezar, mientras que toda la comunidad *software* está de acuerdo en el significado de *objeto*, existen muchas definiciones de componente. Algunas de ellas son:

"Un componente software es una unidad de composición que ofrece interfaces explícitamente especificadas, requiere de unas interfaces e incluye unas interfaces de configuración, incluyendo además atributos de calidad" [Bosch00].

"Un componente es una parte modular, distribuible e intercambiable de un sistema, que encapsula su implementación y presenta un conjunto de interfaces. Un componente está especificado por uno o más clasificadores (clases de implementación) que residen en él y puede ser implementado por uno o más elementos software (ficheros binarios, ejecutables o de script)" [OMG00]

"Un componente es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en espacio y en tiempo" [Szypersk97].

Tal vez la clave de la reutilización esté en la granularidad con la que se definen los componentes. Un componente tiene tanta mayor probabilidad de ser reutilizado cuanto más general es el trabajo que realiza y más concretas sus responsabilidades. Así, es más probable que sea reutilizado un componente que *realice*[8] una lista enlazada, que uno que implemente un *encoder* y más probable que sea reutilizado el *encoder* que el controlador de una articulación. Cuanto más específico sea el dominio de aplicación del componente menos usuarios potenciales habrá. Cuanto más responsabilidades tenga un componente mayor será su necesidad de servicios procedentes de otros componentes y de la infraestructura, mayores sus suposiciones arquitectónicas y, por tanto, mayores sus probabilidades de entrar en conflicto con la arquitectura en la que se pretende integrar. Sólo en los dominios en los que se ha alcanzado una gran madurez y existe un consenso acerca de la división funcional del sistema es posible la creación de un mercado de componentes *software*.

2.3.1.1. Modelos de componentes.

De acuerdo con las definiciones ofrecidas en los párrafos anteriores, los componentes constituyen unidades de composición reemplazables y configurables a partir de los cuales pueden construirse

[8] Realizar en el sentido de realizar una interfaz, es decir ofrecer unos servicios a través de unas operaciones perfectamente definidas.

sistemas como mecanos. La pregunta es, *¿Quién o qué determina como definir, implementar, ensamblar e instalar dichos componentes?* La respuesta la ofrece la definición de [Bachmann00]:

*"Un **componente** es una implementación software que puede ser ejecutada en un dispositivo lógico o físico. Un componente implementa una o más **interfaces**. Estas interfaces reflejan las obligaciones del componente que se describen como **contratos**. Las obligaciones contractuales de los componentes aseguran que los mismos podrán ser desarrollados de forma independiente, pues obedecen ciertas reglas que determinan cómo interaccionan y cómo pueden ser distribuidos (integrados) en entornos estándares"*.

Un sistema basado en componentes se basa en un pequeño número de tipos de componente, cada uno de los cuales desempeña un papel especializado en el sistema y está descrito, como ya se ha dicho, por una interfaz.

Un **modelo de componentes** es un conjunto de tipos de componente, sus interfaces y, adicionalmente, una especificación del patrón o patrones de interacción entre tipos de componentes.

Un ***framework* de componentes** ofrece una serie de servicios en tiempo de ejecución para dar soporte y reforzar el modelo de componentes. En muchos aspectos, los *frameworks* son como sistemas operativos de propósito especial, aunque operando a un nivel de mucha mayor abstracción." [Bachmann00]

Y no hay mucho más que añadir, salvo que algunos autores no distinguen entre modelo de componentes y *framework* de componentes.

Puesto que actualmente hay varios modelos de componentes, realizar una programación basada en componentes significa tener que optar por uno de dichos modelos. Cuanto más estandarizado y estable sea el modelo, mayores posibilidades habrá para desarrollar componentes para terceros o para utilizar componentes desarrollados por terceros. Dada la falta de estándares comunes, la adopción de un modelo de componentes supone hoy por hoy ligarse a una arquitectura y a una tecnología. Todo componente que sea conforme con el modelo elegido podrá integrarse siguiendo procedimientos estándar definidos en la especificación del modelo. Todo el que no lo siga necesitará de procedimientos de integración específicos, cuya dificultad dependerá del grado de compatibilidad entre los modelos[9]. Entre los modelos de componentes actualmente disponibles cabe destacar los siguientes:

- Los modelos de componentes de Microsoft:
  - El modelo COM (*Component Object Model*)
  - El modelo DCOM (*Distributed Component Object Model*)
  - El modelo .NET
- El modelo CCM (*Corba Component Model*)
- El modelo EJB (*Enterprise JavaBeans*)

2.3.1.2. Componentes y objetos.

El concepto de *componente* es mucho más amplio que el de *objeto* en el sentido de que un componente no tiene porqué ser un objeto, ni implementarse como tal, pero es mucho más restrictivo en el sentido de que un componente debe cumplir muchas más condiciones que un objeto.

---

[9] La interoperabilidad entre aplicaciones Java-RMI y CORBA es posible en muchos casos. La especificación del modelo de componentes CORBA (CCM) tiene como requisito explícito la interoperabilidad con EEJB. Además, el OMG está trabajando para conseguir interoperabilidad con aplicaciones basadas en tecnología DCOM de Microsoft.

Ningún lenguaje de programación incorpora abstracciones suficientes para soportar el concepto de componente, ni siquiera los lenguajes orientados a objetos[10]. En particular, la capacidad de los lenguajes de programación para definir interfaces es muy limitada.

Un objeto es la encapsulación de un conjunto de datos y un conjunto de métodos que realizan operaciones sobre esos datos o proporcionan algún servicio a otros objetos a través de las invocaciones de sus métodos.

Es decir, los servicios que proporciona un objeto se definen explícitamente, pero no los servicios que a su vez necesita para proporcionarlos. Algunas características importantes de los componentes como la calidad de servicio ofrecida o su comportamiento temporal no pueden expresarse de forma directa utilizando objetos. Esta falta de correspondencia entre objetos y componentes obscurece la implementación de los componentes, ya que no expresa de forma directa los conceptos de diseño. Mientras no se definan lenguajes de programación que soporten estos conceptos es posible acudir a patrones de diseño [Andrade99].

Algunos lenguajes formales incorporan algunos de los aspectos mencionados en el párrafo anterior, pero presentan otros inconvenientes, como el grado de destreza que requiere su uso y la necesidad de herramientas para obtener código fuente o ejecutables a partir de ellos. La programación orientada a aspectos [Kickzales97] ofrece algunas claves para modelar e incorporar (o eliminar) incrementalmente algunos de estos requisitos, pero también necesita tiempo para madurar y, sobre todo, para ser incorporada de forma masiva en la producción de *software*.

2.3.2. *Frameworks* orientados a objeto

Según [Gamma95] un *framework* es un conjunto de clases que cooperan y forman un diseño reutilizable para un tipo específico de *software*. Un *framework* ofrece una guía arquitectónica partiendo el diseño en clases abstractas y defiendo sus responsabilidades y sus colaboraciones. Un desarrollador personaliza el *framework* para una aplicación particular mediante herencia y composición de instancias de las clases del *framework* .

Según [Fayad97] los *frameworks* pueden clasificarse por su alcance o ámbito de aplicación y por sus mecanismos de extensión. En [García02] se puede encontrar esta clasificación y otros aspectos comentados de los *frameworks*.

Los principales beneficios de los *frameworks* son:

- Modularidad: Los *frameworks* favorecen la modularidad mediante la encapsulación de los detalles de implementación detrás de interfaces estables.
- Reusabilidad: Mediante la definición de componentes genéricos que pueden ser utilizados para crear nuevas aplicaciones. La reutilización de los componentes de un *framework* puede mejorar sustancialmente la productividad y los atributos de calidad del *software*.
- Extensibilidad: Proporcionando mecanismos explícitos que permitan a las aplicaciones extender sus interfaces estables. La extensibilidad del *framework* es esencial para asegurar el rápido desarrollo de nuevas aplicaciones, características y servicios.

Problemas de los frameworks

Los *frameworks* son el máximo exponente de la reutilización en las tecnologías orientadas a objetos, sin embargo presentan también una serie de problemas o inconvenientes.

Los *frameworks* requieren un gran esfuerzo de desarrollo. Como se afirma en [Roberts96] el desarrollo de *frameworks* reutilizables no ocurre simplemente sentándose a pensar acerca del dominio del problema. Nadie tiene la capacidad suficiente para dar con las abstracciones correctas. Los

[10] *Eiffiel* [Meyer98] incorpora muy buenos paradigmas en ese sentido, pero no es un lenguaje de uso común en la industria.

frameworks son el resultado de muchos años de trabajo, pues en ellos se resume la experiencia adquirida en muchos desarrollos.

Los desarrolladores de *frameworks* se enfrentan a muchos compromisos de diseño. Uno de los más cruciales es determinar que componentes del *framework* deben ser variables y cuáles deben ser estables. Una insuficiente variabilidad dificulta la configuración o extensión del *framework* impidiendo que pueda acomodarse a los requisitos de diversas aplicaciones. De la misma manera, una insuficiente estabilidad hace difícil a los usuarios entender el *framework*. Un *framework* debe ser lo suficientemente sencillo para ser aprendido y utilizado, y al mismo tiempo debe proporcionar las suficientes características y prestaciones para ser usado eficientemente. En cualquier caso, es necesario un conocimiento exhaustivo del dominio para el que está orientado el *framework*, el cual debe incluir una teoría del dominio del problema y es siempre el resultado del análisis del dominio bien explícito y formal, bien implícito e informal.

La mantenibilidad de las aplicaciones se dificulta en algunos casos:

- ✓ Como los *frameworks* inevitablemente evolucionan, las aplicaciones que los usan deben evolucionar con ellos.
- ✓ La validación y la corrección de defectos se hace más difícil. Los componentes genéricos son difíciles de evaluar en abstracto. La inversión del control y la falta de un flujo de control explícito dificultan la trazabilidad de la aplicación y puede ser difícil distinguir entre los errores introducidos por el programador y los debidos al *framework*.
- ✓ Por último, los distintos *frameworks* suelen ser incompatibles entre sí, de forma que optar por uno significa ligarse a un cierto tipo de tecnología.

2.3.3. Líneas de producto

Uno de los enfoques más prometedores para conseguir la reutilización de componentes *software* son las líneas de producto [García02]. Las líneas de producto pueden definirse como un conjunto de productos estrechamente relacionados por su funcionalidad y cuyo objetivo es satisfacer las necesidades de un cierto segmento del mercado. La estrategia de las líneas de producto es capturar las partes comunes entre productos y tratarlas como entidades de primera clase. En la terminología técnica de las líneas de producto a estas partes comunes se las denomina *core assets* o simplemente *assets*[11]. Al contrario que en el desarrollo de un único producto, el trabajo se divide entre la ingeniería de dominio (*core assets*) y la ingeniería de aplicación (productos individuales). Los *core-assets* se desarrollan tomando en consideración todos los sistemas. La ingeniería de aplicación consiste en la configuración y ensamblado de los *assets* y en el desarrollo e integración de componentes específicos de producto. Los *assets* principales son la *arquitectura base* de la línea de producto, el conjunto de componentes reutilizables y los productos resultantes.

Algunas de las definiciones propuestas para las líneas de producto son las siguientes:

"Una línea de productos consiste en una arquitectura de línea de productos y en un conjunto de elementos software reutilizables que han sido diseñados para su incorporación en dicha arquitectura. Adicionalmente, la línea de productos incluye los productos que han sido desarrollados utilizando los "assets" mencionados." [Bosch00].

"Una línea de productos software es un conjunto de sistemas intensivos en software que comparten un conjunto común y gestionado de características que satisfacen las necesidades específicas de un particular segmento de mercado o misión y que son desarrollados a partir de un conjunto de "assets" comunes de una forma determinada." [Clements00].

---

[11] Dada la dificultad de encontrar un término castellano para asset que sea comúnmente aceptado (activo, bien, disponible), se utiliza el término en inglés.

Los componentes de una línea de producto se desarrollan considerando todos los productos de la línea. La motivación es compartir los costes de desarrollo y mantenimiento entre todos los productos. La restricción es que la evolución de los productos individuales debe ser conforme con la evolución de la arquitectura y viceversa. Las líneas de producto exigen por tanto una gran capacidad organizativa para gestionar su desarrollo, uso y mantenimiento. En cualquier caso, las líneas de producto no aparecen de forma accidental, sino que requieren un esfuerzo considerable por parte de la organización interesada.

Capítulo 3

Notaciones y Metodologías de Desarrollo de Arquitecturas

3.1. Introducción

Si en el capítulo anterior quedó clara la importancia de la arquitectura de un sistema, este capítulo pretende, por un lado, justificar la importancia de la descripción adecuada de la arquitectura de forma que se facilite su comprensión y utilidad, utilizando para ello el estado de la técnica en métodos de descripción de arquitecturas y ciclos de desarrollo, y por otro lado, mostrar el enfoque metodológico adoptado para esta tesis.

En la práctica, la mayoría de descripciones arquitectónicas son documentos informales, basadas en diagramas de cajas y líneas utilizados con un propósito explicatorio. Las convenciones visuales utilizadas suelen ser específicas del proyecto y propias de cada "arquitecto". Como resultado de ello, las descripciones arquitectónicas son ambiguas, no pudiéndose analizar ni su consistencia ni su compleción, ni ser soportadas por herramienta.

Para mejorar esta situación, un gran número de investigadores sugiere el uso de notaciones más estandarizadas o incluso formales, para la descripción de arquitecturas. En este sentido hay dos fuentes principales de recomendaciones para descripción arquitectónica [Garlan00]:

- Lenguajes de descripción de arquitecturas (ADLs)
- Notaciones de modelado basadas en objetos (en especial UML)

Para profundizar en estos conceptos, el capítulo se inicia con un breve estado de la técnica en métodos de descripción de arquitecturas, con especial interés en la adaptación de UML para poder describir arquitecturas. Además de un lenguaje semi-formal como puede ser UML, es necesario adoptar un método de desarrollo riguroso en el diseño de sistemas en general y de arquitecturas en particular. En este sentido se presentarán diversos métodos existentes (ABD [Bachmann00a], USDP [Jacobson00], COMET [Gomaa00], ROPES [Douglass00], 4 Vistas de Hofmeister[1] [Hofmeister00]) y se justificará

[1] En adelante 4-V.H.

la utilización de ABD, 4-V.H. y COMET en el enfoque metodológico de desarrollo de los trabajos de esta Tesis Doctoral.

3.2. Métodos de descripción de arquitecturas

Desafortunadamente, las principales razones que han hecho que la arquitectura software sea todavía una disciplina por desarrollar dentro de la ingeniería del software, ha sido una falta de [Hofmeister00]:

- Métodos estandarizados para representar la arquitectura.
- Métodos de análisis para predecir si una arquitectura dará como resultado una implementación que cumpla los requisitos establecidos.

Del artículo de Garlan [Garlan00] y de otros muchos que intentan conciliar las necesidades de descripción de arquitecturas con las representaciones orientadas a objetos, como las ofrecidas por UML [Hofmeister99], [Medvidovich99], [Egyed99], [Smolander01] y [Kande00], se puede deducir que **no existe un método único que cumpla todas las expectativas puestas por los diseñadores de arquitecturas para representar claramente las estructuras de un sistema**. Quizás el hecho de utilizar UML sea la tendencia que más se imponga hoy en día, además de por las facilidades que ofrece, por ser ampliamente utilizado en la comunidad del diseño de software y no sólo de software, por los motivos que se exponen en el punto 3.3.

Durante estos últimos años se han propuesto varios lenguajes de descripción de arquitecturas (ver punto 3.2.1). Mientras que estos lenguajes y sus herramientas CASE asociadas difieren en muchos detalles, hay un consenso general acerca de los principales ingredientes de lo que debe ser una descripción arquitectónica [Garlan00]:

- **Componentes.** Representan los elementos computacionales y de almacenamiento de datos de un sistema. Intuitivamente, se corresponderían con las típicas *"cajas"* de una descripción arquitectónica de cajas y líneas. Los componentes pueden tener múltiples interfaces (que se pueden denominar *puertos*[2] - ver punto 3.3.3.2), cada interfaz define un punto de interacción entre un componente y su entorno. Un componente puede tener múltiples puertos del mismo tipo. Típicos ejemplos de componentes son clientes, servidores, filtros, bases de datos, etc.
- **Conectores.** Representan las interacciones entre los componentes de una arquitectura. Se corresponden con las *"líneas"* de los típicos diagramas de *cajas y líneas*. En tiempo de ejecución, los conectores permiten la comunicación y actividades de coordinación entre componentes. Como ejemplos podemos citar simples formas de interacción, como llamadas a procedimiento, tuberías y eventos, y también interacciones más complejas, como protocolos cliente-servidor, etc.
- **Sistemas.** Representan gráficos de componentes y conectores. En general, los sistemas pueden ser jerárquicos: unos componentes y conectores pueden representar subsistemas que tienen una arquitectura interna (representaciones). Cuando un sistema o parte de un sistema tiene una representación, también es necesario explicar la correspondencia entre los interfaces internos y externos. Estos elementos de correspondencia se denominan *bindings*.
- **Propiedades.** Representan información adicional acerca de las partes de una descripción arquitectónica. Típicamente se usan para representar aspectos extra-funcionales de un diseño arquitectónico.
- **Tipos y estilos.** Representan familias de sistemas relacionados. Un estilo arquitectónico típicamente define un vocabulario de tipos de elementos de diseño como una serie de

[2] Aunque *interfaz* y *puerto* son conceptos similares, no son idénticos. Una *interfaz* expone una serie de operaciones que pueden ser invocadas por el entorno de un componente. Un *puerto*, tal como se describe en un ADL, a menudo incluye tanto los servicios ***proporcionados*** por un componente, como los que ***requiere*** de su entorno.

componentes, conectores, puertos, y propiedades junto con reglas para componer instancias de esos tipos.

Estos *ingredientes* compondrán una descripción arquitectónica, recordando que la composición de los mismos para describir la estructura o estructuras de un sistema, se puede visualizar de distintas maneras, como se vio en el punto 2.2.- *Vistas de la arquitectura de un sistema*, del Capítulo 2.

3.2.1. El estado de la técnica en métodos de descripción de arquitecturas

Como se dijo en la introducción, para hacer más rigurosa la tarea de descripción de arquitecturas, muchos autores sugieren el uso de notaciones más estandarizadas o incluso formales, siguiendo principalmente dos fuentes de recomendaciones [Garlan00]:

- **Lenguajes de descripción de arquitecturas.** Un ADL (de sus siglas en inglés ADL: *Architecture Description Language*) concentra su atención en la estructura a alto nivel del sistema en lugar de en los detalles de implementación [Clements96], [Medvidovic99]. Propuestos por la comunidad de investigadores en arquitectura software son lenguajes matemáticos formales, específicamente diseñados para representar arquitecturas software tanto para un dominio particular como para fines de propósito general. Estos lenguajes han madurado en los últimos años. La mayoría proporcionan herramientas que soportan muchos aspectos del diseño y análisis arquitectónico, como la edición gráfica, generación de código, monitorización en *run-time*, detección de anomalías y análisis de prestaciones.
- **Notaciones de modelado basadas en objetos** (en especial UML). Es la otra fuente de recomendaciones, proveniente de la comunidad de orientación a objetos. Este tipo de notaciones han tenido un considerable éxito, proporcionando un mecanismo natural para representar entidades de un dominio y sus relaciones. Incluso para sistemas que no se implementan con orientación a objetos o incluso que nada tienen que ver con el software (representación del hardware [Axelsson00], [Bahill03] o la estructura de negocio de una organización), esta notación proporciona una manera natural de representar la estructura y comportamiento de los sistemas mediante diagramas de clases, de actividad, de secuencia, etc.

En [Clements96] se exponen una serie de requisitos que debe cumplir un buen ADL, entre ellas, podemos destacar aquí las siguientes:

- ✓ Ser capaz de describir todas las estructuras del sistema, tanto las estáticas como las dinámicas.
- ✓ Estar libre de ambigüedades y disponer de un conjunto de símbolos, componentes y conectores, con una semántica lo suficientemente rica como para capturar todas las propiedades externamente visibles de los componentes de cada estructura.
- ✓ Debe incluir reglas que aseguren la consistencia y completitud de la arquitectura para servir de soporte a los procesos de diseño, refinado y validación de la misma.
- ✓ Debe ser capaz de representar, directa o indirectamente, diferentes estilos arquitectónicos.

El propio autor pasa revista a los ADLs existentes en aquel momento, concluyendo que ninguno de ellos satisfacía por completo los requisitos definidos.

Un estudio más reciente sobre el estado del arte en ADLs lo ofrecen Medvidovic y Taylor en un extenso artículo del año 1999 [Medvidovic99], en el que se amplían los estudios realizados por Clements. Ambos autores definen lo que a su juicio es un ADL, describen sus características y realizan una comparativa bastante exhaustiva de los ADLs existentes en la actualidad, en concreto compara los lenguajes Aesop, ArTek, C2, Darwin, LILEANNA, MetaH, Rapide, SADL, UniCon, Weaves y Wright . El artículo ofrece un excelente estado de arte sobre ADLs, aporta un *framewok* para poder

comparar dichos lenguajes, estableciendo para ello las principales características a tener en cuenta en los mismos, y cita todas las referencias significativas sobre el tema.

Los ADLs existentes, en general, son **difíciles de comprender** y no se integran bien con las prácticas de desarrollo de software actuales. Más aún, los ADLs a menudo toman en cuenta sólo una perspectiva particular, desde la cual el arquitecto tiene que modelar todos los aspectos clave del sistema software.

En contraste con este escenario, UML es un lenguaje de modelado general orientado a objetos que proporciona técnicas avanzadas y notaciones que soportan todo el ciclo de vida del modelando de un sistema, desde el análisis de requisitos a la implementación y soporta múltiples vistas. David Garlan, en [Garlan00], se plantea si las notaciones basadas en objetos, como UML, son apropiadas para descripciones arquitectónicas. Muchos autores argumentan que la notación orientada a objetos si es apropiada, justificando que las capacidades actuales son precisamente las necesarias, y ha habido diferentes propuestas para mostrar como los conceptos de ADLs se puede traducir directamente a una notación como UML mediante extensiones del lenguaje o *profiles* [Hofmeister99], [Medvidovic00]. Sin embargo, hasta ahora no se ha aceptado universalmente ninguno de estos métodos, ni siquiera se ha incluido en el estándar de UML, de forma que podamos escoger, sin lugar a dudas, una notación única para representar arquitecturas. En el momento de presentar este trabajo de tesis, el panorama parece que va a cambiar positivamente, con la incorporación en el estándar **UML 2.0.** de conceptos como componentes, conectores y puertos que permiten la representación arquitectónica de sistemas [Björkander03], [Kobryn03], [Selic03]. Este último estándar no se ha incorporado finalmente a la tesis porque en el momento de comenzar la redacción de la misma todavía no se había publicado y no existían herramientas CASE que lo soportaran, por ello en los próximos puntos no se considera UML 2.0.como solución a los problemas expuestos.

3.3. El Lenguaje Unificado de Modelado (UML)

UML es un lenguaje gráfico con sintaxis y semántica semiformal especificadas mediante un metamodelo, texto descriptivo informal y restricciones [OMG00]. En 1997, el *Object Management Group* (OMG) publicó la primera versión del Lenguaje Unificado de Modelado [Booch97]. Desarrollado inicialmente por Jim Rumbaugh, Ivar Jacobson y Grady Booch, que originalmente tenía sus propios métodos (OMT, OOSE, y Booch), y se unieron para proporcionar un estándar abierto. El Lenguaje Unificado de Modelado (*Unified Modeling Language* [UML99a]), como su propio nombre indica, unifica las anteriores notaciones de modelado orientadas a objeto [Booch94], [Rumbaugh91], [Jacobson92] y [Harel90] en un marco de trabajo común, y se ha convertido en poco tiempo en una notación estándar no sólo para la comunidad de investigadores en ingeniería del software, sino también para la industria. Cada vez hay más artículos, documentos técnicos, libros, etc., que hablan de UML y exponen experiencias de utilización de este lenguaje en el análisis, diseño y organización de sistemas complejos.

3.3.1. Ventajas de utilizar UML

Unos de los propósitos de UML era proporcionar a la comunidad de desarrolladores de software un lenguaje de diseño común y estable para que, entre otras cosas, pudieran leer y diseminar la estructura de un sistema y los planes de diseño. Aunque su propósito original era para el diseño detallado, su habilidad para describir elementos y las relaciones entre ellos, lo hace potencialmente aplicable a un campo mucho más amplio, incluida la expresión de la arquitectura de un sistema.

Se podrían enumerar las características y ventajas de utilizar UML:

- Lenguaje de modelado con una definición de su sintaxis y semántica semi-formal.
- UML incorpora y unifica lo mejor de las notaciones de Booch [Booch94], Rumbaugh [Rumbaugh91], Jacobsson [Jacobson92] y Harel [Harel90] y tiene riqueza semántica suficiente para describir todas las vistas de la arquitectura.

- Grande, con una serie de construcciones predefinidas muy útiles.
- Extensible (el usuario del lenguaje puede extender o especializar los aspectos básicos de UML mediante estereotipos, restricciones y valores etiquetados[3]).
- Expresivo (independiente del lenguaje de programación – concepto – hardware, etc.).
- El uso de una notación única ayuda a mantener la coherencia entre las diferentes vistas del sistema y facilita su trazabilidad.
- Independiente del proceso de desarrollo.
- Soporta conceptos de algo nivel (colaboraciones, *frameworks*, patrones, componentes).
- Su difusión, utilización y aceptación es cada vez más grande.
- Es un estándar que sigue unas revisiones.
- Hay gran soporte de herramientas para el lenguaje, tanto comerciales como de libre distribución, Rational Rose, Rhapsody, etc .

Dado que hoy en día UML es ampliamente conocido, y no es objetivo de esta tesis el describir detalladamente los mecanismos que ofrece este lenguaje, se remite al lector a la bibliografía para conocer dichos mecanismos. En [UML99a] y [UML99b] encontrará una especificación detallada del lenguaje y una guía del usuario de UML. Para una somera introducción al lenguaje se puede utilizar [Schmuller00].

En lugar de hacer esta descripción detallada de UML, los próximos puntos expondrán los medios que ofrece UML para expresar arquitecturas y la posibilidad de extensión del lenguaje para soportar dichas descripciones.

3.3.2. UML para describir arquitecturas

Como lenguaje de propósito general, UML no proporciona directamente ciertas construcciones relacionadas con el modelado de arquitecturas software, definidas en la mayoría de ADLs, y que ya se mencionaron en el punto 3.2, principalmente, *componentes, conectores, sistemas, propiedades, restricciones, tipos y estilos arquitectónicos.*

Ante este panorama, muchos autores [Garlan00], [Medvidovic01], [Hofmeister99], [Kande00] proponen solucionar de diversas formas las limitaciones de expresión de arquitecturas, basadas normalmente en los mecanismos de extensión que el propio lenguaje proporciona (estereotipos, restricciones y valores etiquetados).

La idea principal que deriva de todos estos estudios es que hoy en día, a pesar de que UML es el lenguaje de modelado más utilizado, y con más perspectiva de futuro, no está suficientemente maduro en el campo de la expresión de arquitecturas. Aunque existen multitud de extensiones y perfiles para este fin, no hay uno que esté universalmente reconocido y estandarizado, y sería difícil hacer "ganador" a alguno de los métodos presentados, despreciando los demás.

A pesar de ello, no podemos decir que sea imposible expresar arquitecturas con UML, simplemente es cuestión de elegir el profile o método de expresión que más se ajuste a las necesidades del diseño que se esté abordando.

Estos autores coinciden en las tres estrategias que se pueden adoptar para intentar incorporar a UML los mecanismos necesarios para expresar arquitecturas:

1. Usar UML "como es", utilizando sus mecanismos básicos como paquetes, clases, etc.

---

[3] Del inglés *stereotypes, constraints and tagged values.*

2. Aumentar el meta-modelo de UML para soportar directamente los conceptos arquitectónicos necesarios.
3. Restringir el meta-modelo de UML usando mecanismos de extensión definidos en el propio lenguaje (*profiles*).

3.3.2.1. Extensiones de UML

En [Medvidovic00] se hace una detallada descripción de las ventajas e inconvenientes de la adopción de cada una de estas estrategias.

Básicamente, defiende que con la primera estrategia, a pesar de ser la más sencilla, sería difícil expresar ciertos conceptos arquitectónicos (como conectores[4] y estilos arquitectónicos) que no tienen un homólogo en el estándar UML. La segunda posibilidad mencionada, consistiría en añadir construcciones clave necesarias para el modelado de arquitecturas software en el estándar de UML. Sin embargo, el uso de esta segunda opción daría como resultado un lenguaje complejo y demasiado grande, difícil de comprender y usar.

La aproximación que la mayoría de autores coinciden en adoptar [Kande00], [Hofmeister99] y otros, estaría basada en la extensión de UML de manera estandar, usando los mecanismos de extensión estándar de UML, lo que daría como resultado un "*perfil para descripciones de Arquitecturas Software"*.

Un perfil se define como:

"Un contexto de definición consistente para elementos tal como son, pero no de forma limitada, consistente en reglas bien formadas, valores etiquetados, estereotipos, restricciones, semántica expresada en lenguaje natural, extensiones al meta-modelo estándar y reglas de transformación" [OMG99]

El *Object Management Group* [OMG99] ofrece los mecanismos para definir uno de estos perfiles, diferenciados entre mecanismos de extensión *ligeros* y *pesados.* Los segundos permiten adaptar la semántica de UML extendiendo el meta-modelo estándar.

Por otra parte, los mecanismos de extensión ligeros permiten adaptar la semántica de UML sin necesidad de cambiar su meta-modelo. Éstos son los mecanismos que adoptan los autores citados para expresar arquitecturas software a partir de una extensión de UML. Principalmente, se pueden describir como *valores etiquetados, estereotipos* y *restricciones* (es posible expresar restricciones en modelos UML usando el *Object Constraint Language* (OCL) [Warmer98], [OCL97])

3.3.2.2. Otros mecanismos de UML para expresar arquitecturas

Además de los mecanismos de extensión descritos, los principales elementos o *artefactos* básicos que incorpora el propio lenguaje UML y que se pueden usar para modelar arquitecturas se pueden resumir en los siguientes:

- **Clases, Interfaces y Objetos**. Las *clases* son las construcciones primarias para describir la vista lógica de un sistema. Una clase tiene propiedades en forma de atributos, proporciona servicios abstractos en forma de *operaciones*, y puede relacionarse lógicamente con otra clase usando *asociaciones*. Las clases pueden exponer su funcionalidad a través de una serie de *interfaces*, colecciones u operaciones relacionadas. Las clases pueden instanciarse en *objetos*, que se usan en modelos denominados *colaboraciones* para mostrar un comportamiento bajo escenarios particulares.

[4] Algunos conceptos de arquitectura software son muy diferentes de los de UML. Por ejemplo, los conectores son entidades de primera-clase en muchos ADLs. La funcionalidad de un conector se puede abstraer en una clase o componente. Sin embargo, los conectores tienen muchas propiedades que nos son soportadas directamente por una clase UML.

- **Componentes e Instancias de Componentes**. Los *componentes* en UML, se usan para describir las partes físicas de un sistema. Como las clases, los componentes exponen su funcionalidad a través de *interfaces*. Se relacionan entre sí usando relaciones de dependencia. El despliegue de un sistema en un hardware se describe asociando componentes con *nodos hardware*.
- **Paquetes**. UML proporciona un mecanismo de agrupación que se usa para particionar los modelos más grandes en unidades más manejables, llamadas *paquetes*. UML también define un tipo de elemento de agrupamiento llamado *subsistema*, que típicamente se usa para encapsular los modelos de objetos que definen un módulo de granuralidad más gruesa en un sistema.
- **Relaciones**. Los elementos de un modelo se relacionan entre sí mediante *asociaciones* y *dependencias*. Una *dependencia* es la relación más genérica en UML, indica que un elemento depende de alguna manera de la definición de otro elemento. Una *asociación* es una relación más rica que describe una relación abstracta entre clases y el rol que desempeña la clase en dicha relación.

Los artefactos descritos pueden componerse de varias formas en un modelo UML y pueden visualizarse en diagramas que pueden ofrecer distintas *vistas* de la arquitectura. Además, se pueden asociar *anotaciones textuales* con cualquiera de ellos.

UML también define una serie de modelos para describir el comportamiento dinámico de un sistema, incluyendo diagramas de colaboración que especifican el comportamiento del sistema usando *escenarios de interacción* basados en *eventos*, descripciones basadas en *máquinas de estado*, y *casos de uso*.

Un aspecto importante a resaltar en la utilización de UML para describir arquitectura es que la misma notación puede tener un amplio rango de semánticas. Se usa el mismo diagrama básico, el diagrama de clases/objetos, para mostrar la mayoría de los aspectos de la arquitectura. Precisamente esta ventaja se puede tornar en inconveniente, puesto que existe el riesgo de desdibujar la distinción entre arquitectura e implementación. Por esta razón, se usan estereotipos y símbolos especiales para minimizar la confusión entre diferentes vistas.

3.3.3. UML para modelar sistemas de tiempo-real

El dominio para el que se desarrolla este trabajo de tesis está muy relacionado con los sistemas de tiempo-real. Por definición, un sistema (empotrado o no) de tiempo-real controla y monitoriza procesos físicos con restricciones temporales, que pueden ser más o menos estrictas; éste es el caso de las unidades de control de robots. Estos sistemas intensivos en software deben operar bajo restricciones más severas que otros sistemas software y además, funcionar con fiabilidad durante largos periodos de tiempo. Algunas de estas restricciones son inherentes de su dominio, como la planificabilidad, predecibilidad y robustez. Otras de las restricciones vienen de la necesidad de reducir el coste del sistema mediante la reducción de la cantidad de memoria o la capacidad del procesador.

La especificación y diseño de sistemas de tiempo-real es una cuestión compleja, porque estos sistemas requieren no sólo corrección lógica, sino también corrección temporal para poder decir que el sistema funciona correctamente. Durante muchos años se han desarrollado varias técnicas para especificar y diseñar sistemas de tiempo-real, como son las *máquinas de estado finitas* y *Redes de Petri*. Según Kratz [Kratz98], para afirmar que un lenguaje de modelado que tiende por definición a ser *universal*, como UML, puede utilizarse también para especificar sistemas de tiempo-real, habría que discutir si es aplicable en la descripción de tres notaciones de tiempo distintas:

- Tiempo *ordenado*
- Tiempo *relativo*
- Tiempo *absoluto*

Se define tiempo ***ordenado*** como aquel cuya estructura puede ser definida matemáticamente en términos de una serie ordenada de eventos en combinación con una relación de precedencia (de forma que se pueda decir que el evento A_1 precede al evento A_2). El tiempo ***relativo*** es una notación implícita de tiempo-real a través de eventos, relacione entre los eventos y duraciones de las acciones. Las relaciones entre eventos se expresan mediante relaciones de precedencia y la longitud de los intervalos entre eventos se expresa por ejemplo, en segundos. De esta forma, para expresar relaciones entre eventos, no se utilizan puntos de referencia fijos en el tiempo. Por último, el tiempo ***absoluto*** es una notación explícita de tiempo real con la que el comportamiento del sistema se puede especificar para una serie de puntos en el tiempo, con una exactitud dependiente de la granuralidad escogida. De esta forma, en tiempo *absoluto*, los eventos están calibrados según un punto de referencia fijo en el tiempo y por tanto, esta sería la noción cuantitativa de tiempo-real.

Además de considerar qué noción de tiempo es aplicable, un lenguaje de modelado aplicable para tiempo-real debe poder expresar dos tipos de requisitos temporales:

- Requisitos temporales de contexto o dirigidos por eventos
- Requisitos temporales generales

En el primer caso, un evento inicia la acción y es en ese momento cuando el requisito de contexto debe cumplirse, por ejemplo, un evento A obtendrá una respuesta B en 5 segundos como máximo. En el caso de los requisitos generales, aplican a toda la vida del sistema.

En [Kratz98], el autor hace un estudio de si UML puede expresar estos requisitos o no, utilizando varios ejemplos de modelado dinámico y de objetos en combinación con *timers*. Según concluirá en su estudio, UML puede expresar tiempo *ordenado*, tiempo *relativo*, una forma de tiempo *absoluto* y *requisitos temporales de contexto*, pero no *requisitos generales*.

Los dos primeros se pueden expresar mediante los diagramas de secuencia UML, por lo tanto, también se pueden expresar *requisitos de contexto*. Una desventaja de especificar *requisitos de contexto* es que UML no tiene una notación fijada para expresar restricciones de tiempo, por lo que el diseñador tiene que expresarlo *"como pueda"*, por ejemplo, añadiendo notas a un diagrama de secuencia. La principal desventaja de tener que hacerse su propio lenguaje para expresar estas restricciones es que no hay soporte de herramienta.

3.3.3.1. UML-RT

Otra opinión se puede encontrar en [Selic98]. Es interesante observar como Selic justifica el uso de UML sin nuevos conceptos de modelado, es decir usando los mecanismos del estándar y extendiendo su significado usando simplemente valores etiquetados, estereotipos y restricciones, como ya se justificó en el punto 3.3.2. En este documento técnico de Rational, Selic expone cómo utilizando UML de esta forma, se pueden expresar los conceptos de ROOM (*Real Time Object Oriented Method*) [Selic94] para modelar sistemas de tiempo real (cápsulas, puertos y conectores). Posteriormente, el propio Selic aportará su experiencia para que el estándar UML 2.0. ofrezca mecanismos que ayuden a modelar sistemas de tiempo-real [Selic03].

UML-RT es una especialización de UML para una serie de sistemas caracterizados como complejos, dirigidos por eventos y potencialmente distribuidos. Los nuevos conceptos añadidos a UML se derivan de ROOM, que también es una técnica de modelado orientada a objetos optimizada para sistemas de tiempo-real.

El elemento de modelado fundamental en UML-RT es una *cápsula*. Una cápsula representa un flujo independiente de control en un sistema que se comunica con otros objetos solamente mediante el envío y recepción de mensajes a través de objetos interfaz llamados *puertos*. Una cápsula puede tener una estructura interna de modo que sea posible describir los sistemas más complejos de manera jerárquica. Las cápsulas y las conexiones entre sus puertos construyen la estructura de un modelo. El comportamiento de cada cápsula es descrito por una máquina de estados. El estado de una cápsula cambia cuando llega un mensaje al puerto de una cápsula.

Por otra parte, basándose en los trabajos de Selic y en la notación del lenguaje de descripción de arquitecturas ROOM, Soni y Hofmeister [Hofmeister00] proponen una metodología de desarrollo basada en cuatro vistas que se comentará en el punto 3.4.3.1.

3.3.3.2. Componentes, puertos y conectores

Tanto para los métodos de descripción de arquitecturas (ROOM llama a estos componentes *cápsulas*, como se ha visto) como para la notación de Hofmeister y para la nueva versión UML 2.0., los conceptos de componente, puerto y conector pasan a ser entidades de primera clase para describir una arquitectura a nivel conceptual. El concepto de **componente** ya se definió en el capítulo 2, y en la **Fig. 3.1** se puede ver íntimamente ligado a los puertos, que son los puntos de interacción del componente con su entorno en general, o con otros componentes, en particular. Un **puerto** [Selic94] proporciona una manera regular de intercambiar datos y control entre componentes, independientemente de su funcionalidad y granuralidad; ofrecen una puerta a los servicios requeridos y ofrecidos por el componente.

Un componente a su vez, puede descomponerse en otros componentes y conectores. Un puerto no es lo mismo que otra noción más común: *interfaz*. Una interfaz define servicios u operaciones que el componente proporciona, pero no las que usa. Por contra, un puerto define tanto mensajes de entrada como de salida. De esta forma, un componente puede proporcionar servicios, pero también requerirlos. (ver **Fig. 3.1**)

Las interfaces no tienen definida una implementación asociada; sólo ofrecen una lista de operaciones que proporcionan. En cambio, los puertos *pueden* tener una implementación, así que pueden incorporar funcionalidad para adaptar, combinar o procesar operaciones de llegada o de salida. Además, cada puerto tiene un protocolo asociado que establece de qué modo deben ser ordenadas las operaciones de entrada y salida.

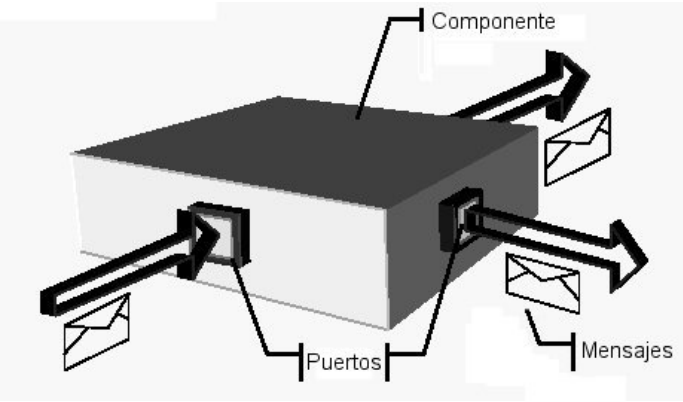


Fig. 3.1.- Concepto de componente con puertos de entrada y salida y paso de mensajes [Selic94]

La conexión entre puertos (y en consecuencia, entre componentes) se hace a través de los conectores:

Los **conectores** median en las interacciones entre componentes, estableciendo las reglas que gobiernan la interacción entre componentes y proporcionando mecanismos auxiliares [Shaw96]. Un conector permite separar la funcionalidad de los componentes de sus patrones de interacción, puesto que están incluidos en los propios conectores. Al igual que el comportamiento funcional del sistema se concentra en los componentes, el control lo hace en los conectores. Para describir el comportamiento de un conector hay que centrarse en los aspectos de control. La separación de los patrones de interacción de los componentes de su funcionalidad exige, como han demostrado diversos autores [Medvidovic99, 00] [Metha00] [Shaw96], considerar a los conectores como entidades de diseño de primera clase, al mismo nivel que los componentes que relacionan (**Fig. 4.2).**

Queda por determinar la manera en que los conectores deben conectar los puertos de los componentes. Para ello habría que responder primero como se conectan ellos mismos a los componentes. Puesto que la única forma de conectarse a un componente es a través de sus puertos, los conectores deben tener a

su vez puertos que sean compatibles con los de los componentes (en las 4-V.H., estos puertos de conectores se denominan **roles**, que a su vez, obedecen a un protocolo asociado, que normalmente será el conjugado del puerto al que se conecta, ver **Fig. 3.2**). Es inmediato inferir que los conectores son a su vez un tipo de componente que ofrece los mismos tipos de puertos que cualquier otro y cuya responsabilidad específica es gestionar protocolos y reglas de negocio. Pero es un componente especial, ya que desde el punto de vista de los componentes que enlaza no existe. Si los componentes percibieran la existencia de un mediador dependerían de alguna forma de él.

Los puertos tienen una relación de composición con los componentes y se conectan con otros puertos (lo que implica finalmente una conexión entre componentes) gracias a los conectores. Por ejemplo, en la **Fig. 3.2** se muestra un esquema de conexión entre dos componentes gracias a un conector tipo *Event*. Al adoptar las 4-V.H. se simplifica esta representación, como veremos en los diagramas conceptuales presentados en el Capítulo 6, de forma que cada puerto sea simplemente un "*cuadradito*" en el borde del componente al que pertenece.

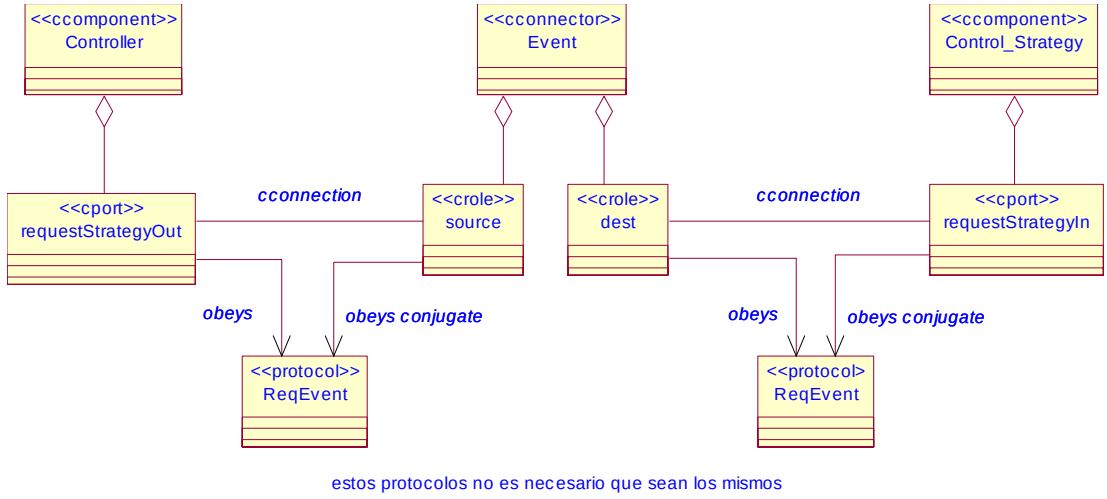


Fig. 3.2.- Ejemplo de conector tipo *Event* entre *Controller* y *Control_Strategy*

3.4. Metodologías de desarrollo de arquitecturas

Tal como se dijo en la introducción de esta tesis, el diseño de sistemas intensivos en software complejos, como son las unidades de control de robots, requiere conocer y seguir disciplinadamente una serie de reglas bien definidas que indiquen como ensamblar y conectar los diferentes elementos del sistema y cómo gestionar el desarrollo y evolución del mismo. Por tanto, además de definir y describir una arquitectura de manera formal o semi-formal, se debería utilizar una metodología sistemática de desarrollo que permita en el futuro reutilizar la arquitectura minimizando los esfuerzos necesarios para ello, aunque la configuración hardware-software sea distinta. De esta forma se conseguiría reducir los costes y los plazos de desarrollo del mismo.

Además del factor económico, utilizar un proceso de desarrollo ayuda a:

- Producir sistemas de calidad consistente
- Producir sistemas fiables con requisitos de comportamiento complejos
- Predecir cuando los sistemas serán completos
- Predecir cuanto costará desarrollar un sistema
- Identificar aspectos críticos durante el desarrollo, permitiendo hacer correcciones cuando sea necesario

- Permitir una eficiente colaboración entre los miembros del equipo que desarrollan el sistema.

Una metodología de desarrollo rigurosa que ofrezca esas ventajas constará de varias partes [Douglass00]:

- Una semántica definida y una notación. Ambos conceptos conciernen al *lenguaje de modelado*, como por ejemplo UML.
- Una serie de trabajos secuenciados.
- Una serie de *artefactos* que sean fruto del proceso de desarrollo y que ofrezcan las diferentes vistas del sistema diseñado.

Resumiendo, una metodología o proceso de desarrollo se podría definir de la siguiente forma [Douglass00]:

"Un proceso de desarrollo describirá las actividades que gobiernan el uso de los elementos del lenguaje y los artefactos de diseño resultantes de la aplicación de éstos en una secuencia definida de actividades".

3.4.1. El ciclo de desarrollo

Mientras que las fases de un proceso de desarrollo identifican lo que se debe hacer, el ciclo de vida del diseño, especifica **cuando** se debe hacer.

En la bibliografía clásica se pueden encontrar típicos esquemas de ciclos de desarrollo [Gomaa00], [Boehm88]. [Douglass00]. Se puede decir que hay dos aproximaciones primarias para secuenciar las fases de desarrollo:

- Ciclos en cascada
- Ciclos iterativos

Los ciclos en cascada son los más comunes y ordenan las fases de manera lineal. Tienen la ventaja de la simplicidad y un fuerte suporte de herramientas, pero su principal desventaja estriba en que, al dar por finalizada una fase de diseño para pasar a la siguiente fase, no se aprovecha de las consecuencias que puede en las fases anteriores las decisiones de diseño hechas posteriormente. La experiencia dicta que el análisis no puede estar finalizado sin haber hecho antes algo de diseño, y el diseño no puede darse por acabado si antes no se hacen algunas pruebas.

Para intentar solucionar este problema surgen los ciclos iterativos. En estos ciclos, cada fase de desarrollo no se ejecuta sólo una vez, sino que se vuelve a ejecutar iterativamente hasta que el sistema es completamente elaborado. Cada *vuelta* da como resultado un *prototipo* que se puede evaluar y probar, donde se puede observar si se cumplen los requisitos establecidos, funcionales y no funcionales, lo cual hace que se eliminen riesgos en fases más tempranas de diseño. Un clásico ejemplo de estos ciclos es el desarrollo en espiral de Boehm (**Fig. 3.3**) [Boehm88], en el que se basan los modelos de desarrollo orientados a objetos como COMET y ROPES.

Además de esta clasificación, por el modelo de ciclo de desarrollo, se puede hacer otra división, también muy importante, según la orientación que tenga el diseño en dichas metodologías. Así, se distingue entre:

- Metodologías de desarrollo orientadas a objeto
- y metodologías de de desarrollo orientadas a la arquitectura

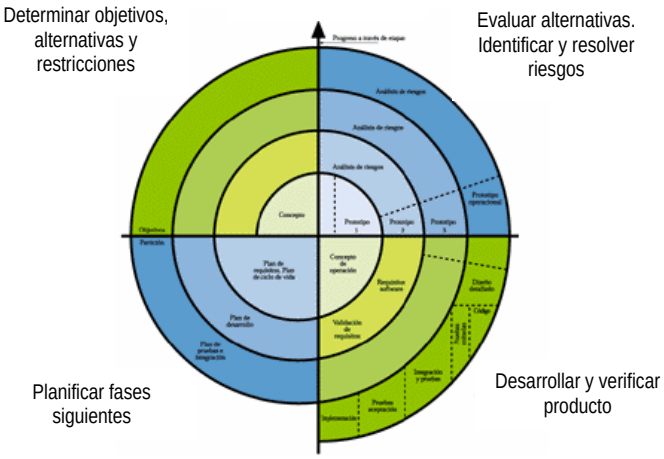


Fig. 3.3.- Ciclo de desarrollo en espiral

3.4.2. Metodologías de desarrollo orientadas a objetos

Existe una gran variedad de metodologías de análisis y diseño, desde las basadas en los principios de la programación estructurada [Yourdon79] hasta las más actuales orientadas a objetos, entre las que destacan las definidas por Booch [Booch94], Jacobson [Jacobson92], Rumbaugh (OMT) [Rumbaugh91], Shlaer-Mellor [Shlaer88], Catalysis [D'Souza99] y USDP [Jacobson00], y más recientemente y orientadas al diseño de sistemas empotrados de tiempo-real, las definidas por Gomaa, método **COMET** [Gomma00], y Douglass, método **ROPES** [Douglass00]. Estas dos últimas se basan en las reglas y principios de diseño definidas por las anteriores (especialmente en [Jacobson92]), utilizan el Lenguaje Unificado de Modelado (UML) e incorporan nuevos conceptos orientados al diseño de sistemas concurrentes de tiempo real.

Simplificando mucho puede decirse que el enfoque de las metodologías de diseño orientadas a objeto se basa en la identificación de los objetos más representativos del dominio considerado, su agrupación en diferentes clases y la definición de los mensajes o servicios intercambiados entre los mismos. Esto es, siguen un enfoque de desarrollo *bottom-up*, desde los objetos particulares que interaccionarán hacia arriba, hasta la agrupación de dichos objetos en subsistemas que compongan la arquitectura. La identificación de los **objetos** más representativos y sus patrones de interacción requiere de una fase previa de **análisis** en la que se estudian las características del dominio del problema.

Para la realización de este análisis, la utilización de los **casos de uso** propuestos por Jacobson [Jacobson92] se ha ido imponiendo con fuerza, y hoy en día se hace difícil hablar análisis orientado a objetos prescindiendo de los mismos. Los casos de uso describen las interacciones de los usuarios finales con el sistema y se utilizan, por tanto, para descubrir su funcionalidad básica. A partir de los casos de uso se construye un modelo de análisis que se va elaborando progresivamente hasta obtener modelos de diseño cada vez más refinados que en última instancia dan lugar a un modelo que se puede implementar en un lenguaje de programación sobre una configuración hardware. Para la definición de los modelos de diseño se aportan heurísticos y criterios que permiten identificar clases y objetos, determinar sus dependencias, definir relaciones de herencia y composición y seleccionar los patrones de interacción que mejor se adapten a las *calidades* exigibles al sistema.

Sea cual sea el enfoque, el proceso está dirigido por casos de uso y el diseño sigue un desarrollo iterativo e incremental en el que los objetivos de cada ciclo se establecen en función de los riesgos a resolver y de los casos de uso considerados. En resumen, son **metodologías dirigidas por riesgos y casos de uso que utilizan los conceptos del diseño orientado a objetos**.

3.4.2.1. COMET, ROPES y USDP

Se puede decir que estos son los principales métodos de desarrollo orientados a objetos que utilizan la notación UML para describir el diseño. Los tres son bastante parecidos y a los tres se le pueden aplicar las consideraciones hechas en los párrafos anteriores.

COMET (*Concurrent Object Modeling and architectural design mEThod*[5]) [Gomaa00] es un método de diseño para aplicaciones concurrentes, distribuidas y de tiempo real. El proceso de desarrollo de COMET es un proceso orientado a objetos que, según dice el propio Gomaa, es compatible con el *Unified Software Development Process*[6] (USDP), propuesto por Jacobson, Booch y Rumbaugh [Jacobson00] y el modelo en espiral de Boehm [Boehm88], cada fase de COMET corresponde con un flujo de trabajo de USDP (requisitos, análisis, diseño, implementación y test)

ROPES (*Rapid Object-oriented Process for Embedded Systems*[7]) es un proceso de desarrollo que enfatiza el prototipado rápido, con la realización de pruebas en niveles tempranos de desarrollo y minimizando los riesgos. El propósito de este proceso es incrementar la calidad del producto final, mejorar la repetitividad y predictividad del desarrollo y reducir los esfuerzos requeridos para desarrollar un producto final con el nivel de calidad exigido. Para conseguir esto, un proceso de desarrollo como ROPES está compuesto de una serie de fases, cada fase contiene actividades y resulta de ellas determinados artefactos. Estas fases iteran una y otra vez (**Fig. 3.4**) para producir los prototipos de desarrollo, que están organizados alrededor de los casos de uso.

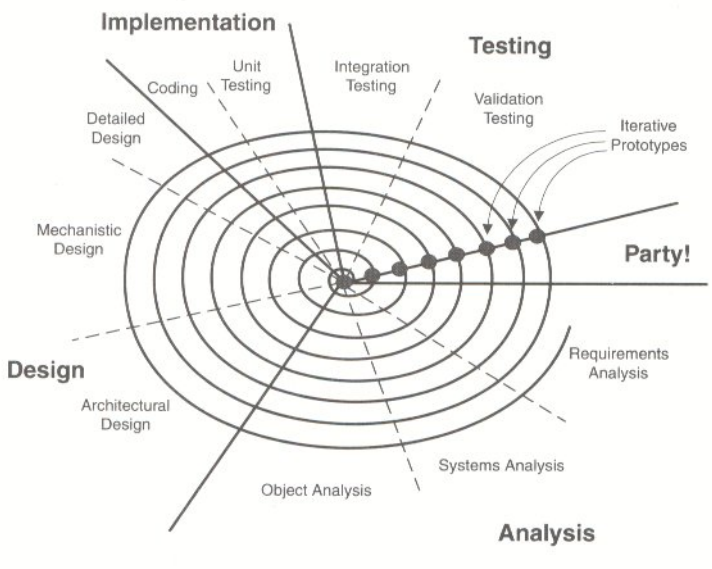


Fig. 3.4 – Ciclo de desarrollo evolutivo e incremental de ROPES

Tanto COMET como ROPES siguen un ciclo de desarrollo evolutivo e incremental que básicamente sigue los siguientes pasos y modelos:

1. Planteamiento de los **requisitos** del sistema.
2. **Análisis** (estático y dinámico), descubriendo los objetos y clases influyentes del sistema.
3. **Diseño** arquitectónico y detallado.
4. **Implementación** y **Test** del sistema diseñado.

---

[5] Método de desarrollo arquitectónico y modelado de objetos concurrentes.
[6] Proceso unificado de desarrollo de software.
[7] Proceso rápido orientado a objetos para sistemas empotrados.

COMET y ROPES dan nombres distintos a estas fases o las organizan en sub-fases distintas, englobando unas dentro de otras, por ejemplo, ROPES incluye la fase de *Análisis de Requisitos* dentro de un modelado de *Análisis* general, mientras que COMET la describe como un paso inicial independiente, pero lo importante es que ambos métodos proponen un ciclo de desarrollo bastante parecido.

En cuanto al USPD, como se ha mencionado, es la base para COMET y se puede decir que también para ROPES, puesto que fue propuesto por los creadores de UML. La diferencia estriba en que fue diseñado para el desarrollo de software "general", mientras que COMET y ROPES, aunque siguen básicamente el mismo ciclo de diseño, hacen consideraciones muy particulares para sistemas complejos, intensivos en software, donde el hardware también es importante y donde es necesario tener en cuenta restricciones de tiempo real, por lo que tienen más aplicación en el ámbito de esta tesis, y se tendrán más en cuenta.

3.4.3. Metodologías de desarrollo orientadas a la arquitectura

Aunque las metodologías anteriormente comentadas aportan enfoques útiles y cubren con mejor o peor fortuna todas las fases del proceso de desarrollo de un sistema intensivo en software, presentan un grave inconveniente para su empleo en este trabajo de tesis: en general, **consideran el diseño de un sistema concreto y no el de una familia de sistemas**. Además, realizan la distribución arquitectónica del sistema en una fase tardía del desarrollo, una vez que ya se han averiguado los objetos que debe haber en el sistema, las clases e incluso el comportamiento dinámico de las mismas, siendo la fase de diseño arquitectónico, normalmente una mera agrupación de objetos en torno a subsistemas y la definición de interfaces entre ellos

En los sistemas concretos es relativamente fácil seleccionar un conjunto reducido, pero lo suficientemente representativo, de casos de uso. Los requisitos pueden especificarse con concreción y las líneas de evolución del sistema pueden acotarse con un alto grado de confianza. Sin embargo, cuando se aborda el diseño de una arquitectura de referencia para una familia de sistemas o para una línea de producto, las premisas anteriores dejan de ser ciertas. Los casos de uso pueden variar mucho entre sistemas, pudiendo tener relevancia en unos sistemas y en otros no tenerla, o bien desarrollarse de formas diferentes. Lo mismo ocurre con los requisitos. A menudo, sólo se conocen de forma general y su especificación varía entre sistemas.

El diseño de una **arquitectura de referencia** requiere de una metodología que tenga explícitamente en cuenta la falta de concreción de los requisitos y maneje la variabilidad entre productos o sistemas. Además, se hace necesario abordar las decisiones estratégicas de diseño arquitectónico al principio del ciclo de desarrollo, de forma que estas decisiones sean tenidas en cuenta en la fase de diseño detallado de la arquitectura más adelante. Por todo ello, deberán tenerse en cuenta otros métodos de desarrollo distintos a los orientados a objetos, como los que se muestran en el siguiente apartado.

3.4.3.1. IEEE-1471, ABD y 4-vistas de Hofmeister

La Práctica Recomendada **IEEE Std 1471-2000** [IEEE Std 1471-2000] para la descripción arquitectónica de sistemas intensivos en software pretende dirigir las actividades de creación, análisis, y mantenimiento de arquitecturas de sistemas intensivos en software, así como la descripción de las mismas. El estándar define de alguna manera los puntos de vista que deben tenerse en consideración para describir la arquitectura, pero a un nivel de abstracción tan alto que las vistas pueden seleccionarse o definirse siguiendo criterios muy diferentes. Las intenciones son buenas, las recomendaciones se rigen por el sentido común, pero se queda a un nivel de abstracción demasiado alto como para guiar el proceso de desarrollo de la arquitectura de principio a fin.

El Método de Diseño Basado en la Arquitectura o **ABD** (de sus siglas en inglés *Architecture Based Design Method*) [Bachmann00a] es un método para el diseño de arquitecturas de referencia,

arquitecturas software para líneas de producto y sistemas de larga vida operativa, desarrollado por el SEI de la CMU[8]. Es un método de desarrollo que diseño de arquitecturas que se basa en los requisitos de calidad y que sigue una filosofía *top-down* al partir de una descomposición funcional del problema y una división del sistema en subsistemas más simples, eligiendo a cada paso el estilo arquitectónico más adecuado. Además, en cada iteración del proceso de diseño, se realiza un proceso de evaluación para ver si se cumplen los requisitos establecidos.

Aunque no es el objetivo de esta tesis evaluar la arquitectura (el propio ABD recomienda que lo haga un organismo distinto), cabe destacar aquí que ABD incluye un mini-proceso ATAM [Kazman00] en cada iteración, lo que hace que este método sea bastante atractivo a la hora de plantear el diseño de la arquitectura propuesta en esta tesis. ABD utiliza los escenarios de calidad como mecanismos de evaluación primarios. Las estructuras propuestas en la arquitectura son examinadas vía escenarios de calidad para confirmar que son alcanzables al nivel de desarrollo presente. Si lo son, se puede proceder al siguiente refinamiento del diseño; si no, se debe reconsiderar el paso a un siguiente nivel, y replantear las estructuras propuestas.

En la tesis de J.A. Pastor, *"Evaluación y Desarrollo Incremental de una Arquitectura Software de Referencia para Sistemas de Teleoperación utilizando Métodos Formales"* [Pastor02-td] se puede encontrar una exhaustiva evaluación de la arquitectura propuesta en [Álvarez97-td], *"Arquitectura Software de Referencia para Sistemas de Teleoperación"*

Una tercera aproximación al proceso de desarrollo de una arquitectura es la ofrecida por Hofmeister, Soni y Nord [Hofmeister00], con sus 4 vistas **(4-V.H.)**. Estos autores proponen una metodología basada en el desarrollo las *vistas Conceptual, de Módulos, de Código y de Ejecución* de una arquitectura. Las bases de su método de diseño son estas cuatro vistas, sus actividades de diseño y las dependencias entre ellas; tanto es así, que ni siquiera ponen un nombre a su metodología, simplemente la basan en las 4 vistas. Al contrario del modelo 4+1 vistas de Kruchten [Kruchten95], usado también en UML, donde todas ellas coexisten representando puntos de vista distintos de la arquitectura, cada vista de [Hofmeister00] corresponde más bien a un nivel de abstracción desde el más alto (vista conceptual) al más bajo (vista de ejecución) así como a una fase progresiva del diseño. Para cada vista, la mayoría de las decisiones de diseño son independientes de las otras vistas, pero hay algunas decisiones que se ven afectadas por las decisiones tomadas más tarde.

Un aspecto interesante del método propuesto en [Hofmeister00] es la notación utilizada. Originalmente utilizaban los clásicos diagramas de cajas y líneas para describir arquitecturas; más tarde, comprendiendo las ventajas de utilizar UML, incorporaron esta notación y la ampliaron con los mecanismos de extensión incorporados en el lenguaje (como se vio en el punto 3.3.2.1) para adoptar en la *Vista Conceptual* una representación de componentes, conectores y puertos basada en la notación de modelado orientado a objetos para sistemas de tiempo-real ROOM [Selic94], [Selic98]. Un hecho que corrobora que esta notación va en la línea adecuada es que ha aportado ideas para los nuevos mecanismos de expresión de arquitecturas que ofrece UML 2.0.

En el Anexo I de esta tesis se puede encontrar una explicación más detallada de ABD y Hofmeister, con sus procesos de desarrollo asociados.

3.5. Enfoque metodológico de la tesis

Durante el desarrollo de este capítulo y el anterior se han estudiado diferentes métodos de desarrollo de sistemas intensivos en software y distintas posibilidades de plantear arquitecturas de referencia reutilizables para una familia de sistemas. Aunque se han podido encontrar muchas arquitecturas para robots en la bibliografía (ver Capítulo 4), ha sido muy difícil encontrar ejemplos de procesos de desarrollo para definir arquitecturas de referencia en el dominio robótico, y no se ha encontrado ningún trabajo similar al planteado como objetivo fundamental de esta tesis: proponer una

[8] *Software Engineering Institute* de la *Carnegie Mellon University,* Pittsburgh, USA.

arquitectura de referencia específica del dominio de las unidades de control de robots de servicio teleoperados.

Como se detalla a continuación, para obtener dicha arquitectura se seguirá principalmente una aproximación centrada en la arquitectura, como la que propone el Método de Diseño Basado en Arquitectura (**ABD**) [Bachmann00a], completándolo con las **4 vistas de Hofmeister** [Hofmeister00], adoptando sobre todo su vista conceptual basada en componentes y utilizando como notación las extensiones de **UML** inspiradas en **ROOM** [Selic94] de este método. En las vistas más próximas a la implementación se incorporarán algunas de las recomendaciones de métodos de desarrollo orientados a objetos y casos de uso, como es **COMET [**Gomaa00].

3.5.1. Metodología de desarrollo

En los puntos 3.4.2 y 3.4.3 se han comparado los métodos de desarrollo orientados a objetos y centrados en casos de uso (desarrollo *bottom-up*) y los orientados a la arquitectura como ABD (*top-down*) y se ha visto que para diseñar una arquitectura de referencia para un dominio, hay que abordar las decisiones estratégicas de diseño arquitectónico al principio del ciclo de desarrollo, de forma que estas decisiones sean tenidas en cuenta más adelante en la fase de diseño detallado de la arquitectura más adelante. Precisamente ABD parece el más adecuado para el desarrollo de esta tesis puesto que es un método que está especialmente enfocado al diseño de **arquitecturas de referencia**, **líneas de producto** y sistemas de larga vida operativa. ABD además tiene muy en cuenta la evaluación de la arquitectura que se está diseñando, incluyendo mini-evaluaciones ATAM [Kazman00] en su proceso de diseño. ATAM es un **método de evaluación** de arquitecturas ampliamente conocido y que ya se ha utilizado en [Pastor02-td] para evaluar la arquitectura de referencia para sistemas de teleoperación propuesta por [Álvarez97-td]. Dado que el propio ABD parte de atributos de calidad para desarrollar el proceso de diseño está claro que el planteamiento de los atributos de calidad (o requisitos de calidad), así como su seguimiento en todo el proceso de desarrollo de la misma, serán actividades de vital importancia.

Puesto que ABD al final proporciona lo que las 4-V.H. llaman *vista conceptual*, el ABD podría considerarse como el primer paso del método de desarrollo de las 4-V.H. Una arquitectura de referencia como la propuesta en esta tesis debe quedarse en la vista conceptual (incluyendo las relaciones posibles del modelo de componentes que también propondrá la arquitectura, tipos de conectores y de componentes y reglas de conexión), puesto que concretar más (vista de módulos, ejecución, etc.) sería plantear un caso específico de un sistema. Las 4-V.H. ofrecen a la fase de diseño conceptual interesantes aportaciones para el estudio sobre variabilidad e impacto sobre la arquitectura de los cambios y la adaptación de los requisitos a los subsistemas propuestos.

Ambos métodos promueven el planteamiento de aspectos fundamentales que condicionan el diseño de la arquitectura a partir del análisis de requisitos. ABD los llama *architectural drivers* (**directrices arquitectónicas**) y Hofmeister *architectural issues*. Para hacer que la arquitectura los cumpla, ambos proponen **opciones** (ABD) o **estrategias** arquitectónicas (4-V.H.) que en general, inciden en la importancia de los **patrones y estilos** arquitectónicos [Buschmann96], cuya importancia para el desarrollo de arquitecturas se destacó en el capítulo anterior.

Es importante resaltar que ABD promueve la distinción entre **requisitos abstractos** (tanto funcionales como de calidad), que sirven para plantear los requisitos que debe cumplir la arquitectura de referencia, y por otra parte, la concreción de dichos requisitos abstractos en los **casos de uso** y **escenarios de calidad**, como la interacción de la arquitectura con un sistema concreto de la posible familia de sistemas.

Las propuestas de ABD, que al final buscan una vista conceptual de la arquitectura (como ellos mismos reconocen) son perfectamente compatibles con [Hofmeister99], puesto que básicamente proponen la subdivisión de componentes de grano grueso en otros más finos para alcanzar la funcionalidad y calidad buscada, eligiendo una serie de opciones o estrategias arquitectónicos. Una contribución importante de la vista conceptual de Hofmeister, que añade utilidad a la división en

subsistemas principales que hace ABD, es la posibilidad de tener una vista donde se observe la interrelación entre los distintos subsistemas en los que se ha dividido inicialmente la arquitectura. Además, ofrece una notación que incorpora componentes, puertos y conectores, ideal para expresar el modelo de componentes abstracto o conceptual que ofrece la arquitectura propuesta en esta tesis.

Sin embargo, se ha dicho que uno de los puntos débiles de ABD es que la ejecución del método termina con unos componentes conceptuales que tienen un grado de generalidad demasiado alto. Las 4-V.H. se podría solapar con ABD en este punto para concretar el diseño arquitectónico, e incluso dotarlo de una notación más estándar, como es UML. Las 4-V.H. promueven la utilización de UML y sus mecanismos de extensión, elevando los componentes conceptuales y conectores a entidades de primer orden en su modelo, cosa que está más acorde con las tendencias actuales y cuya utilidad y conveniencia han sido ampliamente justificados en el Capítulo 2 de esta tesis.

Todos estos factores han influido para que en esta tesis se utilicen los mecanismos más apropiados de ambos métodos intentando aprovechar las ventajas de los dos, sin llegar a adoptar exclusivamente uno de ellos.

Por último, a nivel de implementación, la notación de Hofmeister hace más fácil que ABD la conexión entre los componentes conceptuales y sus implementaciones. En esta fase también se tendrán en cuenta muchos criterios de diseño aportados por procesos de desarrollo orientados a objeto, principalmente COMET, por ejemplo, sus criterios de estructuración y simplificación de tareas para la vista de ejecución, y los patrones propuestos en ROPES para garantizar la seguridad y los requisitos de tiempo-real. Que los procesos de diseño de sistemas particulares tengan influencia en el diseño de una arquitectura de referencia es algo absolutamente lógico, ya que una arquitectura de referencia centrada en un dominio se basa en la experiencia adquirida por los desarrolladores en dicho dominio (ver [Mehta00])

Para poder seguir las metodologías de desarrollo mencionadas, es preciso proporcionarles las entradas necesarias, en forma de características, requisitos y factores de negocio del dominio. En general, la obtención de una arquitectura de referencia específica de dominio implica realizar un **análisis de su dominio de aplicación** que ofrezca como resultado la identificación de las partes comunes a todas las aplicaciones del dominio, así como los requisitos funcionales y de calidad abstractos que constituirán las entradas al método de desarrollo de la arquitectura. Este análisis se realizará en el Capítulo 4.

3.5.2. Notación

Por las razones expuestas en el punto 3.2.1, no se adopta un ADL para representar la arquitectura. Los ADLs son difíciles de comprender y no se integran bien con las prácticas de desarrollo de software actuales. Se escoge UML en su revisión 1.4., para describir la arquitectura y las implementaciones de la misma en esta tesis, además de por las ventajas mencionadas en este capítulo, porque expresa bien la mayoría de las ideas expuestas, y sobre todo porque es ampliamente conocido. Como se dijo en 3.2.1, el estándar UML 2.0. no se ha incorporado finalmente a la tesis porque en el momento de redactarla todavía no se ha publicado y no existen herramientas CASE que lo soporten.

En cuanto a la expresión de arquitecturas con UML, no es el objetivo de esta tesis la discusión y análisis detallado de los distintos métodos de extensión de UML, o la propuesta de alguno adicional. Éste podría ser un tema de debate de otro tipo de estudio más próximo a la teoría de ingeniería del software; sin embargo, sí que es importante la elección de un mecanismo de expresión basado en UML que sea consistente y completo, como el que se propone en [Hofmeister00]. Para la tesis se adopta su notación basada en extensiones de UML inspiradas en ROOM [Selic94] y sus conceptos de componentes, puertos y conectores, lo cual ayuda a ofrecer una visión clara de los servicios que proporciona cada componente y los que requiere (en forma de puertos), y también facilita la representación gráfica de las relaciones entre dichos componentes (conectores que unen los puertos).

A un nivel de mayor detalle, en las 4-V.H. se proponen algo parecido a lo que se puede ver en los procesos de desarrollo como COMET y ROPES, es decir, la utilización de paquetes para expresar subsistemas y capas, utilizando las clases para los módulos, componentes, etc. En el resto de la tesis se

utilizará UML para expresar detalles de implementación de la arquitectura, comportamiento dinámico, casos de uso, etc.

En el paso de la vista conceptual a la vista de módulos para la utilización de la arquitectura en un sistema concreto (Capítulo 7) se optará por utilizar una **orientación a objetos** en los casos de una implementación *software* para el GOYA y Lázaro, traduciendo componentes y puertos a clases. En otras instanciaciones serán bloques de función de un PLC o bloques VHDL.

Uno de los principales beneficios que se les atribuye a los objetos es la posibilidad de alta cohesión entre los datos y las operaciones que los manipulan. La O.O. concibe el sistema global como un conjunto de objetos que realizan acciones y se intercambian mensajes. Dado que este enfoque se asemeja a la forma en que el mundo real funciona, las abstracciones O.O. son más intuitivas y poderosas. Incluso el vocabulario para nombrar los objetos viene del dominio del problema. La perspectiva del objeto, está así, a un nivel más alto, más próximo al dominio del problema y lejos del dominio de la implementación software. Todo esto da como resultado un sistema con bajo acoplamiento entre los aspectos independientes del mismo, mientras que mantiene una buena cohesión entre los aspectos que están inherentemente muy acoplados. Un *sensor*, con datos y operaciones, es un *sensor* en el mundo real y tiene una abstracción a nivel de objetos con datos y operaciones que se pueden realizar con él. De esta forma se justifica que no sólo los elementos *software* que se puedan diseñar, sino también el *hardware*, puedan ser representados como "objetos" de forma eficiente. Por todo esto es justificable que se utilice una orientación a objetos a la hora de plantear la implementación de la arquitectura.

3.6. Herramientas CASE[9] utilizadas

Existen ya en el mercado una buena cantidad de herramientas de diseño que permiten al ingeniero definir el modelo conceptual asociado a un problema y generar a partir de él la estructura o incluso el código de la aplicación. Es posible distinguir tres aproximaciones a esta línea:

Herramientas de enfoque estructural. Proporcionan generación de código a partir de las estructuras estáticas definidas (clases, atributos y tipos) y de las relaciones entre ellas. Entre ellas cabe destacar *System Architect* [Popkin97] y *Rational Rose* [RationalRose00], ésta última incorporando UML, si bien con algunas restricciones sobre el lenguaje definido en [OMG00]. Estas herramientas no generan código relativo al comportamiento, por lo que los desarrolladores deben completar manualmente el código generado.

Herramientas con enfoque de comportamiento. Incorporan la generación de código a partir de máquinas de estados y de la especificación asociada a sus acciones. Una aplicación clara de este enfoque es la simulación y depuración de sistemas. Dentro de esta línea pueden mencionarse *Rhapsody* de i-Logix [Ilogix00], *ObjectTime* [ObjectTime00] y *Rational Rose Real-Time* [Rational02], estas dos últimas basadas en el método *ROOM* [Selic98].

Enfoque de traducción. En este enfoque se incorpora un modelo de arquitectura (patrones, plantillas, etc.) que marca las reglas en el proceso de obtención de código (estrategias de implementación). Las herramientas que encajan en este enfoque proporcionan arquitecturas por defecto que pueden ser adaptadas. De esta forma, el modelo de arquitectura es independiente de la aplicación. La calidad del código obtenido depende de la definición del modelo de arquitectura. Dentro de esta línea se encuentra *BridgePoint* [Bridge00] de *Project Technology*, basada en el método de Shlaer-Mellor [Shlaer88], aunque incorpora UML en las últimas versiones.

De todas estas herramientas, se han utilizado las herramientas de Rational (*Rational Rose y Rational Rose Real-Time*) y *Rhapsody* de i-Logix en menor medida.

---

[9] Siglas en inglés de Ingeniería del Software Asistida por Ordenador *(Computer Aided Software Engineering).*

3.6.1. Rational Rose

Puesto que UML es el lenguaje de modelado elegido para la tesis, es bastante conveniente utilizar herramientas CASE que lo soporten y permita no sólo dibujar los diagramas necesarios, sino también relacionarlos y realizar una tarea de verdadero diseño arquitectónico.

Como herramienta CASE de propósito general se ha escogido *Rational Rose 2002* [Rational02]:

- *Rational Rose* [RationalRose00] es una herramienta de diseño y modelado visual que se inscribe dentro de un grupo de herramientas software desarrollado por *Rational* que proporcionan soporte para un desarrollo iterativo y basado en componentes. Aunque estos dos aspectos son conceptualmente independientes, su uso combinado es bastante natural.
- Permite la inclusión de nuevas utilidades como la herramienta *UML-Mast* [Drake00], [MAST01] desarrollada por la Universidad de Cantabria para el modelado, análisis y diseño de sistemas distribuidos de tiempo real.
- Incorpora mecanismos específicos de distintos lenguajes, entre ellos Ada95 [Rational01].
- Permite enlazar fácilmente los diferentes componentes y conectores de la arquitectura con otros documentos adicionales.
- *Rational Rose 2002* Incorpora la notación UML 1.4.

Rational Rose utiliza la notación UML, soportando los 9 tipos de diagramas definidos por éste. La correspondencia entre los diagramas de UML y las vistas de Krutchen se muestra en la **Fig. 3.5**.

Rational Rose es una herramienta válida para realizar la descripción de la arquitectura implementada en un sistema concreto, pero no se puede expresar la vista conceptual de la arquitectura siguiendo la notación de Hofmeister. Para hacer esto último, se recurre a la herramienta *Rational Rose Real-Time*, que utiliza la notación de UML-RT, basada en ROOM, con lo cual se pueden expresar directamente componentes, puertos y conectores.

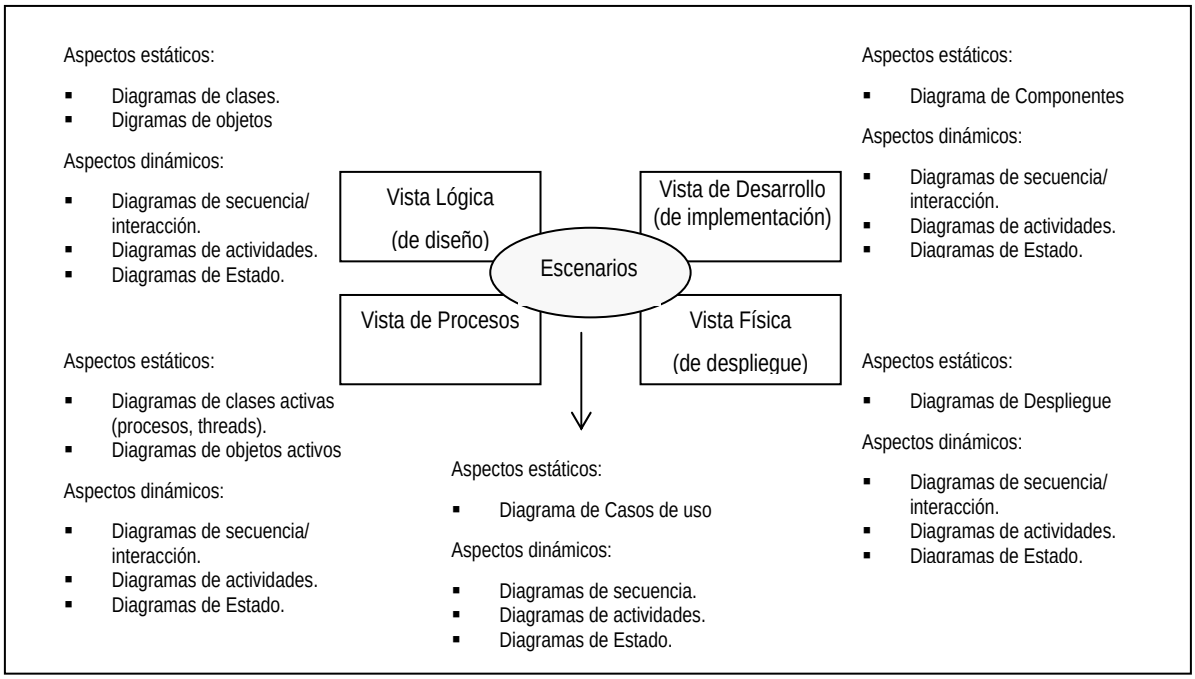


Fig. 3.5.- Diagramas UML y el modelo 4+1 vistas

3.6.2. Rational Rose RT

Rational Rose RT [Rational02] es una herramienta CASE que permite diseñar sistemas de tiempo-real orientados a objetos usando UML-RT [Selic98], ofreciendo cápsulas, con sus diagramas de estado asociados, y puertos. Se ha utilizado en esta tesis para representar la vista conceptual de la arquitectura.

Rational Rose RT permite diseñar un sistema de tiempo-real orientado a objetos proporcionando para ello un entorno visual de modelado. También permite generar automáticamente un programa ejecutable compuesto de código de aplicación, código esqueleto y librerías del sistema. Esta posibilidad de generación de código no se utiliza en la tesis, Rose RT sólo se utiliza para generar los diagramas conceptuales de la arquitectura, utilizando las cápsulas como representación de los componentes conceptuales. No existe la posibilidad de representar conectores al estilo de las 4-V.H., por tanto, para representar los diagramas arquitectónicos conceptuales de esta tesis, se opta por utilizar una línea de conexión estereotipada, si el conector fuera más complejo se podría utilizar un componente intermedio. Esta forma de representación parece ser que será la adoptada por UML 2.0.

Capítulo 4

Estudio del Dominio de los Robots de Servicio Teleoperados

4.1. Introducción

Contrariamente a la imagen que pueda tener el gran público sobre la robótica, más cercana a la ciencia ficción que a la realidad, los robots sólo pueden realizar de forma segura y eficiente labores relativamente sencillas, normalmente repetitivas, como la ejecución de ciertos movimientos previamente definidos utilizando sistemas de referencia perfectamente calibrados. De ahí el éxito de los robots industriales utilizados en procesos de fabricación. Pero cuando se trata de realizar trabajos de cierta complejidad que requieren de una realimentación muy precisa para ser controlados o el robot debe trabajar en entornos no estructurados donde la calibración es difícil o imposible, las cosas cambian radicalmente y un robot industrial probablemente no sea adecuado. Existen muy pocos sistemas de este tipo que hayan probado su eficacia y de ellos ninguno se libra de la existencia de un operador humano que supervise su funcionamiento y corrija sus errores.

La capacidad de percepción, procesamiento e "inteligencia artificial" que tienen muchos robots autónomos diseñados para desenvolverse en entornos no conocidos a priori o para realizar tareas no previstas, actualmente es muy reducida. Habrá que seguir recurriendo por tanto, a la inteligencia y capacidad de adaptación de un ser humano. El problema aparece cuando el entorno en el que se tiene que realizar la tarea es especialmente hostil y/o remoto, como puede ser la manipulación de materiales radioactivos o tóxicos, el espacio exterior o las profundidades submarinas. Es en estas situaciones donde el concepto de teleoperación o telemanipulación adquiere sentido.

En la introducción de esta tesis se definieron los términos telemanipulación, teleoperación y telerobótica como tecnologías que permiten extender las capacidades humanas a entornos peligrosos o remotos. En [Sheridan92] y [Vertut85] se puede encontrar una revisión cronológica de los desarrollos en este campo y en [EURON04] los últimos avances en el mismo. Históricamente, se puede observar que los sistemas de telemanipulación son muy útiles y han sido ampliamente utilizados, principalmente porque permiten la interacción con entornos peligrosos o difícilmente accesibles para el ser humano. Precisamente en el año de publicación de esta tesis, todos los medios de comunicación se hacen eco de los éxitos de la NASA en la exploración de Marte gracias a la telerobótica espacial,

con los robots[1] Spirit y Oportunity (**Fig 4.1.**), que pueden enviar imágenes y realizar tareas de tele-inspección y experimentación sobre el terreno recibiendo órdenes desde la Tierra.

Fig. 4.1. Recreación artística del rover Spirit sobre Marte

En esta tesis se propone una arquitectura de referencia para el dominio de las unidades de control de robots. Para promover el éxito de dicha arquitectura, se centrará su dominio de aplicación en los robots de servicio teleoperados. El hecho de plantear arquitecturas demasiado "genéricas" para dominios más amplios del propuesto en esta tesis hace que no tengan éxito en la reutilización de las mismas en varios sistemas, tal como se verá en el estado de la técnica que se revisa en este capítulo. Al contrario, la propuesta de arquitecturas para dominios más concretos (como por ejemplo: tele-robots de exploración planetaria) garantiza el éxito de una familia de sistemas que se basa en dicha arquitectura.

En los dos capítulos anteriores se ha justificado el interés de la reutilización de componentes o arquitecturas para poder plantear sistemas tan complejos como éstos, garantizando los requisitos para los que están diseñados y que aprovechen las experiencias de desarrollos anteriores. En este capítulo se propone hacer un estudio del dominio de los controladores de robots de servicio teleoperados, utilizando para ello el estado de la técnica actual en dicho dominio. El resultado del análisis de dominio expresará los requisitos que se describen en el siguiente capítulo y que condicionan el diseño de la arquitectura de referencia específica de dominio, que se abordará en el Capítulo 6.

4.2. Análisis del dominio de aplicación

La obtención de una arquitectura de referencia específica de dominio implica realizar un **análisis** de su dominio de aplicación, que permita identificar, coleccionar, organizar y representar la información relevante para el dominio, basándose en el estudio de sistemas existentes y sus historias de desarrollo [Kang90]. Este estudio da como resultado la identificación de las partes comunes a todas las aplicaciones del dominio y es la base sobre la que se sustentan el diseño de una arquitectura genérica de dominio. Esta arquitectura genérica define una plataforma para todas las aplicaciones de la familia y provee una guía para construir un diseño genérico que pueda ser reutilizado y que facilite la implementación de sistemas bien estructurados, mantenibles e incluso modificables mediante mejoras o adaptaciones a nuevos requisitos funcionales.

En la tesis de B. Álvarez [Álvarez97-td] se realiza un análisis de dominio de los sistemas de teleoperación utilizando el método FODA (*Feature-Oriented Domain Analysis*) [Kang90] para plantear una arquitectura software de referencia para estaciones de teleoperación. Para realizar el

[1] El término inglés utilizado en la NASA para este tipo de robots es *rover.*

análisis de dominio según FODA, lleva a cabo una revisión de las arquitecturas y aplicaciones representativas en el dominio considerado, las cuales se analizan según los niveles de abstracción propuestos por [Kramer93] que muestra la **Fig. 4.2**. En esta tesis se seguirá el mismo proceso de acotación del dominio según los niveles de abstracción propuestos por [Kramer93], pero actualizando y ampliando el estudio de dominio hecho en [Álvarez97-td] y acotándolo para las unidades de control de robots de servicio teleoperados.

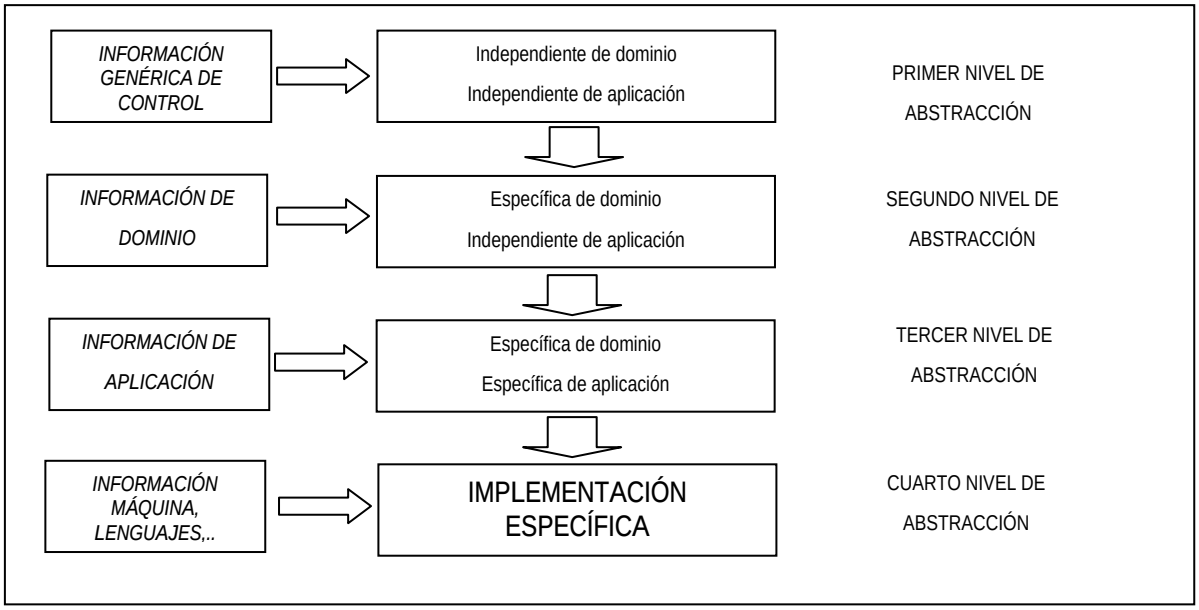


Fig. 4.2.- Niveles de abstracción para la definición de arquitecturas software

Para definir el primer nivel de abstracción (independiente del dominio y de la aplicación), se parte de las arquitecturas de control descritas en la literatura. Con ello se obtiene los aspectos de control que son comunes a todos los sistemas (vehículos, robots, maquinaria, etc.) en los que se asume únicamente que el sistema controlado interacciona con su entorno. Se presta especial interés en las arquitecturas de control para robots en general (normalmente autónomos).

La **teleoperación** (segundo nivel de abstracción) surge con la necesidad de llevar a cabo tareas no repetitivas de cierta complejidad, que no pueden ser programadas de antemano y que a menudo se desarrollan en ambientes hostiles o de difícil acceso. Un sistema teleoperado debe responder tanto a las órdenes del operador como a estímulos externos, integrando control en tiempo real, ya que en este tipo de sistemas, los requisitos temporales deben estar garantizados, para que el sistema pueda responder a estímulos externos en un plazo de respuesta limitado.

Por último, y con objeto de limitar el dominio a los sistemas de teleoperación y reducir la cantidad de información adicional necesaria para la implementación, se definen el tipo de aplicaciones para las que se va a utilizar la arquitectura (tercer nivel de abstracción), en este caso, los sistemas de control de robots de servicio teleoperados.

En los siguientes apartados, siguiendo el orden de acotación del dominio de aplicación definido, se revisan en primer lugar los sistemas de control existentes, con el fin de extraer los aspectos comunes que debería recoger la especificación de una arquitectura de control. A continuación se mencionan los aspectos de tiempo-real que se deben tener en cuenta en el diseño de arquitecturas de control en general y de sistemas de teleoperación en particular, se establecen las propiedades del dominio de la teleoperación y se realiza una acotación del mismo a los sistemas de control de robots de servicio teleoperados.

4.3. Arquitecturas de control de robots: Estado de la Técnica

En esta sección se revisan algunas arquitecturas de control de robots, haciendo una breve descripción de los aspectos más característicos de cada una de ellas. Con esto se abarca el primer nivel de abstracción del proceso de acotación del dominio expuesto en el punto anterior.

4.3.1. Clasificación de arquitecturas de control.

Un problema importante a la hora de plantear un estado de la técnica en arquitecturas de control, es cómo comparar diferentes arquitecturas y estilos arquitectónicos. La mayoría de artículos sobre arquitecturas son solamente descriptivos, muestran lo que hace el sistema y cómo está organizado, pero raramente proporcionan métricas que permitan determinar qué efecto tiene la arquitectura por sí misma en las prestaciones del sistema, si es que lo tiene.

Una aproximación tradicional usada para resolver la dificultad de clasificar y comparar varias arquitecturas, consiste en idear un esquema de clasificación que permita que las arquitecturas comparables sean agrupadas. En [Coste00] se hace un interesante repaso del estado de la técnica en arquitecturas para control de robots (principalmente autónomos) y se recogen los criterios de clasificación de arquitecturas más comunes.

4.3.1.1. Clasificación según la estructura del sistema

Se puede realizar una primera clasificación en cuanto a la estructura del sistema. Los sistemas robóticos suelen ser bastante complejos, y una de las maneras de abordar esta complejidad es dar una **modularidad** a la estructura del sistema, así se reduce la complejidad del sistema descomponiéndolo en componentes más pequeños con un nivel de abstracción bien definido y los interfaces necesarios entre dichos subsistemas. Según dicha modularidad sea mayor o menor, o bien los módulos estén conectados de una forma u otra, podemos hablar de distintas arquitecturas.

- Arquitecturas **centralizadas**
- Arquitecturas **jerárquicas**

La idea de una arquitectura centralizada es que un simple controlador se ejecuta en un ordenador para realizar directamente el control de todo el sistema. Generalmente, estas arquitecturas incluyen una base de datos centralizada con un protocolo de acceso sencillo y tienen la ventaja de no necesitar mecanismos de comunicación entre controladores. Ejemplos de este tipo de arquitecturas se pueden ver en la literatura clásica de control de robots [Hasemann95]. Sin embargo, la falta de modularidad de estas arquitecturas dificulta el mantenimiento del sistema e imposibilita la reutilización de los componentes, por lo que en la mayor parte de los trabajos realizados los controladores están arraigados en una jerarquía en la que interaccionan a través de un protocolo comando-estado.

La estructura de las arquitecturas jerárquicas tiene a menudo forma de árbol, en el que cada controlador tiene uno superior y de cero a muchos subordinados. Los sistemas jerárquicos se pueden ejecutar en un ordenador con un procesador, en un ordenador con varios procesadores o en varios ordenadores. Las ventajas más destacadas de una arquitectura de control jerárquica son:

- Modularidad: Cada controlador se puede tratar como un módulo software, incrementando la facilidad en el desarrollo, y haciendo el sistema más fácil de mantener mediante el uso de plantillas de código.
- Fácilmente extensible y reutilizable.
- Degradación controlada: Si algo va mal, sólo una parte del sistema tiene que parar.
- Facilita un mayor control de errores y en general, el diseño específico.

- Facilita la implementación y realización de pruebas.
- Permite la computación distribuida.
- Permite diferentes tiempos de operación: Generalmente, los controladores a nivel más bajo trabajan más rápido.
- Favorece el trabajo en equipos de desarrollo.

Los inconvenientes principales de estas arquitecturas estriban en la necesidad de comunicación entre los controladores, y la dificultad de depuración cuando ocurren errores en las interacciones entre ellos.

Sin embargo, mientras que la descomposición jerárquica de los sistemas robóticos se considera generalmente una "cualidad deseable", **todavía hay debate en cuanto a la dimensión a la que debemos llegar en esa descomposición modular** [Coste00]:

- ✓ Se podría realizar una descomposición según la dimensión temporal en la que opera cada módulo. Por ejemplo, en la arquitectura RCS-NASREM [Huang91], [Albus87], cada capa en la jerarquía opera a un orden de magnitud temporal más lento que la capa que hay por debajo (**Fig. 4.3**).
- ✓ También se pueden dividir las capas jerárquicas según el nivel de abstracción de las acciones que deben controlar (por ejemplo, las capas más bajas controlan un eje individual, mientras las capas superiores coordinan el comportamiento de varios ejes). Hay muchos ejemplos de este tipo de división jerárquica que se verán en el resto del capítulo: TCA [Simmons94], Laas [Alami00], ORCCAD [Borrelly98] y Subsumption [Arkin89].
- ✓ En algunas situaciones, la descomposición basada en la abstracción espacial puede ser más útil, como cuando se abordan problemas que conllevan navegación local y global, dividiendo así los niveles jerárquicos según se ocupen de una navegación local o una planificación global de la navegación.

La **principal lección** es que diferentes aplicaciones necesitan descomponer los problemas de diferentes maneras, y las arquitecturas necesitan ser suficientemente flexibles para acomodarse a diferentes estrategias de descomposición.

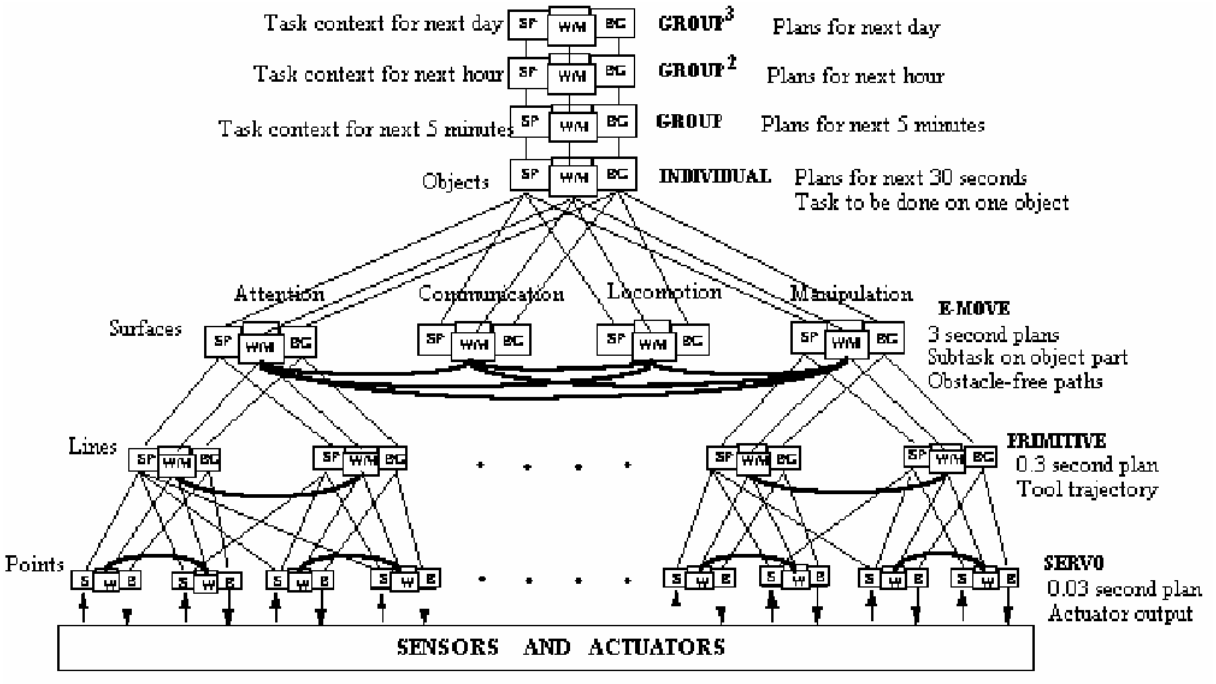


Fig. 4.3.- Descomposición jerárquica-temporal en RCS-NASREM [Huang91]

4.3.1.2. Clasificación según el tipo de control

Además de la estructura de organización de la arquitectura, hay tres aspectos de las arquitecturas de control que son importantes: control, comunicaciones e información. Estos aspectos son independientes, pero deben ser integrados. Sin embargo, cada arquitectura hace énfasis en un conjunto limitado de características, lo que dificulta la comparación de las arquitecturas existentes. Si prestamos atención a los aspectos de control para clasificarlas, podemos distinguir entre [Schlegel01]:

- Arquitecturas de control jerárquicas o **deliberativas**
- Arquitecturas de control **reactivas** y sistemas basados en el comportamiento
- Arquitecturas de control **híbridas**

A.- Arquitecturas de control jerárquicas o deliberativas

La aproximación clásica al diseño de sistemas de control es el estilo **jerárquico[2].** Las arquitecturas **jerárquicas** o **deliberativas** siguen un paradigma "Sensor-Plan-Acción" con una jerarquía de arriba a abajo en la organización de las actividades del robot (**Fig. 4.4**). El robot obtiene información del mundo mediante sus sensores, planea la acción y actúa. Las capas más bajas controlan el movimiento a un nivel físico, mientras las capas más altas planifican la acción a realizar. A cada paso, el robot planea explícitamente el siguiente movimiento. Para que esto sea posible, es necesaria una representación detallada del entorno, junto a una descomposición jerárquica de tareas, que posibilite la división de la tarea inicial en varias subtareas. Suelen primar el control a alto nivel y restringir las comunicaciones a bajo nivel.

Las arquitecturas que siguen este paradigma de forma estricta, **suelen ser lentas** y es prácticamente imposible actualizar la representación del entorno con la frecuencia necesaria. Esta manera de organizar el control suele tener poca flexibilidad y se ha adaptado con dificultad al control de las nuevas generaciones de robots, que tienen que manejar numerosos sensores en lazos de control reactivos.

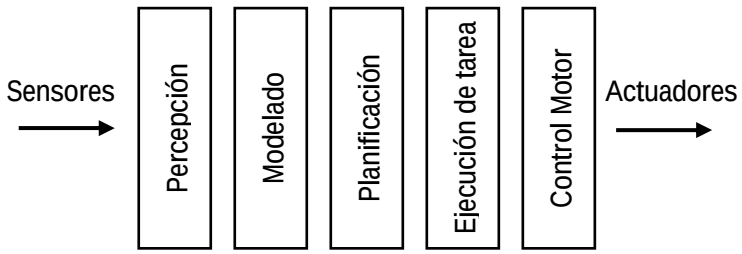


Fig. 4.4. Arquitectura jerárquica o deliberativa [Schlegel01

Un clásico ejemplo de este tipo de arquitectura es la clásica RCS/NASREM (NASA Standard Reference Modell for Telerobot Control System Architecture) [Albus87] desarrollada por la NASA/NIST para el control de robots en estaciones espaciales (**Fig. 4.3**).

El número de niveles jerárquicos depende de la aplicación. Por ejemplo, para el control de un robot móvil hay seis niveles, cada uno capturando una funcionalidad:

- El *nivel de misión*. Permite al operador introducir las misiones que debe cumplir el robot.
- El *nivel de tarea* planificará la misión del robot. Los planes serán enviados al nivel de movimiento elemental y dentro del mismo, al planificador local que se encargará de su ejecución.

---

[2] No confundir con la estructura jerárquica en cuanto a su organización estructural. En el caso del tipo de control, la jerarquía se establece en la división de las tareas a realizar, desde las más simples (control de motores) hasta las tareas más complejas de percepción.

- El *nivel de movimiento elemental*. Nivel intermedio entre el planificador global los niveles inferiores. Trabaja con datos más precisos y con tiempos menores que el planificador global.
- El *nivel de primitiva* tomará referencias externas que permitan periódicamente actualizar la posición del robot, y realizará la tarea de detección de obstáculos.
- El *nivel de servo*, transforma los puntos de una trayectoria en señales eléctricas para sensores y actuadores.

Aunque NASREM es ostensiblemente una arquitectura generalizada, su especificación funcional basada en un análisis de tareas y funcionalidades específicas, limita su generalidad. Específicamente, NASREM es útil solo cuando existen recursos de procesamiento y sensórica adecuados y las tareas dinámicas permiten una construcción de modelos y razonamientos simbólicos. En cuanto a la capacidad de procesamiento, merece la pena comentar que esta arquitectura presenta el problema del ancho de banda de las comunicaciones asociado a la memoria global. El proporcionar a cada proceso la capacidad de acceder a cualquier información contenida dentro de esta "caja negra", en lugar de a información local, trae como consecuencia el inconveniente de manejar un gran volumen de información en un tiempo limitado.

B.- Arquitecturas de control reactivas y sistemas basados en comportamiento

El control **reactivo** está basado en una fuerte conexión de los sensores con los actuadores en el lazo de control del robot. Los sistemas puramente reactivos no usan ninguna representación del entorno, por tanto almacenan una mínima información de estado. Consisten en una colección de reglas que hacen corresponder unas acciones específicas a unas situaciones concretas en que se encuentre el robot. Con esta aproximación, es difícil llegar a diseñar un robot que llegue a realizar tareas complicadas sin llegar a ser demasiado complejo y lento. Típicos ejemplos son los micro-robots desarrollados en los centros de enseñanza para motivar a los alumnos en la comprensión de la robótica. La *inteligencia* de estos robots suele ser puramente reactiva.

Los sistemas **basados en comportamiento** se pueden considerar una extensión del control reactivo. Los sistemas reactivos producen una salida que se puede medir externamente a partir de una interacción de sus reglas con un entorno particular. En contraste, los sistemas en base a comportamiento, especifican internamente esos comportamientos como "propiedades emergentes", como resultado de la interacción de comportamientos de alto nivel con el mundo. La actuación del robot se basa en una combinación o interacción de los comportamientos (**Fig. 4.5**). Estos comportamientos son típicamente de más alto nivel que las reacciones, lo que hace que los sistemas basados en el comportamiento no estén tan limitados como los sistemas reactivos.

Mientras que el principal problema del estilo de control jerárquico - deliberativo era el tiempo gastado en las comunicaciones entre capas, la principal desventaja del la aproximación basada en el comportamiento es la difícil composición de los distintos comportamientos.

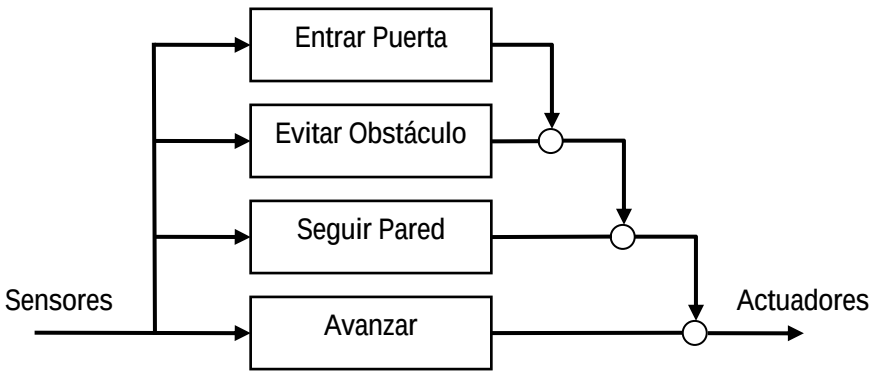


Fig. 4.5.- Arquitectura reactiva o basada en comportamiento [Schlegel01]

Un ejemplo de arquitectura de control basado en el comportamiento clásico en la literatura es la arquitectura **Subsumption** [Brooks86], del que se derivan otros trabajos posteriores que abordan la problemática de la combinación de comportamientos [Connell92b], [Posadas02]. En [Arkin98] se puede profundizar en el estudio de este estilo de control y se pueden consultar otros ejemplos de control reactivo y basado en comportamiento, además de un estudio más detallado de la arquitectura Subsumption.

Otra arquitectura de control de robots móviles autónomos con un estilo de control basado en comportamiento es **Saphira** [Konolige97], en ella se pueden ver reflejados también los aspectos claves mencionados en [Brooks86]. Saphira proporciona la arquitectura del sistema y del control del robot. La arquitectura del sistema proporciona un sistema operativo de micro-tareas y funciones para comunicarse con el hardware del robot. La arquitectura de control del robot contiene representaciones y rutinas para procesado de información de los sensores, representación del entorno y acciones de control del robot. Un rasgo distintivo respecto de otras arquitecturas es que proporciona una fusión eficiente de distintos comandos de comportamiento gracias a un enfoque de lógica *fuzzy*, de hecho los bloques constructivos de los comportamientos son reglas *fuzzy*.

Saphira es una arquitectura de robots móviles autónomos bastante madura, que es adoptada por varios robots comerciales (*Pioneer, Khepera*), es extensible, modular y portable a distintas plataformas gracias a su estructura cliente/servidor. También ha sido utilizada también en el *Jet Propultion Lab* para desarrollar un controlador de robots moviles teleoperados supervisados con respuesta segura ante entornos no-estructurados y peligrosos [Fong01].

C.- Arquitecturas de control híbrido

El control **híbrido** intenta adoptar los mejores aspectos del reactivo y del deliberativo. Típicamente conectan dos componentes distintos: uno reactivo y uno deliberativo, dándole diferentes escalas de tiempo a cada uno. Así, el componente reactivo maneja escalas temporales cortas, mientras que el componente deliberativo opera a escalas temporales más largas. Esto facilita el diseño de un control a bajo nivel eficiente, y además con conexiones a los razonamientos de alto nivel. La parte más delicada de estos diseños suele ser la conexión entre ambos niveles de control para conseguir una mezcla entre reactividad y deliberación.

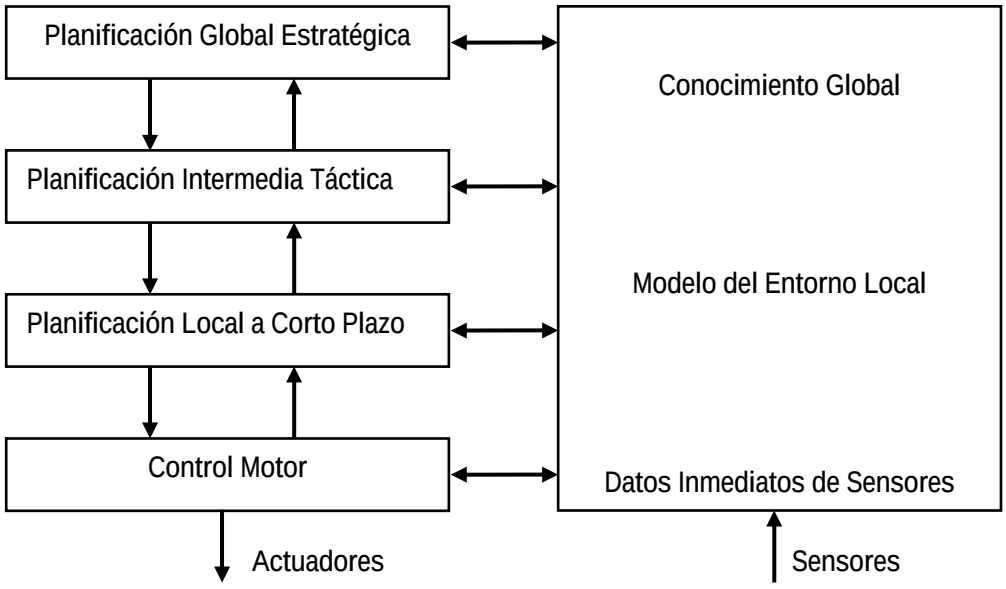


Fig. 4.6. Arquitectura híbrida [Schlegel01]

Varias arquitecturas híbridas ponen juntos los componentes reactivos de la arquitectura en un subsistema separado para asegurar que estas actividades reciben los recursos computacionales adecuados. Las actividades deliberativas que consumen tiempo son ejecutados en algún otro lugar o son planificadas como tareas no-críticas, con objeto de no interferir con las tareas de tiempo-real críticas [Beccari97]

SMARTSOFT [Schlegel99] es una arquitectura multicapa para robots móviles autónomos basada en un control híbrido, como claramente puede verse comparando las figuras **Fig. 4.6** y **Fig. 4.7**. Es similar a 3-Tiered Architecture [Bonasso97] que se verá más adelante. La principal diferencia estriba en el mecanismo de activación y desactivación de *habilidades*, que es más compleja. Además, tiende a ser más modular, soportando diferentes módulos como diferentes procesos, de forma que no compila la capa de *habilidades* completa en un solo proceso (ver **Fig. 4.7**).

La idea básica de la arquitectura del sistema consiste en la división del mismo en tres capas, como se muestra en la **Fig. 4.7**.

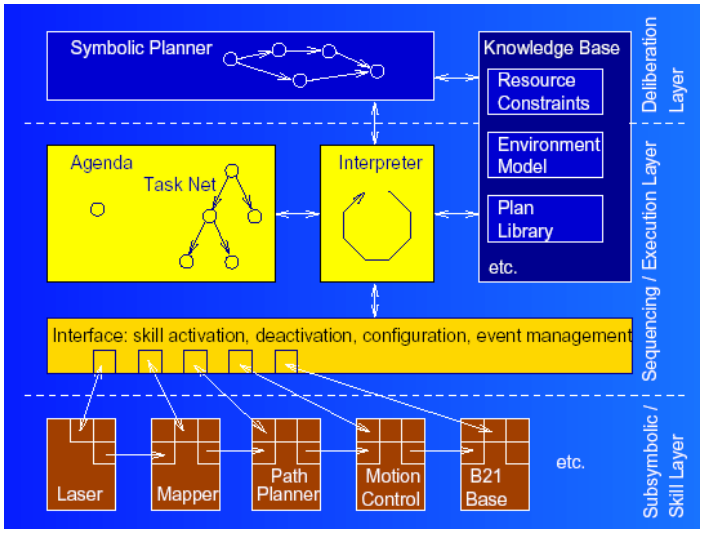


Fig. 4.7.- Esquema general de la arquitectura SMARTSOFT

- La capa *subsimbólica* (reactiva), consiste en una serie de módulos trabajando continuamente, que procesan datos de los sensores y generan comandos para actuadores. El tiempo de ciclo de estos algoritmos es suficientemente corto para asegurar una operación segura y permitir una rápida reacción ante los cambios en el entorno (parte reactiva del sistema).
- La capa *ejecutiva* es responsable de las actividades de coordinación de la capa subsimbólica. Esta capa es responsable de la coordinación de las actividades de la capa subsimbólica. Esto se hace seleccionando las configuraciones apropiadas y teniendo en cuenta los conflictos de recursos disponibles el contexto de ejecución actual. Esta capa sólo funciona en estados discretos que son usados para sincronizar la ejecución en el mundo real con la descripción simbólica del proceso de ejecución esperado.
- La capa *deliberativa* contiene algoritmos que consumen mucho tiempo. Normalmente las principales transiciones ocurren en la capa ejecutiva mientras que la deliberativa está generando una respuesta.

La arquitectura contiene también una base de conocimiento accesible vía un sencillo interfaz de llamada-respuesta. Cada módulo se representa con sus parámetros configurables y los recursos que proporciona.

Como se puede deducir de los párrafos anteriores, en las arquitecturas de control híbridas los comportamientos reactivos y deliberativos se combinan, lo cual incide en los aspectos de tiempo real de estas arquitecturas, puesto que en ellas es normal que comportamientos de tiempo-real no-críticos (normalmente la parte deliberativa), convivan con acciones de tiempo-real críticas, como por ejemplo

en la arquitectura **CIRCA** [Musliner93]. Ésta y otras arquitecturas híbridas, con sus especiales consideraciones de tiempo-real se comentan en el siguiente punto, donde se da un paso más en la acotación del dominio de aplicación de la tesis al estudiar las necesidades de tiempo-real en los sistemas robóticos.

4.3.2. Necesidades de Tiempo-Real en los sistemas robóticos

En la mayoría de los sistemas de control de robots existen tareas o procesos críticos, si se pierde un *deadline* de algún proceso de tiempo-real, se producirá un funcionamiento erróneo del sistema completo, e incluso se podrían originar daños a personas o cosas, dependiendo de las condiciones de seguridad del sistema. En algunos casos, la pérdida ocasional de *deadlines* puede afectar sólo en la calidad de la ejecución de una tarea concreta (por ejemplo, la suavidad en recorrer una trayectoria). Sin embargo, las pérdidas sistemáticas de *deadlines* si que puede afectar al funcionamiento del sistema y hacer que el sistema falle.

Es habitual que el sistema robótico tenga algunos procesos *soft real-time* y otros *hard real-time*, que la mayoría de las veces se tenderá a separar. Hay que tener en cuenta que las propiedades temporales de un sistema de tiempo real depende de la apropiada cooperación de todos los niveles de un diseño: hardware, comunicaciones, sistema operativo y la propia aplicación. El diseño de sistemas *hard real-time* es distinto del diseño *soft real-time* o de sistemas que no son de tiempo real. Así mismo, las arquitectura de un sistema de tiempo-real está fuertemente influenciada por las capacidades y la relación prestaciones/coste de los componentes hardware disponibles.

Sin embargo, a pesar de la presencia de aspectos de tiempo-real en la mayoría de los sistemas robóticos, muchas propuestas de arquitectura no consideran explícitamente las cuestiones de tiempo-real. Simplemente suponen que el sistema que implemente dicha arquitectura será capaz de ejecutar las tareas de manera suficientemente rápida.

Una de las arquitecturas de tiempo-real clásicas de la literatura es la arquitectura de referencia para sistemas de control de tiempo real (**RCS**) desarrollada por el NIST, y que ya ha sido comentada al hablar de arquitecturas jerárquicas. Además de ésta son referentes típicos las arquitecturas **CIRCA**, **TCA, PRS, RAP** y **3T**.

CIRCA (*Cooperative Intelligent Real-time Control Architecture*)[Musliner93]. En las arquitecturas híbridas "cooperativas" como esta, la capa deliberativa juega un papel activo en el proceso de asegurar que los *deadlines* se cumplen en la parte reactiva. Una realimentación desde la capa reactiva permite que el subsistema de Inteligencia Artificial razone sobre la lógica y las restricciones temporales. Así se pueden tomar las decisiones adecuadas para evitar condiciones operativas incorrectas, adaptando el conjunto de reflejos, cambiando los parámetros de tiempo-real, o dando una escala distinta a los objetivos a nivel de tarea (por ejemplo, disminuyendo la velocidad de navegación del robot). Como las capas reactivas y deliberativas están desacopladas, la ejecución de tiempo-real está siempre garantizada en la capa reactiva, mientras que la flexibilidad y la robustez a nivel de tarea se demanda a la capa deliberativa. Esta idea persigue el diseño de un sistema que pueda razonar acerca de las necesidades de tiempo-real y sus propias limitaciones computacionales, con objeto de poder adaptar la computación a los eventos entrantes.

TCA (*Task Control Architecture*) [Simmons94], introduce un controlador de tiempo-real e incluye un análisis preliminar del consumo de tiempo de las tareas. Por otra parte, las aproximaciones típicas orientadas a Inteligencia Artificial abordan los aspectos de tiempo real sólo indirectamente. Por ejemplo, **PRS** (Procedural Reasoning System) [Georgeff89] y **RAP** (Reactive Action Package) [Firby95] proponen mecanismos de planificación reactivos que prestan atención a las condiciones modificadas en el entorno durante el plan de computación. La **3-Tiered Architecture** [Bonasso97] explota **RAP** para implementar el secuenciador, un módulo reactivo que habilita o deshabilita una serie de "habilidades" (comportamientos de bajo nivel) basadas en la dinámica del entorno. Esta adaptación del subsistema reactiva a la tarea en ejecución y ayuda también a reducir la carga computacional global. Mientras que estas aproximaciones aseguran la reactividad en casos de

aplicación específicos, no abordan explícitamente la prioridad entre acciones ni evalúan las restricciones de tiempo-real que deberían tenerse en cuenta en el subsistema reactivo.

Algunas arquitecturas permiten a los diseñadores especificar la frecuencia a la que los comportamientos se deberían ejecutar, y de ese modo, poder planificar su ejecución para satisfacer las restricciones temporales. Por ejemplo, **CIRCA** hace esto usando un planificador explícito en conjunto con un sistema operativo de tiempo real. **ORCCAD** [Simon95] y **ControlShell** [Schneider94] también proporcionan soporte de tiempo-real planificando la ejecución a diferentes frecuencias.

En [Kopetz00] se puede encontrar una interesante reflexión sobre las tendencias en el diseño de sistemas de tiempo-real y las tendencias tecnológicas en este campo. Kopetz concluye que la estructura básica de los sistemas de control de tiempo-real tienden a ser un sistema de tiempo real distribuido, tolerante a fallos, consistente en una serie de nodos de procesamiento que pueden ser sistemas empotrados (SOC) con sus dispositivos periféricos y sensores inteligentes, y finalmente un sistema de comunicación, como por ejemplo un bus de tiempo-real.

Un interesante ejemplo de aplicación de estas tendencias en sistemas de tiempo-real empotrados y distribuidos se puede encontrar en [Barreca02], donde se presenta una arquitectura de control de tiempo-real compuesta por una serie de módulos (sistemas empotrados) intercomunicados con un bus de campo. Se recomienda también leer [Redell98] por su detallado estudio de este tipo de sistemas, incluyendo la comunicación con CAN-bus y una discusión entre dos posibles arquitecturas hardware.

El propio Kopetz propone la *Time-Triggered Architecture* (**TTA**) [Kopetz02] como una arquitectura que proporciona la infraestructura computacional para el diseño e implementación de sistemas empotrados de tiempo-real fiables. En su libro *"Real-Time Systems: Design Principles for Distributed Embedded Applications"* [Kopetz97], se puede consultar un exhaustivo estado del arte en tecnología de sistemas de tiempo-real.

Sistemas operativos de Tiempo-Real

No es objeto de esta tesis el estudio de los Sistemas Operativos de tiempo-real existentes, en [Clawar01-t13] se puede leer un resumen de los principales SO utilizados en robótica. Entre ellos destacan **RT-Linux** y **RTAI**, muy utilizados a nivel de investigación en las Universidades, y cada vez más utilizados en la industria por distribuirse bajo licencia GNU, aunque existen también distribuciones comerciales; se pueden ver interesantes aplicaciones en [Macchelli00], [Macchelli02], [Vidal02], [Bellini02]. Dentro del mundo del software de libre distribución, hay que mencionar las iniciativas **ORK** [Zamorano02] y **MaRTE-OS** [González01] como sistemas operativos para tiempo real en continua evolución, investigación y mejora. Otro sistema operativo, esta vez comercial, es **QNX**, muy flexible e ideal para aplicaciones de tiempo-real. Finalmente, cabe nombrar a **VxWorks** como el SO de tiempo real más ampliamente adoptado para la industria de sistemas empotrados.

En [Beccari97] se mencionan algunos SO especialmente concebidos para robots autónomos, como **CHIMERA** y **HARTIK**. Junto con CHIMERA surge también una iniciativa para constituir un *framework* de componentes basados en puertos en entrada y salida, que sean reconfigurables en cuanto a su disposición y relación, para formar un sistema de control distinto ante cambios en el robot que está controlando (cambios en los sensores, en la configuración del robot, etc) sin necesidad de recompilar e incluso sin necesidad de parar el sistema, es decir, apostando por una reconfiguración dinámica, todo esto dentro de un entorno de tiempo-real [Stewart97]. En este artículo se plasma la diferencia entre reconfiguración estática y dinámica y entre componentes generales y componentes modulares.

4.3.3. Últimas tendencias en arquitecturas de control para robots

A los robots actuales se les exige que se desenvuelvan cada vez en más situaciones y que realicen de forma eficiente tareas cada vez más complejas. Hay una creciente demanda de que tales sistemas realicen no sólo una tarea, sino una serie de operaciones distintas ejecutadas en entornos dinámicos,

dinámicos, no-estructurados y que además lo hagan durante largos periodos de tiempo sin interrupción. Además se busca que puedan ser reconfigurados, ampliados según nuevas necesidades.

Sin embargo, en la actualidad, los sistemas de control e interfaces proporcionados por los distintos fabricantes de robots comerciales son propietarios y a menudo específicos de una unidad concreta. Esto es un obstáculo para la integración en los procesos de automatización que causa costes considerables durante las fases de puesta en marcha y configuración, además de los obstáculos comentados en su posible modificación o reconfiguración. Por todo ello hay un interés creciente en las arquitecturas modulares y *frameworks* de componentes que fomentan la creación de sistemas abiertos fácilmente modificables y ampliables. Este tipo de iniciativas suscita especial atención en el ámbito de la mecatrónica y la robótica de servicio, donde los sistemas se diseñan específicamente para una aplicación concreta, y por lo tanto interesa mucho que puedan ser modificados en su vida operativa ante cambios en la aplicación o ante nuevos requisitos.

4.3.3.1. Arquitecturas modulares orientadas a objetos y a componentes

Quizá los últimos y más exitosos trabajos realizados en arquitecturas orientadas a objetos sean los trabajos llevados a cabo en el JPL (*Jet Propulsion Laboratory, California Institute of Technology*) para los *rovers* que la NASA ha venido utilizando para la exploración de Marte (desde el *Mars Pathfinder* hasta los actuales *Spirit* y *Opportunity*). Son buenos ejemplos de arquitecturas y *frameworks* para desarrollar sistemas robóticos autónomos o teleoperados de manera supervisada. Se observa una tendencia en incrementar la autonomía de estos *rovers*, de forma que para realizar una misión no dependan exclusivamente de las órdenes recibidas desde la Tierra, dado que por la distancia hasta Marte, el tiempo de retardo para que llegue una orden es bastante grande. Uno de los últimos trabajos para aumentar esta autonomía es la arquitectura **CLARAty**[3] [Nesnas03].

Se puede observar en la mayoría de las arquitecturas para robots autónomos, como las presentadas anteriormente [Schlegel99], [Bonasso97], que suelen tener tres niveles: *Funcional*, *Ejecutivo* y de *Planificación*, ordenados desde la parte más reactiva y próxima a los sensores y actuadores hasta la capa más deliberativa, que realiza la planificación. Una diferencia entre la arquitectura CLARAty las arquitecturas convencionales de 3 capas es la distinción entre niveles de *granularidad* y de *inteligencia*. Las arquitecturas convencionales ponen la granularidad y la inteligencia en el mismo eje, es decir, conforme nos movemos hacia abstracciones más altas del sistema, la inteligencia también se hace más alta. Esto no es cierto en CLARAty, donde la inteligencia y la granularidad están en ejes diferentes. En otras palabras, la descomposición del sistema permite un comportamiento inteligente a niveles más bajos, manteniendo la estructura de los distintos niveles de abstracción. Se puede decir que mantiene un concepto similar a algunos sistemas híbridos.

CLARAty es una arquitectura para robots específica de dominio diseñada con cuatro objetivos principales:

- Reducir las necesidades de construir una nueva infraestructura robótica para cada nuevo esfuerzo de investigación.
- Simplificar la integración de nuevas tecnologías en los sistemas robóticos ya existentes.
- Acoplar fuertemente los algoritmos a nivel declarativo y basados en procedimiento.
- Operar un número heterogéneo de rovers con diferentes capacidades físicas y arquitecturas hardware.

Como se puede observar, estos objetivos coinciden con las intenciones de los últimos desarrollos de arquitecturas para robots y en concreto, con algunos de los objetivos marcados en esta tesis doctoral.

---

[3] *Coupled Layered Architecture for Robotic Autonomy* – Arquitectura acoplada por capas para autonomía de robots.

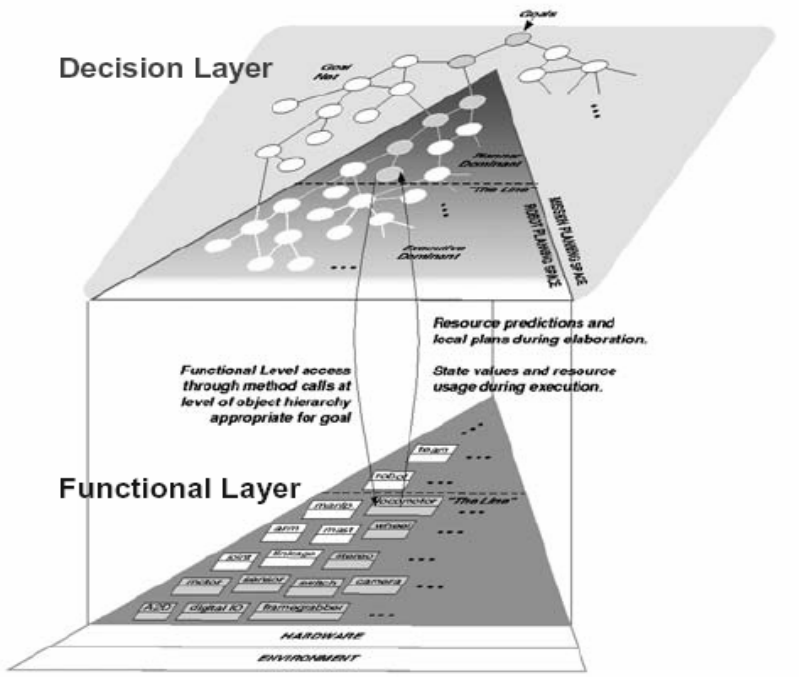


Fig. 4.8.- Visión conceptual de la arquitectura CLARAty

CLARAty proporciona un *framework* basado en componentes genéricos y reutilizables que pueden ser adaptados a un número de plataformas robóticas heterogéneo. CLARAty consiste en dos capas distintas: una Capa Funcional (*Functional Layer)* y una Capa de Decisión (*Decision Layer*). La primera define varias abstracciones del sistema y adapta los componentes abstractos a dispositivos reales o simulados. Proporciona también un marco de trabajo y algoritmos adecuados para la autonomía de bajo y medio nivel. La Capa de Decisión proporciona la autonomía del sistema a alto nivel, con la cual razona acerca de los recursos globales y las restricciones de la misión. Esta capa accede a información de la Capa Funcional según múltiples niveles de granularidad.

Una interesante cualidad de la Capa Funcional es que está orientada a objetos [Nesnas01], de forma que abstrae y proporciona interfaces de los dispositivos que el robot debe controlar, especializando las clases definidas y extendiendo los interfaces según la necesidad mediante mecanismos de herencia y composición. Esta orientación a objetos puede ser estructurada para coincidir directamente con la modularidad anidada del hardware y permite que la funcionalidad básica y la información de estado de los componentes del sistema sean codificada y compartimentada. Todos los objetos contienen una funcionalidad básica por si mismos que es accesible desde otras partes de la Capa Funcional y también directamente desde la Capa de Decisión

Como ejemplo de la especialización en la orientación a objetos, un *vehículo* proporciona un interfaz para cualquier tipo de plataforma móvil, ya sea con ruedas, con patas o híbrido. Una especialización funcional de *vehículo* es el *vehículo con ruedas*. Esta especialización introduce le concepto de la movilidad con ruedas y la configuración de éstas. Esta especialización funcional extiende la interfaz de *vehículo* para incluir capacidades adicionales. En la **Fig. 4.9** se puede ver otro interesante ejemplo de especialización en este caso, de la herramienta manipuladora que llevaría el *rover*.

Por no extender demasiado este capítulo, se remite al lector a que consulte las referencias [Nesnas01], y [Nesnas03], y se pasa a profundizar en la tendencia actual hacia los *frameworks* de componentes.

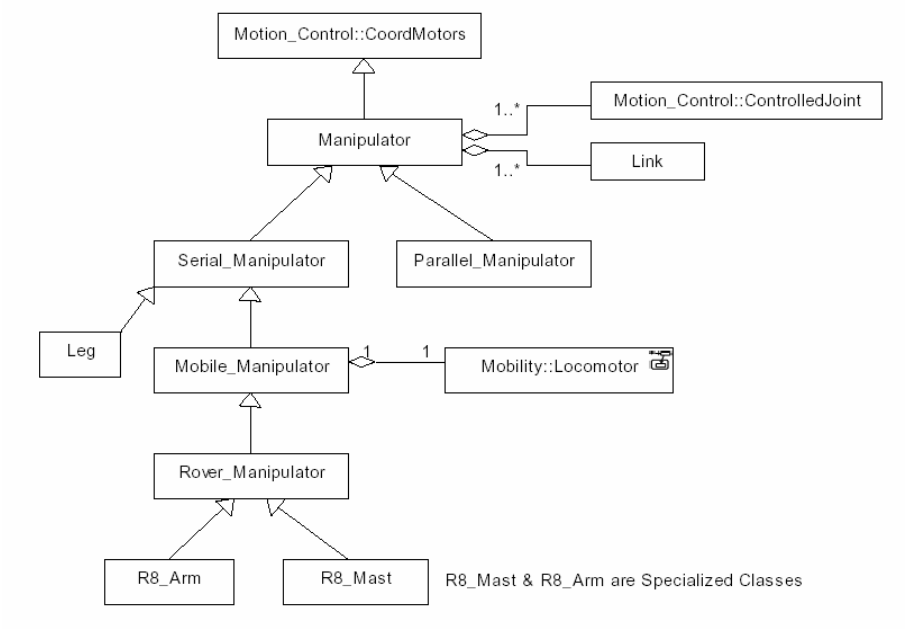


Fig. 4.9.- Especialización en la clase manipulador

4.3.3.2. Frameworks de componentes

Una de las principales tendencias actuales en investigación y propuesta de arquitecturas de control de robots se basa, no ya en la proposición de una determinada arquitectura para un robot concreto, sino en el desarrollo de *frameworks de componentes* que ofrezcan una serie de módulos o componentes fundamentales con interfaces bien definidos, servicios ofertados y requeridos, las reglas para combinarlos, los patrones de interacción entre ellos, e incluso herramientas para configurar distintas arquitecturas según sean los requisitos del sistema a implementar. De hecho, la causa principal de que muchos robots se queden como prototipos de laboratorio es que suele ser muy difícil compartir y reutilizar el software que ha hecho posible esos robots debido a que, en general, hay una falta de especificación estándar para el desarrollo de software para robots [Mallet02b].

Los creadores de *frameworks* de componentes defienden la *flexibilidad* de contar con componentes para construir distintas arquitecturas. Esta flexibilidad es entendida principalmente en los siguientes aspectos:

- La posibilidad de reemplazar unos componentes por otros alternativos con el mismo interfaz pero con diferentes implementaciones.
- El hecho de que gran parte de la infraestructura diseñada puede ser reutilizada y simplemente sean necesarias ciertas extensiones para ejecutar aplicaciones nuevas o de propósito especial.
- La posibilidad de reutilizar el mismo diseño pero optar por diferentes optimizaciones. Por ejemplo: *hard real-time*, distribución a gran escala, reducción en la capacidad de procesamiento disponible, distribución en distintos procesadores, acceso seguro, gran número de ejes sincronizados, reducida capacidad de comunicación, etc.

Algunos fabricantes de robots han propuesto *frameworks software*, que son un intento hacia una especificación de sistemas robóticos. Son notables las contribuciones comerciales de *ABB* y Adept [Adept91], *iRobot* y su *framework* Mobility [Mobility02] o *ActivMedia* y Saphira [Konolige96]. Sin embargo, estos *frameworks* no son demasiado *abiertos* (el código normalmente es propietario) ni extensibles, en el sentido de que fueron hechos para los pocos robots vendidos por estas compañías. Con estos *frameworks* se pueden concebir robots muy complejos y autónomos, pero normalmente sus misiones y sensores son fijos y predefinidos.

En este entorno y con la perspectiva de resolver estos problemas, los creadores de **SMARTSOFT**, junto con los creadores de **ORCCAD** y la Universidad belga de Leuven, proponen dentro del marco de **EURON** (Red Europea de Robótica [EURON00]) el proyecto **OROCOS** (Open Robot Control Software) [OROCOS].

OROCOS es un *framework* modular, distribuido configurable, *soft* y *hard* real-time para el control de movimiento avanzado [Bruyninckx02]. Este *framework* separa la estructura del control (por ejemplo, la subdivisión en distintos hilos de control, comunicación entre procesos, manejo de eventos, distribución entre nodos de procesamiento, funcionalidad requerida del SO de tiempo-real, etc) de su funcionalidad (por ejemplo, interpolación del movimiento, algoritmos de control, procesamiento de sensores, et). Proporciona componentes independientes de la arquitectura que se pueden agrupar para formar aplicaciones de control de movimiento. En el artículo "*A Software Framework for Advanced Motion Control"* [Bruyninckx02], el lector puede comprender los fundamentos de OROCOS y su línea de trabajo.

Hay que destacar que aunque el *framework* de componentes no propone una arquitectura, en las aplicaciones de ejemplo que proporciona, se basa en la arquitectura **LAAS**, que al igual que SMARTSOFT, se trata de una arquitectura multi-capa (decisión – ejecución – funcional) [Mallet02a]. Los componentes diseñados en OROCOS, por lo menos hasta la actualidad, se centran en la capa funcional.

Las posibilidades que ofrecen estos *frameworks* de componentes son enormes, y llevan a su máxima expresión la idea de hacer sistemas modulares, reconfigurables, portables y extensibles. Por otra parte, los principales problemas que el autor de esta tesis observa en *frameworks* como OROCOS son su generalidad y el intento de abarcar dominios muy amplios (OROCOS da un dominio de aplicación tan amplio como "desarrollo de controladores de movimiento"). OROCOS nace con la idea de construir un *robot genérico*, por lo menos a nivel de software, pero el hecho de ser tan genérico puede ser un arma de doble filo, puede solucionar muchos problemas, pero precisamente por ser muchos pueden quedar resueltos a medias. Otro inconveniente importante y que repercute en su aplicación en dominios amplios es la granularidad de los componentes: adoptan una micro-arquitectura para cada tipo de componente, con una granularidad definida; el hecho de que sea obligatorio combinar dichos componentes para generar una arquitectura más amplia (del sistema global) hace que el rendimiento obtenido pueda no ser precisamente el deseado, aunque bien es cierto, que se debe adoptar el compromiso de llegar a perder cierto rendimiento a costa de aumentar la posibilidad de reutilización y reconfiguración. Finalmente, como se dijo en el Capítulo 2, desarrollar *frameworks* de componentes es una tarea muy compleja en la que deben estar implicados varios centros de investigación con dilatada experiencia en el desarrollo de productos software para robots.

En la actualidad OROCOS tiene una primera versión que se puede descargar de su web (www.orocos.org) y pretenden en un futuro incluir una herramienta visual de prototipado rápido. Todavía está en desarrollo y no se conocen en la actualidad implementaciones de este *framework.*

Hay pocas iniciativas conocidas parecidas a OROCOS que hayan propuesto *frameworks* de componentes de código abierto para control de robots, sólo cabe destacar aquí:

MCA-2 (*Modular Controller Architecture*) es un *framework* de tiempo-real, modular, transparente a la red para controlar robots [Scholl01], [MCA2]. Surge por la necesidad de sus desarrolladores de MCA-2 de contar con una plataforma software común para todos los robots que desarrollan, en su mayoría programados en C++ sobre plataformas RT-Linux. MCA está estructurada según la clásica estructura de control jerárquico. En el nivel más bajo residen las unidades de interfaz con el *hardware*, y según se asciende más en la jerarquía del sistema, las unidades se hacen más abstractas. En el sistema existen dos tipos de flujo de datos: los datos de los sensores y los datos de control, estos flujos de datos circulan entre los distintos módulos del sistema. Según se organicen dichos módulos, se tendrá la arquitectura necesaria para un sistema concreto. Todos los métodos están integrados en módulos estandarizados con interfaces unificados, a su vez, estos módulos se pueden combinar en grupos gracias a una herramienta de prototipado rápido y tras compilarlos, se pueden ejecutar en RT-Linux, Linux o Windows.

TCA (*Task Control Architecture*), ya mencionado anteriormente [Simmons94]. No es un *framework* de componentes en el sentido de OROCOS. TCA simplifica la construcción del control a nivel de tarea de robots móviles. Se entiende por "nivel de tarea" la integración y coordinación de la percepción, planificación y control de tiempo-real para alcanzar una serie de objetivos requeridos (tareas). TCA proporciona un *framework* de control general y pretende ser utilizado para una gran variedad de robots. Se podría definir como una especie de ORB (object request broker). Basada en un control estructurado, incorpora componentes de manejo de excepciones por capas y componentes de planificación de alto nivel. Incorpora restricciones temporales que asegura la respuesta a excepciones dentro de un intervalo de tiempo. Implementa un controlador central encargado de todas las operaciones, y un buffer entre sensores y actuadores. Las decisiones de control están explícitamente representados en TCA. Un problema de TCA es que la estructura del controlador a bajo nivel se deja sin especificar. También se dejan las tareas de concurrencia, monitorización y manejo de errores para añadirse después de que el sistema es capaz de tratar las situaciones nominales.

Fuera del dominio de la robótica, es más típico encontrar *frameworks* de componentes orientados a control de tiempo-real en general, con código abierto y para ejecutarse sobre RT-Linux, Linux o RTAI, como son **OSACA** (*Open System Architecture for Controls within Automation Systems*) [OSACA01], *Open Modular Architecture Controls* [OMAC00], desarrollada en el NIST o *Open Components for Embedded Real-Time Applications* [OCERA03], en la que participa el DISCA de la Universidad Politécnica de Valencia.

Prototipado rápido basado en frameworks de componentes

En el entorno de la visión por computador es usual encontrar bibliotecas de componentes, muchas de ellas comerciales, que permiten realizar sobre una imagen el procesamiento deseado, simplemente haciendo entrar la imagen a un *pipeline* de procesamiento, formado por estos componentes, y obteniendo como resultado la imagen procesada.

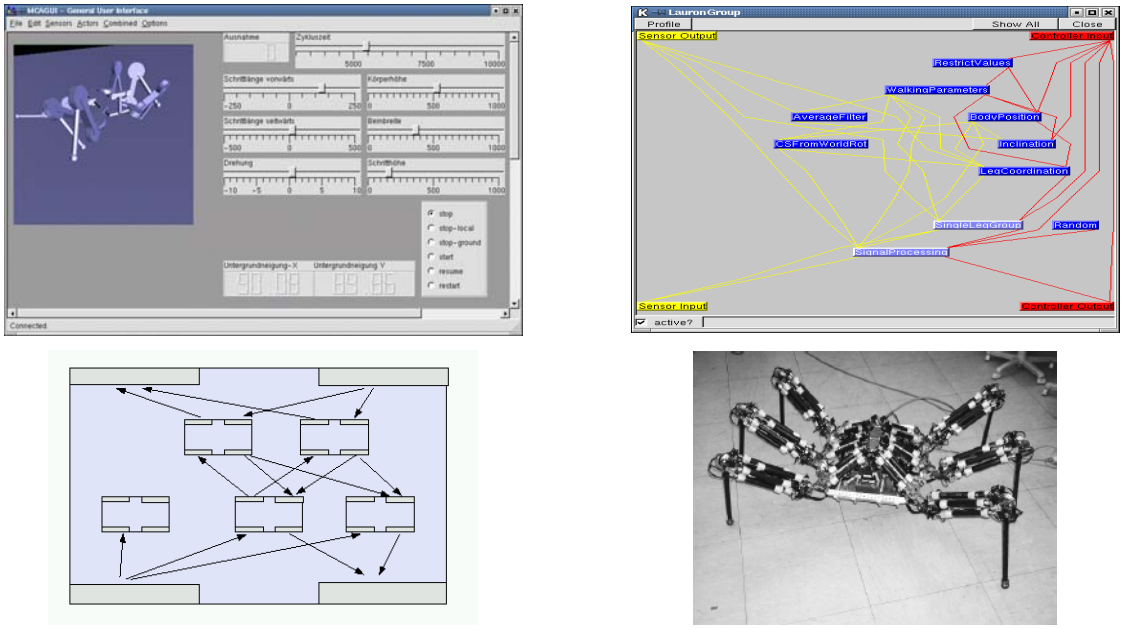


Fig. 4.10.- MCA-2: Editor gráfico de estructura de control de robot y ejemplo de aplicación

En el dominio del diseño de unidades de control para robots a partir de arquitecturas de referencia o de *frameworks* de componentes, sería deseable poder realizar este mismo proceso, en este caso, para poder "ensamblar" los componentes del *framework* robótico como si fueran piezas de mecano, pero por las características especiales del dominio, se hace más difícil abordar una solución como la mencionada para visión por computador.

No obstante, existen algunas propuestas que proporcionan entornos gráficos de programación para especificar estrategias de control. A partir de estos lenguajes gráficos se puede generar código automáticamente. Arquitecturas que utilizan esta técnica son ControlShell, ORCCAD y Labview. [Coste00]. En [Cox98] se puede ver también un interesante ejemplo de lenguaje visual para programación de robots. OROCOS y en MCA-2 proporcionan entornos visuales de programación y configuración de control de robots (en OROCOS está actualmente en desarrollo) que se apoyan en el *framework* de componentes (**Fig. 4.10**).

4.4. El dominio de los robots de servicio teleoperados

El último paso en la acotación del dominio de estudio de la tesis se produce al estudiar las características de los controladores de robots en los robots de servicio teleoperados. Para poder realizar en el capítulo siguiente el análisis de requisitos de este dominio, además de presentar ejemplos de arquitecturas, se expondrán las características propias de este tipo de sistemas, de su entorno y del tipo de tareas que desempeñan.

4.4.1. Caracterización del dominio de aplicación

La teleoperación, a lomos de los avances de la robótica, es una disciplina en auge. Trabajos que anteriormente no se realizaban o se llevaban a cabo en condiciones penosas son ahora posibles mediante el uso de mecanismos que pueden ser operados remotamente desde zonas seguras. La preocupación por las condiciones de trabajo de los operarios y por la seguridad de las instalaciones son cada día mayores y en consecuencia aumenta el número de aplicaciones posibles. Sin embargo, aún queda casi todo por hacer.

La tendencia actual en el campo de la robótica es intentar sacar a los robots de los talleres y darles un cierto nivel de inteligencia que les permita trabajar en diversos tipos de entornos, menos estructurados, y en operaciones no repetitivas. Para ello es necesario realizar un avance importante en cuanto a sensores, sistemas de visión y navegación, y lo que es más importante, un avance de las técnicas de inteligencia artificial que permitan la integración y el comportamiento coherente de todos los subsistemas. En el otro extremo se encuentran los manipuladores teleoperados, a los que actualmente se les pretende dotar de cierto grado de autonomía para la realización automática de determinados procedimientos. En ellos se requiere avanzar en los sistemas de visión, en los interfaces hombre-máquina, que son cruciales en teleoperación, y en los controladores. En definitiva, el objetivo final de la robótica y la teleoperación es el mismo: total flexibilidad con completa autonomía (¿el ser humano?), aunque el camino por el que ambas han avanzado ha sido hasta ahora totalmente distinto, pero no divergente.

4.4.1.1. Características de los robots de servicio

La diferencia fundamental entre un robot industrial y un robot de servicio es que este último está específicamente diseñado para realizar tareas muy especificas de mantenimiento, reparación e inspección de instalaciones, sistemas o componentes (inspeccionar y limpiar conducciones de aire acondicionado, inspeccionar y reparar componentes principales de una planta industrial, etc.), mientras que el diseño de los robots industriales se orienta a realizar tareas muy repetitivas en entornos muy diferentes (**Fig. 4.11**).

Desde el punto de vista físico ambos se componen de una serie de elementos interconectados entre sí y que componen su anatomía. Estos elementos van desde su unidad de control hasta sus elementos terminales (pinzas o herramientas). Entre ambos extremos están los accionadores, actuadores, sistema de transmisión y la propia estructura mecánica del robot. Sin embargo, existen algunas diferencias. En el caso de los robots de servicio el diseño mecánico global del mismo está orientado a la aplicación final. Mientras que en los robots industriales se parte de un robot de una tipología más o menos

general (robot cartesiano, polar, antropomórfico, *scara*, etc.) y es la herramienta final la que se adapta al proceso.

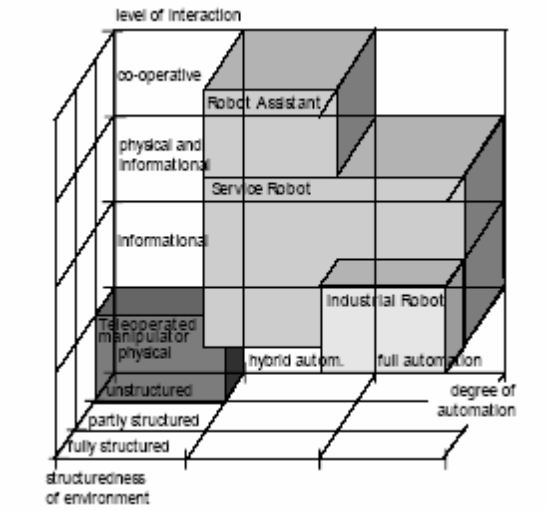


Fig. 4.11.- Características del entorno, grado de automatización y nivel de interacción en robots [EURON04]

Desde el punto de vista funcional, en la mayor parte de las ocasiones los robots industriales son programados una sola vez para realizar una tarea repetitiva. En el caso de los robots de servicio la situación es muy diferente. Los entornos no suelen estar claramente determinados, y en muchas ocasiones pueden estar cambiando durante el servicio en cuestión. En [EURON04] se puede consultar un extenso estado de la técnica de los robots de servicio. En este informe se ponen ejemplos de aplicación de robos de servicio en las áreas de medicina, rehabilitación, limpieza, construcción, servicios militares, rescate, agricultura, y otros servicios.

Puesto que en los robots de servicio el entorno puede ser muy cambiante, suelen ser sistemas teleoperados con un determinado grado de automatización. No solo en el entorno de los robots de servicio se da esta situación, en general, el mayor problema de la automatización de un proceso se encuentra en poder reaccionar ante eventos inesperados. La supervisión e incluso la intervención de un operador en el lazo de control, provee las ventajas de la detección *on-line* de posibles variaciones en el entorno y corrección de fallos. Los operadores también pueden usar su inteligencia para decidir entre distintas estrategias de ejecución de tareas.

4.4.1.2. Propósito de los Sistemas de Teleoperación

Atendiendo al propósito de los sistemas de teleoperación, y recordando la definición:

*"Los términos **telemanipulación, teleoperación** y **telerobótica** son términos que indican la capacidad de llevar a cabo operaciones en un entorno remoto; de hecho, el usuario de estos sistemas es capaz de interactuar con objetos remotos y ejecutar tareas impredecibles a priori y/o no repetitivas en ambientes hostiles o inaccesibles, sin necesidad de estar físicamente próximo a ellos. Se tiene que seguir contando con la inteligencia de un operador, que es el que toma las decisiones o al menos las más críticas"*

Se pueden extraer las palabras clave asociadas al término teleoperación, como la posibilidad de realizar trabajos apartando al operador de entornos que pueden reunir algunas, varias o todas las siguientes características:

- Remotos (incluso en otros planetas)
- Inaccesibles (interior de tuberías, conductos de aire)
- Hostiles (volcanes, profundidades marinas)
- Peligrosos (tóxicos, radiactivos, altas presiones, proyecciones de material)

- Impredecibles (entornos no estructurados)
- Predecibles (entornos estructurados)

Teniendo que realizar en ellos tareas:

- Peligrosas (aplicaciones de mantenimiento en centrales nucleares)
- Repetitivas (limpiar o pintar una superficie)
- No repetitivas (mover objetos en un entorno cambiante)
- Cualificadas (cambio de aisladores en redes de alta tensión)
- Sujetas a fatiga (limpieza por granallado de superficies)
- Limitadas en el tiempo (por legislación, en centrales nucleares o por fatiga, granallado)
- Difíciles (operaciones en una central nuclear o en un transbordador espacial)
- Precisas (aquellas que una máquina realiza mejor que un ser humano)

También se estableció la diferencia entre un robot industrial y uno de servicio. Recordamos aquí el propósito para el que está especialmente diseñado un robot de servicio es para realizar tareas muy específicas de:

- Mantenimiento
- Reparación
- Inspección
- Cirugía asistida
- Construcción
- Ocio

La mayoría de tareas que aquí se enumeran suelen requerir un **diseño morfológico especial** del robot, manipulador o dispositivo mecatrónico, para que pueda realizar dicha tarea, presentando así **morfologías muy distintas y orientadas a la aplicación**.

Por lo que se ha dicho, se puede deducir que los robots de servicio son candidatos perfectos a ser teleoperados (excepto algunas aplicaciones, la mayoría en el campo del ocio – robots antropomórficos, de "compañía", como AIBO, guías de museos, etc.).

Aunque se trata de sistemas teleoperados, puede ser necesario realizar ciertos procesos autónomos (que pueden ser secuencias de movimientos almacenadas para realizar tareas repetitivas o comportamientos reactivos autónomos ante imprevistos que requieren una reacción inmediata). Estos comportamientos autónomos deben ser al menos conocidos, o idealmente, supervisados por un operador, que podría contar con la posibilidad de influir en ellos (corregir una trayectoria) o viceversa, que la reacción autónoma se superponga a las órdenes de un operador (como por ejemplo, parar ante un obstáculo).

4.4.1.3. Elementos típicos de un sistema de teleoperación

Una vez definidos los términos que entran en juego en el campo de estudio de la teleoperación, podemos describir los elementos típicos que componen un sistema de teleoperación.

De forma general, un sistema de teleoperación consta de los siguientes elementos (ver **Fig. 4.11**):

1.- **Operador** o teleoperador: ser humano que realiza a distancia el control de la operación. Su acción puede ir desde un control directo hasta un control supervisado, con el que únicamente se ocupa de monitorizar y de indicar objetivos y planes cada cierto tiempo.

3.- **Interfaz hombre-máquina** (HMI): conjunto de dispositivos que permiten la interacción del operador con el sistema de teleoperación. Se considera al manipulador maestro como parte del interfaz, así como a los monitores de vídeo, panel de operador, pantalla de control, o cualquier otro dispositivo que permita al operador mandar información al sistema y recibir información del mismo.

2.- **Dispositivo teleoperado**: podrá ser un manipulador, un robot, un vehículo o dispositivo similar con sus correspondientes accionamientos. Es la máquina que trabaja en la zona remota y que está siendo comandada o controlada por el operador.

4.- **Canales comunicación**: conjunto de dispositivos que modulan, transmiten y adaptan el conjunto de señales que se transmiten entre la zona remota y la local. Estos canales de comunicación deberán estar adaptados a las necesidades de ancho de banda y a los requisitos propios del entorno en el que se trabaja (ruidos electromagnéticos, contaminación, etc.).

5.- **Unidad de control local**: Dispositivos de control y procesamiento de las señales que llegan del operador a través de los canales de comunicación y que deben ser procesadas para realizar las acciones requeridas. A su vez recoge la información de los sensores y la utiliza para realizar sus tareas de control y proporcionar a su vez, una realimentación al operador.

5.- **Sensores**: conjunto de dispositivos que recogen la información, tanto de la zona local como de la zona remota, para ser utilizada por el interfaz y el control.

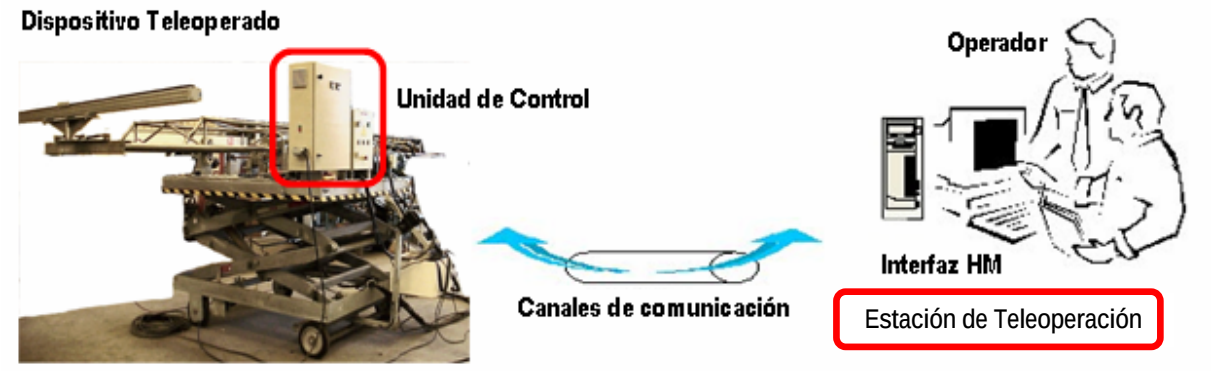


Fig. 4.12.- Elementos básicos de un sistema de teleoperación

4.4.1.4. Características de los sistemas de teleoperación

En la siguiente lista se detallan las características que presentan los robots de servicio teleoperados. Estas caracterísiticas, junto con el propósito de dichos sistemas son una de las entradas del ABD que, junto con los factores de negocio, mercado potencial y objetivos de la organización, sirven para plantear los requisitos funcionales y de calidad de la arquitectura a diseñar (ver Capítulo 5).

- Características relacionadas con las propiedades del entorno y las tareas a realizar

- **Gran dispersión de características de los mecanismos teleoperados**. Como consecuencia de la especialización de estos sistemas, la complejidad y características de los vehículos, brazos y herramientas utilizados varían enormemente de unas aplicaciones a otras. Pueden encontrarse desde dispositivos muy sencillos que podrían operarse con una botonera, hasta mecanismos extraordinariamente complejos y sofisticados.
- **Gran variedad de entornos de funcionamiento**. Los entornos en los que es necesario que trabajen estos sistemas, ya se ha comentado que son muy diversos y pueden llegar a ser muy adversos, lo que marca drásticamente su diseño mecánico, y por tanto, su funcionamiento y necesidades de control. En sus características físicas pueden llegar a influir también cuestiones como el peso, la autonomía energética, etc. No sólo para las partes mecánicas, sino que también se encuentran muchos ambientes agresivos para los sistemas electrónicos, de comunicaciones, e incluso estructurales, desde altas temperaturas a bajas, presencia de ruido electromagnético, altas

presiones, etc. Esto implica que deben ser sistemas robustos, tolerantes a fallos y fácilmente **mantenibles**, que permitan incluso sustituir de forma sencilla subsistemas dañados u obsoletos.

- **Alto grado de especialización.** Cada sistema se diseña para realizar una actividad determinada en un entorno muy concreto, los robots de servicio pueden tener que sustituir al ser humano en la realización de actividades muy complicadas, con un nivel de inteligencia que hace precisa la intervención de un humano como teleoperador. Aunque en ocasiones es posible utilizar vehículos, robots o herramientas comerciales, son más frecuentes los diseños específicos, perfectamente adaptados al entorno de trabajo y especializados en la realización de tareas muy concretas. Esta característica contrasta con la necesidad de realizar varias tareas que se les suele pedir a los robots autónomos modernos.

- Características relacionadas con las configuraciones mecatrónicas

- **Distintas configuraciones de mecanismos.** El mecanismo o mecanismos a teleoperar suelen ser una herramienta o un manipulador, un brazo con una herramienta ensamblada en su extremo o un vehículo provisto de un brazo dotado con la correspondiente herramienta. Otras configuraciones son más raras, pero no imposibles: vehículos sin ningún brazo o herramienta, utilizados exclusivamente para labores de inspección, vehículos que portan más de un brazo, etc.
- **Diversas configuraciones de sensores y actuadores.** Relacionado con las distintas configuraciones de mecanismos, nos encontramos con distintos números de articulaciones a controlar, compuestos de actuadores de diferente naturaleza, neumáticos, eléctricos, hidráulicos. También nos encontramos con un número elevado de sensores de distintos tipos, tanto digitales como analógicos (fines de carrera, sensores de proximidad, de presión, temperatura, etc).

- Características relacionadas con las plataformas de implementación

- **Gran variedad de plataformas**, consecuencia también de la especialización. Podemos encontrarnos diferentes sistemas operativos, de tiempo-real o no, ejecutándose sobre PCs industriales o empotrados, estaciones de trabajo, sistemas hardware basados en microcontroladores o en FPGAs, autómatas programables, etc. Incluso habiendo escogido como plataforma un PC, puede tener distintas posibilidades de bus (VME, VXI, SCXI, ISA, PCI, PC-104).
- **Gran variedad de componentes hardware adicionales específicos del dominio**. Una unidad de control suele contar con numerosos sistemas hardware COTS[4] de distintos fabricantes, con sus propios drivers, principalmente para entrada/salida de información, como tarjetas de adquisición analógicas o digitales, convertidores AD/DA, Timers/Contadores, Modulación PWM, entrada de encoder, tarjetas de control de motores, etc.
- **Incorporación de herramientas software muy diversas** como utilidades de simulación, que en ocasiones se utilizan *on-line* y otras únicamente para el entrenamiento, planificación o test previo de las operaciones, sistemas de navegación, mapas del entorno, visión artificial, etc. Incluso en algunos casos aparecen librerías o componentes COTS que se deben integrar con la aplicación.
- **Gran variedad de lenguajes de programación** para implementar el software. Además de los lenguajes típicos del software como C/C++, Ada, Pascal, etc., en el dominio de las unidades de control para sistemas robóticos nos podemos encontrar con lenguajes orientados al diseño de hardware, como VHDL o bien múltiples lenguajes estructurados para autómatas programables.

- Características relacionadas con el funcionamiento y control

- **Diferentes grados de autonomía y automatismos.** Habitualmente el operador ordena cada una de las acciones que deben realizar los mecanismos, limitándose las acciones automáticas a aquellas que tienen que ver con la detección de fallos y la parada segura. Pero no siempre es así,

[4] *Commercial off-the-shelf*: componentes comerciales ya fabricados.

en ocasiones es posible planificar una determinada secuencia de operaciones, que los dispositivos ejecutan de una vez, limitándose el operador a iniciar y supervisar el proceso. El grado de autonomía puede ser variable según el tipo de sistema.

- **Diferentes patrones de reactividad, algoritmos y políticas de control.** Como consecuencia del punto anterior, muchos de estos sistemas deben ser capaces de reaccionar de forma autónoma ante cambios en el entorno o como consecuencia del desarrollo de las actividades que realizan. En otros casos, el comportamiento viene dictado casi exclusivamente por las órdenes del operador. Incluso existe la posibilidad de que se puedan combinar y elegir en el mismo sistema distintos modos de coordinación y control. En todo caso, según el grado de autonomía requerido, será necesario poder cambiar las estrategias o algoritmos de control.
- **Diferentes modos de operación.** Relacionado con los dos puntos anteriores, estos sistemas pueden proporcionar la posibilidad de adoptar distintos modos de operación en un mismo sistema, según cómo se realice el control, proximidad al sistema teleoperado, nivel de autonomía, etc: local, remoto, colaborativo, autónomo, etc
- **Variedad de parámetros de funcionamiento**. Según el entorno y la aplicación de que se trate, muchos parámetros de funcionamiento, como los referentes al movimiento (velocidad, aceleración, fuerza, flexibilidad, etc) del mismo serán variables, lo cual tendrá influencia en las necesidades de respuesta.

- Factores que afectan al rendimiento del sistema

- **Requisitos de tiempo real,** mucho más estrictos cuanto más bajo es el nivel de los procesos de control. Aunque es más probable que estos requisitos afecten a los procesos del nodo local, pueden aparecer en cualquier lugar dependiendo de las características del sistema. Las necesidades más estrictas pueden ser tareas críticas correspondientes a comportamientos reactivos, y las menos estrictas, o no-críticas, a comportamientos deliberativos, que suelen estar desacoplados entre sí.
- **Carácter distribuido.** En la mayoría de las aplicaciones hay al menos dos nodos, uno correspondiente a la plataforma de teleoperación en el que se ejecuta la interfaz de usuario y los procesos de control de más alto nivel y otro local a los dispositivos, en ocasiones embarcado en los mismos, que suele ocuparse de las tareas de control de más bajo nivel (**Fig. 4.11**). Esta frontera entre nodos de teleoperación y de control es variable según donde se localice el procesamiento. A su vez, el control local de los dispositivos puede ser también distribuido en distintos nodos de computación.
- **Presencia de enlaces de comunicaciones**. Como consecuencia del punto anterior, casi siempre existe un enlace de comunicaciones, adaptado a las necesidades de ancho de banda y a los requerimientos propios del entorno en el que se trabaja (ruidos electromagnéticos, contaminación, etc.) y que por tanto, varía mucho de unas aplicaciones a otras (Ethernet, CAN-bus, diversos buses de campo, comunicaciones inalámbricas, etc.) y que tienen protocolos de comunicación distintos.
- **Las necesidades de transmisión de datos** normalmente no deben ser muy elevadas (la cantidad de información a transmitir normalmente no será grande, excepto de casos especiales de realimentación sensorial y líneas de video, que normalmente serán líneas dedicadas), pero en todo caso deben ser medios deterministas.

- Características relacionadas con la seguridad

- **Requisitos de seguridad por lo general muy estrictos**, pues suelen estar en juego la integridad de personas e instalaciones.
- **Suelen estar sujetos a normativas de seguridad**, que incluyen la previsión de contingencias, alarmas y la creación de protocolos de reacción ante fallos, sistemas de parada segura, etc.

- **Niveles de acceso restringidos**. No todos los operarios que manejen el sistema están igual de cualificados o tienen los mismos permisos de seguridad para cambiar la configuración del sistema o realizar ciertas operaciones.

- Características relacionadas con la vida y uso de los sistemas

- **Vida operativa muy variable de todo el sistema o de alguna de sus partes.** Algunos sistemas están pensados para trabajar durante años, en otros casos se definen para operaciones muy puntuales impuestas por algún tipo de contingencia. La variabilidad de su vida operativa puede no ser total, sino que sea necesario sustituir sólo algunos de sus subsistemas por quedarse obsoletos o por estar dañados.
- **Realización de tests y pruebas de diagnóstico.** Sobre todo en las fases de puesta en marcha, mantenimiento o reparación de fallos, estos sistemas suelen incorporar mecanismos de test, ensayos y pruebas de diagnóstico de su funcionamiento.
- **Utilización de subsistemas externos**. Además de las propias interfaces del operador, que pueden ser de muy diversos tipos, los sistemas de teleoperación suelen incluir otros subsistemas que sirven de ayuda, como sistemas de visión, realimentación de fuerza, sistemas de navegación auxiliares, etc.
- **Colaboración con otros sistemas**. A veces, los robots de servicio colaboran entre si para alcanzar objetivos comunes a través de canales de comunicación y estrategias de coordinación diversos.
- **Gran variedad de interfaces de usuario**, adaptadas a la características de los mecanismos, a la naturaleza del trabajo a realizar y a las preferencias de los operadores. La mayoría de los sistemas incluyen interfaces gráficas, que en ocasiones se complementan con *joysticks* más o menos sofisticados. La realimentación de pares y fuerzas hacia el operador empieza a tener relevancia en algunas actividades muy concretas. En cualquier caso la interfaz debe ofrecer al operador una imagen clara y consistente del estado del entorno y de los dispositivos, para que éste pueda supervisar las operaciones.
- **Mecanismos de puesta en marcha, arranque, parada, sustitución de herramienta, etc.** Estos sistemas llevan implícitos mecanismos de arranque, parada, rearme, rearranque tras una parada de emergencia, sustitución de herramienta, cambio de operación, etc., que pueden ser complejos y requerir el cumplimiento de un protocolo de arranque, de puesta en marcha, etc.

4.4.1.5. Aplicaciones y futuros desarrollos

Desde los primeros desarrollos de la teleoperación, la industria nuclear ha sido el principal consumidor de sistemas de teleoperación [Larcombe94]. Sin embargo, con el paso de los años se fue viendo su aplicabilidad a otros sectores, especialmente relacionados con las industrias de servicio. En la referencia clásica [Sherindan92] o en las más actuales [Schilling00] y [EURON04] se puede encontrar una descripción muy pormenorizada de la mayoría de estas aplicaciones, de las que aquí se presentará una breve enumeración.

- **Aplicaciones nucleares:** La utilidad del sistema de teleoperación radica en poder tratar y manipular sustancias radiactivas, así como moverse por entornos contaminados, sin peligro para el ser humano. Entre sus principales aplicaciones están: manipulación y experimentos con sustancias radioactivas, operación y mantenimiento de instalaciones (reactores, tuberías, instalaciones de elaboración de combustible nuclear, etc.), desmantelamiento y descontaminación de instalaciones, y finalmente actuación en desastres nucleares. De hecho, ya se comentó en los antecedentes de esta Tesis que el grupo de investigación en el que se enmarca este trabajo, el DSIE, tiene una dilatada experiencia en el campo de las aplicaciones de la teleoperación en centrales nucleares [Iborra03]
- **Aplicaciones submarinas:** son también numerosas, aunque en este caso la mayoría de los manipuladores van sobre un vehículo submarino, denominado R.O.V. (Remote Operated Vehicle),

que también va teleoperado. La utilidad de esto sistemas radica en poder acceder a ciertas zonas y profundidades donde le es imposible o peligroso a un submarinista. Entre sus principales aplicaciones están: inspección, mantenimiento y construcción de instalaciones submarinas, minería submarina, e inspección de suelo marino [Liddle00]. No hace mucho tuvimos ocasión de comprobar la utilidad de este tipo de vehículos en las operaciones de inspección del *Prestige* tras el desastre ecológico ocasionado por su naufragio en aguas de Galicia.

- **Aplicaciones espaciales:** son el tercer grupo de aplicaciones más importante de los sistemas de teleoperación. Tienen el reto añadido de tener que trabajar con retardos temporales en las comunicaciones, lo que las hace especialmente problemáticas. Entre sus principales aplicaciones están: experimentación y exploración planetaria (normalmente con vehículos tipo *rover*), mantenimiento y operación de satélites, construcción y mantenimiento de estaciones espaciales. En este capítulo se muestran varias arquitecturas pensadas para este dominio [Albus87], [Nesnas03], [Fong02].

- **Aplicaciones médicas:** recientemente se ha fortalecido de forma importante la aplicación de las tecnologías de la teleoperación al sector médico. Desde los primeros desarrollos de prótesis o dispositivos de asistencia a discapacitados hasta la más novedosa de la telecirugía, o el telediagnóstico, aunque éste no pertenezca estrictamente al sector de la teleoperación [Darío96]. También es interesante mencionar en el campo de la medicina, la ayuda que pueden prestar los robots en la rehabilitación de enfermos y la ayuda a personas discapacitadas [Hans02].

- **Otras aplicaciones:** la teleoperación también ha entrado con fuerza en otros sectores a los que en principio no estaba enfocada. Entre éstos se pueden citar los siguientes:

  Aplicaciones de construcción [GonzálezDS00] y minería [Fink97], mantenimiento de líneas en tensión [Aracil02], mantenimiento e inspección de instalaciones [Abderrahim99], [Lorenc97], intervención en desastres naturales, incendios, etc., aplicaciones militares y entretenimiento, donde cabe destacar los robots guías de museos, y robots para jugar como AIBO (ver ejemplos actuales de todas estas aplicaciones en [EURON04]). Robots de limpieza dentro y fuera de las casas, limpieza de buques en astilleros [Ortiz00], robots para agricultura y bosques, para eliminar minas [García03], robots de de vigilancia e inspección.

En cuanto a los futuros desarrollos, el rango de aplicaciones irá en aumento a medida que las tecnologías avancen y se vaya investigando en la mejora de cada uno de los elementos que componen un sistema de teleoperación. El futuro pasa por una madurez en las tecnologías que en los últimos años se están incorporando a la teleoperación. Tecnologías propias de la robótica y de la realidad virtual.

Por último, otro gran avance es el que se está dando con respecto a las arquitecturas globales de los sistemas [Albus87], [Coste00], [Granot01], [Nesnas03]. La tendencia apunta a intentar alejar al operador del bucle de control, dando al sistema remoto el mayor nivel de autonomía posible. Esto es especialmente útil en el caso de las aplicaciones espaciales, en las que el retraso en las comunicaciones limita de forma práctica la capacidad de control del operador. Se tiende también a utilizar simuladores que se actualizan con la información proveniente de los sensores del entorno, de manera que el operador pueda simularla antes paso a paso para después volcarla sobre el manipulador remoto, y que éste la ejecute de forma automática [Álvarez01a].

Se percibe también una tendencia a intentar globalizar y hacer más asequible y simple la teleoperación a nivel terrestre pero entre muy grandes distancias. Es notoria la abundante investigación que existe en la actualidad sobre la teleoperación de robots utilizando el canal de comunicación universal Internet [Fiorini97], [Grange00], [Benali01].

4.5. Arquitecturas de control para teleoperación

Una vez estudiado el estado de la técnica y las tendencias en el dominio de los sistemas de control para robots en general, acotaremos más el dominio de estudio de la tesis si nos centramos en el control de sistemas teleoperados.

4.5.1. Modos de control teleoperado

El método más común para la teleoperación de manipuladores o de vehículos es el **control directo o manual**: el operador maneja el manipulador o el vehículo usando algún tipo de joystick o interfaz mientras que visualiza la operación a través de monitores de video (**Fig. 4.13-izq**). Puesto que todas las decisiones de control dependen del operador, las prestaciones del sistema están directamente ligadas a las capacidades del operador humano. Por tanto, numerosos factores como la habilidad, entrenamiento, etc, juegan un papel importante en la manera en que el sistema funciona. Otros factores, como el ancho de banda de la comunicación pueden influir en la eficacia de la operación. Un ejemplo de este tipo de control sería [Aracil02] ROBTET, *"Robot teleoperado para líneas eléctricas de alta tensión"*, y la mayoría de aplicaciones de tele-cirugía.

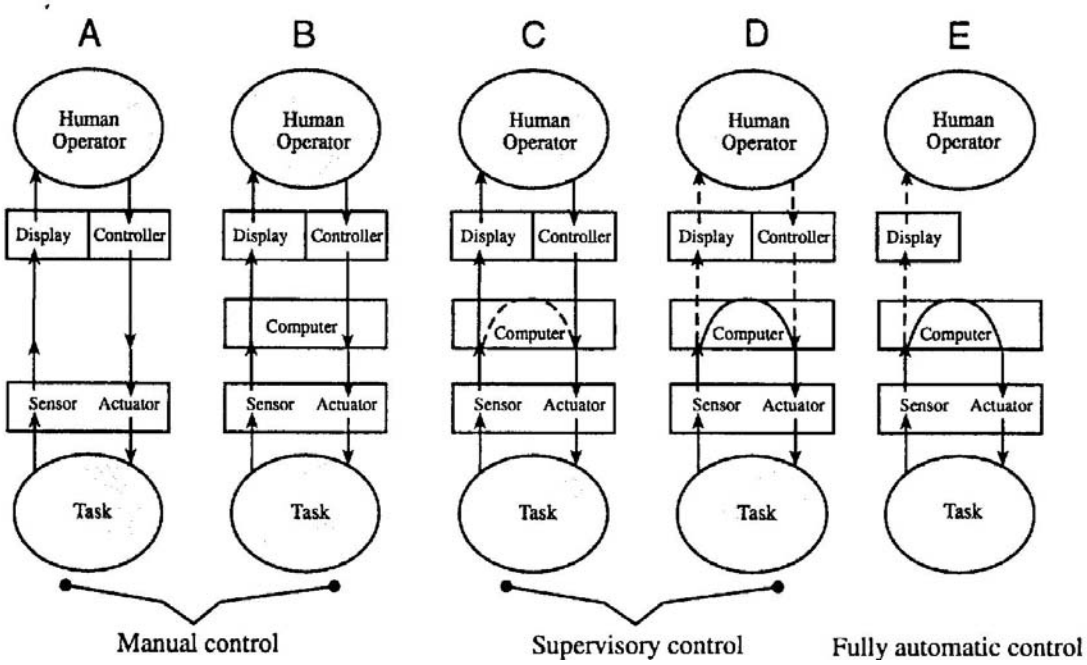


Fig. 4.13.- Distintos modos de control para un sistema teleoperado [Granot01]

Con el **control supervisado** (**Fig. 4.13-centro**), el operador humano divide el problema en una secuencia de tareas que el robot realiza por sí mismo [Sheridan92]. Una vez que cede el control al robot, el humano típicamente asume el rol de monitorizar. Sin embargo, también puede controlar al robot de manera intermitente, cerrando algunos lazos de comando, o puede controlar algunas variables mientras que deja otras al robot. En [Granot01] estos modos distintos de control supervisado se matizan con los nombres de *negociado* y *compartido*. Pone como ejemplo de control compartido el caso de NASREM/RCS, donde se diseña un interfaz de operador para cada nodo inteligente, es decir, el operador tiene acceso directo a cada tarea y puede controlarla remotamente cuando lo crea necesario.

Un **control totalmente autónomo** (**Fig. 4.13-der**), en realidad, raramente es totalmente autónomo, ya que el humano da los objetivos de alto nivel, que el robot realiza de manera independiente. La diferencia entre el control supervisado y el totalmente autónomo es la naturaleza del objetivo. En el supervisado, los objetivos están limitados y la planificación de tareas es llevada a cabo primariamente por el humano. Con el totalmente autónomo, los objetivos son más abstractos y el robot es responsable de la planificación.

Con los modelos convencionales, se alcanzarán prestaciones pobres (o fallos) si el humano no reconoce que el robot no está preparado para realizar una tarea o solventar una situación. Además, ninguno de los modelos se puede acomodar efectivamente a un amplio rango de usuarios. El control

directo, por ejemplo, está limitado a usuarios entrenados y expertos porque la dificultad y el riesgo son altos.

Como se puede observar en los esquemas de la **Fig. 4.13**, lo que caracteriza los sistemas teleoperados y los diferencia claramente del resto de sistemas robóticos es la presencia del operador humano, el cual, como se ha visto, puede cobrar más o menos importancia en el control del robot.

Aunque parece fácilmente demostrable el hecho de que la inteligencia del sistema teleoperado la proporcione el operador humano, la experiencia nos dice que hay una serie de limitaciones y errores posibles en el teleoperador que hay que tener en cuenta y que aparecen por el hecho de introducir a un ser humano en el lazo de control [Fong01], entre ellos, se pueden destacar:

- ✓ Prestaciones pobres (control impreciso, fallo al detectar obstáculos)
- ✓ Errores cognitivos (imagen mental del entorno)
- ✓ Errores de percepción (fallo de juicio, distancia, etc.)
- ✓ Pérdida de la percepción de la situación del sistema teleoperado (desorientación, pérdida de contexto)
- ✓ Otros (tiempos de reacción, fatiga, etc)

También hay que tener en cuenta los fallos o limitaciones por parte de las comunicaciones (ancho de banda, confiabilidad, determinismo) y del propio sistema (falta de robustez, poca flexibilidad, fallos no previstos, etc.)

El control **colaborativo** propuesto en [Fong02] pretende superar algunas de las limitaciones expuestas que tienen los sistemas convencionales, a través de la colaboración y el diálogo. En los modelos de control supervisado o totalmente autónomo, si el robot tiene dificultades, la única opción que tiene es continuar realizando la tarea pobremente o parar. Con el control colaborativo, el robot tiene la opción de pedir asistencia al humano, proporcionar información, ayuda para realizar una percepción o reconocimiento, etc. Dado que el trabajo a realizar es compartido y asignado a través del diálogo operador-robot, el humano es incluido automáticamente en un lazo de control cuando sea necesario. Esto difiere de manera significativa de otros modos de control, que requieren que el usuario decida cómo, cuando y donde debería ser asignado el control. En el control colaborativo, el usuario se ve incluido en múltiples lazos de control, no sólo en tareas de ejecución, sino también en el lazo de percepción, reconocimiento, etc.

Como limitaciones de este planteamiento estarían las propias de un control adaptativo. Además es difícil validar y verificar las situaciones en las que el robot necesita ayuda. Para colaborar se necesita que cada uno sepa lo que el otro puede hacer.

Implementación de los sistemas de control

En cuanto a la fabricación e implementación del control, las soluciones más tradicionales al problema de teleoperación eran puramente electromecánicas, es decir, sistemas de teleoperación y control basados en el empleo de circuitos eléctricos y neumáticos [Granot01]. Las soluciones más avanzadas incluyen el empleo de electrónica, hardware y software. Una primera solución es la utilización de autómatas programables. Indudablemente el grado de automatización y asistencia al operador es muy limitado y en ocasiones insuficiente. Sin embargo, son soluciones muy robustas y seguras, motivo por el cual su grado de implantación en la industria es muy elevado.

Los sistemas informáticos de teleoperación basados en el empleo de estaciones de trabajo, PC´s industriales con Sistemas Operativos de tiempo-real o hardware reconfigurable son soluciones mucho más flexibles y que pueden facilitar mayores prestaciones al operador, en cuanto a la calidad de los interfaces (dispositivos de visualización y envío de órdenes). Sin embargo, queda todavía mucho trabajo por hacer a la hora de desarrollar aplicaciones robustas y de fácil mantenimiento, que a la vez garanticen los requisitos de tiempo real.

4.5.2. Estado de la técnica

Quizás la primera arquitectura o modelo de referencia rigurosa planteada para teleoperación es **RCS** (*Real-Time Control Systems, a Reference Model Architecture for Teleoperation and Telerobotic applications*), fue desarrollada en el NIST (National Institute Systems Technology) como un estándar de referencia en 1990, puede revisarse un análisis de la misma en [Álvarez97-td]. Durante años, RCS se ha adaptado a diferentes entornos, incluyendo la fabricación flexible, aplicaciones para el espacio y operaciones subacuáticas. La filosofía RCS está basada en la teoría de máquinas inteligentes de Albus [Albus91]. Esta arquitectura fue adoptada por la NASA en su versión más conocida (**NASREM** [Albus87]) y ya comentada en el punto 4.3.1.1 de este mismo capítulo como una arquitectura general teleautónoma con pretensiones de ser una arquitectura utilizable en multitud de dominios de aplicación, aunque enfocada para sistemas de control de telerobots especiales. Además de lo ya comentado, un problema importante que tiene NASREM, es que resulta complejo intentar acomodar cada posible escenario en una estructura jerárquica muy rígida, y esto responde parcialmente al hecho de que no es fácil trasladar la arquitectura teórica a una implementación práctica [Stewart97].

Arquitectura mixta basada en RCS [Granot01]

En [Granot01] se propone una arquitectura mixta entre RCS del NIST y los sistemas basados en comportamiento. Esta arquitectura permitiría al operador humano iniciar la acción a los niveles más altos, en entornos que permitan la automatización completa, y además ser capaz de interferir a todos los niveles en cualquier momento, limitado sólo por sus habilidades. También puede cambiar los comportamientos del sistema y prácticamente interferir en las acciones llevadas a cabo en tiempo-real.

Para entornos no estructurados, la descomposición de tareas no es muy útil, porque el modelado del entorno y las acciones de planificación basadas en este modelo son muy limitadas en el tiempo y consumen muchos recursos. La filosofía de RCS basada en asumir planes alternativos, como respuesta a eventos inesperados, está limitada por la frecuencia de los cambios en el entorno, teniendo en cuenta que los recursos del sistema son limitados. Con el objeto de permitir operaciones autónomas del telerobot bajo un control estrictamente supervisado en condiciones de ambiente altamente cambiante, la arquitectura de [Granot01] remplaza la descomposición funcional de tareas a bajos niveles de la jerarquía de control. Con ello se reducen la necesidad de planear ciertas acciones previamente a la reacción a eventos inesperados.

La idea de teleoperar un robot, de forma que dicho robot tenga un cierto comportamiento autónomo que pueda ser supervisado por un operador humano a la vez que suministra las órdenes para cumplir objetivos de alto nivel, es una tendencia recurrente también en otros ejemplos de la bibliografía especializada:

Una arquitectura para YAIR [Posadas02]

Por ejemplo, en [Pérez02] y [Posadas02] se propone una arquitectura para robots móviles teleoperados. Es una arquitectura híbrida de tres niveles: reactivo, deliberativo y de interfaz. Esta última capa tiene entidad propia porque plantea la utilización de un patrón *blackboard* y como medio de comunicación CAN-bus [Bosch91], y el sistema de comunicación SC [Posadas97], un sistema de comunicación apropiado para el control remoto y el acceso a sistemas de tiempo-real con arquitectura distribuida, de forma que haya un intercambio de información transparente entre los distintos nodos de procesamiento.

La arquitectura de control del robot es híbrida: la parte deliberativa la proporciona la planificación remota de un teleoperador, que sugiere objetivos globales, consistiendo la parte reactiva en un conjunto de comportamientos autónomos de seguridad (evitar obstáculos). También contempla la posibilidad de que el robot esté realizando una tarea autónoma (avanzar en línea recta) y que el operador pueda modificar dicha trayectoria si lo estima conveniente.

La composición de las órdenes recibidas desde la estación de teleoperación (p.ejm., la dirección de movimiento deseada), con las órdenes de comportamiento reactivo (p.ejm., evitar obstáculos) produce

los comandos apropiados que son mandados al controlador de movimiento. El resultado es un comportamiento *amigable* por parte del robot teleoperado, ya que el operador no tiene que preocuparse de evitar los obstáculos, simplemente le marca un objetivo global que debe alcanzar.

CLARAty [Nesnas03]

La misma filosofía se imprime en la arquitecutra **CLARAty**, del *Jet Propulsion Laboratory*, comentada en el punto 4.3.3 y que también es adoptada por la NASA en la actualidad para sus *rover*s de exploración planetaria. Dado que la enorme distancia con Marte hace imposible que las órdenes lleguen en un tiempo inferior a 15 minutos y la realimentación de estado sufre el mismo retardo, se hace necesario un comportamiento autónomo que permita reaccionar ante eventos imprevistos, que puedan suceder mientras ejecuta el plan global proporcionado por el teleoperador.

Safeguarded Teleoperation Controller [Fong01]

En la arquitectura de control para robots teleoperados, *Safeguarded Teleoperation Controller*, propuesta por el Instituto de Robótica de la *Carnegie Mellon University* [Fong01], se también una línea parecida para vehículos de exploración de otros planetas, haciendo especial énfasis en la necesidad de que un operador humano tenga la posibilidad de rectificar las decisiones que tome el robot de forma autónoma cuando se mueve en entornos no estructurados, a este modo de control lo llaman control colaborativo. También estudian especialmente distintos interfaces de teleoperación que faciliten la labor incluso a un operador no especializado.

La arquitectura del controlador consiste en una serie de módulos distribuidos, conectados por comunicaciones entre procesos (IPC) que usan un *framework* "publicar-subscribir". Algunos de los módulos se ejecutan de forma aislada y operan asíncronamente. Otros módulos, particularmente esos que procesan los datos de sensores u operan con el hardware del robot, presentan requisitos de tiempo y operan según el sistema Saphira [Konolige97], comentado en el punto 4.3.1.2

Otras arquitecturas

También se han propuesto otras arquitecturas de teleoperación, normalmente para sistemas específicos, para realizar unas tareas concretas. Muchas de ellas, al igual que NASREM, están relacionadas con la teleoperación de vehículos u otro tipo de sistemas destinados a operar en el espacio. Las más destacables son las siguientes:

SMART [Anderson95], [Graves99] (*Sequential Modular Architecture for Robotics and Teleoperation*) es otro ejemplo de arquitectura que pretende ser genérica para sistemas telerobóticos, que ha sido diseñada con componentes separados alrededor de un protocolo de comunicación. Esta arquitectura desarrollada en Sandia National Labs (USA) es un buen ejemplo del uso de la orientación a objetos para construir sistemas telerobóticos. El diseño usa una red de comportamientos a bajo nivel que se activan y desactivan por medio de una capa simbólica de alto nivel. La aplicación de SMART al problema de teleoperación en tanques de almacenamiento subterráneos ilustra el hecho de que las arquitecturas generalizadas son capaces de realizar tareas en el mundo real. Una debilidad de SMART es que sus herramientas de construcción off-line realmente no soporta cambios de modo o reconfiguración dinámica.

MOTES [Backes90], [Backes93] (*Modular Telerobot Task Execution System*)

Entre los trabajos anteriores de telerobótica anteriores a CLARAty llevados a cabo por el Jet Propulsion Lab se encuentra una arquitectura para secuenciar comandos para un sistema telerobótico. MOTES permite control supervisado, teleoperado y compartido.

En MOTES se utilizan varias colas FIFO de comandos. Define una cola de comandos enviados desde el control, y una cola de comandos a ser ejecutados en unas condiciones específicas. Uno de los problemas con MOTES es la naturaleza secuencial de las colas de tareas.

ROTEX [Hirzinger93]

Este es otro ejemplo de arquitectura para control compartido humano-autónomo. Se desarrolló para el primer robot espacial teleoperado de la historia, en un experimento llevado a cabo por una misión del *"Spacelab-Mission D2"* dentro del COLUMBIA en abril de 1993.

ROTEX usa una arquitectura jerárquica con tres niveles: un nivel de planificación de tarea que genera una lista de subtareas ordenadas en una estructura gráfica, un nivel de coordinación de tarea que secuencia las tareas, y un nivel de ejecución de tarea que lleva a cabo las acciones de control. La planificación de tareas la lleva a cabo a-priori de forma manual el diseñador del sistema. Ha demostrado su capacidad en el uso de aplicaciones en misiones críticas y al abordar nuevas tareas.

En **[Lin95]** se presenta un sistema para control de robots móviles con realimentación de múltiples sensores. El sistema soporta cuatro modos de control teleoperado (directo, negociado, compartido y supervisado) y permite a los operadores asistir interactivamento en el modelado del entorno.

Todas estas arquitecturas tienen los problemas que se han expuesto al comentar cada una. Hay que destacar también que la mayoría de ellas no están especialmente pensadas para ser reutilizadas en diversos robots teleoperados (excepto NASREM, que por otra parte adolece de una rigidez que dificulta la reutilización).

4.5.2.1. Teleoperación a través de Internet

Las posibilidades de comunicación entre la estación de teleoperación y la unidad de control del robot son todas las existentes en la actualidad, desde las comunicaciones por radio, *wireless*, incluyendo buses de campo *wireless* [Decotignie02], buses de campo estándar [Thomesse02], [Bosch91], hasta las importantes conexiones LAN y en general la universalmente utilizada *World Wide Web*. En [Bates98] se hace una exposición detallada de estas distintas posibilidades de comunicación. Sobre el campo de las comunicaciones en Internet hay multitud de estudios donde se estudian las posibilidades de teleoperación en Internet, teniendo en cuenta que el hecho de ser una red no-determinista marca las posibilidades de distribución y comunicación de tiempo-real. Entre los más destacados, podríamos nombrar a [Grange00], con título "*Effective Vehicle Teleoperation on the World Wide Web"*, donde se exponen las nuevas técnicas y características especiales, para la minimización del uso del ancho de banda, la optimización de la interacción hombre-computador, integración de sensores, etc., se defiende que hay que explotar las comunicaciones *www* para hacerlas accesibles a la teleoperación efectiva en Internet, asimismo se presenta un estado de la técnica de sistemas de teleoperación desarrollados para operar en Internet y propone una arquitectura *WebDriver*. Otro interesante estudio se presenta en [Salzmann02], donde se exponen los retos que se deben satisfacer para permitir una interacción de tiempo-real efectiva en Internet para soluciones de control remoto, no sólo para telerobótica, sino también para otras aplicaciones de control.

4.6. Conclusiones del estudio del dominio

Se podrían extraer varias conclusiones del estudio de dominio realizado en este capítulo, así como de las propuestas del estado de la técnica, que servirán para plantear el desarrollo de una arquitectura de referencia para el control de robots de servicio teleoperados:

Se ha destacado la importancia de contar con arquitecturas de referencia que favorezcan la reutilización, escalabilidad y modificabilidad de los diseños de sistemas complejos. Parece claro que la mejor forma de abordar la complejidad de sistemas como los controladores de robots es dar una modularidad a la estructura de los mismos. La utilización de conceptos como componentes, puertos y conectores propuestas en el desarrollo de software basado en componentes parece ser una de las soluciones más adecuadas para dotar de modularidad a tales sistemas. El hecho de que estas ideas surjan de la disciplina de Ingeniería del Software no implica que los componentes sean siempre implementados en software, sino que también pueden ser componentes hardware, COTS, módulos de autómatas, código VHDL traducido al hardware de una FPGA, etc.

Sin embargo, la tendencia hacia la modularidad, reutilización, sistemas abiertos, etc. se ve obstaculizada por la falta de estándares, lo cual hace difícil el desarrollo más rápido de la robótica, y

ocasiona que muchos de los robots desarrollados en laboratorio se queden allí [Mallet02b]. Las arquitecturas que hagan progresar a la robótica serán una verdadera solución de futuro si se llega a estandarizar los componentes propuestos en dichas arquitecturas, de forma que se pueda llegar a fabricar robots y sistemas mecatrónicos de una manera realmente modular y reconfigurable.

La utilización de componentes no proporciona por sí misma la estructura de control a un sistema, en este sentido hay dos vertientes posibles: el planteamiento de *frameworks* de componentes que puedan construir arquitecturas distintas, o bien el planteamiento de arquitecturas basadas en componentes con reglas definidas y donde las interacciones entre componentes puedan ser variables, de forma que se puedan instanciar en distintos sistemas de un dominio

La inmensa mayoría de los sistemas robóticos descritos en el estado del arte son sistemas concretos, siendo muy difícil encontrar arquitecturas de referencia para un dominio. Los *frameworks* tienden a ser una solución, pero dejan demasiado abierta y a merced del desarrollador la posibilidad de combinar componentes para construir sistemas. También hay una carencia general de especificación rigurosa de requisitos y de descripción formal o semi-formal de las arquitecturas con lenguajes como UML. Esta tesis intenta aportar rigurosidad en el diseño de arquitecturas para robots, de hecho, los estudios iniciados en la misma y en [Pastor02-td] sirven como punto de partida para el proyecto DYNAMICA [DYNAMICA03].

En cuanto a las distintas opciones de control y arquitecturas extraídas del estudio del estado de la técnica, la inmensa mayoría de las arquitecturas, incluso en el caso de utilizar *frameworks* como OROCOS, proponen una descomposición del sistema robótico en capas decisión-ejecutiva-funcional en un planteamiento de estilo de control híbrido (reactivo-deliberativo).

Quizá una de las arquitecturas que mejor refleja las últimas tendencias en el desarrollo de robots semi-autónomos sea CLARAty. Esta arquitectura propone una fusión a nivel decisión-ejecutiva, pero que básicamente es una especie de estructura híbrida que combina los niveles reactivo y deliberativo. También destaca como elemento diferenciador la capacidad de llevar la inteligencia a niveles bajos del control, al contrario de la mayoría de arquitecturas, que plantean la inteligencia en el mismo eje que la granuralidad. Tanto en CLARAty como en OROCOS, se utilizan técnicas de orientación a objetos que permiten caracterizar los típicos elementos del dominio de aplicación, de forma que a partir de clases genéricas se pueden especializar los objetos necesarios en una implementación mediante mecanismos de herencia y composición. Por ejemplo, es interesante la presencia de clases y objetos de abstracción del hardware con una interfaz de referencia que se propone en estos trabajos, favoreciendo enormemente la reutilización y la modernización del hardware utilizado.

Una aportación importante de la presente tesis frente a los mencionados trabajos consiste no sólo en la propuesta de una arquitectura de referencia, sino también en la descripción de un modelo de componentes conceptuales que pueden utilizarse para definir arquitecturas concretas para los sistemas del dominio. También en ACROSET se tendrá muy en cuenta la posibilidad de interacción entre componentes, independientemente de que los mismos sean software o hardware (de hecho, en el Capítulo 7 se proponen distintas implementaciones combinando elementos hardware y software).

Está claro que, cuantas más generalidades se quieran conseguir, cuanto más aspectos se quiera que cubra la arquitectura, más difícil será que los cumpla todos. En el momento de instanciar ACROSET en un sistema concreto habrá que llegar a un compromiso entre los requisitos del sistema. Por ejemplo, si se quiere que la instanciación de la arquitectura sea altamente reutilizable, será difícil que tenga las mejores prestaciones temporales.

A nivel de implementación, y según las disponibilidades tecnológicas actuales, las últimas tendencias se dirigen hacia sistemas de control de tiempo-real distribuidos, tolerantes a fallos, consistentes en una serie de nodos de procesamiento que pueden ser sistemas empotrados con sus dispositivos periféricos y sensores inteligentes, y finalmente un sistema de comunicación, como por ejemplo, un bus de tiempo-real [Kopetz02]

En cuanto a teleoperación, la tendencia observada al estudiar el estado de la téncia es que una arquitectura para sistemas teleoperados debe dejar abierta la posibilidad de aumentar la autonomía del

sistema siempre que la tecnología lo permita (tendencia a la supervisión). Habrá que tener en cuenta la contribución que puedan hacer sistemas externos a dicha inteligencia, no sólo el operador, por consiguiente, la interoperabilidad de los sistemas diseñados debe estar siempre presente.

También se tiende a abandonar la tendencia de dividir estrictamente los procesos que se llevan a cabo en la estación de teleoperación frente a aquellos que se realizan en la unidad de control; la capacidad de distribución de los sistemas y la posibilidad de comunicaciones cada vez más rápidas y fiables hacen posible que la línea divisoria entre unidad de control y la estación de teleoperación sea variable. Una arquitectura de referencia como ACROSET debe tener en cuenta dicha variabilidad.

Se puede concluir por tanto, que la tendencia en teleoperación es la telerobótica: un humano supervisa sistemas controlados de manera autónoma, capaces de tener cierto grado de percepción y acción inteligente. El operador humano debería concentrar su atención en tareas que requieren grandes capacidades (planificación de trabajos, interpretación del entorno, etc) y dejar al sistema de control las tareas que también son complejas, pero que se pueden realizar de manera autónoma (evitación de obstáculos, realización de secuencias programadas, etc) [Granot01].

Capítulo 5

Análisis de requisitos y estrategias para obtener ACROSET

5.1. Introducción

En el enfoque metodológico de la tesis expuesto en el Capítulo 4 se propuso seguir una metodología de desarrollo centrada en la arquitectura como la que propone el Método de Diseño Basado en Arquitectura (ABD), no para proponer la arquitectura de un sistema concreto, sino para llegar a una arquitectura de referencia para un dominio determinado. La arquitectura ACROSET no pretende cubrir todos los sistemas de teleoperación posibles, sino un subconjunto de los mismos: las unidades de control de robots de servicio teleoperados. No es realista plantear el diseño de arquitecturas de referencia para dominios excesivamente amplios, en los que los diferentes sistemas demandan distintos, y a veces, incompatibles compromisos de diseño.

Para realizar los pasos del ABD, que se pueden consultar en [Bachmann00a] y en el Anexo I de esta tesis, es necesario proporcionar una serie de entradas al método, que se muestran resumidas en la **Fig. 5.1.**

El primer y más obvio paso a la hora de definir la *arquitectura de un sistema o de una familia de sistemas* es determinar aquellos requisitos que tienen una importancia determinante en el éxito o fracaso de la misma. El análisis de requisitos que se realizará en este capítulo servirá para determinar las entradas del ABD para el diseño de la arquitectura (óvalo de la izquierda de la **Fig. 5.1**), es decir:

- Directrices de la arquitectura
- Requisitos funcionales abstractos
- Requisitos de calidad abstractos
- Opciones arquitectónicas
- Restricciones

Para sugerir las directrices de la arquitectura y los requisitos funcionales y de calidad abstractos, es necesario contar con la información del dominio (características y propósito de los sistemas que

pertenecen al dominio) a partir del estudio realizado en el capítulo anterior. Los factores de negocio (mercado potencial, y objetivos de la organización) se describirán en este capítulo, concretando de esta forma las necesidades de la organización que desarrolla la arquitectura para el dominio definido anteriormente.

En el siguiente capítulo se utilizará esta información para seguir los pasos de ABD y plantear la arquitectura, con sus componentes básicos, subsistemas principales y relaciones. En el Capítulo 7, se concretarán los requisitos abstractos en casos de uso y escenarios de calidad para un sistema determinado, y se utilizará la arquitectura de referencia propuesta para dotar de una arquitectura concreta a los sistemas desarrollados.

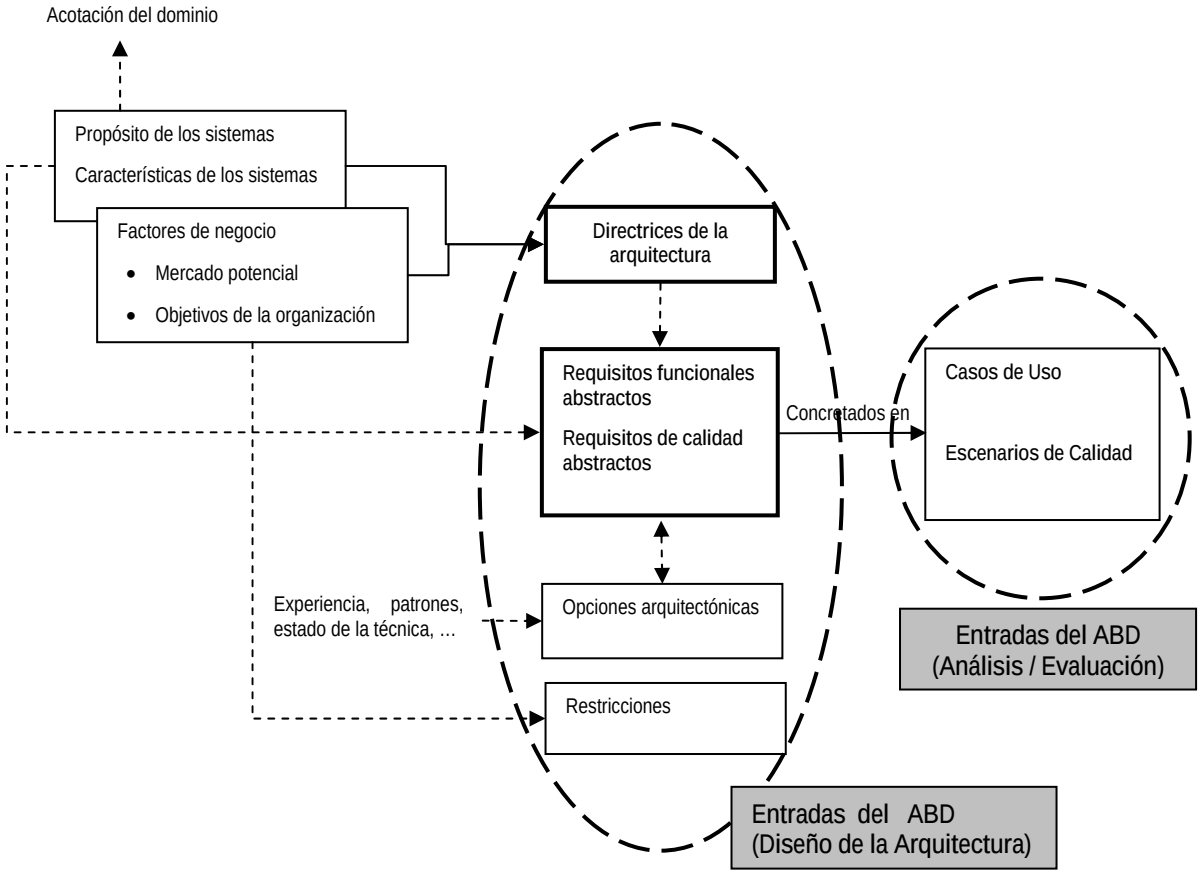


Fig. 5.1.- Obtención de las entradas del método ABD

Puesto que el arquitecto revisa los requisitos y según estos, procede con el diseño arquitectónico, puede necesitar volver a los *stakeholders* para negociar los cambios en los requisitos que hicieran falta y si fuera necesario, llegar a un compromiso en la adopción de requisitos que son contrarios entre sí.

El propósito de los sistemas y sus características sirven para acotar el dominio de sistemas considerado. El dominio de la teleoperación es muy extenso, existiendo sistemas de muy diverso tipo. Como se ha dicho en los capítulos anteriores.

Los factores de negocio cubren una amplia gama de aspectos, que pueden clasificarse según diferentes criterios. Cualquier clasificación es buena siempre que muestre claramente [Kazman00]:

- La necesidad, el propósito y las características de los sistemas.
- Las características del mercado potencial y el tipo de producto que demanda.

- Los objetivos de negocio que se pretenden.
- Las posibles restricciones técnicas y de negocio.

A partir de los factores de negocio, de las características y propósito de los sistemas y de las directrices que inicialmente se puedan derivar de los mismos, es posible caracterizar los atributos de calidad de la arquitectura. Puesto que dichos requisitos se definen para una familia de sistemas, su grado de concreción es menor cuanto mayor sea el rango de sistemas considerados. Esta **variabilidad** de los requisitos está contemplada en el ABD a través de los **requisitos funcionales abstractos** y de los **requisitos de calidad abstractos**.

Este capítulo se organiza siguiendo las pautas descritas en los párrafos anteriores, y que se pueden observar esquematizadas en la **Fig. 5.1.** En primer lugar se describen los usos y usuarios más relevantes de la arquitectura y los propósitos de la organización desarrolladora. A continuación se presentan las directrices observadas en primera instancia, basados en las funcionalidades y calidades críticas. Posteriormente se caracterizan los requisitos abstractos que deben conducir a plantear la arquitectura de referencia, para finalizar con las opciones arquitectónicas con que se cuenta.

5.2. Factores influyentes en la propuesta de la arquitectura

En el capítulo anterior se hizo un estudio del dominio de los robots de servicio teleoperados, estableciendo el propósito y características de estos sistemas. En este punto se finaliza el estudio determinando los factores de negocio, mercado potencial y objetivos de la organización que desarrolla la arquitectura, como punto de partida para obtener los requisitos funcionales y de calidad abstractos, así como las directrices arquitectónicas que son las primeras entradas de ABD para proponer en el capítulo siguiente la arquitectura de referencia ACROSET.

5.2.1. Factores de negocio

5.2.1.1. Mercado potencial

Uno de los factores de negocio más importantes es el mercado al que está destinado el producto. El mercado potencial de los sistemas de teleoperación considerados está constituido por:

- Empresas u organizaciones que de forma regular o esporádica deben realizar actividades que suponen riesgos para los operarios (centrales nucleares, plantas químicas, refinerías de petróleo, plataformas petrolíferas, astilleros, policías y cuerpos de seguridad del estado, ejército, etc).
- Es bastante habitual que se trate de grandes empresas, que subcontratan la mayor parte de los trabajos de mantenimiento a empresas de servicios, reservándose únicamente el control de calidad. Si participan en algún proyecto relacionado con un sistema de teleoperación utilizable en sus instalaciones su contribución suele limitarse a la especificación de requisitos.
- Empresas de servicios a las cuáles se subcontratan la mayoría de los trabajos y que son, por las razones ya explicadas, las primeras interesadas en sustituir a sus operarios por mecanismos teleoperados. Es frecuente que estas empresas aborden por sí mismas el diseño de sistemas de teleoperación si cuentan con el nivel tecnológico suficiente.
- Empresas u organismos que realizan labores de investigación y que necesitan manipular materiales en zonas peligrosas y entornos hostiles (p.ejm. estudio de volcanes, profundidades marinas).

Los párrafos anteriores presentan un escenario general que conoce, no obstante, bastantes excepciones. Los trabajos de mantenimiento de instalaciones no son las únicas aplicaciones posibles de los sistemas

de teleoperación, pero constituyen lo que podríamos llamar el grueso del negocio y en ellas no centraremos en los párrafos siguientes.

La clasificación de los trabajos que demandan las empresas según su planificación permite razonar acerca del tiempo disponible para el desarrollo de los sistemas y de su impacto económico, en términos de coste/beneficio. Según esta clasificación las actividades pueden dividirse en:

- Actividades rutinarias, previamente planificadas y ejecutadas habitualmente de forma periódica.
- Actividades esporádicas, no planificadas pero previsibles.
- Actividades imprevistas.

Las **actividades rutinarias** y previamente planificadas proporcionan a las empresas de servicios su fuente regular de ingresos. Su forma de ejecución está por lo general descrita en procedimientos de obligado cumplimiento definidos por la empresa suministradora del servicio, la propiedad y el fabricante de los equipos. La mayor parte de las operaciones de inspección y mantenimiento de las centrales nucleares y plantas químicas caen dentro de esta categoría. Dada la frecuencia con la que se realizan este tipo de actividades, la sustitución de operarios por máquinas incrementa espectacularmente la seguridad de las operaciones, disminuye en gran medida las necesidades de personal cualificado[1] y favorece una planificación racional de los recursos humanos.

Las **actividades espóradicas**, no previamente planificadas, pero previsibles, responden al suceso de incidencias durante las operaciones de mantenimiento planificadas o durante el régimen de funcionamiento de las instalaciones. La propiedad suele exigir la definición de medidas de contingencia para este tipo de incidentes si estima que la probabilidad de que se produzcan es elevada, si han ocurrido en el pasado, aun cuando sean muy poco probables, si existe alguna normativa o recomendación al respecto o si alguna empresa las ofrece como parte de sus servicios. Una vez que el cliente exige tales medidas, éstas pasan a formar parte del conjunto de actividades rutinarias, tanto si se realizan como si no. La empresa que esté en condiciones de ofrecer las mejores medidas de contingencia está en una posición ventajosa respecto de las demás.

Las **actividades imprevistas** responden al suceso de incidencias y accidentes que o bien son imprevisibles o bien su probabilidad de ocurrencia es tan pequeña que no se implementan métodos de contingencia para los mismos. Cuando ocurren es necesaria una gran capacidad de reacción. La resolución de este tipo de incidentes se realiza sobre la marcha, aprovechando los recursos existentes o implementando unos nuevos a toda prisa. La empresa que sea capaz de dar una solución a corto plazo se coloca en una situación privilegiada respecto de las demás y respecto de la propiedad.

Otro aspecto a considerar es el **perfil de los operadores** de los sistemas de teleoperación. Contrariamente a lo que pueda pensarse no suele ser personal con un gran bagaje técnico, al menos no en lo que al *software* y al uso de ordenadores se refiere. La mayoría de las empresas no contratan a personal especializado, sino que reciclan al personal que realiza las operaciones manualmente. Este tipo de usuario demanda una interfaz fácil de usar, que se adapte en la medida de lo posible a sus habilidades y conocimientos y que le asista en la ejecución de las operaciones.

5.2.1.2. Objetivos de la organización

En este caso la organización que define la arquitectura no es una empresa, sino una institución pública de enseñanza e investigación, la Universidad Politécnica de Cartagena. En concreto, el autor de esta tesis y los directores de la misma pertenecen al Grupo de Investigación DSIE (División de Sistemas e Ingeniería Electrónica), que engloba personal de los Departamentos de Automática e Ingeniería de Sistemas, Tecnología Electrónica y Tecnologías de la Información y la Comunicación.

---

[1] El personal debe ser igualmente cualificado, pero no se reduce la fatiga y el estrés asociado a la realización de actividades peligrosas, pues realiza las mismas desde zonas seguras.

Existe un conocimiento previo amplio del dominio de estudio. Como se dijo en el primer capítulo de esta tesis, la organización ha desarrollado bastantes robots de servicio teleoperados, aunque centrándose siempre en la plataforma de teleoperación[2]. Últimamente el DSIE ha realizado algunos desarrollos para la unidad de control de robots de limpieza de barcos [GOYA98a] [EFTCoR02].

El producto final no es un sistema de teleoperación concreto, sino una arquitectura de referencia. Los objetivos que se pretenden alcanzar con el desarrollo de tal arquitectura se inscriben dentro de la línea de investigación iniciada por Álvarez [Álvarez97-td] y suponen una continuación de la misma.

Tales objetivos pueden resumirse en los siguientes puntos:

- Obtención de una arquitectura de referencia, esta vez para la unidad de control local del sistema teleoperado, que tenga en cuenta:
  - ✓ Tanto los requisitos de calidad ya contemplados en [Álvarez97-td] y [Pastor02-td] como los que se añaden en este capítulo, específicos del nuevo dominio considerado.
  - ✓ Una caracterización precisa de los atributos de calidad de la unidad de control.
  - ✓ Los avances que sobre notaciones formales y metodologías de diseño y evaluación de arquitecturas se han producido en estos últimos años.
- Y que permita en la medida de lo posible:
  - ✓ La definición de arquitecturas específicas para sistemas concretos.
  - ✓ La reutilización, modificación y adaptación de la arquitectura en su implementación en diferentes sistemas del dominio.
  - ✓ El desarrollo rápido de aplicaciones.
  - ✓ La generación de un conjunto de componentes *software* reutilizables en las diferentes aplicaciones de la familia.
  - ✓ La integración de componentes *hardware* y *software* comerciales.
- Formar a personal investigador en el uso de técnicas formales o semi-formales de descripción, diseño y evaluación de arquitecturas.
- Crear un marco en el que puedan llevarse a cabo líneas de investigación más especializadas, relacionadas con los subsistemas descritos en la arquitectura o con dominios de teleoperación más reducidos.
- Establecer lazos de cooperación con la industria e impulsar líneas de investigación que puedan aplicarse en la misma a través de la correspondiente transferencia de tecnología.
- Desarrollar aplicaciones de bajo coste. Este es un factor de negocio muy importante para la organización con objeto de ser competitivos frente a otros productos comerciales, como se demuestra en los últimos prototipos que ha fabricado el DSIE, como GOYA y EFTCoR, que incluyen como premisa el bajo coste del producto final.

5.3. Directrices de la arquitectura

Las "directrices[3] de la arquitectura", según [Bass99], son la combinación de requisitos funcionales, de calidad y negocio que modelan la arquitectura. Si se pueden cumplir sus directrices, el sistema será

[2] En aquellos trabajos, siempre se ha dividido un sistema de teleoperación en dos partes bien diferenciadas: la plataforma de teleoperación y la unidad de control remota. Esto se debía en gran medida a que la unidad de control del robot solía ser comercial y sin posibilidad de modificación.

[3] Del inglés *architectural drivers*

diseñado satisfactoriamente. En primera instancia, a un nivel como el que nos encontramos, en el que se han establecido las características del dominio para el que se diseña la arquitectura de referencia y conocemos las necesidades críticas de negocio de la organización que la desarrolla, un *arquitecto* con experiencia puede echar un vistazo a dichos factores y características, e identificar cuatro o cinco puntos de variación arquitectónicos diferentes.

Directrices Fundamentales de la Arquitectura	
Relacionadas con el propósito de los sistemas	
D1	La arquitectura servirá para desarrollar unidades de control robots de servicio teleoperados, que deben controlar y coordinar las acciones del robot y herramientas según las órdenes del operador remoto que le llegan del sistema de teleoperación, o de otros usuarios externos y según la información que dispone de los sensores.
D2	Las operaciones a realizar por el robot se centrarán sobre todo en operaciones de mantenimiento, reparación e inspección no repetitivas, a realizar en entornos hostiles, cambiantes, remotos y/o inaccesibles para un operador humano, lo que imposibilita la actuación totalmente autónoma del robot.
D3	Se busca el desarrollo rápido y de bajo coste del mayor número posible de sistemas dentro del dominio de la arquitectura.
Relacionadas con la orientación a componentes	
D4	Diferentes instanciaciones de la arquitectura deben poder compartir, reutilizar, los mismos componentes.
D5	Se deben adoptar políticas que permitan una clara separación entre dichos componentes y sus patrones de interacción.
Relacionadas con la adaptación a distintas implementaciones	
D6	La implementación de los componentes de la arquitectura podrá ser *software* o *hardware*, pudiendo ser también componentes comerciales COTS. La implementación podrá realizarse sobre plataformas diversas, incluso distribuidas.
D7	Adaptabilidad a diferentes mecanismos. Debe ser fácil añadir y quitar componentes y capacidades del sistema.
Relacionadas con la integración de inteligencia	
D8	Debe ser posible que los sistemas adopten ciertas acciones autónomas sencillas (principalmente reactivas) que complementen las acciones teleoperadas o incluso misiones pre-programadas (deliberativas).
D9	Debe poder gestionar distintos modos de control, distintos interfaces de teleoperación y otros sistemas externos que interaccionan con el sistema, incluso de forma simultánea.
Relacionadas con el rendimiento	
D10	La presencia de enlaces de comunicaciones, la posibilidad de distribución y de diferentes particionados *hardware*/*software* debe tenerse en cuenta.
D11	Los requisitos de tiempo-real críticos son condicionantes para el funcionamiento correcto del sistema, aunque en los sistemas también se encuentran requisitos que no son de tiempo-real.
Relacionadas con la seguridad[4]	
D12	Las operaciones sólo deben ejecutarse si existe una confianza razonable en que el resultado de las mismas no va a poner en peligro la integridad de bienes y equipos.
D13	Deben adoptarse mecanismos de tolerancia a fallos, tratamiento de excepciones, errores, mecanismos de diagnóstico para prevención de fallos, etc.

Tabla 5.1.- Directrices arquitectónicas fundamentales para ACROSET

[4] En la bibliografía escrita en inglés, se diferencia entre safety y security, que se traducen de la misma forma, seguridad, pero hacen referencia a conceptos distintos. Safety se refiere más bien a la seguridad de obtener los resultados esperados y security hace referencia más bien a asegurar que el acceso al sistema lo hacen sólo las personas que tengan permiso de acceso. La directriz referente a seguridad se identifica con safety.

Sin embargo, la determinación de directrices no es siempre un proceso *top-down*. Puede involucrar una investigación más detallada de algunos aspectos de los requisitos para comprender las implicaciones arquitectónicas que pudieran tener.

A un nivel mayor de detalle, las directrices se determinarán de los requisitos particulares de los subsistemas o componentes conceptuales. Hay que destacar que las directrices no dependen de los detalles de los requisitos funcionales, sino de una abstracción de los mismos. Esto quiere decir, que las actividades de diseño pueden empezar tan pronto como las directrices se establezcan de manera fiable, lo cual puede suceder muy pronto en el ciclo de desarrollo, dependiendo de la familiaridad que tengan los desarrolladores con el dominio.

De una primera observación del propósito de los robots de servicio teleoperados, las características de este dominio, en concreto, las características de la unidad de control y los factores de negocio que influyen en la organización que desarrolla la arquitectura ACROSET, vistos en el punto anterior, se pueden extraer una serie de directrices fundamentales (**Tabla 5.1**).

Puede llamar la atención que de entrada se establezcan tantas directrices arquitectónicas. Probablemente en este caso son muy numerosas por el amplio conocimiento que se tiene del dominio de aplicación y por el planteamiento riguroso de características de estos sistemas que se ha establecido. En otros casos, quizá de partida no se puedan plantear tantas directrices, y tendrán que ir surgiendo en el proceso iterativo de desarrollo de la arquitectura de un sistema o familia de sistemas.

5.4. Requisitos de calidad abstractos

Los requisitos de una arquitectura de referencia rara vez pueden expresarse concretamente. La especificación de requisitos en un alto nivel de abstracción se hace necesaria por dos razones:

- ✓ La necesidad de capturar las variantes de funcionalidad entre todos los sistemas de la familia.
- ✓ La imposibilidad de conocer con antelación los requisitos específicos de cada sistema concreto.

Pero aún en este nivel de abstracción es necesario realizar una caracterización de los atributos tan precisa como sea posible. El rango en el que se mueven las variantes debe describirse siempre que se conozca y es aconsejable[5] enumerar los **mecanismos[6] arquitectónicos** u opciones arquitectónicas más adecuadas para la obtención de los atributos de calidad. Soni, Hofmeister y Nord, en [Hofmeister99], denominan como "estrategias" a estas opciones arquitectónicas, como se comenta en el propio ABD.

El enfoque de este apartado es el siguiente: partiendo de las directrices definidas en el apartado anterior y de las características de los sistemas de teleoperación expuestas en el Capítulo 4, se desglosará cada atributo general en sus diferentes aspectos. Cada uno de estos aspectos puede ser considerado como un atributo de calidad *abstracto* que debe caracterizarse en la medida de lo posible. Para ello, se utilizará la plantilla mostrada en la **Fig. 5.2**, inspirada en la propuesta de [Bass00] para la caracterización de atributos de calidad completamente abstractos (no ligados a ningún sistema).

La elaboración de estas plantillas debe proporcionar:

- Una organización jerárquica de los requisitos. En el primer nivel, los requisitos de las plantillas son completamente abstractos. A medida que se avanza en profundidad la especificación se va haciendo más concreta. Las hojas del árbol se corresponden con los requisitos de sistemas específicos.

---

[5] La identificación de mecanismos arquitectónicos es realmente parte del proceso de diseño, sin embargo el ABD recomienda enumerarlos al mismo tiempo que se especifican los requisitos. Esta enumeración no compromete ninguna decisión, pero puede ayudar a tomarla.

[6] Mecanismo: Estructura por medio de la cual los componentes colaboran para proporcionar un comportamiento que satisfaga los requisitos del problema [Booch 1994].

- Una base de conocimiento para la selección de estilos arquitectónicos y patrones de diseño. Este extremo se consigue mediante la inclusión en las plantillas de los mecanismos de diseño candidatos para la obtención de un atributo.

Número Atributo: Nombre del Atributo

Descripción general.

Se describe en unas pocas líneas el propósito del atributo.

Escenarios Abstractos.

Se definen escenarios abstractos que ayuden a determinar el alcance de los aspectos descritos en el punto anterior.

Los escenarios deben describirse en términos de los estimulos proporcionados al sistema y de las respuestas que debe proporcionar el mismo. Si existen modelos formales asociados al atributo, es aconsejable que dichos estímulos y respuestas se correspondan con los parámetros de entrada y salida de alguno de ellos. Esta forma de proceder facilita la selección de estrategias y mecanismos arquitectónicos y también el posterior análisis de la arquitectura.

Estrategias y Mecanismos arquitectónicos.

Definen los patrones y estilos arquitectónicos más apropiados para satisfacer el atributo de calidad, para la realización de los escenarios generales. Su enumeración explícita facilitará la comparación entre las alternativas posibles y ayudará a una posterior evaluación de la alternativa escogida frente a las otras posibles.

Las estrategias se refieren a paradigmas y técnicas de programación generales, mientras que los mecanismos apuntan a soluciones más concretas. Por ejemplo, estrategias son el encapsulado, la separación de conceptos o proponer el uso de alguna técnica de asignación de recursos. Los Mecanismos definen patrones arquitectónicos básicos (adaptador [Gamma95]) o complejos (cliente/servidor [Shaw96]) y describen técnicas concretas para implementar las estrategias arquitectónicas (planificación mediante prioridades fijas, ejecutivo cíclico, etc).

Relación con otras directrices.

Se enumeran las directrices con las que se relaciona y se explica brevemente el motivo de la relación (interacciona con..a través de…, mecanismos incompatibles con.., se deriva de…, condiciona la…, favore a…, es favorecido por…, etc)

Relación con directrices de la arquitectura.

Se explica brevemente como se deriva el requisito de los factores de negocio y del propósito y características de los sistemas.

Observaciones.

Cualquier información que se estime relevante no incluida en los apartados anteriores.

Fig. 5.2.- Modelo para las Plantillas de Atributo

Con objeto de no cansar al lector con excesivos detalles, las plantillas elaboradas se incluyen en el Anexo II, comentándose sus aspectos más relevantes en los siguientes apartados e incluyéndose un resumen de las mismas en la **Tabla 5.3.**

De la misma manera, las responsabilidades funcionales abstractas de los sistemas del dominio de la tesis se abordarán en el punto 5.5. Estos requisitos funcionales abstractos deberán sufrir refinamientos sucesivos hasta dar lugar a requisitos funcionales concretos aplicables a sistemas específicos, por ejemplo, los presentados en el Capítulo 7.

5.4.1. Atributos de calidad abstractos

Siguiendo el esquema propuesto en [Bass00] los atributos de calidad abstractos se clasifican en las siguientes categorías:

- Aspectos del Negocio y Objetivos de la Organización
- Aspectos de la Modificabilidad.
- Aspectos del Rendimiento.
- Aspectos de la Seguridad.
- Aspectos de la Fiabilidad/Disponibilidad.
- Aspectos de la Usabilidad.
- Aspectos de la Interoperabilidad.

5.4.1.1. Aspectos del Negocio y Objetivos de la Organización

El desarrollo rápido de aplicaciones es un objetivo marcado por la organización a nivel particular, es decir, para los sistemas que suele desarrollar con transferencia de tecnología a la empresa. A nivel de la arquitectura de referencia para el dominio considerado, puede que no sea un requisito para todos los sistemas, pero sí que es muy frecuente en cualquier empresa privada, puesto que las exigencias de competitividad son cada vez más elevadas y también se presenta en organizaciones que se dedican a la investigación, como Universidades y otros organismos públicos, puesto que este requisito, suele ir muy ligado con la *reutilización* de componentes, la *modularidad* de los sistemas, etc., que no sólo inciden en la rapidez de desarrollo de nuevos sistemas, sino también en la garantía de calidad de los mismos.

El coste competitivo de los productos desarrollados también se ha incluido por ser un objetivo típico de organizaciones que necesitan competir con otras empresas. Si se desarrolla un producto, lo lógico es que no haya fondos ilimitados para desarrollarlo, y por supuesto, que su comercialización no implique precios demasiado elevados. Por tanto, al diseñar, se debe estudiar qué elementos de los disponibles ofrecen las prestaciones requeridas a un coste mínimo.

Aunque estos son los requisitos que normalmente impone la organización que propone la arquitectura de referencia (DSIE), no tienen porqué ser los requisitos de cualquier sistema a desarrollar a partir de esta arquitectura. Se verá que ACROSET facilita el desarrollo rápido de aplicaciones a un coste competitivo gracias a los componentes de abstracción del *hardware* y a la posibilidad de integrar COTS en la implementación de la arquitectura, pero esto no impide que se desarrollen sistemas que no impongan estos requisitos.

5.4.1.2. Aspectos de la Modificabilidad

Los diferentes aspectos de la modificabilidad se refieren a la portabilidad, a la capacidad de distribución, a la integración de servicios y utilidades, a la adaptabilidad frente a cambios en las interfaces de usuario y en las unidades de control local, a las modificaciones relacionadas con cambios en el entorno operativo, en los mecanismos y en las misiones y finalmente, con la posibilidad de integrar herramientas, bibliotecas y componentes *software* o *hardware* comerciales. La modificabilidad es un aspecto al que hay que prestar especial atención por el hecho de tratarse de una

arquitectura de referencia, puesto que la generalidad buscada, normalmente implica que deberá poder adaptarse a la modificación de sus posibles implementaciones.

La adaptabilidad a diferentes entornos de operación puede considerarse desde dos puntos de vista:

- Posibilidad de construir aplicaciones que operan en diferentes entornos.
- Posibilidad de adaptar un sistema dado para que pueda trabajar en diferentes entornos.

Puesto que la adaptación a diferentes entornos se basa en el uso de ciertos servicios, este atributo puede verse como un caso particular de la incorporación de nuevos servicios, y como aquel está relacionado con la posibilidad de utilizar herramientas y componentes *software* comerciales.

En el segundo caso la posibilidad de adaptar al sistema para trabajar en diferentes entornos está estrechamente relacionada con la posibilidad de adaptarlo a diferentes mecanismos y a diferentes misiones. Cada mecanismo tiene unas propiedades (mecánicas, eléctricas, estructurales, etc.) que le permiten realizar ciertas actividades en ciertos entornos, (por ejemplo, debería ser capaz de incorporar un cálculo de cinemática distinto si el mecanismo cambia), y también ser capaces de admitir la incorporación de nuevos sensores o bien de nuevas interfaces de teleoperación si así fuera necesario. Si el cambio entre entornos implica a dos entornos estructurados normalmente es suficiente con disponer de utilidades de modelado y representación gráfica. Si el entorno cambia, basta con modificar el modelo. En entornos no estructurados la utilidad de estas herramientas es limitada y puede ser necesaria la incorporación de un sistema de visión artificial u otras utilidades que permitan al sistema recoger información de su entorno y actuar en consecuencia.

La adaptabilidad a diferentes mecanismos (brazos, vehículos y herramientas) está muy relacionada con la propiedad anterior y viene motivada por:

- La especialización de los sistemas: Cada sistema controla unos mecanismos específicos.
- La posibilidad de que un mismo sistema realice diferentes trabajos. Para ello pueden ser necesarias diferentes combinaciones de mecanismos (p.e: un mismo brazo robotizado debe portar diferentes herramientas según el tipo de operación que debe realizar).

La adición, sustitución o modificación de los mecanismos que constituyen el sistema es una actividad bastante habitual, que a menudo debe realizarse con el sistema en funcionamiento. El logro de este requisito implica un modelado muy cuidadoso de las características estructurales y dinámicas de los mecanismos, la definición de interfaces abstractas de acceso a los mismos y la definición de controladores genéricos. También, como se ha comentado antes, puede ser necesario adaptarse no a un nuevo mecanismo, sino a una diferente configuración, que implique, por ejemplo, un cambio en el cálculo cinemático, u otros.

La adaptabilidad a diferentes misiones está relacionada con:

- La especialización de los sistemas: Cada sistema debe realizar un conjunto de misiones, en general muy específicas.
- La mantenibilidad del sistema: La ejecución de una misión determinada está sujeta a modificaciones. Durante la vida de un sistema es posible que surjan nuevas actividades a realizar.

Una misión es la ejecución por los mecanismos de una serie de actividades, realizadas en secuencia o según la lógica impuesta por una máquina de estados. En ocasiones una misión compleja no es más que la composición de otras más simples. Una misión es sobre todo una *pieza de funcionalidad*, que implica la ejecución de ciertos procedimientos y el acceso a diferentes servicios. Es necesario separar los aspectos ligados a la misión del resto de la funcionalidad.

Puesto que las misiones se realizan en un determinado entorno y utilizando determinados mecanismos, la adaptabilidad a nuevas misiones está ligada a la adaptabilidad a diferentes entornos, mecanismos y configuraciones (combinaciones vehículo-brazo-herramienta). En ocasiones la misión sólo implica a un mecanismo (realización de un proceso de herramienta o ejecución de un movimiento o una

secuencia de movimientos). En otras implica a varios. Para que varios mecanismos colaboren en una misión es necesario que sincronicen sus actividades e intercambien mensajes. Esto revela una nueva dimensión de la adaptabilidad a nuevas misiones: la necesidad de establecer medios de comunicación flexibles entre los controladores de los mecanismos.

Puesto que la realización de ciertas misiones necesita de ciertos servicios, la adaptabilidad a nuevas misiones está relacionada con la incorporación de nuevos servicios, y a través de ésta con la posibilidad de utilizar herramientas y componentes *software* comerciales. Aún más: A menudo las misiones deben seguir un procedimiento más o menos estricto que debe ser seguido por el operador. La aplicación debe guiar la ejecución de las misiones, razón por la cual la adaptabilidad a nuevas misiones está relacionada con la adaptabilidad a nuevas interfaces y a la usabilidad.

La portabilidad se refiere a la necesidad de los sistemas a adaptarse a diferentes infraestructuras, entendiéndose éstas como los servicios que proporcionan el **sistema operativo**, los **enlaces de comunicaciones**, distintas **distribuciones *hardware-software*** y el *middleware*. La independencia de la infraestructura es una directriz de diseño que se deriva directamente de las características de los sistemas. Aun cuando la portabilidad puede no ser un requisito de todos sistemas del dominio, diferentes sistemas pueden requerir diferentes infraestructuras. La necesidad de utilizar *hardware* de control específico, que será diferente en cada sistema, refuerza aún más este requisito. La definición de una interfaz abstracta de acceso a la infraestructura y la adopción de estándares son claves para lograr la portabilidad.

La capacidad de distribución se refiere a la posibilidad de asignar los diferentes subsistemas a diferentes nodos del sistema, a la adaptabilidad a diferentes despliegues. La capacidad de distribución viene motivada por el carácter distribuido de los sistemas, que habitualmente constan de dos o más nodos, por la necesidad de ubicar las tareas más críticas o más exigentes en términos de computación en nodos específicos y de definir estrategias de recuperación y de tolerancia a fallos (si un nodo cae sus responsabilidades pueden ser asumidas por otro). La capacidad de distribución exige ser muy cuidadoso en la división de la funcionalidad y en la definición de los patrones de interacción entre susbsistemas que pueden ejecutarse en diferentes nodos (sólo debe admitirse el paso de mensajes).

La **independencia de la infraestructura de comunicaciones** está muy relacionada con la capacidad de distribución y la portabilidad, puesto que al distribuir el sistema según diferentes despliegues, pueden aparecer modos de comunicación diferentes a los que la arquitectura debe poder adaptarse.

La posibilidad de integrar componentes o herramientas *software* comerciales viene motivada por la necesidad de cumplir el resto de los requisitos en un plazo de tiempo razonable (reducción de costes, aumento de calidad y a veces, aumento del rendimiento). Debe existir la posibilidad de usar componentes comerciales que ofrezcan servicios interesantes para el sistema, bien directamente (herramientas *software*), bien indirectamente (a través de bibliotecas *software*). Aunque puede considerarse un caso particular de la integración de servicios, presenta sin embargo diferencias importantes:

Los COTS[7] pueden utilizarse para proporcionar dichos servicios directamente o para implementarlos.

El uso de COTS tiene su problemática propia:

- Compatibilidad entre diferentes versiones.
- Compatibilidad entre COTS.
- Suposiciones arquitectónicas de los COTS.
- Sustitución de unos COTS por otros.

La **posibilidad de integrar componentes *hardware* comerciales o de desarrollo propio**, es muy importante en este tipo de sistemas. Aquí hay que tener en cuenta muchos factores. Quizá la mejor manera de explicarlo sea con un ejemplo:

---

[7] *Commercial off-the shelf.*

Un elemento que casi con seguridad aparecerá en la unidad de control de un robot, es el controlador de una de sus articulaciones, normalmente un servo-motor eléctrico (dependiendo de la configuración del robot, puede ser lineal, angular, relacionada con otras articulaciones según una cinemática, etc), aunque también podría ser un accionamiento neumático o hidráulico. Digamos que ese control se hace inicialmente implementando un componente *software* con una interfaz bien definida, que ofrece una serie de servicios, como la posibilidad de mover el motor, preguntar por su estado, su posición, etc. La ventaja de una implementación totalmente *software* estriba sobre todo en su **flexibilidad**, en las posibilidades prácticamente infinitas de realizar el control deseado. Puede ser que el **rendimiento** alcanzado al controlar la articulación de esta forma sea el adecuado, y se llegue a plantear una serie de algoritmos de control probados que puedan seleccionarse o cambiarse incluso en tiempo de ejecución.

Llegados a este punto, quizá interese mejorar el rendimiento del sistema distribuyendo este control en un nodo de procesamiento distinto; se puede optar por implementarlo a nivel *hardware* diseñado de manera específica, como por ejemplo, implementándolo en una FPGA, o bien utilizando un *hardware* comercial. La solución de la FPGA probablemente aumente el rendimiento del sistema, prácticamente sin perder flexibilidad en el diseño. La solución del *hardware* comercial (por ejemplo, una tarjeta controladora de motores) puede acortar enormemente los **tiempos de desarrollo** y aumentar también el rendimiento del sistema, probablemente a costa de perder flexibilidad.

Optar por una solución u otra depende de las necesidades de la aplicación, del sistema a diseñar. Lo que sí debe garantizar la arquitectura es que cualquiera de estas vías sea posible, que si se quiere optar inicialmente por una, para después cambiar a otra, no sea necesario reestructurar todo el sistema. Básicamente la garantía de que esto se pueda hacer pasa por que las **interfaces** estén bien definidas construyendo en su caso una capa de abstracción del *hardware* que permita reutilizar estas interfaces, y teniendo en cuenta también los **conectores** y los patrones de interacción entre los componentes del sistema.

Integración de servicios o utilidades. Es muy habitual que los sistemas de teleoperación necesiten utilidades de simulación, representación gráfica, visión artificial, navegación, realimentación de pares y fuerzas, etc. Estos servicios pueden emplearse en línea proporcionando información al resto de los subsistemas o fuera de línea para actividades de entrenamiento o de definición de misiones. Las aplicaciones más sencillas sólo demandan alguna de ellas (a veces, ninguna), las más complejas pueden necesitar de muchas. Se debe facilitar dicha integración con interfaces bien definidas.

La integración de servicios está estrechamente relacionada con la posibilidad de integrar *software* y *hardware* comercial en los sistemas, en especial para implementar servicios de representación gráfica, de cálculos cinemáticos, de navegación, de realimentación sensorial, de seguridad, etc.

Adaptabilidad a distintas interfaces de usuario. Los sistemas deben adaptarse a interfaces de usuario muy diferentes, que incluyen dispositivos específicos como *joysticks*, botoneras, reproducciones a escala de los dispositivos, utilidades de visión artificial, etc. y con muy distintas necesidades de representación gráfica y disposición de las ventanas. Las interfaces de usuario son uno de los componentes del sistema más sujetos a sufrir variaciones tanto entre sistemas como durante la vida operativa de un sistema. La separación de la interfaz de usuario del resto de la aplicación es, por tanto, fundamental. La adaptabilidad a distintas interfaces de usuario puede considerarse un caso particular de la adaptación a nuevos servicios y está estrechamente relacionada con la posibilidad de utilizar herramientas y componentes *software* comerciales.

La gestión de distintos modos de control y operación es otro aspecto fundamental, pues lo normal es que en este tipo de sistemas coexistan diferentes modos de control (local, teleoperado, supervisado, colaborativo, autónomo) y de operación (modo calibración, simulado, test, realización de secuencias programadas, etc), y que el sistema deba garantizar el paso de uno a otro según los niveles de permiso que tenga el operador y la operación que se esté realizando, incluso la interacción de varios modos a la vez (como cuando se está realizando un movimiento autónomo y el operador desea corregir ciertos comportamientos).

5.4.1.3. Aspectos del Rendimiento

Dado lo amplio del dominio, el rendimiento sólo puede caracterizarse con un muy alto grado de abstracción, ya que se desconocen los eventos concretos a los que deben responder los diferentes sistemas, así como los plazos en que los mismos deben ser procesados. No obstante, es posible caracterizar algunos aspectos generales del comportamiento temporal que podrán ser refinados más adelante. Desde el punto de vista arquitectónico conviene clasificar las tareas según su nivel de criticidad, el tipo de eventos a los que deben responder y la forma en que el sistema debe responder a dichos eventos. En general, aunque en los sistemas de teleoperación pueden existir tareas que respondan a cualquier combinación de criticidad, tipo de eventos y características de la respuesta, es posible distinguir dos categorías principales:

- **Tareas críticas** que deben realizarse dentro de unos plazos estrictos. Suelen estar relacionadas con procesos de control de los mecanismos (vehículo-brazo-herramienta) cercanos al *hardware*, tales como el control del bucle de realimentación de los servos, la lectura de sensores y el accionamiento de los actuadores y procesos o servicios relacionados con la parada segura del sistema. Tradicionalmente estas tareas se asocian al comportamiento reactivo del control.
- **Tareas menos críticas o no-críticas.** Son en general procesos de control de alto nivel no directamente relacionados con el funcionamiento seguro del sistema. La planificación y secuenciación de las misiones, la captura de datos que pueden procesarse en diferido y dicho procesamiento, ciertos procesos de diagnosis, la actualización de la interfaz de usuario, el suministro de ciertos servicios, etc. entran habitualmente dentro de esta categoría. En general, tareas más asociadas con el comportamiento deliberativo del control.

Interacciones del rendimiento	
Portabilidad	• Los mecanismos que facilitan la portabilidad (nivel de acceso) pueden suponer un empeoramiento del rendimiento a través de las llamadas entre módulos. • No todos los sistemas operativos ofrecen los servicios que pueden ser necesarios (procesos ligeros, planificación con prioridades, predictabilidad, etc). • No todos los enlaces de comunicaciones tienen el ancho de banda adecuado, ni son determinísticos. • Por otro lado, la posibilidad de utilizar buses de alto rendimiento y sistemas operativos de tiempo real favorecen el rendimiento.
Integración de servicios	Si las tareas críticas necesitan de ciertos servicios, el acceso a los mismos y su procesamiento se convierten asimismo en tareas críticas. Los mecanismos que facilitan la integración de servicios (mediadores, módulos de desacoplo, etc.) pueden suponer un empeoramiento del rendimiento a través de las llamadas entre módulos.
Uso de COTS	Deben proporcionar el rendimiento adecuado y no hacer suposiciones arquitecturales que afecten o entren en conflicto con las estrategias seleccionadas. La posibilidad de integrar *hardware* específico puede mejorar mucho el rendimiento.
Flexibilidad en el despliegue	Puede mejorar el rendimiento, pues facilita la asignación de recursos a diferentes nodos y favorece la definición de nodos *autosuficientes*. (Minimización de las comunicaciones).
Otros aspectos de la modificabilidad	En general la modificabilidad se consigue aumentando los niveles de indirección, que suponen una degradación del rendimiento.
Seguridad	La seguridad depende del rendimiento. Si las tareas críticas no cumplen sus plazos el funcionamiento puede degradarse hasta límites inaceptables.
Usabilidad	El rendimiento favorece la usabilidad. La información que se ofrece al operador debe reflejar el estado actual del sistema y los comandos deben activarse *inmediatamente*, dentro de los límites de la percepción humana.

Tabla 5.2 – Interacciones del rendimiento con otros atributos

Las prestaciones de la plataforma *hardware* de implementación, la necesidad de un volumen mayor o menor de **transmisión de datos**, la velocidad a la que se deba mover el sistema, la rapidez de respuesta, la necesidad de actualización del estado de los sensores, el tipo de actualización que se realice, la presencia de más o menos nodos de procesamiento o de *hardware* dedicado son ejemplos importantes de factores que influirán muchísimo en las estrategias de planificación que se deban adoptar para garantizar los tiempos de respuesta de las tareas críticas. Como se puede deducir de lo anterior, el rendimiento interacciona con el resto de los atributos de calidad, normalmente entrando en conflicto con ellos como se muestra en la Tabla 5.2. Se deberá adoptar un compromiso en cada sistema concreto que se implemente a partir de la arquitectura para favorecer un aspecto u otro frente al rendimiento, o bien priorizar el rendimiento frente a los demás requisitos. La arquitectura debe ser lo suficientemente flexible para adaptarse a distintos compromisos, pero a su vez, deberá garantizar unos mínimos aspectos (relacionados con modificabilidad, seguridad, etc.) aunque el rendimiento global se vea perjudicado.

5.4.1.4. Aspectos de la Seguridad

Antes de reflexionar sobre los aspectos de seguridad que hay que tener en cuenta para el dominio de aplicación, hay que definir una serie de términos que en muchas ocasiones se confunden: *seguridad, fiabilidad, fallo, error, excepción*.

Según se define en [Douglass00], por sistema **seguro**[8] se entiende un sistema que no crea accidentes que puedan causar daños a bienes o personas. Por **fiable**[9], se entiende que será un sistema que continúa funcionando durante largos periodos de tiempo (ver siguiente punto). Un sistema seguro puede fallar, pero sin necesidad de originar un accidente. Un sistema puede ser seguro, pero no fiable y viceversa. En muchos sistemas de tiempo-real, a menudo estos dos conceptos van unidos porque no tienen definidas condiciones de fallo-seguro (por ejemplo, en el controlador de vuelo de un avión el sistema debe ser seguro y fiable). En entornos de seguridad-crítica, los sistemas no sólo deben funcionar cuando todo va bien, sino que deben tener comportamientos seguros cuando algo falla.

Un **fallo** es un evento que sucede en un instante particular del tiempo. Este concepto es distinto del **error**, que es un fallo sistemático. Un error es un defecto persistente debido a un error de diseño o implementación.

El manejo de **excepciones**, a pesar de tener una influencia primordial para conseguir que un sistema sea tolerante a fallos, no se contempla a nivel de diseño arquitectónico, sino en la fase de diseño detallado. Según [Douglass00], son un tipo especial de señales que se envían en un nivel muy próximo a la implementación, en la programación, porque corresponden a errores de rango, fallos de lectura, *overflow*, etc., que se han de prever añadiendo al programa mecanismos de gestión de excepciones, pero está implícito en el código, no es un componente de diseño conceptual.

En el dominio concreto de los sistemas de teleoperación, la seguridad y la prevención de accidentes es un aspecto crítico. Aparte del riesgo inherente al uso de los propios mecanismos, éstos trabajan en zonas de riesgo donde la probabilidad de incidencias y accidentes es de por sí más alta de lo habitual. Los accidentes se deben a muchos factores (en ocasiones hace falta que confluyan varios para que se produzca el accidente), algunos de los cuáles tienen poco que ver con el *software*. De entre ellos, se pueden destacar [Alvarez97-td]:

- Mal funcionamiento del sistema de control (*hardware* y *software*).
- Acceso indebido del personal a la zona de trabajo de los mecanismos.
- Errores humanos de los operarios.
- Roturas de partes mecánicas.

---

[8] En inglés se distingue entre *safety*: seguridad relativa a la ocurrencia de fallos que puedan provocar accidentes, y *security*: aspectos relacionados con la protección y control de acceso a un sistema.

[9] La disponibilidad va ligada normalmente con la fiabilidad (define el tiempo medio entre fallos).

- Liberación de energía almacenada.
- Sobrecarga de los mecanismos.
- Medio ambiente o herramientas peligrosas.

Ciertos factores dependen exclusivamente de las medidas de seguridad de las instalaciones y de las que puedan definirse en los procedimientos de ejecución de las operaciones. Otros dependen del *software* y la fiabilidad del mismo. En los sistemas de teleoperación es necesario supervisar constantemente el funcionamiento del sistema, gestionar la ejecución de los comandos y proporcionar mecanismos de parada segura que se activen por orden del operador o cuando se detecten situaciones de riesgo o errores graves de funcionamiento.

La gestión de los comandos es necesaria para impedir que se ejecuten comandos inseguros y asegurar que los comandos se ejecutan según el plan previsto. Puede considerarse un caso particular de la monitorización del estado, sin embargo la ejecución de comandos introduce ciertas condiciones particulares. Una parte del estado del sistema debe supervisarse siempre, haya o no haya un comando en curso, otras partes son relevantes precisamente por el comando que se está ejecutando. Es necesario separar los aspectos específicos de cada comando.

La seguridad influye fuertemente en el diseño del sistema, ya que sus componentes deben pensarse desde un principio para ser seguros. Los componentes críticos deben proporcionar interfaces a través de las cuales un servicio de diagnóstico o una tarea de supervisión puedan detectar fallos en su funcionamiento. Asimismo, la seguridad está estrechamente relacionada con el rendimiento y la disponibilidad. Por un lado, los requisitos temporales deben ser garantizados para asegurar que la información disponible sobre el sistema refleja fielmente su estado, pero por otro las tareas encargadas de monitorizar el funcionamiento del sistema suponen una sobrecarga del mismo. En general los mecanismos y estrategias arquitecturales favorecen la disponibilidad son los mismos que favorecen la seguridad.

Un aspecto distinto de la seguridad es el relacionado con el control de acceso al sistema y permisos de ejecución de ciertas funciones del mismo, modificación de parámetros de configuración, etc. Se debe garantizar el acceso a dichas funciones sólo a los operarios cualificados, (a través de claves de acceso u otros mecanismos), precisamente para que no se produzcan empleos incorrectos o peligrosos del sistema.

5.4.1.5. Aspectos de la Disponibilidad/Fiabilidad

El que un sistema sea seguro no garantiza que no halla fallos. La disponibilidad se mide en términos de la tasa de fallos (número de fallos por unidad de tiempo) y del tiempo de reparación (número de reparaciones por unidad de tiempo o porcentaje de tiempo que el sistema está de baja debido a reparaciones). Sin embargo, estos son aspectos muy dependientes del sistema y deberán abordarse cuando se emplee la arquitectura en los sistemas concretos. Por el momento es suficiente con caracterizar la disponibilidad mediante la identificación de los servicios o funciones más críticos de los sistemas, y enumerar las estrategias y mecanismos que debe proporcionar la arquitectura para que dichos servicios estén disponibles y muestren un comportamiento fiable. En general, dichas estrategias y mecanismos se basan en aplicar diversas formas de redundancia homogénea o heterogénea. Los aspectos a destacar de este atributo son:

- Definición de modos degradados de funcionamiento.
- La inexistencia de modos comunes en los que un único fallo puede provocar la caída de todo el sistema.
- La disponibilidad y fiabilidad de los enlaces de comunicaciones: Los fallos en los enlaces de comunicaciones entre nodos deben al menos detectarse y si es posible corregirse.
- La disponibilidad y fiabilidad de las interfaces de usuario: El operador debe disponer de forma permanente de una interfaz de usuario mínima que le permita ejecutar al menos la parada

segura del sistema y el rearranque del mismo. Asimismo, debe existir una interfaz reducida mediante la cual puedan ejecutarse los comandos más básicos de los mecanismos.

- La disponibilidad y fiabilidad de los mecanismos de parada segura. Debe existir al menos un mecanismo de parada segura accesible por el operador y cuyo funcionamiento no dependa del comportamiento de la aplicación.

Cuando se aborda el estudio de la disponibilidad/fiabilidad suelen tenerse en cuenta dos aspectos importantes relacionados con el funcionamiento seguro del sistema:

- La existencia de mecanismos de fallo seguro (*fail-safe*)
- La posibilidad de admitir modos de funcionamiento degradado.

Fallo seguro significa que cuando ocurre alguna circunstancia que impide el funcionamiento normal del sistema (el fallo de un componente *hardware* o *software*, el fallo de las comunicaciones, un corte de energía eléctrica, etc.) el sistema debe pasar a un estado conocido y seguro. El significado de lo que es seguro depende de cada sistema. Algunos ejemplos son:

- Estado *sistema apagado*
  - ✓ Parada de emergencia – cortar alimentación inmediatamente.
  - ✓ Parada de producción – cerrar *(shutdown)* el sistema tan pronto como se complete la tarea que se está realizando.
  - ✓ Parada de protección – cerrar el sistema inmediatamente pero sin quitar la alimentación
- Cierre *(shutdown)* parcial – llevar a un nivel de funcionalidad degradado
- Mantener *(hold)* – no se podrán realizar tareas funcionales, pero sí se realizarán acciones de seguridad de manera automática.
- Control manual o externo – el sistema continua funcionando pero sólo vía entradas externas.
- Modo degradado. Se renuncia a la parte de funcionalidad que falla, hasta que consigue repararse.
- Reiniciar *(restart)* – el sistema se reinicia. Posiblemente la medida de recuperación más popular (por desgracia) de todos los sistemas informáticos.

El dominio del problema generalmente hace desestimar varias de estas opciones. Por ejemplo, un motor de un avión no puede pararse en caso de fallo, a menos que haya otro motor de reserva. Ciertos dispositivos de asistencia médica pueden *cerrarse* y avisar al usuario, aunque algunas veces entran en una condición de monitorización (cierre parcial). Cuando una persona entra en un área peligrosa, un sistema robótico debe parar la tarea que esté realizando antes de apagar, para proteger a las personas y equipos (parada de producción) [ANSI99].

Para cada sistema hay que determinar la política más adecuada, que puede ser alguna de las anteriores o una combinación de varias. También se deben garantizar mecanismos de diagnóstico y test para asegurar el funcionamiento del sistema. Se debe asegurar la mantenibilidad del sistema pudiendo sustituir partes del mismo sin tener que modificar el resto.

5.4.1.6. Aspectos de la Usabilidad

La usabilidad es un atributo difícil de caracterizar incluso en sistemas concretos, pues depende de muchos factores, algunos de ellos ligados a la arquitectura y otros no. Por otro lado, la casuística que puede generarse a través de los casos de uso y los escenarios es infinita. Por ello, en este caso, en lugar de presentar plantillas, es más conveniente destacar aquellos atributos arquitectónicos que tienen una incidencia directa para conseguir que un sistema concreto sea cómodo y fácil de usar. Entre ellos cabe destacar:

- Adaptabilidad a distintas interfaces de usuario. Cada una de ellas con las características más apropiadas para el tipo de operador y el tipo de trabajo a realizar.
- Botón o botones de parada segura fácilmente accesibles e identificables.
- Posibilidad de integrar servicios y utilidades que asistan al operador en el entrenamiento y ejecución de sus trabajos.

Todos estos aspectos han sido ya tratados en el apartado correspondiente, por lo que no se insistirá más sobre ellos. Sí se resalta aquí el hecho de que la unidad de control debe garantizar estos aspectos, principalmente a nivel de respuesta temporal, integración de interfaces, accesibilidad a variables que se quieran monitorizar, etc.

5.4.1.7. Aspectos de la interoperabilidad

Como ya se ha comentado, la interoperabilidad define la capacidad de un sistema para trabajar con otros sistemas.

El mantenimiento de los cascos de embarcaciones incluye diversas tareas, incluyendo reparaciones, soldaduras, la eliminación de la pintura vieja y el repintado. Algunas de estas operaciones presentan riesgos para los operarios, ya que se realizan sobre andamios y en una atmósfera plagada de polvo y emanaciones tóxicas.

Las operaciones de eliminación de pintura y repintado son relativamente sencillas y pueden realizarse de forma semiautomática, mediante mecanismos muy económicos, requiriéndose la intervención del operador sólo para iniciar los trabajos y en ciertas situaciones muy específicas. Ahora bien, los cascos de ciertos buques son muy grandes y es difícil que un solo robot se adapte a todos los contornos. Por otro lado, esas mismas dimensiones hacen posible simultanear operaciones de varios mecanismos siempre que se tomen ciertas precauciones. Se trata básicamente de:

- *Poner a varios mecanismos a eliminar pintura de forma simultánea sobre diferentes paños.*
- *Poner a otros mecanismos a repintar los paños tan pronto como los mecanismos de eliminación de pintura se hayan alejado lo suficiente.*
- *Sincronizar las tareas de todos los mecanismos de modo que no se solapen los paños y que el repintado no empiece hasta que los mecanismos de eliminación de pintura se encuentren lo bastante lejos.*

La automatización completa de las operaciones no es económicamente viable (tal vez tampoco técnicamente), pero el papel del operario puede reducirse a:

- *Introducir las misiones y repartir el trabajo.*
- *Iniciar el proceso.*
- *Resolver situaciones de bloqueo y atender a las alarmas.*
- *Reorganizar las misiones.*

Una vez introducidas las misiones y repartido el trabajo, los mecanismos pueden llevarlas a cabo con un alto grado de autonomía. Las misiones que introduce el operador son de muy alto nivel. No se trata de ordenarle a un mecanismo que se mueva de un punto a otro o de activar una herramienta, sino de que ejecute una misión compleja en la que pueden producirse muchos eventos a los que el sistema debe responder por sí mismo. Es posible implementar incluso un reparto dinámico del trabajo, de forma que cada mecanismo tengan en cuenta el trabajo ya realizado por los demás.

Fig. 5.3.- Características del Sistema GOYA influyentes en la Interoperabilidad (extracto de [GOYA98a])

La interoperabilidad entre sistemas heterógeneos es un atributo muy díficil de conseguir. Sus misiones pueden ser diferentes y su interpretación de la información puede no tener nada que ver. Sin embargo, en el caso que nos ocupa, la interoperabilidad se plantea entre sistemas que comparten la *misma* arquitectura, por tanto:

- Todos los sistemas comparten un mismo propósito. Aunque cada sistema pueda estar especializado en la realización de un pequeño conjunto de actividades, todos ellos están pensados para permitir el control a distancia de una serie de mecanismos.
- Las interfaces de acceso a los diferentes sistemas son (deben ser) bastante homogéneas, pues todas responden a las reglas de interacción definidas en la arquitectura.
- Todos los sistemas manejan información que a un cierto nivel de abstracción puede considerarse equivalente y que interpretan de formas similares (alarmas, eventos, estado de los dispositivos, comandos básicos comunes, etc.).

Además, el nivel de colaboración que se plantea es limitado:

- Los sistemas mantienen un alto grado de independencia. Aunque todos ellos colaboran en una misión más amplia, las responsabilidades de cada sistema están perfectamente definidas y desde su perspectiva no necesitan saber si la información que les llega procede de otro sistema.
- No es necesario un gran intercambio de información entre los mismos, aunque sí el paso de ciertos mensajes. Los estrictamente necesarios para coordinar sus acciones.

Con todo ello, la interoperabilidad se plantea como la posibilidad de descomponer una misión en misiones más simples asignando cada una de ellas a un sistema de teleoperación determinado y realizándose todas ellas de forma simultánea y coordinada. La **Fig. 5.3** describe un ejemplo del tipo de interoperabilidad que se persigue.

En la tabla siguiente se resumen los principales aspectos de los atributos de calidad abstractos mencionados en los anteriores párrafos. Para consultar de forma más detallada estos aspectos, se remite al lector al Anexo II donde se adjuntan las plantillas de atributos de calidad para ACROSET.

Sobre todo la modificabilidad y el rendimiento están muy relacionados y generalmente de forma opuesta, esto es, el mayor cumplimiento de una de estas directrices haga que sea difícil llegar a cumplir la otra. Evidentemente habrá que llegar a un compromiso, a un acuerdo entre los *stakeholders* que influyen en la arquitectura para cumplir en la medida de lo posible uno de ellos sin perjudicar al otro.

Atributos de Calidad Abstractos

1. Caracterización del Negocio y Objetivos de la Organización

1.1. Desarrollo rápido de aplicaciones:
- 1.1.1. Posibilidad de reutilización de componentes.
- 1.1.2. Modularidad en el diseño de los sistemas.

1.2. Coste competitivo de los productos desarrollados:
- 1.2.1. Posibilidad de integrar componentes COTS.

2. Caracterización de la Modificabilidad

2.1. Adaptabilidad a diferentes entornos de operación
- 2.1.1. Cambiar un sistema incorporando nuevos servicios para adoptarse a distintos entornos.
- 2.1.2. Un mismo sistema adaptable a distintos entornos.
- 2.1.3. Adaptabilidad a entornos estructurados y no estructurados

2.2. Modificaciones relacionadas con cambios en los mecanismos y en las misiones.
- 2.2.1. Adición/Eliminación/Modificación de mecanismos (brazos, vehículos, herramientas).
- 2.2.2. Cambios en la estructura y la cinemática de los mecanismos.
- 2.2.3. Adición/Eliminación/Modificación de controladores de mecanismos.
  - *2.2.3.1. Adición/Eliminación/Modificación de comandos, alarmas e información de estado.*
  - *2.2.3.2. Cambios en la máquina de estados de los mecanismos.*
  - *2.2.3.3. Adición/Eliminación/Modificación de Eventos.*
- 2.2.4. Adición/Eliminación de Misiones.
- 2.2.5. Cambios en las misiones.
  - *2.2.5.1. Modificación de los pasos de las misiones (Adición/Elimin./Modificación de paso, cambio de orden).*
  - *2.2.5.2. Modificación de la sincronización (Adición/Elimin./Modificación de eventos o señales de sincronismo).*
- 2.2.6. Dada una combinación vehículo-brazo-herramienta: Cambiar el vehículo, el brazo o la herramienta.
- 2.2.7. Adaptabilidad a cambios en los requisitos de rendimiento.
  - *2.2.7.1. Cambios en el tiempo de ejecución, plazos y períodos de las tareas.*
  - *2.2.7.2. Cambios en el número de recursos requeridos por las tareas.*
  - *2.2.7.3. Cambios en la capacidad de los enlaces de comunicaciones y en las plataformas de ejecución.*

2.3. Portabilidad:
- 2.3.1. Posibilidad de cambio de plataforma y sistema operativo.
- 2.3.2. Posibilidad de cambio de enlaces y protocolos de comunicación.

2.4. Capacidad de distribución:
- 2.4.1. Posibilidad de migrar procesos o subsistemas de un nodo a otro (en tiempos de compilación o carga.

2.5. Integración de componentes comerciales
- 2.5.1. Posibilidad de utilizar componentes, librerías o herramientas *software* comerciales
- 2.5.2. Posibilidad de utilizar componentes *hardware* comerciales.

2.6. Integración de Servicios y Utilidades:
- 2.6.1. Posibilidad de incluir/excluir servicios especiales (gráficos, simulaciones, cálculos cinemáticos, visión artificial, etc).
- 2.6.2. Posibilidad de incluir sistemas *hardware*/*software* comerciales que integren nuevos servicios.

2.7. Adaptabilidad a distintas interfaces de usuario:
- 2.7.1. Posibilidad de recibir órdenes y proporcionar información a cualquier interfaz que interaccione con el sistema.

2.8. Gestión de distintos modos de control y de operación:
- 2.8.1. Garantía del paso de un modo de operación a otro.
- 2.8.2. Posibilidad de operar según distintos modos de control.

Atributos de Calidad Abstractos (Continuación)

3. Caracterización del Rendimiento

3.1. Rendimiento de las Tareas Críticas:
- 3.1.1. Las tareas relacionadas con el control directo sobre los mecanismos deben realizarse dentro de unos plazos estrictos.

3.2. Rendimiento de las Tareas No-Críticas:
- 3.2.1. Los procesos de control de alto nivel no relacionados directamente con el funcionamiento seguro del sistema pueden retrasar sus tiempos de respuesta siendo el funcionamiento del sistema correcto.

3.3. Rendimiento de las comunicaciones
- 3.3.1. Los comandos deben entregarse a la unidad de control local dentro de unos plazos determinados.
- 3.3.2. El estado de los mecanismos debe llegar al sistema de teleoperación dentro de unos plazos determinados.

4. Caracterización de la Seguridad

4.1. El sistema de teleoperación debe evitar situaciones que pongan en peligro a personas y bienes.

4.2. Supervisión del estado del sistema y de la ejecución de los comandos:
- 4.2.1. El operador debe ser informado dentro de unos plazos de cualquier mal funcionamiento del sistema. (Detección de errores graves y reporte automático al operador según políticas de información que razonablemente aseguren que éste no va a ignorar las alarmas)
- 4.2.2. Se debe activar automáticamente una parada segura si se detectan condiciones que pueden poner en peligro a bienes o personas.
- 4.2.3. Deben existir medios independientes para que el operador active una parada segura, fácilmente accesibles e identificables.
- 4.2.4. Deben establecerse mecanismos para evitar situaciones de riesgo, en especial las relacionadas con el acceso a las zonas de operación, las fuentes de alimentación (pérdida o sobrecarga), la sobrecarga de motores y el fallo de los enlaces de comunicación.

5. Caracterización de la Fiabilidad/Disponibilidad

5.1. Comunicaciones Fiables
- 5.1.1. Los enlaces de comunicación deben ser fiables.
- 5.1.2. Los enlaces deben ser recuperables.

5.2. Modos de operación degradados
- 5.2.1. En caso de fallo debe habilitarse un modo de operación degradado que permita el acceso a los comandos más básicos.
- 5.2.2. Deben existir algún mecanismo para efectuar una parada segura en caso de fallo de cualquiera de los subsistemas.

5.3. Estado de fallo-seguro. (Definición dependendiente de la aplicación).
- 5.3.1. En caso de fallo grave de los sistemas local y de teleoperación deben pasar a un estado conocido y seguro, según los sistemas, esto puede suponer:
  - 5.3.1.1. Parada de emergencia. (En todos los sistemas).
  - 5.3.1.2. Parada de protección. (En muchos sistemas).
- 5.3.2. En caso de fallo menos grave del sistema de teleoperación el sistema debe pasar a un estado conocido y seguro, según los sistemas esto puede suponer.
  - 5.3.2.1. Parada de emergencia. (En los sistemas más simples)
  - 5.3.2.2. Parada de protección.
  - 5.3.2.3. Paso a modo degradado o manual.
- 5.3.3. Evitar modos comunes (evitar que los fallos se propaguen por el sistema afectando a los módulos encargados de la supervisión de la seguridad).

5.4. Parada Segura siempre disponible.
- 5.4.1. En el peor de los casos debe existir una forma de efectuar una parada segura.

Atributos de Calidad Abstractos (Continuación)	
6.	Caracterización de la Usabilidad
6.1.	Posibilidad de fácil acceso de distintas interfaces de usuario a la funcionalidad ofrecida por la unidad de control.
6.2.	Posibilidad de fácil acceso de otros sistemas usuarios de la unidad de control
6.3.	Presencia de botones de parada segura fácilmente accesibles e identificables
6.4.	Posibilidad de integrar servicios y utilidades que asistan al operador en el entrenamiento y ejecución de sus trabajos
7.	Caracterización de la Interoperabilidad
7.1.	Descomponer una misión en misiones más simples asignando cada una de ellas a un sistema de teleoperación determinado y realizándose todas ellas de forma simultánea y coordinada.

Tabla 5.3.- Atributos de calidad abstractos

5.5. Requisitos funcionales abstractos

Los requisitos funcionales suelen estar más ligados a los sistemas que a la arquitectura, puesto que expresan cual debe ser la funcionalidad que se le exige a un sistema concreto. No obstante, como los requisitos funcionales suelen ser muy numerosos, pueden ser organizados en diferentes niveles de abstracción, siendo concretados en los *casos de uso* de un sistema. Nótese, a modo de reflexión, que los métodos de diseño orientados a objetos, que siguen una proceso de diseño *bottom-up*, de lo particular a lo general, están guiados por *casos de uso*, es decir, por la funcionalidad del sistema, mientras que los métodos de diseño orientados a la arquitectura, como ABD o las 4-vistas de Hofmeister, que siguen un proceso de diseño *top-down*, están guiados por los requisitos de calidad de la arquitectura. Lo que sí debemos asegurar al diseñar la arquitectura es su predisposición a cumplir los requisitos funcionales (de hecho, al plantear las directrices de la misma, se están mezclando requisitos funcionales, de calidad y de negocio y organización).

En consecuencia, se proponen en este apartado una lista inicial de requisitos funcionales abstractos que pueden acomodar la mayoría de las funcionalidades del dominio y que deberán ser concretados en los casos de uso correspondientes al ser implementado un sistema concreto.

En una fase posterior, entrando ya en el diseño de la arquitectura como propone ABD (ver capítulo 6), los requisitos funcionales para un elemento de diseño (qué hacer), serán traducidaos a resposabilidades del elemento de diseño (cómo hacerlo). Las responsabilidades de un elemento de diseño serán redefinidas y organizadas en grupos que pueden representar distintos elementos dentro de la arquitectura, lo cual provocará la aparición de los subsistemas fundamentales de la arquitectura.

Requisitos Funcionales Abstractos	
1. Modos de Operación	1. Operacional: El operador gobierna los mecanismos. El número de modos de operación presentes depende del tipo de sistema. Los modos deben ser cambiados mediante una acción explícita del operador si tiene los permisos de acceso adecuados y puede llegar a la UdC[10] por cualquiera de las interfaces habilitadas. 1.1. Modo Teleoperado. 1.1.1. Seguro (sólo están disponibles los comandos seguros) 1.1.2. Normal (acceso a todos los comandos) 1.1.3. Asistido (ciertos comandos del operador pueden ser rectificados por el robot) 1.2. Modo Autónomo o semi-autónomo. 1.2.1. Autónomo (el robot se comporta de forma autónoma según una misión programada) 1.2.2. Supervisado (el comportamiento autónomo del robot puede ser corregido por el operador) 2. Mantenimiento: Se accede a la UdC para cambiar parámetros, configurar o probar el sistema 2.1. Modo de Programación (Definición y registro de misiones) 2.2. Modo de Configuración 2.2.1. Calibración 2.2.2. Cambio de parámetros de configuración del sistema 2.3. Modo de Test (Prueba y diagnóstico de los elementos y funcionalidades del sistema) 3. Simulación: El operador interacciona con mecanismos simulados. La UdC puede estar presente en mayor o menor medida en esta simulación, pero no debe accionar mecanismos reales. (En sistemas más complejos y en aquellos que integran servicios de simulación y entrenamiento).
2. Chequeo de Viabilidad de los comandos de operador	1. Chequeo del rango de los parámetros. 2. En modo normal sólo están disponibles los comandos seguros. Un comando es seguro si está permitido en el estado operacional actual y su ejecución deja al sistema en un estado conocido y seguro.
3. Control de Mecanismos (vehículo, robot, manipulador) y herramientas.	1. Comando de movimiento y operación 2.1. Movimientos tipo Jog (indicación del grado de movimiento de cada eje) 2.2. Movimientos tipo Joy (desplazamiento hacia una zona determinada del espacio) 2.3. Movimientos coordinados de ejes (seguimiento de trayectorias, etc) 2.4. Secuencias de movimiento 2.5. Movimientos especiales (aplicar fuerzas, avanzar hasta tope mecánico, seguimiento maestro-esclavo, etc) 2.6. Movimientos de manipuladores en el mismo sentido de los puntos anteriores dependiendo de la complejidad de los mismos 2.7. Accionar, activar/desactivar herramientas 2. Configuración de movimiento y operación 2.8. Habilitar/deshabilitar ejes, articulaciones, manipuladores, herramientas, vehículos, etc. 2.9. Cambiar parámetros del movimiento (velocidad, aceleración, par, etc) 2.10. Cambiar tipo de movimiento (coordinado, lineal, de un eje, etc.) 2.11. Cambiar parámetros de actuación de la herramienta. 3. Configuración del control del movimiento y operación 2.12. Comandos de calibración 2.13. Comandos de configuración de los controladores (sintonización, parámetros del control, etc) 2.14. Comandos de seguridad (parada de emergencia y cualquier otro comando que siga normativas de seguridad) 2.15. Comandos especiales (arranque, parada, rearranque tras parada, parada segura, etc) 4. Realización de movimientos y operaciones 2.16. Los movimientos deben realizarse con la precisión y velocidad requeridas. 2.17. Detección de colisiones (en todos los sistemas). 2.18. Prevención de colisiones (en algunos sistemas). 2.19. Condiciones de seguridad que se vean afectadas por la ejecución de los comandos.
4. Coordinación y sincronización de mecanismos	1. Debe ser posible programar y realizar secuencias de operaciones (en la mayoría de los sistemas) y automatizar ciertas acciones y misiones (en algunos sistemas), siendo posible incluso añadir ciertas acciones autónomas sencillas. 2. Si el sistema incluye más de un mecanismo debe ser posible coordinar y sincronizar sus acciones para la realización de misiones que requieran su funcionamiento simultáneo, incluyendo la coordinación de los mecanismos con la herramienta (en algunos sistemas) 3. En los sistemas más simples el operador puede encargarse de la coordinación entre mecanismos, en los más complejos, la coordinación debe ser automática, de acuerdo con el desarrollo de la misión en curso.

[10] UdC = Unidad de Control

Requisitos Funcionales Abstractos (Continuación)	
5. Acceso y notificación del estado de los mecanismos y el entorno	1. La UdC debe recopilar y proporcinar información fiable de: 1.1. El estado de los mecanismos (posición, estado activación, alarmas, etc,..) 1.2. El estado de los sensores y actuadores del sistema 1.3. El estado del entorno 2. Debe garantizar mecanismos de acceso seguro a la información (multiples lectores-escritores, requisitos de tiempo-real, etc) 3. El operador debe ser informado de cualquier mal funcionamiento, alarma, etc. 4. Deben garantizarse mecanismos de recuperación de información y estado ante paradas intermedias, parada segura, intercambio de herramientas, etc.
6. Acceso a los mecanismos	1. Acceso remoto a los mecanismos. 2. Enlaces de comunicaciones eficientes y fiables. (Algunos sistemas pueden requerir protocolos de comunicaciones deterministas). 3. Si el sistema incluye más de un mecanismo (p.e: vehículo-brazo-herramienta), cada mecanismo debe poder ser accedido y gobernado individualmente.
7. Servicios de Configuración	5. El sistema debe poder configurarse de acuerdo a: 1.4. Los mecanismos a controlar (diferentes tipos de brazos, vehículos y herramientas). 1.5. Cambios en elementos (posicionador, manipulador, motor, tarjeta de control, herramienta, etc). 1.6. Tipos de control que tengan estos mecanismos (PID, fuzzy, todo/nada, etc) 1.7. La misión o misiones a realizar (con o sin cambio de herramienta) 1.8. El entorno de trabajo (Especialmente importante en aquellos sistemas que usan representaciones gráficas del entorno y los mecanismos y realizan simulaciones de las misiones Distintas configuraciones cinemáticas) 1. Algunas configuraciones se podrán hacer en tiempo de carga. Otras en tiempo de ejecución.
8. Servicios Especiales	Dependiendo de dónde se ponga la línea divisoria entre las funcionalidades del sistema de teleoperación y la UdC, podremos encontrar en esta última, algunas funcionalidades que en otros sistemas podríamos encontrar en la estación de teleoperación, como las siguientes: 1. Simulación de misiones (en algunos sistemas). 2. Cálculos cinemáticos (en muchos sistemas) 3. Cálculo de trayectorias y detección de colisiones (En algunos sistemas) 4. Capacidades de visión artificial y sistemas de navegación (En algunos sistemas).
9. Interfaz de Usuario	De todas las funcionalidades definidas en esta tabla, para cada implementación concreta en un sistema, según los casos de uso del mismo, habrá que seleccionar: 1. Cómo envía la UdC el estado, con qué periodicidad, y qué información debe ofrecer 2. Cómo se capturan las órdenes del operador (tipos de interfaces muy diversos: joysticks, botoneras, paneles de operador, estaciones de teleoperación, etc) y recibe la UdC los comandos adecuados y qué comandos pueden ser. 3. Cómo se gestiona la información de sistemas de realimentación sensorial especiales (realimentación de fuerzas, sistemas de visión estereoscópica, etc). 4. Cómo se interactúan con la gestión de alarmas, advertencias, errores, etc 5. Se deben definir mecanismos de seguridad y emergencia según normas. 6. Qué otros sistemas interaccionan con la UdC y qué tipo de información se intercambian..

Tabla 5.4. – Requisitos funcionales abstractos

5.6. Restricciones

Las restricciones son decisiones de diseño preestablecidas que limitan los requisitos de diseño e implementación, como paso previo al desarrollo de modelos o arquitecturas. La mayor parte de estas decisiones vienen determinadas por los factores de negocio (inversiones previas en componentes *hardware* comerciales, interoperabilidad con sistemas ya existentes, uso de las plataformas disponibles, utilización de un determinado sistema operativo, etc), aunque en ocasiones responden a motivos técnicos (un arquitecto basándose en su experiencia puede imponer un sistema operativo determinado, aun cuando existan otros que ofrezcan servicios similares). Las únicas restricciones que se podrían definir en esta fase son las referentes a los requisitos temporales y de seguridad, ya que la arquitectura podría ser trasladada a cualquier plataforma, por lo que las limitaciones *hardware* y *software* vendrán dadas por la implementación de cada uno de los módulos.

El tema de seguridad y prevención de accidentes en los sistemas robotizados es un aspecto crítico, y desde luego, los requisitos temporales de un sistema teleoperado deben ser garantizado para asegurar que la información que recibe el operador es válida. El cumplimiento de los tiempos de respuesta de

los aspectos críticos inherentes de estos sistemas será determinante en los aspectos de seguridad necesarios en los mismos.

Puesto que en esta tesis se está proponiendo una arquitectura de referencia para un dominio, no es lógico que se pongan muchas restricciones de diseño. En todo caso, aparecerán cuando se utilice dicha arquitectura para implementar un sistema concreto o una familia de sistemas.

Como ejemplo de las restricciones puestas a un sistema, se pueden mencionar algunas de las que condicionan el diseño del proyecto EFTCoR [EFTCoR02]:

- ✓ El sistema debe cumplir con las regulaciones de seguridad y medio ambiente impuestas por la Unión Europea.
- ✓ El sistema de reciclado sólo será satisfactorio si se puede reutilizar el 60 % de la granalla utilizada.
- ✓ Los diferentes astilleros donde sería utilizado el sistema, con sus entornos de operación distintos, condicionan el diseño para poder ser utilizado en todos ellos, como desean los socios del proyecto.
- ✓ Las condiciones de la zona donde debe operar el sistema (varios buques serán reparados a la vez en dique seco, hay muchos obstáculos) obligan a que el sistema sea fácilmente transportable de un sitio a otro y de pequeño tamaño.

5.7. Opciones y estrategias arquitectónicas

Para cada uno de los requisitos funcionales y de calidad existe un estilo o conjunto de estilos que permiten su cumplimiento. Algunos requisitos pueden admitir varias opciones, otros sólo una. Las opciones deben enumerarse para proceder más adelante a la selección de las más apropiadas. La enumeración de estas opciones es parte del proceso de diseño, sin embargo a menudo esta enumeración se realiza durante la fase de especificación de requisitos, constituyendo en este caso una entrada más del ABD.

Un *estilo arquitectónico* consiste en una colección de tipos de componentes junto con una descripción del patrón de interacción entre ellos [Bass98]. Ejemplos de tipos de componentes son cliente, servidor, capa, proceso, etc. Como tales, los tipos de componentes deben incorporar una funcionalidad suficiente para implementar los patrones de interacción, pero no hay funcionalidad de la aplicación asociada con ellos. Uno de los pasos más importantes en el método ABD es la elección de un estilo arquitectónico dominante para un conjunto particular de requisitos. Las directrices arquitectónicas permiten escoger un estilo arquitectónico gracias a la interacción de requisitos, funcionalidad y atributos de calidad.

Una directriz puede incluir cualquier combinación de requisitos funcionales, de calidad y de negocio. Por ejemplo, imaginemos que una unidad de control tiene la siguiente directriz arquitectónica:

"Procesar gran cantidad de datos con en condiciones de tiempo-real estricto"

Aquí, la parte funcional es el requisito de transportar grandes cantidades de datos; el aspecto de calidad de esta directriz es que debe cumplir ciertas prestaciones de tiempo-real. Un estilo arquitectónico resultante de esta directriz es la *adopción de una estrategia de planificación de tiempo-real.* Ahora bien, esta estrategia depende tanto de los aspectos funcionales como de calidad de la directriz arquitectónica. Esto es, si la cantidad de datos no fuera demasiado grande, la prestación de tiempo-real podría alcanzarse bajo unas disciplinas de planificación; por el contrario, si las prestaciones de tiempo real no fueran demasiado estrictas, se podrían adoptar otras disciplinas de planificación para movilizar la gran cantidad de datos. Es la combinación de las dos, lo que dirige la elección de un estilo determinado.

Este ejemplo llama la atención en otro aspecto: probablemente hasta que no se tengan los requisitos concretos (escenarios de calidad y casos de uso) de un sistema, no se podrán decidir algunos estilos

arquitectónicos tan específicos como el del ejemplo. Esto demuestra el interés por plantear la mayor cantidad posible de aspectos de **variabilidad** de los sistemas del dominio para una arquitectura de referencia.

5.7.1. Estilos arquitectónicos

Para cada requisito arquitectónico, se debe enumerar las posibles opciones arquitectónicas que satisfagan dicho requisito. Esta enumeración proviene de la consideración de patrones de diseño [Buschmann96], estilos arquitectónicos [Shaw96], algunas herramientas particulares (sockets, llamada a procedimiento remoto RPC, CORBA, etc) y la experiencia del arquitecto.

De la lista de posibles elecciones, se hará una selección para la que todos los requisitos de la arquitectura serían potencialmente satisfechos (en la mayoría de los casos se deben adoptar compromisos entre el cumplimiento de los requisitos porque muchos de ellos son contrarios entre sí).

En [Bass98] se encuentran algunas definiciones interesantes ya utilizadas en el Capítulo 2 al definir la arquitectura de un sistema. Además de esas definiciones, se encuentran algunos patrones muy utilizados en el diseño de arquitecturas, en concreto, se discuten los patrones del sistema (manifestados en **estilos arquitectónicos)** y patrones de diseño. El uso de estos patrones como ayuda en la propuesta de opciones arquitectónicas es muy interesante, porque ofrecen un paquete de decisiones de diseño que han sido ampliamente probadas en otras arquitecturas y que pueden solucionar problemas comunes.

Algunos patrones arquitectónicos típicos, organizados según relaciones *es-un* son:

Componentes independientes		
Procesos comunicándose	Sistemas de eventos	
	Invocación implícita	Invocación explícita

Flujo de datos	
Secuencial por lotes	Tuberías y filtros

Centrado en datos	
Repositorio	Pizarra

Máquina virtual	
Intérprete	Sistema basado en reglas

Llamar y devolver		
Programa principal y subrutinas	Orientación a objetos	Por capas

En [Shaw96] se puede encontrar una clasificación de estilos arquitectónicos basados en las características de los mismos.

5.7.2. Patrones y estrategias arquitectónicas

Las opciones arquitectónicas no se basan sólo en patrones arquitectónicos, son también estrategias posibles ante requisitos establecidos (ver Anexo II). Dichas estrategias pueden ser de este estilo:

- Adoptar arquitecturas por capas
- Encapsular componentes e incluir módulos de desacoplo
- Separar de conceptos según dimensiones de interés

- Usar standards
- Separar según su comportamiento de tiempo-real en componentes críticos de componentes no-críticos.
- Usar RMA para predecir las prestaciones del sistema
- Introducir modos de recuperación y gestión de errores.
- Utilizar patrones de comunicación que favorezcan la distribución en el despliegue
- Utilizar mecanismos de abstracción, herencia y composición

En el siguiente capítulo, donde se plantea el modelo conceptual de ACROSET, se facilita un cuadro con las distintas opciones arquitectónicas establecidas para cada requisito y se justifica la elección de algunas de ellas para proponer la arquitectura de referencia.

5.7.2.1. Arquitecturas por capas y encapsulación

La adopción de una arquitectura por capas es un método de reducción de la complejidad de un sistema probado ampliamente a lo largo del tiempo en numerosos sistemas. Por ejemplo, se hace encapsulando componentes externos (como *software* o *hardware* COTS) o separando servicios proporcionados por sistemas por medio de *software* de interfaz de usuario. Pueden ser usadas para proporcionar soporte a la reutilización, asignando servicios comunes a una capa de servicios de aplicación. Las capas también son útiles proporcionando independencia entre partes del sistema, de forma que, por ejemplo, un cambio en el sistema operativo, no afecte al sistema completo.

Existen dos clases de arquitecturas por capas según la posibilidad de acceso a capas inferiores: si existe la posibilidad de acceso a las capas inferiores desde *cualquier* capa superior, es una arquitectura ***open-layered***. Las arquitecturas por múltiples capas favorecen la reutilización gracias a su modularidad, en las ***close-layered*** los componentes de una capa sólo son visibles para los componentes de la misma capa o por los de la capa *inmediatamente superior*. En las *open-layered* sus componentes son accesibles desde cualquier capa superior. De esta forma se favorece el rendimiento, a costa de complicar la coordinación entre componentes. La reutilización sigue estando garantizada.

Encapsular componentes

La estrategia arquitectónica de encapsular componentes está muy relacionada con las arquitecturas por capas, existen diversos enfoques para encapsular componentes que se pueden tener en cuenta:

Encapsular el *hardware* específico de dominio. Puesto que en un sistema habrá componentes que interaccionen con el *hardware*, es interesante crear un componente específico responsable de la comunicación con dicho *hardware*. Esta estrategia ayuda a aislar al resto de los componentes del sistema cuando el *hardware* cambia. El sistema interacciona con el *hardware* específico del dominio, por tanto, debería ser suficientemente flexible para permitir que nuevos modelos de *hardware* del dominio (nuevos controladores, versiones más avanzadas de tarjetas, etc.) puedan introducirse y se puedan especificar nuevas configuraciones *hardware*. Usar un componente de abstracción del *hardware* específico del dominio minimiza los efectos de cambios en dicho *hardware*.

Encapsular el *hardware* de propósito general. Encapsular el *hardware* del sistema permite que se puedan hacer cambios en el mismo con poco o ningún impacto sobre las aplicaciones. A la inversa, esta estrategia facilitará la introducción de nuevas características a la aplicación sin requerir modificaciones en el *software* que maneja el *hardware*. Esta estrategia favorece la **independencia del sistema operativo y de la plataforma** de implementación aplicada junto a la utilización de una arquitectura por capas que favorezca la portabilidad y se separa el código dependiente de la plataforma. Básicamente consistirá en aislar las llamadas al sistema operativo en una capa que ofrezca una interfaz común al sistema, independientemente del sistema operativo que se utilice.

Los componentes COTS, como puede ser el propio sistema operativo, tienen un gran impacto en un significativo número de componentes. Para reducir el esfuerzo y el tiempo necesario en adaptar el sistema cuando estos cambio se producen, es conveniente:

- ✓ **Usar estándares**. El uso de estándares cuando sea posible reduce el impacto de los cambios en la tecnología del *software*. Usar interfaces de SO estándar, como POSIX, facilita la portabilidad a otros sistemas operativos en el futuro.
- ✓ **Desarrollar interfaces específicos del producto a los componentes externos.** Cuando los estándares son inestables o ausentes, se deben crear estándares internos. Desarrollar interfaces específicos del producto reduce las dependencias de los componentes externos o estándares inestables.
- ✓ **Encapsular la comunicación con los dispositivos.** Crear una interfaz de manejador de dispositivos, que consiste en desarrollar una interfaz POSIX estándar a todos los dispositivos. Se crea un manejador de dispositivos virtual que implementa esta interfaz entre los manejadores de dispositivos y las aplicaciones. Proporciona acceso a los dispositivos existentes en el mismo procesador o en procesadores distintos que la aplicación que los utiliza. También proporciona un servicio que puede notivicar a los usuarios de los dispositivos asíncronamente de su disponibilidad. [Hofmeister00]

5.7.2.2. Separación de conceptos según dimensiones de interés

Esta estrategia, u opción arquitectónica puede que sea la más importante y por tanto merece una mención especial. Está relacionada con muchos requisitos de calidad, en concreto, en [Hofmeister99] aparece relacionada con la necesidad de *fácil adición y eliminación de características.* Sobre todo favorece a los requisitos asociados a la modificabilidad, teniendo un impacto positivo en la reutilización y el desarrollo rápido de nuevas aplicaciones, en la mantenibilidad del sistema, etc.

Es importante usar esta estrategia para incorporar flexibilidad ante cambios posteriores en la vista de módulos. Sugiere separar o descomponer los componentes y módulos según sus dimensiones de interés, entendiendo como tales, la dedicación al procesamiento, comunicación, control, datos e interfaz de usuario, seguridad, etc. Por ejemplo, si se diseña un proceso de adquisición, sería muy útil separar los aspectos de procesamiento de los aspectos de control. Esto proporciona la posibilidad de usar los módulos de procesamiento en otros procesos de adquisición o en otros contextos. La aplicación de esta estrategia también favorecerá la localización y trazabilidad de los requisitos respecto a los elementos de diseño. Cuando los requisitos cambien, será posible reutilizar la estructura existente, cambiando sólo los componentes afectados, o añadiendo nuevos componentes para llegar a una nueva solución de manera más rápida. Esta estrategia arquitectónica se especializa en varios aspectos que influyen directamente en la propuesta de los componentes de la arquitectura ACROSET, como se verá en el capítulo siguiente:

Separar el control de la inteligencia. Adoptando esta división se pueden modificar las estrategias de planificación, o añadir nuevos comportamientos autónomos independientemente del control de los dispositivos.

Separar el control de los datos. Principalmente se separa la información del estado de componentes respecto al control de los mismos. De esta forma, la información puede ser actualizada de forma independiente, siguiendo patrones de actualización distintos de los que tiene el control de los mismos, incluido la aparición de múltiples lectores suscritos a dicha información, etc.

Separar algoritmos y procedimientos de adquisición. Para facilitar el intercambio y la adición de algoritmos de control, procesamiento, adquisición, etc., es conveniente separar los algoritmos del control propiamente dicho. El conocido patrón política o estrategia es utilizado para este fin.

Separar la configuración del sistema respecto a su funcionamiento. Cualquiera de los componentes conceptuales de la arquitectura pueden tener parámetros de configuración, incluidos distintos algoritmos de control, estrategias de coordinación, algoritmos de inteligencia, parámetros de

movimiento, incluso la aparición de componentes o la desactivación de otros, que pueden ser configurados.

Separación de la interacción con los usuarios. Para evitar que la estructura de un sistema se vea influenciado por el tipo de usuario que se conecte al mismo, debe aislarse la interacción con los usuarios (recordar que no necesariamente tiene que ser un usuario humano y que pueden presentarse varios usuarios a la vez). Además de las típicas funciones de interfaz del sistema, debe realizar la gestión de comandos y dirigir la orden al componente encargado de procesarlo, según el tipo de comando que se trate.

Separar el control de las medidas de seguridad para ese control (diagnóstico, gestión de errores) Esta importante directriz permite que el sistema funcione de manera segura, permitiendo que las políticas de seguridad puedan ser intercambiadas y variables, sin variar la estructura del resto del sistema [Selic96]. Se refiere a aquellos aspectos "secundarios" respecto a la funcionalidad principal de un sistema. Esas actividades secundarias que define como "control" se pueden concretar entre otras, en las siguientes actividades y mecanismos:

- Activación (*start up)* y desactivación del sistema
- Detección de fallos y recuperación ante los mismos
- Mantenimiento preventivo
- Monitorización de prestaciones y agrupación de estadísticas
- Sincronización con los sistemas de control externos
- Instalación on-line de nuevo *software* y *hardware*

En [Selic96] se justifica la separación entre estas tareas de control de la funcionalidad. Es deseable mantener el protocolo de control tan separado como sea posible del protocolo funcional de forma que puedan evolucionar independientemente. Para alentar esta separación, es útil distinguir el interfaz de control de un objeto o componente de sus otros interfaces funcionales. También es deseable separar las políticas de control de los mecanismos de control, puesto que las políticas de control suelen cambiar Las políticas de control son realizadas por el *software* que toma las decisiones basado en la realimentación y que ejecuta comandos que aseguren esas decisiones. Los mecanismos de control son los componentes que proporcionan la realimentación y que responden a los comandos.

Por ejemplo, en un sistema de comunicación, un componente que detecta el fallo de un enlace de comunicaciones es parte del mecanismo de control. Sin embargo, un componente que realiza el procedimiento de recuperación del fallo podría ser parte de las políticas de control. Separando claramente los dos, es posible cambiar la política de recuperación sin afectar al mecanismo de detección de fallos.

5.7.2.3. Estrategias relacionadas con el rendimiento

Separar componentes de tiempo-real críticos de los no críticos. Respecto a la preocupación por los requisitos de **tiempo-real críticos** influyentes en el rendimiento del sistema, hay una estrategia muy importante que facilita incluso la distribución en distintos procesadores en el posterior despliegue e implementación, y es la separación de los componentes que tienen necesidades de tiempo real críticas respecto a los que probablemente tengan necesidades no-críticas.

Usar RMA. También se debe usar RMA (*Rate Monotonic Analysis)* para predecir las prestaciones del sistema y verificar de antemano si el sistema es planificable.

No compartir memoria global directamente. Otra estrategia a tener en cuenta para favorecer el redimiento consiste en no permitir que las aplicaciones compartan memoria global directamente. Toda la información compartida entre aplicaciones debería ser compartida mediante mensajes entre procesos

o mediante un manejador de datos. Las respuestas a los requerimientos del manejador de datos son asíncronas para prevenir bloqueos entre aplicaciones.

Utilizar mecanismos de sincronización entre tareas. Fundamentalmente para evitar bloqueos. Algunos lenguajes, como Ada95 tienen estos mecanismos incluidos en el propio lenguaje, e incorporan conceptos como los objetos protegidos para prevenir interbloqueos.

5.7.2.4. Estrategias relacionadas con el desarrollo de los sistemas

En muchos sistemas aparecen requisitos relacionados con el plan de desarrollo de los sistemas, los costes y las personas que los realizan. En la mayoría de ocasiones se busca minimizar los costes de desarrollo, e incluso que el producto final tenga un precio competitivo en el mercado, para ello, se proponen las siguientes estrategias [Hofmeister00]:

- ✓ Reusar componentes específicos de dominio que ya existan en la organización
- ✓ Comprar antes que construir. Esta estrategia favorece el cumplimiento de una planificación *agresiva*.
- ✓ Hacer que sea fácil añadir nuevas características al sistema. Esto se consigue utilizando las opciones arquitectónicas de encapsulación y separación de conceptos mencionadas anterioremente. La programación orientada a aspectos para implementar favorece la utilización de esta estrategia.

5.7.2.5. Opciones arquitectónicas para seguridad y tolerancia a fallos

Estas opciones arquitectónicas están muy relacionadas con la última estrategia comentada en la separación de conceptos: "*Separar el control de las medidas de seguridad para ese control*". Hay varios patrones arquitectónicos ampliamente probados para conseguir funcionamientos seguros para este tipo de sistemas, que se pueden consultar en [Douglass00], [Buschmann96]. Se pueden resumir en los siguientes:

Redundancia homogénea. Usa canales idénticos para incrementar la fiabilidad. Todos los canales redundantes se ejecutan en paralelo y sus salidas son comparadas. Detecta fallos, pero no errores. Incrementa considerablemente los costes.

Redundancia diversa. Proporciona canales redundantes que se implementan por medios distintos.

Monitor-Actuador. El canal actuador lleva a cabo las acciones y el canal monitor compreba lo que se supone que debe estar haciendo el actuador, para ello, monitoriza el entorno físico para asegurarse de que los resultados del actuador son apropiados. Los sensores usados por los actuadores deben ser distintos que los que usan los canales de monitorización

***Watchdog* (perro guardián)**. Es un concepto común en los sistemas empotrados de tiempo-real. Un *watchdog* es un componente que recibe mensajes de otros componentes de forma periódica según una secuencia definida. Si falla esta secuencia, el *watchdog* inicia alguna acción correctiva (*reset*, *shutdown*, alarma, etc.) Suelen ser simples y a menudo se implementa con soporte *software* para prevenirlos de fallos en el *hardware*.

Ejecutivo de seguridad. Se usa un coordinador de seguridad centralizado encargado de monitorizar la seguridad y controlar la recuperación del sistema ante fallos. Actúa como un *watchdog* muy inteligente que traza y coordina toda la monitorización y acción de seguridad, de hecho, entre otras señales recoge los *time-outs* de distintos *watchdogs* que puede haber en el sistema. Un ejecutivo de seguridad proporciona un punto consistente y centralizado para procesamiento de seguridad, lo cual simplifica el *software*, que de otro modo estaría más enmarañado con comprobaciones que obscurecen el propósito principal de la aplicación.

Excepciones. Son un tipo especial de señales que se envían en un nivel muy próximo a la implementación, en la programación, porque corresponden a errores de rango, fallos de lectura,

overflow, etc., que se ha de prever y añadir al programa mecanismos de gestión de excepciones, pero está implícito en el código, no es un componente de diseño conceptual – estaría en el diseño detallado.

En [Hofmeister00] se pueden encontrar algunas estrategias adicionales, como:

- ✓ Introducir un modo de operación de recuperación
- ✓ Hacer que todos los datos en el punto de recuperación sean persistentes y accesibles.
- ✓ Definir políticas de gestión de errores y excepciones.
- ✓ Encapsular los componentes de diagnóstico.

5.7.3. Opciones arquitectónicas asociadas a la variabilidad

Las estrategias comentadas favorecen especialmente al tratamiento de la variabilidad en los sistemas del dominio. No obstante, existen trabajos relacionados específicamente con la gestión de la variabilidad de líneas de producto que se han adoptado para ACROSET.

Las investigaciones actuales sobre el modelado de la variabilidad de requisitos en líneas de producto muestran que no hay una notación estándar que explícitamente modele la variabilidad. En [Trigaux03] se describen y comparan diez propuestas para abordar la descripción de la variabilidad, de acuerdo a la notación, expresividad, aspectos en común y propósitos. Como resultado de este análisis, se propone una clasificación de dichas propuestas en dos categorías: las que extienden los casos de uso de UML y las orientadas a *aspectos*[11]. La segunda es más apropiada para nuestros propósitos puesto que es la utilizada por los expertos del dominio. Entre todas las notaciones, la más popularmente aceptada es la de Bosch [Bosch01], que se enmarca en la categoría de orientación a *aspectos*.

Resulta de especial interés para este apartado de opciones arquitectónicas, los mecanismos que ofrece Bosch en el mismo estudio para abordar la variabilidad en el contexto de las líneas de producto *software*. Estos mecanismos ayudan a los diseñadores a incorporar los requisitos de variabilidad en etapas tempranas del desarrollo. Entre los mecanismos de variabilidad ofrecidos, caben destacar los siguientes, porque tienen aplicación directa en ACROSET:

- El mecanismo arquitectónico **centrado en la infraestructura**, puede aplicarse tanto en el diseño arquitectónico y en tiempo de ejecución del sistema. El propósito de este mecanismo es facilitar la sustitución de componentes de la arquitectura. Para ello, los ***conectores*** se convierten en entidades de primera clase.
- El mecanismo de **reorganización de la arquitectura** se aplica en la fase de diseño arquitectónico. Se usa cuando las conexiones entre componentes necesitan ser reorganizadas. La variabilidad reside en la configuración requerida por las herramientas de configuración. Si se han adoptado previamente mecanismos centrados en la infraestructura, entonces la aplicación de este mecanismo descansa en las facilidades proporcionadas por la infraestructura.
- El mecanismo de **componente arquitectónico variable** se aplica en la fase de diseño detallado. Este mecanismo se usa cuando un componente arquitectónico puede ser reemplazado por otro que puede tener una interfaz distinta.

5.7.4. Opciones arquitectónicas para conectores

Quizá la principal preocupación al diseñar sistemas robóticos es la **necesidad de manejar la complejidad de las interacciones**, tanto entre el sistema y su entrono, como las interacciones entre los componentes individuales del sistema. Un objetivo importante de la arquitectura *software* para una aplicación distribuida, como puede ser un sistema robótico, es proporcionar un diseño concurrente

[11] *Features*

basado en paso de mensajes que sea altamente configurable. En otras palabras, el objetivo es que la misma arquitectura *software* debería poder ser traducida a muy diferentes configuraciones de sistemas. Esto es, una aplicación dada podría ser configurada para tener cada subsistema localizado en su propio nodo físico separado, o alternativamente tener todos o algunos de sus subsistemas localizados en el mismo nodo físico. Para alcanzar esta flexibilidad, es necesario diseñar la aplicación de tal modo que la decisión acerca de cómo traducir subsistemas a nodos físicos no sea hecha en tiempo de diseño, sino más tarde, cuando se haga la configuración del sistema. Consecuentemente, la comunicación entre esas tareas que se separan en subsistemas deben ser restringidos a la comunicación de mensajes.

Una arquitectura como la propuesta en esta tesis se basa en la infraestructura proporcionada por componentes, puertos y conectores. Precisamente lo que hace que la arquitectura reutilizable, que haya comunicación entre los componentes que encapsulan distintos conceptos, es la presencia de conectores y la posibilidad de variar los mismos. Para implementar estos conectores se pueden utilizar muchos de los patrones de comunicación existentes, que se pueden consultar en [Douglass00], [Schlegel02] y [Bruyninckx02]:

Event

El caso más sencillo de relación entre componentes se dará cuando la relación entre ellos sea 1-a-1 o bien 1-a-0 en el mismo *thread*, en este caso un componente solicitará los servicios ofrecidos por el otro simplemente llamando al procedimiento ofrecido por el otro componente (procedimientos almacenados en los puertos correspondientes). Se puede dar también el caso de colaboración entre componentes activos (con su propio *thread* de control) y objetos pasivos.

En este tipo de comunicaciones se dice que se envía un *evento*, que dependiendo de la forma de interacción, puede ser de distintos tipos:

- Con bajo acoplamiento. Comunicación asíncrona. Se manda una orden y no se espera una respuesta inmediata, la tarea sigue su procesamiento, y posteriormente puede ser interrumpida por otro evento asíncrono de respuesta. Puesto que productor y consumidor pueden funcionar a diferentes velocidades, se puede construir una cola FIFO entre ambos. Si no hay mensajes disponibles cuando el consumidor solicita uno, el consumidor es suspendido.
- Fuertemente acoplada. Comunicación síncrona. El productor envía un mensaje al consumidor y espera inmediatamente una respuesta:
  - ✓ Con respuesta. El productor envía un mensaje al consumidor, que éste acepta, procesa, y entonces envía la respuesta al productor. Entonces ambos pueden seguir su procesamiento. El consumidor se suspende si no hay mensajes disponibles. En este tipo de comunicación no se implementan colas entre el productor y el consumidor.
  - ✓ Sin respuesta. El productor espera la respuesta de aceptación del mensaje, cuando el consumidor acepta el mensaje, libera al productor, entonces ambos continúan. El consumidor se suspende si no hay mensajes disponibles. En este tipo de comunicación no se implementan colas entre el productor y el consumidor.

Según lo anterior, se concluye que los conectores de tipo *Event* se pueden especializar en los tipos:

- *Evento asíncrono*
- *Evento síncrono con respuesta*
- *Evento síncrono sin respuesta*

A continuación se presentan algunos tipos de conectores basados en patrones de asociación de objetos o de componentes, para conocer con más profundidad este patrones se recomienda acudir a la bibliografía recomendada [Buschmann96].

Patrón Container-Iterator

Este patrón se utilizaría para resolver la asociación 1-n entre un puerto y n puertos. El conector tendría asociada una clase container y posiblemente *iterators* cuyo cometido será que el puerto individual sepa donde está cada uno de sus puertos asociados, que se puedan incluir nuevos puertos, etc.

Cliente-Servidor

En este tipo de comunicación uno o varios clientes solicitan la información proporcionada por un servidor, cada uno de los implicados con su propio *thread* de control. El servidor en principio no conoce los clientes que se conectarán, simplemente publica sus servicios ofrecidos y conserva una lista de clientes conectados.

Cuando el servicio requerido atraviesa los límites del procesador, los objetos deben estar desacoplados. Los típicos servicios operativos que abordan la comunicación entre procesos incluyen sockets (protocolos TCP/IP y UDP) y llamadas a procedimiento remoto (RPC). Ninguna de estas aproximaciones requiere una implementación distinta de la clase cliente.

Los modelos cliente-servidor se pueden especializar según varios patrones:

Patrón Observador

Este patrón se considera muy importante en ACROSET puesto que un puerto (*subject*) pueda mantener una lista de otros puertos (*observer*) que están suscritos a la información que proporciona el primero, de forma que les avise a todos cuando la información se ha actualizado. Este tipo de conector sería idóneo para que múltiples componentes puedan acceder a la vez a la información que proporciona otro componente. El uso de un mecanismo de registro de datos permite a un componente *software* pedir una notificación siempre que un dato cambia. De esta manera, los componentes *software* no necesitan hacer un muestreo periódico en busca de datos.

Atendiendo a la posibilidad de distribución en distintos procesadores de los componentes de la arquitectura, se puede especializar el patrón *Observador* en las variantes *Proxy* y *Broker*:

Patrón Proxy y Broker

Son formas especializadas del patrón *Observador* y son útiles en situaciones donde los clientes no están en el mismo espacio de direcciones que el servidor y cuando no es conocido donde residirán clientes y servidor en tiempo de ejecución. Las formas especializadas de estos patrones son el ORB (*Object Request Broker*), donde el cliente no conoce la identidad del servidor, pero le pide a su ORB local que le conecte al servidor, y el SRB (*Service Request Broker*), donde los componentes no hacen llamadas a métodos de una interfaz, ni siquiera conocen el API exacta de la interfaz, sino que el cliente demanda un servicio, el SRB interpreta esta demanda, busca un servidor que proporcione ese servicio, y traduce la demanda del cliente a un formato que el servidor pueda comprender.

Los patrones de interacción arriba descritos son muy sencillos y suelen estar directamente soportados por cualquier modelo de componentes. El patrón observador es algo más complejo, pero en general los modelos de componentes y los entornos de programación proporcionan los medios para realizarlo de forma bastante directa. Los patrones de interacción *que conocen* los componentes no deben ir más allá de este pequeño conjunto de interacciones básicas. El resto debe ser manejado por los conectores.

Para concluir este punto, se puede consultar el artículo [Gowdy00], que a nivel de diseño detallado, ofrece una interesante comparativa entre *toolkits* de comunicaciones entre procesos aplicables en robótica. Los mecanismos IPC se han usado mucho en computación paralela y distribuida, de hecho, existen multitud de librerías de comunicación de propósito general. Según algunos autores, muy pocas librerías son adecuadas para las aplicaciones de robótica porque la aplicabilidad de una de estas librerías no está determinada sólo por su eficiencia en el intercambio de datos, sino también en cómo satisface su modelo de mensajes y funciones el flujo de datos de la arquitectura del robot [Fong01]. Se han desarrollado numerosas librerías IPC para robótica (IPT, NDDS, NML, TCA/TCX/IPC, RTC) En [Gowdy00] se puede encontrar un comentario de estas librerías y una interesante comparación entre distintos métodos de comunicación entre procesos. Evidentemente, la elección de un mecanismo de

comunicación entre procesos depende también de la implementación que se haga de la arquitectura y del lenguaje de programación utilizado escribir el código de los distintos módulos en los que se distribuya. También se puede revisar un estado de la técnica en comunicaciones en computación distribuida y fiable en [Bates98].

Nota importante: hay que destacar que esta presentación de patrones de comunicación que posibilitan la adopción de distintos conectores se ha desarrollado independientemente del medio físico existente en la comunicación entre procesadores que pudiera existir en una distribución del sistema. Se podría implementar indistintamente usando memoria compartida, redes Ethernet, o varias clases de buses (CAN bus, etc). Desacoplando los componentes encargados de la comunicación (en este caso los puertos), es posible cambiar fácilmente el protocolo sin tener que cambiar el resto de componentes del sistema.

Capítulo 6

Una Arquitectura de Referencia para Unidades de Control de Robots de Servicios Teleoperados

6.1. Introducción

El objeto de este capítulo es describir la arquitectura de referencia ACROSET, así como las decisiones que han guiado su diseño. Como proceso de desarrollo se sigue utilizando ABD [Bachmann00a], pero se adopta la notación de las 4 vistas de Hofmeister[1] [Hofmeister00], inspirada en ROOM [Selic94] para sistemas de tiempo-real, entre otras cosas. De esta manera se ofrece la posibilidad de establecer un modelo de componentes conceptuales y conectores, tal como se justificó en el Capítulo 4, en el punto *"Enfoque metodológico de la tesis"*.

El método ABD propone que, a partir de las directrices, los requisitos funcionales abstractos y de calidad y los posibles estilos, opciones y estrategias arquitectónicas que se han definido en el capítulo anterior, se descomponga el sistema en subsistemas recursivamente. Al final del método se obtiene una vista conceptual de la arquitectura, que en la notación 4-V.H. estará basada en componentes, puertos y conectores. Los **requisitos funcionales** se logran asignando una parte de la funcionalidad del sistema a cada uno de los subsistemas o componentes resultantes de las sucesivas descomposiciones del mismo. Los **requisitos no funcionales** se alcanzan mediante la selección del estilo arquitectónico que mejor se adapte a los mismos.

En ABD, 4-V.H. y cualquier otro método de desarrollo centrado en la arquitectura, la decisión más crítica se toma al **proponer** cuáles deben ser los subsistemas, módulos o componentes conceptuales de la arquitectura a partir de las directrices de la misma, los requisitos funcionales y de calidad. Como se puede deducir fácilmente, esta será la fase más **creativa**, que depende más de la experiencia, capacidad de abstracción y de razonamiento del diseñador o *arquitecto*. Por desgracia, también es la fase que menos se puede *sistematizar* según un proceso probado de desarrollo; además suele ser la

[1] En adelante 4-V.H.

parte más **crítica** y la que más influye en el posterior éxito y cumplimiento de los requisitos planteados de los sistemas.

No significa esto que la propuesta de los subsistemas, componentes, etc., sea arbitraria; precisamente la caracterización de los sistemas de teleoperación que se ha hecho en el capítulo 4, la descripción de requisitos funcionales, de calidad, de negocio y directrices de la arquitectura del capítulo 5, así como el conocimiento o propuesta de distintas opciones arquitectónicas, junto con la propia experiencia en el dominio del *arquitecto*, le ayudarán en esta fase creativa y en el planteamiento de la arquitectura.

Este capítulo se estructura presentando en primer lugar una **visión global** de la arquitectura, resaltando las directrices fundamentales que llevan a proponer una primera división en subsistemas, para después pasar a describir los **componentes fundamentales** en los que se ha dividido cada uno de estos subsistemas, asignando una parte de la funcionalidad a cada componente. A partir de las **opciones arquitectónicas** que se definieron en el capítulo anterior, se justifican las estrategias escogidas para cumplir los requisitos funcionales y de calidad mencionados, promoviendo la aparición de distintos componentes y la organización de los mismos según distintos patrones de interacción.

Al final del capítulo se muestran las tablas que resumen la agrupación de requisitos funcionales abstractos en responsabilidades, las estrategias arquitectónicas seguidas y su reflejo en la arquitectura.

Los componentes propuestos en ACROSET pueden estar presentes o no en un sistema implementado a partir de esta arquitectura dependiendo de cuales sean sus requisitos concretos, como se verá en el capítulo siguiente. Lo mismo se puede decir con el tipo de conectores que se utilicen. Lo importante es que la arquitectura sea lo suficientemente flexible para adaptar los mecanismos de interacción entre sus componentes a las características particulares de cada sistema dentro del dominio de aplicación.

6.2. Visión global de la arquitectura

El aspecto más crucial a la hora de definir arquitecturas de referencia es determinar qué partes de la misma deben ser estables y cuáles deben ser variables, asociando a éstas últimas mecanismos de configuración o extensión. El éxito o el fracaso de las arquitecturas de referencia dependen en gran medida de su capacidad de tratar la variabilidad entre los distintos sistemas del dominio considerado.

Del estudio del dominio de aplicación de la arquitectura, cuyas características se resumen en el punto 4.4.1.4. del Capítulo 4, se deduce que el principal objetivo de la arquitectura debe ser tratar con la gran variabilidad que existirá entre las distintas aplicaciones, e incluso la variabilidad que se puede dar en un mismo sistema, en previsión de futuras ampliaciones, o modificaciones para adaptarse a nuevos requisitos. El estudio comparativo realizado por J.A. Pastor en [Pastor02-td] revela que la mayoría de las variaciones no se refieren a los componentes del sistema, cuya funcionalidad puede que permanezca invariable, sino más bien a los patrones de interacción entre dichos componentes.

Por tanto, podríamos considerar que entre las directrices arquitectónicas deducidas en el capítulo anterior (**Tabla 5.1.**), aquellas que tratan de la variabilidad o modificabilidad, serán las más influyentes en el diseño de la arquitectura (**Tabla 6.1**).

Para alcanzar los requisitos de calidad que están implícitos en las directrices arquitectónicas será necesario llegar a unos **compromisos** entre los diferentes atributos de calidad. El hecho de llegar a un compromiso implica **priorizar** unos requisitos de calidad frente a otros, sabiendo que probablemente el hecho de favorecer a uno frente a otro, cuando están relacionados, suele ocasionar una pérdida de calidad en el menos favorecido (por ejemplo, en la adopción de modificabilidad frente a rendimiento). La elección deberá hacerse promoviendo un acuerdo entre los diferentes *stakeholders* que influyen en el diseño y posterior implementación de la arquitectura. La adopción de dichos compromisos se producirá tanto en el diseño de la propia arquitectura como en la implementación de un sistema concreto, que normalmente tendrá requisitos y restricciones distintas a otro implementado a partir de la misma arquitectura.

Directrices relacionadas con la variabilidad	
D3	Se busca el desarrollo rápido y de bajo coste del mayor número posible de sistemas dentro del dominio de la arquitectura.
D4	Diferentes instanciaciones de la arquitectura deben poder compartir, reutilizar, los mismos componentes.
D5	Se deben adoptar políticas que permitan una clara separación entre dichos componentes y sus patrones de interacción.
D6	La implementación de los componentes de la arquitectura podrá ser *software* o *hardware*, pudiendo ser también componentes comerciales COTS. La implementación podrá realizarse sobre plataformas diversas, incluso distribuidas.
D7	Adaptabilidad a diferentes mecanismos. Debe ser fácil añadir y quitar componentes y capacidades del sistema.
D8	Debe ser posible que los sistemas adopten ciertas acciones autónomas sencillas (principalmente reactivas) que complementen las acciones teleoperadas o incluso misiones pre-programadas (deliberativas).
D9	Debe poder gestionar distintos modos de control, distintos interfaces de teleoperación y otros sistemas externos que interaccionan con el sistema, incluso de forma simultánea.

Tabla 6.1.- Directrices arquitectónicas fundamentales relacionadas con la variabilidad de los sistemas

A partir de estas directrices fundamentales, y según recomienda el ABD, en los próximos puntos se describen los razonamientos que conducen a la propuesta de una primera división de la arquitectura en subsistemas[2].

6.2.1. Infraestructura de composición de la arquitectura (D4, D5)

Puesto que debería ser posible que diferentes sistemas compartan los mismos componentes (directriz D4), además de utilizar componentes, puertos y conectores, se deberán definir las reglas e infraestructura común que permita a los componentes ser ensamblados o conectados de diferentes formas, cambiando principalmente sus interacciones, como define la segunda directriz arquitectónica.

A menudo, los componentes tienen que interaccionar de formas complejas, de acuerdo con protocolos más o menos complicados en los que pueden intervenir más de dos componentes. En este caso, se hace más nítida la necesidad de tener en cuenta la directriz D5, se hace necesario ***separar la funcionalidad básica del componente de sus patrones de interacción***. Los conectores deben gestionar los protocolos de interacción entre componentes, definiendo sus reglas y los roles que cada componente debe desempeñar en la interacción. Los cambios de protocolo son relativamente frecuentes (especialmente si hay que integrar componentes desarrollados por terceros). Si se incluyen los detalles de los protocolos de interacción dentro de los componentes, cualquier cambio en los protocolos supone también un cambio en dichos componentes. Si se separan, es posible definir colecciones de patrones de interacción intercambiables sin que los componentes se vean afectados (ver opciones arquitectónicas al final del Capítulo 5).

Si los protocolos de interacción se embeben en los conectores es posible, por un lado, reutilizar los mismos componentes en arquitecturas muy diferentes y, por otro, adaptar los patrones de interacción a las necesidades del sistema, tanto de forma estática como dinámica. Las interacciones pueden variar:

- ✓ Durante los procesos de desarrollo y mantenimiento.
- ✓ Durante el funcionamiento del sistema.
- ✓ Entre sistemas de la línea.

[2] La directriz D3 impregna de alguna manera a todas las demás, por tanto no se considera directamente, sino implícitamente en los razonamientos asociados a las demás directrices.

La expresión de la vista conceptual de ACROSET mediante diagramas UML del estilo de la vista conceptual de [Hofmeister00] permitirá describir claramente los componentes de la arquitectura y sus puertos, así como las distintas opciones de conexión que ofrecen los conectores.

6.2.2. Importancia del *hardware* para ACROSET (D6, D7)

Una vez decidida la descripción de la vista conceptual de la arquitectura mediante componentes, puertos y conectores, habrá que empezar a definir qué tipos de componentes deben aparecer. Las directriz 3 establece que la implementación de los componentes que se definan podrá ser *software* o *hardware*. Se deber permitir además que tales componentes puedan ser COTS. Así mismo, las plataformas de implementación podrán ser muy diversas, contemplando la posibilidad de distribución. Para cumplir con esta directriz, será necesario identificar primero los componentes típicos de esta clase de sistemas, que pueden aparecer a distintos niveles de granularidad:

Al nivel más bajo, un robot tendrá como componentes *hardware* fundamentales, **sensores** y **actuadores**. El próximo nivel lo marcarían los controladores de dichos actuadores que tengan asociados a su vez algunos sensores (por ejemplo, un controlador de motor además del actuador, necesita al menos la realimentación de un sensor). En el siguiente nivel de abstracción, nos encontraremos por ejemplo con coordinadores de varios *motores controlados*, y así sucesivamente, hasta llegar a un coordinador global de robot y herramienta.

Estas funcionalidades pueden tener una implementación *hardware* o *software*. Muchos de los posibles componentes pueden encontrarse en el mercado como dispositivos *hardware* y tarjetas de control de motores o incluso como paquetes *software* comerciales para una plataforma dada. Por ejemplo, un coordinador de tres controladores de ejes que controlen el movimiento de 3 servomotores puede estar implementado gracias a una tarjeta controladora de ejes, con lo cual, se está asumiendo la funcionalidad de varios componentes en la misma tarjeta.

Como se deduce del párrafo anterior, la posibilidad de integrar *software* o *hardware* comercial (COTS) en los sistemas implementados a partir de la arquitectura es una cualidad muy deseable. Para facilitar dicha integración, cada componente *hardware* tendrá su equivalente "*virtual*" de forma que para el resto de componentes que acceden a él, sea indistinto si su implementación es *hardware* o *software*. Estos componentes *virtuales* o de *abstracción del hardware*, permiten modelar las características de los componentes físicos del sistema, definiendo sensores, actuadores, controladores de movimiento, etc. Permitirían incluso la definición de librerías de componentes y el intercambio de implementaciones *hardware* y *software* de los dispositivos representados con un mínimo impacto.

Por tanto, el primer subsistema o *componente de grano grueso* que surge lo denominaremos **Subsistema de Coordinación, Control y Abstracción de Dispositivos (CCAS[3])**. El control que realizaría sería meramente funcional, al igual que haría una tarjeta controladora de ejes, recibiendo comandos de movimiento más o menos complejos (mover una articulación, mover varias a la vez siguiendo una trayectoria, etc). Se sugiere por tanto, una división jerárquica de la funcionalidad en niveles de coordinación y control que podrán ser accesibles o no según las relaciones de contenido o agregación que se establezcan (ver punto 6.3.1). Por las características de este subsistema (presencia de lazos de control, coordinación de acciones que se están ejecutando a la vez, etc.), normalmente tendrá que satisfacer requisitos temporales estrictos.

La adaptabilidad frente a diferentes mecanismos (directriz D7) se garantiza por la estructuración del subsistema en distintos componentes que se ocupan de coordinación y control y que pueden ser agrupados para distintas implementaciones, con lo cual, también será fácil añadir y quitar componentes y funcionalidades nuevas al sistema. Para ello se propone separar también las estrategias de control y coordinación del componente de forma que puedan ser cambiadas, incluso en *tiempo de ejecución*, por ejemplo, estrategias como: control PID de un eje, estrategia de coordinación consistente

[3] *Coordination, Control and Abstraction Subsystem*

en que todos los ejes alcancen el objetivo a la misma vez, realización de la cinemática inversa del mecanismo, etc.

6.2.3. Añadiendo inteligencia (D8)

Tal como se ha definido el primer subsistema posible de la arquitectura, el CCAS simplemente recibiría órdenes de un operador o de un sistema externo que acceda a la funcionalidad del sistema. Sin embargo la directriz D8 sugiere la necesidad de que los sistemas adopten ciertas acciones autónomas que complementen las acciones teleoperadas.

En efecto, en la mayoría de sistemas será muy conveniente la posibilidad de añadir ciertos comportamientos autónomos, que se puedan combinar o superponer, a las acciones ordenadas por el operador. Por ejemplo, evitar obstáculos, detección de colisiones, etc., o muy frecuentemente, la ejecución de misiones preprogramadas. Para introducir esta "inteligencia" autónoma a la arquitectura, frente a la proporcionada por el operador, se opta por una importante estrategia, consistente en ***la separación del control y coordinación de acciones, respecto al "razonamiento" que se debe hacer para ejecutar esa acción***. Por ello, deberá aparecer en la arquitectura un **Subsistema de Inteligencia (IS[4])**, que se encargue, no sólo de proporcionar comportamientos reactivos, sino también de almacenar misiones preprogramadas que pueda ejecutar el sistema, y que serían solicitadas por una orden del operador, realizando en dicho caso el control de ejecución de la misión. Por ejemplo, limpiar un área del barco, ejecutar procedimiento de calibración, soldar un perímetro determinado, ejecutar secuencias varias, etc.

Granularidad e inteligencia no tienen necesariamente que estar acopladas, como se encuentra en muchos sistemas donde los niveles de control superiores corresponden a niveles de inteligencia también superiores. En el caso de ACROSET**, la inteligencia se encuentra en un eje distinto al de la funcionalidad**, y puede evolucionar o componerse de manera diferente. Por ejemplo, se pueden conseguir niveles de inteligencia elevados accediendo directamente a los sensores y actuadores. Dependiendo del tipo de componentes que tengamos en una implementación, existirá la posibilidad de acceder directamente a granularidades más finas, según las relaciones de agregación o composición que existan entre componentes.

Los comportamientos *"inteligentes"* no serán solamente reactivos o programados, podrían incorporarse comportamientos deliberativos autónomos, como realizar una planificación sencilla para seguir una trayectoria, planificar una ruta o los próximos movimientos que debe realizar, etc. ACROSET permite estos comportamientos puesto que expone claramente la funcionalidad del sistema para que pueda ser utilizada por el operador o por un sistema "inteligente".

6.2.4. Interacción con el usuario (D9)

Al contemplar la posibilidad de que coexistan comportamientos autónomos (reactivos o deliberativos) con las órdenes provenientes del operador (directriz D9), o incluso de otros sistemas externos, surge la necesidad de plantear un subsistema de interacción con los subsistemas usuarios, o con usuarios externos, que haga de interfaz entre la funcionalidad (CCAS) y los usuarios de dicha funcionalidad.

Al subsistema de coordinación, control y abstracción de dispositivos le debe ser indiferente cuál sea el origen de la inteligencia; puede provenir del operador, en caso de un control totalmente teleoperado, o bien de una serie de secuencias programadas en la propia unidad de control, o bien de una serie de comportamientos autónomos reactivos, o incluso de sistemas externos de navegación. Todos ellos son considerados **usuarios** de la funcionalidad. El **Subsistema de Interacción con los Usuarios (UIS[5])** hará de *puente* entre la funcionalidad y los distintos usuarios (externos o internos) de esta funcionalidad. También deberá gestionar la combinación o coordinación de órdenes provenientes de

[4] Intelligence Subsystem
[5] User Interaction Subsystem

los distintos usuarios, para cumplir con la directriz D9, que contempla la existencia de distintos modos de control en el sistema. En este sentido, los sistemas basados en el comportamiento son una fuente de ideas para sugerir la combinación de comportamientos [Arkin89], [Connell92a], [Posadas02].

6.2.5. Consideraciones sobre Seguridad, Gestión y Configuración (D12, D13)

Además de las directrices relacionadas con la variabilidad de los sistemas hay que tener en cuenta las otras directrices. Quizá las más importantes de entre ellas son las relacionadas con el funcionamiento seguro y con el rendimiento de los sistemas del dominio de ACROSET (**Tabla 6.2**).

Directrices relacionadas con el rendimiento y la seguridad	
D10	La presencia de enlaces de comunicaciones, la posibilidad de distribución y de diferentes particionados *hardware*/*software* debe tenerse en cuenta.
D11	Los requisitos de tiempo-real críticos son condicionantes para el funcionamiento correcto del sistema, aunque en los sistemas también se encuentran requisitos que no son de tiempo-real.
D12	Las operaciones sólo deben ejecutarse si existe una confianza razonable en que el resultado de las mismas no va a poner en peligro la integridad de bienes y equipos.
D13	Deben adoptarse mecanismos de tolerancia a fallos, tratamiento de excepciones, errores, mecanismos de diagnóstico para prevención de fallos, etc.

Tabla 6.2.- Directrices arquitectónicas fundamentales relacionadas con el rendimiento y la seguridad de los sistemas

Debido a las características de los sistemas de este dominio (sistemas críticos), deben adoptarse mecanismos de tolerancia a fallos, tratamiento de excepciones, errores, mecanismos de diagnóstico para prevención de fallos, etc, tal como refleja la directriz D13.

Teniendo en cuenta también que los robots de servicio, incluso los teleoperados, suelen estar en contacto con equipos o personas que pueden ser dañados ante un mal funcionamiento del sistema, los requisitos relacionados con la **seguridad** (a nivel de protocolos de seguridad y mecanismos que hagan al sistema tolerante a fallos) serán prioritarios a un nivel parecido a los de modificabilidad, normalmente sin interferir con ellos de forma negativa [ANSI99]

El hecho de que los robots de servicio teleoperados se desenvuelvan normalmente en un ambiente hostil, agresivo y con otras características adversas descritas en el capítulo anterior, hace que la **disponibilidad** del sistema deba ser alta. El ambiente dicta también que los sistemas desarrollados en este dominio deban ser robustos, tanto físicamente, como en su *hardware* y *software*, lo cual está relacionado tanto con la seguridad como con la disponibilidad.

Asimismo, debe ofrecerse al usuario una interfaz para configurar todo el sistema, no sólo de los parámetros típicos de un componente, sino incluso la presencia o no de ciertos componentes o la posibilidad de variación de los patrones de interacción entre ellos. La directriz **D7** también influye en la aparición de este subsistema, puesto que éste responde a la necesidad de configurar los sistemas en función de los mecanismos que incluyan y de las capacidades que deban poseer.

También debe haber un componente encargado de construir la aplicación, de arrancar el resto de componentes del sistema según un orden establecido. También debe *gestionar* el funcionamiento de la aplicación y su correcta finalización.

Para responder a estas necesidades de seguridad y gestión del funcionamiento del sistema, y siguiendo la estrategia arquitectónica de *"Separar el control de la monitorización de dicho control"* [Selic96] en ACROSET se propone un **Subsistema de Seguridad, Gestión y Configuración (SMCS)**. El objetivo de este subsistema es separar las actividades de control llevadas a cabo en el CCAS de las de monitorización, seguridad y configuración. La separación de estos dos tipos de actividades, junto con la existencia de interfaces de diagnóstico permite definir estrategias de tratamiento de fallos

jerárquicas e implementar estrategias de recuperación a diferentes niveles. La definición de este tipo de componentes debe ser posterior a la de los componentes a monitorizar. Sin embargo, aunque todavía no se pueda decir mucho de ellos, su mera existencia impone condiciones al resto de componentes. Así:

- ✓ Los componentes deben ofrecer interfaces de diagnóstico y configuración.
- ✓ Todos los componentes de un tipo determinado deben ofrecer las mismas interfaces de diagnóstico y configuración o al menos un conjunto de servicios de diagnóstico comunes.
- ✓ Cuanto más regulares sean estas interfaces más posibilidades habrá de que los componentes sean realmente intercambiables.
- ✓ Todos los componentes de un mismo tipo deben instalarse de la misma manera.

6.2.6. Consideraciones sobre Rendimiento

Debido a la relación contraria del **rendimiento** con el resto de requisitos de calidad, el hecho de priorizar los anteriores requisitos puede hacer que los relacionados con el rendimiento se vean perjudicados. No obstante, aunque la implementación de ACROSET en un sistema altamente modificable y seguro pudiera presentar peores prestaciones que un sistema no modificable diseñado de forma exclusiva, las necesidades de tiempo-real, sobre todo las críticas, deben quedar satisfechas. Este cumplimiento viene impuesto por definición, ya que los tiempos de respuesta deben garantizarse para que el funcionamiento del sistema sea correcto.

En la **Tabla 6.5** al final del capítulo se puede ver la asignación de prioridades que se le ha dado a cada requisito de calidad, divididos en 3 niveles, desde el menos prioritario (*) al más prioritario (***). Además, en esta tabla se ofrecen las opciones arquitectónicas asociadas a cada requisito de calidad y que se utilizarán para proponer la primera configuración conceptual de ACROSET, así como la posterior descomposición de los componentes principales en componentes de menor granularidad. Algunos de los requisitos de calidad y sus opciones arquitectónicas asociadas que se exponen en la **Tabla 6.5** han aparecido durante el proceso de desarrollo de la arquitectura, no al principio, como podría hacer suponer la exposición tabulada de todos ellos. Se hace así por claridad de exposición.

6.2.7. Transformación de requisitos en responsabilidades

El último paso que propone ABD para plantear los subsistemas principales es la transformación de requisitos en responsabilidades, para después, organizar dichas responsabilidades en grupos de responsabilidades relacionadas que puedan representar distintos elementos en la arquitectura. En la **Tabla 5.1.** del capítulo anterior, se pueden encontrar los *requisitos funcionales abstractos*, que se han agrupado y transformado en responsabilidades según muestra la **Tabla 6.4** al final de este capítulo. La utilización de estrategias arquitectónicas, como se ha visto, ha guiado la propuesta conceptual de esta arquitectura. Todo ello, junto al estudio del dominio de aplicación y a los razonamientos expuestos en los puntos anteriores sirve para proponer una primera división de la arquitectura en subsistemas tal como se muestra en la **Fig. 6.1**. En la **Tabla 6.3** se puede ver un resumen de las responsabilidades de estos subsistemas en los que se divide la arquitectura.

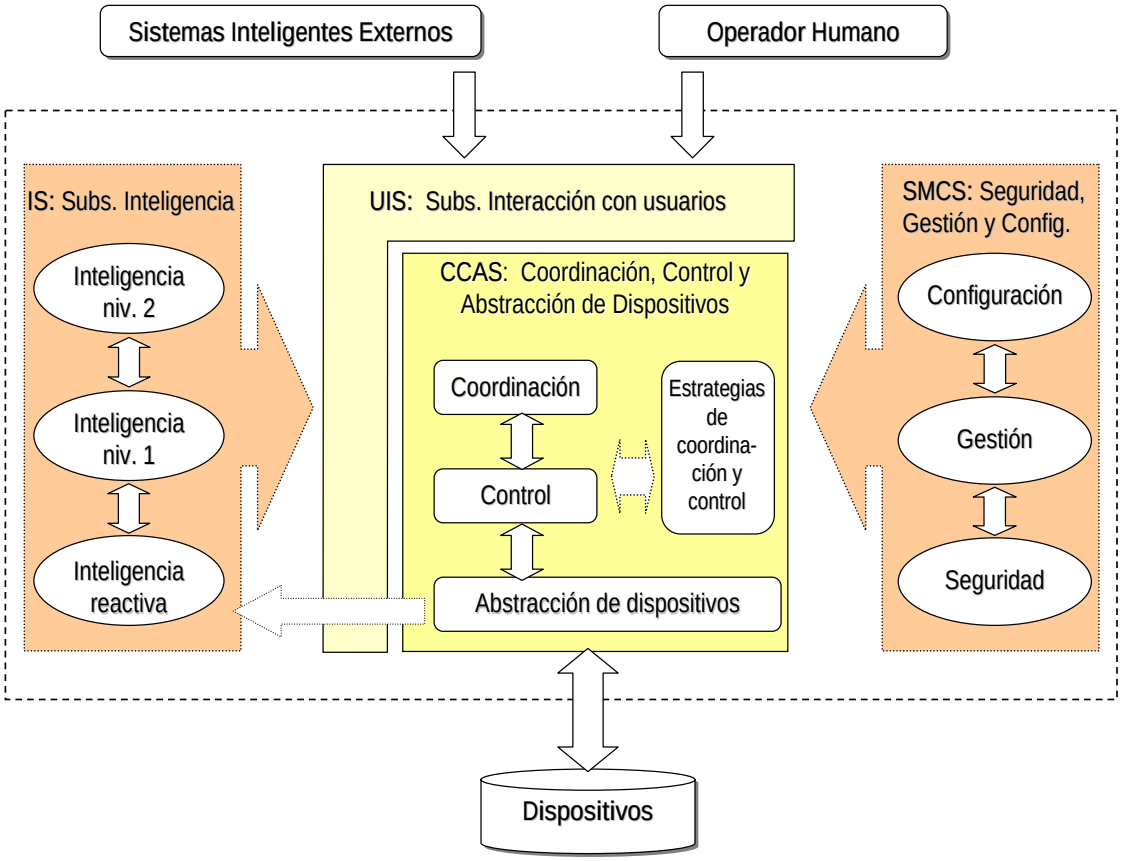


Fig. 6.1.- Subsistemas principales de ACROSET[6]

[6] La notación de este diagrama no corresponde a UML ni 4-V.H. Es un diagrama conceptual de organización e interacción de los subsistemas.

Subsistema	Responsabilidades
Coordinación, Control y Abstracción de Dispositivos. (CCAS – *Coordination, Control and Abstraction of Devices Subsystem*)	Coordinación funcional y control de los dispositivos que componen el sistema. Abstracción o representación del *hardware* presente en el sistema. Utilización de estrategias de coordinación y control intercambiables y configurables. Ofrecer acceso individual a los distintos mecanismos y dispositivos.
Inteligencia (IS – *Intelligence Subsystem*)	Supervisión y control de la ejecución de misiones pre-programadas. Realización de ciertos procesamientos autónomos sencillos, algunas tareas de planificación, etc. Generación de ciertos comportamientos reactivos que deban combinarse con las órdenes del operador (evitación de obstáculos, etc.)
Interacción con los Usuarios (UIS – *User Interaction Subsystem*)	Proporcionar al operador y a otros posibles usuarios del CCAS (internos o externos) acceso a los servicios del sistema y mostrarle estado del mismo. Interpretar los comandos que llegan de los usuarios y redireccionarlos al subsistema o componente correspondiente. Interpretar si los comandos son viables según el estado del sistema. Gestión del modo de control activo en el sistema. Arbitraje (coordinación y combinación si es necesario) de las órdenes que lleguen a la vez de distintos usuarios.
Seguridad, Gestión y Configuración (SMCS – *Security, Management and Configuration Subsystem*)	Seguridad: Monitorizar el estado y correcto funcionamiento del resto de subsistemas. Realizar diagnósticos al arranque del sistema. Informar al usuario ante eventos que puedan producirse en el diagnóstico Testear periódicamente el funcionamiento de dispositivos y componentes del sistema. Implementar políticas de tolerancia a fallos. Gestión Monitorizar el sistema en su conjunto. Almacenar el estado y diagrama de estado global del sistema y gestionar la ejecución del mismo, incluyendo la puesta en marcha (construcción) y finalización de la aplicación. Mantener información sobre el estado global del sistema, y posibilitar su recuperación tras estados especiales en acciones de recuperación de alto nivel Arranque y parada de los sistemas según una secuencia adecuada. Ubicar los diferentes componentes de la aplicación. Activación/Desactivación de subsistemas. Configuración Ofrecer una interfaz para la configuración de componentes, parámetros de funcionamiento y estrategias de control del sistema. Ofrecer una interfaz para instalación y desinstalación de componentes y conexión y desconexión. Comprobar consistencia de las configuraciones e interacciones entre subsistemas.

Tabla 6.3.- Resumen de las responsabilidades de los principales subsistemas de ACROSET

Tras identificar estos primeros subsistemas, ABD propone seguir un proceso de diseño descendente, en el que dichos subsistemas tendrán que ser refinados progresivamente. Este refinamiento dará lugar a la descomposición de los subsistemas en componentes conceptuales que se expresarán siguiendo la notación de las 4-V.H. Las opciones arquitectónicas escogidas hasta ahora y los atributos de calidad satisfechos son muy importantes, pero todavía quedan muchos por tener en cuenta. Precisamente el recorrerlos por orden de prioridad en la **Tabla 6.5**, y observar las opciones arquitectónicas asociadas a los mismos, ayudará a proponer los nuevos componentes conceptuales y conectores que aporten calidad a la funcionalidad abstracta de la arquitectura.

6.3. Componentes conceptuales fundamentales de ACROSET

La división de la arquitectura en subsistemas que se ha hecho en el punto anterior y la asignación de responsabilidades a esos subsistemas, siguiendo el ABD, aunque es bastante útil para tener una visión global de la arquitectura, es poco explícita como punto de partida para implementarla en un sistema concreto.

Los subsistemas recogen bien la división de funcionalidad, e incluso las relaciones entre los mismos, en términos de estilos arquitectónicos, patrones de diseño y plantillas. No obstante, quedan lejos del potencial expresivo que tienen los componentes, tal como se definieron en el Capítulo 3. Además, la conveniente separación de los componentes de los patrones de interacción entre ellos, exige la consideración de los conectores como entidades de primera clase, al mismo nivel que los propios componentes. Según se justificó en el *"Enfoque metodológico de la tesis"* del capítulo 4, se adopta la notación de 4-V.H. [Hofmeister00][7] para proponer la vista conceptual de la arquitectura porque permite la concreción de la arquitectura en componentes conceptuales, puertos y conectores.

Para cada uno de los subsistemas definidos anteriormente se hará una propuesta de componentes conceptuales fundamentales y conectores. Al tratarse de una arquitectura de referencia, la vista conceptual de la arquitectura indicará las características de los componentes y conectores, e incluso mostrará algunas posibilidades de conexión y configuración, pero **no todos los detalles**, como qué instancias de componentes y conectores existen, como se da en el caso del diseño de arquitecturas de un sistema concreto.

Las responsabilidades, entradas y salidas de los componentes conceptuales propuestos, se traducen en los puertos y conectores que se observan en los diagramas que se mostrarán en los siguientes puntos. En este alto nivel de abstracción, y dado que los conectores pueden ser de distintos tipos cuando se implemente la arquitectura en un sistema concreto, sólo se les distingue a priori según transfieran principalmente datos o control. En el siguiente capítulo se seguirán las opciones arquitectónicas del Capítulo 5 para concretar las conexiones a nivel de implementación. Como se observa, se han establecido los límites de la arquitectura en el contacto con los dispositivos controlados y los sistemas

---

[7] Consideraciones sobre la notación de Hofmeister: Esta notación está inspirada en ROOM [Selic94], adoptando la mayoría de sus representaciones (puertos y componentes). La principal diferencia es que añade una representación distinta para conectores y roles de esos conectores. Actualmente no se conoce ninguna herramienta que adopte la notación de Hofmeister. Rational Rose RT [Rational02] sigue la notación de UML-RT [Selic98], que utiliza UML para expresar los conceptos de ROOM. Se ha utilizado dicha herramienta adoptando los siguientes compromisos:

- Componentes y puertos se representan igual en ambas notaciones. No se distingue entre tipos de puertos, como hace ROOM y UML-RT porque las 4-V.H. no lo hace.
- Dado que ROOM y Rational Rose RT no adoptan un símbolo determinado para un conector, para adoptar el significado de la notación 4-V.H. se hacen las consideraciones siguientes: Un conector une un puerto *origen* con un puerto *destino* que se representan como un cuadradito relleno (*fuente*) y un cuadradito vacío (*destino*). El componente se nombra con el estereotipo <<*conector*>>. Si tuviera algún diagrama adicional asociado se le añadiría una clase asociada a la conexión. Este estereotipado se está adoptando en el nuevo estándar UML 2.0.
- La unión entre los puertos de un componente y los puertos de los componentes en los que se descompone son *bindings*, se distinguen porque unen puertos del mismo color y dicha unión no tiene un conector asociado.

usuarios externos. Este límite en el diagrama conceptual es independiente de la posterior implementación y despliegue de la arquitectura, que puede ser distribuida y contar con todos o sólo una parte de los componentes conceptuales que se muestran.

6.3.1. Subsistema de Control, Coordinación y Abstracción de los Dispositivos (CCAS)

En el punto 6.2.2 se hizo una disertación sobre la importancia del *hardware* en el subsistema de control de dispositivos, como consecuencia de la directriz 3. Se propuso que la arquitectura interna de este subsistema debería basarse en una distribución por capas jerárquicas de granularidad más gruesa conforme se asciende desde los sensores y actuadores, hasta un posible coordinador del robot completo (ver ventajas del estilo arquitectónico *"Arquitecturas por capas"* en capítulo anterior).

Concretando un poco más, para proponer los componentes fundamentales del subsistema de control de dispositivos, se sigue la estrategia arquitectónica *"Encapsular componentes"*, más específicamente, *"Encapsular el hardware específico del dominio"* y *"Proporcionar interfaces virtuales"* para varios tipos de dispositivos, lo que conduce a la aparición de componentes específicos de coordinación y control, así como componentes *hardware* virtuales (sensores y actuadores). De este modo se facilita el logro de uno de los principales objetivos de la arquitectura, que sería aislar la especificación de los componentes respecto de su implementación, de forma que no haya que modificar el resto de componentes si se cambia la implementación de uno de ellos.

La siempre presente estrategia de *"Separación de conceptos"* se concreta en el CCAS en las estrategias *"Separar componentes encargados de gestionar mecanismos distintos"* y sobre todo en la estrategia *"Separar coordinación y control"*. Estas opciones arquitectónicas vienen a favorecer principalmente la **adaptabilidad de los sistemas a diferentes mecanismos y configuraciones de mecanismos**, así como su posibilidad de **cambiar los despliegues**, permitiendo la distribución o la implementación *hardware* de algunos de esos componentes, sin que se vea afectada la estructura global del sistema implementado a partir de ACROSET. La definición de interfaces virtuales para los componentes *hardware* permite también adaptarse a las actualizaciones de componentes *hardware* específicos del dominio, que suelen ser frecuentes.

Todas estas estrategias llevan a que este subsistema **refleje la estructura física de los mecanismos controlados**, como consecuencia de las sucesivas capas de abstracción de la funcionalidad de los distintos dispositivos que se quieren controlar.

Así, ACROSET considera las siguientes capas que engloban sucesivamente una funcionalidad mayor:

- Capa 1: Características abstractas de los dispositivos elementales: sensores y actuadores.
- Capa 2: Controladores de una Unidad[8] de Dispositivo Simple (**SUC**s: Simple Unit Controllers)
- Capa 3: Controladores de una Unidad de Mecanismo[9] (**MUC**s: Mechanism Unit Controllers)
- Capa 4: Controladores de una Unidad de Robot (**RUC**s: Robot Unit Controllers)

Es muy importante recordar que los componentes que forman estas capas podrán ser implementados en *hardware* o *software*, pudiéndose dar todas las posibilidades: todos los componentes *software*, todos *hardware*, algunos *software* y algunos *hardware*. No hay que olvidar que en el dominio de

[8] El término Unidad ya restringe el uso de un controlador por cada entidad unitaria que se quiera controlar: un robot, un manipulador, una herramienta, un mecanismo, etc.

[9] Aunque se entiende por mecanismo un conjunto de *articulaciones* y enlaces mecánicos que puede ser controlado, un *"vehículo"* tendría una notación conceptual equivalente: la coordinación de una serie de SUCs. También se podría incluir en este razonamiento (como mecanismo) cualquier dispositivo mecatrónico que conste de dispositivos simples controlados que hayan de coordinarse. Como notación general, debido a la dificultad de englobar todos estos conceptos en uno, se adopta MUC: Coordinador de SUCs que componen una unidad.

aplicación de esta tesis, es bastante común que muchos de estos componentes sean implementaciones *hardware* comerciales (COTS), por tanto, no sería extraña la implementación de los MUCs que existiesen conceptualmente en el sistema, como tarjetas comerciales controladoras de motores. Según la estrategia *"Encapsular componentes específicos del dominio", a*unque la implementación del componente sea *hardware*, siempre existirá el componente "virtual" o de **abstracción del *hardware***, que proporcione la interfaz de dicho *hardware* al resto del sistema. Cuando se usan componente COTS, ACROSET ofrece dos posibles soluciones para incorporarlos a la arquitectura de un sistema: si el componente COTS se usa comúnmente, ACROSET define su componente "virtual", como se ha dicho. Si no se usa habitualmente, se usa el patrón *Puente*[10] para adaptar un componente virtual existente al interfaz del componente COTS usado.

6.3.1.1. Sensores y Actuadores "virtuales"

Los componentes más simples de la arquitectura serán los propios **sensores** y **actuadores**. En este caso siempre serán componentes de abstracción del *hardware* (**Fig. 6.2**):

Sensor es un componente de abstracción del *hardware* o representación de los dispositivos que normalmente será pasivo, y podrá tener acceso a él cualquier componente o capa superior. En el diseño detallado de un sistema se decidirá cuántas tareas y de qué tipo estarán asociadas a estos componentes, actualizándolos conforme leen el *hardware* a través de los *drivers* de los dispositivos.

Como nota a tener en cuenta en el diseño del resto de componentes, Sensor, no sólo será un componente que abstraiga los sensores relacionados directamente con el control del movimiento. En el CCAS descansa la representación abstracta de los dispositivos y de su estado, por tanto estarán incluidas las abstracciones de todos los sensores del sistema, (temperatura, proximidad, orientación y un largo etcétera), lo que implica que deberán ser accesibles desde cualquier componente de la arquitectura que los necesite. Un componente Sensor podrá tener múltiples lectores simultáneos, que podrán leer la información que necesiten o suscribirse para ser notificados de un cambio en dicha información.

Actuator también es un componente de abstracción del *hardware* o representación de los dispositivos, en este caso, cada uno de los *n* actuadores que haya en el sistema estarán representados por un Actuator, pero al estar destinado más a la escritura que a la lectura, a la hora de implementarlo, normalmente estará asociado a una tarea que se activa cuando llega alguna orden de actuación de los elementos de control superiores. Al contrario de lo que pasaba con Sensor, en ocasiones puede haber conflictos si múltiples escritores quieren acceder a este componente, por lo que habrá que incorporar algún elemento de coordinación en los conectores que lleguen a un Actuator.

Aunque estos son los componentes más sencillos que puede haber en el sistema en la frontera con el *hardware*, el contacto con el mismo se hará gracias a los *drivers* de los dispositivos, que están encapsulados en un subsistema con una interfaz estable (capa o subsistema de **acceso al sistema operativo**), pero cuya implementación puede ser distinta según sea el sistema operativo y los dispositivos utilizados (estrategia *"Separar el código dependiente de la plataforma"*). Este subsistema no se representa en los diagramas de la arquitectura siguientes.

*6.3.1.2. Controladores Unitarios (*UCs)*

Los componentes fundamentales del CCAS son los *Controladores Unitarios (SUC, RUC y MUC)*, que pueden encontrarse en un sistema con una cardinalidad de cero a muchos (el mínimo número de estos elementos que podría presentarse en el CCAS de un sistema concreto al menos debe ser un SUC).

Cada uno de estos **UCs* es un componente dependiente de estado que comparte con los demás una estructura similar. Por una parte, contiene un gestor del diagrama de estado del propio componente, que normalmente es el propio controlador o coordinador. Este gestor del diagrama de estado decide,

[10] *Bridge pattern*

dependiendo del estado actual del componente, si un comando debe ser ejecutado o no, o si el estado del componente debería cambiar en respuesta a una señal externa. También controla cada tarea creada por el componente. Por otra parte, la parte del componente (por ejemplo, una tarea periódica de control) que lleva a cabo el principal propósito del componente (control o coordinación) sigue el patrón *Estrategia*, de forma que el comportamiento del componente puede ser modificado fácilmente, incluso en tiempo de ejecución.

SUC

Los *controladores de una unidad de dispositivo simple* (SUC) se definen en la segunda capa de la arquitectura y constituyen el elemento de control de grano más fino de ACROSET. Los SUCs hacen las funciones de un controlador de dispositivo y contiene una estrategia de control (Strategy) además de sus puertos de entrada y salida, como se observa en la **Fig. 6.2**. El SUC regularía el lazo de control formado con el actuador correspondiente y realimentado por un sensor, por ejemplo, en el control de movimiento de un motor. Así se realizaría la típica regulación de un servo-accionamiento. El controlador también debería gestionar los límites de dicha regulación con la información proporcionada por sensores adicionales (por ejemplo, por unos finales de carrera, en caso de regular un movimiento con límites de recorrido).

El SUC genera los comandos para el actuador (Actuator) de acuerdo con las órdenes que recibe de otro componente de una capa superior que acceda a él a través del puerto SUC_Control, la información recibida de los sensores (SensorDataIn), que describen el estado del dispositivo controlado, así como de la política de control (algoritmo) que tenga activa en ese momento (Strategy). Esta política o estrategia de control puede ser intercambiable en tiempo de ejecución adoptando el patrón estrategia [Buschmann96]. Por ejemplo, se podría cambiar el típico algoritmo de control tipo PID por uno *fuzzy*, sin necesidad de cambiar el controlador.

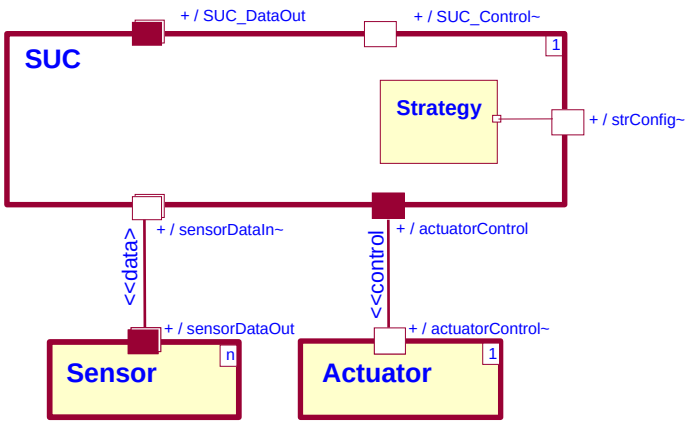


Fig. 6.2.- SUC: Simple Unit Controller

En una implementación típica, un sistema tendría tantos SUCs como lazos de control. Por ejemplo, en un robot, habría tantos controladores como ejes a controlar. Este componente, al igual que los demás, *podrá* tener puertos de configuración (como strConfig, para cambiar la estrategia de control), diagnóstico y gestión según sea la implementación y los requisitos particulares de un sistema.

Múltiples lectores podrán obtener información del estado del control a través de SUC_DataOut y sólo un escritor a la vez puede proporcionarle las instrucciones típicas de control que tenga (veremos más adelante que en caso de que hubiera múltiples escritores, debe existir un módulo de arbitraje, pero el SUC sólo recibirá órdenes por un puerto de entrada).

Normalmente, los SUCs estarán sujetos a requisitos de tiempo-real crítico, por lo que será fácil encontrarlos implementados en *hardware*. Cuando son implementados en *software* están sujetos a severos requisitos de tiempo real sobre sistemas operativos y plataformas. En caso de que los controladores fueran implementados con componentes *hardware* COTS (p.ej. tarjetas controladoras),

la función que tendría el componente SUC en el sistema sería la mera representación de los servicios ofrecidos por la tarjeta controladora, es decir, de nuevo una representación abstracta de los dispositivos.

MUC

En un tercer nivel de granularidad se definen los Controladores de una Unidad de Mecanismo (MUC). Como se puede observar en la **Fig. 6.3**, este componente estaría compuesto de un coordinador (Coordinator) con su correspondiente estrategia de coordinación (Strategy), sus puertos de entrada y salida y los SUCs cuyo comportamiento coordina.

El componente Coordinator tiene como misión la coordinación de los distintos SUCs que estén conectados a él de acuerdo a los comandos e información que recibe, siguiendo la estrategia de coordinación que esté activa. Cada SUC tendrá conocimiento sólo de sí mismo y de los sensores y actuadores con los que se comunica, por tanto necesitan un coordinador que les envíe las órdenes adecuadas para cumplir el objetivo común.

El MUC modela el control de un mecanismo completo (vehículo, manipulador, etc) y la estrategia de coordinación almacena el algoritmo de coordinación. Por ejemplo, para un manipulador dado podría ser su cinemática inversa, que sería distinta si cambiara la su configuración (número de grados de libertad, límites, etc). Esta división entre controlador y política de control se hace siguiendo la estrategia arquitectónica de *"Separar el control de los algoritmos de control y coordinación"*, lo cual facilita enormemente la adaptabilidad a diferentes mecanismos y configuraciones de mecanismos, puesto que si, por ejemplo, en esta estrategia de coordinación se encuentra almacenada la cinemática de un determinado mecanismo, si la configuración del mecanismo cambia (p.ejm. se añade o elimina un grado de libertad), simplemente hay que cambiar dicha cinemática almacenada en Strategy.

En el sistema también puede haber varios coordinadores, atendiendo a la estrategia arquitectónica *"Separar componentes encargados de gestionar mecanismos distintos"*, así, por ejemplo, en un robot móvil con manipulador, podría haber un coordinador de los distintos controladores de ejes del robot, otro para los movimientos del vehículo y otro para el manipulaldor y herramienta.

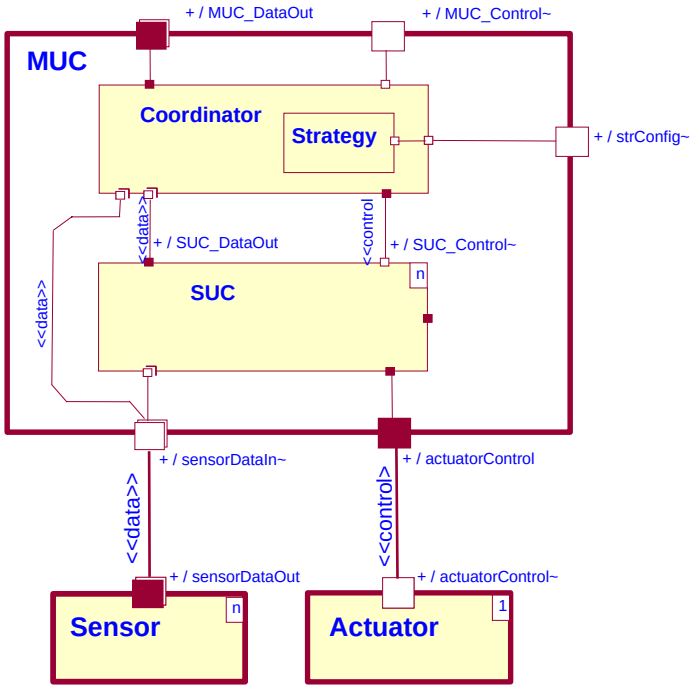


Fig. 6.3.- MUC: Mechanism Unit Controller

Se observa en la **Fig. 6.3** que Coordinator recibe información, además de los SUCs que coordina, de los sensores que necesite para realizar esta coordinación, que pueden ser distintos de aquellos que emiten información para los SUCs. La posibilidad de acceso directo a los componentes internos del MUC será una decisión que se tome en la instanciación de ACROSET, dependiendo de las características del sistema. De hecho, aunque los MUCs podrían ser implementados en *hardware* o en *software*, es muy usual que sean tarjetas controladoras de motores comerciales que restringen el rango de comandos posibles sobre sus componentes internos. De esta forma, el MUC será un componente físico que no permite el acceso directo a sus componentes internos, sino a través de su propia interfaz. Por otro lado, si el MUC se implementa como un *software* que coordina varios SUCs *hardware* o *software*, podría ser un componente lógico, o subsistema, pudiéndose acceder directamente a sus componentes internos, incluso para posibilitar su distribución en diferentes procesadores.

Las relaciones de agregación o composición entre estos componentes dependerán mucho de su implementación y distribución. En COMET [Gomaa00] se diferencia entre subsistemas agregados y compuestos siguiendo el criterio de que los objetos que son parte de un subsistema compuesto debe residir en la misma localización, pero los objetos en localizaciones *geográficas* diferentes nunca pueden estar en el mismo subsistema compuesto. Los objetos contenidos en un subsistema agregado, que podrían ser objetos compuestos, se agrupan juntos por su similitud funcional o porque interaccionan entre ellos en un mismo caso de uso. En el siguiente capítulo se verá como se han traducido los componentes ACROSET (presentados como compuestos, según la notación de 4-V.H.) en una implementación de subsistemas agregados o compuestos.

En cualquier caso, el MUC abstrae la funcionalidad de un mecanismo completo, y para el resto de componentes de la arquitectura, presenta una interfaz (a través de sus puertos) que evita tener que preocuparse de la implementación o modificación de los componentes internos del MUC.

RUC

Finalmente, a un cuarto nivel de granularidad se define el RUC (Robot Unit Controller). Este componente modela el control sobre un robot completo. Por ejemplo, un robot compuesto de un vehículo con un brazo manipulador y varias herramientas intercambiables. Como muestra la **Fig. 6.4**, un RUC está compuesto de:

- MUCs, habrá uno por cada mecanismo a controlar, el mecanismo estará compuesto a su vez de actuadores, cada uno de ellos controlados por un SUC.
- Un coordinador que genera los comandos para los MUCs y coordina sus acciones, de acuerdo a órdenes que recibe, la información que recopila y la estrategia de coordinación activa.
- Una estrategia (Strategy), que a su vez es un componente del coordinador, que contenga el algoritmo que debe seguir Coordinator para coordinar las acciones de los distintos MUCs y SUCs que haya en el sistema. Por ejemplo, en un robot compuesto de un vehículo con un manipulador, podría contener una solución cinemática generalizada que tenga en cuenta la posibilidad de mover el vehículo para alcanzar un objeto si el manipulador no puede llegar al mismo.

En la **Fig. 6.4** se puede apreciar que Coordinator genera órdenes tanto para MUCs como para SUCs. A este nivel pueden aparecer también los SUCs porque puede que algunos subsistemas físicos con entidad propia que componen un robot, estén compuestos sólo de un accionamiento, y por lo tanto, no necesiten el nivel de coordinación que ofrece un MUC, por ejemplo, en el caso de una herramienta sencilla. El hecho de que el SUC que controla una herramienta cuente con una estrategia de control que puede cambiarse, según el patrón de diseño "*estrategia*" o "*política*", es muy útil en caso de que un manipulador maneje varias herramientas. Cuando el robot cambie de herramienta, la estrategia de control del SUC se podrá intercambiar fácilmente sin necesidad de cambiar el SUC ni sus puertos.

Al igual que pasaba con los MUCs, se observa en la **Fig. 6.4** que Coordinator recibe información, además de los MUCs y SUCs que coordina, de los sensores que necesite para realizar esta coordinación, que pueden ser distintos de aquellos que emiten información para los MUCs ó SUCs.

De la misma forma, en la vista conceptual de la arquitectura, el RUC se presenta como un compuesto, pero dependiendo de la implementación para un sistema concreto, podrá ser un agregado en lugar de un compuesto. En general, el RUC puede ser un componente bastante complejo que contenga componentes *hardware* y *software* y puede exponer una gran variedad de interfaces dependiendo de la complejidad del sistema controlado.

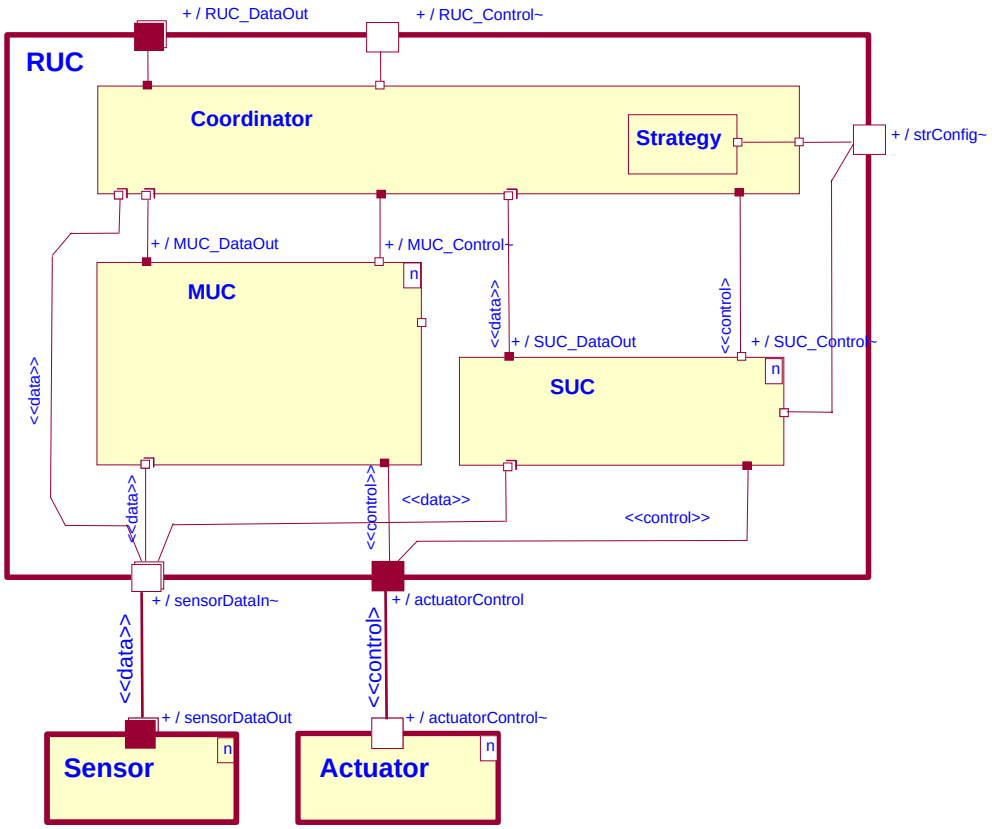


Fig. 6.4.- RUC: Robot Unit Controller.

Habiendo definido SUCs, MUCs y RUCs, parece lógico definir también un GUP, *Group Unit Controller*, capaz de coordinar el comportamiento de un grupo de robots cooperativos. Sin embargo, la arquitectura propuesta en esta tesis no va más allá de los RUCs. Hay una buena razón para ello: la *inteligencia* requerida para controlar una articulación o un mecanismo que agrupe varias articulaciones o un robot teleoperado completo es limitada, bien conocida y puede ser incluida en componentes reutilizables. Sin embargo, la inteligencia requerida para que varios robots trabajen coordinadamente demanda una aproximación más flexible, que queda fuera del ámbito de estudio de esta tesis.

Nota 1: En todos los esquemas se ha representado Sensor y Actuator como componentes independientes a los que acceden los componentes de control y coordinación. Si estos últimos componentes tuvieran una implementación *hardware*, puede que no hubiera acceso directo a Sensor y Actuator por estar conectados físicamente al *hardware* de control, en ese caso no aparecerían como componentes en el sistema, sino que estarán contenidos en el SUC, MUC, etc. que abstraiga el *hardware* de control, y se accedería a ellos a través de sus interfaces.

Nota 2: La presencia de todos los componentes descritos anteriormente no es obligatoria en un sistema, dependerá de las características de su implementación y requisitos buscados en el mismo. Tampoco tienen necesariamente que existir todas las conexiones que aparecen en los diagramas presentados, o incluso las que no aparecen (configuración, diagnóstico, etc.) Todo ello dependerá de la implementación que se haga de la arquitectura. Lo que sí debe garantizarse es la accesibilidad a la funcionalidad que proporciona el **CCAS**, (reflejo de la funcionalidad real del sistema), por parte de cualquier otro componente de la arquitectura, en concreto, de **IS** a través del **UIS** que servirá de

puente hacia la "*inteligencia*" del sistema, bien sea la proporcionada por un operario humano (comportamiento teleoperado) o la dotada por un planificador, navegador, o cualquier otro componente que sugiera comportamientos autónomos.

Nota 3: En los diferentes diagramas conceptuales de la arquitectura presentados hasta ahora no se han representado tipos de conectores debido a que se podrá optar por las diversas opciones comentadas al final del Capítulo 5. La decisión de adoptar una u otra dependerá en gran medida de si los objetos están distribuidos en distintos *threads*, *tareas* o *procesos*, incluso en distintos procesadores, o bien si existe entre ellos una relación de composición. A la hora de implementar la arquitectura en un sistema concreto se podrán utilizar los distintos mecanismos de comunicación según la distribución que se haga de ACROSET en un sistema concreto y las características y necesidades de los componentes que interconecten.

6.3.2. Subsistema de Inteligencia (IS)

La composición de SUCs, MUCs y RUC da como resultado una arquitectura jerárquica donde las decisiones fluyen de arriba abajo y la información de abajo a arriba. Esta arquitectura se adapta muy bien a los sistemas dirigidos por un operador, donde los comportamientos autónomos no existen o se reducen a una serie de acciones básicas de seguridad. También podría adaptarse bien a sistemas donde el comportamiento autónomo se limitara a reaccionar cuando se cumplan una serie reglas básicas que puedan añadirse a los controladores y coordinadores, y que les permitan tomar decisiones inmediatas, notificándolas a su vez al nivel inmediatamente superior (por ejemplo, parar el movimiento ante un obstáculo, parar un motor ante una sobrecarga, etc.). Sin embargo, hay sistemas donde el comportamiento autónomo no es nada simple. En tales casos, se hace necesario añadir un componente de inteligencia (que corresponde al IS en la **Fig. 6.1**) que integre más información y acceda a una funcionalidad mayor que la reactividad básica de un componente del CCAS.

Este componente de inteligencia se puede considerar **un *usuario* más** del subsistema de coordinación, control y abstracción de dispositivos (CCAS), al igual que lo es el operador, en el sentido que recoge información del CCAS y gracias a los algoritmos y reglas que incorpora, envía los comandos que harán que los componentes del CCAS realicen las acciones de control necesarias. Podría haber otros sistemas externos que incorporen *inteligencia* al sistema, por ejemplo un navegador, un sistema de visión capaz de determinar rutas libres de obstáculos, etc., o incluso varios operadores humanos (un operario próximo al robot que maneja una interfaz sencilla, que es supervisado por un operador experto que se encuentra en una estación de teleoperación remota). Estos sistemas de *inteligencia* pueden ser muy diversos, lo importante es que cualquiera de ellos conozca y pueda acceder a la interfaz adecuada que ofrecerá el CCAS a los usuarios.

Una de las directrices de ACROSET relacionadas con la modificabilidad (D9), sugiere que la arquitectura debe contemplar la presencia de distintos modos de control, teniendo en cuenta que en algunos de ellos pueden coexistir al mismo tiempo órdenes del operador junto a comportamientos autónomos. En el Capítulo 3, se expusieron los siguientes modos de control:

- Manual
- Exclusivo
- Compartido
    - Supervisado
    - Colaborativo o cooperativo
- Autónomo

Los modos de control compartidos (tanto el supervisado como el cooperativo), introducen el problema de la combinación de comportamientos y acciones ordenadas por un operador con las acciones autónomas que puedan sugerir los componentes *inteligentes* del sistema. Este problema ha sido especialmente tratado en las arquitecturas basadas en el comportamiento, donde múltiples

comportamientos emergentes del sistema deben combinarse para lograr una acción combinada de todos ellos.

Algunos autores han propuesto estrategias de combinación de comportamientos en este tipo de arquitecturas [Arkin89] [Graves01], y también para arquitecturas híbridas, donde el comportamiento reactivo convive con comportamientos deliberativos, incluyendo la posibilidad de un control cooperativo [Connell92b], [Posadas02] que haga posible la interacción directa con un robot mientras ejecuta una serie de comportamientos autónomos que se combinan para cumplir el objetivo buscado. Las estrategias de combinación de estos comportamientos suelen basarse en el modelado de las órdenes del operador como si fuera un comportamiento más del sistema que inyecta sus comandos directamente en la red de arbitraje [Connell90], aunque también puede haber una estructura de control superior que simplemente conecte o desconecte dichos comportamientos [Graves01].

La aproximación propuesta en ACROSET para el caso más complejo (coexistencia de órdenes provenientes de varias fuentes) consiste en la superposición del comportamiento autónomo del robot sobre el comportamiento guiado por el operador, en el caso de un modo de control supervisado o compartido, proporcionando medios para integrar ambos comportamientos del estilo de las arquitecturas híbridas cooperativas [Connell90] [Posadas02]. Esta aproximación no implica ningún cambio en los componentes definidos hasta ahora, sino nuevas fuentes de órdenes para los mismos. En el caso de un control puramente teleoperado o puramente autónomo sería más fácil, simplemente el UIS se encargaría de inhibir las órdenes provenientes del IS ó del operador según sea el caso.

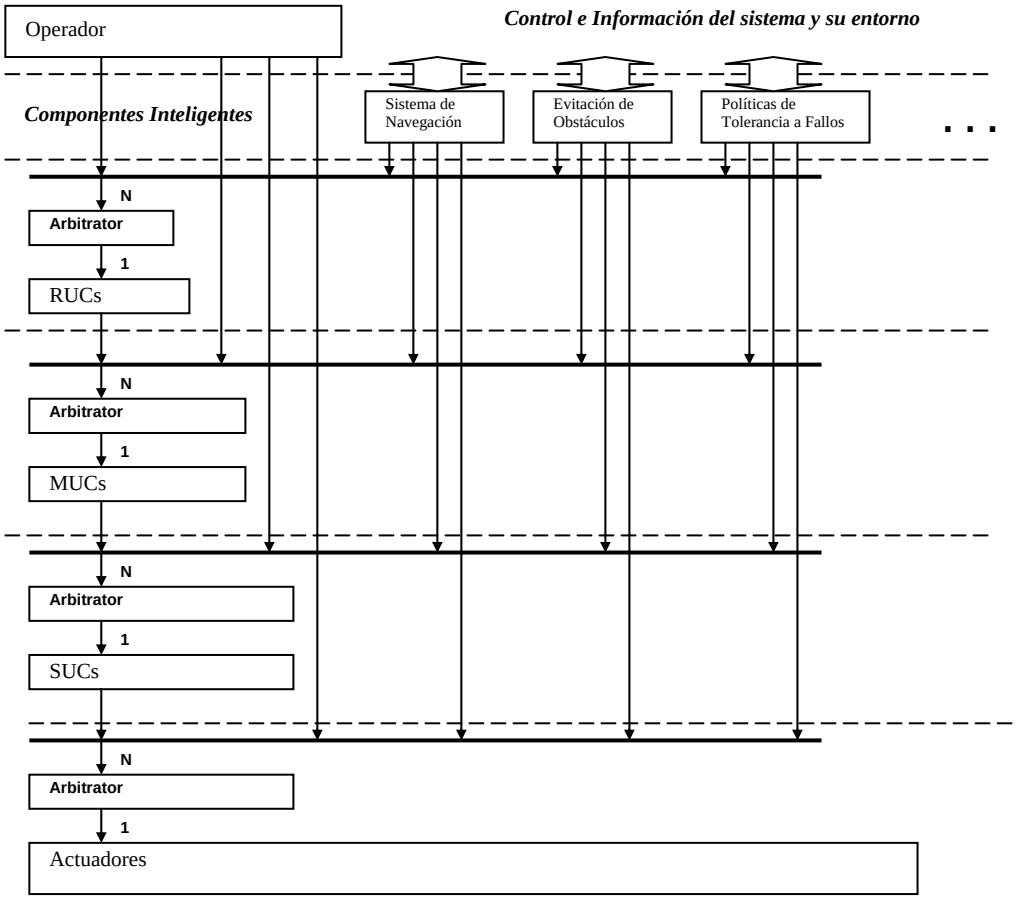


Fig. 6.5.- Superposición de comportamientos autónomos sobre los dirigidos por el operador mediante Arbitrators

La información que necesita el/los sistema/s de inteligencia del entorno y del sistema, lo obtiene del CCAS, pudiendo acceder a cualquier nivel que necesite (sensores, SUC, MUC, RUC), simplemente tiene que suscribirse al sensor o componente cuya información necesite. Con esta información y los

algoritmos, reglas programadas o heurísticos, los sistemas de inteligencia generan órdenes para el CCAS que deben combinarse con las órdenes del operador. Esta integración de órdenes y la resolución de conflictos que pudieran surgir entre ambos comportamientos, deben hacerse en el punto de interacción de ambos con el CCAS, ya sea a nivel de RUC, MUC, SUC o a nivel de los actuadores.

El componente Arbitrator surge para resolver la superposición de comportamientos. Tiene varios puertos de entrada y uno sólo de salida, a través de los puertos de entrada recoge las órdenes de varias fuentes posibles y las combina o selecciona según sea su estrategia de decisión, que será configurable y podrá variar de un sistema a otro o en el mismo sistema, según sea el modo de control. Por su puerto de salida emite la orden combinada hacia el componente de destino. Dicha orden se dirige al puerto de entrada del componente a quien va dirigida.

Gracias al Arbitrator, el componente que recoge el comando de control (RUC, MUC ó SUC) no necesita ser modificado si son varios los componentes que le dan órdenes en lugar de uno, o si cambia el modo de operación. Su puerto de entrada de comandos sólo recibe una orden, resultado del arbitraje efectuado por el Arbitrador. La utilización de estos componentes de arbitraje está fuertemente relacionada con la necesidad de separar la funcionalidad de los patrones de interacción entre componentes, al fin y al cabo, se puede considerar un conector complejo que une varios componentes con uno sólo, pudiendo tener, entre otras, las siguientes responsabilidades o políticas de arbitraje:

- Dar prioridad a una orden de mayor importancia (emergencia, etc)
- Dar prioridad a órdenes provenientes directamente del operador humano respecto a las provenientes de comportamientos autónomos, o a la inversa, dependiendo del modo de control
- Anular las órdenes provenientes de capas superiores cuando se está utilizando las capas inferiores en actuaciones de seguridad (parada segura, etc.)
- Encolamiento en un *buffer* de órdenes compatibles
- Control de acceso a componentes para diagnóstico
- Etc.

Como se observa en la **Fig. 6.5**, cada componente de una capa puede acceder a los puertos de información y control de componentes de capas inferiores (siempre que sean accesibles). Aunque esta figura muestra todas las conexiones posibles, en la mayoría de los casos, la interacción de los componentes de inteligencia sobre el CCAS será al mismo nivel que el operador, es decir, las órdenes de todos los sistemas *usuarios* del CCAS accederán al nivel del RUC a través del Arbitrator del nivel superior. Puede haber sistemas en los que ciertos componentes de seguridad y evitación de obstáculos necesiten acceder directamente a los SUCs, pero en todo caso, el Arbitrator por el que accedan a cada SUC deberá notificar a las capas superiores que está siendo accedido.

Este escenario se refleja en la figura en la **Fig. 6.6**, donde se puede ver como los comportamientos inteligentes del sistema acceden al mismo nivel que el operador, entrando por el RUC, con un componente Arbitrator, que como se verá en el próximo punto, estará situado en el Subsistema de Interacción con los Usuarios (UIS), y será el encargado de componer y filtrar los comportamientos emergentes de los distintos módulos de inteligencia y del operador para generar los comandos adecuados para el controlador del robot RUC. También se observa un comportamiento inteligente reactivo que además de acceder al primer Arbitrator, accede de forma directa a algunos de los SUCs del sistema para poder reaccionar rápidamente ante obstáculos que hagan peligrar al robot. El resto de Arbitrators no son necesarios, pues no hay acceso directo a ellos, por eso aparecen en la **Fig. 6.6** representados por una línea de puntos.

Si se observa la misma figura, entre los componentes que aportan inteligencia al sistema no se diferencia si son componentes de la unidad de control o son sistemas externos. En realidad, es indiferente para el CCAS de donde provengan las órdenes o a quien devuelva el estado del sistema, simplemente ofrece sus puertos de conexión. En caso de que el acceso se haga a través del UIS, no habrá conflictos entre los distintos componentes del CCAS por órdenes que se interfieran, porque se

componen en el Arbitrator de mayor nivel, pero si el acceso se hace en un nivel inferior, como por ejemplo el acceso del módulo de evitación de obstáculos al Arbitrator que hay sobre un SUC, entonces sí podría haber conflictos con los componentes superiores que le envían órdenes al mismo SUC. Para abordar estos casos, sería necesario que el componente de inteligencia que accede a capas inferiores también **notificara** a las capas superiores. Por ejemplo, si un subsistema de evitación de obstáculos paraliza una articulación de un robot para evitar que choque, deberá notificar a su vez al MUC que ejecuta la cinemática inversa del robot para que, según las normas de su estrategia de coordinación, decida si puede alcanzar el objetivo calculando una nueva cinemática con una de las articulaciones paralizadas, o debe parar todo el robot.

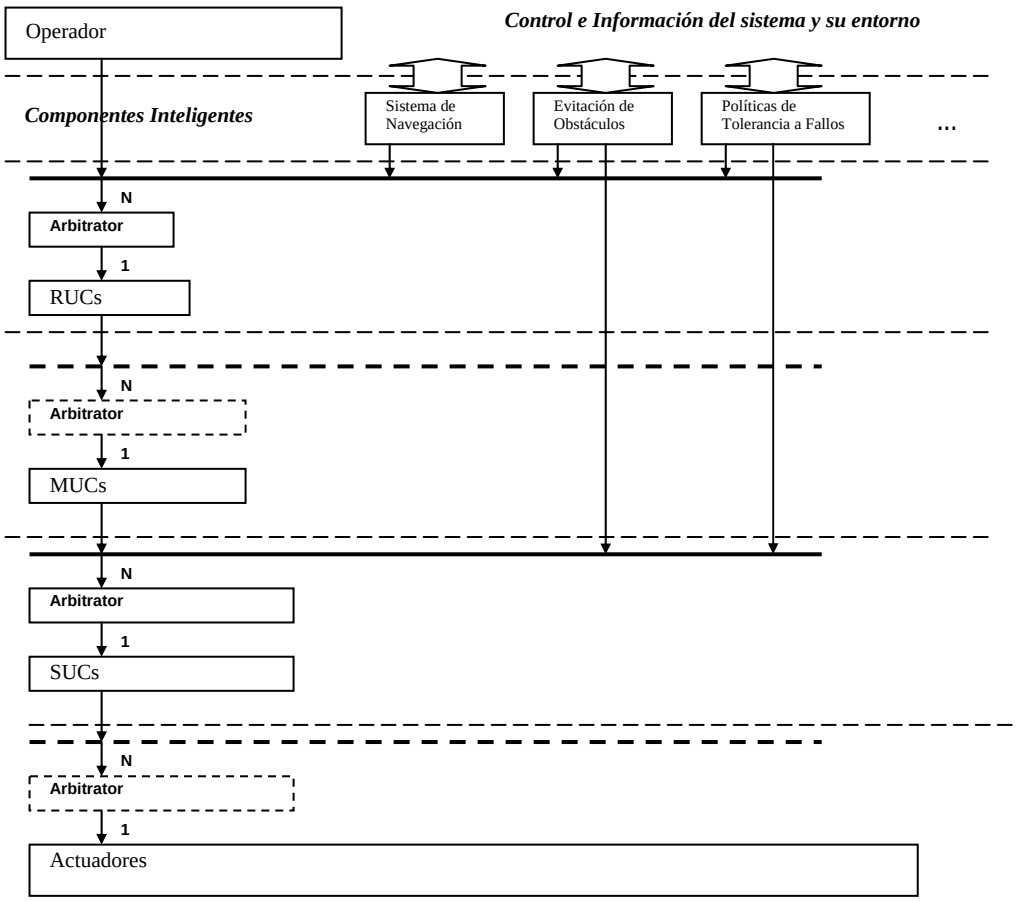


Fig. 6.6.- Escenario de arbitraje donde el subsistema de evitación de obstáculos accede directamente a SUCs

En el IS puede haber comportamientos inteligentes de distintos niveles, siguiendo la opción arquitectónica *"Separar los distintos niveles de inteligencia"*, al fin y al cabo, una nueva especialización de la estrategia principal *"Separación de conceptos según dimensiones de interés"* que aporta principalmente su beneficio en la modificabilidad de los sistemas y de la propia arquitectura. Cada nivel de inteligencia reside en un componente distinto, que igual que pasaba con los controladores, puede tener incluida una estrategia Strategy, de forma que se siga separando la gestión de la inteligencia del algoritmo, secuencia o programa almacenado. Por ejemplo, si se quiere ejecutar una secuencia de movimientos, se divide el gestor de la secuencia, que es el que genera las órdenes adecuadas para el CCAS, de la secuencia en sí. La secuencia puede ser intercambiada, pueden almacenarse varias secuencias, varias misiones, comandos programados (desplazar al origen, llevar a posición de seguridad, calibración, etc).

Las acciones "reflejas" del robot, relacionadas más bien con mecanismos de seguridad, como por ejemplo, parar un motor por sobrecalentamiento, serán incorporadas a la estrategia de los SUCs para que cuando se produzcan pueda actuar de forma inmediata. Incluso hay acciones de seguridad que

deben implementarse directamente en *hardware*, limitándose el controlador a notificar la alarma o el error cuando se produzca.

Con esta política de ir añadiendo componentes de inteligencia que generen comportamientos autónomos y que se puedan combinar con las acciones del operador (al estilo de las arquitecturas basadas en el comportamiento), se consigue una arquitectura fácilmente escalable y muy flexible. Si en la operación del robot se van desarrollando nuevos comportamientos autónomos, se pueden ir incorporando al sistema fácilmente, modificando tan solo los Arbitrators encargados de combinar los distintos comportamientos.

6.3.3. Subsistema de Interacción con los Usuarios (UIS)

El subsistema de Interacción con los Usuarios (UIS) hace de interfaz entre el CCAS y sus usuarios, tanto usuarios externos, como el operador y sistemas de inteligencia externos (sistemas de navegación, visión artificial, etc) como el subsistema de inteligencia IS, que se contempla también como un usuario más del CCAS.

Las interfaces de usuario pueden ser muchas y acceder a cualquiera del resto de los subsistemas. Las restricciones respecto a su número y organización debe imponerlas la arquitectura de cada sistema. En todo caso, el mismo UIS puede realizar distintas interfaces para ofrecer unos servicios distintos según las necesidades de los sistemas externos que *usan* la unidad de control, así como distribuir los comandos que llegan de un usuario hacia los componentes adecuados que deben tratar ese comando (control, configuración, gestión de misión, etc.)

En este subsistema estarían encapsuladas las interfaces a todos estos sistemas externos, haciendo de *puente* entre los comandos que provengan de estos usuarios y la interfaz común a todos que ofrece el CCAS. Además debería contener un gestor de modo (Mode_Manager), para controlar el cambio de modo de operación y configurar de distintas maneras, según sea este modo, al Arbitrator de más alto nivel, encargado de arbitrar entre los comandos procedentes de los distintos usuarios, como se ha comentado en el punto anterior. La estrategia de arbitraje dependerá del modo de control que esté activo en cada momento.

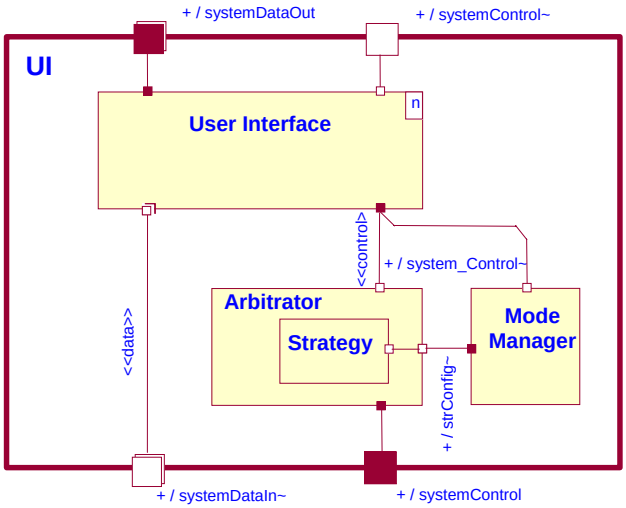


Fig. 6.7.- Subsistema de Interacción con los Usuarios (UIS)

La implementación del componente User_Interface será distinta para cada usuario que se conecte al sistema, puesto que estará encargado, entre otras cosas, de establecer la comunicación con el usuario del sistema y de traducir los comandos según la interfaz común que ofrece el CCAS (según la estrategia *"Desacoplar la interacción con el usuario"*). Estos comandos, así como el estado del sistema que se devuelve, podrán tener distintos formatos (entradas digitales de una interfaz electro-mecánica, comandos provenientes de una estación de operación compleja por medios de comunicación

diversos, etc.). Se podría decir que este componente sería como uno de los componentes de abstracción del *hardware* del CCAS, sólo que aquí abstrae la información de entrada de cada usuario. Al ser *dispositivos* distintos, tendrán diferente implementación.

Las interfaces de usuario son uno de los subsistemas más proclives a sufrir variaciones tanto durante el desarrollo del sistema como a lo largo de su vida operativa. Es imposible evitar que las modificaciones de los subsistemas a los que acceden se propaguen a las mismas. Si se amplían los servicios que proporciona un subsistema habrá que añadir el control correspondiente. Si se produce nueva información de estado habrá que representarla de alguna manera. Sin embargo, sí puede evitarse fácilmente que las modificaciones en las interfaces de usuario se propaguen al resto de los subsistemas y con un diseño cuidadoso de las mismas es posible añadir y suprimir fácilmente elementos de las mismas.

6.3.4. Subsistema de Seguridad, Gestión y Configuración (SMCS)

En el capítulo anterior se definieron los requisitos de seguridad del dominio de ACROSET, como uno de los factores más importantes. La utilización de uno u otro estilo arquitectónico que favorezca la seguridad, depende de las necesidades concretas del sistema. En todo caso, la arquitectura debe garantizar que se pueden adoptar si así se cree necesario. Su fácil modificabilidad y estructuración en componentes distintos siguiendo la separación según dimensiones de interés, parace garantizar dicha adopción. La posibilidad de acceso a todos los componentes a través de puertos de diagnóstico específicos, y la posibilidad de que los conectores puedan ser de distintos tipos (por ejemplo, adoptar un patrón observador, para que un componente de diagnóstico se pueda suscribir a los componentes que desee), también contribuyen a que se puedan adoptar estos mecanismos de seguridad.

En la visión general de la arquitectura mostrada en la **Fig. 6.1** ya se planteaba que el subsistema de seguridad, gestión y configuración se dividiera en las tres funcionalidades que su propio nombre indica. Por tanto, este subsistema incluirá tres componentes fundamentales:

- Safety_Mgr. Será un componente que monitoriza y diagnostica el correcto funcionamiento del sistema.
- Configuration_Mgr. Será el componente encargado de gestionar la configuración del resto de componentes de un sistema.
- Application_Mgr. Será el componente que almacene el diagrama de estado de la aplicación completa y gestione la viabilidad de realizar ciertas operaciones en el sistema. Además será el encargado de gestionar el arranque del sistema, creando el resto de componentes según el orden adecuado.

La aparición de un componente dedicado exclusivamente a la monitorización y el diagnóstico del correcto funcionamiento del sistema parece acercar la adopción del mecanismo *"Ejecutivo de seguridad"* como opción arquitectónica para gestionar la seguridad del mismo. Opcionalmente, en los subsistemas IS y CCAS se podrán añadir *watchdogs*, cuyas señales sean recogidas por el ejecutivo de seguridad Safety_Mgr.

Entre las tareas de diagnóstico que debe realizar Safety_Mgr se encuentran:

- ✓ Diagnóstico al arranque del sistema
- ✓ Testear que siempre se reciben una señales, *watchdog*, etc.
- ✓ Testear periódicamente el funcionamiento de dispositivos y componentes del sistema
- ✓ Informar al usuario ante eventos que se produzcan en el diagnóstico

Las tareas a realizar por el componente de configuración Configuration_Mgr, serán:

- ✓ Ofrecer una interfaz para la configuración de los componentes y parámetros de funcionamiento del sistema.

- ✓ Ofrecer una interfaz para cambios en las estrategias de control y coordinación de los componentes de control.
- ✓ Ofrecer una interfaz para la instalación y desinstalación de componentes y para su conexión y desconexión con otros componentes.
- ✓ Comprobar que los componentes se configuran e instalan de forma coherente.

.Gracias a la posibilidad de añadir puertos de configuración a cada componente se puede acceder a los mismos sin problema. En todo caso, para realizar acciones de configuración, habrá que asegurarse de tener permiso para ello según el diagrama de estado global del sistema almacenado en Application_Mgr.

Las tareas a realizar por el componente de configuración Application_Mgr, serán:

- ✓ Arranque y parada del sistema según una secuencia correcta.
- ✓ Crear y ubicar los diferentes componentes de la aplicación
- ✓ Monitorizar el sistema en su conjunto.
- ✓ Almacenar el estado global del sistema ejecutándose sobre este gestor el diagrama de estado general del sistema, de forma que no se permita pasar de un estado a otro más que por los eventos reflejados en dicho diagrama y posibilitar su recuperación tras estados especiales.
- ✓ Activación/Desactivación de subsistemas
- ✓ Gestión acciones de recuperación de alto nivel (en conexión con las políticas de tolerancia a fallos definidas en Safety_Mgr).

6.3.5. Comportamiento dinámico de los componentes

Un sistema no consiste sólo en su estructura, cada componente de dicho sistema tendrá un comportamiento dinámico, expresado con diagramas de estado, de actividad o de secuencia. En la implementación de la arquitectura, cada componente tendrá asociado un diagrama de estado según sea la funcionalidad del componente. En concreto, ya se comentó que los componentes del CCAS son dependientes de estado y comparten una estructura similar. Cuentan con un gestor del diagrama de estado y una tarea periódica de control que se activará cuando haga falta para llevar a cabo el principal propósito del componente, siguiendo una estrategia intercambiable.

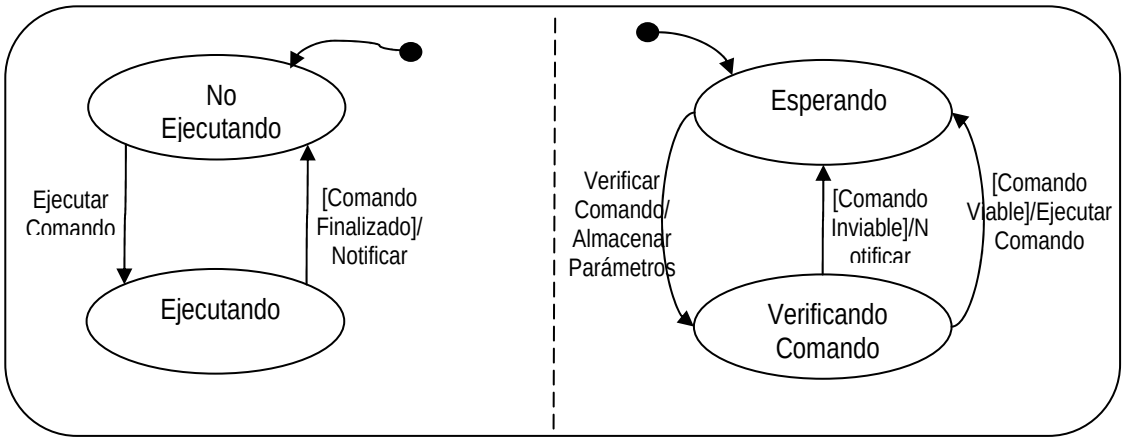


Fig. 6.8.- Ejemplo de diagrama de estado de un componente.

Por ejemplo, en la **Fig. 6.8** se puede ver un ejemplo de este comportamiento: Se trata de un diagrama de estado concurrente, porque, por un lado, el componente siempre estará esperando la entrada de un comando para verificar si se puede realizar o no y por otro lado estará controlando la ejecución de dicho comando. Si es viable, generará la orden de ejecutar comando, que ofrece dos posibilidades:

- Delegar la ejecución del comando a un componente de una capa inferior (o directamente a un actuador o un controlador hardware)
- Realizar el control software del comando, activando una tarea de control periódica.

En cualquier caso, aunque se esté ejecutando esa nueva tarea, el componente debe seguir estando disponible para procesar un nuevo comando.

6.3.6. Rendimiento

Al mencionar las posibles opciones de conectores entre componentes surge una importante interacción con los requisitos que influyen en el rendimiento del sistema. Se ha mencionado la posibilidad de distribuir los componentes en distintos procesadores, en este caso, la adopción de mecanismos de comunicación como RPC o la adopción de patrones como *Proxy* o *Broker* cobran importancia, incluso la adopción de *middleware standard* orientado a distribución de objetos, como CORBA [OMG-CORBA] pueden ser opciones para cumplir el requisito M9 *"Adaptabilidad a diferentes despliegues"*.

Cumplir los requisitos temporales en estos sistemas es fundamental, como se ha comentado en capítulos anteriores. El primer paso para poder cumplir esos requisitos es plantear una arquitectura suficientemente flexible para implementar distintas estrategias y algoritmos de planificación.

La estrategia de *"Separación de conceptos"* seguida hasta ahora en la arquitectura de referencia, se ha reflejado en el particionado de ACROSET en componentes distintos según sean algoritmos o estrategias, componentes que realizan la comunicación (conectores y puertos), controladores, coordinadores, y algoritmos que proporcionan comportamientos inteligentes a diferentes niveles. Dicha separación es fundamental para alcanzar esa flexibilidad y debe hacerse teniendo en cuenta también qué componentes participarán en procesamiento de tiempo-real y qué componentes no lo harán. Por ejemplo, normalmente el CCAS tendrá asociado comportamientos de tiempo real, puesto que estará relacionado con lazos de control de movimiento (bien sea a nivel de *software* o como abstracción de *hardware* en forma de componentes COTS). Los procesamientos relacionados con la planificación y ejecución de misiones, normalmente no tendrán necesidades de tiempo-real estrictas, excepto, quizá los comportamientos reactivos que afectan a la seguridad, de cuya reacción *a tiempo* puede depender la integridad de sistemas y personas. En todo caso, se observa que tal como se ha dividido la arquitectura, se puede diferenciar entre componentes con necesidades de tiempo-real críticos frente a los no-críticos de manera flexible. Incluso si fuera necesario, se tomaría la estrategia de subdividir un componente en parte crítica y no-crítica.

Esta división según la criticidad de los requisitos de tiempo-real, requiere una especial consideración en el diseño de interfaces entre las partes de tiempo-real del sistema frente a las que no lo son, en especial en los casos en los que se comparten recursos, para evitar bloqueos, etc. Hay varias estrategias de sincronización para compartir recursos que se pueden aplicar (**Tabla 6.5-R1**)

Es interesante el planteamiento de la distribución del sistema para alcanzar las prestaciones y rendimientos requeridos. Puede que para que se cumplan los plazos de respuesta haya que distribuir el sistema en distintos procesadores, algunos de ellos con un sistema operativo de tiempo-real. Las comunicaciones entre los procesadores son también importantes, sobre todo si tienen que comunicar sistemas con necesidades críticas. En estos casos, las comunicaciones deben ser determinísticas, lo que quiere decir que debe conocerse el plazo de respuesta máximo de una comunicación a nivel de su transporte por la red. Ejemplo de una comunicación no-determinística es Ethernet, donde el tiempo de transmisión depende del tráfico presente en la red. Algunos buses de campo, como CAN-bus sí son deterministas. Por ello y por otras causas CAN es un bus muy empleado en distribuciones de sistemas de tiempo-real. Además del medio de comunicación, es importante estudiar las propias comunicaciones, minimizando la transferencia y transformación de datos para poder realizar más transmisiones en menos tiempo.

Otro factor determinante para estudiar el posible rendimiento del sistema es la caracterización de eventos presentes en el sistema y sus patrones de llegada. Estos datos son primordiales para analizar la

planificabilidad del sistema y el análisis de los plazos de respuesta. Además de conocer si los mensajes son síncronos o asíncronos, es muy importante definir pronto si los eventos son *periódicos* (con su periodo de llegada y *jitters*) o *aperiódicos* o *eventuales (*mínimo tiempo entre llegadas, velocidad media de llegada, desviación estándar, etc*)*, porque según sean los resultados, incidirán en la viabilidad de construir el sistema a un coste razonable, cumpliendo por tanto una de las directrices importantes de la arquitectura.

Comprendiendo las características de los mensajes y eventos del sistema, es posible realizar un análisis temprano de prestaciones y elegir entre las opciones de arquitecturas de procesador disponibles.

En el siguiente capítulo, cuando se aborde una implementación práctica de ACROSET y se tenga que decidir qué procesos estarán presentes en el sistema, cuales son más o menos críticos, por una parte habrá que plantear mecanismos de comunicación que compartan recursos, *buffers,* colas, objetos protegidos, etc., por otra parte abordar el diseño de interfaces y decidir el despliegue del sistema en distintos procesadores y finalmente habrá que decidir exactamente dónde se produce la división crítica entre procesos de tiempo-real crítico y no-crítico y el tipo de comunicación que se adopta.

6.3.6.1. Análisis temporal

El análisis del comportamiento temporal de un sistema es muy importante para predecir si la estructura de tareas propuesta es planificable según los recursos *hardware* de los que se disponen. Es decir, se analiza si se cumplen los plazos de respuesta previstos según las necesidades temporales de cada tarea y del tipo de procesador sobre el que se ejecutan. De esta manera se previenen, antes de la implementación y ejecución de un programa, posibles fallos por pérdida de un plazo de respuesta, que en algunos sistemas podrían ser catastróficos (sistemas de seguridad crítica).

ACROSET no se puede analizar temporalmente como arquitectura de referencia porque no sugiere un modelo de tareas fijo, como se hacía en [Álvarez97-td], sino que ofrece un modelo de componentes que pueden instanciarse de muy diversas formas y según distintas combinaciones *software/hardware*. La elección de número y tipo de tareas concurrentes a ejecutarse en un nodo dependerá de la implementación que se haga, no de la arquitectura en sí. Por tanto, será el diseñador que instancie ACROSET en un sistema concreto el que realice los análisis temporales que considere oportunos, dependiendo del modelo de tareas que proponga. Si el modelo elegido no es planificable, la arquitectura de referencia no limita las posibilidades del diseñador para elegir una estructura de tareas distinta o para distribuir la aplicación en distintos procesadores; todo lo contrario, ACROSET se ha diseñado para favorecer dicha posibilidad de variación, gracias a la separación entre los componentes y sus patrones de interacción, de manera que se puedan cumplir los requisitos temporales de un sistema.

En [PFC04c] se pueden consultar varios ejemplos de análisis temporal basado en *Rate Monotonic Analysis* RMA [Burns01] utilizando la extensión para Rational Rose, UML-MAST [Drake00]. Dicho análisis se realiza sobre el modelo de tareas propuesto en [Álvarez97-td] cambiando los patrones de interacción entre las mismas [Ortiz03] y sobre varias combinaciones de tareas para la implementación que se hizo de ACROSET en el sistema GOYA [GOYA98a]. Para hacer el análisis RMA se deben cumplir ciertas condiciones:

- ✓ Se utiliza un sistema operativo de tiempo real y la planificación de tareas se basa en prioridades fijas y es expulsiva.
- ✓ La comunicación entre tareas se basa en un protocolo adecuado, como el protocolo de techo de prioridad, para acortar la inversión de prioridades.

6.4. Tablas de agrupación de requisitos y opciones arquitectónicas asociadas

Agrupación de requisitos y asignación de responsabilidades a los subsistemas		
Requisitos funcionales abstractos	Responsabilidades	Subsistema
Comandos de operación y movimiento El acceso a los mecanismos debe garantizarse, incluso en configuraciones distribuidas. Si existe más de un mecanismo debe poder accederse a cada uno.	Coordinación funcional y control de los dispositivos que componen el sistema. Abstracción o representación del *hardware* presente en el sistema. Utilización de estrategias de coordinación y control intercambiables y configurables.	Coordinación, Control y Abstracción de Dispositivos (CCAS – *Coordination, Control and Abstraction of Devices Subsystem*)
Debe ser posible programar y realizar secuencias de operaciones y automatizar ciertas misiones. Se podrá incluir comportamientos autónomos sencillos al sistema (evitación de obstáculos, etc) Se deben poder reaccionar rápidamente ante eventos que hagan peligrar el sistema o al entorno y usuarios.	Supervisión y control de la ejecución de misiones programadas que tenga almacenadas el sistema. Realización de ciertos procesamientos autónomos sencillos, como simulación de los siguientes comandos según el estado actual del sistema para prevenir colisiones, algunas tareas de planificación, etc. Generación de ciertos comportamientos reactivos que deban combinarse con las órdenes del operador (evitación de obstáculos, etc.)	Inteligencia (IS – *Intelligence Subsystem*)
Debe poderse cambiar el modo de operación en tiempo de ejecución. Debe realizarse el arranque y parada del sistema con seguridad, también después de paradas de emergencia. Deben garantizarse mecanismos de recuperación de información y estado ante paradas intermedias, parada segura, intercambio de herramientas, etc.	Monitorizar el sistema en su conjunto. Almacenar el estado global del sistema ejecutándose sobre este gestor el diagrama de estado general del sistema. Arranque y parada según una secuencia adecuada. Ubicar los componentes de la aplicación. Activar/Desactivar subsistemas. Mantener información sobre el estado global del sistema, y posibilitar su recuperación tras estados especiales en acciones de recuperación de alto nivel.	Gestor de Aplicación en el Subsistema de Seguridad, Gestión y Configuración (SMCS – *Security, Management and Configuration Subsystem*)
Se podrá comprobar el funcionamiento correcto del sistema. El usuario debe conocer alarmas, malfuncionamientos y otros resultados de diagnóstico	Monitorizar el estado y correcto funcionamiento del resto de subsistemas. Realizar diagnósticos al arranque del sistema. Informar al usuario ante eventos que puedan producirse en el diagnóstico. Testear periódicamente el funcionamiento de dispositivos y componentes del sistema. Implementar estrategias de tolerancia a fallos.	Componente de Seguridad (monitorización y diagnóstico) en el Subsistema de Seguridad, Gestión y Configuración (SMCS – *Security, Management and Configuration Subsystem*)
Se deben poder añadir nuevas misiones y cambiar la configuración de las mismas. Se debe poder cambiar la configuración del sistema Se deben poder cambiar los parámetros de configuración de los movimientos y operaciones. Cambios en las estrategias de control y coordinación de los controladores	Ofrecer una interfaz para la configuración de componentes, parámetros de funcionamiento y estrategias de control del sistema. Ofrecer una interfaz para instalación y desinstalación de componentes y conexión y desconexión. Comprobar consistencia de las configuraciones e interacciones entre subsistemas.	Componente de Configuración en el Subsistema de Seguridad, Gestión y Configuración (SMCS – *Security, Management and Configuration Subsystem*)

Múltiples sistemas usuarios con diferentes canales de comunicación deben poder acceder al sistema, a veces incluso de manera simultánea. Deben garantizarse mecanismos de acceso seguro a la información. Poder interpretar si los comandos son viables	Proporcionar al operador y a otros posibles usuarios del sistema (internos o externos) acceso a los servicios del sistema y mostrarle estado del mismo. Interpretar los comandos que llegan de los usuarios y redireccionarlos al subsistema o componentes correspondientes. Interpretar si los comandos son viables según el estado del sistema. Gestión del modo de control activo en el sistema. Arbitraje (coordinación y combinación si es necesario) de las órdenes que lleguen a la vez de distintos usuarios.	Interacción con los Usuarios (UIS – *User Interaction Subsystem*)

Tabla 6.4.- Agrupación de requisitos y asignación de responsabilidades a los subsistemas

Asignación de prioridades a los requisitos de calidad abstractos y opciones arquitectónicas asociadas					
ATRIBUTO	Requisitos de calidad abstractos		Opciones arquitectónicas	Pri	Reflejo en ACROSET
Organización y negocio	O1	Posibilidad de desarrollo rápido de sistemas	-Separación de conceptos* -Arquitecturas por capas -Encapsular componentes	**	Organización por capas del CCAS. Componentes de abstracción del *hardware*. División de los diferentes subsistemas y componentes.
	O2	Posibilidad de costes competitivos en transferencia tecnológica			
Modificabilidad	Relacionados con el entorno y los usuarios del sistema				
	M1	Adaptabilidad a diferentes entornos de operación	-Separación de conceptos: separar el comportamiento del modelado del entorno y los dispositivos -Incorporar distintos niveles de inteligencia, separados del control de los dispositivos	***	Componentes de abstracción del *hardware*. Descomposición del IS en componentes de inteligencia a distinto nivel.
	M2	Adaptabilidad a diferentes interfaces de usuario	-SdC: separación de la interacción con el usuario -Módulos de desacoplo	**	Interfaces de usuario en el UIS
	M3	Integración de servicios y utilidades	Patrones de comunicación (cliente-servidor, mediador, editor-suscriptor) -Módulos de desacoplo Utilización de Middleware (CORBA, etc)	**	Acceso directo de los sistemas de inteligencia al CCAS. Componentes de abstracción del *hardware*.
	Relacionados con las tareas a realizar				
	M4	Adaptabilidad a diferentes misiones	SdC: Separar entre control e "inteligencia", de los algoritmos de planificación y diferentes estrategias de reacción.	***	Separación del IS. Componentes *Strategy* dentro de los componentes de control en el CCAS
	M5	Adaptabilidad a distintos modos de operación	SdC: Separar entre control e "inteligencia", de los algoritmos de planificación y diferentes estrategias de reacción. Distintos niveles de inteligencia Distintos niveles de acceso Gestión de viabilidad de comandos	**	Gestor de modo en el UIS Componentes *Arbitrator* Gestor de aplicación

	Relacionados con la variabilidad de implementación				
	M6	Adaptabilidad a diferentes mecanismos	SdC: separación entre control y gestión de la información. Parametrizar características de componentes Separar el control de los algoritmos de control y coordinación -Encapsular componentes -Separar configuración del resto del sistema Herencia, composición y tipos de datos abstractos Abstracción del *hardware* en componentes virtuales. Separar componentes encargados de gestionar mecanismos distintos.	***	MUCs distintos para cada mecanismo Puertos en cada componente junto a patrón observador y estrategia Componentes *Strategy* Componentes *Arbitrator* Componente de configuración en SMCS Componentes de abstracción del *hardware*
	M7	Adaptabilidad a diferentes configuraciones de mecanismos			
	M8	Independencia del sistema operativo y de la plataforma	SdC: separar el código dependiente de la plataforma. Arquitecturas por capas que favorezcan la portabilidad.	**	Subsistema de interfaz con operaciones específicas del sistema operativo
	M9	Adaptabilidad a diferentes despliegues	SdC: aislar componentes según dimensiones de interés. Dividir control y coordinación en diferentes componentes. Patrones de comunicación basados en paso de mensajes que faciliten la distribución de componentes (mediador, cliente-servidor, suscriptor consumidor) RPC si se distribuye en distintos procesadores. Utilización de middleware	***	División por capas del CCAS. Clasificación de componentes RUC, MUC, SUC Composición o agregación de componentes del CCAS. Distintos tipos de conectores. Separación de puertos de flujo de datos de los de control
	M10	Independencia de la infraestructura de comunicaciones	Aislar componentes encargados de la comunicación. Patrones basados en paso de mensajes. Proxy, Broker. Utilización de middleware como CORBA. Adoptar standards de comunicación.	**	Puertos de acceso a componentes que pueden cambiar fácilmente de protocolo. Distintos tipos de conectores.
	M11	Integración de componentes *software* comerciales	Encapsulación, módulos de desacoplo.	*	Módulos de desacoplo. Puertos de componentes.
	M12	Integración de componentes *hardware* comerciales	Encapsular *hardware* específico de dominio. Interfaces virtuales para varios tipos de dispositivos. Abstracción, herencia y composición.	***	Componentes de abstracción del *hardware*. Composición o agregación de los componentes del CCAS.
Rendimien -to	R1	Tareas críticas	Asignación flexible de módulos a procesos. Posibilidad de distribuir tareas en distintos procesadores.	**	División del CCAS como subsistema con restricciones críticas. Clasificación de los componentes

			Utilizar mecanismos de coordinación entre tareas. Usar RMA para predecir rendimiento y prestaciones. Soporte de comunicaciones basadas en prioridad. Posibilidad de adopción de distintos algoritmos de planificación.		del IS en críticos o no críticos. Diferentes patrones de comunicación en conectores. Separación de algoritmos de la funcionalidad (Strategy) Separación de puertos de flujo de datos de los de control
	R2	Tareas no-críticas	Separar componentes críticos de no-críticos. Esquemas de prioridad fija (RMS). Estrategias de sincronización para compartir recursos.	*	Componentes *Arbitrator* Patrón observador Objetos protegidos
	R3	Adaptabilidad a diferentes necesidades de transmisión de datos	Minimizar transferencia de datos. Comunicaciones deterministicas. Minimización transformaciones de datos. Arbitraje y asignación de recursos.	*	Puertos de acceso a componentes que pueden cambiar fácilmente de protocolo. Distintos tipos de conectores. Componentes *Arbitrator*. Separación de puertos de flujo de datos de los de control.
Seguridad	S1	Supervisión del estado del sistema	Encapsular los componentes de diagnóstico. Diagnóstico periódico del sistema Definir políticas de manejo de errores y excepciones. Patrón Firewall:. Separar canales seguros de los no seguros. Separar el control de la medida de seguridad de ese control Adopción de patrones de seguridad (Redundancia, Monitor-Actuador, *Watchdog*, Ejecutivo de Seguridad) Auto-test de componentes	***	Subsistema SMCS separado de la funcionalidad e inteligencia. Separar componentes de seguridad o diagnóstico, configuración y gestor de la aplicación en el SMCS Separación de algoritmos de seguridad de la ejecución y gestión de la seguridad. Puertos de diagnóstico en los componentes. Diagramas de estado en componentes de control y gestor de aplicación.
	S2	Gestión de comandos	Gestión de estados y modelado de comandos. Separación de conceptos y encapsulación. Intérpretes de comandos. Simuladores para decidir la viabilidad de comandos. Controladores para comandos y mecanismos.	*	Subsistema UIS. Componente de gestión de modo de operación. Simulación de viabilidad de comandos en el UIS. Subsistema CCAS
	S3	Mecanismos de parada segura	Introducir modos de recuperación de operaciones. Hacer persistentes y accesibles los datos en los puntos de recuperación. Separar canales seguros de los no seguros. Llevar a un estado de parada segura cuando se produce un fallo	***	Componente de seguridad y gestor de aplicación en el SMCS. Puertos de datos en los componentes. Diferentes tipos de conectores. Separación de puertos de flujo de datos de los de control

			irrecuperable. Introducir políticas de seguridad. Botón de parada de emergencia.		
Fiabilidad Disponibili-dad	F1	Mantenibilidad del sistema	Utilizar un proceso de desarrollo flexible. Arquitectura por capas. Tolerancia a fallos: Redundancia de partes críticas. Definir estados de mantenimiento del sistema e intercambio de elementos.	**	Distribución por capas del CCAS Subsistema SMCS Puertos de configuración y diagnóstico en los componentes Gestor de aplicación
	F2	Disponibilidad de las comunicaciones	Tolerancia a Fallos (mediadas dinámicas, en tiempo de ejecución, de prevención, detección y corrección de fallos): Reenvío de mensajes - número de secuencia - utilización de protocolo - duplicado de enlaces - adaptabilidad a diferentes despliegues - control de tráfico.	**	Subsistema SMCS Separación de puertos de flujo de datos de los de control. Diferentes patrones de interacción en los conectores.
	F3	Disponibilidad de los mecanismos de parada segura	Botón de parada directamente cableado a los dispositivos. Políticas de seguridad conocidas por usuarios del sistema.	***	Componente de seguridad en el SMCS
Usabilidad	U1	Fácil conectividad y utilización de servicios	Arquitectura por capas. Interfaces de usuario comunes. Publicación de servicios. Separación de la interfaz de usuario del resto del sistema. Módulos de desacoplo. Arbitraje de acceso simultáneo al sistema. Gestión de prioridad. Utilizar o definir standards de interfaz con dispositivos. Adopción de standards en comunicaciones y paso de mensajes. Utilización de middleware (CORBA) y RPC.	*	Distribución por capas en el CCAS. Subsistema UIS. Componentes *Arbitrator.* Módulos de desacoplo. Gestor de aplicación. Módulos de abstracción del *hardware*. Puertos que pueden cambiar protocolo de comunicación. Diferentes tipos de conectores.
Interopera-bilidad	I1	Interacción con sistemas externos	Utilizar o definir standards de interfaz con dispositivos. Adopción de standards en comunicaciones y paso de mensajes. Utilización de middleware (CORBA) y RPC. Módulos de desacoplo.	*	Subsistema UIS Diferentes interfaces de usuario en el UIS. Módulos de abstracción del *hardware*. Módulos de desacoplo. Puertos en los componentes

Tabla 6.5.- Asignación de prioridades a los requisitos de calidad abstractos y opciones arquitectónicas asociadas

Capítulo 7

Aplicación de ACROSET en Robots de Limpieza de Barcos

7.1. Introducción

Una vez propuesta la arquitectura de referencia ACROSET, una forma de validarla consiste en implementarla en varios sistemas del dominio para el que está definida. Como se dijo en los antecedentes de esta tesis (ver Capítulo 1), los últimos trabajos del DSIE se centran en robots de limpieza de barcos, que son un buen ejemplo del dominio mencionado.

En este capítulo se explicará el análisis de un prototipo de robot de servicio para limpieza de barcos (GOYA), la instanciación de ACROSET para obtener la arquitectura de este sistema y su implementación en la unidad de control del mismo. A continuación se planteará la modificación y mejora del prototipo en un nuevo proyecto actualmente en ejecución, el EFTCoR. Finalmente para demostrar la adaptabilidad de la arquitectura a diferentes implementaciones (incluida la utilización de *hardware* y la traducción de algunos componentes a módulos *hardware*) se mostrarán los diseños realizados para un PLC[1] en el proyecto EFTCoR y una implementación totalmente *hardware* en una FPGA[2].

Para la utilización de ACROSET en un sistema concreto, como el que se trata en este capítulo, ABD es aplicable sólo en la fase de análisis (punto **7.2**), donde se plantea la concreción de los requisitos funcionales abstractos en **casos de uso**, y de los requisitos de calidad y de negocio abstractos en **escenarios de calidad**. A partir de ese punto, se adoptará la metodología de las 4-V.H. para generar la vista de módulos y la vista de ejecución, donde entran en juego requisitos del sistema más concretos, como la necesidad de un rendimiento dado, las limitaciones del *hardware* disponible, la adopción de compromisos entre prestaciones y modularidad, etc. En estos niveles próximos a la implementación se

[1] *Programmable Logic Controller.* Autómata Programable.

[2] *Field Programmable Gate Array.* Circuito integrado reconfigurable.

pueden adoptar algunas recomendaciones de la metodología COMET[3] [Gomaa00], sobre todo los criterios de estructuración de tareas. Estos aspectos se tratarán en el punto **7.3**.

La vista de código propuesta en [Hofmeister00] quizá sea menos importante en el ámbito de esta tesis, y aunque se esbozarán algunos detalles de las implementaciones realizadas utilizando los lenguajes Ada95, KOP y AWL, en un PLC y VHDL en una FPGA, quizás excede el dominio de esta tesis comentar el código realizado.

7.2. Análisis del sistema GOYA

7.2.1. Justificación y propósito del sistema

El propósito del sistema GOYA es proporcionar los medios para una limpieza semiautomática y respetuosa con el medio ambiente de los cascos de embarcaciones de gran tamaño [GOYA98a]. Esta limpieza implica la eliminación de pinturas con componentes altamente contaminantes y cancerígenos que incluyen en su composición metales pesados. Los medios de limpieza actuales consisten en chorrear el casco con diferentes granallas que se proyectan sobre el mismo a alta presión[4]. Estas operaciones, que se realizan manualmente sobre andamios (**Fig. 7.1-a**), implican riesgos para los operarios y provocan la dispersión de las granallas y los materiales de desecho en extensas áreas alrededor de los astilleros. La mayor parte de los países de la Unión Europea han prohibido la realización de estas operaciones en tanto no se resuelvan los problemas que acaban de mencionarse. La imposibilidad de realizar estas tareas supone una importante pérdida de competitividad para los astilleros ya que los armadores buscan servicios de mantenimiento integrales. No poder realizar la limpieza del casco supone casi siempre la pérdida de un cliente. Así, los principales objetivos del GOYA son:

- ✓ Alejar a los operarios de la zona de limpieza.
- ✓ Confinar los residuos resultantes y reutilizar la granalla.
- ✓ Mantener o reducir el tiempo y los costes de las operaciones de limpieza.

A más largo plazo, el sistema GOYA debe proporcionar una plataforma para la realización de trabajos de mantenimiento del casco más complicados (pintado, soldadura, reparaciones de diverso tipo, etc.) que actualmente son, como la limpieza, llevados a cabo de forma manual por operarios que trabajan sobre grúas o andamios.

El proyecto GOYA dio como resultado un prototipo operativo capaz de realizar tareas de limpieza por chorreado de granalla en un paño[5] completo de manera vertical, adaptándose a una inclinación máxima de 45º del casco de un barco. También es capaz de realizar labores de *spotting*[6] con una baja eficiencia. El proyecto EFTCoR [EFCoR02] surgió posteriormente como continuación del proyecto GOYA, y tiene un carácter europeo, involucrando a varias empresas y universidades.

7.2.2. Características y requisitos del sistema

Los requisitos del sistema GOYA están descritos con todo detalle en [GOYA98b]. No se trata de cansar al lector con una enumeración detallada de los mismos, sino de resumir algunas características

---

[3] Incuso en el modelo de análisis se adopta la notación UML utilizada por COMET para plantear el modelo estático del problema.

[4] En algunas ocasiones se utiliza agua a alta presión, pero los armadores prefieren granalla porque con ella se obtiene un acabado mejor que favorece la adherencia de la pintura.

[5] Superficie del casco que puede abarcar el sistema de limpieza automatizado sin necesidad de volver a posicionarlo.

[6] Limpieza de puntos singulares y área de pequeño tamaño dispersas por el casco

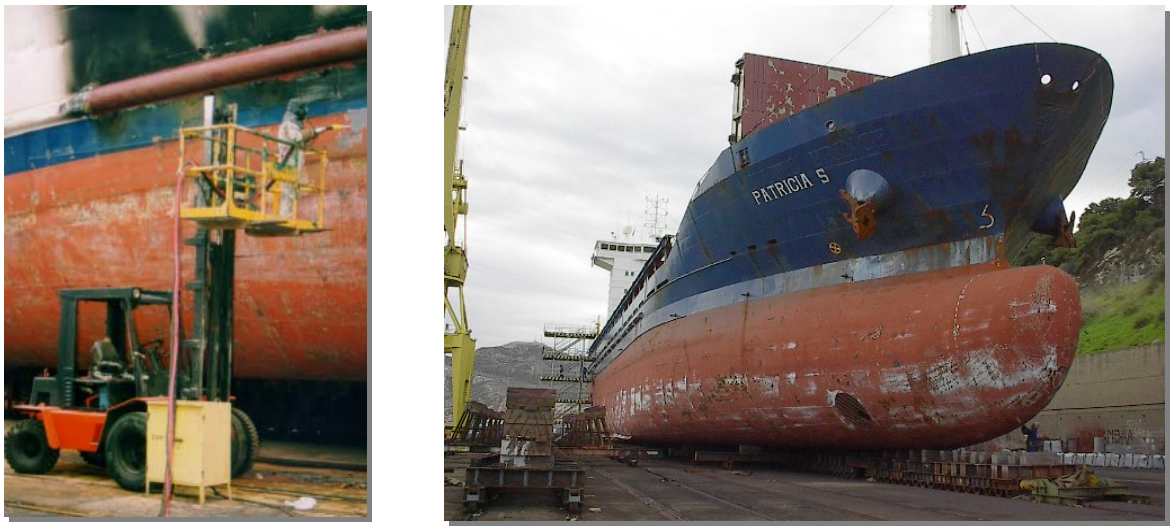
que permitan entender el tipo de sistemas que se pretende desarrollar y relacionar sus propósitos, componentes y requisitos con:

- ✓ La arquitectura de referencia propuesta en esta tesis y su aplicabilidad para este sistema.
- ✓ La concreción de los requisitos funcionales y de calidad abstractos en requisitos funcionales concretos y escenarios de calidad.

En primer lugar, se pretende que el proceso de limpieza se realice no ya de forma teleoperada, sino semiautomática. En primera instancia, el sistema es posicionado en la zona del barco adecuada, y es conducido a una posición de partida segura. Una vez en esta posición, debe poder realizar la limpieza de un paño completo, cuyas dimensiones dependen de la estructura mecánica y cinemática del sistema.

Se podría decir que el sistema GOYA es, más que un tele-manipulador, un robot teleoperado (ver Capítulo 1). Por un lado, el sistema debe ser capaz de actuar de forma autónoma, limpiando paños enteros del casco de un buque sin la asistencia o supervisión constante del operador. Por otro, dadas las características del entorno de operación, no puede liberarse de una cierta supervisión. En general, durante la mayor parte del tiempo, el comportamiento reactivo programado predomina sobre el deliberativo (órdenes del operador), pero cuando éste ocurre tiene prioridad sobre aquél.

Los movimientos deben adaptarse al contorno del casco y a las singularidades de su superficie. El entorno es en gran medida no estructurado (ver **Fig. 7.1-b**), de forma que la información sobre dicho entorno debe obtenerse a partir de diferentes formas de sensorización. Los cascos de los barcos tienen superficies relativamente fáciles de recorrer y otras muy complicadas, bien por la presencia de abultamientos, agujeros, salientes y ventanas, bien por su geometría, plana en unas zonas y muy curvada en otras. Esta variabilidad en el entorno de operación hace pensar que será necesario diseñar distintos sistemas robóticos, cada uno adaptado a un problema específico, en lugar de un robot genérico para cualquier situación. Para favorecer la reutilización, se consideró oportuno seguir un patrón de diseño común para los robots a diseñar para este dominio: Utilizar un sistema de posicionamiento primario consistente por lo general en un vehículo-grúa comercial que pudiera cubrir grandes áreas del barco y un sistema de posicionamiento secundario montado en el sistema primario que pudiera situar una herramienta en un área relativamente pequeña y con la suficiente precisión. En el GOYA, el sistema primario es un vehículo-grúa de tijera y el secundario un sencillo posicionador de herramienta.



a) Operario chorreando b) Casco de buque para chorrear sólo algunas zonas (*"spotting"*)

Fig. 7.1.- Entorno de operación de los sistemas GOYA y EFTCoR

La calidad del acabado es un parámetro importante, que depende del avance del cabezal de limpieza, de la presión del chorreo y de la granalla empleada[7]. El funcionamiento del cabezal de limpieza y del posicionador deben sincronizarse de acuerdo con el desarrollo de la operación. La posición del vehículo-grúa depende del paño y se mantiene constante mientras éste no cambia.

Dadas las dimensiones de los cascos, puede haber varios mecanismos realizando operaciones de mantenimiento simultáneas en diversas partes del mismo. Hay que evitar solapamientos y colisiones. Deben existir servicios que permitan definir las misiones a realizar y repartirlas entre los diferentes sistemas.

La fiabilidad, la disponibilidad y especialmente la seguridad son requisitos importantes. En los entornos industriales el tiempo es dinero. La tasa de fallos debe ser muy pequeña y el tiempo de recuperación mínimo. En ningún caso el tiempo de operación puede ser superior al empleado con las técnicas actuales y deben proporcionarse modos de funcionamiento degradados que permitan realizar las operaciones en caso de un fallo parcial del sistema. En particular:

- Si falla el modo de funcionamiento automático, el sistema debe poder trabajar en un modo puramente teleoperado.
- En caso de fallo general del sistema de control los dispositivos deben poder accionarse directamente desde botoneras provistas a tal efecto.

A corto plazo, las principales fuentes de variabilidad entre sistemas están constituidas por:

- El empleo de diferentes tecnologías de limpieza: chorreo con granalla, chorreo de agua a alta presión y láser.
- Las dimensiones y formas de los barcos pueden ser muy diferentes. Incluso en un mismo barco, las zonas a limpiar son muy distintas (paredes verticales, finos de proa y popa, fondos, etc), lo que sugiere la posibilidad de utilizar vehículos-grúa diferentes, adaptados a diferentes tamaños y formas del casco.
- Hay grandes diferencias entre limpiar pequeñas zonas dispersas a lo largo del casco del barco (*spotting*) y limpiar toda la superficie del barco (*full-blasting*), que inciden principalmente en la manera de desplazar el cabezal de limpieza e influye en el rendimiento del sistema.
- Los dos puntos anteriores sugieren la posibilidad de utilizar diferentes posicionadores de herramienta de limpieza (e incluso distintas herramientas), adaptados a diferentes tamaños, formas del casco y procesos de limpieza.

Conviene que el vehículo-grúa sea un mecanismo comercial, en tanto que los cabezales y posicionadores serán en general diseños específicos que, sin embargo, estarán en su mayor parte constituidos por componentes mecánicos, eléctricos, neumáticos e hidráulicos comerciales. Hay una necesidad muy fuerte de poder integrar dispositivos *hardware* específicos asociados a tales componentes que habitualmente son suministrados con los mismos.

A medio plazo la variabilidad viene determinada por la ampliación de las operaciones de mantenimiento a realizar en el casco del buque. Puesto que el sistema puede extenderse para considerar otras operaciones de mantenimiento, los requisitos de adaptabilidad a nuevos mecanismos y misiones son muy importantes. Operaciones como el pintado apenas suponen nuevos requisitos, pero la soldadura o el mecanizado son muy exigentes en términos de precisión del posicionamiento:

- La precisión de posicionamiento y la complejidad de la cinemática tenderán a hacerse mucho más exigentes.

[7] Si se utilizan tecnologías láser o agua a presión los parámetros son evidentemente otros. Las técnicas de chorreo son las más empleadas por la calidad superficial y rendimiento alcanzados.

- Aspectos relacionados con el comportamiento dinámico de los mecanismos y con la flexión de los posicionadores, que pueden despreciarse para la limpieza, tendrán que tomarse en consideración.
- El sincronismo entre posicionador y herramienta se irá haciendo cada vez más complejo.
- Los servicios de definición y reparto de misiones deberán ampliarse para considerar las nuevas operaciones.
- Los requisitos de las nuevas operaciones pueden forzar a emplear brazos industriales comerciales que proporcionen las suficientes prestaciones.

Otras fuentes importantes de variabilidad tanto a corto como a largo plazo son:

- Puede cambiar *hardware* específico asociado a los accionamientos y a la sensorización.
- Según las capacidades exigibles a los sistemas, parte de los componentes pueden implementarse en *hardware* o en *software*.
- Pueden cambiar la infraestructura de comunicaciones. Existe un interés muy fuerte por parte de los astilleros en el uso a medio plazo de tecnologías de comunicaciones inalámbricas
- Pueden cambiar las plataformas de ejecución dentro de ciertos límites.

7.2.3. Elementos principales del sistema.

Para cumplir los requisitos expuestos en el punto anterior, se propone el sistema GOYA (**Fig. 7.2**), que básicamente consiste en una plataforma elevadora con un posicionador que acerca un cilindro neumático al casco del barco. Este cilindro mueve un soporte con el cabezal de limpieza de un extremo a otro de su recorrido, limpiando a su paso la franja correspondiente del buque. A continuación la plataforma elevadora baja para situar al cabezal en la franja contigua. Siguiendo este proceso sucesivamente, habrá limpiado de forma automática un paño completo de dimensiones (3 x 5 metros).

Los componentes principales del sistema GOYA son:

- Un vehículo-grúa que porta al resto de los mecanismos y cuyo objetivo es posicionar al sistema en el paño[8] del casco donde se debe realizar la limpieza.
- Mecanismos posicionadores que sitúan al cabezal de limpieza sobre diferentes áreas del paño.
- El cabezal de limpieza, cuyas características varían en función de la tecnología empleada (láser, agua o granalla), de las granallas utilizadas y de los parámetros de calidad exigibles a la limpieza.
- Un sistema de sensorización y actuación que permitirá capturar la información del estado del robot para que la unidad de control, siguiendo instrucciones del operador o bien generando ciertos comportamientos autónomas, pueda activar ciertos actuadores (hidráulicos, neumáticos y/o eléctricos) que doten de acción al robot GOYA.

La **Fig. 7.2** muestra el prototipo GOYA en el que pueden observarse algunos de estos componentes. En la figura **Fig. 7.4.-a,b** se pueden ver imágenes reales del prototipo.

[8] Se entiende por *paño* el área que puede alcanzar el sistema automatizado sin tener que mover el vehículo-grúa que transporta el elemento de limpieza.

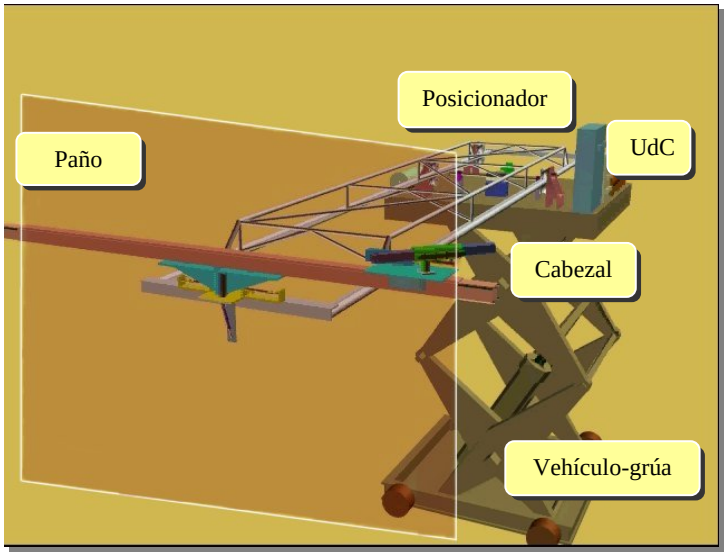


Fig. 7.2.- Prototipo GOYA: sistema mecánico

Además de los sistemas electro-mecánicos y los propios sensores y actuadores que constituyen el robot GOYA, hay una serie de subsistemas en los que se divide el GOYA y que se esquematizan en la **Fig. 7.3** junto con el tipo básico de información que intercambian Los subsistemas se han definido siguiendo los siguientes criterios:

- Cada subsistema responde a una necesidad bien definida dentro del sistema global
- Cada subsistema incluye una funcionalidad fuertemente relacionada
- Cada subsistema tiene valor por sí mismo y se puede desarrollar independientemente del resto de los subsistemas, dado que es posible identificar claramente desde el principio las interfaces que deberían mostrar. Incluso se puede tratar de sistemas completos COTS.

Siguiendo por tanto estos criterios, se definen los siguientes sistemas (**Fig. 7.3**):

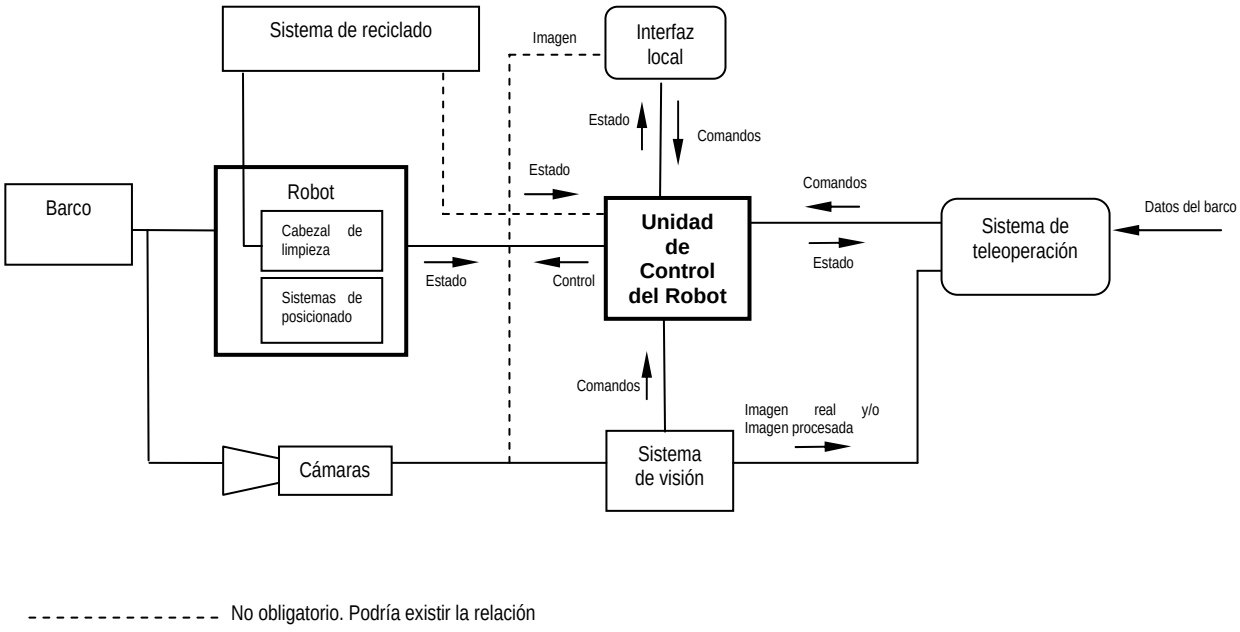


Fig. 7.3.- Esquema simplificado de subsistemas que componen el GOYA

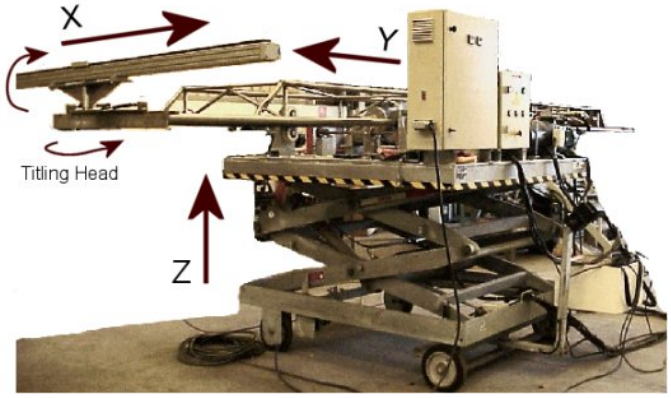
- **Los sistemas de operación.** En cuanto a la capacidad de operación sobre el sistema que tienen los operarios del mismo, se pueden diferenciar dos tipos de sistemas de operación.
    - **Un sistema de teleoperación remoto** (**Fig. 7.4-d**), que ofrece a un teleoperador la funcionalidad del robot y la información del estado del sistema, incluyendo así la posibilidad de gestión y administración del proceso de limpieza completo [PFC01b]. Dependiendo de las necesidades del sistema, y si existe un sistema de operación local, se puede convertir simplemente en un sistema de **monitorización** y **gestión** de las tareas realizadas por el robot. Típicamente puede ser una estación de trabajo o PC enlazado por red con la unidad de control
    - **Un sistema de operación local** que se diferencia del anterior, no sólo por su proximidad al robot, que podría ser lo de menos, sino por su funcionalidad: podría ser simplemente una botonera electromecánica que ordenara las tareas más básicas (**Fig. 7.4-e**), o podría ser una PDA [PFC01c] o un panel de operador portátil (**Fig. 7.4-f**) con una funcionalidad mayor. Debe tener accesible una seta de emergencia cableada directamente al robot, por requisitos de seguridad [ANSI99].
- **La unidad de control (UC)** que es la encargada de controlar los dispositivos robóticos (sistemas de posicionado y herramientas) usadas en las tareas de mantenimiento de acuerdo a los comandos recibidos del operador y los comandos y señales generados por otros subsistemas (**Fig. 7.4-c**). Para el GOYA se eligió un PC-Industrial con dos tarjetas de entrada/salida digital y una lectora de *encoders*.
- **El sistema de visión**, que se compromete a realizar tres clases de funcionalidad:
    - Ofrecer al operador imágenes reales del proceso.
    - La concerniente a la inspección de calidad del casco del barco, que determina si la calidad alcanzada tras el tratamiento es la esperada (SA 2 ½).
    - Capacidades de navegación, que proporciona información para dirigir la herramienta de limpieza sólo a las zonas del paño que necesitan tratamiento[9].
- **El sistema de reciclado**, encargado de recoger los residuos del área de trabajo y reciclarlos para que pueda ser utilizada la granalla una vez liberada de los residuos de pintura.

De todos los sistemas mencionados que constituyen el sistema GOYA, la Unidad de Control (UC) es el sistema cuya arquitectura debe ser fruto de la instanciación de ACROSET. El resto de sistemas que interaccionan con la Unidad de Control se han definido como sistemas usuarios y tienen unas interfaces bien definidas, que deben poder acoplarse con las interfaces de la Unidad de Control.

Para plantear con propiedad la instanciación de la arquitectura, se propone, por un lado, la concreción de los requisitos abstractos en casos de uso y escenarios de calidad que sugiere ABD, (7.2.4), y por otro lado, un modelo de análisis del problema, como se propone en COMET [Gomaa00], que entre otros aspectos, defina claramente las relaciones que debe tener la unidad de control con el resto de sistemas (7.2.5).

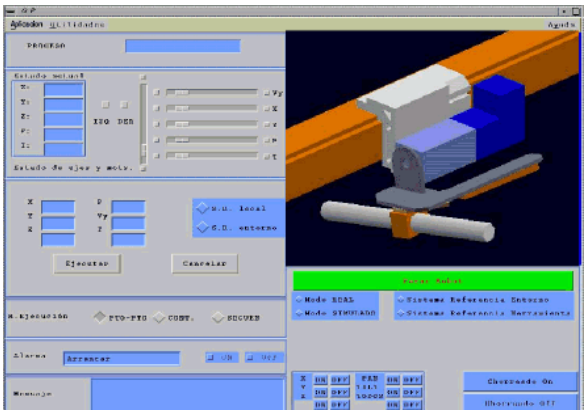
[9] Esta funcionalidad no se incorporó finalmente al prototipo GOYA, pero si al sistema EFTCoR.

a) GOYA en pruebas sobre plancha



b) Cinemática del GOYA

c) PC Industrial como Unidad de Control



d) Interfaz de usuario en el Sistema de Teleoperación

e) Botonera de control local

f) HMI portátil– Interfaz local

g) Cabezal de chorreado

h) Bomba impulsora de granalla

Fig. 7.4.- Sistema GOYA y sus componentes

7.2.4. Casos de uso y escenarios de calidad

Según el ABD [Bass99], los requisitos funcionales y de calidad abstractos que condicionaron el diseño de ACROSET, se concretan en casos de uso y escenarios de calidad respectivamente al instanciar la arquitectura en un sistema concreto como el GOYA.

Además de los casos de uso del sistema, el primer paso para comprender el proceso a realizar debe darse analizando dicho proceso dentro del conjunto de actividades a realizar mientras el barco se encuentra en el astillero. Para ello, en el Anexo III de la tesis se puede consultar un modelo completo de datos de las actividades de mantenimiento en un astillero y un diagrama de flujo de dichas actividades.

El sistema implementado en el prototipo GOYA (**Fig. 7.4-a y b**), es manejado localmente por un usuario mediante una *botonera* electromecánica (**Fig. 7.4-e**), por la cual accede a la realización de movimientos sencillos de posicionamiento, arranque, parada de emergencia y realización de la tarea automática *"limpiar paño"*. Gracias a una conexión de red se tiene acceso remoto teleoperado al sistema, disponiendo de una funcionalidad mucho mayor, que incluye visualización y monitorización del trabajo realizado, con imágenes de video en tiempo-real, así como funciones de simulación de movimientos, configuración y diagnóstico (**Fig. 7.4-d**). Toda esta funcionalidad se resume en los diagramas de casos de uso expuestos en el siguiente punto.

7.2.4.1. Casos de uso

La **Fig. 7.5** muestra los casos de uso de un usuario local que accede al sistema a través de una botonera electromecánica. Estos casos de uso son significativos para describir el funcionamiento básico del sistema y sus objetivos primarios. En el Anexo III se realiza una exposición detallada de los mismos.

.

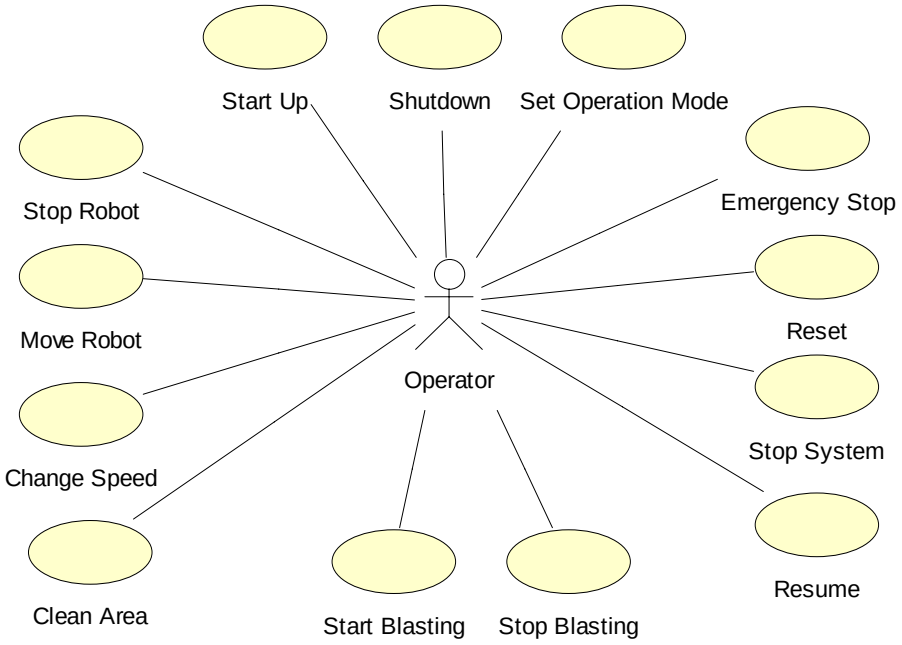


Fig. 7.5.- Casos de uso básicos del sistema

Al añadir la funcionalidad a la que se puede acceder con una interfaz de usuario más compleja (PDA, panel de operador portátil o estación de teleoperación), aparecen nuevos casos de uso relacionados con la configuración del sistema, realización de operaciones más complejas, etc. Hay suficientes casos de uso como para agruparlos en una jerarquía de *paquetes de casos de uso,* como se muestra en la **Fig. 7.6**. Se muestran en el mismo diagrama y se describen en **Tabla 7.1** los diferentes actores que aparecen en el sistema. La especificación de los casos de uso de mayor detalle se puede consultar en el Anexo III y sobre todo en el documento de especificación de requisitos del proyecto GOYA [GOYA98b].

7.2.4.2. Escenarios de calidad

Al igual que los casos de uso hacen concretos los requisitos funcionales abstractos, los escenarios de calidad hacen concretas las *calidades* abstractas de la arquitectura. De los atributos de calidad de la arquitectura que se pueden ver en el Anexo II, se pueden concretar, para el caso del GOYA, algunos escenarios de calidad clasificados según el atributo al que atañen.

Referidos a la modificabilidad

- Añadir una nueva interfaz de usuario (p.ejem. una PDA)
- Sustituir las tareas que realizan el control por tarjetas controladoras de motores o control distribuido. Integración de otros componentes *hardware*.
- Cambiar modo de operación, o interacción entre los dos modos
- Reutilizar los componentes del GOYA, ampliar a nuevos componentes
- Cambiar la configuración mecánica. Añadir un modelo de cinemática.
- Añadir un manipulador que soporte la herramienta
- Añadir nuevas misiones
- Cambiar parámetros del movimiento, límites, velocidades, tipos de motores, etc.

Relacionados con el rendimiento

- Cambiar la asignación de tareas críticas/no-críticas
- Estudio de la planificabilidad del sistema ante diferentes escenarios operativos

Relacionados con la seguridad y mantenibilidad

- Reacción del sistema ante diferentes tipos de errores
- Procesamiento de distintas alarmas

Relacionados con la interoperabilidad

- Comunicación con otros robots
- Comunicación con diversas interfaces de operador

Operadores		Sistemas		Otros
Local	Accede directamente al equipo con una botonera o PDA, normalmente a una funcionalidad básica.	Teleoperación - Monitorización	Sistema remoto que permite monitorizar el proceso o incluso manejar el equipo.	Robot
Experto – Remoto	Puede acceder a la funcionalidad completa del equipo.	S. de visión	Proporciona imágenes al operador de zonas de difícil acceso.	Herramienta
Administrador del sistema	Accede a opciones de configuración más avanzadas	Posicionador primario	Vehículo o grúa normalmente comercial que posiciona al robot, como elemento secundario.	

Tabla 7.1.- Actores en el diagrama de casos de uso

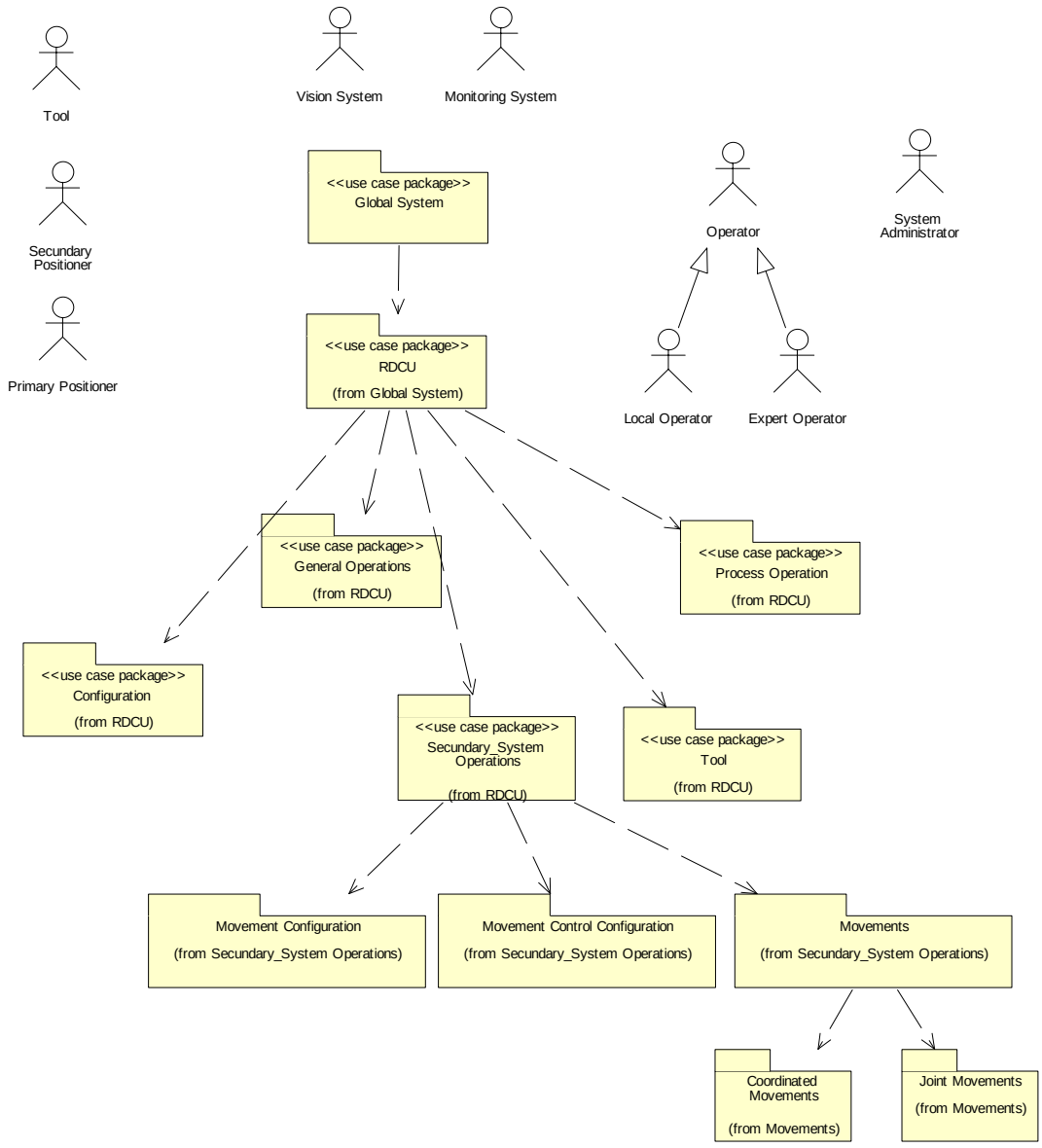


Fig. 7.6.- Jerarquía de casos de uso en el sistema GOYA

7.2.5. Modelo de análisis del sistema

Las características del sistema GOYA explicadas en los puntos anteriores se plasman en diagramas UML en el modelo de análisis del problema, para aprovechar las ventajas vistas en el Capítulo 3.

7.2.5.1. Modelo estático del sistema

Gracias al diagrama conceptual estático de la **Fig. 7.7**, se obtiene una visión general de los distintos sistemas que hay en el GOYA y las relaciones entre ellas. Este diagrama delimita el alcance del sistema detallando el tipo de sistemas usuarios y dispositivos con los que interacciona.

Es de destacar la estereotipación de cada una de las clases para indicar el tipo de participación que tiene en el sistema. Se siguen las recomendaciones de COMET [Gomaa00] para este diagrama y para el resto de diagramas UML del capítulo.

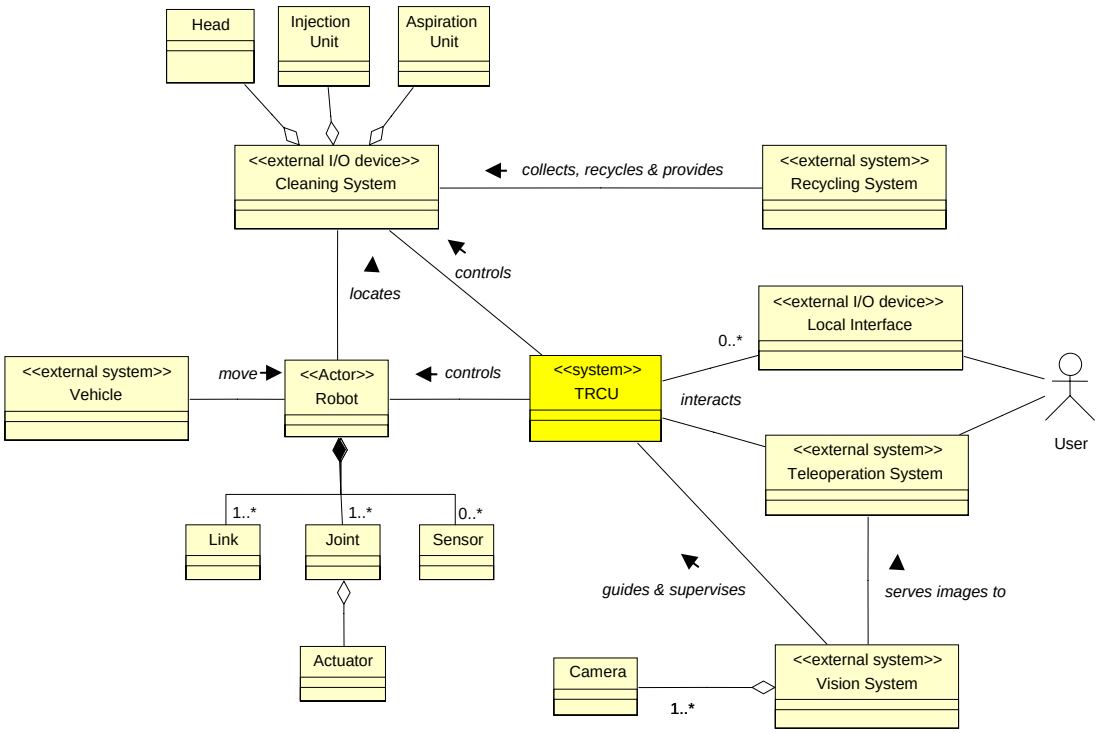


Fig. 7.7.- Modelado estático del dominio del problema. TRCU es la unidad de control (*Teleoperated Robot Control Unit*)

7.2.5.2. Modelo dinámico general del sistema

El comportamiento global del sistema se puede representar con un diagrama de estado como el de la **Fig. 7.8**. En el mismo se puede observar como además de los estados de arranque y parada del sistema (Initializing y Sutting down) existen otros tres estados principales: Estados de Error, Operacional y Configurando (Error, Operational y Configuring). Desde cualquier estado se puede pasar al estado de Error, donde se debe solucionar el mismo, y volver al estado original. El estado Operacional del sistema es precisamente aquel que el sistema puede realizar la labor para la que ha sido diseñado.

El estado operacional se puede descomponer en los subestados que se muestran en la **Fig. 7.9**. Como se observa, es un estado concurrente que distingue el **movimiento** del **proceso de herramienta** (en este caso *Blasting* – chorreado). El sistema puede encontrarse en movimiento o parado y en ambos estados puede estar chorreando o no estarlo. Se han incluido unas transiciones que invocan alarmas;

por ejemplo, si el sistema se encuentra parado y chorreando durante un largo periodo, debe generarse una alarma y parar de chorrear para evitar una erosión excesiva del casco (After(elapsed time)/Stop blasting, alarm [in Blasting]).

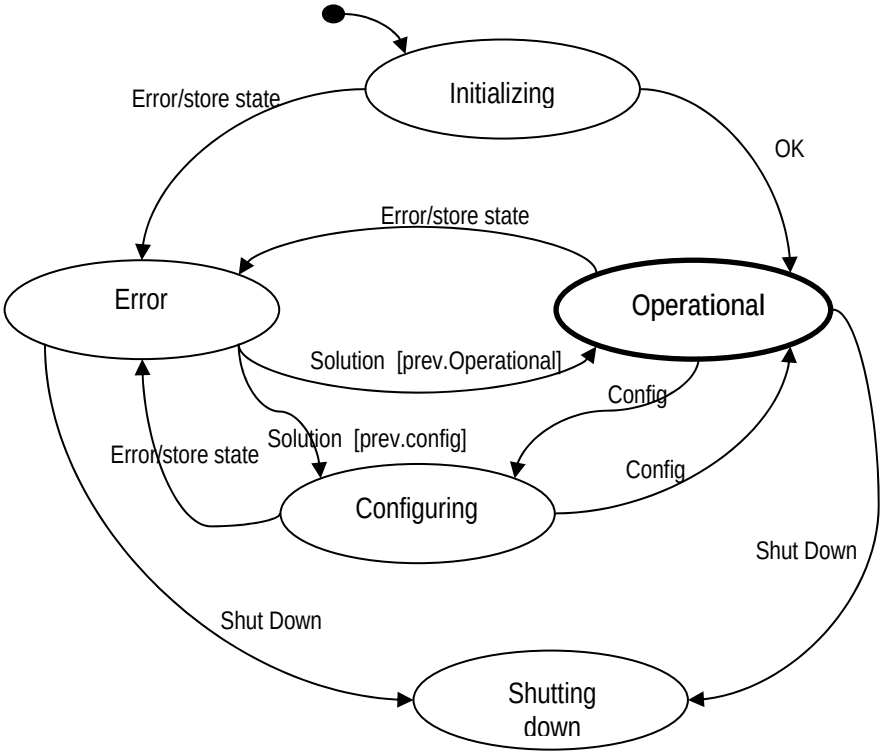


Fig. 7.8.- Diagrama de estado general del sistema

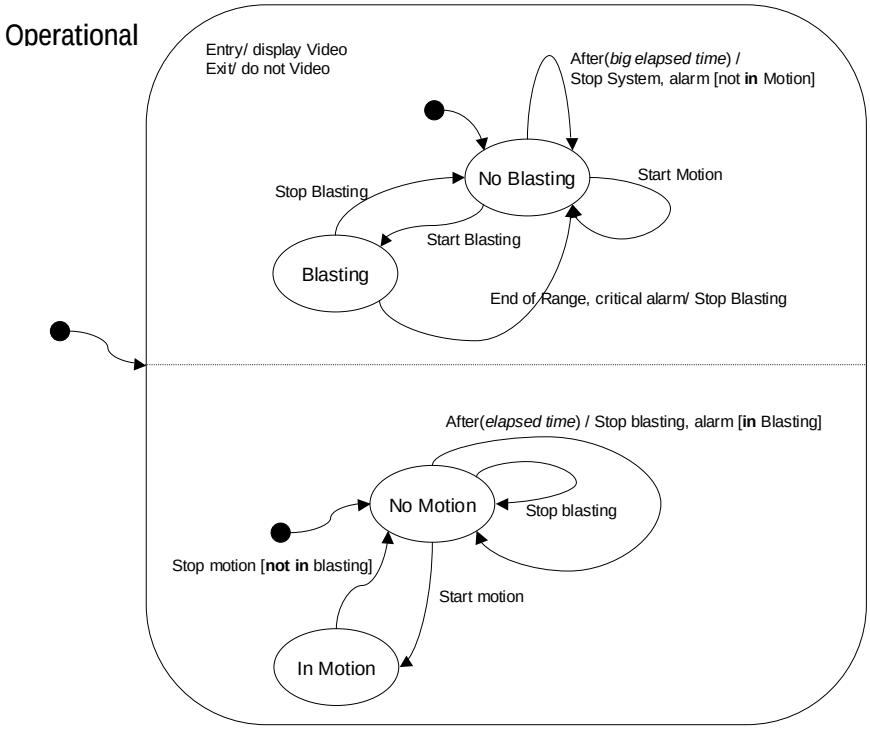


Fig. 7.9.- Diagrama de sub-estados del estado *Operational* de la Fig. 7.8

El análisis descrito en esta sección es suficiente para tener una idea general, pero precisa, de las características del GOYA. No obstante, el lector puede encontrar más detalles en el Anexo III y en los documentos de requisitos y diseño detallado del sistema en [GOYA98a] [GOYA98b]. En el siguiente punto se explica la instanciación de la arquitectura ACROSET en el diseño de la unidad de control del prototipo GOYA.

7.3. Utilización de la arquitectura ACROSET en el sistema GOYA

El análisis del sistema realizado en el punto anterior, particularmente los requisitos funcionales y de calidad concretos, modelan la instanciación de ACROSET para este sistema particular, condicionando la cardinalidad de SUCs, MUCs y RUCs, así como los puertos de estos componentes que, junto con sus interfaces, dotan de funcionalidad al CCAS.

La vista conceptual de esta instanciación de ACROSET se describe en el siguiente punto; en el resto de apartados, se detalla la implementación de la arquitectura instanciada para el sistema GOYA, gracias a la vista de módulos, de ejecución y de código. Estas vistas siguen la filosofía de 4-V.H., pero con unas ligeras variaciones, sobre todo en la vista de módulos: los componentes, puertos y conectores se traducen, no a módulos generales, sino a clases, objetos y relaciones entre los mismos, ya que se ha optado por utilizar la orientación a objetos para esta implementación de la arquitectura obtenida a partir de ACROSET.

7.3.1. Vista conceptual de la arquitectura en el sistema GOYA

En la unidad de control del robot GOYA aparecen todos los subsistemas definidos en ACROSET (UIS, IS, CCAS, SMCS – ver **Fig. 7.11**). En los siguientes puntos se muestran los componentes que integran cada uno de estos subsistemas, explicando las razones que justifican dicha instanciación. Las funciones principales que deben realizar estos subsistemas están resumidas en la **Tabla 6.1** del capítulo anterior.

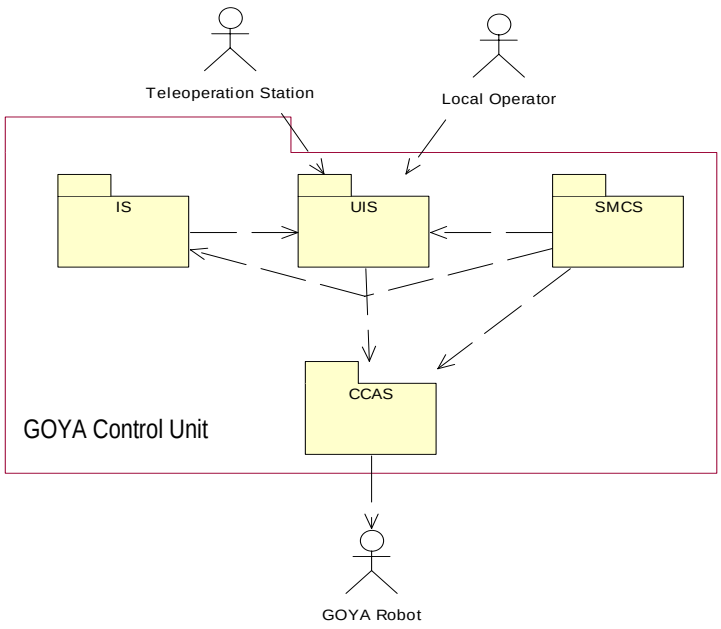


Fig. 7.11.- Subsistemas de ACROSET en el sistema GOYA

7.3.1.1. CCAS en GOYA

El *Subsistema de Coordinación, Control y Abstracción del Hardware* para el sistema GOYA se puede ver en la **Fig. 7.12**. Puesto que el sistema GOYA no cuenta con servomotores controlados por tarjetas o *drivers* comerciales por el hecho de tratarse de un sistema sencillo con control de motores *todo/nada*, se ha optado por implementar el control de los motores directamente en la unidad de control. Por tanto, los únicos componentes *hardware* del sistema son los sensores y actuadores, que serán modelados como componentes de abstracción del *hardware* en la arquitectura del GOYA.

Los sensores modelados son de tres tipos:

- **Encoders** (Encoder en **Fig. 7.12**) Proporcionan una salida digital cuya frecuencia equivale a la velocidad del eje al que están acoplados. El número de pulsos contados a partir de una referencia proporciona la posición relativa del eje. En el sistema hay uno para cada eje.
- **Finales de carrera** (EoS[10] en **Fig. 7.12**) Contactos físicos con salida digital todo/nada que marcan el final del recorrido de cada eje en ambos sentidos. En el sistema hay dos para cada eje. También existen otros dos que se activan cuando el cabezal de limpieza está posicionado correctamente sobre el casco del barco (HeadPsEos[11] en **Fig. 7.12**). En la **Fig. 7.13** se puede apreciar la existencia de otros dos finales de carrera (SecEoS) que actúan como elementos de seguridad en prevención de choque de algún elemento móvil con obstáculos o personas.
- **Sensor de presión** (Press en **Fig. 7.13**) Sensor de salida analógica que indica la presión del chorro.

Los actuadores modelados son de dos tipos:

- **Electroválvula**. (EValve en **Fig. 7.13**) Accionamiento electro-neumático todo/nada que permite la activación y desactivación de la herramienta de chorreado.
- **Motor asíncrono con velocidad regulable por variador de frecuencia**. (FVCMotor en **Fig. 7.12**). Desde el componente de abstracción del *hardware* FVCMotor se accede a un variador de frecuencia que a su vez es el encargado de accionar al motor asíncrono. Este componente FVCMotor es más complejo que otros componentes tipo Actuator puesto que, al estar relacionado con el variador en lugar de con el motor directamente, admite una parametrización de ciertos aspectos del variador, así como la recepción de información del mismo (por eso tiene un puerto de datos).

Los componentes de abstracción del *hardware* (Sensorized_Joint y Sensor) tienen puertos de control y datos que los comunican con los componentes superiores, pero su información interna se actualiza por acceso directo a los drivers de las tarjetas de entrada/salida.

Con objeto de controlar los tres ejes del sistema (Sensorized_Joint), compuesto cada uno por un motor, un encoder y dos finales de carrera, se deben incorporar tres componentes tipo Motor_SUC. Cada SUC controla un eje según su estrategia de control (*todo/nada, PID*) que puede ser intercambiada (patrón estrategia o política), según la información que recibe de los sensores por sus puertos de entrada SensorData_IP (uno por sensor) y las órdenes que recibe de las capas superiores por su único puerto sucControl_IP.

La acción conjunta de los tres SUCs controladores de los ejes del GOYA está coordinada por un componente Coordinator según su estrategia de coordinación, que al igual que pasa con los SUCs, puede ser intercambiada a través del puerto de configuración config_IP. También evalúa si la acción requerida puede ser realizada según el diagrama de estado del MUC. El conjunto Coordinador y SUCs son los componentes internos de un controlador del mecanismo MUC *"Plataforma de Elevación XYZ"* (XYZ_Crane en **Fig. 7.12**).

---

[10] Del inglés *End of Stroke.*

[11] Del inglés *Head Position End of Stroke.*

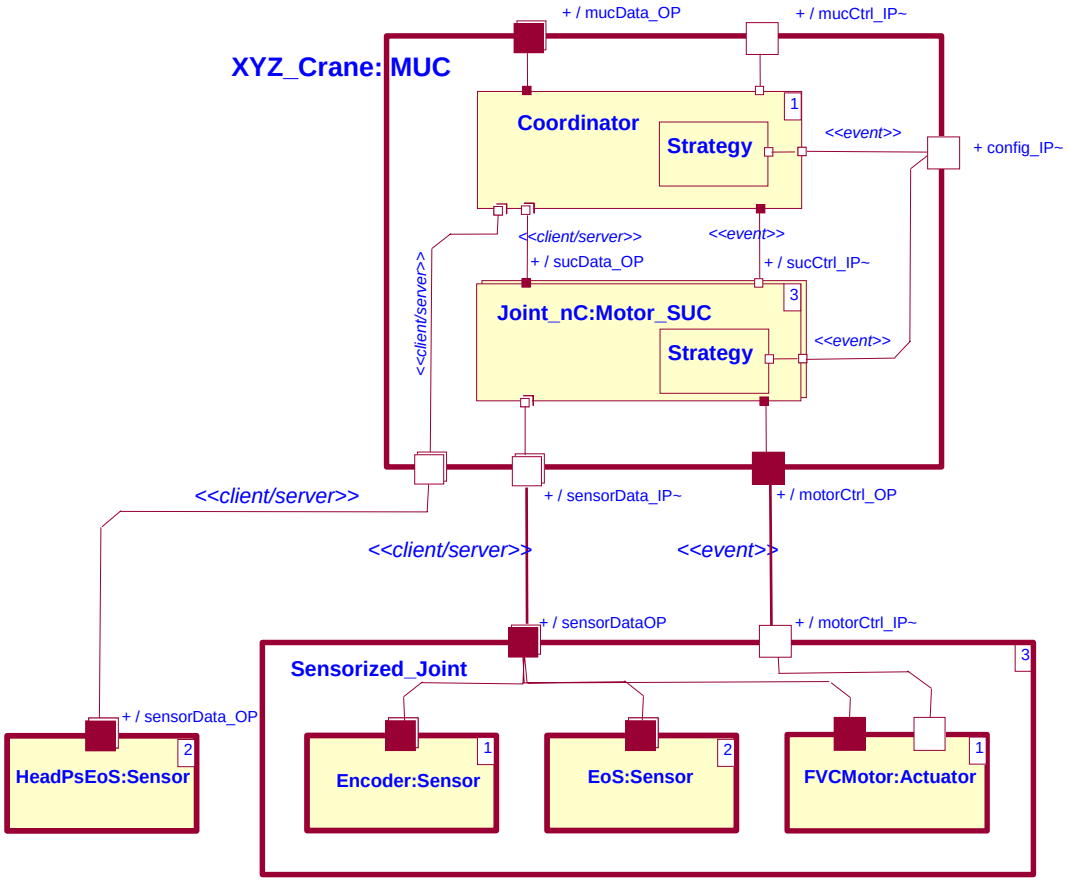


Fig. 7.12.- *MUC* controlador de ejes y *Sensorized_Joint* como abstracción del *hardware* en GOYA

La estrategia de coordinación del MUC depende de la configuración del mecanismo. En el caso del GOYA, nos encontramos con un robot cartesiano, cuya cinemática inversa es trivial. La coordinación es muy sencilla y se limita, por lo general, al ajuste de las velocidades de los motores para llegar al mismo tiempo a un punto. En el caso de un robot con cinemática inversa más compleja habría que realizar los cálculos cinemáticos para lograr traducir los giros de los motores en desplazamientos en las coordenadas XYZ.

La siguiente capa de granularidad la ofrece el controlador del robot completo RUC. Si se une la funcionalidad ofrecida por el MUC XYZ_Crane, con un controlador de la herramienta y se coordinan sus acciones mediante un coordinador de robot, estará recogida toda la funcionalidad del robot en un solo componente de control: el RUC (ver **Fig. 7.13**).

El coordinador del RUC redirige los comandos del operador hacia MUCs y SUCs, cuando no deba tratarlos él mismo, y también genera los comandos de control para el MUC y el SUC de la herramienta que coordina. La herramienta consiste en una boquilla de chorreado que proyecta granalla a alta presión cuando la electroválvula (eValve) está abierta. El control de esta electroválvula es realizado por el SUC Tool, que también recibe información de la presión de granallado. El proceso de coordinación consistirá principalmente en realizar tareas de limpieza que implican desplazamientos y accionamientos de la herramienta simultáneas y sincronizados. El coordinador se encarga también de evaluar el diagrama de estado del RUC para ver si puede realizar la tarea requerida.

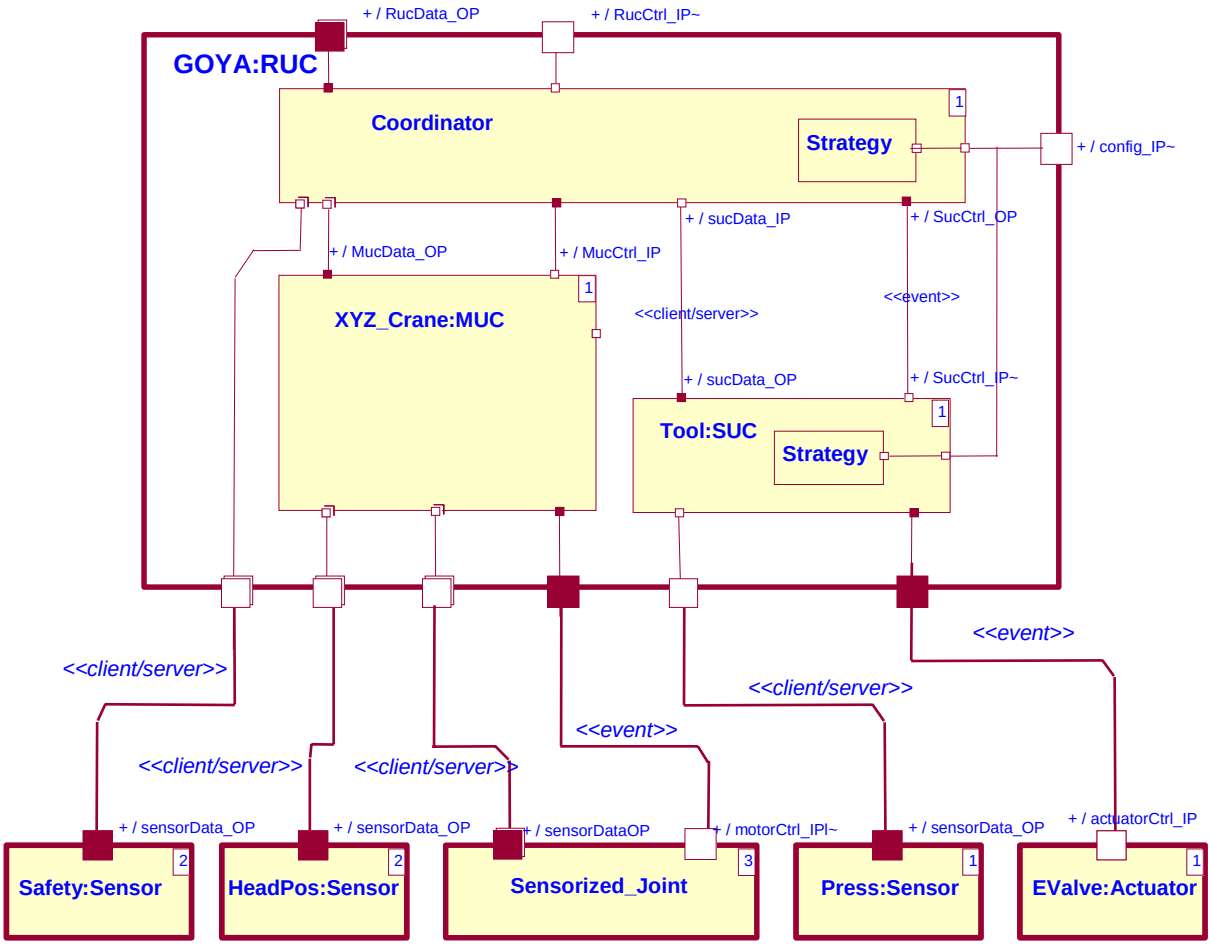


Fig. 7.13.- Componentes del CCAS en el sistema GOYA

Se puede observar en la **Fig. 7.12** que se han cambiado el estereotipo de los conectores de ACROSET (<<*data*>> y <<*control*>>) por el modo de comunicación que se realizará en el sistema GOYA. La comunicación entre los puertos de datos será del tipo *cliente/servidor*, sin especificar el tipo concreto, dado que al tratarse de un prototipo, se pretende cambiar el modo de comunicación durante la vida del mismo, incluso es posible que el sistema se distribuya. El patrón de comunicación podrá ser del tipo *Observador*, *Proxy* o *Broker*, según el tipo de distribución. En cualquier caso, se producirá una notificación hacia las capas superiores cuando haya un cambio significativo de los datos. Las acciones de control son del tipo *evento*, generado cuando sea necesario realizar la acción. En la vista de ejecución se concretará el tipo de comunicación utilizado para esta implementación.

7.3.1.2. UIS en GOYA

El *Subsistema Intérprete de Usuario* deberá realizar la interfaz, interpretar y coordinar los mensajes provenientes de tres tipos de usuarios de la funcionalidad proporcionada por el CCAS (ver **Fig. 7.14**):

- Un operador remoto que accede a la UC[12] mediante la estación de teleoperación.
- Un operador local que accede a la UC mediante una botonera electro-mecánica.
- El subsistema de inteligencia IS, que se ejecuta en la propia UC, pero accede al CCAS a través del UIS.

[12] Unidad de Control.

La multiplicidad 3 en el componente User_Interface de la **Fig. 7.14** indica precisamente la existencia de estos tres tipos de usuarios de la funcionalidad del sistema. Cada uno de estos componentes genera tres tipos de mensajes (eventos) para el sistema que se agrupan en tres tipos de puertos de salida:

- ✓ rucCtrl_OP: comandos de control del RUC
- ✓ config_OP: comandos de configuración del sistema (en el caso del GOYA, este puerto se conecta con config_IP del SMCS, que a su vez gestiona toda la configuración del sistema)
- ✓ intelCtrl_OP: comandos de control de los usuarios hacia el Subsistema de Inteligencia (IS).

ACROSET no permite que varios usuarios den órdenes a la vez a un mismo componente (aunque un mismo componente sí puede suministrar datos a varios componentes a la vez). Para solucionar esta situación, ACROSET propone los componentes Arbitrator, encargados de arbitrar entre las órdenes provenientes de varias fuentes y proporcionar una orden única al componente de destino. El Arbitrator puede tener diferentes estrategias de arbitraje o de combinación de comandos, que pueden intercambiarse[13]. En el caso del GOYA, Arbitrator funciona como un conmutador, según el modo de control en el que se encuentre el sistema (control por operador local, control por operador remoto, control autónomo) rechaza las órdenes provenientes del usuario que en ese momento no tenga el control. La única orden que es aceptada en cualquier modo es la "parada de emergencia". No hay ningún inconveniente en que la información del sistema se sirva a todos los usuarios independientemente de que tengan o no el control. Dicha información se recoge a través de los puertos de datos rucData_IP, que recibe datos del robot (del RUC), y mngData_IP, que recibe datos de los componentes de gestión y configuración.

Se ha incorporado un componente de gestión de modo (Mode_Manager), para almacenar el modo de control actual y el diagrama de estado de los posibles modos de control, con objeto de gestionar el cambio de modo sólo si es posible.

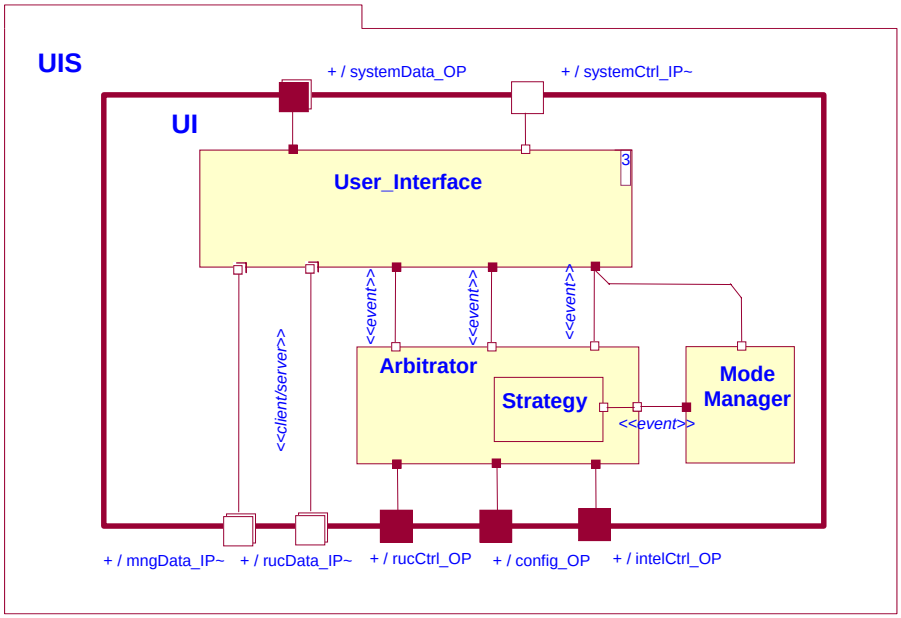


Fig. 7.14.- Subsistema Intérprete de Usuarios en GOYA

[13] Por ejemplo, un modo de arbitraje puede ser simplemente negar las órdenes del usuario con menor prioridad, mientras que otro modo puede ser combinar las órdenes según la prioridad de cada comando individual.

7.3.1.3. IS en GOYA

El *Subsistema de Inteligencia*, en el caso del GOYA, simplemente se encarga de ordenar la ejecución de las secuencias automáticas almacenadas (Sequences). El subsistema consta de un único componente SeqExec[14] del tipo Programmed_Inteligence (**Fig. 7.15**). En posteriores ampliaciones del prototipo GOYA se podrían incorporar otro tipo de componentes que proporcionen inteligencia autónoma al sistema, por ejemplo, componentes reactivos al estilo de una arquitectura basada en comportamiento (evitar obstáculos, seguir trayectorias, etc.) También podrían incluirse componentes de planificación o de inteligencia programada. Todos estos componentes aportan comportamientos inteligentes de alto nivel que pueden acceder incluso a los componentes en el CCAS a más bajo nivel (directamente sobre sensores y actuadores). En estos casos habría que incorporar árbitros (Arbitrators) a la entrada de los puertos de control de sensores y actuadores para que pudiera realizarse una combinación adecuada de comportamientos autónomos y dirigidos por los controladores SUCs y MUC.

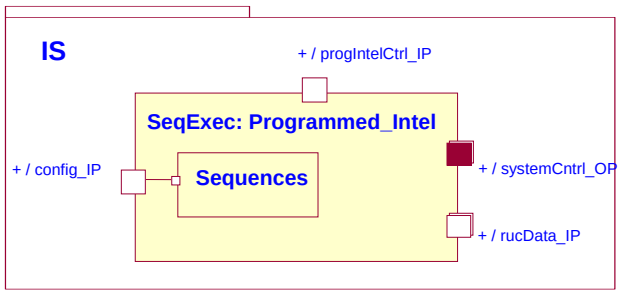


Fig. 7.15.- Subsistema de Inteligencia en GOYA

El componente SeqExec genera comandos de control del sistema y los envía a través del puerto systemCntrl_OP de forma que entran al UIS como cualquier otro usuario del sistema. La realimentación de los datos del RUC necesarios para realizar su control entran por el puerto rucDataIP. Estos puertos conectan al IS con la interfaz de usuario correspondiente en el UIS. SeqExec contiene uno o varios ficheros de secuencias almacenadas (Sequences), que pueden ser modificadas, eliminadas o añadidas por el operador (a través del puerto config_IP). Ejemplos de secuencias son: *"limpiar paño completo"* y *"proceso de calibración automático"*. El operador humano no tiene por qué ser el único que proporcione nuevas secuencias de movimiento a este componente, por ejemplo, en el punto **7.4** se explicará cómo en el proyecto EFTCoR un sistema de visión externo analiza la superficie a limpiar y genera un fichero de secuencia con los puntos a los que tiene que desplazarse la herramienta. El puerto progIntCtrl_IP ofrece una interfaz de control de la inteligencia, para arrancar la ejecución de una secuencia, pararla, recoger información del proceso de ejecución, etc.

7.3.1.4. SMCS en GOYA

El *Subsistema de Seguridad, Gestión y Configuración* consta de los componentes que se pueden ver en la **Fig. 7.16**. Estos componentes deben realizar fundamentalmente las tareas de:

- Construcción de la aplicación al arranque, por ejemplo, crear clases, objetos y tareas, conectar puertos, etc. (App_Builder dentro del Application_Manager)
- Controlar el funcionamiento de la aplicación como tal en su conjunto (Application_Manager).
- Controlar el funcionamiento seguro de la aplicación (Safety_Manager)
- Cambiar aspectos básicos de la configuración del sistema (Config_Manager).

[14] Del inglés *Sequence Executor*

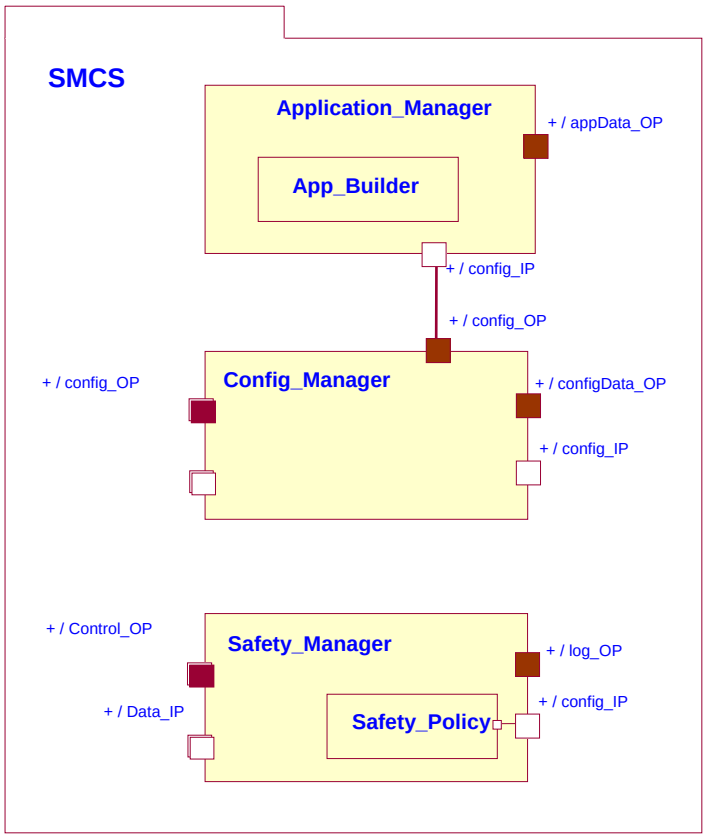


Fig. 7.16.- Subsistema de Seguridad, Gestión y Configuración en GOYA

El componente Application_Manager almacena el diagrama de estado de la aplicación global, y gestiona además de la evolución de éste, los procesos de arranque y finalización de la aplicación gracias a App_Builder. No tiene puertos de acceso al resto de componentes porque su función es actuar de constructor de toda la aplicación. Tiene un puerto de entrada config_IP que se relaciona con el Config_Manager para que cuando el usuario decida realizar cambios de configuración en tiempo de ejecución (conexión y desconexión de puertos), pueda acceder a las referencias de dichos puertos, que sólo están disponibles para Application_Manager. El puerto de salida appData_OP puede proporcionar información del estado de la aplicación.

El componente Config_Manager tiene otros puertos de salida de configuración config_OP, conectados a todos los puertos config_IP del resto de componentes. Cuando un usuario del sistema quiera cambiar la configuración de cualquier componente deberá realizar la petición a Config_Manager, que se encargará de gestionar el cambio, a través de su puerto config_IP. Durante la operación, este componente recibirá datos del sistema a través de su puerto rucData_IP y remitirá al usuario información sobre el proceso a través de configData_OP.

El componente Safety_Manager en el sistema GOYA recibe las alarmas y errores del sistema así como datos periódicos de su funcionamiento a través de los puertos Data_IP y los gestiona según la política de seguridad activa en ese momento. El cambio de política de seguridad (Safety_Policy) se realiza a través del puerto config_IP. El componente realiza las acciones de control necesarias sobre los componentes a través de sus puertos de control (Control_OP) y reporta dichas acciones al usuario a través de log_OP.

7.3.2. Vista de módulos

Para la implementación de la arquitectura de control del sistema GOYA se ha optado por un control *software* completo ejecutándose en un PC-Industrial con Sistema Operativo RT-Linux [Barabanov97]. Para programar el *software* de control se ha utilizando el lenguaje Ada95 [Ada95]. A esta decisión se llega principalmente por los siguientes factores:

- Por ser GOYA un sistema mecatrónico diseñado especialmente para la aplicación y sobre todo, por tratarse de un prototipo, es fundamental que el sistema de control sea muy flexible y fácilmente modificable. También por tratarse de un prototipo, se puede experimentar también con distintas distribuciones de tareas a realizar en la estación de teleoperación y la unidad de control. ACROSET, por su diseño, permite distribuir fácilmente sus componentes en distintos nodos de computación. El contar con el anexo para sistemas distribuidos GLADE [Pautet98] de Ada95, facilita también la distribución [PFC00], [PFC02].
- Los controladores de los motores se implementan en *software*, integrados con el resto de la aplicación, porque el control de posición de los ejes no necesita una precisión elevada, no tiene requisitos temporales muy estrictos: los motores a controlar son motores asíncronos industriales, cuya velocidad se regula con un variador de frecuencia *Schneider Altivar 11*; además las velocidades de los ejes son bajas y la respuesta ante una petición de parada es rápida.

La vista de módulos tal como se plantea en [Hofmeister00] queda demasiado "abierta" e imprecisa, es un poco confusa en la traducción y además no permite una traducción directa a clases. En general, se ha encontrado el mismo problema de traducción entre la vista conceptual de una arquitectura y las vistas próximas a la implementación en los métodos de desarrollo orientados a la arquitectura revisados (ABD, 4-V.H., PPOA [Fernández04]).

Al tratarse de una implementación *software* de ACROSET, hubo que decidir que orientación se daba en la traducción de componentes a módulos. Se optó por una orientación a objetos por las ventajas que ofrece por sí misma (mencionadas en el Capítulo 3 y demostradas en el dominio de los robots teleoperados entre otros trabajos, por la arquitectura CLARAty [Nesnas03]) y por la falta de soporte directo de componentes en los lenguajes de programación existentes. El recientemente publicado UML 2.0. confirma lo adecuado de traducir componentes, puertos y conectores en clases, objetos y relaciones entre esas clases.

7.3.2.1. CCAS en GOYA

Siguiendo el mismo esquema de explicación que el del punto **7.3.1**, se exponen a continuación los módulos (clases y objetos) que aparecen en la aplicación de ACROSET para el sistema GOYA, comenzando por el CCAS. En la **Fig. 7.17** se pueden observar conjuntamente todos los objetos del CCAS con sus interrelaciones principales. En las siguientes figuras se exponen las interfaces de las clases a las que pertenecen, incluidos los puertos.

La clase Motor_SUC contiene los puertos que aparecen en la figura con estereotipos <<*InPort*>> y <<*OutPort*>>, como puertos de entrada y salida, tanto de datos (Data) como de control (Ctrl) y configuración (Config). Los puertos tienen una relación de composición con la clase que los contienen, puesto que no tiene sentido su existencia individual: los puertos pertenecen al componente y se crean y se destruyen con él. Las operaciones de los puertos de control corresponden a los eventos (event) que otros componentes pueden enviar al componente SUC. Los puertos de datos son genéricos, y tienen la misma interfaz para cualquier componente, el tipo de dato se define y concreta dependiendo del tipo de componente. Además de los puertos, la clase Motor_SUC contiene otros objetos que almacenan datos: Alarm, Error, State y una objeto del tipo Control_Strategy que contiene el algoritmo de control que realiza el SUC. Esta estrategia de control sigue el patrón de diseño "estrategia" o "política" con objeto de que pueda ser dinámicamente intercambiada manteniendo la misma interfaz para el objeto que la utiliza, en este caso, el controlador Motor_SUC.

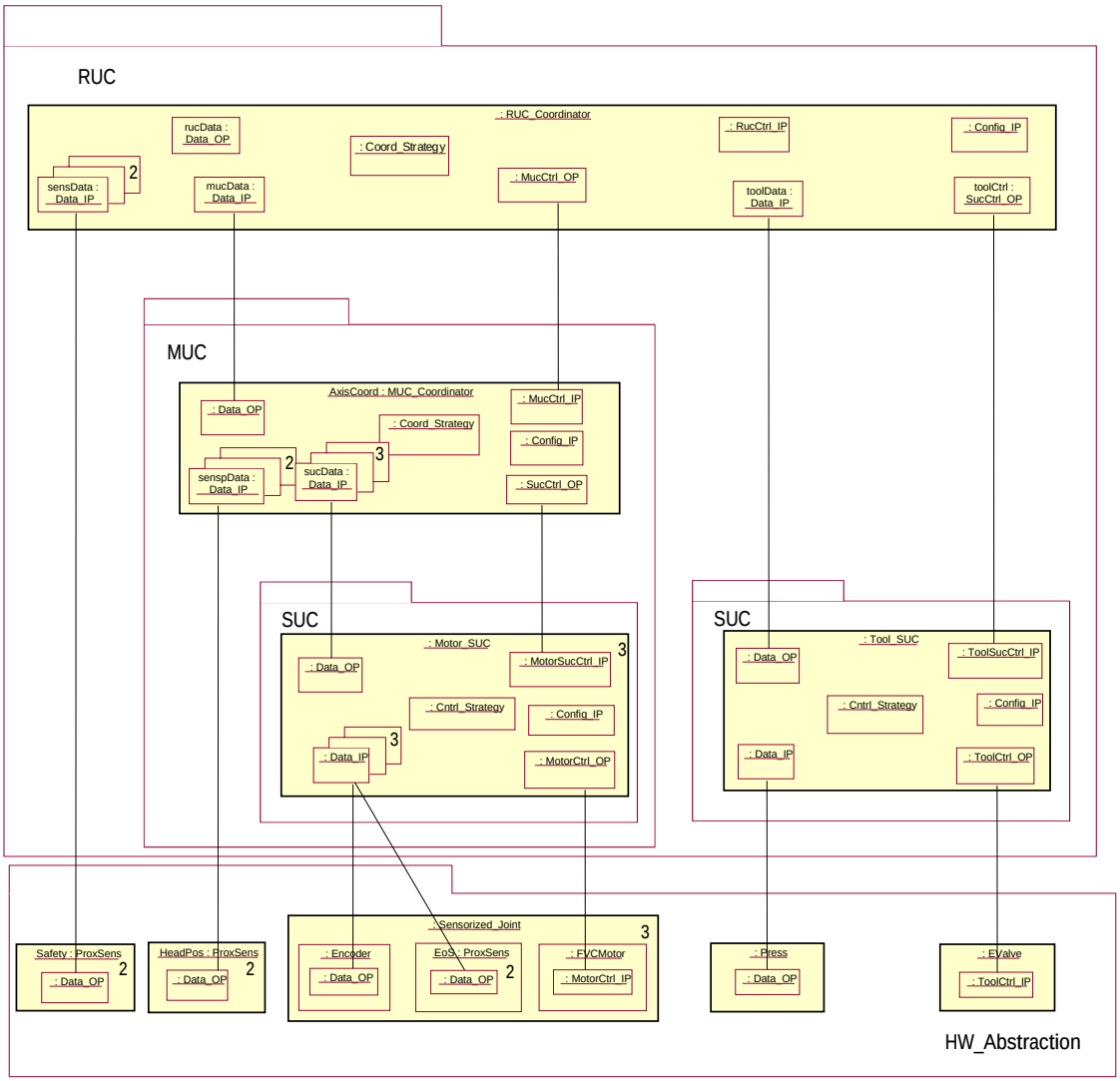


Fig. 7.17.- Vista de módulos: Todos los objetos del CCAS

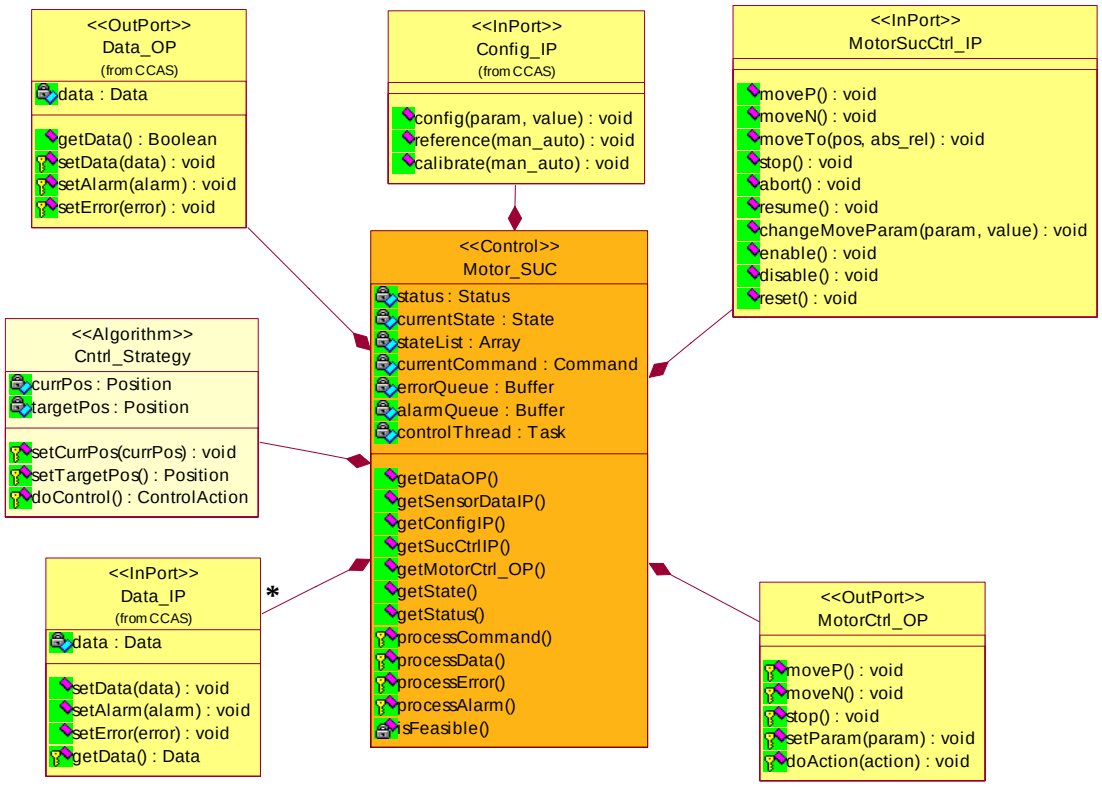


Fig. 7.18.- Motor_SUC: Controlador de un motor eléctrico con variador de frecuencia[15]

En el caso de un SUC controlador de herramienta, todas las clases presentadas en la **Fig. 7.18** permanecen con la misma interfaz, excepto los puertos de control, que deben adaptarse a los distintos eventos de control relacionados con la herramienta, como se ve en la **Fig. 7.19**. Las clases Data_IP, Data_OP, Config_IP, Ctrl_Strategy presentan la misma interfaz para todos los componentes instanciados en el CCAS, variando en algunos aspectos la implementación y tomando como genéricos algunos tipos de datos[16].

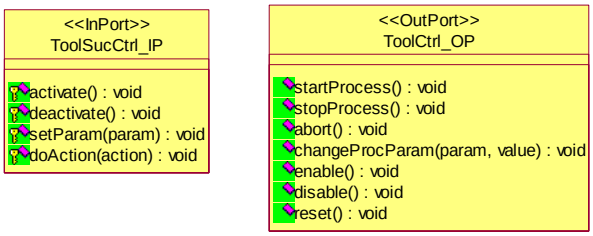


Fig. 7.19.- Tool_SUC: Puertos de control de entrada y salida (herramienta de chorreo)

Los siguientes diagramas representan las clases de los coordinadores del MUC y SUC, que implementan la funcionalidad completa de estos componentes:

---

[15] En todos los diagramas, si no se especifica *multiplicidad* es 1.

[16] Por esta razón, en los diagramas siguientes al de la Fig. 7.18, no se representan los atributos y operaciones de estas clases comunes, sólo los de aquellas clases con métodos distintos y atributos distintos.

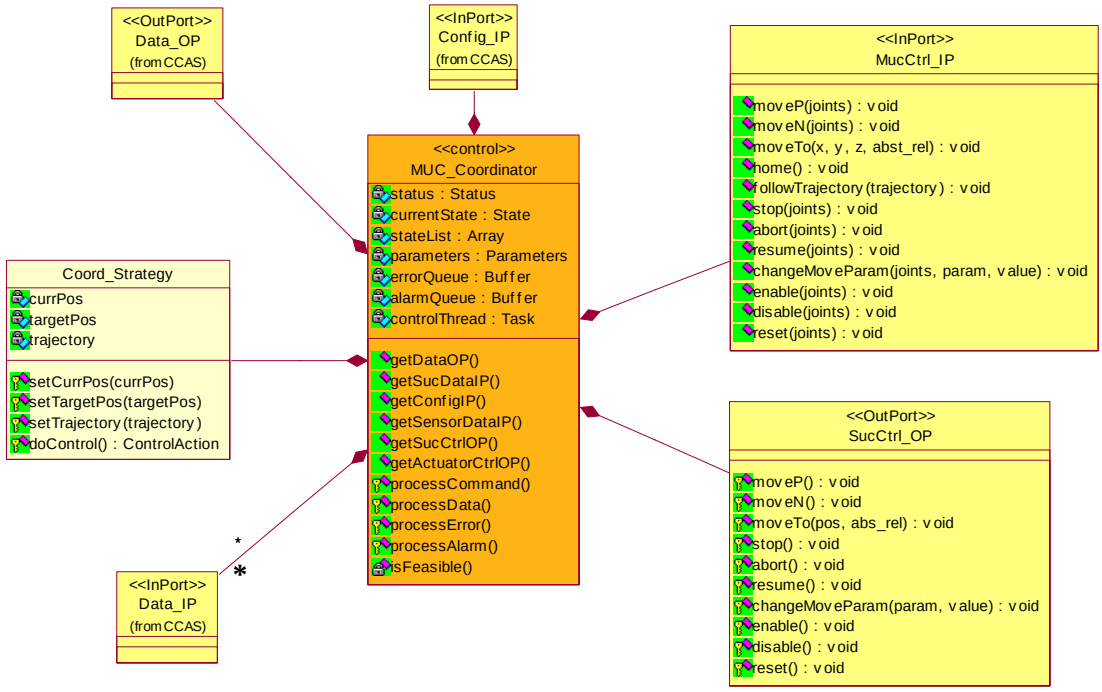


Fig. 7.20.- MUC_Coordinator: Coordinador de un mecanismo articulado

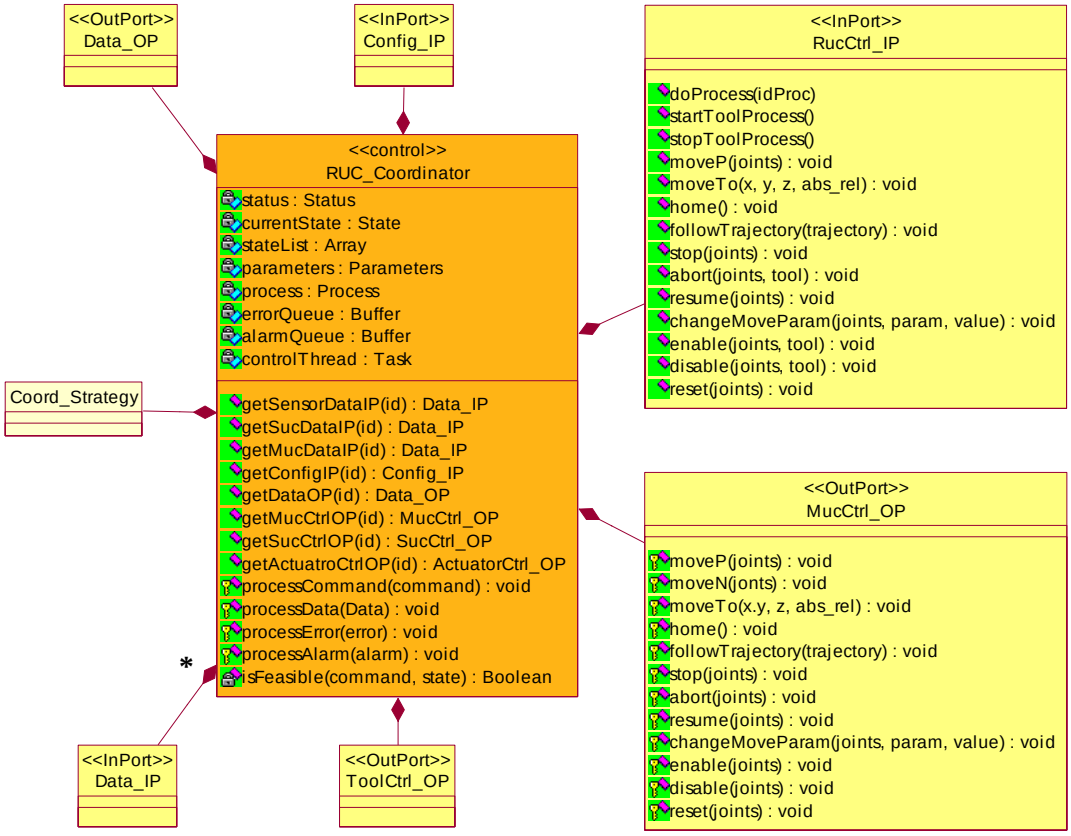


Fig. 7.21.- RUC_Coordinator: Coordinador del GOYA con un mecanismo y una herramienta

Finalmente, las clases "entidad" que aparecen involucradas en el CCAS son las que se pueden ver en la **Fig. 7.22**

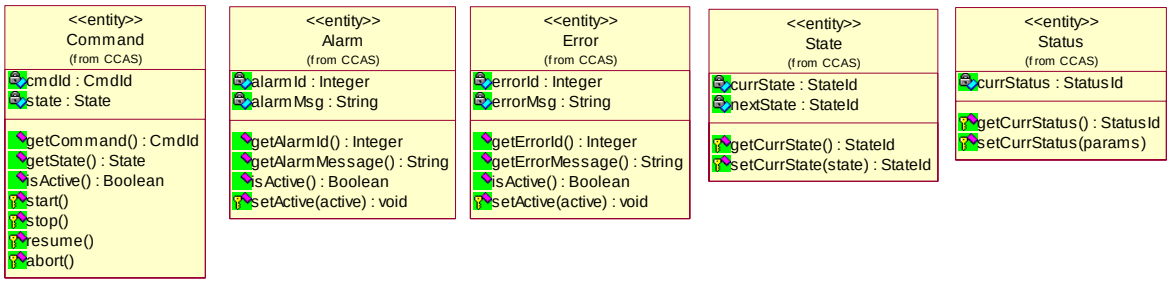


Fig. 7.22.- Clases "entidad"

7.3.2.2. UIS en GOYA

Para la implementación del UIS mostrado en **Fig. 7.14** se ha decidido utilizar un patrón *fachada*[17], puesto que este componente, que realiza la interpretación de todas las órdenes provenientes de los distintos usuarios del sistema, se puede ver como una clase compuesta que agrupa todas las interfaces con los distintos usuarios (**Fig. 7.23**). No se muestran en la figura sus métodos y atributos porque coinciden con los ya definidos en los demás subsistemas, excepto los puertos systemCtrl_IP y systemData_OP. La interfaz de los mismos no es más que una agrupación del resto de puertos, en el caso de acceder al sistema contando con toda la funcionalidad. En el caso de que se acceda con una HMI más simple (p.ej. una botonera), los puertos mencionados incluirán sólo los comandos mínimos permitidos por la funcionalidad de dicha HMI.

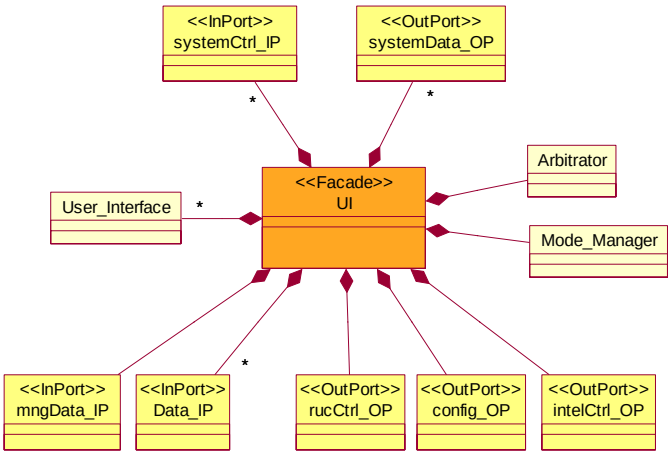


Fig. 7.23.- *Fachada* Intérprete de Usuarios (UI)

Los principales componentes de UI son las interfaces de usuario (User_Interface), que se encargan de traducir los comandos provenientes de elementos tan distintos como un sistema de teleoperación, una interfaz electromecánica o el propio subsistema de inteligencia (IS) a los comandos disponibles en el RUC. También se encarga de interpretar algunos comandos de configuración, gestión, y control de la inteligencia, así como de la recogida de datos de estos componentes.

Hay que resaltar la multiplicidad (*) que aparece en el diagrama de la **Fig. 7.23** por lo que significa en cuanto a la escalabilidad, posibilidad de ampliación a otros sistemas usuarios, modificabilidad, etc. En efecto, la multiplicidad de los componentes User_Interface, systemCtrl_IP y systemData_OP significa

[17] *Facade pattern*

que puede haber *n* usuarios del sistema, cada uno de los cuales podrá acceder a la funcionalidad que defina su User_Interface y puerto de entrada correspondiente (no se podrá acceder a las mismas funciones con una botonera que con una estación de teleoperación compleja). Si en cualquier momento se añade un nuevo usuario al sistema, simplemente debe añadirse una nueva interfaz User_Interface y un nuevo puerto si fuera necesario.

El objeto de arbitraje de la clase Arbitrator cumple el importante cometido de garantizar que **no sea necesario modificar** ninguno de los demás componentes que hay en el sistema independientemente del número de usuarios del mismo (incluidos nuevos componentes de inteligencia), puesto que implementa una estrategia de arbitraje modificable según el modo de control que pueda tener el sistema. El componente Mode_Manager se encarga de gestionar el cambio de modo. De esta forma *n* usuarios del sistema emiten *n* órdenes que se combinan, encolan, o rechazan si es necesario en el Arbitrator para generar **una sóla órden** que se emitirá por el puerto de salida correspondiente, dependiendo de si se está ejecutando un modo de control teleoperado puro, supervisado o compartido o bien totalmente autónomo.

7.3.2.3. IS en GOYA

El *Subsistema de Inteligencia*, en el caso del GOYA, simplemente se encarga de ordenar la ejecución de las secuencias automáticas almacenadas o programadas en su interior. Aunque el ejecutor de secuencias es automático, hace falta que un usuario ordene el comienzo de la secuencia (a través del puerto progIntelCtrl_IP) y que reciba una notificación por el final del la ejecución o cualquier otra incidencia que se pueda producir (puerto data_OP).

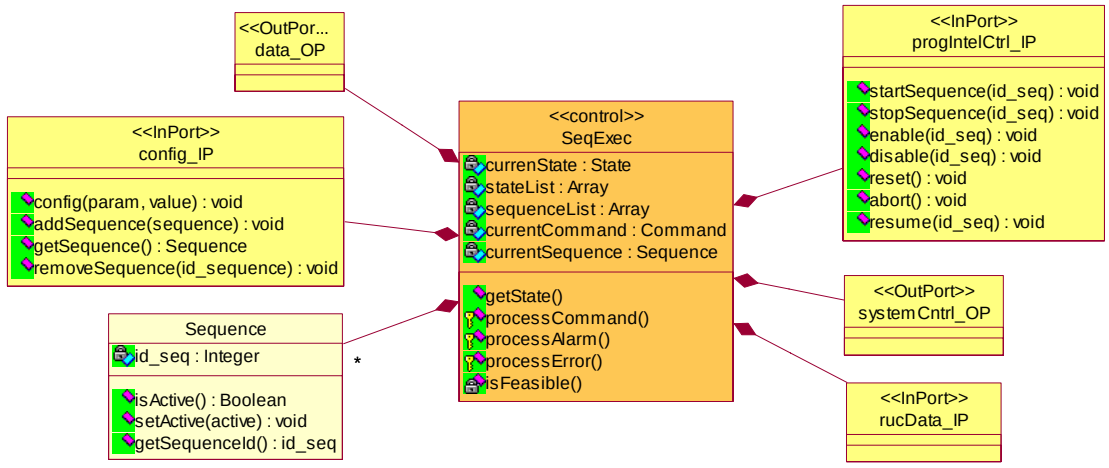


Fig. 7.24.- Subsistema de Inteligencia en GOYA: Un ejecutor de secuencias

7.3.2.4. SMCS en GOYA

El *Subsistema de Seguridad, Gestión yConfiguración* (**Fig. 7.25**) se encarga de gestionar la aplicación, desde el arranque de la misma (incluida la construcción de las clases y tareas mediante App_Builder), hasta el mantenimiento del diagrama de estado general de la aplicación. El componente de configuración (Config_Mngr) ofrece una interfaz a los usuarios del sistema de todas las opciones de configuración que existen en el mismo, mediante su puerto config_IP, les proporciona información del estado de la configuración a través del puerto configData_OP (que obtiene a su vez del sistema mediante rucData_IP).

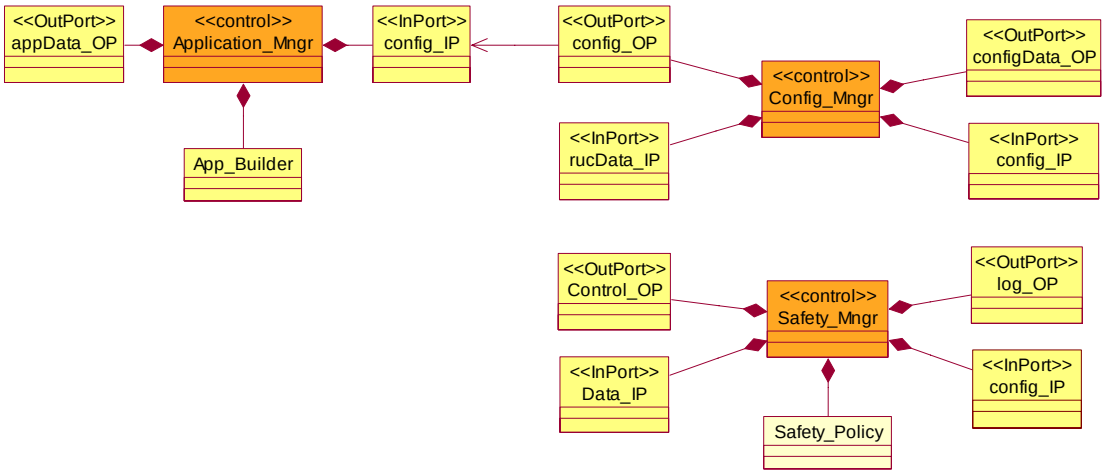


Fig. 7.25.- SMCS en GOYA: Gestores y puertos

La clase encargada del control de la seguridad ante fallos del sistema (Safety_Mngr), incluida la gestión de alarmas y errores, puede seguir varios patrones de seguridad que pueden consultarse en [Douglass00]. Entre ellos destacan los patrones:

- **Monitor-Actuador:** Una parte del sistema actúa, controla, y otra parte monitoriza el entorno para asegurarse que las acciones del actuador son apropiadas. Normalmente necesita sensores adicionales distintos a los presentes en el canal del actuador.
- ***Watchdog* (perro guardián):** Es un componente que recibe mensajes de otro componente de manera periódica. Si el servicio se produce demasiado pronto o tarde o no ocurre, el *watchdog* inicia alguna acción correctiva.
- **Ejecutivo de seguridad:** Este patrón utiliza un coordinador centralizado para monitorizar las acciones de seguridad y controlar la recuperación del sistema ante fallos. El ejecutivo de seguridad actúa como un *watchdog* más inteligente que recoge y coordina todas las acciones de seguridad. De esta forma, se proporciona un punto centralizado y consistente de seguridad y control de fallos para sistemas complejos.

El componente Safety_Mngr se pondría concebir como un *watchdog* que, según se va complicando y ampliando el sistema, tenderá a convertirse en un ejecutivo de seguridad que recoge información periódica del resto de componentes para controlar su funcionamiento seguro. Lo importante de separar esta gestión en un componente distinto estriba en que pueden cambiarse fácilmente los chequeos y políticas de tolerancia a fallos sin necesidad de cambiar a los componentes funcionales del sistema.

7.3.2.5. Diagramas de estado de las clases tipo <<control>>

Las clases definidas como <<*Control*>> son clases que tienen asociado un diagrama de estado. En este punto se presentan los diagramas de estado más significativos, quedando los restantes (muy similares a los aquí presentados) a disposición del lector en [GOYA99]:

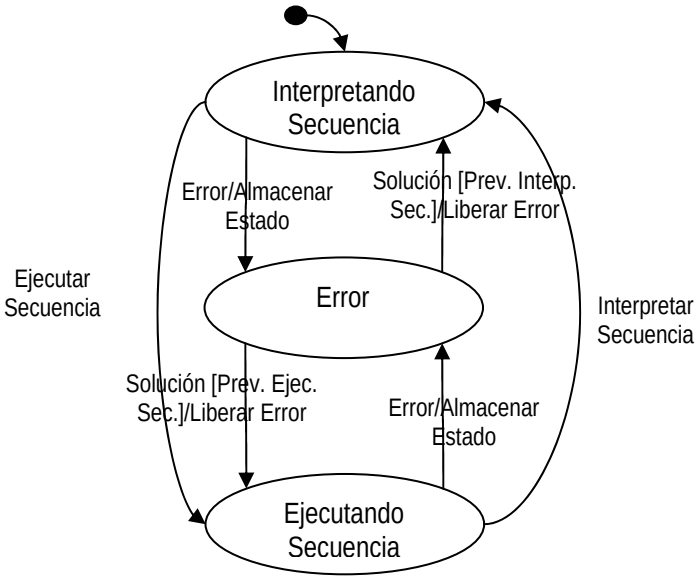


Fig. 7.26.- Diagrama de sub-estados del estado Procesar Secuencia del Sistema Inteligencia (IS).

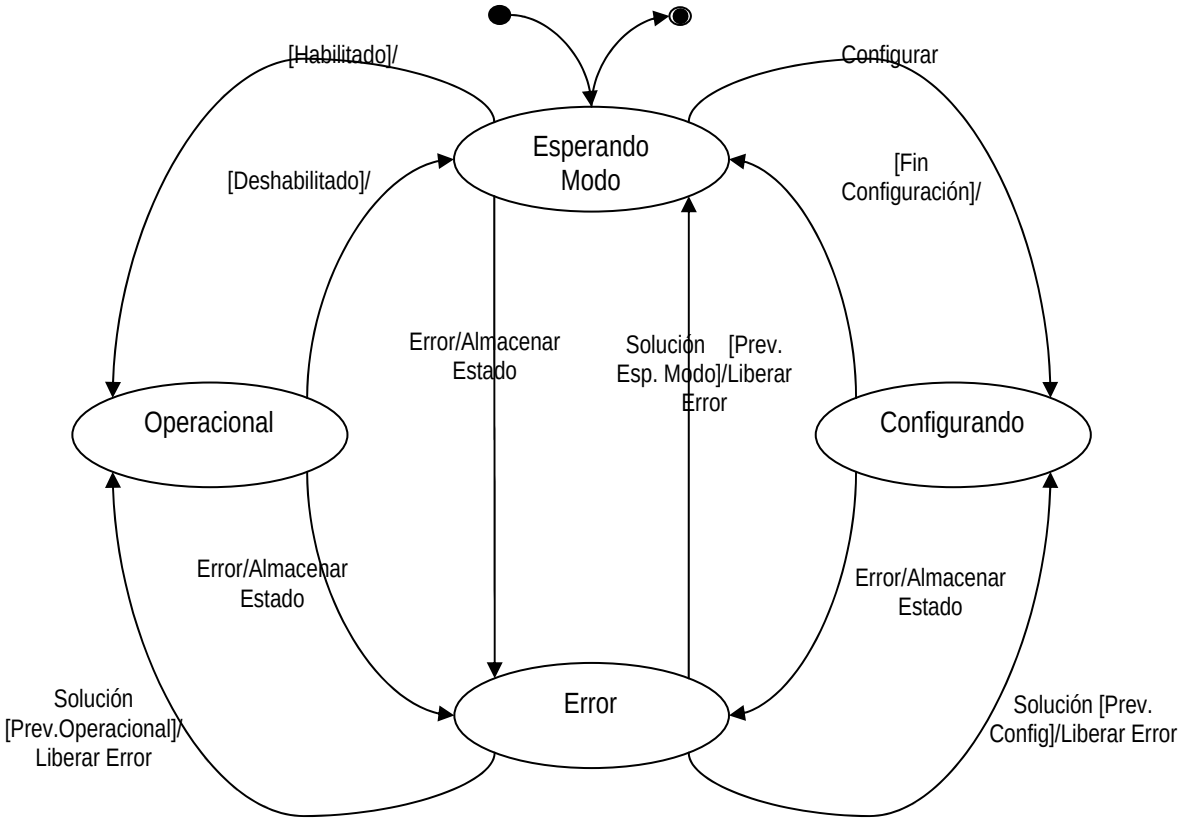


Fig. 7.27.- Diagrama de estado del RUC, MUC y SUC

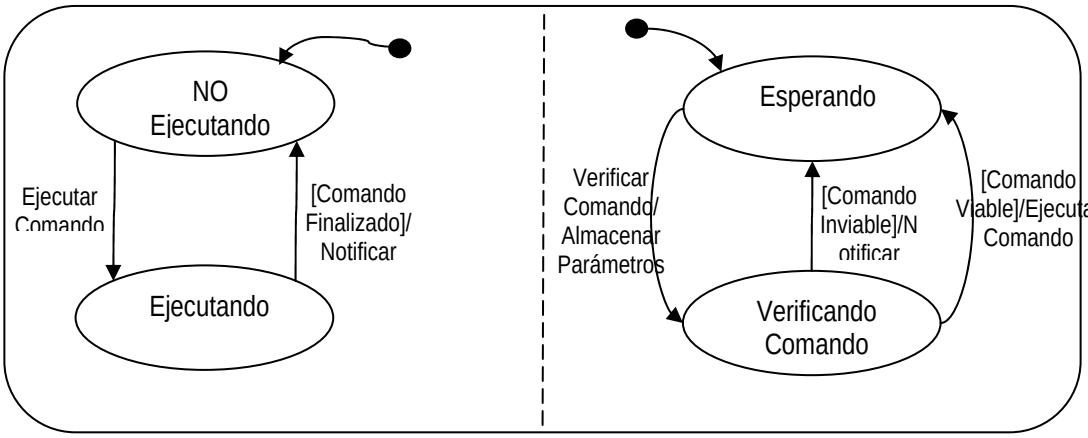


Fig. 7.28.- Diagrama de sub-estados del estado Operacional del SUC.

7.3.3. Vista de ejecución

La vista de ejecución de la arquitectura describe la estructura de un sistema en términos de sus elementos ejecutables (tareas de sistemas operativos, procesos, hilos, espacios de direcciones, etc) [Hofmeister00]. Dado que esta traducción de la arquitectura a elementos ejecutables puede variar bastante a lo largo del tiempo (incluso durante el desarrollo), es importante diseñar la arquitectura de forma que se pueda adaptar fácilmente a los cambios. Los factores que influyen en las decisiones que se tomen en esta vista de ejecución son principalmente:

- ✓ Rendimiento
- ✓ Requisitos de distribución
- ✓ Plataformas de ejecución (tanto a nivel *software* como *hardware*)

Las decisiones de diseño *hardware* que son particularmente relevantes para la implementación del *software* son el número y tipo de dispositivos en el sistema (particularmente procesadores) y los medios de comunicación física que los enlazan. Los *stakeholders* deben llegar a un compromiso entre los recursos disponibles y las prestaciones buscadas de forma que se alcance una solución óptima según las restricciones del sistema y los requisitos marcados [Douglass00].

En algunos sistemas, la vista de ejecución puede ser tan sencilla como una sola tarea ejecutándose en un procesador. Según se incrementa la complejidad del sistema, puede ser necesaria un paralelismo "real", que sólo se puede conseguir distribuyendo los subsistemas en diferentes procesadores. Para ello hace falta, sobre todo, que los interfaces de los subsistemas estén bien definidos, que los patrones de interacción entre ellos lo permitan (incorporando módulos de desacoplo, RPC, patrones como Proxy, etc) y por supuesto, encapsular el HW para que pueda acoplarse fácilmente a la arquitectura global del sistema. Todo esto es lo que se ha perseguido en el diseño de ACROSET y en este capítulo se está intentando demostrar.

En el diagrama de despliegue de la **Fig. 7.29** se pueden observar los distintos procesadores y dispositivos que encontramos en el sistema GOYA. Los dos únicos procesadores del sistema son la estación de teleoperación (Teleoperation Station) y la unidad de control del robot (GOYA_CU). Como se dijo al principio de este capítulo, la estación de teleoperación es una estación de trabajo donde se ejecutan diferentes procesos de simulación cinemática y se presenta la interfaz para el usuario remoto. La unidad de control consiste en un PC Industrial con sistema operativo RT-Linux donde se ejecutan todas las tareas de los diferentes subsistemas que se verán en este punto. El resto de dispositivos son:

- Interfaz Electromecánica (Electro_Mech_HMI). Una botonera que permite a un operario controlar de manera local al robot.

- Tarjetas de entrada/salida *"conectadas"* en la Unidad de Control:
  - Entrada/Salida Digital (D_I/O Card). Conectadas a ellas todos los sensores digitales del sistema (finales de carrera, interruptores de posicionamiento[18]), así como las salidas para accionamiento de motores y electroválvula para la herramienta. El accionamiento de los motores se hace por medio de variadores de frecuencia.
  - Lectora de encoders (Encoder Card). Tarjeta con contadores rápidos para leer hasta tres encoders de manera simultánea.
  - Entrada analógica (A_I Card). Tarjeta lectora de señales analógicas para interpretar la señal del sensor de presión de la herramienta y de otros que puedan añadirse.

En la primera versión del prototipo GOYA no se realizó una distribución de la aplicación en distintos procesadores, todas las tareas de la arquitectura se ejecutan concurrentemente en el nodo GOYA_CU.

Todas las tarjetas de interfaz con sensores y actuadores ofrecen su información de manera *pasiva*, es decir, las tareas de la unidad de control encargadas de obtener información de las mismas deberán realizar una lectura periódica de los registros de las tarjetas y detectar si se produce un cambio en sus valores; en ningún caso las tarjetas generan *interrupciones software* o *hardware*. Este dato es muy importante para aplicar los criterios de elección de tareas que se debe realizar en el planteamiento de la vista de ejecución de la arquitectura.

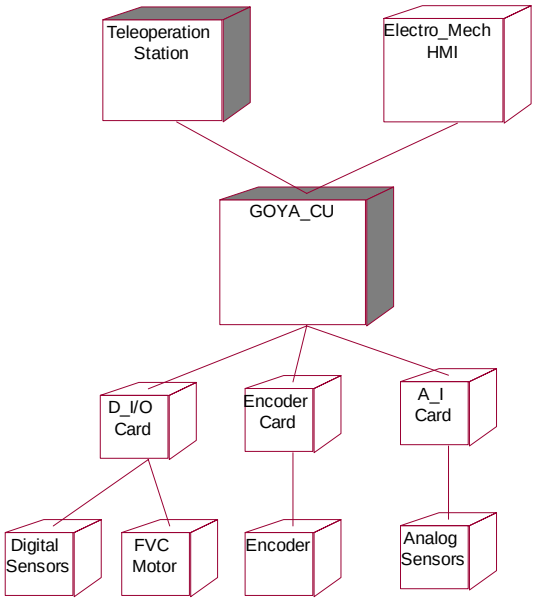


Fig. 7.29.- Vista de ejecución: diagrama de despliegue del sistema GOYA

La mayoría de métodos de desarrollo sugieren que el punto inicial para proponer la vista de ejecución de un sistema consiste en la asociación de cada elemento conceptual de alto nivel de la arquitectura con un elemento de ejecución. En el caso de la instanciación de ACROSET hecha para el GOYA y observando los diagramas presentados en los puntos anteriores, dicha asignación resultaría en un diagrama de tareas para todo el sistema (incluidos CCAS, UIS, IS y SMCS) como el de la **Fig. 7.30**. En este diagrama se observa que cada componente de control asociado a un RUC, MUC y SUC se ha

[18] Muchos de los sensores relativos a la seguridad, como interruptores que se activan si se choca con un obstáculo, se conectan directamente al *hardware* por motivos de seguridad. Lo mismo pasa con algunos analógicos, como los de sobre-temperatura de los motores, que de hecho, están incluidos en el propio variador. En todo caso, después de actuar, notifican de su acción a la Unidad de Control.

asignado a una tarea (TRUC_Coord, TMotor_SUC[19], etc). Igualmente hay una tarea asociada al ejecutor de secuencias (TProgIntel), al intérprete de usuario (TUI) y a cada uno de los objetos activos del SMCS. En el caso de los componentes de abstracción del *hardware* (sensores y actuadores en el caso del GOYA), adelantándonos al proceso de agrupación de tareas que se verá más adelante, se han agrupado funcionalmente en tres conjuntos: sensores digitales (TDSensors), *encoders* (TEncoders) y actuadores (TActuators). Los tres objetos que aparecen en la parte inferior del diagrama no son tareas, sino los objetos pasivos que representan las interfaces con las tarjetas de adquisición de datos (*drivers*).

En este diagrama se han representado también los tipos de mensajes que intercambian estas tareas. Como se puede ver, todos son mensajes asíncronos (tanto comandos, como datos). Los objetos pasivos que aparecen en el diagrama de colaboración representan las tarjetas de entrada/salida. Puesto que estas tarjetas son pasivas, es necesario realizar una comprobación periódica (*polling*) para ver si los datos han cambiado, por eso las tareas de comunicación con el hardware serán periódicas, como se verá en el siguiente punto.

7.3.3.1. Aplicación de los criterios de estructuración de tareas de COMET

Demasiadas tareas pueden llevar a un sistema a incrementar innecesariamente su complejidad debido a una gran necesidad de comunicación entre tareas y sincronización, así como un aumento de la sobrecarga del sistema por el gran número de cambios de contexto necesarios. Por ello, el diseñador debe adoptar un **compromiso** entre introducir tareas para simplificar y clarificar el diseño y, por otra parte, no introducir demasiadas para no perjudicar con ello el rendimiento. Como apoyo a la adopción de este compromiso, Hassan Gomaa propone en su método COMET [Gomaa00] una serie de **criterios de estructuración y agrupación de tareas**. Los primeros ayudan al diseñador a decidir el número y tipos de tareas que debe haber en el sistema, y los segundos le permiten analizar distintas agrupaciones de tareas según diversos criterios.

Las tareas pueden ser activadas periódicamente o de manera aperiódica. Una tarea puede exhibir más de un criterio de estructuración de tareas. Primero se aplican los criterios de estructuración:

- Tareas de entrada/salida
- Tareas internas
- Prioridad de tareas.

Con ello resulta una traducción directa de objetos a tareas. En una segunda fase se aplican los criterios de agrupación de tareas que se verán más adelante, para reducir el número de tareas del sistema.

Caracterización de tareas

Como primer paso para definir las tareas es necesario especificar las características de los dispositivos que se relacionan con el sistema, así como los datos que se intercambian. Esto se ha realizado en el análisis del principio del capítulo y al principio de este punto. En la **Tabla 7.2** se muestran resumidas las decisiones de estructuración de tareas referentes a los tres primeros puntos (entrada/salida, internas, prioridad) tomadas para el GOYA. Hay que destacar que algunas tareas pueden cumplir los criterios de clasificación de distintos grupos, por eso aparecen repetidas. Por ejemplo, un SUC tiene una tarea de control periódica crítica.

La estructuración de tareas del diagrama de la **Fig. 7.30** se puso en marcha en el prototipo GOYA teniendo en cuenta criterios de claridad de diseño. Por ello se asignó cada tarea a un componente de modelado básico en ACROSET, sin pensar demasiado en el rendimiento. Se tuvo en cuenta además que, excepto las tareas críticas que se han definido como periódicas, el resto de tareas no sobrecargan demasiado al sistema. Además, dadas las características del prototipo, los requisitos de tiempo-real no

[19] Estas tareas ejecutan un diagrama de estado, pero pueden necesitar activar de forma concurrente una nueva tarea para realizar el control de un comando (ver Nota 1 en Tabla 7.2)

son especialmente estrictos, y la pérdida de un plazo de respuesta no ocasiona graves problemas al sistema (los mecanismos de seguridad, parada por choque ante obstáculos, parada de emergencia, etc., están implementados en *hardware*).

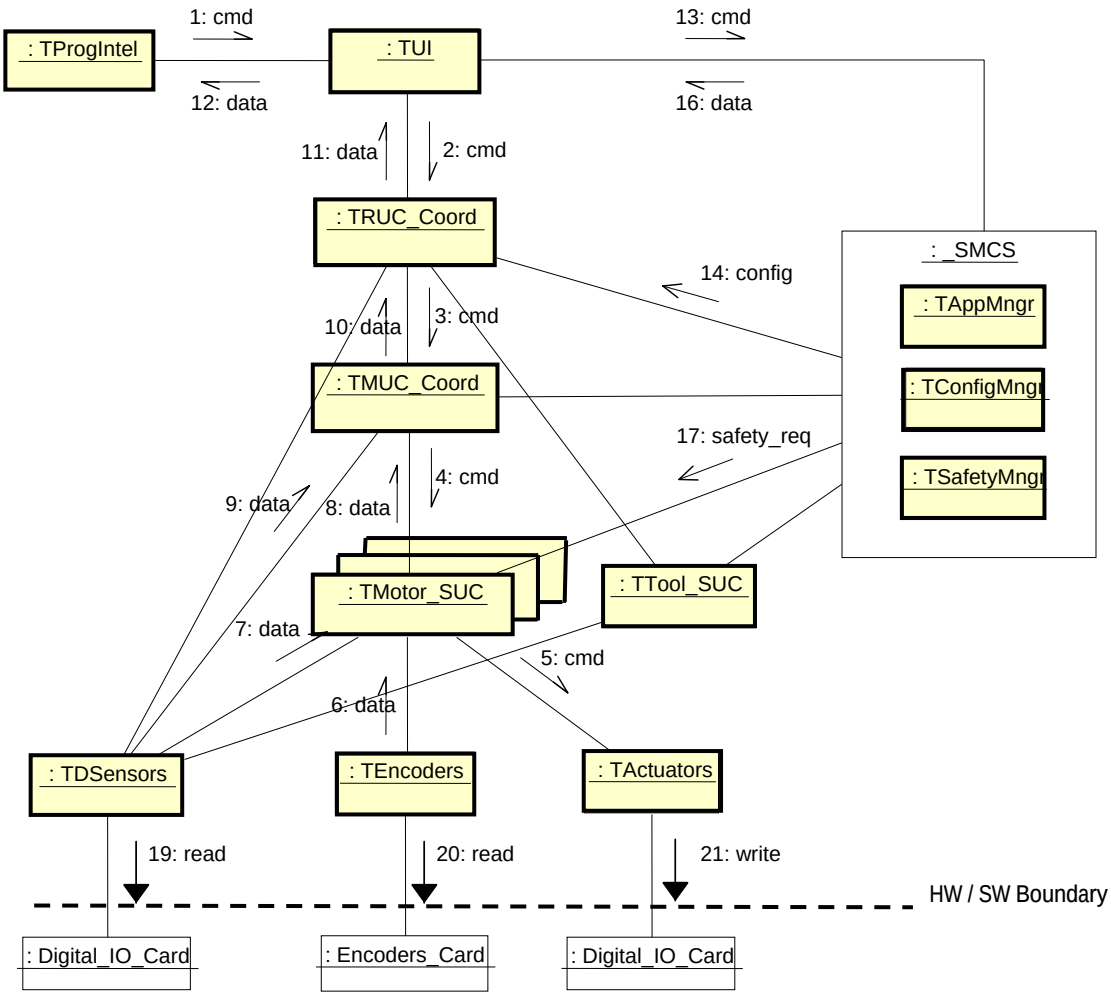


Fig. 7.30.- Vista de ejecución: Tareas iniciales con especificación de mensajes[20]

[20] Con borde más grueso se representan las tareas. Sin color de relleno se representan objetos pasivos, que en este caso son los dispositivos hardware externos (Tarjetas) a los que se accede periódicamente.

Grupo de tareas	Clase	Aplica a tarea:	Explicación
Tareas de interfaz con dispositivos de entrada/salida	Aperiódicas	En GOYA no hay dispositivos de entrada asíncronos.	Cuando un dispositivo *hardware* genera llamadas asíncronas al sistema (interrupciones) que se deben atender de inmediato para no perderlas.
	Periódicas	TDSensors TEncoders	Un dispositivo pasivo, como las tarjetas de entrada/salida de datos, serán sondeadas periódicamente a una frecuencia que dependerá de lo crítica que sea la entrada y cada cuanto tiempo se espera que cambie. También depende del retraso que sería tolerable aceptar en una notificación.
	Pasivas	TActuators	Se usan cuando se interactúa con dispositivos pasivos que no necesitan sondearse periódicamente. Típicamente tareas de interfaz con dispositivos de salida, puesto que la tarea sólo se activa con el comando que se desea comunicar a tal dispositivo de salida.
Tareas internas	Aperiódicas	Todas, excepto TDSensors y TEncoders	Tareas activadas de manera interna para ejecutar actividades a ser ejecutadas según demanda. Se activan cuando llega un mensaje asíncrono.Después de ejecutar la acción espera al siguiente evento.
	Periódicas	Tareas de control activadas en TRUC, TMUC y TSUC según demanda (Nota 1).	Tareas internas que ejecutan periódicamente una acción. Se activan mediante un evento de temporizador. El periodo de la tarea es el tiempo entre sucesivas activaciones.
	Control	TProgIntel, TRUC, TMUC, TSUC (Nota 1)	Tareas de control que ejecutan un diagrama de estado de ejecución secuencial (no incluyen concurrencia).
	Interfaz de Usuario	TUI	Tarea de interfaz con el usuario secuencial. La velocidad de esta tarea suele estar condicionada a la velocidad de la interacción con el usuario.
Requisitos de tiempo-real de tareas[21]	Críticas	TSUC, TMUC, TRUC	Tareas que no pueden perder un plazo de respuesta.
	No-críticas	El resto	La pérdida de un plazo de respuesta puede ser admisible por el sistema.

Tabla 7.2.- Caracterización de tareas en GOYA

*Nota 1: Las tareas correspondientes a los elementos de control de la arquitectura (*TRUC_Coord, TMUC_Coord, TMotor_SUC *y* TTool_SUC *son tareas que realiza un diagrama de estado secuencial (ver Fig. 7.28), donde las transiciones de un estado a otro se produce con la llegada de eventos asíncronos. El diagrama de estado representado en la Fig. 7.28 muestra concurrencia entre estados: efectivamente, un SUC puede responder a un evento asíncrono, pero si ese evento corresponde a un comando de control que se debe ejecutar en el SUC, deberá activarse una nueva tarea, en este caso periódica, que realice el control (ejecutando comando) hasta que termine la ejecución de dicho comando o bien se aborte dicha ejecución por la llegada de un comando más prioritario.*

7.3.3.2. Aplicación de los criterios de agrupación de tareas de COMET

De entre los criterios de agrupación de tareas ofrecidos por COMET, sólo se han considerado aplicables en la implementación de ACROSET para el GOYA los siguientes:

Agrupación temporal. Las tareas candidatas podrán ser agrupadas si:

- ✓ No hay dependencia secuencial entre ellas.
- ✓ Son periódicamente activadas (por ejemplo, por un *timer*) para realizar una actividad.

[21] COMET nombra a este grupo como "Prioridad de Tareas", pero se considera más adecuado denominar a este grupo "Requisitos de tiempo-real de las tareas".

Las candidatas ideales para esta agrupación son las tareas de lectura de sensores pasivos. En el caso del GOYA, las tarjetas de entrada/salida escriben/leen en un registro que debe ser periódicamente sondeado para detectar un cambio. Por ello, una sola tarea se encarga de leer todos los sensores digitales y analógicos y otra tarea se encarga de la escritura. Este criterio ya se había aplicado al presentar el diagrama de la **Fig. 7.30**. Cuestiones a tener en cuenta:

- Si una tarea es más prioritaria que otra no deberían ser agrupadas.
- Como es lógico, si se van a ejecutar en procesadores separados no pueden agruparse.
- Se da preferencia a las tareas que funcionalmente están relacionadas (sensores – actuadores, etc). Agrupar tareas que no están funcionalmente relacionadas no es deseable desde el punto de vista del diseño, aunque si hay una sobrecarga excesiva del sistema podría hacerse necesario.
- Periodo de muestreo: Sólo combinar si los periodos de muestreo son múltiplos.

Llevar al extremo este criterio podría tener los mismos problemas que un ejecutivo cíclico: tendría que haber un procedimiento supervisor que se ejecute en un periodo correspondiente al máximo común divisor de todos los periodos de las actividades agrupadas (p.ejm. si las actividades se realizan cada 15, 20, 25 ms, la tarea combinada debería activarse cada 5 ms: por eso sólo es conveniente aplicar la agrupación de tareas si sus periodos de activación son múltiplos). El ciclo principal es muy largo si son de diferente orden de magnitud.

Agrupación de tareas tipo *control*: Un objeto controlador, que ejecuta un diagrama de estado secuencial se traduce a una tarea de control. Lo que aplica a los SUCs , MUCs y RUCs, si observamos sus diagramas de estado (**Fig. 7.27** y **Fig. 7.28**) es lo siguiente:

> **Actividades dependientes del estado que son activadas o desactivadas por el objeto tipo *control* por una transición de estado.** Consideremos una actividad que se realiza cuando se produce una transición de estado (p.ejm. la ejecución de un comando) y se sigue ejecutando continuamente hasta que se desactiva en otra transición de estado (p.ejm. cuando se recibe la orden de parar la ejecución o bien cuando ha terminado el proceso). En este caso, la actividad de control debería estructurarse como una tarea paralela, porque tanto el objeto que sigue el diagrama de estado (el SUC puede seguir recibiendo comandos) como la tarea de control necesitan estar activas concurrentemente.
>
> En el caso del GOYA, cuando un SUC, MUC o RUC debe ocuparse de la gestión de algunos comandos de control, sobre todo los de movimiento, se puede crear una nueva tarea asociada al comando o bien despertar una tarea de control que permanece suspendida (relacionado con la **Nota 1** de la página anterior), en ningún caso esta tarea podrá ser agrupada con la que maneja el diagrama de estado.

Criterios de inversión de tareas

La proliferación de tareas se puede reducir drásticamente aplicando los criterios de inversión de tareas, que en el caso del GOYA, se reduce a la inversión de múltiples instancias de tareas para los SUC. Efectivamente, los SUCs son objetos tipo control idénticos que se han separado en tareas distintas porque cada SUC puede estar en un estado distinto al mismo tiempo. En lugar de hacerlo así, se pueden incluir todos los objetos SUC en una misma tarea, separando la información del estado de cada objeto en distintos objetos entidad pasivos.

A pesar de haber agrupado todos los objetos tipo SUC que ejecutan un diagrama de estado en una tarea, siguen existiendo (y no se pueden agrupar) una tarea por SUC que se dedica a controlar la ejecución de comandos determinados, como el control de un movimiento.

En el caso del GOYA, lo normal sería que el MUC completo se sustituyera por una tarjeta COTS controladora de tres motores, quedando en la implementación simplemente un componente de abstracción del *hardware* que sería un MUC como objeto pasivo, con lo cual reduciría enormemente la

sobrecarga; mientras tanto, se puede recurrir a criterios de agrupación de tareas para disminuir el número de las mismas.

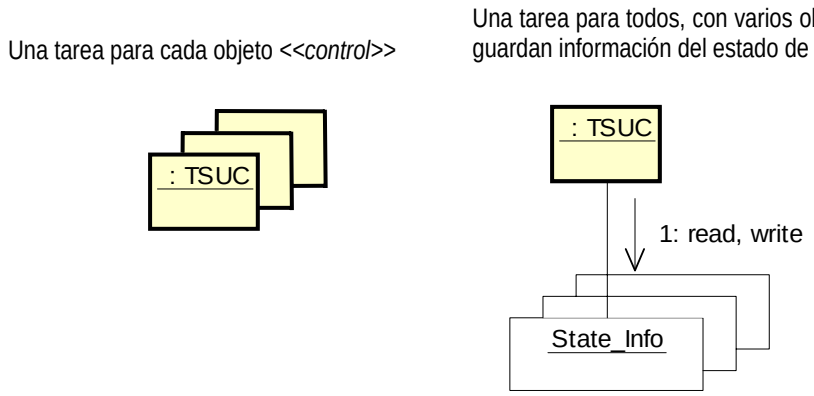


Fig. 7.31.- Aplicación del criterio de inversión de tareas con múltiples instancias en el caso de los SUCs

Como dice Gomaa al aplicar el criterio de **agrupación temporal**, aunque agrupar tareas que no están funcionalmente relacionadas (TRUC con TMUC ó TSUCs) no es deseable desde el punto de vista del diseño, se puede hacer si hay una sobrecarga excesiva del sistema. De esta forma, se podría incluso agrupar las tareas del SUC también con el MUC e incluso con el RUC (con la parte que gestiona el diagrama de estado). Quedaría una tarea global de control con una serie de objetos pasivos almacenando el estado y con tareas periódicas de control asociadas a RUC, MUC y SUC que se activan sólo en demanda de algunos comandos específicos de control *software*.

Definición de comunicación entre tareas

Después de estructurar el sistema en una serie de tareas concurrentes, el siguiente paso que propone COMET consiste en definir las interfaces de las tareas como parte de los distintos protocolos de paso de mensajes, sincronización de eventos o acceso a objetos de ocultación de información. Este paso ya se muestra en el diagrama de tareas de la **Fig. 7.30**. Ya se ha definido en la vista de módulos las interfaces de los componentes y de los puertos de acceso de dichos componentes a nivel de funcionalidad (todos los tipos de acciones que pueden realizarse con un componente divididas según su relación en puertos distintos). En el punto **7.3.2.1** se habló de los puertos y la adaptabilidad a distintos conectores mediante mecanismos de herencia. Precisamente los tipos de sincronización entre tareas establecen tipos de conectores distintos, y por tanto, distintos tipos de puertos de comunicación:

- Comunicación por mensajes
  - Asíncrona: El emisor continúa su ejecución.
  - Síncrona (cita): El emisor espera a que el receptor reciba el mensaje.
  - Invocación remota (cita extendida): El emisor espera a que el receptor reciba el mensaje, y la respuesta de éste.
- Sincronización de eventos
- Interacción mediante objetos intermediarios.

En el GOYA, la comunicación entre tareas (**Fig. 7.30**) se realiza por paso de mensajes asíncronos, quedando así desacopladas las tareas. Puesto que pueden llegar varios eventos, y la comunicación de datos puede tener múltiples lectores, a cada tarea se asocia un objeto protegido intermediario que garantiza que no se produzcan conflictos en la comunicación. Como se ya se dijo al describir el diagrama, el acceso a las tarjetas de adquisición se hace mediante tareas periódicas que leen directamente de los *drivers* de las tarjetas.

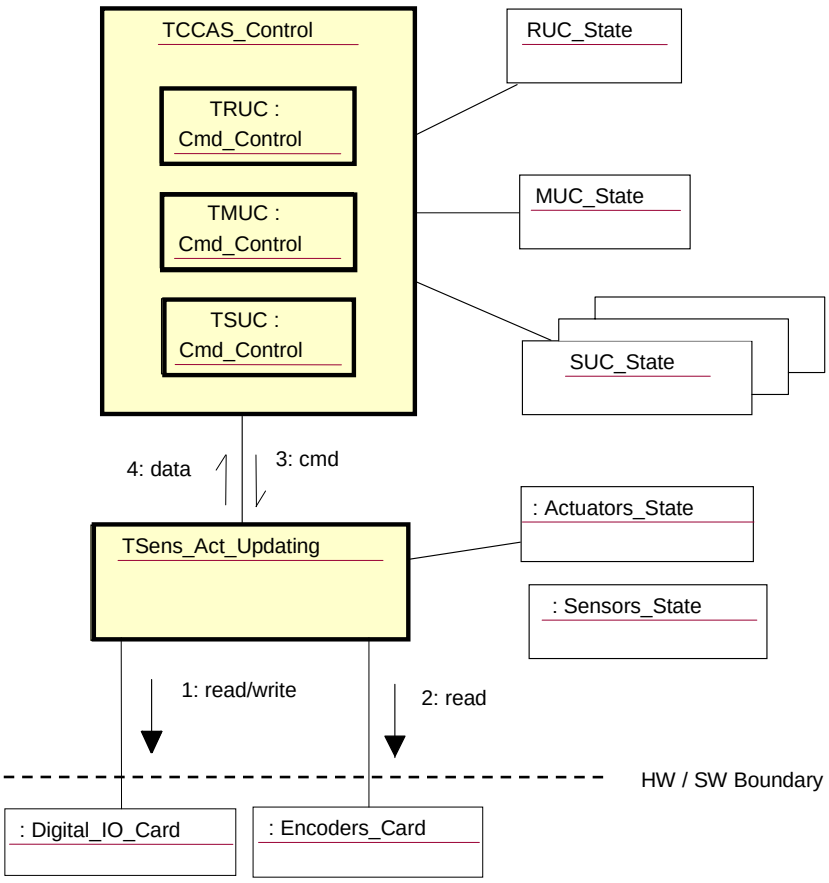


Fig. 7.32.- Agrupación al máximo de tareas en el CCAS del GOYA[22]

En el siguiente punto, se explica cómo se traduce la variabilidad de los conectores según distintos protocolos en las interfaces de los puertos.

7.3.4. Vista de código

Según [Hofmeister00], la vista de código de una arquitectura es donde se describe cómo los módulos, sus interfaces y sus dependencias se traducen a componentes específicos del lenguaje de programación que se utilice para el *software* de control.

En el caso del sistema GOYA, toda la arquitectura se ha implementado en *software* en un único procesador. Este aspecto, que favorece poco al rendimiento de un sistema complejo, es justificable en el GOYA al tratarse de un prototipo de bajo coste, donde se ha buscado utilizar componentes industriales muy comunes (motores asíncronos con variadores de frecuencia), y no hay necesidades de control preciso (el control de los motores es *todo/nada* y el posicionamiento se hace por *par*). En la continuación de los trabajos en el proyecto EFTCoR, los componentes *hardware* cobran mayor importancia y el control de posicionamiento es más preciso.

El contar con toda la arquitectura implementada en un solo procesador ha permitido experimentar con distintas asignaciones de tareas y realizar distintos análisis de tiempo-real utilizando UML-MAST [MAST01], [PFC04c], variar rápidamente los componentes, etc. Puesto que toda la unidad de control es *software*, se eligió el lenguaje de programación Ada95, especialmente diseñado para sistemas de tiempo-real, que permite programar *software* seguro, fiable, robusto y de fácil evolución. En el ilustrativo documento *"Developing Software that Matters"* [Gasperoni01] y en [Ada-Answ04] se

[22] Con borde más grueso se representan las tareas. Sin color de relleno se representan objetos pasivos.

exponen una serie de justificaciones de por qué Ada es un lenguaje que ofrece mucha más seguridad en el desarrollo de *software* para sistemas críticos en seguridad o de tiempo-real frente a otros lenguajes como C/C++ , Java, etc.

Como la tesis quizás no es el ámbito más adecuado para discutir el código de una aplicación, se pueden encontrar detalles del código generado en los proyectos [PFC00],[PFC01a],[PFC01b] y [PFC02]. No obstante, se ha estimado conveniente introducir aquí algunos aspectos de implementación y de diseño detallado que se consideran interesantes para futuras instanciaciones de ACROSET.

El acceso a los componentes de abstracción del *hardware* deberá hacerse con objetos protegidos, puesto que puede haber múltiples lectores y escritores. Ada95 favorece enormemente la utilización de mecanismos necesarios en la mayoría de sistemas de tiempo-real críticos: objetos protegidos, sincronización de tareas, bloqueo de tareas, etc. [Ada95]

7.3.4.1. Los puertos y la variabilidad asociada a los conectores

Como se vio en el capítulo 3, los puertos compatibles de diferentes componentes se unen gracias a los conectores, según un protocolo. Los puertos asociados a los componentes ofrecen (y requieren) dos tipos distintos de servicios: Por una parte ofrecen o requiere la funcionalidad del componente relativa a los servicios que el componente proporciona. Por ejemplo, en el caso de un SUC que controla un *joint*, servicios como *moveTo, stop, getPosition,* etc. Estos servicios han sido representados por las operaciones de los puertos en los diagramas anteriores. Por otra parte, los puertos también proporcionan la infraestructura de comunicación necesaria dependiendo del tipo de conector, de hecho, implementan los servicios necesarios para cumplir el protocolo de comunicación acorde con su conector.

El cambio de un conector, que implica básicamente el cambio de protocolo de comunicación con los puertos, debe reflejarse en el cambio de los servicios del puerto referidos a la comunicación, pero no de los referidos a la funcionalidad de componente que contiene al puerto en cuestión. Para separar estos dos conceptos, se definen tantos tipos de puerto como posibles modos de comunicación; por otra parte, se definen los tipos de puertos según la funcionalidad que ofrecen (puertos de control de un SUC, de salida de datos, de entrada de datos de un sensor, etc).

Para hacer que un puerto definido, cumpla un protocolo u otro, simplemente habrá que utilizar mecanismos de herencia, como se observa en la **Fig. 7.33**. Al igual que el ejemplo que se representa una conexión directa (Simple_OP) y un objeto observable (Subject_OP) se podrían dar todos los casos que se presentaron al final del Capítulo 5, distintos patrones de comunicación [Douglass00], [Buschmann96], [Gamma95], [Schlegel02].

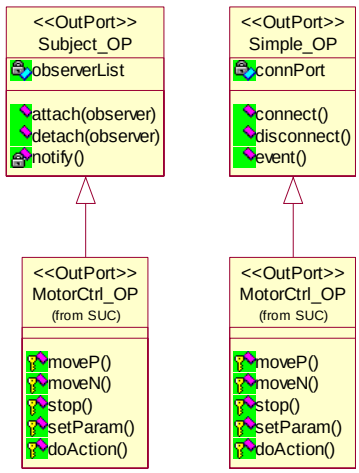


Fig. 7.33.- Herencia de distintos protocolos de comunicación por puertos del mismo tipo

Al instanciar ACROSET para un sistema, la parte funcional de un puerto será invariable si no cambia la funcionalidad del sistema, sin embargo, la parte asociada a la comunicación podrá intercambiarse fácilmente (en tiempo de compilación) dependiendo del conector que se utilice, utilizando simplemente los mecanismos de herencia, como se ha dicho. Nótese que esta exposición sobre variabilidad de conectores y los patrones y clases utilizadas para solucionarlos ha sido independiente del **medio físico** sobre el que se apoya la comunicación entre procesos: se puede implementar usando memoria compartida, redes Ethernet, buses diversos, etc., será cuestión de implementación de los distintos tipos de puertos.

En el GOYA se han utilizado tres tipos de conectores, utilizando puertos con los protocolos correspondientes al método de comunicación:

- Comunicación de datos entre los componentes: Patrón observador, se permite a cualquier componente que se pueda suscribir a los datos proporcionados por otro componente. Los puertos de salida y entrada de datos son *subject* y *observer* respectivamente.
- Control de un componente sobre otro: Comunicación directa por paso de eventos asíncronos.
- Obtención de datos del hardware: Llamada a procedimiento. Los objetos de abstracción del hardware (su tarea o tareas de actualización), acceden directamente de forma periódica a la capa de abstracción del sistema operativo llamando a los *drivers* de las tarjetas de adquisición de datos.

7.3.4.2. Abstracción, herencia y composición

Observando los componentes principales de ACROSET (incluidos sensores y actuadores virtuales), y las implementaciones presentadas en la vista de módulos, es lógico pensar que independientemente del tipo de sensor, actuador, controlador, etc., que nos encontremos en un sistema real, se podrán definir unas interfaces comunes a los dispositivos que pueden particularizarse utilizando herencia y composición, para llegar a los componentes concretos de un sistema determinado.

El comportamiento de dichos componentes no suele variar mucho entre las aplicaciones, por tanto se puede abstraer a una forma genérica. Por ejemplo, consideremos el ejemplo de un sistema de control de movimiento que puede ser representado por componentes genéricos ControlledMotor:SUC y CoordinatedMotors:MUC, con una serie de operaciones definidas que favorezcan la reutilización (aunque su implementación sea distinta). Cada sistema robótico tiene una arquitectura interna de movimiento que refleja el *hardware* de control. Cada componente *hardware* introduce restricciones arquitectónicas en el sistema.

En una implementación, se podría usar tarjetas de control de movimiento *commercial off-the-shelf* (COTS). En una segunda implementación, se podría usar una tarjeta de diseño propio con chips COTS. En una tercera implementación, se podría cerrar el lazo de control usando *software* ejecutándose en un procesador empotrado. Mientras que éstas son tres implementaciones diferentes de un sistema de control de movimiento, los requisitos de comportamiento del motor controlado son los mismos. En cualquiera de ellas, se desea poder ordenar comandos de movimiento, perfiles de velocidad y control de trayectoria, así como leer las posiciones, velocidades, aceleraciones y status. Para una persona que realiza un sistema de navegación para un robot móvil, sólo es necesario comprender el comportamiento del componente en lugar de tener que comprender el comportamiento detallado de su implementación y los detalles del *hardware*.

Los componentes ControlledMotor y CoordinatedMotors, tipo SUC y MUC respectivamente, son una representación abstracta del control de movimiento que definen lo que se supone que los componentes deben hacer. Estos componentes ocultan los detalles de implementación sin comprometer características particulares del *hardware* y mantienen información del estado del componente que abstraen. En ACROSET se ha generalizado aún más presentando no ya un "*motor controlado"* sino un

SUC, que en su implementación concreta en un sistema puede ser un ControlledMotor, ControlledTool, etc.

Estos componentes en su implementación, pueden ser clases abstractas que se realicen en objetos activos que a su vez se compongan de sensores y actuadores controlados por dicho objeto, etc. Puede ser útil que un objeto que vaya a acceder, por ejemplo a un sensor (o al puerto de acceso al mismo), conozca sólo la interfaz abstracta de dicho sensor; de esta forma tendrá la misma referencia, y no habrá que cambiar el componente, independientemente de que luego el sensor ofrezca distintas informaciones o se implemente de maneras distintas.

El caso de los puertos de entrada de datos es un típico ejemplo de aplicación de los tipos de datos genéricos, mecanismo que también ofrece Ada95 para permitir que una misma clase sirva para varias implementaciones, particularizando en cada caso el tipo de dato que se va a utilizar.

7.4. Aplicaciones de ACROSET en el proyecto EFTCoR.

Quizá una de las mejores formas de demostrar la adaptabilidad de la arquitectura ACROSET a diferentes implementaciones sea mostrar trabajos reales en los que ha sido utilizada con éxito, todos ellos en el dominio de los robots de servicio teleoperados para la limpieza de barcos. En concreto, ha sido utilizada para la familia de robots del proyecto EFTCoR, que surge como continuación del GOYA.

El proyecto europeo EFTCoR [EFTCoR02] "*Enviromental friendly and cost-effective technology for coating removal*", es un proyecto GROWTH del *V Programa Marco* de la Unión Europea. El objetivo global del proyecto es el desarrollo de una nueva tecnología para la limpieza de cascos de embarcaciones para su posterior pintado, desarrollando para ello una tecnología fiable y económica para la eliminación de la pintura, con la que se obtenga al menos la misma calidad de acabado superficial que con las técnicas actuales y con la que puedan reducirse las emisiones de residuos tóxicos, si no totalmente, hasta límites tolerables. Para alcanzar este objetivo, el proyecto considera:

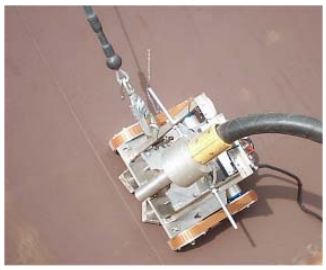
- El desarrollo de granallas reciclables, que permitan reducir su consumo hasta en un 90%.
- El desarrollo de cabezales de limpieza que permitan confinar los residuos, evitando su emisión o reduciendo la misma en al menos un 90%.
- El desarrollo de un sistema de tratamiento y reciclado de residuos.
- El desarrollo de sistemas robóticos teleoperados o semi-automáticos de bajo coste para el posicionamiento de los cabezales de limpieza. El objetivo de estos sistemas es liberar a los operarios de un trabajo duro y peligroso. Deben proporcionar la capacidad de carga suficiente para soportar cabezales de limpieza complejos, capaces de confinar los residuos.
- Desarrollo de un sistema de control de calidad basado en técnicas de visión artificial capaz de detectar y clasificar los defectos y comprobar las calidades superficiales obtenidas tras la limpieza.
- Desarrollo de nuevas pinturas con mejores propiedades de adherencia. Una mayor adherencia permite peores acabados superficiales (menor rugosidad) y por tanto disminuye la necesidad de granallado y permite utilizar otras tecnologías de eliminación de pintura (agua a alta presión).

Por razones de limitación de espacio, no se exponen aquí el estudio de requisitos y el análisis del problema tal como se hizo con el proyecto GOYA. Se remite al lector a consultar toda esta información en [EFTCoR03-d1] y [EFTCoR03-d2] y de manera más resumida en el proyecto fin de carrera [PFC04b].

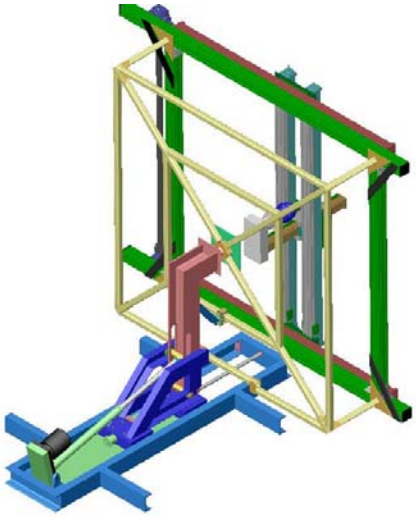
Para conseguir estos objetivos, se proponen una serie de dispositivos mecatrónicos y robots teleoperados (ver **Fig. 7.34**), cuyo diseño detallado se puede consultar en el documento [EFTCoR04-d1]. Estos dispositivos deben ser equipos robustos para funcionar en un ambiente industrial agresivo,

de fácil manejo, acoplables a otros dispositivos de posicionamiento primarios comerciales, como grúas (**Fig. 7.34-f**) y *cherry-pickers* (**Fig. 7.34-a**), y que utilicen en la medida de lo posible componentes *hardware* COTS de fácil sustitución y mantenimiento.

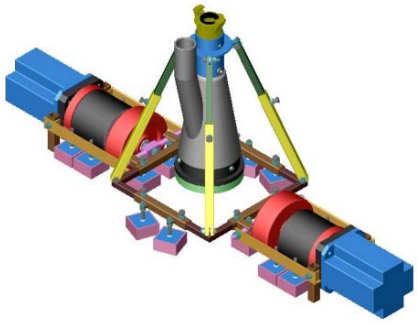
a) Mesa XYZ montada sobre una *cherry-picker* comercial



b) Robot móvil escalador

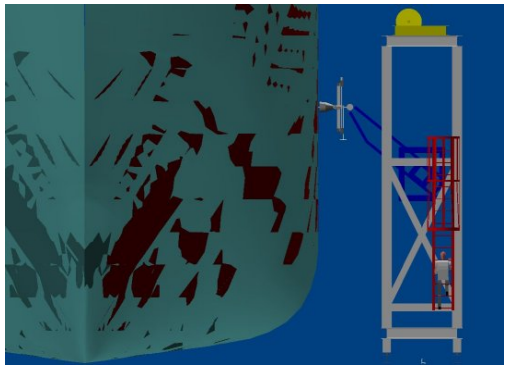


c) Modelo de la mesa XYZ



d) Segunda versión del robot móvil escalador

e) Mesa XYZ construida



f) Mesa XYZ montada en una grua-torre

Fig. 7.34.- Diferentes soluciones de posicionador primario y secundario para *grit-blasting*[23]

Las aplicaciones de ACROSET en el diseño de las unidades de control de estos dispositivos han sido principalmente de dos tipos:

- Implementación de la arquitectura en un autómata programable (PLC) que controla una mesa XYZ de posicionamiento de la herramienta (**Fig. 7.34-a,c,e**).

[23] Chorreado con granalla.

- Implementación en un *Embedded-PC* para el control de un robot móvil escalador que limpia el casco con su cabezal incorporado (**Fig. 7.34-b,d**).

7.4.1. Control de Mesa XYZ: Implementación en un PLC

La primera instanciación de la arquitectura ACROSET para el proyecto EFTCoR ha sido el sistema de control de una mesa XYZ que posiciona un cabezal de limpieza (**Fig. 7.34-e**). El cabezal consiste en una manguera que proyecta granalla y que está confinada en una campana mecánica con aspiración para recoger los residuos que se originan al impactar la granalla en la superficie pintada del barco y extraer la pintura. El posicionador primario, encargado de transportar la mesa XYZ y colocarla en su posición de operación, puede ser de diversos tipos, como grúas (**Fig. 7.34-f**) y *cherry-pickers* (**Fig. 7.34-a**). Al ser equipos comerciales, suelen llevar su propia unidad de control que se gobierna de manera independiente. La unidad de control diseñada es la que controla la mesa XYZ según las instrucciones de un tele-operador o bien las órdenes que genera un sistema de visión, encargado de reconocer los defectos de la superficie que deben ser limpiados (*spotting*).

En respuesta a los requisitos industriales especiales del proyecto EFTCoR [EFTCoR03-d2], se decidió implementar la unidad de control en un PLC (SIMATIC S7-300 series) y un bus de campo PROFIBUS-DP como se muestra en la **Fig. 7.35**, que son componentes industriales standard de amplia disponibilidad, robustos y fácilmente mantenibles que facilitan la integrabilidad, interoperabilidad y mantenibilidad de todo el sistema. El desarrollo del *software* se está realizando en el entorno de programación Step 7 con los lenguajes KOP y AWL de SIEMENS [SIMATIC02]. El diseño detallado del sistema de control se puede consultar en [EFTCoR04-d2].

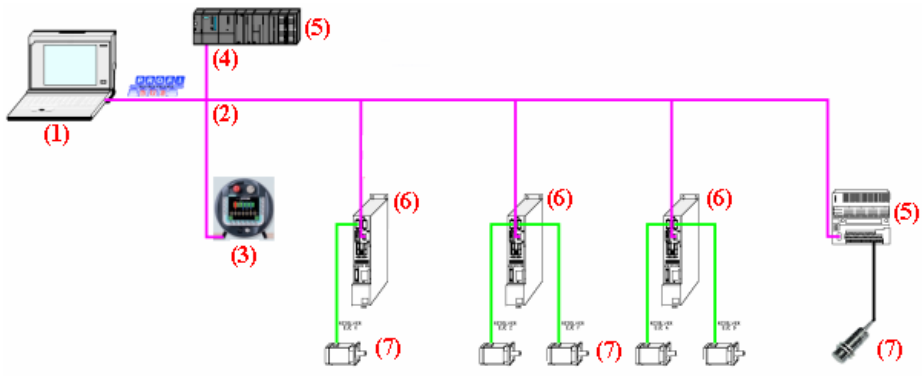


Fig. 7.35.- Arquitectura *hardware* de la unidad de control de la mesa XYZ en EFTCoR

- **PROFIBUS-DP** (2 en **Fig. 7.35**). Bus de campo especialmente adaptado a la comunicación entre sistemas de automatización y periferia descentralizada.

- **Unidad de programación PG** (1 en **Fig. 7.35**), PC conectado vía PROFIBUS con el PLC.

- **SIEMENS MOBILE PANEL 170** (3 en **Fig. 7.35**), panel de operador industrial.

- **PLC SIMATIC S7-314C-2DP** (4 en **Fig. 7.35**)**.** El programa de control se ejecuta en esta CPU. Integra también un interfaz DP par PROFIBUS, actúa como maestro del bus PROFIBUS-DP.

- **Modulos de entrada/salida** (5 en **Fig. 7.35**), para la conexión de sensores y acturadores. Hay dos tipos: uno integrado en la CPU y un módulo de periferia descentralizada, que al estar conectado por PROFIBUS con la CPU, reduce considerablemente los cables y facilita la adición de nuevos sensores al sistema.

- **Módulo de regulación SIMODRIVE 611U** (6 en **Fig. 7.35**), elementos de control de los servomotores del sistema. Cada módulo consiste en una tarjeta para el control de dos ejes de manera independiente.

- **Sensors and Actuators** (7 en **Fig. 7.35),** sensores de proximidad inductivos que actúan como finales de carrera y levas de referencia. Servomotores SIEMENS.

La propia estructura del robot y los componentes *hardware* elegidos condicionan la instanciación de la arquitectura, puesto que ellos influyen en el número de SUCs y MUCs que deban aparecen en la arquitectura de control, el número de sensores y también las relaciones de agregación o composición que se establezcan entre los componentes de la arquitectura, como se puede observar en la **Fig. 7.36**. Este diagrama muestra los componentes integrados en el CCAS son los que se muestran en la **Fig. 7.24**. En este caso, el RUC engloba toda la funcionalidad requerida para gobernar la mesa XYZ y la herramienta (cabezal de limpieza), incluyendo los sensores y actuadores.

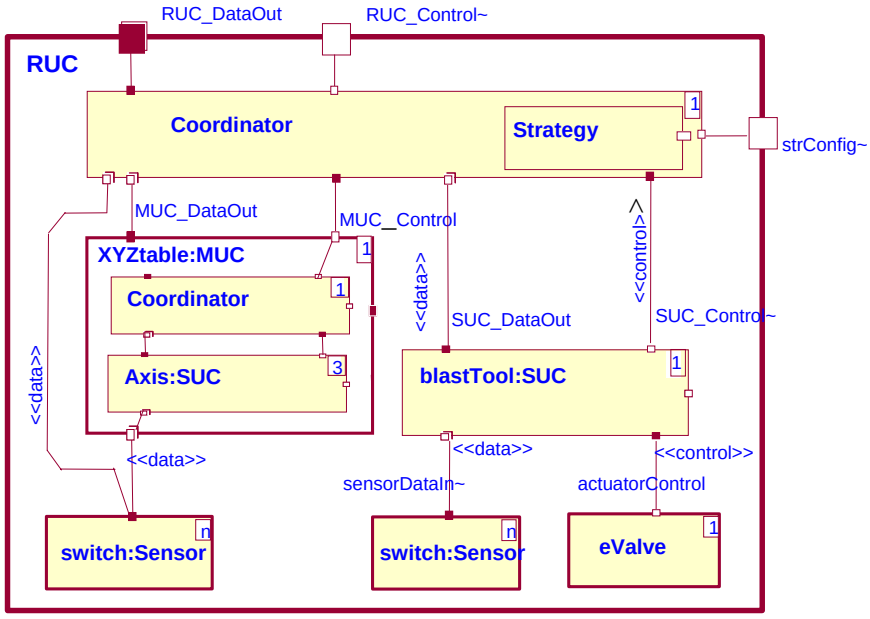


Fig. 7.36.- Componentes del CCAS en la unidad de control de la Mesa XYZ

Como se puede observar en la misma, no aparecen ni encoders ni motores como actuadores independientes (como pasaba en el proyecto GOYA – **Fig. 7.12**), sino que al haber elegido unos controladores de motores SIMODRIVE, los SUCs en esta implementación se convertirán simplemente en componentes de abstracción del *hardware* contenidos en el MUC, en un "*puente*" entre el *software* y el *hardware*, no teniendo que realizarse el control a nivel *software*, sino a nivel *hardware*. Esto demuestra la adaptabilidad de ACROSET a diferentes tipos de particiones *hardware*/*software* y su posibilidad de distribución. El control de la herramienta, sin embargo, se hace a nivel *software*: un SUC programado en el PLC que controla el accionamiento de la herramienta.

El MUC, como coordinador de los distintos SUCs *hardware*, sí que se implementa en *software* en el PLC, así como el coordinador del SUC. El MUC será un componente contenedor de toda su funcionalidad interna, y ofrecerá de manera compacta el control de una mesa XYZ. Igualmente el RUC será un componente compacto que contiene a todos los demás componentes y que ofrece la funcionalidad completa de mesa y herramienta a quien la quiera utilizar. En este caso, los usuarios serán el sistema de inteligencia, y los usuarios externos (operador y sistema de guiado por visión artificial), que interactúan con el CCAS a través de las interfaces presentes en el UIS.

A la hora de traducir la arquitectura a código en el entorno de programación Step 7 para PLCs de SIEMENS se ha utilizado principalmente el lenguaje KOP (contactos) y AWL [SIMATIC02]: cada SUC, MUC y RUC se convierte en un PLC *Function Block (FBs)* [SIMATIC02]. Son los FBs los bloques que hacen posible emular la programación orientada a objetos con el autómata, puesto que cada FB se puede interpretar como una clase que puede ser instanciada. Sólo es necesario programar un FB para cada elemento genérico de la arquitectura y posteriormente realizar cuantas instancias sean

necesarias de este FB genérico. Así por ejemplo un FB genérico para el control de un eje se puede utilizar para dar lugar a tres controladores de ejes que se encarguen, respectivamente, de los ejes X, Y y Z. La programación de un autómata como este es estructurada y la ejecución del programa es secuencial, de forma que cada bloque programado en el mismo es accedido secuencialmente por orden de aparición en el programa.

Con esta filosofía, la programación de un MUC realizada en el autómata seguirá una jerarquía como la que se muestra en la **Fig. 7.37**. Cada FB 10 es una instancia de un SUC, con sus datos de instancia en el DB 7X y con una serie de bloques de función FC que estructuran su contenido. También se representan en el diagrama las relaciones de composición que existen entre los componentes implementados.

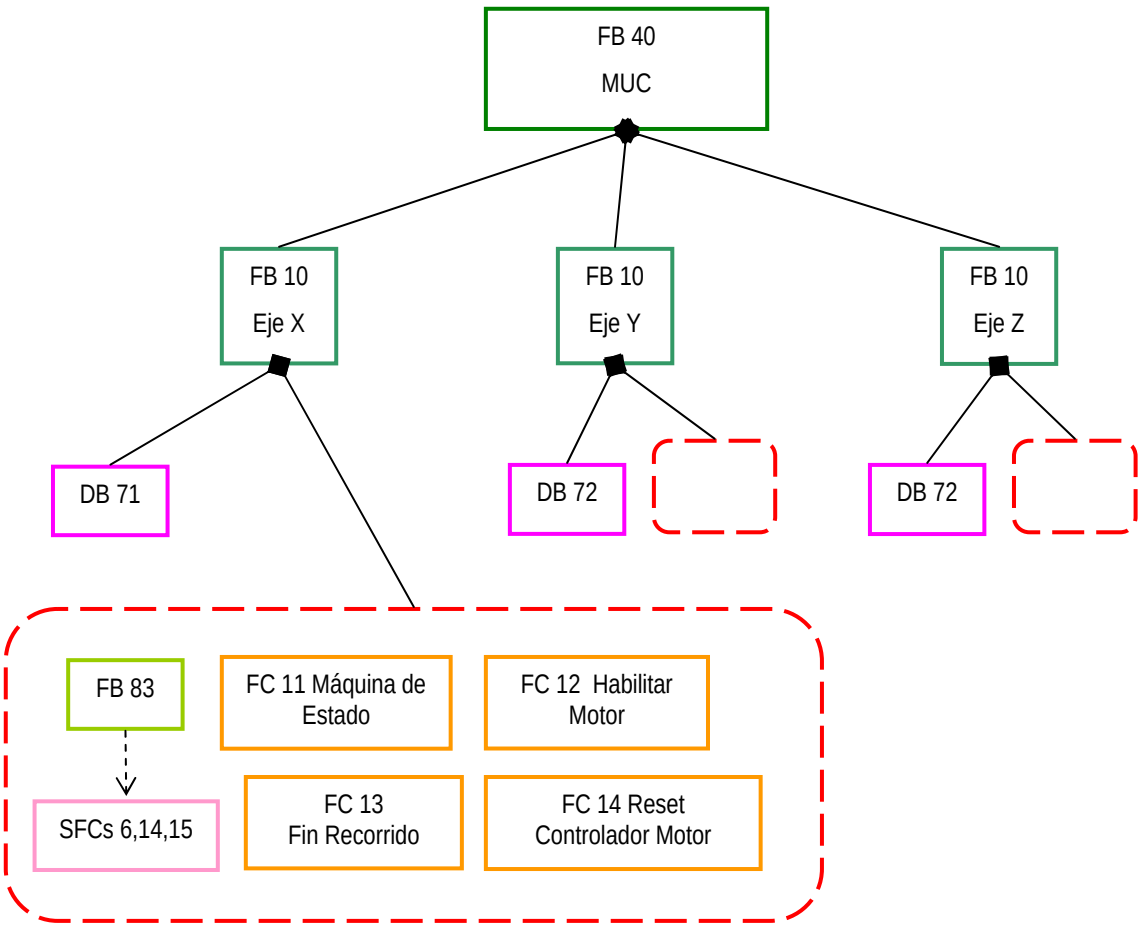


Fig. 7.37.- MUC implementado en el PLC

7.4.1.1. Interfaces de los componentes principales

Los puertos de entrada y salida de los componentes fundamentales de ACROSET, se convierten ahora en interfaces de los bloques de función FBs, diferenciados por grupos. En la **Tabla 7.3** y **Tabla 7.4** se puede ver en detalle las interfaces de entrada y salida de un SUC genérico para control de motores. En el proyecto fin de carrera [PFC04b] se puede consultar en detalle el diseño de estas interfaces.

Interfaces de entrada del SUC				
Grupo	Nombre		Tipo	Descripción
Inicialización	EN		BOOL	Activar/Desactivar el bloque de función que implementa el SUC
	NUM_DB		ENTERO	Indica el número de DB en el que se almacenan los datos del eje
	LADDR			Indica la dirección de memoria de E/S para la comunicación con el regulador SIMODRIVE
Configuración	CONF_START		BOOL	Iniciar configuración
	CONF_END			Finalizar configuración
	CONF_PARAM			Configurar parámetros
	CONF_VALUES		STRUCT	Configurar los valores de los parametros
		PARAM_NUMBER	ENTERO	Número de párametro a configurar
		MIN		Valor mínimo del parámetro
		MAX		Valor máximo del parámetro
		DEFAULT		Valor por defecto del parámetro
Movimiento	HOME		BOOL	Desplazar el sistemas al origen de referencia
	MOVE_ABS_REL			Indica si la posición establecida en MOVE_POS es ABSOLUTA (TRUE) ó RELATIVA (FALSE)
	MOVE_P			Movimiento en sentido positivo
	MOVE_M			Movimiento en sentido negativo
	MOVE_TO			Movimiento hacia una posición preestablecida, en POS
	POS		ENTERO	Posición hacia la que se desplaza mediante el comando MOVE_TO
Servicio	ACK_FAULT		BOOL	Acuse de fallo
	ENABLE			Habilitar el SUC
	DISABLE			Deshabilitar el SUC
	RESET			Resetar el SUC
	STOP			Parar el movimiento que se este realizando en ese momento
	RESUME			Continuar el movimiento que previamente ha sido parado con el comando STOP
	ABORT			Abortar el movimiento que se esta realizando en ese momento
	CHANGE_SPEED			Cambiar la velocidad
	SELECT_SPEED			Selección de velocidad entre rápida (TRUE) y lenta (FALSE)
	CHANGE_ACEL			Cambiar aceleración
	SPEED		ENTERO DOBLE	Velocidad de desplazamiento del motor en mm/min
	ACELERATION			Aceleración de desplazamiento del motor mm/s^2
	DECELERATION			Deceleración de desplazamiento del motor mm/s^2
Referenciado	REF_AUTO_START		BOOL	Iniciar referenciado automático
	REF_MAN			Tomar el punto de referencia
Señales	HWLIMIT_P		BOOL	Final de carrera *hardware* en sentido positivo
	HWLIMIT_M			Final de carrera *hardware* en sentido negativo
	REF_CAM			Leva de referencia *hardware* del eje que controla el SUC

Tabla 7.3.- Interfaces de entrada del SUC

Interfaces de salida del SUC			
Nombre		Tipo	Descripción
ENO		BOOL	Bloque de función que implementa el SUC Activado/Desactivado
STATE		STRUCT	Estado actual del diagrama de estados del SUC, en el que este se encuentra
	WAIT_MODE	BOOL	Esperando señal de entrada que indique el Modo de Operación del SUC
	OPERATIVE	BOOL	El SUC puede realizar alguna operación
	CONFIG	BOOL	EL SUC puede ser configurado
	ERROR	BOOL	EL SUC esta procesando un error
STATUS		STRUCT	Estado general en el que se encuentra el SUC
	ENABLE	BOOL	SUC habilitado
	REFERENCED	BOOL	SUC referenciado
	CONFIGURATED	BOOL	SUC configurado
	TASK_DONE	BOOL	SUC ha terminado la tarea demandada.
	BUSY	BOOL	SUC ocupado (realizando alguna operación)
ALARM		BOOL	SUC muestra alguna alarma
ID_ALARM		ENTERO	Alarma ocurrida en SUC
MESSAGE		BOOL	SUC muestra algún mensaje
ID_MESSAGE		ENTERO	Mensaje mostrado por el SUC
AXIS_POS		ENTERO DOBLE	Posición actual del eje
AXIS_SPEED		ENTERO DOBLE	Velocidad actual del eje
ERROR		BOOL	Existencia de un error en el SUC
NUM_ERROR		ENTERO DOBLE	Identificación del tipo de error

Tabla 7.4.- Interfaces de salida del SUC

7.4.2. Control de vehículo móvil trepador: Implementación en un *Embedded-PC*

La segunda instanciación de ACROSET en el proyecto EFTCoR se está realizando para diseñar la unidad de control de un pequeño vehículo capaz de "trepar" por el casco de un barco gracias a imanes permantentes (**Fig. 7.34-b,d**). Este vehículo transporta sobre la superficie del barco un cabezal de chorreado idéntico al que posiciona la mesa XYZ, de forma que puede proyectar granalla a alta presión y recoger los residuos que se generan al eliminar la pintura de la superficie cuando impacta la granalla. Como el sistema descrito anteriormente, el vehículo es dirigido por un operador humano, pero también debe ser capaz de realizar algunas tareas autónomas, como evitación de obstáculos o ejecución de algunas trayectorias de limpieza sencillas.

La **Fig. 7.38** muestra el CCAS instanciado para este sistema. Como se puede ver, se han implementado dos MUCs diferentes: uno para el control del vehículo (vehicle) y otro para el control del manipulador (manipul). El primero contiene un SUC para controlar cada motor eléctrico de los dos que mueven el vehículo. El otro MUC controla también dos SUCs, uno para cada eje del manipulador que posiciona el cabezal de limpieza sobre el casco. El vehículo usa la misma herramienta que la mesa XYZ, por lo que blasTool es un SUC conceptualmente igual al de la mesa, pero se ha implementado de diferente manera. De hecho, la implementación de todo el sistema es radicalmente distinta a la que se ha realizado para la mesa XYZ, ya que para ésta, la implementación se hacía en un PLC, con la programación en el entorno Step7, y sin embargo, para el vehículo, se hará utilizando el lenguaje Ada95 y usando como plataforma un *Embedded PC* con sistema operativo RT-Linux [Barabanov97].

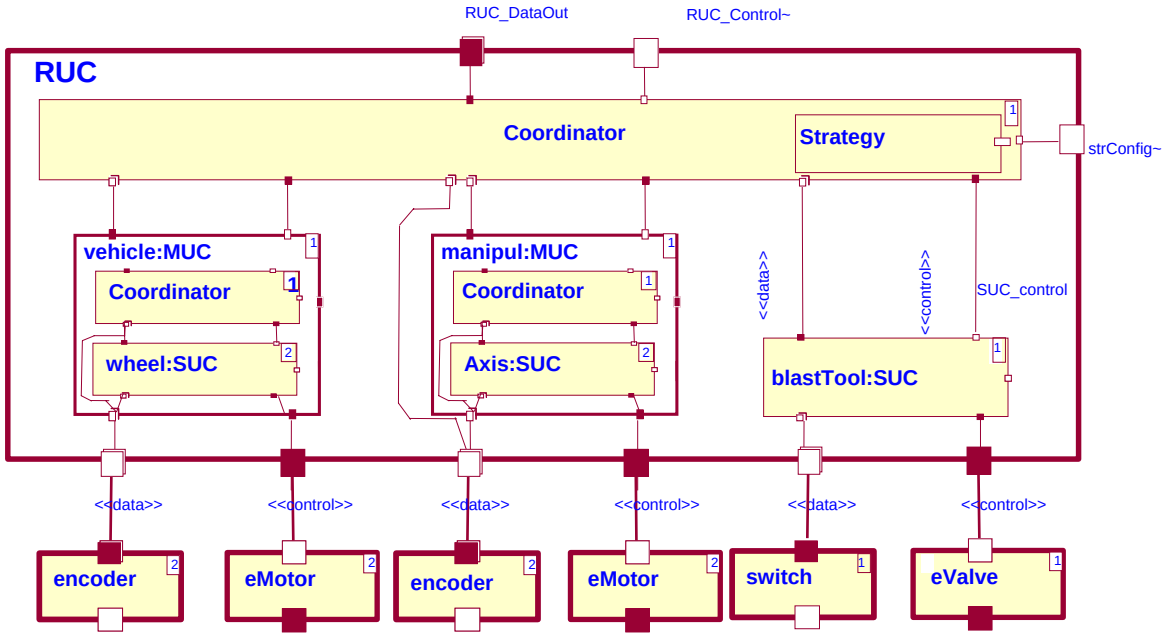


Fig. 7.38.- CCAS para el vehículo móvil trepador

Al contrario de lo que pasaba en el caso de la mesa, los controladores de movimiento no serán implementados mediante *hardware* COTS, sino por medio de paquetes Ada que implementan las interfaces definidas por ACROSET. En este caso, los componentes de abstracción del *hardware* (sensores y actuadores) están fuera del RUC puesto que se deja la posibilidad de añadir módulos de inteligencia que accedan directamente al control de motores (habrá que insertar componentes de arbitraje Arbitrator de bajo nivel) y reciban información de los sensores. Como en el caso anterior, la aplicación se diseña para permitir que un operador humano u otro sistema externo acceda a la funcionalidad del CCAS a través del UIS. En el sistema de inteligencia IS típicamente se programan dos comportamiento: evitación de obstáculos y ejecución de rutas. Los componentes del IS que implementan estos comportamientos obtienen la información necesaria de los sensores del vehículo y generan los comandos oportunos para el CCAS. La integración entre comportamientos inteligentes y los comandos del operador se resuelve en el UIS mediante un Arbitrator de alto nivel.

La plataforma de ejecución para el controlador del vehículo consiste en un *Embedded-PC* montado sobre el robot con posibilidades de expansión basadas en el bus PC/104. Este bus [PC104] es un standard industrial ampliamente utilizado que incorpora múltiples ventajas, como resistencia a vibración, modularidad, robustez mecánica, pequeño factor de forma (96 x 115 mm), bajo consumo, etc. Además, puede ser fácilmente extendido con tarjetas que proporcionan la clase de funciones necesarias por la mayoría de los robots (entrada/salida digital y analógica, controladores de movimiento, expansión PCMCIA, etc). El sistema operativo elegido ha sido RT-Linux [Barabanov97], el cual hace posible tener aplicaciones de tiempo-real ejecutándose mientras se conserva toda la potencia de una distribución Linux (con algunas restricciones). Para poder compilar el programa de control hecho en Ada al núcleo del sistema operativo y que las tareas puedan ejecutarse como *threads* del nucleo, el Grupo de Informática Industrial de la Universidad Politécnica de Valencia ha adaptado el compilador de Ada de GNAT al sistema operativo RTLinux [Masmano03].

7.4.3. Implementación *hardware* de ACROSET en una FPGA

Con las instanciaciones de ACROSET para los sistemas explicados hasta ahora se ha demostrado la flexibilidad de la arquitectura para adaptarse a implementaciones muy diferentes (desde programación completamente *software* en Ada95 a una implementación en autómata programable), incluyendo en

algunas de estas implementaciones componentes *hardware* COTS, lo cual demuestra lo sencillo que resulta cambiar algunos de los componentes *software* por componentes *hardware* comerciales.

Para dar un paso más en la demostración de la flexibilidad de ACROSET, se ha realizado también la implementación de algunos de los componentes típicos de la arquitectura (SUCs y MUCs) en una FPGA, de forma que se demuestra también la posibilidad de implementar ACROSET totalmente en *hardware*. De manera más precisa, se ha realizado una implementación de un MUC que coordina a cuatro SUCs que a su vez controlan tres ejes de un robot didáctico 4U4 y una pinza de agarre (ver **Fig. 7.39**).

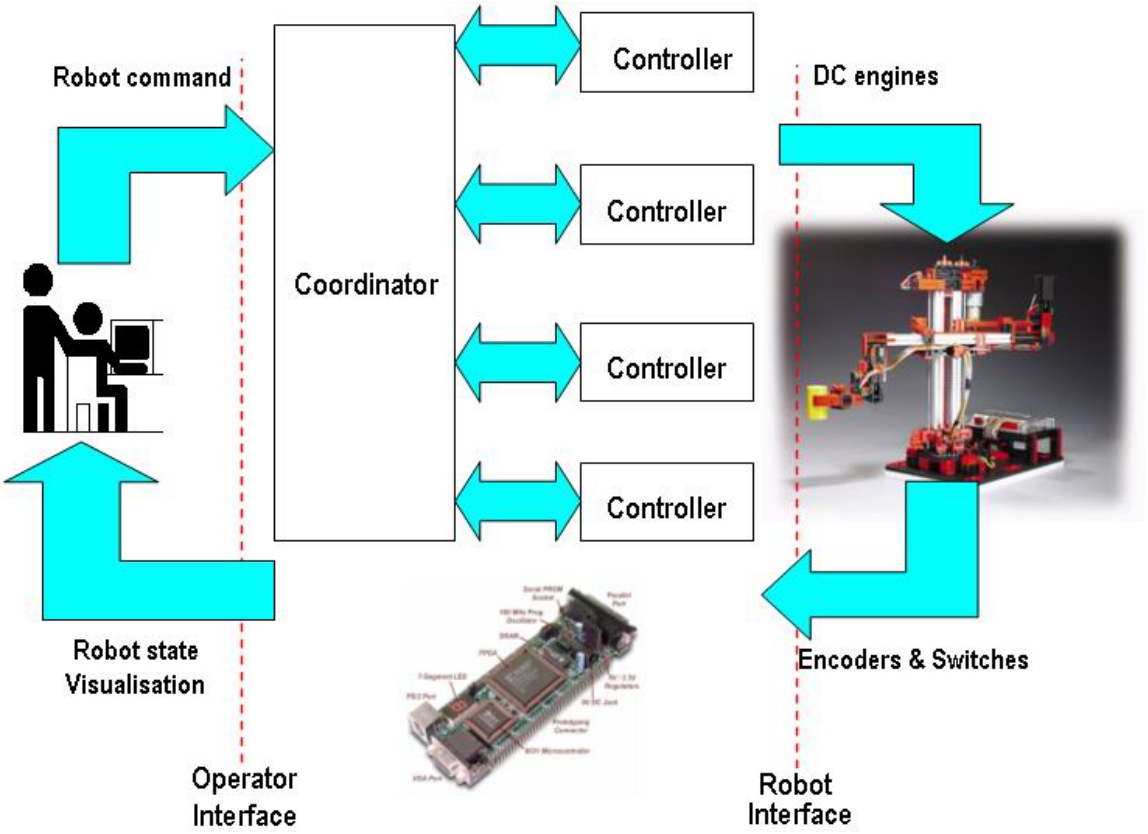


Fig. 7.39.- Sistema de control propuesto para robot didáctico

Partiendo de la arquitectura ACROSET, se ha realizado una implementación totalmente *hardware* de la misma, originando una implementación *VHDL* [VHDL93] jerárquica y parametrizada para su posterior síntesis en una FPGA de Xilinx, de tal manera que pueda servir para diferentes configuraciones de robots [Requena03], [PFC04a]. Al igual que se definió en ACROSET, se dispone de distintos SUCs controladores de un eje (Controller en **Fig. 7.40**), coordinados a su vez el coordinador del MUC (Coordinator en **Fig. 7.40**). En la implementación en VHDL se pretende que las órdenes de control de todos los ejes, accedan al *coordinador* en un único vector, y que sea éste el encargado de distribuir las diferentes órdenes y coordenadas a los distintos *controladores* de cada motor y realizar la coordinación de sus acciones si fuera necesario. A su vez, cada *controlador* se encarga de accionar cada motor y recibir la información de posición mediante el empleo de *encoders* y finales de carrera.

En el diseño *hardware* realizado, la instanciación de uno o más coordinadores o controladores no supone ningún problema a la hora de generar código, puesto que se utilizan genéricos y parametrizables (ver [VHDL93]), de tal manera que se les pueda usar en cualquier diseño, independientemente del ancho de bus y número de ejes y la conectividad y la interoperabilidad están totalmente garantizados. En el caso concreto de esta aplicación, la simplicidad del robot didáctico permite tratar el control de la herramienta como el de cualquier otro eje.

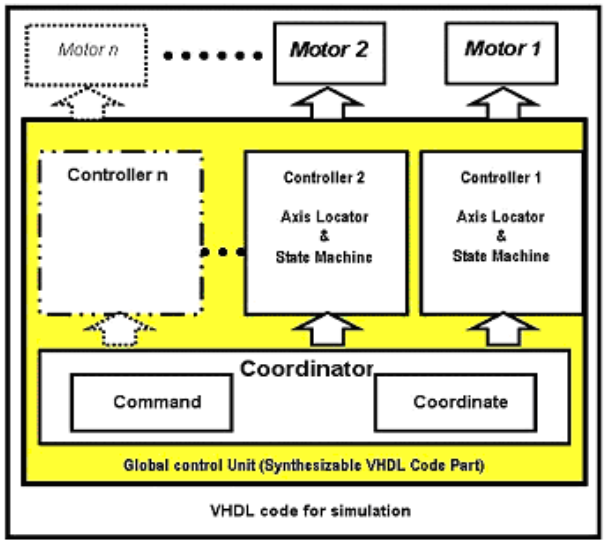


Fig. 7.40.- Diagrama conceptual de programación VHDL

En la **Fig. 7.41** se ofrece una vista del diagrama de bloques de los componentes (aquí si son componentes *hardware*) que integran el SUC y sus *pines* de entrada. Internamente, los bloques están programados en VHDL e implementan el diagrama de estado del controlador y del proceso de control mismo. Una de las grandes ventajas que ofrece el adoptar esta arquitectura es la fácil adaptación de la estructura global al propio sistema de control. En este ejemplo se ha utilizado un control todo/nada, pero se podría optar por emplear otro tipo de sistema de control, como un PID o un control *Fuzzy* implementado en la misma FPGA, sin tener que variar apenas la estructura. Creando librerías de controladores, se podrían seleccionar diferentes tipos de control para cada eje, pudiéndose alcanzar un gran número de configuraciones y pudiendo por tanto diseñar a medida las unidades de control, dependiendo del robot y la aplicación de éste.

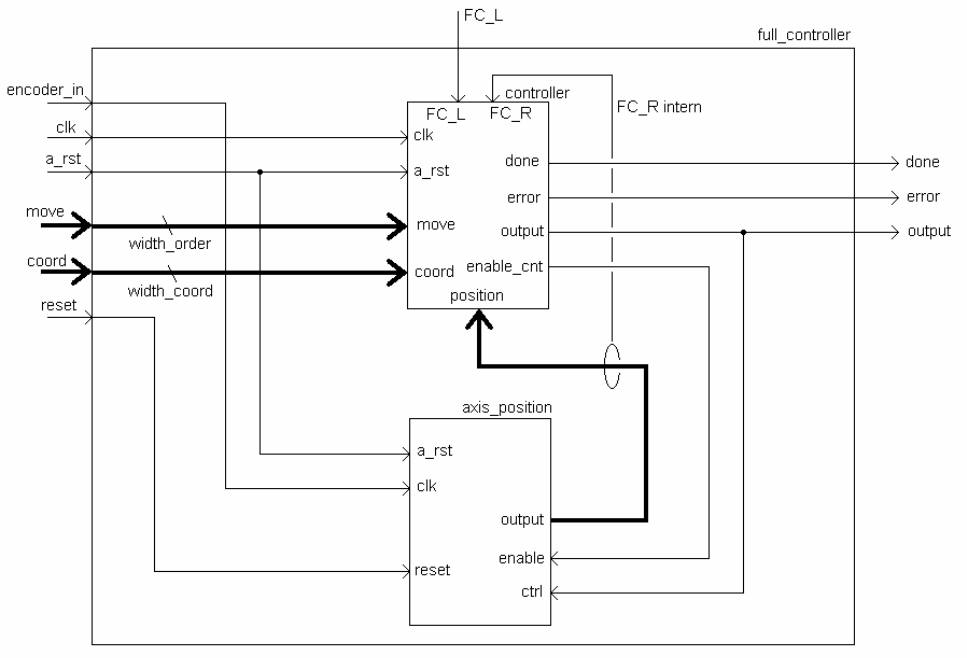


Fig. 7.41.- Diagrama de bloques de la implementación del SUC Controller

El *coordinador* del MUC (**Fig. 7.42**) se encarga de separar las órdenes de movimiento de las coordenadas en cada uno de los comandos y para cada motor. El diseño consta de *'n'* registros para almacenar el código de movimiento y *'n'* para almacenar las coordenadas, siendo *'n'* el número de motores. Los comandos entran al *coordinador* agrupados en un único vector y salen separados en dos *arrays* de vectores de órdenes y de coordenadas.

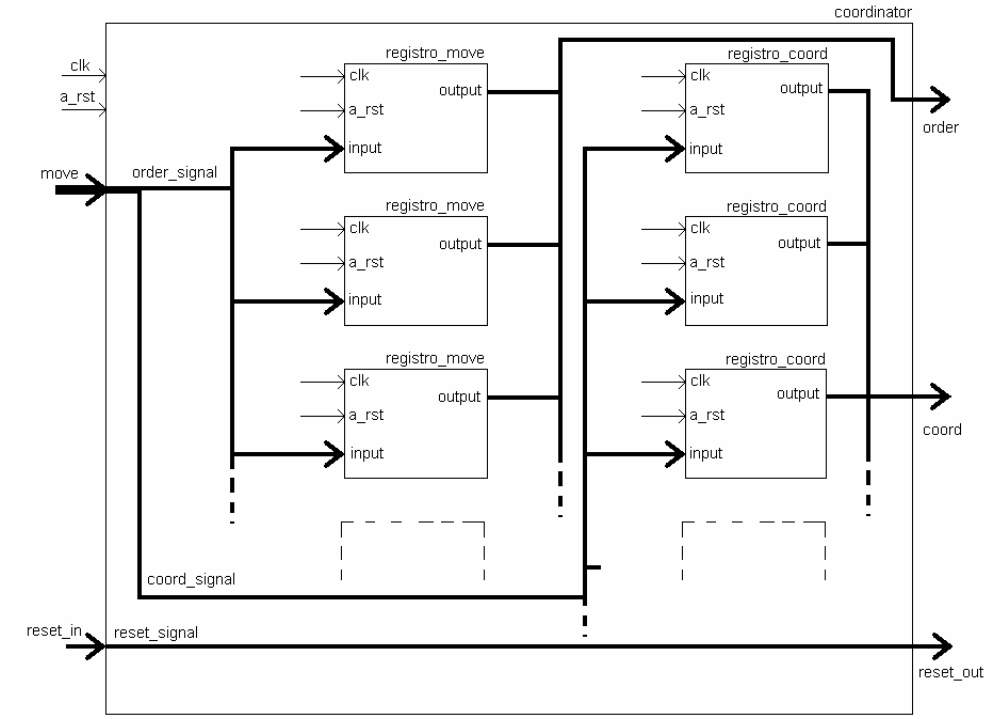


Fig. 7.42.- Diagrama de bloques de la implementación del coordinador del MUC: Registros de orden y posición

Capítulo 8

Conclusiones y Trabajos Futuros

8.1. Conclusiones

Las especiales características de las unidades de control de los sistemas robóticos teleoperados exigen un enfoque muy riguroso para su desarrollo. Una característica esencial de estas unidades es que engloban en un solo sistema elementos software y hardware. Por ésta y otras razones, ya comentadas en esta tesis, las técnicas convencionales de programación no pueden abarcar la complejidad de estos sistemas y, mucho menos, garantizar la reutilización de sus componentes, ni su evolución y mantenimiento a lo largo del tiempo. La arquitectura propuesta en esta tesis, ACROSET, ofrece una respuesta a estos problemas a un nivel de abstracción lo suficientemente alto como para tener en cuenta la variabilidad de las características y requisitos del dominio considerado.

El diseño de ACROSET ha sido un trabajo muy arduo que ha requerido un conocimiento exhaustivo del dominio, teniendo que analizar los aspectos comunes a todos los sistemas y las posibles fuentes de variabilidad. Dicha variabilidad viene dada tanto por las diferencias entre sistemas, como por su evolución a lo largo del tiempo. Así, una de las principales características de ACROSET es su flexibilidad. Sin embargo, el esfuerzo ha merecido la pena: tras haber definido ACROSET y haberla utilizado en varios sistemas, se puede constatar que **disponer de tal arquitectura facilita enormemente el desarrollo de nuevas aplicaciones, puesto que permite la reutilización de modelos y componentes**.

Pero ACROSET es algo más que una arquitectura. Por muy flexible que sea una arquitectura, existe un límite en su capacidad de adaptación frente a nuevos requisitos. Como ya se ha explicado, al diseñar una arquitectura se opta por priorizar el cumplimiento de ciertos requisitos en detrimento de otros. Por muchas similitudes que compartan dos sistemas, si difieren demasiado en sus requisitos, los compromisos de diseño tendrán que ser diferentes y darán lugar a arquitecturas diferentes. Y, sin embargo, las similitudes siguen ahí y con ellas la posibilidad de utilizar componentes comunes. Antes de explicar cómo resuelve ACROSET este dilema, merece la pena recordar que la característica esencial de una arquitectura de referencia es que puede ser utilizada para generar las arquitecturas específicas de sistemas concretos. Para ello, las arquitecturas de referencia deben:

- ✓ Ser estables. La arquitectura debe reducir el espacio de soluciones, definiendo una serie de restricciones o reglas de diseño que sean óptimas para el dominio considerado. Una arquitectura en continua evolución no permite la definición de componentes reutilizables.
- ✓ Proporcionar un conjunto de mecanismos de configuración y extensión que permitan producir arquitecturas para sistemas específicos.

En un dominio tan grande como el que se considera en esta tesis, la estabilidad puede lograrse de dos maneras (no excluyentes):

1. Manteniendo un grado de generalidad muy alto.
2. Definiendo los elementos que aparecen a diferentes niveles de abstracción en todos los sistemas y definiendo las reglas que permiten combinarlos.

ACROSET realiza ambas cosas. Por un lado, ACROSET describe los principales subsistemas que pueden aparecer en cualquier arquitectura de un sistema de teleoperación concreto, así como sus responsabilidades y relaciones. En este sentido es una **arquitectura de referencia**. Por otro lado, describe un modelo en el que se definen los componentes típicos de los sistemas del dominio, así como sus patrones de interacción típicos y deja que el arquitecto los combine de la forma más adecuada para cada caso. Componentes y conectores son ambos entidades de primera clase. En este sentido ACROSET es, más que una arquitectura, un **modelo abstracto de componentes**.

ACROSET ha sido validada mediante su empleo para la generación de arquitecturas de diferentes sistemas (**Tabla 8.1**):

Sistema	Unidad de Control	Implementación de ACROSET
Prototipo de limpieza de barcos GOYA	PC - Industral	Ada95
Controlador de mesa XYZ en proyecto EFTCoR	PLC	KOP y AWL en entorno Step 7 Sistemas *hardware COTS*
Vehículo móvil trepador en el proyecto EFTCoR	*Embedded - PC*	Ada95
Robot didáctico	FPGA	VHDL Sistema *hardware* reconfigurable

Tabla 8.1.- Empleo de ACROSET en la generación de distintas arquitecturas de sistemas

De estas instanciaciones pueden extraerse las siguientes conclusiones referidas a la implementación de la arquitectura, que demuestran el cumplimiento de las premisas marcadas por el diseño de la misma:

- ✓ Las diferentes instanciaciones de la arquitectura se han podido realizar fácilmente a partir de un mismo conjunto de componentes.
- ✓ Se han podido utilizar distintas plataformas de ejecución (un PC industrial, un PLC, una FPGA) y en todas ellas la instanciación de la arquitectura y la posterior implementación de los sistemas ha sido un éxito.
- ✓ La viabilidad de instanciación de la arquitectura no se ha visto afectada por las distintas configuraciones *hardware/software* de las diferentes implementaciones.
- ✓ Ha sido posible utilizar distintos lenguajes para la implementación de los componentes.
- ✓ Se han podido integrar fácilmente componentes *hardware* (tanto COTS como de desarrollo propio), sin que la sustitución posterior de estos componentes suponga la necesidad de modificación del resto de la estructura de control.

- ✓ Se ha probado que la arquitectura puede adaptarse fácilmente a diferentes mecanismos y configuraciones de mecanismos, gracias a la posibilidad de composición y configuración de distintos controladores (SUCs y MUCs) y el cambio de sus patrones de interacción.
- ✓ Ha sido posible variar fácilmente los patrones de interacción entre componentes gracias a la utilización de puertos y conectores. Se ha demostrado esta posibilidad de variación, no sólo en distintos sistemas, sino también en un mismo sistema, donde ha sido fácil cambiar un modo de comunicación entre componentes por otro sin necesidad de cambiar los componentes, solamente cambiando los puertos y conectores.
- ✓ La integración de la inteligencia ha sido razonablemente fácil. Distintos sistemas de inteligencia externos (sistemas de visión, navegadores) se han incorporado utilizando la funcionalidad proporcionada por el *Subsistema de Coordinación, Control y Abstracción de Dispositivos* CCAS, simplemente añadiendo interfaces al *Subsistema de Intérprete de Usuario* UIS y ampliando los mecanismos de arbitraje del componente Arbitrator en este último subsistema. La integración de comportamientos inteligentes preprogramados en el *Subsistema de Inteligencia* IS ha sido también posible. Los mecanismos de utilización del CCAS por parte de este subsistema son los mismos que los de cualquier sistema de inteligencia externo al controlador del robot.

Finalmente, se puede concluir que con esta arquitectura se satisfacen también los requisitos de negocio y objetivos de la organización establecidos al comenzar su desarrollo. La utilización ACROSET favorece el desarrollo rápido y de bajo coste de nuevos sistemas dentro de su dominio de aplicación, sin mermar en ningún caso la calidad del sistema, como se ha demostrado en los proyectos que está realizando el DSIE, en los que ACROSET está sirviendo de base. En esta tesis se ha demostrado esta afirmación con la familia de robots del proyecto EFTCoR. El proyecto GOYA supuso la plataforma inicial para la propuesta de esta arquitectura.

Sobre el método de desarrollo de la arquitectura y la notación utilizada

Aunque se han podido encontrar muchas arquitecturas para robots en la bibliografía (ver Capítulo 4), ha sido muy difícil encontrar ejemplos de procesos de desarrollo para definir arquitecturas de referencia en el dominio robótico, y no se ha encontrado ningún trabajo similar al planteado como objetivo de esta tesis.

En la génesis de los trabajos de esta tesis doctoral se pensó en utilizar una metodología orientada a objetos como COMET o ROPES. Estas metodologías están estructuradas en torno a *casos de uso* y son apropiadas para desarrollar un sistema concreto. Sin embargo, tras algunas pruebas de desarrollo se vio que no son del todo útiles para expresar los conceptos abstractos y generales necesarios para definir una arquitectura de referencia. En el caso de esta tesis, cuyo objetivo es precisamente proponer una arquitectura de referencia para un dominio concreto, pero amplio a la vez, se hizo evidente que tales metodologías no podían ser las líneas que guiaran el proceso de desarrollo. Se buscó entonces una metodología específicamente orientada hacia el diseño de arquitecturas, utilizándose finalmente el *Architecture Based Design Method* ABD [Bachmann00a]. ABD parte de requisitos funcionales y de calidad lo suficientemente amplios y abstractos como para abarcar todo el dominio. Sin embargo, aunque ABD resultó apropiado para el diseño, utiliza abstracciones difíciles de expresar con UML, lo que complica el enlace entre el modelo conceptual resultante y una implementación concreta. Para resolver este extremo se ha adoptado el modelo de 4 vistas propuesto por Hofmeister, Soni y Nord[1] en [Hofmeister00]. Este modelo parte de las mismas entradas que ABD, pero llega a plantear un modelo conceptual que puede ser expresado en UML, donde componentes y conectores son ambos entidades de primera clase. Además, ofrece mecanismos para concretar dicho modelo conceptual en la arquitectura de un sistema determinado.

No obstante, el uso inicial de COMET y ROPES (descrito en [Ortiz02a] y [Ortiz02b]) sirvió para adquirir experiencia en el diseño de sistemas concretos. Esta experiencia se demostró necesaria para poder tomar algunas decisiones clave a la hora de diseñar ACROSET utilizando ABD y 4-V.H.

[1] En adelante 4-V.H.

Además, algunos de los criterios y patrones expuestos en COMET y ROPES han podido ser incorporados en el proceso de desarrollo, especialmente cuando se trata de instanciar ACROSET sobre sistemas concretos. Entre estos cabe destacar los criterios de estructuración de tareas de COMET y los patrones propuestos en ROPES para garantizar la seguridad y los requisitos de tiempo-real. Que los procesos de diseño de sistemas particulares tengan influencia en el diseño de una arquitectura de referencia es algo absolutamente lógico, puesto que una arquitectura de referencia centrada en un dominio se basa en la experiencia adquirida por los desarrolladores en dicho dominio (véase [Mehta00]).

De la utilización de estos métodos pueden extraerse las siguientes conclusiones.

- ✓ Las fases iniciales de ABD y 4-V.H. son muy parecidas y pueden solaparse o utilizarse indistintamente.
- ✓ La orientación de ABD hacia el planteamiento de requisitos abstractos, así como el enfoque hacia la evaluación de la arquitectura, lo hacen más adecuado para el desarrollo de arquitecturas de referencia.
- ✓ Uno de los puntos débiles de ABD es que la ejecución del método termina con unos componentes conceptuales que tienen un grado de generalidad demasiado alto.
- ✓ En todos los métodos revisados orientados hacia la arquitectura (ABD, Hofmeister, PPOA) hay un salto entre la vista conceptual de la arquitectura y su implementación.
- ✓ En la 4-V.H. la vista de módulos no es ni mucho menos definitiva, es un poco confusa en la traducción y además no permite una traslación directa a clases.

No obstante, a pesar de estas dificultades, los métodos han podido ser seguidos o adaptados a las necesidades de la tesis. Precisamente esta combinación se puede considerar una de las aportaciones de la tesis, como se ve en el punto 8.2.

Sobre la notación de la arquitectura

Para describir la arquitectura, las distintas instanciaciones de la misma y diferentes aspectos de implementación, se escogió UML en su revisión 1.4., por las razones expuestas en el Capítulo 3. También se comentó allí que el estándar UML 2.0. no se ha incorporado finalmente a la tesis porque en el momento de redactarla todavía no se ha publicado y no existen herramientas CASE que lo soporten.

Las dificultades encontradas al utilizar UML se han debido más a las herramientas que a la notación. En efecto, la notación de la vista conceptual de 4-V.H. se ha podido seguir en todos sus conceptos, excepto en la expresión de conectores. Hofmeister propone que los conectores y roles se dibujen explícitamente como se puede observar en el Anexo I y en [Hofmeister00], sin embargo, no existe ninguna herramienta que incorpore directamente dicha notación. Para suplir esta carencia, la utilización de Rational Rose Real Time ha permitido realizar los diagramas con componentes y puertos, adoptando para los conectores una representación simple basada en estereotipos (ver Capítulo 6). Las ideas de ROOM [Selic94], UML-RT [Selic98] y Hofmeister son incorporadas al estándar UML 2.0., que se publica el mismo año de presentación de esta tesis, lo que supone una mejora en las posibilidades de expresión de arquitecturas.

Otro problema encontrado con las herramientas CASE es que ninguna de las utilizadas permite expresar la jerarquía de los componentes en un solo diagrama. Para la realización de los diagramas del Capítulo 6 ha sido necesario hacer una superposición de los mismos con objeto de expresar dicha jerarquía en un diagrama completo. Las herramientas de Rational tampoco permiten incorporar paquetes en los diagramas de clases o de objetos, lo cual dificulta la expresión de subsistemas de la arquitectura.

8.2. Aportaciones de esta tesis

En opinión del autor, con esta tesis se han cumplido los objetivos marcados en el primer capítulo, matizándose algunos aspectos y apareciendo algunas novedades según la tesis ha ido evolucionando hasta su conclusión.

Las principales aportaciones de esta tesis son:

- **La arquitectura de referencia para el dominio** de las unidades de control de robots de servicio teleoperados, ACROSET, que define los principales subsistemas que deben o pueden aparecer en cualquier arquitectura concreta, sus responsabilidades y sus relaciones.
- **Un modelo de componentes conceptual** en el que se definen los componentes que pueden aparecer en cualquier sistema del dominio así como sus patrones de interacción típicos. Este modelo incorpora los requisitos definidos en [Pastor02-td] y ampliados en esta tesis, en especial la separación de los componentes de sus patrones de interacción, gracias al uso de puertos y conectores.
  - La separación entre componentes y conectores proporciona un marco extraordinariamente flexible tanto para la definición de arquitecturas para sistemas concretos, como para la evolución de dichas arquitecturas.
  - El hecho de que los componentes sean conceptuales independiza ACROSET de un modelo de componentes concreto. Así, ACROSET puede trasladarse a cualquier modelo de componentes que sea capaz de soportar sus conceptos.

Otras aportaciones de esta tesis son:

- **Un estudio exhaustivo del estado de la técnica** en arquitecturas de control de robots, acotando el dominio hasta los robots de servicio teleoperados.
- Como fruto de este estudio, una **especificación genérica de requisitos** para la unidad de control de este tipo de sistemas (véase Anexo II). Las plantillas de atributo definidas en el Anexo II pueden utilizarse para la generación de requisitos de sistemas concretos y como base de conocimiento aplicable al desarrollo de nuevos sistemas. Dichas plantillas inicialmente descritas en [Pastor02-td] han sido ampliadas y refinadas para el dominio considerado por ACROSET.
- La **caracterización de los casos de uso típicos** en un robot de servicios teleoperado, extensible a todo su dominio, que se puede consultar en el Anexo III de esta tesis
- **Un proceso de desarrollo riguroso** basado en ABD y en las propuestas de la 4-V.H., así como en algunos de los criterios de diseño orientados a objeto y de sistemas de tiempo real utilizados en COMET y ROPES, utilizando UML y sus extensiones para expresar arquitecturas. En el estudio del estado de la técnica, ha sido muy difícil encontrar ejemplos de procesos de desarrollo para definir arquitecturas de referencia en el dominio de la robótica. De hecho, no se ha encontrado ningún trabajo similar al planteado como objetivo de esta tesis. Así, el propio proceso de desarrollo se puede considerar otra aportación de la tesis.
- Un **conjunto de implementaciones de componentes** (sensores, SUCs, etc.) realizadas en Ada95 y en el entorno STEP 7.
- La **diseminación de resultados** que se detalla en la siguiente sección.
- Las propias **conclusiones de la tesis** comentadas en el punto anterior.

Divulgación de resultados

Como resultado de la investigación realizada en el entorno de la tesis y para divulgación de la misma se han publicado varios artículos en revistas y congresos de nivel internacional:

- Ponencias en congresos

2000

[Ortiz00] Ortiz F, Iborra A, Marín F, Álvarez B, Fdez Meroño J.M. "GOYA: A teleoperated system for blasting applied to ships maintenance". 3rd International Conference on Climbing and Walking Robots. CLAWAR´2000. Octubre, 2000. ISBN 1-86058-268-0

2001

[Molina01] Molina JP, Carrión JA, Ortiz F, Alvarez B, Iborra A, Fernandez J.M. "GOYA: A Teleoperated System for Blasting Applied to Ships Maintenance", Consolidation of Technical Advances in the Protective & Marine Coatings Industry PCE 2001. Antwerp, Bélgica, Marzo 2001.

[Álvarez01b] Álvarez B, Iborra A, Sánchez P, Ortiz F, Pastor J.A. "Experiences on the Product Synthesis of Mechatronic Systems using UML in a *Software* Architecture Framework", 1st International Conference on Information Technology in Mechatronics ITM'01, Estambul, Turquía, 1-6 Octubre 2001

[Iborra01] Iborra A, Álvarez B, Ortiz F, Marín F, Fernández JM., "Service Robot for Hull-Blasting", 27th Annual Conference of the IEEE Industrial Electronics Society. Denver, USA, 29 Noviembre 2001.

2002

[Álvarez02] Álvarez B, Ortiz FJ, Martínez A , Sánchez, P, Pastor JA, Iborra A., "Towards a Generic *Software* Architecture for a Service Robot Controller", 15th IFAC World Congress, Barcelona. Julio 2002.

[Ortiz02a] Ortiz F, Martínez A, Álvarez B, Navarro P, Iborra A, Fernández-Meroño JM., "Modelling a *Software* Architecture for Robots Control Using UML and COMET Architectural Design Method", V Jornadas de Tiempo Real. Cartagena, España. Febrero 2002

[Ortiz02b] Ortiz F, Martínez AS, Álvarez B, Iborra A, Fernández JM, "Development of a Control System for Teleoperated Robots Using UML and Ada95", 7th Ada-Europe International Conference on Reliable *Software* Technologies, Viena, Austria. Junio 2002.

2003

[Ortiz03] Ortiz FJ, Álvarez B, Pastor JA, Sánchez P., "A Case Study in Performance Evaluation of Real-Time Teleoperation *Software* Architectures using UML-MAST", 8th Ada-Europe International Conference on Reliable *Software* Technologies. Toulouse, Francia. Junio 2003.

2004

[Pastor04] Pator JA, Álvarez B, Sánchez P, Ortiz F, "A Layered Architectural Component Model for Service Teleoperated Robots", Jornadas de Ingeniería del Software y Bases de Datos. Málaga, España. Noviembre 2004

- Artículos en revistas

De las comunicaciones a congresos mencionadas, dos de ellas, [Ortiz02b] y [Ortiz03], son publicadas completas en las revistas listada en el SCI/JCR con índice de impacto:

[Ortiz02b] P. 113-124. Lecture Notes on Computer Science - LNCS 2361 Ed. Springer-Verlag, ISSN 0302-9743. Alemania. Junio 2002

[Ortiz03] P. 417-418. Lecture Notes on Computer Science - LNCS 3655 Ed. Springer-Verlag, ISSN 0302-9743. Alemania. Junio 2003

Y otra, que no ha sido expuesta en ningún congreso, ha sido publicada en *Dedicated Systems e-Magazine*:

[Requena03] Requena F, Ortiz FJ, Suardíaz J, Iborra A, "Adaptation of Real-Time *Software* Robot Control Units to Generic *Hardware* Architectures". Dedicated Systems e-Magazine (publicación en Internet) (http://www.dedicated-systems.com/Magazine/emagazine/magazineform.aspx?year=2003&quarter=2). Septiembre 2003. Bélgica.

- Pendientes de aceptación

[Ortiz05a] Ortiz F, Pastor JA, Alonso D, Álvarez D, F. Losilla, "A Reference Architecture for Managing Variability among Teleoperated Service Robots", 20th IEEE International Conference on Robotics and Automation. Barcelona. Abril 2005.

[Ortiz05b] Ortiz FJ, Álvarez B, Losilla F, Rodríguez D, Ortega N, "An Implementation of a Teleoperated Robot Control Architecture on a PLC And Field-Bus Based Platform", 16th IFAC World Congress, Praga, República Checa. Julio 2005.

[Ortiz05c] Ortiz F, Alonso D, Álvarez D, Pastor JA, "A Reference Control Architecture for Service Robots Implemented on a Climbing Vehicle". 10th Ada-Europe International Conference on Reliable *Software* Technologies. York, UK. Junio 2005.

8.3. Trabajos futuros

Las actividades futuras que pueden derivar de esta Tesis Doctoral son:

A corto plazo

- ✓ Adoptar los mecanismos de UML 2.0. para expresar arquitecturas.
- ✓ Finalizar el proyecto EFTCoR y validar las ventajas de ACROSET sobre el proyecto terminado.
- ✓ Realizar nuevas combinaciones *hardware-software* y probar ACROSET en nuevas plataformas.
- ✓ La formalización, cuando sea posible, de los requisitos descritos en las plantillas de atributo definidas en el Anexo II y su transformación para que puedan ser mantenidas con ayuda de herramientas.
- ✓ Utilizar infraestructuras de componentes como Java-Beans, Corba, .NET, etc. En el entorno del proyecto DYNAMICA ya se están desarrollando componentes .NET a partir de ACROSET utilizando PRISMA [Pérez03].

A medio y largo plazo:

- ✓ Proponer un nuevo método de desarrollo de arquitecturas de referencia en el marco de adopción de técnicas formales o semi-formales, basado en los estudios realizados en esta tesis sobre diferentes métodos de diseño y la utilización combinada de ellos que se ha hecho.
- ✓ Validar la arquitectura en otros sistemas que pertenezcan al dominio considerado, pero donde la diferencia de sus requisitos funcionales sea mayor que entre los sistemas en los que se ha aplicado ACROSET hasta ahora (que estén fuera del subdominio de robots de limpieza de barcos).
- ✓ Incorporar nuevos comportamientos inteligentes. Profundizar en el estudio de la integración de inteligencia y control para conseguir comportamientos autónomos más complejos.

- ✓ Estudiar si la arquitectura puede ser ampliada a otros dominios, principalmente aumentando la autonomía del robot.
- ✓ Proponer un *framework* de componentes que lleve más lejos las aportaciones de ACROSET, pudiendo reutilizarse directamente dichos componentes.
- ✓ A más largo plazo, y partiendo de dicho *framework* de componentes, se podría realizar un entorno de desarrollo gráfico para composición rápida de unidades de control de robots de servicio teleoperados.

Muchos de los trabajos futuros aquí señalados se han iniciado ya con el proyecto **DYNAMICA**. Este proyecto impulsado por grupos de varias Universidades [DYNAMICA03], entre ellos, el grupo DSIE de la UPCT, surge para aprovechar las tendencias más actuales en el desarrollo de sistemas *software*, el Desarrollo de *Software* Basado en Componentes (DSBC) y el Desarrollo de *Software* Orientado a Aspectos (DSOA) [Kiczales97] con el objetivo de establecer un marco de desarrollo en el que puedan definirse diferentes arquitecturas, reutilizando los componentes comunes e intentando satisfacer los requisitos contradictorios. Para conseguir estos objetivos será necesaria la identificación de aquellos aspectos de los sistemas de operación que puedan ser susceptibles de variación y por tanto impliquen una modificación en el modelo arquitectónico empleado. El autor de esta tesis, como integrante del DSIE colabora en el subproyecto **ANCLA** [ANCLA03] que abordará dicho estudio de variabilidad. El trabajo de tesis aquí presentado aporta un modelo de componentes conceptuales que sirve de punto de partida para el proyecto DYNAMICA

El objetivo que se pretende alcanzar con el subproyecto ANCLA es la especificación y diseño de una arquitectura dinámica de *software* para el desarrollo de sistemas de teleoperación, que pueda ser configurada en función de las necesidades particulares de cada sistema, reutilizando de esta forma no sólo componentes sino también patrones de interacción entre los mismos. Para ello, será necesario alcanzar los siguientes objetivos parciales:

- ✓ La identificación de aquellos *aspectos* tanto funcionales como no funcionales del sistema de teleoperación que puedan ser susceptibles de variación y, por tanto, impliquen una modificación en el modelo arquitectónico empleado.
- ✓ La obtención de una especificación formal de dicha variabilidad (tanto la variabilidad propia de los requisitos funcionales como las diferentes necesidades de interacción entre componentes). El objetivo es hacer uso del lenguaje de definición de componentes de PRISMA [Pérez03] y de su lenguaje de configuración para definir instancias y especificar la topología del sistema de teleoperación.
- ✓ El rediseño arquitectónico del *software* para sistemas de teleoperación dedicados al mantenimiento de cascos de buques. Será necesaria la integración del diseño dinámico resultante en el diseño arquitectónico original.
- ✓ Implementación del *software* en un prototipo para pruebas que incluya la dinámica de control requerida y desarrollada en el subproyecto CARATE del mismo proyecto DYNAMICA.

Anexo I

Conceptos Fundamentales de ABD y 4 Vistas de Hofmeister

1.1. Introducción

En este anexo se ofrece un resumen de los aspectos fundamentales de los procesos de desarrollo asociados a ABD y 4 vistas de Hofmeister[1]. Finalmente se comentan las similitudes y diferencias de ambos métodos, que justifican la utilización de ambos en esta tesis.

1.2. El método de diseño basado en la arquitectura (ABD)

El Método de Diseño Basado en la Arquitectura o ABD (de sus siglas en inglés *Architecture Based Design Method*) es un método para el diseño de arquitecturas software para líneas de producto y sistemas de larga vida operativa, desarrollado por el SEI (*Software Engineering Institute*)de la CMU (*Carnegie Mellon University*) en Estados Unidos. El método se basa en tres pilares:

- Descomposición funcional del problema, basada en los criterios de bajo acoplamiento y alta cohesión.
- Realización de los requisitos de calidad y de negocio a través de la elección de los estilos arquitectónicos adecuados.
- El uso de *plantillas software*. Las plantillas software en el contexto del ABD se refieren a los patrones de interacción de los elementos de la arquitectura entre sí y con la infraestructura que utilizan y al conjunto de responsabilidades comunes que dichos elementos deben asumir.

Las entradas del método están constituidas por la lista de requisitos y restricciones que debe satisfacer la arquitectura. Los requisitos se clasifican dentro de tres categorías: requisitos funcionales, requisitos

[1] En adelante, 4-V.H.

de calidad y requisitos de negocio e, independientemente de la categoría a la que pertenezcan, se dividen en abstractos y concretos. **Los requisitos abstractos se utilizan para generar la arquitectura, los concretos para validarla**.

El ABD considera que cada sistema se compone de dos partes, una correspondiente a la aplicación y otra correspondiente a la infraestructura que la soporta y sobre la cual se ejecuta (sistema operativo, *middleware*, servidores externos, bases de datos pre-existentes, etc). El ABD considera ambas partes y ambas contribuyen en la descomposición del sistema y en la definición de la arquitectura.

El método ABD descompone el sistema en subsistemas recursivamente. Es decir, las mismas reglas que se aplican para descomponer el sistema en susbsistemas se aplican a su vez para descomponer dichos subsistemas en otros más simples. Los requisitos funcionales se logran asignando una parte de la funcionalidad del sistema a cada uno de los subsistemas o componentes resultantes de las sucesivas descomposiciones del mismo. Los requisitos no funcionales se alcanzan mediante la selección del estilo arquitectónico que mejor se adapte a los mismos. Cada descomposición se examina por un lado desde las perspectivas de las vistas lógica, de concurrencia y de despliegue y por otro desde el punto de vista de las plantillas software. En los siguientes apartados se presentarán los fundamentos del método y su terminología. Una vez descritos estos aspectos se pasará a la descripción del método.

1.2.1. Elementos de diseño y vistas arquitectónicas

El ADB considera elementos de diseño al *sistema*, a cada uno de los *subsistemas* que resultan de las sucesivas descomposiciones del mismo y a los *componentes conceptuales* que pertenecen a los subsistemas finales (aquellos que no se han dividido en otros subsistemas). Cada uno de estos elementos tiene asignada una parte de la funcionalidad que se corresponde con las responsabilidades que debe asumir. Asimismo, todos los elementos de diseño tienen una *interfaz conceptual* que describe las entradas que reciben y las salidas que producen.

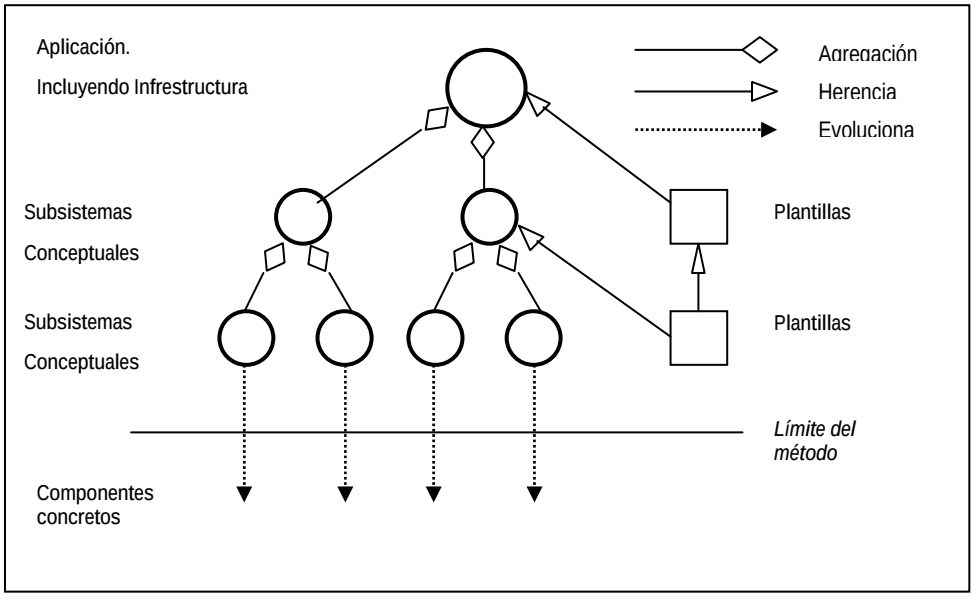


Fig. 1.1.- Descomposición del sistema en elementos de diseño

La **Fig. 1.1** muestra la relación entre los distintos elementos de diseño. Al nivel más alto se encuentra el *sistema*, que se descompone en dos o más *subsistemas conceptuales*. Puesto que el método es recursivo estos subsistemas pueden descomponerse en otros de acuerdo con las mismas reglas que rigen la descomposición del sistema. En el último nivel se encuentran los *componentes conceptuales*. Los *componentes concretos* que aparecen en la parte inferior de la **Fig. 1.1** son componentes para los cuales se ha alcanzado un compromiso de diseño acerca de los objetos y procesos que contienen y de la estructura de los mismos. El ABD no se ocupa de estos componentes, que deben ser diseñados a

partir de los componentes conceptuales utilizando otros métodos. En cada uno de los niveles se definen *plantillas software* que describirán más adelante.

Las vistas de la arquitectura que utiliza el ABD se basan en el modelo 4+1 de Krutchen, si bien con algunas restricciones. Así, el ABD no utiliza la vista de desarrollo y la vista de procesos, que el ABD denomina de concurrencia, se limita a examinar aspectos relacionados con la existencia de múltiples usuarios, el acceso a recursos compartidos, el arranque del sistema y la identificación de las actividades que pueden ejecutarse en paralelo. La vista de casos de uso se completa con la definición de escenarios que describan los requisitos no funcionales con el mayor grado de detalle posible.

1.2.2. Plantillas software e infraestructura del sistema

El concepto de plantilla software en el ABD describe una taxonomía de los elementos de diseño basada en sus patrones de interacción y en las responsabilidades que los diferentes elementos de diseño deben asumir. Los criterios que definen dicha taxonomía y determinan el contenido de las plantillas son los siguientes:

- **Interacción con los servicios.** Patrones que describen como un tipo determinado de elementos de diseño interacciona con los servicios provistos por el resto de los elementos de diseño. Por ejemplo, si hay un servicio de diagnóstico, todos los elementos que proporcionan datos a dicho servicio deben usar el mismo protocolo.
- **Interacción con la infraestructura.** Patrones que describen como un tipo determinado de elementos de diseño interacciona con la infraestructura. Por ejemplo, se determina utilizar sólo los servicios del sistema operativo definidos en el estándar POSIX.
- **Responsabilidades comunes[2].** Responsabilidades que aunque no estén relacionadas con la interacción con los servicios o con la infraestructura son comunes a todos los elementos de diseño de un tipo determinado. Por ejemplo, la forma en que un determinado tipo de elementos debe llevar a cabo el tratamiento de las excepciones.

Las plantillas software sirven para:

- ✓ Facilitar la integración de componentes y subsistemas. Todos los elementos de diseño que compartan un determinado patrón de interacción pueden integrarse de la misma manera.
- ✓ Identificar los patrones de comportamiento comunes a diferentes elementos de diseño y que por tanto son susceptibles de implementarse en componentes reutilizables en todos esos elementos.
- ✓ Construir un esqueleto del sistema lo más independiente posible de la funcionalidad asignada a los elementos de diseño. Si el esqueleto del sistema no depende de la funcionalidad se facilita el crecimiento incremental de las aplicaciones. El uso de plantillas constituye en este sentido una forma de abordar las variantes de funcionalidad.
- ✓ Facilitar el uso de modelos formales, los cuales se construyen a partir de los patrones de interacción entre los elementos y no de su funcionalidad específica.

1.2.3. Directrices de la arquitectura, estilos arquitectónicos y división de la funcionalidad

Las directrices de la arquitectura (del inglés *architectural drivers*) son la combinación de requisitos funcionales, de calidad y negocio, que conforman la arquitectura. Si se cumplen dichas directrices, entonces el sistema puede ser diseñado satisfactoriamente. En el nivel más alto, las directrices de la

[2] En [Bachmann00] se denomina a esas responsabilidades "*citizenship responsabilities*", responsabilidades de ciudadanía u obligaciones del buen ciudadano

arquitectura se determinan a partir del propósito del sistema y de los factores de negocio más críticos. A niveles más detallados, las directrices se determinan de los requisitos de un subsistema o un componente conceptual particular.

Por ejemplo, la finalidad de un sistema de simulación de vuelo es entrenar pilotos (propósito), lo cual requiere una alta fidelidad en la simulación y la ejecución de las maniobras en tiempo real. La empresa desea desarrollar una línea de producto (factor de negocio), lo que requiere una alta adaptabilidad del sistema a las características de diferentes aeronaves.

Las directrices de la arquitectura no dependen de los detalles de los requisitos funcionales, sino de la abstracción de los mismos. En el ejemplo de arriba, el hecho de que las aeronaves a simular tengan uno, dos o cuatro motores, no es determinante, sino la capacidad de procesar grandes cantidades de datos en tiempo real. Diferentes directrices implican diferentes arquitecturas. Uno de los pasos más importantes del ABD es la selección de un estilo o conjunto de estilos arquitectónicos dominantes, bajo el criterio de que el estilo seleccionado debe ser el que mejor se adapte a las directrices de la arquitectura. Como se ha explicado en el Capítulo 3, un estilo arquitectónico consiste en una colección de componentes acompañada por la descripción de los patrones de interacción entre los mismos. Pero los componentes a los que se refiere esta definición son conceptuales y no se les supone ninguna funcionalidad específica. El estilo seleccionado no soporta funcionalidad alguna hasta que la funcionalidad del sistema sea repartida y asignada a sus componentes. Y esta asignación de funcionalidad nuevamente depende de las directrices de la arquitectura. En el ejemplo del simulador de vuelo la adaptabilidad es una *directriz* que *dicta* que los cálculos correspondientes a cada motor se hagan en una unidad de computación separada. En el ABD el término *división de funcionalidad* se refiere al reparto de los requisitos funcionales entre los elementos de diseño en función de los criterios que determinen las directrices arquitectónicas.

1.2.4. El ABD en el ciclo de vida del sistema. Entradas y salidas del método

El ABD puede inscribirse dentro del Proceso de Desarrollo Basado en la Arquitectura (*Architecture Based Development Process*) definido por Bass y Kazman en [Bass99]. Dicho proceso se resume en la **Fig. 1.2**. En ella puede observarse que el proceso de diseño se encuentra entre la definición de requisitos y el análisis de la arquitectura. El proceso puede repetirse de forma iterativa, resolviendo en cada iteración una parte del diseño.

1.2.4.1. Entradas del ABD

Las entradas del método están constituidas por la lista de requisitos y restricciones que debe satisfacer la arquitectura. Los requisitos se clasifican dentro de tres categorías: requisitos **funcionales**, requisitos de **calidad** y requisitos de **negocio** e, independientemente de la categoría a la que pertenezcan, se dividen en abstractos y concretos. Los requisitos abstractos se utilizan para generar la arquitectura, los concretos para validarla**.**

Diseñar una arquitectura para una familia de sistemas implica ser capaz de prever las variaciones entre los distintos miembros de la familia de forma que la arquitectura pueda adaptarse a las mismas. Las **variantes** son aspectos que cambian de un sistema a otro y pueden referirse a cambios en la funcionalidad, en la plataforma o en el entorno. Los **invariantes** son aquellos aspectos comunes a todos los sistemas del dominio, línea de producto o familia de sistemas considerada.

1.2.4.2. Requisitos funcionales abstractos

Las arquitecturas de referencia deben ser aplicables a todos los sistemas de una familia o dominio, por tanto los requisitos que debe cumplir son distintos que los que debe satisfacer la arquitectura de un sistema específico. Un ejemplo:

Arquitectura de un Sistema de Teleoperación concreto:

Las alarmas deben ser procesadas antes de que transcurran 50 ms desde su activación.

Arquitectura de referencia:

Las alarmas deben ser procesadas antes de que transcurra un intervalo de tiempo determinado.

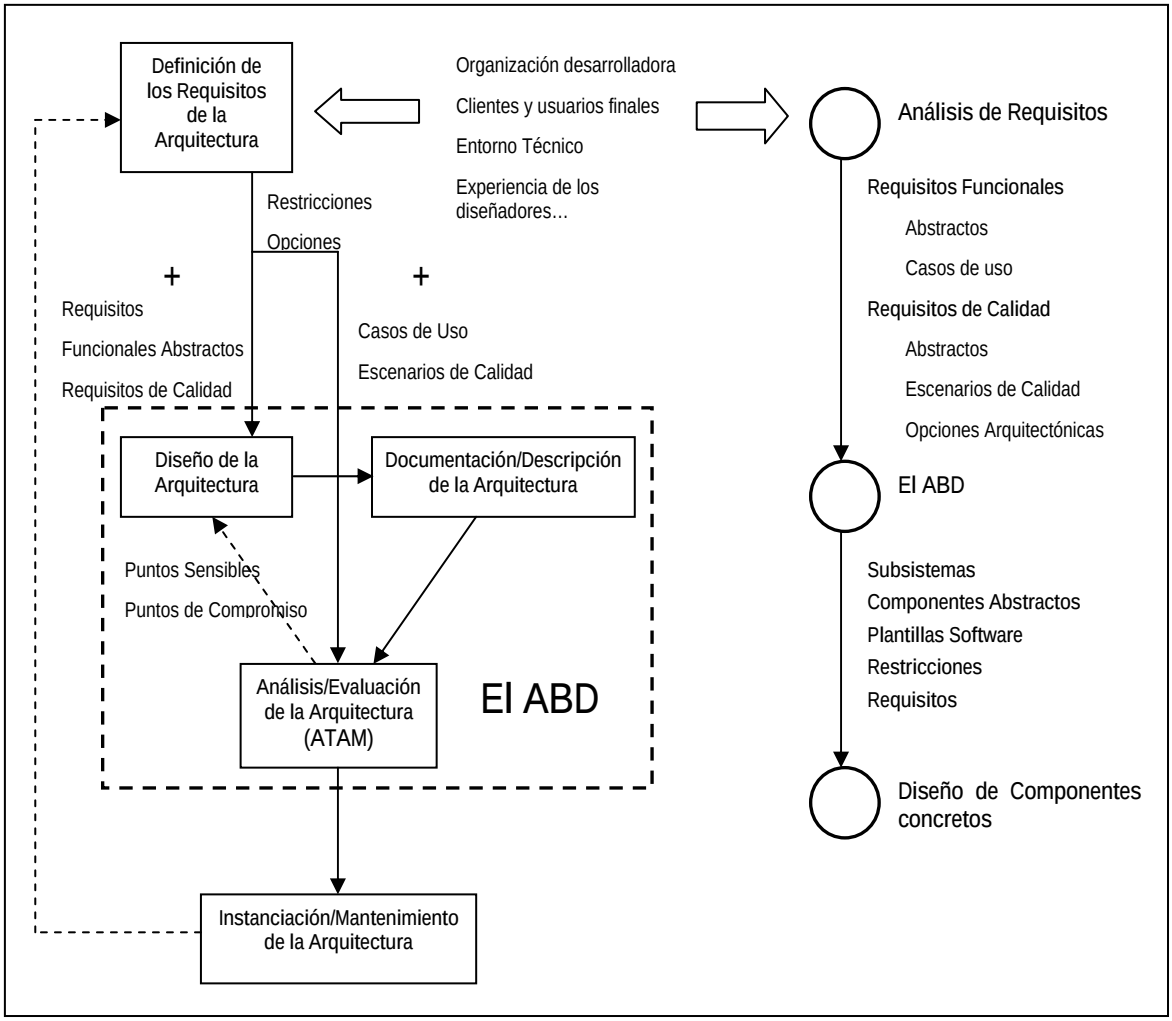


Fig. 1.2.- El método de desarrollo basado en arquitectura y el ABD [Bass99]

Los requisitos de una arquitectura de referencia se caracterizan por su falta de especificidad. Esta falta de especificidad es necesaria para capturar las variantes de funcionalidad. En ocasiones el rango de las variantes se conoce, al menos en uno de sus extremos y el requisito de arriba puede hacerse más específico:

Las alarmas deben ser procesadas antes de que transcurra un intervalo que en los sistemas más críticos es de 40 ms.

O, si se conocen ambos extremos,

Las alarmas deben ser procesadas antes de que transcurra un intervalo de tiempo que oscila, según los sistemas, entre 40 y 500 ms.

En otras, no es posible hacer suposiciones de este estilo, el requisito queda completamente indeterminado y sólo puede formularse de una manera *abstracta,* mediante sentencias como las del ejemplo anterior. Una arquitectura de referencia debe incluir mecanismos que aseguren que los requisitos *abstractos* pueden alcanzarse *en general*. En el ejemplo anterior la arquitectura debería incluir algún mecanismo que permita planificar las tareas y asignarles prioridades. La especificación

de los requisitos en un alto nivel de abstracción es necesaria tanto para capturar las variantes de funcionalidad como por el hecho de que dichas variantes no son siempre conocidas con antelación.

1.2.4.3. Casos de uso

Un caso de uso es la *descripción concreta* de una interacción entre uno o más actores y el sistema. Los casos de uso representan un caso específico de interacción entre los actores y un sistema de la familia, en el que los requisitos funcionales pueden expresarse de forma precisa. Los casos de uso concretos pueden ser instancias específicas de los abstractos. Los casos de uso son fáciles de generar y su número crece rápidamente. Es necesario clasificarlos y priorizarlos de forma que sea posible seleccionar un número manejable de los mismos suficientemente representativo del comportamiento del sistema.

1.2.4.4. Calidades abstractas y requisitos de negocio

El ABD denomina *calidades* a los requisitos no funcionales o, según la terminología de Bass [Bass98], a los *atributos de calidad no observables en tiempo de ejecución*. Al igual, que los requisitos funcionales, las calidades deben estar expresadas de forma abstracta, para que abarquen a todos los sistemas de la familia. Pero incluso, dentro de ese nivel de abstracción hay que caracterizar las calidades de la forma más precisa posible. Decir que la arquitectura de teleoperación objeto de este trabajo de tesis debe ser modificable es lo mismo que no decir nada. Decir que debe ser modificable con respecto al tipo de robots y herramientas es decir algo. Decir que el sistema debe ser modificable respecto a determinados cambios en el robot y las herramientas, especificando hasta donde sea posible la naturaleza de dichos cambios, es empezar a entender el problema.

El ABD no distingue entre calidades y requisitos de negocio, ya que estos últimos deben traducirse a atributos de calidad concretos.

1.2.4.5. Escenarios de calidad

Al igual que los casos de uso hacen concretos los requisitos funcionales abstractos, los escenarios de calidad hacen concretas las *calidades* abstractas. Por ejemplo:

Calidad abstracta:

> *El sistema debe a adaptarse a robots de distintas características.*

Escenarios de calidad:

> *El sistema debe adoptarse a robots con diferente número de grados de libertad.*
>
> *El sistema debe admitir el uso de diferentes algoritmos para el cálculo de la transformada cinemática inversa.*
>
> *El sistema debe adaptarse a diferentes tipos de accionadores eléctricos, neumáticos o hidráulicos.*

Al igual que los casos de uso los escenarios de calidad son fáciles de generar y su número crece rápidamente. Nuevamente es necesario clasificarlos y priorizarlos en función de las directrices de la arquitectura, con objeto de examinar sólo los más relevantes.

1.2.4.6. Restriccione.

Las restricciones son decisiones de diseño preestablecidas. La mayor parte de estas decisiones vienen determinadas por los factores de negocio (inversiones previas en componentes software comerciales, interoperabilidad con sistemas ya existentes, uso de las plataformas disponibles, utilización de un determinado sistema operativo, etc), aunque en ocasiones responden a motivos técnicos (un arquitecto

basándose en su experiencia puede imponer un sistema operativo determinado, aun cuando existan otros que ofrezcan servicios similares).

1.2.4.7. Opciones arquitectónicas

Para cada uno de los requisitos funcionales y de calidad existe un estilo o conjunto de estilos que permiten su cumplimiento. Algunos requisitos pueden admitir varias opciones, otros sólo una. Las opciones deben enumerarse para proceder más adelante a la selección de las más apropiadas. La enumeración de estas opciones es parte del proceso de diseño, sin embargo a menudo esta enumeración se realiza durante la fase de especificación de requisitos, constituyendo en este caso una entrada más del ABD.

1.2.5. Actividades del ABD. Ejecución del método.

1.2.5.1. Los pasos del método

De acuerdo con el ABD, cada elemento de diseño se caracteriza por el conjunto de requisitos que debe satisfacer (funcionales y de calidad), la plantilla asociada al mismo y una serie de restricciones de diseño. Cuando un elemento se descompone el resultado son dos o más elementos hijos, cada uno con sus propios requisitos, plantillas y restricciones. La **Fig. 1.3** muestra la secuencia de descomposición de un elemento de diseño que, como puede observarse, comienza por la definición de la vista lógica, continúa con la definición de las vistas de concurrencia y de despliegue, prosigue con la verificación de las mismas de acuerdo con los requisitos de partida y acaba con un bucle de realimentación entre la verificación y la definición de la vista lógica. Aunque no se muestran en la figura se contempla la posibilidad de otros bucles de realimentación entre la verificación y las vistas de concurrencia y despliegue y entre las diferentes vistas entre sí. Las definiciones de las diferentes vistas no son procesos aislados. Por ejemplo, al definirse la vista de concurrencia puede identificarse funcionalidad adicional que debe incorporarse a la vista lógica.

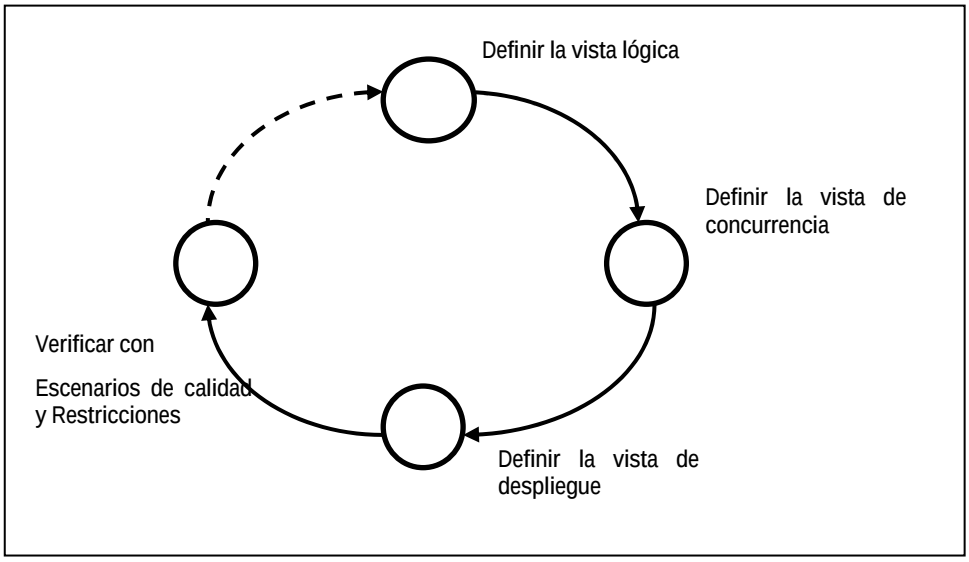


Fig. 1.3.- Pasos para la descomposición de un elemento de diseño

La **Fig. 1.1** mostraba el árbol de elementos de diseño que resulta después de la ejecución del método. El ABD no impone ningún orden de generación del mismo, aunque establece algunos criterios. Por ejemplo, si el dominio está convenientemente caracterizado es más fácil recorrer el árbol en horizontal pues se tiene una idea bastante precisa de los subsistemas principales. Si se detectan riesgos que deben

abordarse en etapas tempranas puede ser necesaria la implementación de un prototipo y el recorrido *en profundidad* de los subsistemas implicados.

1.2.5.2. Definición de la vista lógica

La **Fig. 1.4** muestra los pasos a seguir para la definición de la vista lógica: descomposición de la funcionalidad, selección del estilo arquitectónico básico, asignación de funcionalidad a los componentes definidos en el estilo, refinamiento de las plantillas y verificación con casos de uso y escenarios de cambio. La división de la funcionalidad permite descomponer la misma en diferentes grupos según los criterios que dicten las directrices de la arquitectura, las cuales determinan además el estilo arquitectónico mediante el que deben organizarse los elementos resultantes de la descomposición. Dicho estilo arquitectónico define una serie de componentes a los cuales debe asignárseles cada uno de los grupos de funcionalidad resultantes de la descomposición.

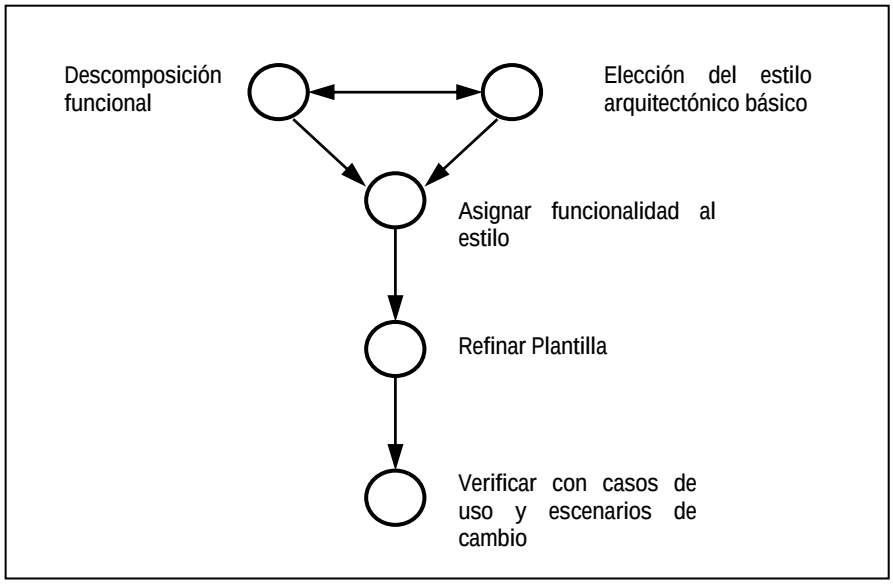


Fig. 1.4.- Pasos para la descomposición de un elemento de diseño

1.- División de la funcionalidad

Cada elemento de diseño tiene asignadas unas responsabilidades para satisfacer sus requisitos de diseño. Estas responsabilidades, que están constituidas por la funcionalidad que realiza el elemento y los atributos de calidad que le son propios, pueden dividirse en distintos grupos que pueden ser asignados a los elementos de diseño resultantes de su descomposición. Los criterios de división vienen determinados por los atributos de calidad que debe satisfacer el elemento de diseño.

Si la modificabilidad es el atributo más importante, los grupos de responsabilidades se establecen según los siguientes criterios:

- ✓ Coherencia Funcional. Bajo acoplamiento y alta cohesión.
- ✓ Aquellas responsabilidades que exhiben patrones de datos y comportamiento similares deben agruparse juntas.
- ✓ Los diferentes grupos deben definirse en niveles de abstracción similares.
- ✓ Localización de las responsabilidades. Aquellas responsabilidades que proporcionan servicios a otros servicios no deben ser agrupadas con las responsabilidades puramente locales.

Si el atributo clave es el rendimiento la división debe basarse en:

- ✓ La minimización del flujo de datos entre los elementos.

- ✓ La minimización de la frecuencia de los cálculos.

El objetivo de la descomposición es generar grupos de funcionalidad con una granularidad tal que permita:

- ✓ Representar una descomposición de las responsabilidades asignadas al elemento de diseño.
- ✓ Mantener el número de grupos dentro de un rango manejable. (El método aconseja entre 3 y 15).
- ✓ Cada uno de los grupos generados lleva asociados:
- ✓ Una caracterización de sus variantes y dependencias.
- ✓ Una descripción de sus flujos de entrada y salida de datos.
- ✓ Una relación de las entidades externas con las que interacciona.

2.- Elección del estilo arquitectónico

Cada uno de los elementos de diseño tiene asociado un estilo arquitectónico o patrón dominante, que define la forma en que el elemento de diseño realiza sus responsabilidades. La selección de dicho estilo se basa en las directrices arquitectónicas correspondientes a dicho elemento. Una vez que el estilo arquitectónico ha sido elegido, debe adaptarse a los requisitos de calidad del elemento de diseño y a las opciones arquitectónicas[3] identificadas para satisfacer dichos requisitos. Para ello, se examina cada uno de los requisitos de calidad y se determina si tiene alguna relevancia en el elemento que va a ser descompuesto. Si es así, se elige alguna de las opciones arquitectónicas apropiadas para conseguirlos y se aplica al estilo arquitectónico seleccionado. Es aconsejable utilizar la misma opción para un mismo atributo siempre que sea posible.

El resultado de elegir y refinar un estilo arquitectónico es una colección de *tipos* de componentes. Ejemplos de tipos de componentes son clientes, servidores, dispositivos virtuales, etc., que aunque en principio están vacíos de funcionalidad, sugieren un determinado tipo de comportamiento o de interacción.

3.- Asignación de funcionalidad al estilo y definición de interfaces conceptuales

La elección de un determinado estilo arquitectónico implica el uso de ciertos tipos de componentes. Los grupos de funcionalidad resultantes de la descomposición de la misma deben ser asignados a los componentes determinados por el estilo. Esto implica determinar cuantas instancias deben existir de cada uno de los tipos de componentes y que funcionalidad debe incluir cada uno. Los componentes resultantes de esta asignación son los elementos de diseño en los que va a descomponerse el elemento de diseño considerado. El siguiente paso es identificar las interfaces conceptuales de cada uno de estos elementos de diseño.

Cada uno de los elementos resultantes de la descomposición tiene asociadas una colección de plantillas heredadas de sus padres en la jerarquía de elementos que define ABD. El refinado de las plantillas consiste en ir añadiéndoles responsabilidades a medida que avanza el método. Estas responsabilidades que, recordemos, se refieren a los patrones de interacción del elemento con los servicios y la infraestructura y a patrones de comportamiento comunes a diversos elementos, deben implementarse mediante componentes concretos en algún momento del proceso de diseño.

1.2.5.3. Definición de la vista de concurrencia

El propósito de describir una vista de concurrencia es determinar aquellas actividades del sistema que deban ejecutarse en paralelo, con objeto de descubrir los puntos de sincronización y de acceso a recursos a compartidos.

---

[3] En ocasiones esas opciones determinan el estilo dominante, en otros casos lo modulan o refinan.

El examen de la vista de concurrencia se realiza en términos de *threads virtuales*. Cada *thread* virtual es un camino de ejecución secuencial dentro del programa, un modelo dinámico o alguna otra representación de un flujo de control. No se consideran *threads* virtuales a aquellos definidos por el sistema operativo. Los *threads* virtuales se utilizan para describir diferentes secuencias de actividades y sus puntos de sincronización y de acceso a recursos compartidos. Estos aspectos pueden implicar la adición de nueva funcionalidad en la forma de gestores de recursos compartidos que gestionen el acceso a los mismos, que deben incorporarse a la vista lógica asignándolos a los elementos de diseño o a las plantillas correspondientes. La utilización de casos de uso que describan los efectos que producen varios usuarios accediendo simultáneamente al sistema o que examinen los efectos del arranque y parada del sistema es especialmente útil durante el desarrollo de esta vista.

1.2.5.4. Definición de la vista de despliegue

Si el sistema incluye más de un procesador es necesario describir como pueden asignarse elementos de diseño a los mismos. Un *thread* virtual no tiene una ubicación determinada y puede trasladarse a través de una red de un procesador a otro. ABD denomina *thread físico* a aquel que se ejecuta en un procesador determinado.

1.2.6. Conclusiones y críticas al método

El ABD está pensado para el diseño de arquitecturas software para líneas de producto y tiene en consideración la falta de especificidad de los requisitos y la variabilidad entre los diferentes sistemas de un determinado dominio. Asimismo, el ABD tiene muy en cuenta los atributos de calidad del sistema y sus interacciones, los cuales son determinantes para elegir los estilos arquitectónicos más apropiados. Los requisitos abstractos y las opciones arquitectónicas pueden expresarse usando los escenarios generales y las plantillas de mecanismo y de atributo descritas en el capítulo 3. Además, el ABD define las pautas generales para dividir el sistema en subsistemas, para asignar responsabilidades a los mismos y para identificar plantillas de comportamiento y de relación con la infraestructura aplicable a todos los subsistemas. Finalmente, como se muestra en la **Fig. 1.2**, el ABD define un proceso de diseño en el que la evaluación de la arquitectura es un paso explícitamente considerado e incluye la realización de un *mini*-proceso ATAM en cada iteración.

Sin embargo, como puede observarse en la **Fig. 1.1**, la ejecución del ABD termina una vez que se han definido los componentes conceptuales de más bajo nivel. Dichos componentes siguen teniendo un grado de generalidad muy alto. El ABD no proporciona criterios para la estructuración de los mismos en clases y tareas, ni para su despliegue en nodos físicos, ni para la traducción de sus interfaces conceptuales en interfaces concretas.

Aunque una descripción de la arquitectura al nivel de componentes conceptuales es por sí misma un artefacto útil, pues define las características generales de los elementos del sistema y sus patrones de interacción, sentando las bases para sucesivos refinamientos, no es suficiente para la implementación de un sistema, ni para definir componentes software reutilizables. Tales componentes deben ofrecer y recibir servicios a través de interfaces concretas con signaturas bien definidas. La definición de tales interfaces requiere la existencia de un modelo de componentes que defina las reglas mediante las cuales puede realizarse dicha definición. El ABD permitirá definir las líneas generales de la arquitectura.

1.3. Las 4 Vistas de Hofmeister

Hofmeister, Soni y Nord en [Hofmeister00] proponen un proceso de desarrollo dirigido por cuatro vistas de una arquitectura: *conceptual*, de *módulos*, de *ejecución* y de *código*. Esta separación está más cerca de un proceso de desarrollo que del concepto de 4+1 vistas [Krutchen95] que utiliza UML, debido a que cada vista es más bien una fase del desarrollo *top-down* de la arquitectura de un sistema,

comenzando por el modelo conceptual de cómo debe ser dicha arquitectura y finalizando la vista de código, a nivel de implementación.

1.3.1. Propósito de las 4 vistas

Cada una de las vistas cumple un cometido en la descripción de una arquitectura, pero también se corresponde a una fase del desarrollo de la arquitectura de un sistema:

Vista conceptual. Describe la arquitectura en términos de sus mayores elementos de diseño y las relaciones entre ellos. En esta vista, el arquitecto diseña las características funcionales del sistema. Los elementos básicos utilizados son los componentes, con puertos que definen los servicios requeridos y proporcionados por los componentes, y los conectores entre puertos, que se encargan de *transportar* los mensajes que definen dichas interacciones.

En la vista conceptual, los problemas y soluciones se ven en términos del dominio. Deberían ser relativamente independientes de una técnica particular software o hardware. El tipo de preguntas que debería responder esta vista son del tipo:

> *¿Cómo cumple el sistema los requisitos?*
>
> *¿Cómo se integran los componentes COTS y como interaccionan con el resto del sistema?*
>
> *¿Cómo se incorpora un hardware o software específico de dominio al sistema?*
>
> *¿Cómo se ofrece soporte a las líneas de producto?*
>
> *¿Cómo se pueden minimizar el impacto de los cambios en los requisitos o en el dominio de aplicación?*

Vista de módulos. En esta vista, los componentes, puertos y conectores de la vista conceptual se traducen a subsistemas y módulos. Aquí, el arquitecto decide cómo la solución conceptual se puede realizar con las tecnologías actuales. Las principales preguntas a las que responde esta vista son:

> *¿Cómo se puede traducir el producto a una plataforma software?*
>
> *¿Qué servicios utiliza el sistema, y dónde lo hace?*
>
> *¿Qué dependencias entre módulos deben ser minimizadas?*
>
> *¿Cómo se puede maximizar la reutilización de los módulos y subsistemas?*
>
> *¿Qué técnicas se pueden utilizar para aislar al producto de los cambios en las platarformas software y hardware comerciales utilizadas?*

Vista de ejecución. Es la vista en tiempo de ejecución del sistema, que define la traducción de módulos en entidades de ejecución y sus atributos, como uso de memoria y asignación de *hardware*. Aquí se toman decisiones como si es necesario usar una librería, o si se usan *threads* o procesos, aunque estas decisiones a su vez, realimentan la vista de módulos y provocan cambios en ella.

Una parte importante de la vista de ejecución es el flujo de control. La vista conceptual describe el flujo de control lógico, mientras que la vistas de ejecución refleja dicho flujo de control desde el punto de vista de la plataforma de ejecución. La vista de ejecución se corresponde con las vistas de tareas y de despliegue de los procesos de desarrollo orientados a objetos y responde al siguiente tipo de preguntas:

> *¿Cómo puede alcanzar el sistema sus requisitos de rendimiento, recuperación ante fallos y reconfiguración?*
>
> *¿Cómo se puede optimizar el uso de recursos disponibles?*
>
> *¿Cómo se cumple con las necesidades de concurrencia, redundancia y distribución sin añadir demasiada complejidad a los algoritmos de control?*
>
> *¿Cómo se puede minimizar el impacto de los cambios en la plataforma de ejecución?*

Vista de código. Es la encargada de capturar cómo los módulos e interfaces se traducen a código fuente, y cómo las entidades de ejecución se convierten en ficheros ejecutables o componentes software o hardware COTS. Se trata también la organización del código en directorios afecta a la posibilidad de construir un sistema y se hace más importante cuando hay varias versiones o líneas de producto. Son importantes para esta vista las siguientes cuestiones:

> *¿Cómo puede reducirse el tiempo y esfuerzo dedicado a realizar nuevas versiones de un producto?*
>
> *¿Cómo se puede reducir el tiempo de desarrollo final del producto?*
>
> *¿Qué herramientas de desarrollo son necesarias?*
>
> *¿Cómo se debe hacer el proceso de integración y test?*

Como se puede observar, el desarrollo asociado a las 4-V.H. presta especial atención a los requisitos relacionados con la ***modificabilidad*** de los sistemas y arquitecturas diseñados, a la integración del hardware y del software, la utilización de COTS, así como el enfoque de líneas de productos. Todo ello, unido a la notación utilizada (especialmente la vista conceptual) y la orientación hacia el diseño de arquitecturas, hacen a las 4-V.H. adecuadas para ser utilizadas en esta tesis.

1.3.2. Elementos que componen las vistas

Cada una de las cuatro vistas tiene elementos particulares, con sus interfaces, atributos, comportamiento y relaciones entre sí, como se verá en este punto. Alguna de estas vistas también tiene una configuración, que restringe a los elementos definiendo las reglas que deben cumplir para un sistema particular.

Vista conceptual

Los elementos básicos de la vista conceptual son los **componentes**, con **puertos** a través de los cuales ocurren las interacciones, y **conectores** con ***roles*** que definen cómo se conectan a los puertos. Cada puerto tiene un **protocolo** asociado que establece de qué modo deben ser ordenadas las operaciones de entrada y salida.

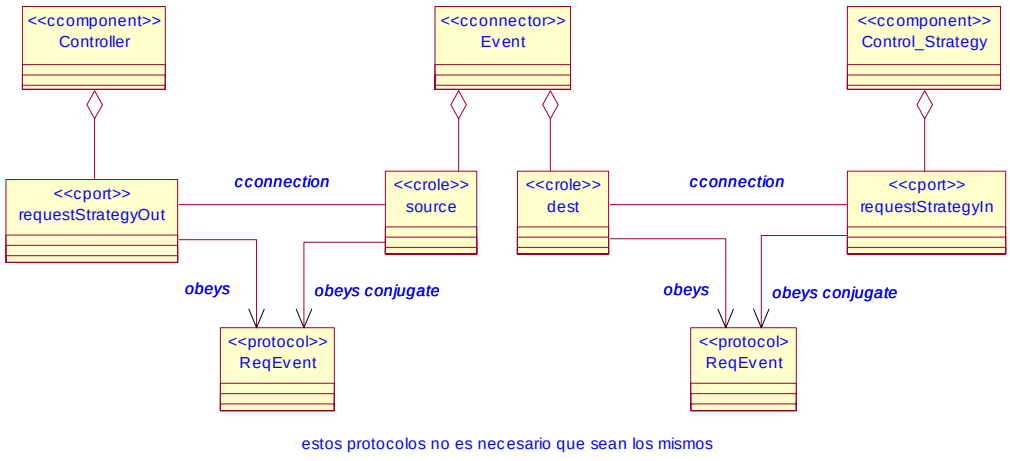


Fig. 1.5.- Ejemplo de conector tipo *Event* entre *Controller* y *Control_Strategy*

La conexión entre puertos (y por consecuencia, entre componentes) se hace a través de los conectores. Un conector permite separar la funcionalidad de los componentes de sus patrones de interacción, puesto que están incluidos en los propios conectores. Al igual que los componentes tienen puertos, los conectores tienen *roles* que son los que se conectan al puerto compatible en el componente, siguiendo un protocolo que será el conjugado del puerto al que se conecta (ver **Fig. 1.5**). Es inmediato inferir

que los conectores son a su vez un tipo de componente que ofrece los mismos tipos de puertos que cualquier otro y cuya responsabilidad específica es gestionar protocolos y reglas de negocio. Pero es un componente especial, ya que desde el punto de vista de los componentes que enlaza no existe. Si los componentes percibieran la existencia de un mediador dependerían de alguna forma de él.

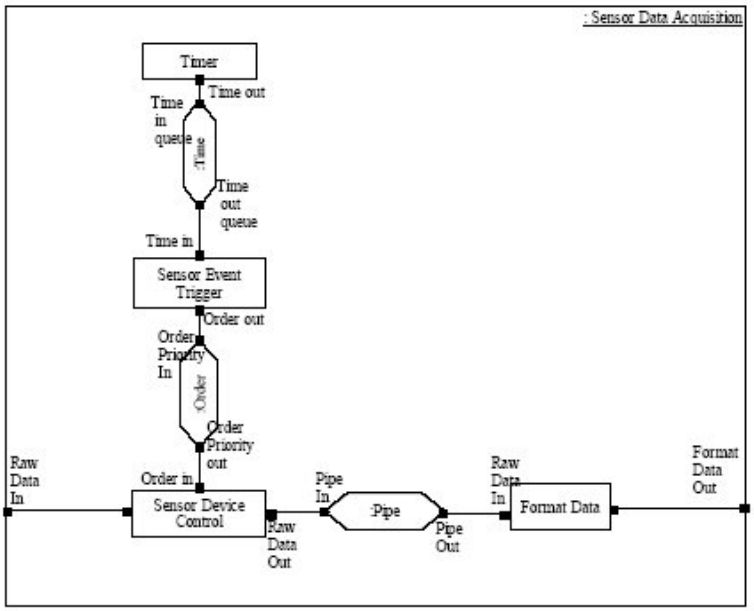


Fig. 1.6.- Vista conceptual

En la vista conceptual, los componentes, puertos y conectores se representan como clases estereotipadas. Se usa:

- Un diagrama de clases UML para mostrar la configuración estática. Para simplificar la vista, se utilizan representaciones gráficas distintas para cada clase estereotipada, así, los componentes serán *"cajas"*, los puertos son pequeños cuadrados en el borde de dichas cajas, los conectores son etiquetas con círculos en los extremos que representan los roles (ver **Fig. 1.6**)
- Declaración de protocolos inspirada en ROOM [Selic94] y en los diagramas de secuencia o de estado UML para mostrar los protocolos a los cuales se adhieren los puertos (ver **Fig. 1.7**).
- Diagramas de secuencia UML que muestran una secuencia particular de interacciones entre un grupo de componentes.

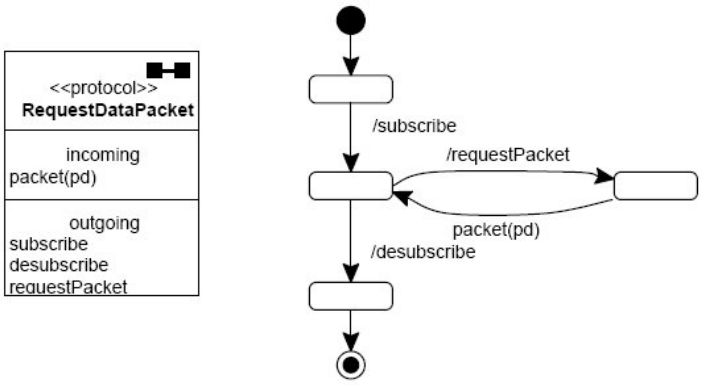


Fig. 1.7.- Protocolo para un puerto de entrada de paquetes

Vista de módulos

En la vista de módulos, los subsistemas se descomponen en módulos, y los módulos se asignan a capas en concordancia con sus dependencias tipo *usa*. No hay configuración para la vista de módulos porque ésta define los módulos y sus relaciones inherentes, pero no cómo se deben combinar para un producto particular.

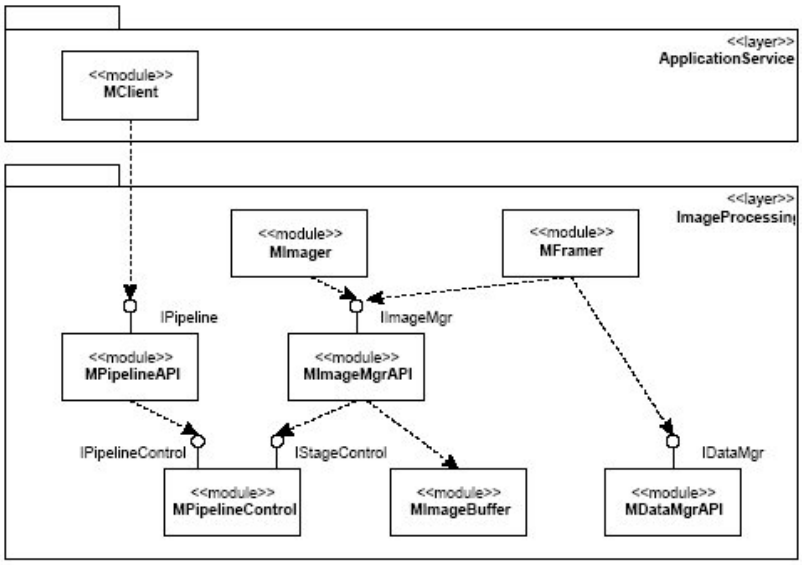


Fig. 1.8.- Vista de módulos. Se presentan agrupados en capas

En algunas ocasiones, los puertos, conectores y componentes se combinan dando lugar a un solo módulo. Hofmeister, Soni y Nord no usa la notación de *componente* UML para expresar un módulo porque en esta vista los módulos son abstractos, no son los módulos físicos de código fuente. Para representar las interfaces entre módulos se utiliza la representación UML (pequeño círculo conectado al módulo). En la **Fig. 1.8**, además de un ejemplo de interfaces y módulos, se representan las relaciones "*usa*" entre ellos y la inclusión de estos módulos en capas. Los módulos se representan como clases estereotipadas y los subsistemas y capas como paquetes estereotipados según la notación de UML. También se usan:

- Tablas para describir la transformación entre la vista conceptual y la de módulos.
- Diagramas de paquetes UML para mostrar la descomposición en subsistemas.
- Diagramas de clases UML para mostrar las dependencias entre módulos.
- Diagramas de paquetes UML para mostrar las dependencias entre capas y la asignación de módulos a las capas.

Diagrama de ejecución

La vista de ejecución de la arquitectura determina cómo los módulos se combinan en un determinado producto mostrando cómo se asignan a procesos en tiempo de ejecución. Aquí, estos procesos y los mecanismos de comunicación entre ellos se enlazan para formar una configuración determinada (ejemplo en **Fig. 1.9**)

Módulos, enlaces de comunicación y procesos presentan multiplicidad explícita, lo cual tiene las mismas implicaciones que la configuración conceptual, es decir, se muestra todas las configuraciones permitidas en un solo diagrama.

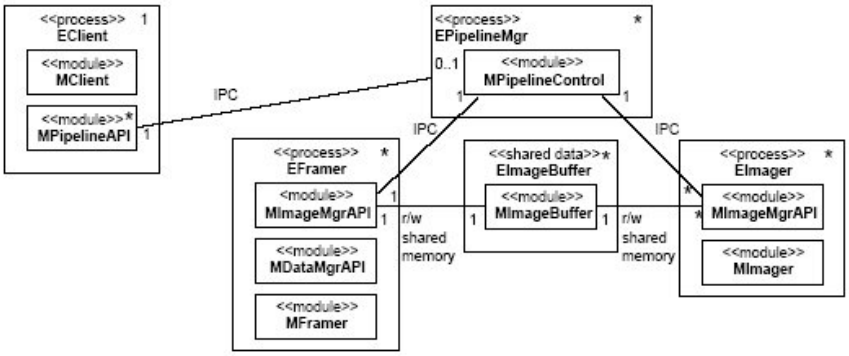


Fig. 1.9.- Vista de ejecución

Para la vista de ejecución, los procesos en tiempo de ejecución se representan como clases estereotipadas y los enlaces de comunicación como asociaciones. También se usan:

- Diagramas de clases UML para mostrar la configuración estática.
- Diagramas de secuencia UML para mostrar el comportamiento dinámico de una configuración, o las transiciones entre configuraciones.
- Diagramas de estado o de secuencia para mostrar el protocolo de un enlace de comunicación.

Vista de código

La vista de código de la arquitectura contiene los ficheros y directorios, y como pasa en la vista de módulos, no tiene una configuración. Las relaciones definidas en la vista de código aplican a todos los productos, no sólo a uno en particular.

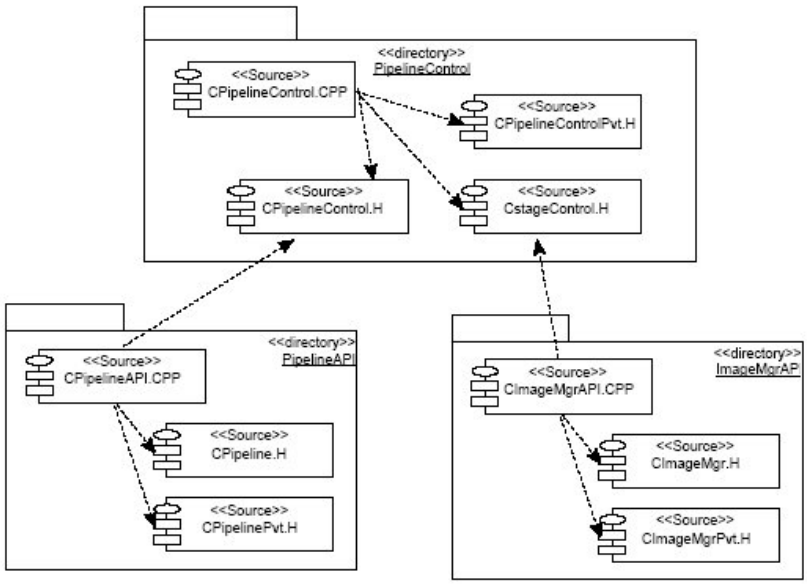


Fig. 1.10.- Vista de código

En esta vista, los módulos e interfaces de la vista de módulos son particionados en ficheros de código fuente de un lenguaje de programación particular. Los ficheros fuente son organizados en directorios. Se usa la notación UML de *"componente"* para representar los ficheros, y la notación de paquetes para los directorios (ver **Fig. 1.10**). Ambos, ficheros y directorios tienen estereotipos para clarificar su significado. La **Fig. 1.10** muestra también las relaciones de dependencia "*incluye"* mediante el estereotipo <<*include*>> si el diagrama contiene más de un tipo de dependencia.

1.3.3. Proceso de desarrollo asociado a las vistas

Cada una de las vistas tiene sus propias tareas de diseño, pero todas se estructuran de una manera similar, incluyendo las tareas: *Análisis Global, Tareas de Diseño Centrales* y *Tareas de Diseño Finales*).

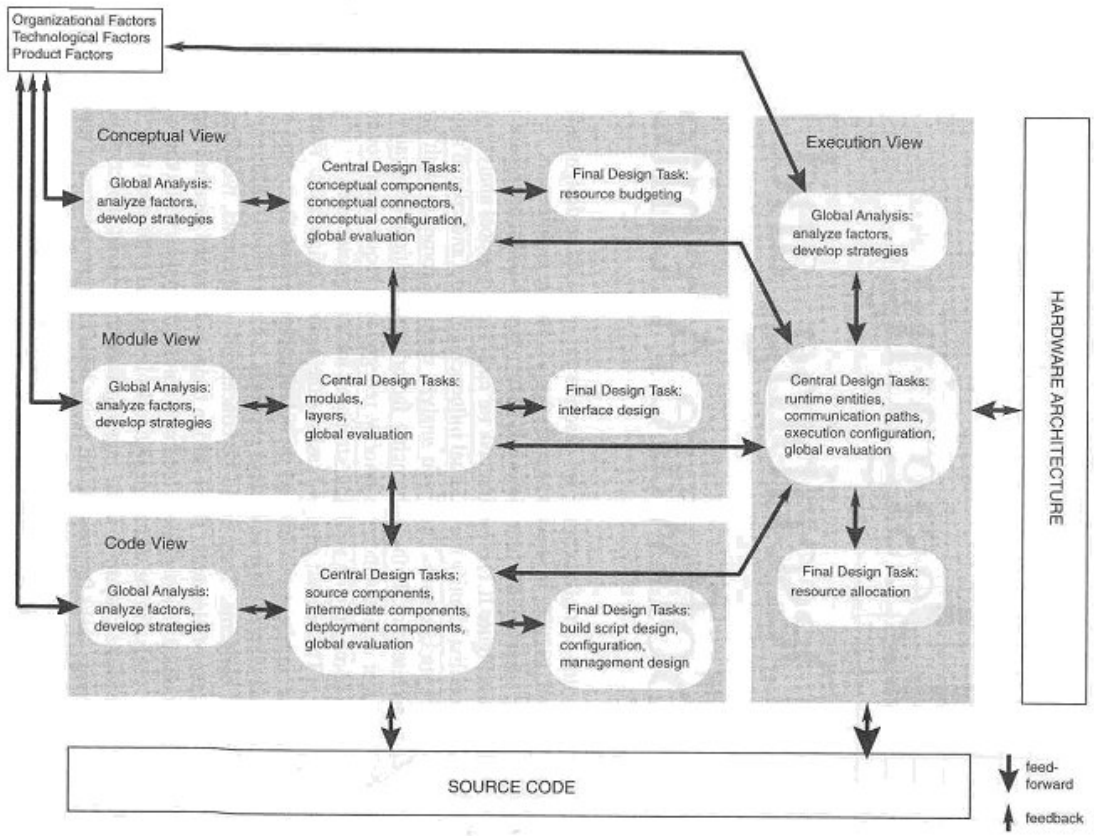


Fig. 1.11.- Tareas de diseño de las 4 vistas de Hofmeister

En la fase de **análisis global**, se identifican los factores externos y los requisitos críticos que podrían afectar a la arquitectura. Una vez identificados, se analizan con objeto de proponer las estrategias de diseño de la arquitectura que permitan cumplir estos factores y requisitos. Si no pueden ser satisfechos, se debe decidir quien tiene prioridad, renegociando un requisito o cambiando alguno de los factores externos para que puedan surgir las estrategias que finalmente se adoptarán en el diseño de la arquitectura.

Al realizar las **tareas de diseño centrales** se definen en primer lugar los elementos de la vista de la arquitectura que se trate, así como las relaciones entre ellos, para después definir cómo se configuran estos elementos (si es apropiado para esa vista). Cada una de las vistas tiene diferentes tipos de elementos, como se ha visto, desde componentes conceptuales a módulos, procesos, ficheros, etc.

Entre estas tareas de diseño centrales se incluye una evaluación global, que no produce una salida separada, sino que es una tarea en curso en la fase de diseño central. Las tareas de diseño centrales de la mayoría de las vistas tienen múltiples fuentes de entrada, así que uno de los aspectos de la evaluación global consiste en decidir cual de las fuentes de información se usa en cada momento. El segundo aspecto abordado por la evaluación global radica es la comprobación de si las decisiones tomadas en el diseño tienen impacto en otras tareas de diseño previas. Si es así, se debe volver a ellas y revisar el diseño. El tercer aspecto de la evaluación es evaluar si las decisiones de diseño realizadas en esta fase tienen impacto entre sí.

La **tarea de diseño final** de cada una de las vistas está menos acoplada con el resto de tareas. Ejemplos de estas tareas pueden ser la definición de interfaces y la estimación de recursos necesarios. Puede haber algún tipo de realimentación hacia las tareas de diseño centrales, pero la mayoría de ellas no deberían verse afectadas por esta tarea de diseño final.

En la fase de **análisis global de la vista conceptual** se analizan factores que influyen en el diseño de la arquitectura y proponen estrategias arquitectónicas de manera parecida a la que propone ABD como entrada del método, y que en esta tesis se ha realizado en el Capítulo 5. De forma parecida a las plantillas de atributo utilizadas que propone ABD, en las 4-V.H. se proponen unas plantillas de clasificación de factores (organizacionales, tecnológicos y de producto) como las que se muestran en el ejemplo de la **Tabla 1.1**.

Factor (organiz., tecn., producto)	Flexibilidad y Cambiabilidad	Impacto
P5. Detección de fallos		
P5.1. Clasificación del error	La clasificación del error puede cambiar durante el desarrollo. Es afectado por sus características funcionales, el modelo de interacción con el usuario, y el hardware.	Hay un moderado impacto en todos los componentes. Se afecta la forma de notificación del error
Los errores se clasifican de acuerdo a su tipo y severidad		

Tabla 1.1.- Ejemplo de plantilla de clasificación de factores influyentes en el diseño de una arquitectura

Una vez establecidos todos los factores que pueden influir en la arquitectura, se estudia su impacto para ver cuales son los factores predominantes (equivaldrían a las *directrices arquitectónicas* de ABD), y una vez determinados esos factores predominantes (denominados *Architecture Design Issue*), se generan unas plantillas que serán determinantes para diseñar la arquitectura (**Tabla 1.2**) del estilo de las plantillas de atributo de ABD incluidas en el Anexo II.

Architecture Design Issue: (p.ejm. Cambios en el Hardware de Propósito General y Específico de Dominio). Se anticipan cambios en el hardware de propósito general y específico del dominio. Reducir el esfuerzo y tiempo para adaptar el sistema a nuevo hardware.	
Factores Influyentes	T1.1. La velocidad del procesador cambia, incluso durante el desarrollo del proyecto. T2.1. El hardware de adquisición de imágenes cambia cada 3 años, etc.
Solución	Separar el software que interactúa directamente con el hardware. Las siguientes estrategias se deberían aplicar primero en la vista conceptual para introducir componentes que encapsulen el hardware
Estrategia	Encapsular hardware específico de dominio. Usar una abstracción para el sistema de adquisición de imágenes minimiza los efectos de los cambios en dicho hardware. Encapsular el hardware de propósito general. Así tendrá menor impacto en las aplicaciones, además, esta estrategia soporta la introducción de nuevas características de la aplicación sin requerir modificaciones del software que maneja el hardware.
Estrategias relacionadas	Estrategias como *"Comprar en lugar de construir"* y *"Reutilizar lo que existe en casa, hardware específico de dominio"*.

Tabla 1.2.- Ejemplo de plantilla de descripción de un factor predominante con estrategias asociadas

Anexo II

Plantillas de Atributo

Se consideran las siguientes plantillas:

Modificabilidad:

M1: Independencia del sistema operativo y de la plataforma.

M2: Adaptabilidad a diferentes despliegues.

M3: Independencia de la infraestructura de comunicaciones.

M4: Integración de servicios y utilidades.

M5: Adaptabilidad a diferentes interfaces de usuario.

M6: Adaptabilidad a diferentes entornos de operación.

M7: Adaptabilidad a diferentes misiones.

M8: Adaptabilidad a diferentes mecanismos (vehículos, brazos o herramientas).

M9: Adaptabilidad a diferentes configuraciones de mecanismos.

M10: Integración de componentes software comerciales.

M11: Integración de componentes hardware comerciales.

Rendimiento:

R1: Tareas Críticas.

R2: Tareas no críticas.

Seguridad:

S1: Supervisión del estado del sistema.

S2: Gestión de Comandos.

S3: Mecanismos de Parada Segura.

S4: Acceso a la Interfaz de Usuario.

Disponibilidad/Fiabilidad

D1: Disponibilidad de las Comunicaciones.

D2: Disponibilidad de las Interfaces de Usuario.

D3: Disponibilidad de los Mecanismos de Parada Segura.

Interoperabilidad

I1: Escenarios de la interoperabilidad.

M1: Independencia del sistema operativo y de la plataforma.

Descripción:
Los sistemas deben adaptarse a cambios de plataforma y sistemas operativo.

Escenarios Abstractos:
Estímulo: Cambio del sistema operativo en alguno de los nodos incluidos en el sistema.
Respuesta: El número de cambios debe ser limitado y no debe propagarse por el sistema.

Estrategias:
Separación de conceptos.
Encapsular hardware de propósito general.
Uso de estándares (p.e: POSIX)

Mecanismos:
Nivel de acceso a servicios del sistema operativo que proporcione una interfaz abstracta de acceso a los mismos.

Atributos relacionados:
Rendimiento ↓ : Aumenta el número de llamadas.
Flexibilidad en el despliegue físico ↑: No dependencia del S.O. de las plataformas.

Directrices y características de las que se deriva:
La implementación de los componentes de la arquitectura podrá ser software o hardware, pudiendo ser también componentes comerciales COTS. La implementación podrá realizarse sobre plataformas diversas, incluso distribuidas.

- Especialización de los sistemas: No todas las aplicaciones de teleoperación demandan los mismos servicios del sistema operativo.
- Caracter distribuido de los sistemas: Los procesos asignados a los diferentes nodos pueden demandar servicios diferentes (pe: algunos nodos necesitan S.O. en tiempo real y otros no).
- Mantenibilidad de los sistemas: No todas las partes del sistema evolucionan al mismo ritmo. Puede ser necesario cambiar el S.O., manteniendo el resto de la aplicación.

M2: Adaptabilidad a diferentes despliegues

Descripción:
Diferentes procesos, controladores y susbsistemas deben poder asignarse a distintos nodos, al menos en tiempo de compilación y carga.

Escenarios Abstractos:
Estímulo: Asignación de uno o varios procesos a diferentes nodos.
Estímulo: Asignación de diferentes subsistemas a diferentes nodos.
Respuesta: El sistema debe seguir ofreciendo el mismo o similar comportamiento.

Estrategias:
Separación de conceptos. Indirección.
Mecanismos de comunicación, preferiblemente asíncronos, basados en el paso de mensajes.
Criterios de estructuración del sistema en subsistemas (COMET, [Gomma00]).
Crierios de despliegue (COMET, [Gomma00]).
Uso de Middleware (RPC, DCOM, CORBA, DCE, Java RMI, etc.)

Mecanismos:
- Proxy/Stub.
- Broker
- Criterios de estructuración del sistema en subsistemas:
  - ✓ Agregación: Las entidades que formen parte del mismo agregado deben estar en el mismo subsistema.
  - ✓ Localización: Si dos entidades pueden ser potencialmente ubicadas en nodos diferentes deben asignarse a diferentes subsistemas.
  - ✓ Servicios:Clientes y servidores deben estar en diferentes subsistemas.
  - ✓ Control: Un controlador y las entidades con las que interacciona *directamente* deben estar en un mismo subsistema.
- Criterios de despliegue.
  - ✓ Proximidad a los dispositivos físicos: Proximidad de los componentes a los dispositivos físicos que controlan o de los que reciben datos
  - ✓ Autonomía: Incrementar autonomía de los nodos. Asignar a cada nodo responsabilidades concretas lo menos acopladas posible con las del resto.
  - ✓ Rendimiento: Asignar tareas más críticas a nodos específicos.
  - ✓ Interfaces de usuario y servidores pueden asignarse a diferentes nodos.

Atributos relacionados:
Rendimiento ↑ : Se favorece el paralelismo. Asignación de tareas críticas a nodos específicos.
Disponibilidad ↑ : En caso de que un nodo deje de funcionar, pueden asignarse a otro sus responsabilidades.

Directrices y características de las que se deriva:
Diferentes instanciaciones de la arquitectura deben poder compartir, reutilizar, los mismos componentes.
Se deben adoptar políticas que permitan una clara separación entre dichos componentes y sus patrones de interacción.
La implementación de los componentes de la arquitectura podrá ser software o hardware, pudiendo ser también componentes comerciales COTS. La implementación podrá realizarse sobre plataformas diversas, incluso distribuidas.

- Carácter distribuido de los sistemas. Habitualmente los sistemas constan de dos nodos, pero puede haber más o uno sólo.
- Posibilidad de incluir utilidades de simulación, visión artificial, sistemas de navegación, etc., que puedan actuar como proveedores de servicios (servidores) ubicados en diferentes nodos.
- Rendimiento. Posibilidad de ubicar las tareas más críticas o más exigentes en términos de computación en nodos específicos.

M3: Independencia de la infraestructura de comunicaciones.

Descripción:
Los sistemas deben adaptarse al cambio de los enlaces de comunicaciones entre los diferentes nodos físicos del sistema.

Escenarios Abstractos:
Estímulo: Cambio del enlace de comunicaciones entre dos nodos dados.
Estímulo: Cambio del protocolo de comunicaciones subyacente.
Estímulo: Adición/Eliminación de mensajes.
Estímulo: Modificación del formato de los mensajes.
Respuesta: El número de cambios debe ser limitado y no debe propagarse por el sistema.

Estrategias:
Separación de conceptos. Indirección.
Encapsular la comunicación con los dispositivos.
Uso de Middleware

Mecanismos:
Nivel de acceso a servicios de comunicaciones que proporcione una interfaz abstracta de acceso a los mismos.

Atributos relacionados:
Rendimiento ↓ : Aumenta el número de llamadas.
Uso de COTS ↑: Empleo de *middleware*.
Flexibilidad en el despliegue físico ↑

Directrices y características de las que se deriva:
Se deben adoptar políticas que permitan una clara separación entre dichos componentes y sus patrones de interacción.

La implementación de los componentes de la arquitectura podrá ser software o hardware, pudiendo ser también componentes comerciales COTS. La implementación podrá realizarse sobre plataformas diversas, incluso distribuidas.

- Especialización de los sistemas: No todas las aplicaciones de teleoperación demandan los mismos enlaces de comunicaiones.
- Carácter distribuido de los sistemas.
- Mantenibilidad de los sistemas: No todas las partes del sistema evolucionan al mismo ritmo. Puede ser necesario cambiar el enlace de comunicaciones manteniendo el resto de la aplicación.

Observaciones:
Puede considerarse un caso particular de la adaptabilidad a diferentes plataformas y sistemas operativos, sin embargo las comunicaciones tienen suficiente entidad en los sistemas de teleoperación como para ser consideradas por separado.

M4: Integración de servicios y utilidades.

Descripción:
Diferentes servicios o utilidades deben poder integrarse en el sistema de teleoperación.

Escenarios Abstractos:
Estímulo: Incorporación de una nueva utilidad o suministrador de servicios al sistema.
Estímulo: Eliminación de una utilidad o servicio existente en el sistema.
Estímulo: Modificación de los servicios provistos por una determinada utilidad del sistema.
Respuesta: El número de cambios debe ser limitado y no debe propagarse por el sistema.

Estrategias:
Separación de conceptos.
Usar estándares.
Desarrollar interfaces específicas con los componentes externos.
Encapsulación.
Módulos de desacoplo.

Mecanismos:
Cliente-servidor.
Mediador.
Editor-Subscriptor.
Almacén abstracto de datos.
Comunicaciones Asíncronas.

Atributos relacionados:
Rendimiento: ↓ Se incrementa el número de llamadas entre módulos (mediadores).
↑ Incorporación de servicios encargados de tareas específicas.
Adaptación a diferentes interfaces de usuario ↑: Posibilidad de incorporar herramientas software comerciales.
Utlilización de COTS: .↑ Posibilidad de incorporar herramientas software comerciales.
Desarrollo rápido de aplicaciones.
Usabilidad ↑: Incorporación de servicios útiles para el operador.
Disponibilidad ↑: Posibilidad de replicar servicios.

Directrices y características de las que se deriva:

La implementación de los componentes de la arquitectura podrá ser software o hardware, pudiendo ser también componentes comerciales COTS. La implementación podrá realizarse sobre plataformas diversas, incluso distribuidas.

- Especialización de los sistemas: No todas las aplicaciones de teleoperación demandan los mismos servicios o utilidades.
- Es habitual que se necesiten utilidades de simulación, representación gráfica, visión artificial, navegación, realimentación de pares y fuerzas, etc. Las aplicaciones más sencillas sólo demandan alguna de ellas (a veces ninguna), las más complejas pueden necesitar de muchas.
- Mantenibilidad de los sistemas: Posibilidad de añadir/eliminar utilidades o servicios según se vayan necesitando.

M5: Adaptabilidad a diferentes interfaces de usuario.

Descripción:
Los sistemas deben adaptarse a interfaces de usuario muy diferentes, que incluyen dispositivos específicos como joysticks, botoneras, reproducciones a escala de los dispositivos, utilidades de visión artificial, etc. y con muy distintas necesidades de representación gráfica y disposición de las ventanas.

Escenarios Abstractos:
Estímulo: Adición, eliminación, modificación de las ventanas o de los elementos que contienen.
Estímulo: Adición de nuevas interfaces (botoneras, joysticks, reproducciones de los dispositivos, etc) o sustitución de unas interfaces por otras (ratón por joystick, interfaz gráfica por textual, etc).
Respuesta: El resto del sistema no debe verse afectado.

Estrategias:
Separación de la interfaz de usuario del resto de la aplicación.
Separación de conceptos.
Uso de bibliotecas y utilidades comerciales.

Mecanismos:
Módulos de desacoplo.
Modelo-Vista-Controlador.
Editor-Subscriptor.

Atributos relacionados:
Rendimiento: ↓ Se incrementa el número de llamadas entre módulos (mediadores, niveles).
↑ La interfaz puede ejecutarse en nodos específicos, liberando al resto.
Utlilización de COTS: .↑ Posibilidad de incorporar utilidades software comerciales.
Usabilidad ↑: Incorporación de servicios útiles para el operador.
Disponibilidad ↑: Posibilidad de replicar interfaces.

Directrices y características de las que se deriva:
La implementación de los componentes de la arquitectura podrá ser software o hardware, pudiendo ser también componentes comerciales COTS. La implementación podrá realizarse sobre plataformas diversas, incluso distribuidas.

Debe poder gestionar distintos modos de control, distintos interfaces de teleoperación y otros sistemas externos que interaccionan con el sistema, incluso de forma simultánea.

- Especialización de los sistemas: No todas las aplicaciones de teleoperación demandan las mismas interfaces de usuario.
- Es habitual que se necesiten utilidades de simulación, representación gráfica, visión artificial, navegación, realimentación de pares y fuerzas, etc. Las aplicaciones más sencillas sólo demandan alguna de ellas (a veces ninguna), las más complejas pueden necesitar de muchas.

Observaciones:
Puede considerarse un caso particular de la adaptación a nuevos servicios. Está estrechamente relacionada con la posibilidad de utilizar herramientas y componentes software comerciales.

M6: Adaptabilidad a diferentes entornos de operación.

Descripción:
Los sistemas deben adaptarse para trabajar en diferentes entornos operativos.

Escenarios Abstractos:
Estímulo: Cambio en la geometría del entorno (entornos estructurados).
Estímulo: Cambio de entorno estructurado a no estructurado.
Estímulo: Cambio de las condiciones ambientales del entorno.
Respuesta: El número de cambios debe ser limitado y no debe propagarse por el sistema.

Estrategias:
Separación de conceptos.
Modelado del entorno y de los dispositivos.
Incorporación de utilidades de visión artificial.

Atributos relacionados:
Usabilidad ↑: Incorporación de servicios útiles para el operador.
Uso de COTS ↑: Incorporación de servicios útiles para el operador.
Adaptabilidad a diferentes misiones y mecanismos.

Directrices y características de las que se deriva:
Las operaciones a realizar por el robot se centrarán sobre todo en operaciones de mantenimiento, reparación e inspección no repetitivas, a realizar en entornos hostiles, cambiantes, remotos y/o inaccesibles para un operador humano, lo que imposibilita la actuación totalmente autónoma del robot.

Se busca el desarrollo rápido y de bajo coste del mayor número posible de sistemas dentro del dominio de la arquitectura.

- Especialización de los sistemas: Cada sistema de teleoperación trabaja en un entorno determinado.

Observaciones:
Puesto que la adaptación a diferentes entornos se basa (ver estrategias) en el uso de ciertos servicios, este atributo puede verse como un caso particular de la incorporación de nuevos servicios, y como aquel está relacionado con la posibilidad de utilizar herramientas y componentes software comerciales.

Es necesario considerar este atributo desde dos puntos de vista:

- Posibilidad de construir aplicaciones que operan en diferentes entornos.
- Posibilidad de adaptar un sistema dado para que pueda trabajar en diferentes entornos.

En el segundo caso la posibilidad de adaptar al sistema para trabajar en diferentes entornos está estrechamente relacionada con la posibilidad de adaptarlo a diferentes mecanismos y a diferentes misiones. Cada mecanismo tiene unas propiedades (mecánicas, eléctricas, estructurales, etc.) que le permiten realizar ciertas actividades en ciertos entornos.

Si el cambio entre entornos implica a dos entornos estructurados normalmente es suficiente con disponer de utilidades de modelado y representación gráfica. Si el entorno cambia, basta con modificar el modelo. En entornos no estructurados la utilidad de estas herramientas es limitada y puede ser necesaria la incorporación de un sistema de visión artificial.

M7: Adaptabilidad a diferentes misiones.

Descripción:
Posibilidad de adaptar al sistema para la realización de nuevas misiones.

Escenarios Abstractos:
Estímulo: Modificación misión existente:
Adición/Eliminación de pasos de la misión.
Modificación del orden de los pasos de la misión.
Modificación de la *máquina de estados* de la misión.
Estímulo: Adición de nuevas misiones.
Estímulo: Eliminación de misiones.
Estímulo: Construcción de nuevas misiones a partir de las existentes.
Respuesta: Las modificaciones deben ser limitadas y no propagarse por el sistema.

Estrategias:
Separación de conceptos.
Separa inteligencia de la funcionalidad.
Arquitecturas por capas

Mecanismos:
Asignar la ejecución de las misiones a componentes software específicos (controladores de misión).
Encapsular los conceptos propios de cada misión en componentes separados.

Atributos relacionados:
Adaptabilidad a diferentes entornos y mecanismos.
Incorporación de servicios.
Comunicaciones (sincronismo entre mecanismos).

Directrices y características de las que se deriva:
Las operaciones a realizar por el robot se centrarán sobre todo en operaciones de mantenimiento, reparación e inspección no repetitivas, a realizar en entornos hostiles, cambiantes, remotos y/o inaccesibles para un operador humano, lo que imposibilita la actuación totalmente autónoma del robot..

Se busca el desarrollo rápido y de bajo coste del mayor número posible de sistemas dentro del dominio de la arquitectura.

Diferentes instanciaciones de la arquitectura deben poder compartir, reutilizar, los mismos componentes.

Debe ser posible que los sistemas adopten ciertas acciones autónomas sencillas (principalmente reactivas) que complementen las acciones teleoperadas o incluso misiones pre-programadas (deliberativas).

- Especialización de los sistemas: Cada sistema debe realizar un conjunto de misiones, en general muy específicas.

Observaciones:
Una misión es la ejecución por los mecanismos de una serie de actividades, realizadas en secuencia o según la lógica impuesta por una máquina de estados. En ocasiones una misión compleja no es más que la composición de otras más simples. La adaptabilidad a nuevas misiones no depende de las características particulares de cada misión, su funcionalidad, sino de la forma de organizar dicha funcionalidad. Es necesario separar los aspectos ligados a la misión del resto.

Puesto que las misiones se realizan en un determinado entorno y utilizando determinados mecanismos, la adaptabilidad a nuevas misiones está ligada a la adaptabilidad a diferentes entornos, mecanismos y configuraciones (combinaciones vehículo-brazo-herramienta).

En ocasiones la misión sólo implica a un mecanismo (realización de un proceso de herramienta o ejecución de un movimiento o una secuencia de movimientos). En otras implica a varios. Para que varios mecanismos colaboren en una misión es necesario que sincronicen sus actividades e intercambien mensajes. Esto revela una nueva dimensión de la adaptabilidad a nuevas misiones: la necesidad de establecer medios de comunicación flexibles entre los controladores de los mecanismos.

Puesto que la realización de ciertas misiones necesita de ciertos servicios, la adaptabilidad a nuevas misiones está relacionada con la incorporación de nuevos servicios, y a través de ésta con la posibilidad de utilizar herramientas y componentes software comerciales.

Aún más: A menudo las misiones deben seguir un procedimiento más o menos estricto que debe ser seguido por el operador. La aplicación debe guiar la ejecución de las misiones, razón por la cual la adaptabilidad a nuevas misiones está relacionada con la adaptabilidad a nuevas interfaces y a la usabilidad.

M8: Adaptabilidad a diferentes mecanismos (vehículos, brazos o herramientas).

Descripción:
Adaptación de los sistemas a diferentes mecanismos (brazos, vehículos y herramientas).

Escenarios Abstractos:
Estímulo: Modificación del mecanismo:
- Modificación de la estructura.
- Modificación de la cinemática.
- Modificación de los dispositivos asociados.
  - Modificación/Adición/Eliminación accionadores y sensores
  - Modificación/Adición/Eliminación dispositivos E/S
- Modificación del control
  - Modicación/Adición/Eliminación de comandos o funcionalidad.
  - Modificación máquina de estados.
  - Modificación/Adición/Eliminación de mecanismos de sincronismo (eventos, mensajes…)

Adición de un nuevo mecanismo.
Eliminación de un mecanismo.
Sustitución de un mecanismo por otro.

Estrategias:
Separación de conceptos.
Alta cohesión/bajo acoplamiento
Encapsulación, tipos abstractos de datos.
Herencia y composición.

Mecanismos:
Separación de aspectos estructurales, cinemáticos y de control.
Modelado incremental de la funcionalidad (implementación de interfaces abstractas).
Definición de mecanismos (brazos, vehículos, herramientas) abstractos.
Uso de herencia e interfaces (clases abstractas)
Uso de plantillas y genéricos.
Definición de una interfaz abstracta de acceso a los mecanismos.

Atributos relacionados:
Adaptabilidad a nuevas misiones
Adaptabilidad a nuevos entornos

Directrices y características de las que se deriva:
Las operaciones a realizar por el robot se centrarán sobre todo en operaciones de mantenimiento, reparación e inspección no repetitivas, a realizar en entornos hostiles, cambiantes, remotos y/o inaccesibles para un operador humano, lo que imposibilita la actuación totalmente autónoma del robot..

Se busca el desarrollo rápido y de bajo coste del mayor número posible de sistemas dentro del dominio de la arquitectura.

Diferentes instanciaciones de la arquitectura deben poder compartir, reutilizar, los mismos componentes.

La implementación de los componentes de la arquitectura podrá ser software o hardware, pudiendo ser también componentes comerciales COTS. La implementación podrá realizarse sobre plataformas diversas, incluso distribuidas.

Adaptabilidad a diferentes mecanismos. Debe ser fácil añadir y quitar componentes y capacidades del sistema.

- Especialización de los sistemas: Cada sistema debe realizar un conjunto de misiones, en general muy específicas.
- Gran dispersión de características de los mecanismos teleoperados. Como consecuencia de la especialización, la complejidad y características de los vehículos, brazos y herramientas utilizados varían enormemente de unas aplicaciones a otras.
- Mantenibilidad: Los mecanismos están sujetos a continuas mejoras. En ocasiones se estima oportuno sustituir un mecanismo por otro, pero manteniendo el resto de las características del sistema (cambio del brazo o la herramienta por otros más modernos, modificaciones en los mismos, etc.)

M9: Adaptabilidad a diferentes configuraciones de mecanismos.

Descripción:
Adaptación a diferentes combinaciones de robots, vehículos y herramientas.

Escenarios Abstractos:
Estímulo: Cambio/Adición/Eliminación de vehículo.
Estímulo: Cambio/Adición/Eliminación de brazo.
Estímulo: Cambio/Adición/Eliminación de herramienta.
Respuesta: Las modificaciones deben ser limitadas y no propagarse por el sistema.

Estrategias:
Separación de conceptos.
Mecanismos de sincronización.

Mecanismos:
Controlador global (por encima de los controladores de los dispositivos)
Interfaces abstractas para el sincronismo (definición de una interfaz abstracta de acceso a los mecanismos).

Atributos relacionados:
Extensibilidad.
Adaptabilidad a nuevos mecanismos.
Adaptabilidad a nuevas misiones.

Directrices y características de las que se deriva:
Las operaciones a realizar por el robot se centrarán sobre todo en operaciones de mantenimiento, reparación e inspección no repetitivas, a realizar en entornos hostiles, cambiantes, remotos y/o inaccesibles para un operador humano, lo que imposibilita la actuación totalmente autónoma del robot..

Se busca el desarrollo rápido y de bajo coste del mayor número posible de sistemas dentro del dominio de la arquitectura.

Diferentes instanciaciones de la arquitectura deben poder compartir, reutilizar, los mismos componentes.

La implementación de los componentes de la arquitectura podrá ser software o hardware, pudiendo ser también componentes comerciales COTS. La implementación podrá realizarse sobre plataformas diversas, incluso distribuidas.

- Distintas configuraciones de mecanismos. Cada sistema debe realizar un conjunto de misiones, en general muy específicas, que requieren el uso de diferentes combinaciones de mecanismos.

M10: Integración de componentes software comerciales.

Descripción:
Debe existir la posibilidad de usar componentes comerciales que ofrezcan servicios interesantes para el sistema, bien directamente (herramientas software), bien indirectamente (a través de bibliotecas software).

Escenarios Abstractos:
Estímulo: Uso de COTS para la implementación de susbsistemas.
Respuesta: Los cambios deben ser limitados y no propagarse por el sistema.

Estrategias:
Separación de conceptos.
Encapsulación.
Módulos de desacoplo.

Mecanismos:
Cliente-servidor.
Adaptadores (Wrappers)
Mediador.

Atributos relacionados:
Rendimiento: ↓ Se incrementa el número de llamadas entre módulos (mediadores).
↑ Incorporación de servicios encargados de tareas específicas.
Adaptación a diferentes interfaces de usuario ↑: Posibilidad de incorporar herramientas software comerciales.
Usabilidad ↑: Incorporación de servicios útiles para el operador.
Disponibilidad ↑: Posibilidad de replicar servicios.
Rapidez de desarrollo

Directrices y características de las que se deriva:
Se busca el desarrollo rápido y de bajo coste del mayor número posible de sistemas dentro del dominio de la arquitectura.

La implementación de los componentes de la arquitectura podrá ser software o hardware, pudiendo ser también componentes comerciales COTS. La implementación podrá realizarse sobre plataformas diversas, incluso distribuidas.

Observaciones:
Puede considerarse un caso particular de la adaptación a nuevos servicios, sin embargo hay algunas diferencias:

- El uso de los COTS no se limita a la incorporación o implementación de servicios.
- Los COTS tienen su problemática propia:
  - ✓ Compatibilidad entre diferentes versiones.
  - ✓ Compatibilidad entre COTS.
  - ✓ Suposiciones arquitecturales de los COTS.
  - ✓ Sustitución de unos COTS por otros.

Este atributo está expresado con una gran generalidad, que es a estas alturas inevitable. Hasta que no se plantee el diseño de la arquitectura no se podrá determinar cuáles son los componentes susceptibles de ser implementados por COTS. En general, los mismos mecanismos y estrategias que favorecen el resto de las *adaptabilidades* favorecen también el uso de COTS.

M11: Integración de componentes hardware comerciales.

Descripción:
Debe existir la posibilidad de usar componentes hardware comerciales que permitan reemplazar algunos de los componentes sofware por estos componentes hardware.

Escenarios Abstractos:
Estímulo: Uso de COTS para la implementación de susbsistemas.
Estímulo: Uso de COTS para implementar algunos componentes de la arquitectura.
Respuesta: Los cambios deben ser limitados y no propagarse por el sistema.

Estrategias:
Separación de conceptos.
Encapsulación del hardware específico del dominio.
Módulos de desacoplo.

Mecanismos:
Cliente-servidor.
Adaptadores.
Componentes de abstracción del hardware.
Mediador.

Atributos relacionados:
Rendimiento: ↓ Se incrementa el número de llamadas entre módulos (mediadores).
↑ Incorporación de servicios encargados de tareas específicas.
↑ Paralelismo real que disminuye la sobrecarga del sistema.

Disponibilidad ↑: Posibilidad de rápida sustitución de componentes.
Rapidez de desarrollo

Directrices y características de las que se deriva:
Se busca el desarrollo rápido y de bajo coste del mayor número posible de sistemas dentro del dominio de la arquitectura.

La implementación de los componentes de la arquitectura podrá ser software o hardware, pudiendo ser también componentes comerciales COTS. La implementación podrá realizarse sobre plataformas diversas, incluso distribuidas.

Observaciones:
Este atributo está expresado con una gran generalidad, que es a estas alturas inevitable. Hasta que no se plantee el diseño de la arquitectura no se podrá determinar cuáles son los componentes susceptibles de ser implementados por COTS. En general, se puede prever que los componentes de control presentes en una implementación serán susceptibles de ser cambiados por controladores COTS.

R1: Tareas críticas

Suelen estar relacionadas con procesos de control de los mecanismos (vehículo-brazo-herramienta) cercanos al hardware. Por ejemplo:

- ✓ Control del bucle de realimentación de los servos.
- ✓ Lectura de sensores y accionamiento de actuadores.
- ✓ Procesamiento de alarmas relacionadas con el funcionamiento del hardware.
- ✓ Procesos o servicios relacionados con la parada segura del sistema.

En este tipo de tareas, los eventos pueden llegar al sistema de forma períodica (muestreo) o esporádica (alarmas e interrupciones), deben procesarse dentro de unos plazos determinados, sin excepciones, y habitualmente en un orden determinado. Las comunicaciones entre nodos entran dentro de esta categoría si las tareas que se ocupan de estos aspectos están distribuidas por el sistema. También entran dentro de ella los procesos servidores si los servicios que suministran son necesarios para llevar a cabo estas tareas.

Comportamiento temporal de tareas críticas

Descripción:

Las tareas críticas deben realizarse dentro de un plazo determinado desde su activación por el evento correspondiente. El comportamiento del sistema debe ser predecible.

Patrón de llegada de eventos: periódico, esporádico.

Respuesta: Dentro de un plazo.

Escenarios Abstractos:

Estímulos:

Variación del período de activación de las tareas.
Variación del tiempo de procesamiento de las tareas.
Variación del número de tareas.
Variación de los plazos de las tareas.
Ejecución de tareas en distintos nodos o en un solo nodo (carga de las comunicaciones y características de los enlaces de comunicaciones).

Respuesta: Procesamiento de las tareas dentro de sus plazos.

Estrategias:

- Minimización del intercambio de datos (las comunicaciones, ya sea a través de enlaces físicos, de paso de mensajes o de llamadas a procedimiento son el principal factor de carga del sistema).
- Buses y protocolos de comunicaciones determinísticos.
- Minimización del número de transformaciones de datos y del número de cálculos.
- Asignación de recursos: Estrategias para repartir los recursos disponibles.
- Usar RMA para predecir prestaciones.
- Arbitraje y uso de recursos: Estrategias para resolver conflictos de acceso a recursos (prioridades, desalojos,etc.) y optimizar el uso de los mismos.

Mecanismos:

- Asignación de recursos:
  - ✓ Balance de carga: Repartir la carga entre los procesadores.
  - ✓ *Bin packing*: Medida de la capacidad disponible y asignación en función de las necesidades de las tareas.
- Arbitraje y uso de recursos:
  - ✓ Planificación con desalojo (*Preemtive*)
    - Planificación por prioridades dinámicas (*earliest deadline first scheduling*).
    - Planificación por prioridades fijas (*rate monotonic analysis, deadline monotonic*).
    - Estrategias de servicio aperiódico (planificación de procesos aperiódicos en un contexto periódico, *slack stealing*).
  - ✓ Exclusión mutua, sincronización.
    - Zonas críticas, semáforos, monitores, *rendezvous*, etc.
  - ✓ Otros.
    - Caché, tablas pre-calculadas, hardware específico, etc.

R1: Tareas críticas (Continuación)

Atributos relacionados:

Modificabilidad:

- *Portabilidad (plataformas, sistemas operativos, enlaces de comunicaciones).*

  Los mecanismos que facilitan la portabilidad (nivel de acceso) pueden suponer un empeoramiento del rendimiento a través de las llamadas entre módulos.

  No todos los sistemas operativos ofrecen los servicios que pueden ser necesarios (procesos ligeros, planificación con prioridades, predictabilidad, etc).

  No todos los enlaces de comunicaciones tienen el ancho de banda adecuado, ni son determinísticos.

  Por otro lado, la posibilidad de utilizar buses de alto rendimiento y sistemas operativos de tiempo real favorecen el rendimiento.

- *Integración de servicios:*

  Si las tareas críticas necesitan de ciertos servicios, el acceso a los mismos y su procesamiento se convierten asimismo en tareas críticas. Los mecanismos que facilitan la integración de servicios (mediadores, módulos de desacoplo, etc.) pueden suponer un empeoramiento del rendimiento a través de las llamadas entre módulos.

- *Uso de COTS.*
  Deben proporcionar el rendimiento adecuado y no hacer suposiciones arquitecturales que afecten o entren en conflicto con las estrategias seleccionadas. La posibilidad de integrar hardware específico puede mejorar mucho el rendimiento.

- *Flexibilidad en el despliegue:*
  Puede mejorar el rendimiento, pues facilita la asignación de recursos a diferentes nodos y favorece la definición de nodos *autosuficientes*.(Minimización de las comunicaciones).

- *Seguridad:*
  La seguridad depende del rendimiento. Si las tareas críticas no cumplen sus plazos el funcionamiento puede degradarse hasta límites inaceptables.

- *Usabilidad:*
  El rendimiento favorece la usabilidad. La información que se ofrece al operador debe reflejar el estado actual del sistema y los comandos deben activarse *inmediatamente*, dentro de los límites de la percepción humana.

Directrices y características de las que se deriva:

La presencia de enlaces de comunicaciones, la posibilidad de distribución y de diferentes particionados hardware/software debe tenerse en cuenta.

Los requisitos de tiempo-real críticos son condicionantes para el funcionamiento correcto del sistema, aunque en los sistemas también se encuentran requisitos que no son de tiempo-real.

- Requisitos de tiempo real.

Observaciones:

Entre los mecanismos que se han enumerado para gestionar el comportamiento temporal de las tareas críticas no se han incluido ni el ejecutivo cíclico ni algoritmos de colas. El ejecutivo cíclico, aunque puede ser la solución optima para *un* sistema, no es capaz de adaptarse a las variantes de la familia. La teoría de colas modela muy bien el comportamiento medio del sistema (throughput, caso peor, factor de utilización, etc.), pero los algoritmos de planificación basados en prioridades son más adecuados para gestionar tareas con plazos fijos.

R1: Tareas no críticas

Tareas menos críticas

Son en general procesos de control de alto nivel no directamente relacionados con el funcionamiento seguro del sistema. La planificación y secuenciación de las misiones, la captura de datos que pueden procesarse en diferido y dicho procesamiento, ciertos procesos de diagnosis, la actualización de la interfaz de usuario, el suministro de ciertos servicios, etc. entran habitualmente dentro de esta categoría.

Descripción:

Las tareas menos críticas deben realizarse a una tasa (throughput) que proporcione un funcionamiento del sistema que pueda considerarse aceptable. El comportamiento del sistema debe ser predecible.

Patrón de llegada de eventos: periódico, esporádico.

Respuesta: Throughput (tareas realizadas o eventos servidos por unidad de tiempo).

Escenarios Abstractos:

Estímulos:

Variación del período de activación de las tareas.
Variación del tiempo de procesamiento de las tareas.
Variación del número de tareas.
Variación del througput requerido.
Ejecución de tareas en distintos nodos o en un solo nodo (carga de las comunicaciones y características de los enlaces de comunicaciones).

Respuesta:

Throughput.
Pueden definirse plazos y número de pérdidas de plazo (faltas) admisible, además de otros parámetros (jitter o máxima variación admisible en el intervalo de tiempo de respuesta entre el procesamiento de dos eventos, respuesta media y caso peor, etc)

Estrategias:

- Minimización del intercambio de datos (las comunicaciones, ya sea a través de enlaces físicos, de paso de mensajes o de llamadas a procedimiento son el principal factor de carga del sistema).
- Buses y protocolos de comunicaciones con suficiente ancho de banda.
- Minimización del número de transformaciones de datos y del número de cálculos.
- Asignación de recursos: Estrategias para repartir los recursos disponibles.
- Arbitraje y uso de recursos: Estrategias para resolver conflictos de acceso a recursos (prioridades, desalojos,etc.) y optimizar el uso de los mismos.

Mecanismos:

- Asignación de recursos:
  - ✓ Balance de carga: Repartir la carga entre los procesadores.
- Arbitraje y uso de recursos:
  - ✓ Planificación con desalojo (*Preemtive*)
    - Planificación por prioridades fijas (*rate monotonic analysis, deadline monotonic*).
    - Modelos de colas.
  - ✓ Exclusión mutua, sincronización.
    - Zonas críticas, semáforos, monitores, *rendezvous*, etc.
  - ✓ Otros.
    - Caché.

Atributos relacionados:

Pueden hacerse los mismos comentarios que en el caso de las tareas críticas.

Directrices y características de las que se deriva:

La presencia de enlaces de comunicaciones, la posibilidad de distribución y de diferentes particionados hardware/software debe tenerse en cuenta.

Los requisitos de tiempo-real críticos son condicionantes para el funcionamiento correcto del sistema, aunque en los sistemas también se encuentran requisitos que no son de tiempo-real.

Observaciones:

La actualización de la interfaz de usuario y la respuesta ofrecida por los servicios que se integren en el sistema caen habitualmente, aunque no siempre, dentro de esta categoría. Las comunicaciones son más o menos críticas dependiendo de la distribución de las responsabilidades entre nodos y del grado de autonomía de los mismos.

S1: Supervisión del estado del sistema.

Descripción:
Debe supervisarse el correcto comportamiento de los mecanismos y de todos aquellos subsistemas, tanto hardware (accionadores, motores, tarjetas de control, tarjetas E/S, enlaces de comunicaciones, etc) como software que tengan alguna influencia en la seguridad del sistema.

Escenarios Abstractos:
Estímulos:
a) Se produce una sobrecarga de motores o accionadores.
b) Se corta el suministro de energía.
c) Se produce un mal funcionamiento en los bucles de control de los mecanismos.
d) Se produce un mal funcionamiento de algún módulo hardware.
e) Se produce un mal funcionamiento de algún módulo software.
f) Se cortan las comunicaciones entre nodos.
g) Se activa una determinada alarma.

Respuestas:
a) Se detecta la sobrecarga y se desactivan los accionadores.
b) Se detecta el corte de alimentación y se activa el mecanismo de parada segura.
c) Se detectan las anomalías de los bucles de control y se detiene el proceso en curso (si son los servos, se detiene el movimiento, si es el control del proceso de herramienta se detiene dicho proceso) hasta que el control pueda reestablecerse.
d) Se detectan los fallos de los módulos y si es posible se prescinde de ellos o se sustituyen por otros.
e) Se detectan los fallos de los módulos y si es posible se prescinde de ellos o se sustituyen por otros.
f) Se detectan los fallos en las comunicaciones y si es posible se reestablecen.
g) Las alarmas deben procesarse en un plazo determinado que asegure que las causas que las producen son procesada antes de que se originen situaciones peligrosas para la integridad de personas y equipos.

En todos los casos se debe advertir al operador y si se estima que los errores son lo suficientemente graves se activa automáticamente la parada segura del sistema.

Estrategias y Mecanismos:
- Separación de conceptos.
- Bajo acoplamiento (evita la propagación de fallos), Alta cohesión (Facilita el tratamiento local y precoz de los errores).
- Detección de fallos.
- Tolerancia a fallos: Duplicación de servicios críticos (redundancia). Recuperación de errores, etc.
- Servicios de diagnosis y componentes diseñados para proporcionar información a dicho servicio.
- Auto-test de los componentes.
- Tratamiento de excepciones.
- Tareas específicas que supervisen y procesen periódicamente el estado del sistema.
- Tareas que se activen de forma automática ante la ocurrencia de ciertos eventos (pe: ciertas alarmas).
- Simulación on-line de los comandos.

Atributos relacionados:
- *Rendimiento:*
  Los requisitos temporales deben ser garantizados para asegurar que la información disponible sobre el sistema refleja fielmente su estado.

  Las tareas encargadas de la seguridad suponen una sobrecarga del sistema.

  A través del rendimiento se ven afectados otros atributos.

- *Modificabilidad:*
  Los componentes deben pensarse desde un principio para ser seguros. Los componentes críticos deben proporcionar interfaces a través de las cuales un servicio de diagnóstico o una tarea de supervisión pueda detectar fallos en su funcionamiento.

  Los principios de bajo acoplamiento y alta cohesión que facilitan el tratamiento de errores favorecen la modificabilidad en todos sus aspectos.

- *Disponibilidad:*
  Los mecanismos y estrategias arquitecturales que permiten la obtención de la disponibilidad son precisamente los mismos que pueden garantizar este aspecto de la seguridad.

S1: Supervisión del estado del sistema (Continuación).

Directrices y características de las que se deriva:
Deben adoptarse mecanismos de tolerancia a fallos, tratamiento de excepciones, errores, mecanismos de diagnóstico para prevención de fallos, etc.

Observaciones:
A menudo se olvida que las fuentes de error no sólo tienen su origen en los mecanismos y en el hardware asociado, sino también en la propia aplicación. Plazos que se pierden, subsistemas que se bloquean, excepciones que no se tratan adecuadamente, etc., pueden ser la causa de errores que pongan en peligro la seguridad de los operarios y comprometan la integridad de los equipos.

Las medidas software deben ser complementadas con otras de diferente índole. Por ejemplo, de poco vale implementar mecanismos software de parada segura si los dispositivos mecánicos no incorporan frenos o sistemas de enclavamiento que puedan entrar en funcionamiento cuando se corta la alimentación de los accionadores.

S2: Gestión de Comandos.

Descripción:
Gestión de los comandos para:

- Impedir que se ejecuten comandos inseguros.
- Asegurar que los comandos se ejecutan según el plan previsto.

Escenarios Abstractos:
Estímulos:
a) El sistema entra en un estado en el que no es seguro ejecutar ciertos comandos.
b) La ejecución del comando es compatible con el estado, pero no puede determinarse si su ejecución es segura.
c) El comando no se ejecuta según el plan previsto.

Respuestas:
a) Inhabilitación de comandos no compatibles con el estado del sistema.
b) Comprobar la viabilidad de un comando antes de su ejecución (un comando es viable si su ejecución deja al sistema en un estado conocido y seguro).
c) Monitorizar la ejecución de los comandos comprobando que se realizan según el plan previsto (no se observan discrepancias intolerables entre los estados real y esperado, no vencen timeouts, etc.).

Estrategias y Mecanismos:
- Gestión de estados y modelado de comandos:
  - ✓ Cada estado tiene asociados una serie de comandos que pueden ejecutarse en el mismo.
  - ✓ Cada comando puede ejecutarse cuando el sistema se encuentra en ciertos estados.
- Separación de conceptos y encapsulación:
  - ✓ Encapsular con cada estado los comandos que se pueden ejecutar en el mismo.
  - ✓ Encapsular con cada comando los estados en los que se puede ejecutar.
- Controladores para los comandos.
- Controladores para los mecanismos.
- Simulación previa de los comandos.

Atributos relacionados:
- *Rendimiento:*
  Los requisitos temporales deben ser garantizados para asegurar que la información disponible sobre el sistema refleja fielmente su estado. Los comandos deben ser monitorizados en tiempo real.

  A través del rendimiento se ven afectados otros atributos.
- *Integración de servicios y uso de COTS:*

  La simulación de comandos puede requerir del uso de herramientas muy potentes.
- *Modificabilidad*:
  Es posible que las medidas de seguridad se hagan más estrictas y que comandos que antes estaban habilitados en un estado tengan que deshabilitarse, o que las tolerancias entre estados real y esperado se hagan más pequeñas. También podría ocurrir lo contrario.

Directrices y características de las que se deriva:
La presencia de enlaces de comunicaciones, la posibilidad de distribución y de diferentes particionados hardware/software debe tenerse en cuenta.

Deben adoptarse mecanismos de tolerancia a fallos, tratamiento de excepciones, errores, mecanismos de diagnóstico para prevención de fallos, etc.

Observaciones:
Puede considerarse un caso particular de la monitorización del estado, sin embargo la ejecución de comandos introduce ciertas condiciones particulares. Una parte del estado del sistema debe supervisarse siempre, haya o no haya un comando en curso, otras partes son relevantes precisamente por el comando que se está ejecutando. Es necesario separar los aspectos específicos de cada comando.

La gestión de los comandos tiene que ver ante todo con la funcionalidad del sistema, que no es un atributo específicamente arquitectural. Sin embargo, el cumplimiento de este requisito representa una sobrecarga de actividades que puede afectar bastante al cumplimiento de otros atributos *más arquitectónicos*:

El estudio de la viabilidad de los comandos puede requerir del uso de ciertos servicios. La relación comandos/estados debe tenerse en cuenta a la hora de diseñar los módulos de control y su comportamiento dinámico (el estado del sistema puede cambiar durante la ejecución del comando, por causas achacables al mismo o no). La supervisión de la ejecución tiene importantes consecuencias en el rendimiento y a través de él en otros atributos del sistema.

La simulación on-line de los comandos puede suponer una fuerte sobrecarga del sistema. Un esquema alternativo, pero bastante útil, es aprovechar los datos de una simulación off-line para generar los estados esperados, que se van comparando periódicamente con los reales.

S3: Mecanismos de Parada Segura.

Descripción:

Deben disponerse mecanismos de parada de emergencia que se activen por orden del operador o cuando se detecten situaciones de riesgo o errores graves de funcionamiento.

Todas las interfaces del sistema deben disponer de un dispositivo de seguridad que permita detener inmediatamente a los mecanismos.

Escenarios Abstractos:

Estímulos:

a) El operador acciona el *botón* de parada.
b) Se detectan errores graves en el sistema.

Respuestas:

- Los mecanismos se detienen inmediatamente y entran en un estado conocido y seguro.

Estrategias y Mecanismos:

- Servicios de diagnosis.
- Tratamiento de excepciones.
- Tareas específicas que supervisen y procesen periódicamente el estado del sistema.
- Tareas que se activen de forma automática ante la ocurrencia de ciertos eventos (pe: ciertas alarmas).
- Simulación on-line de los comandos.
- Botón de parada segura.

Atributos relacionados:

Pueden hacerse los mismos comentarios que en los dos casos anteriores y además:

- *Usabilidad:*
  Los dispositivos de parada segura deben ser fácilmente identificables y accesibles por el operador.
  Su funcionamiento no debe depender del funcionamiento de la aplicación. Un icono en la interfaz de usuario no es suficiente. Deben proporcionarse dispositivos hardware que accedan directamente a los dispositivos.

  En ocasiones (actividades de prueba, programación y mantenimiento) puede ser necesario que el usuario (operador, programador o mantenedor) deba realizar una acción específica (p.e. mantener pulsado un botón) para accionar los dispositivos. En estas situaciones la parada segura está accionada por defecto.

Directrices y características de las que se deriva:

Las operaciones sólo deben ejecutarse si existe una confianza razonable en que el resultado de las mismas no va a poner en peligro la integridad de bienes y equipos.

Deben adoptarse mecanismos de tolerancia a fallos, tratamiento de excepciones, errores, mecanismos de diagnóstico para prevención de fallos, etc.

S4: Acceso a la Interfaz de Usuario.

Descripción:
El acceso al software de control y arranque debe estar limitado mediante códigos de seguridad.

Escenarios Abstractos:
Estímulos: Un usuario no autorizado intenta acceder al sistema.
Respuestas: Se piden login y password.

Estrategias y Mecanismos:
- Contraseña.

Directrices y características de las que se deriva:
Las operaciones sólo deben ejecutarse si existe una confianza razonable en que el resultado de las mismas no va a poner en peligro la integridad de bienes y equipos.

Observaciones
Puede ser aconsejable registrar las acciones realizadas por el operador para descubrir las causas de posibles errores.

D1: Disponibilidad de las Comunicaciones.

Descripción:
Los fallos en los enlaces de comunicaciones entre nodos deben al menos detectarse y si es posible corregirse.

Escenarios Abstractos:
Estímulos:
a) Un mensaje es enviado/recibido incorrectamente.
b) Un nodo no responde o no envía información a la tasa requerida.
c) No es posible reestablecer las comunicaciones con un nodo.
d) Excesivo tráfico. Pérdida de rendimiento.

Respuestas:
a) Al menos el nodo emisor detecta el fallo. Reintenta. Si tras un número determinado de reintentos persiste el fallo se establece un nuevo enlace de comunicaciones.
b) Los nodos receptores detectan el fallo. Se intenta restablecer las comunicaciones. Si tras un número determinado de reintentos el fallo persiste, existen las siguientes alternativas (los sistemas deben implementar al menos una de ellas):
   - ✓ Asignar las responsabilidades del nodo silencioso a otro nodo.
   - ✓ Si es posible trabajar sin los servicios que proporciona el nodo, se prescinde del mismo (modo de operación degradado)
   - ✓ Si es imposible trabajar sin los servicios del nodo, se activa parada segura.
c) Igual que el anterior.
d) Se detecta la sobrecarga. Existen las siguientes alternativas (los sistemas deben implementar al menos una de ellas):
   - ✓ Reconfigurar el despliegue físico para minimizar las comunicaciones.
   - ✓ Si las condiciones de seguridad lo permiten se disminuye el tráfico (modo de operación degradado). Si no, se activa parada segura.

En todos los casos el operador debe ser informado y en la medida de lo posible (dependiendo de la gravedad del fallo) se le debe dar la opción de continuar o detener la operación.

La reconfiguración del despliegue puede realizarse en tiempo de carga (parada del sistema y rearranque con nueva configuración).

Estrategias y Mecanismos:
- Tolerancia a Fallos(Medidas dinámicas, en tiempo de ejecución, de prevención, detección y corrección de fallos).
  - ✓ Reenvío de mensajes.
  - ✓ Número de secuencia en los mensajes.
  - ✓ Utilización de protocolo subyacente orientado a conexión.
  - ✓ Duplicado, replicado de enlaces de comunicaciones.
  - ✓ Adaptabilidad a diferentes despliegues (en tiempo de carga).
  - ✓ Control de tráfico.

Atributos relacionados:
- *Modificabilidad: Adaptabilidad a diferentes enlaces de comunicaciones:*
  Favorece la implementación de los mecanismos arriba mencionados, al *independizar* las comunicaciones del resto de la aplicación..
- *Modificabilidad: Adaptabilidad a diferentes despliegues físicos:*
  Favorece la implementación de los mecanismos arriba mencionados.
- *Seguridad: Supervisión del Estado del Sistema.*
  Puede considerarse como el mismo atributo visto desde otro punto de vista.

Directrices y características de las que se deriva:

Deben adoptarse mecanismos de tolerancia a fallos, tratamiento de excepciones, errores, mecanismos de diagnóstico para prevención de fallos, etc.

Observaciones:
Aunque los mecanismos de tolerancia a fallos se definen para el tiempo de ejecución, no es necesario que los sistemas reconfiguren su despliegue en este tiempo, siempre que:

- ✓ La parada y rearranque del sistema puedan realizarse en un tiempo razonable y
- ✓ Los mecanismos puedan mantenerse en un estado seguro mientras se reconfigura el sistema:

Dada la forma en que habitualmente se realizan los trabajos de teleoperación, la flexibilidad que proporciona la reconfiguración del despliegue en tiempo de ejecución no compensa la complejidad que implica.

D2: Disponibilidad de las Interfaces de Usuario.

Descripción:

El operador debe disponer de forma permanente de una interfaz de usuario mínima que le permita ejecutar al menos la parada segura del sistema y el rearranque del mismo.

Debe existir una interfaz reducida mediante la cual puedan ejecutarse los comandos más básicos de los mecanismos.

La probabilidad de que la interfaz de usuario se bloquee y obligue a utilizar la interfaz mínima debe ser pequeña, debiendo establecerse el valor de los parámetros de fiabilidad/disponibilidad de cada sistema concreto.

Escenarios Abstractos:

Estímulos:

a) La interfaz de usuario se bloquea.
b) La interfaz no se recupera pese a repetidos intentos por rearrancarla y los mecanismos se encuentran en una situación comprometida (es *necesario* moverlos).
c) La interfaz parece funcionar, pero los mecanismos no responden.
d) Una parte de las ventanas de la interfaz no aparecen o no se ejecutan correctamente.
e) El teclado, el ratón, el joystick, etc no responden.

Respuestas:

a) Existe una interfaz mínima alternativa mediante la cual se puede ejecutar la parada segura y rearrancar el sistema.
b) Existe una interfaz reducida mediante la cuál se pueden ejecutar comandos básicos.
c) Existe una forma de control alternativa (botonera, acceso directo a controladores de bajo nivel, etc) que permite operar los mecanismos de forma básica.
d) Igual que el anterior.
e) Igual que el anterior.

Estrategias y Mecanismos:

- Interfaces alternativas que:
  - ✓ Accedan al resto de la aplicación como la interfaz original, aunque implementando sólo una parte de la funcionalidad de aquella (p.e.: sustitución de la interfaz gráfica por interfaces textuales que permitan introducir los comandos más básicos)
  - ✓ Puenteen al resto de la aplicación (p.e: setas de seguridad, botoneras cableadas directamente a los accionadores de los dispositivos, etc).

Atributos relacionados:

- *Modificabilidad: Adaptabilidad a diferentes interfaces de usuario:*
  Favorece la implementación de los mecanismos arriba mencionados, al *independizar* la interfaz de usuario del resto de la aplicación..
- *Seguridad: Parada Segura:*
  Puede considerarse un aspecto del mismo atributo, pero considerado desde otro punto de vista.

Directrices y características de las que se deriva:

Deben adoptarse mecanismos de tolerancia a fallos, tratamiento de excepciones, errores, mecanismos de diagnóstico para prevención de fallos, etc.

D3: Disponibilidad de los Mecanismos de Parada Segura.

Descripción:
Debe existir al menos un mecanismo de parada segura accesible por el operador y cuyo funcionamiento no dependa del comportamiento de la aplicación.

Escenarios Abstractos:
Estímulos:
- La aplicación se bloquea.

Respuestas:
- El operador puede ejecutar una parada segura.

Estrategias y Mecanismos:
- Botón de parada segura directamente cableado a los dispositivos.

Atributos relacionados:
- *Seguridad: Parada Segura:*

Directrices y características de las que se deriva:

Deben adoptarse mecanismos de tolerancia a fallos, tratamiento de excepciones, errores, mecanismos de diagnóstico para prevención de fallos, etc.

Las operaciones sólo deben ejecutarse si existe una confianza razonable en que el resultado de las mismas no va a poner en peligro la integridad de bienes y equipos.

I1: Escenarios de la interoperabilidad.

Descripción:

- Posibilidad de repartir las diferentes actividades de una misión entre diferentes sistemas de teleoperación. O dicho de otra manera, posibilidad de descomponer una misión en misiones más simples asignando cada una de ellas a un sistema de teleoperación determinado.
- Establecer mecanismos de sincronismo y coordinación de forma que los diferentes sistemas puedan acompasar el ritmo de sus trabajos.
- Establecer mecanismos que permitan un control global de todos los sistemas de teleoperación compatible con los sistemas de control propios de cada sistema individual.

Escenarios Abstractos:

Estímulos:

a) Se quiere repartir una misión compleja entre diferentes sistemas de teleoperación, de forma que cada uno realice una parte del trabajo.
b) Uno de los subsistemas ha terminado una parte de su trabajo, pero no puede continuar hasta que otro sistema termine una parte del suyo.
c) El operador de los sistemas desea consultar el estado global del sistema o de un subconjunto de los subsistemas.
d) El operador de los sistemas desea realizar una acción que afecta a todos los subsistemas o a una parte de los subsistemas.
e) El operador de los subsistemas desea reestructurar el reparto de tareas.
f) Se detectan problemas que deben ser resueltos a nivel de sistema individual.

Respuestas:

a) Existe una interfaz a través de la cual puede repertirse la misión entre los diferentes sistemas.
b) Existen mecanismos de coordinación y sincronismo entre los dispositivos que les permiten acompasar el ritmo de sus trabajos o detenerse si es necesario.
c) Existe una interfaz a través de la cual puede inspeccionarse el estado de cada subsistema, que ofrece además información global.
d) Existe una interfaz a través de la cual puede accederse a una parte de los comandos de cada subsistema. Puede realizarse un broadcast.
e) Idem a)
f) Puede delegarse el control a la interfaz de operador de cada sistema individual.

Anexo III

Complementos al Modelo de Análisis

3.1. Introducción

En este anexo se ofrecen algunos diagramas complementarios al modelo de análisis del sistema GOYA realizado en el Capítulo 7: Modelo de datos de las actividades de mantenimiento y desarrollo detallado de casos de uso. En concreto, se ofrece un modelo de datos para las actividades de mantenimiento en un astillero, una caracterización detallada de los casos de uso para un robot de limpieza de barcos.

3.2. Modelo de Datos Limpieza de Buques

3.2.1. Justificación

El siguiente modelo de datos se ha construido a partir de la información aportada por Izar Carenas [IZAR-CT] y viene a representar la información pertinente en el proceso de reparación de la cubierta de buques, dejando fuera aspectos de contratación, pedidos, facturación, etc. que son accesorios para el objetivo que nos ocupa.

Se ha construido un Modelo Entidad-Relación clásico como comienzo para entender mejor el problema, aunque no es un diagrama UML, si que está normalizado, con lo que quedan descartados problemas de redundancia o pérdida de información. Las cardinalidades mínimas y máximas son importantes estén correctamente especificadas. Su modificación, por restricciones del dominio del problema, conllevará lo más probable la necesidad de suprimir o añadir entidades a dicho diagrama.

3.2.2. Algunas restricciones de integridad o de cardinalidad relevantes

Las siguientes restricciones se deducen del diagrama Entidad-Relación aunque se detallan para facilitar su compresión:

- Un buque puede venir al astillero a reparación un número indeterminado de veces (o ninguna).
- Un buque tiene un armador asociado y sólo uno.
- Un buque tiene un número específico (al menos uno) de áreas que necesitan tratamiento de entre un catálogo genérico de áreas de buques. El grado de detalle asociado a la superficie del área dependerá de cómo se introduzca en el sistema.
- Un área de un buque para una fecha de entrada concreta necesita de ninguna a muchas capas de pintura.
- El área de un buque posee un estado concreto de su pintura.
- Un área de un buque necesita de ninguno a varios tratamientos cada uno de ellos a una calidad concreta de entre un catálogo de tratamientos y de acabados posibles.
- Un área catalogada de un buque posee un número de fallos específicos de dicho área (mínimo uno). Cada uno de dichos fallos puede presentar un tipo de incrustación específica y un tipo de corrosión, de entre un catálogo de corrosiones e incrustaciones posibles.

3.2.3. Descripción de las entidades

No se han incluido los atributos de cada entidad para simplificar la compresión del sistema. En este apartado, además de describir las entidades, se detallarán algunos de dichos atributos sobre todo aquellos que tengan que ver con restricciones de integridad o datos fundamentales para entender el problema. Se han subrayado los atributos que forman la clave primaria (posiblemente compuesta). Se han indicado en cursiva los atributos que son *clave ajena* en otra entidad (en la cual son o forman parte de la clave primaria). Los puntos suspensivos indican que hay que completar la lista con los datos relativos a la entidad y que, en principio, no estarán relacionados como clave ajena con ninguna otra entidad por lo que su compleción puede ser a posteriori. El orden en el que se exponen es el que facilita en mayor medida la compresión del modelo.

Entidad: Buque.

Descripción: Cada entidad Buque representa al barco físico del cual sólo existe un ejemplar.

Atributos: Código del Buque, *Código del Armador*, Fecha de Construcción, Nombre, Tonelaje, Tipo, ...

Entidad: Armador.

Descripción: Los distintos armadores.

Atributos: Código del Armador, Fecha de Construcción, Nombre, Tonelaje, Tipo,...

Entidad: Estado Buque.

Descripción: Representa las distintas situaciones en las que el buque ha llegado al astillero para reparación.

Atributos: *Código del Buque*, *Código del Armador*, Fecha de entrada al astillero, ...

Entidad: Áreas Tratamiento.

Descripción: Incluye las distintas áreas de un buque que pueden ser objeto de tratamiento en el astillero. Por ejemplo, Obra viva, línea de flotación, obra muerta, etc. Recogerá todas las posibles partes de todos los dintintos buques.

Atributos: Código Área, Nombre del Área,...

Entidad: Áreas de un Buque.

Descripción: Incluye las distintas áreas de un buque que ha entrado al astillero y necesitan tratamiento. De las áreas que no necesitan tratamiento no se recoge información alguna.

Atributos: *Código Área*, *Código del Buque*, *Fecha de Entrada del Buque*, *Código Estado Pintura*,...

Entidad: Estado Pintura.

Descripción: Incluye las distintas situaciones en las que puede estar la pintura del casco de los buques. Las básicas son Bueno, Aceptable, Pobre, ...

Atributos: Código Estado Pintura, Nombre del Estado,...

Entidad: Tipos de Pintura.

Descripción: Incluye los distintos tipos de pintura. Se pued extender el modelo por este lado a partir de los esquemas de pintura proporcionados por los fabricantes. Se ha simplificado la cuestión pensando en que se recoge información del tipo "1 Capa de F-723-G", ...

Atributos: Código Tipo Pintura, Nombre del Tipo, Nombre del Fabricante,...

Entidad: Elementos para Pintado.

Descripción: Recoge los distintos elementos físicos empleados para el pintado de los buques cuando es relevante recoger información relativa a que se ha utilizado (por ejemplo, por ser de mayor coste para el astillero).

Atributos: Código Elemento para Pintado, Nombre del Elemento,...

Entidad: Capas de Pintura por Área.

Descripción: Representa las distintas capas de pintura (utilizando determinados elementos de pintado) para cada una de las Áreas de cada uno de los buques. Dicho de otra forma, permite saber qué capas, con qué acabado, en qué parte del barco, para un barco dado y qué elementos se han utilizado. Se ha simplificado el modelo relacional asumiendo se indica a lo sumo un elemento de pintado.

Atributos: *Código Área*, *Código del Buque*,, *Código Tipo Pintura*, , *Código para Pintado*, Calidad del Acabado,...

Entidad: Tipos de Chorreado.

Descripción: Incluye los distintos tipos chorreado que se pueden aplicar a los buques en lo que a la calidad del acabado se refiere (SA 2, SA 2 1/2, ...). Entre este conjunto se elegirá uno específico para el tratamiento que reciba un área específica.

Atributos: Código Tipo Chorreado, Nombre Estándar del Tipo de Chorreado,...

Entidad: Tratamientos.

Descripción: Se refiere a los distintos tratamientos que puede recibir un buque en las distintas áreas que lo conforman. Limpieza a presión, desengrasado, chorreado, etc. son algunos ejemplos.

Atributos: Código Tratamiento, Nombre del Tratamiento,...

Entidad: Tratamientos por Área.

Descripción: Incluye los distintos tratamientos que deben aplicarse en cada una de las áreas de cada uno de los buques. El tipo de chorreado que se aplica para un área de un buque es único.

Atributos: *Código Tratamiento*, *Código Área*, *Código del Buque*, *Fecha de Entrada del Buque*, *Código Tipo Chorreado*,...

Entidad: Fallos Posibles.

Descripción: Incluye los fallos que pueden presentar las cubiertas de cascos de buques. Por ejemplo, burbujas, pérdida total, pequeños desprendimientos, etc.

Atributos: Código Fallo, Nombre del Fallo,...

Entidad: Fallos Detectados por Área.

Descripción: Incluye los fallos que presenta el área específica de un buque que ha entrado a reparación.

Atributos: Código Fallo, *Código Área*, *Código del Buque*, *Fecha de Entrada del Buque*, *Código Tipo Corrosión*, *Código Tipo Incrustación*,...

Entidad: Tipos de Corrosión.

Descripción: Incluye los distintos tipos de corrosión en las zonas que se ha detectado pérdida total de pintura. Incluye "Superficie oxidada sin picaduras", "picaduras profundas", Burbujas oxidadas con pequeñas picaduras debajo", etc.

Atributos: Código Tipo Corrosión, Nombre Tipo Corrosión,...

Entidad: Tipos de Corrosión por Área.

Descripción: Esta entidad está dedicada a recoger qué tipos de pérdida de pintura tiene asociado el fallo de un área de buque "pérdida total de pintura". Destacamos el que se recoge qué tipo de corrosión particular a la pérdida total de pintura trajo un área específica de un buque dado en la fecha que entró en el astillero para reparación.

Atributos: *Código Tipo Corrosión*, Código Fallo, *Código Área*, *Código del Buque*, *Fecha de Entrada del Buque*,,...

Entidad: Tipos de Incrustación.

Descripción: Esta entidad representa las distintas incrustaciones que puede presentar un área específica.

Atributos: Código Tipo Incrustación, Nombre del Tipo de Incrustación,...

3.2.4. Diagrama Entidad-Relación

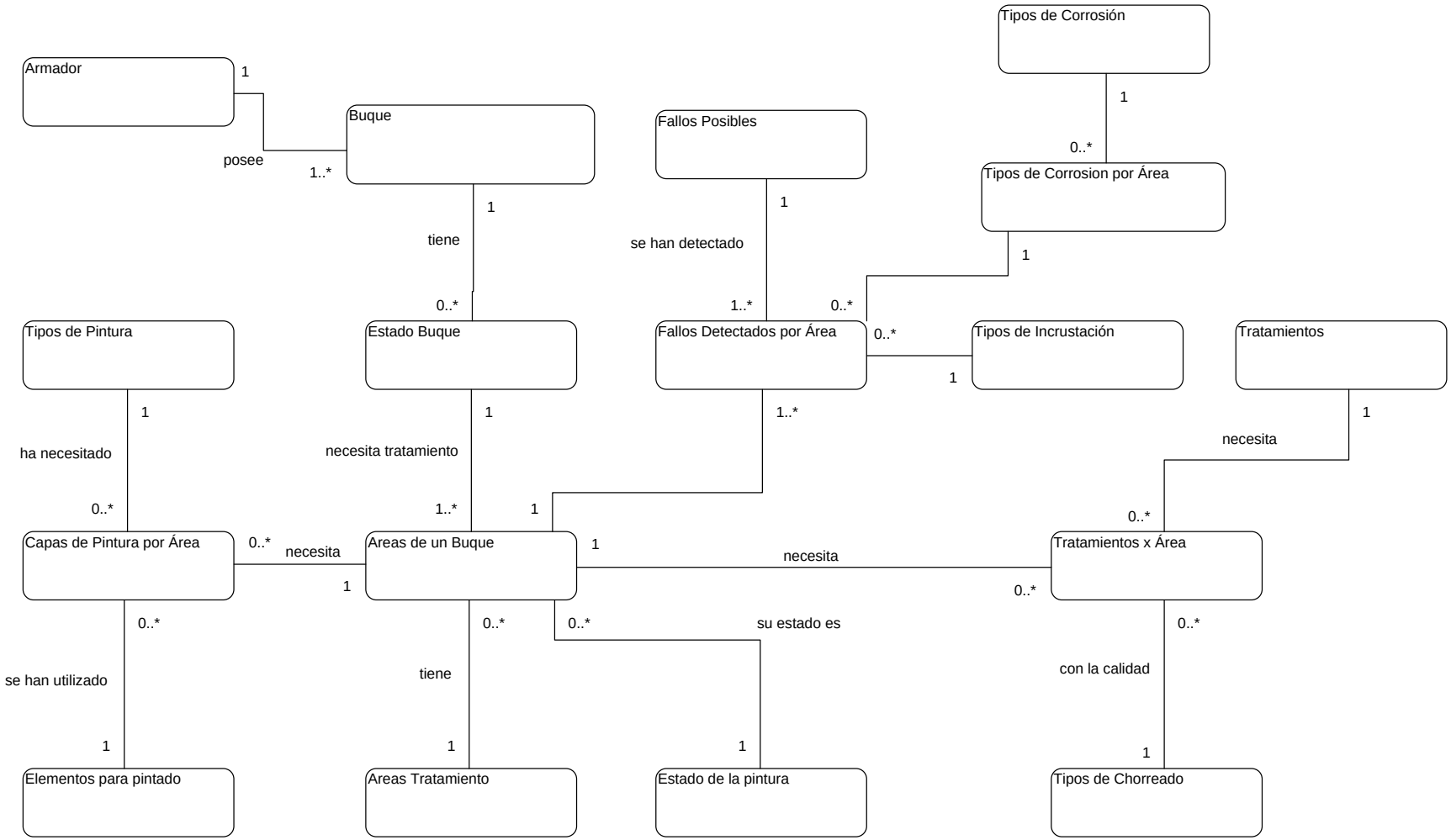


Fig. 3.1.- Diagrama entidad-relación

3.2.5. Especificación del proceso de limpieza de un barco

Antes de definir los casos de uso, habrá que conocer claramente en qué consiste el proceso de limpieza de un barco y en qué fase del mismo interviene el sistema objeto de diseño. Para ello se cuenta con la información proporcionada por el astillero IZAR [IZAR-CT], como potencial usuario del sistema GOYA. Teniendo en cuenta las diferentes personas que intervienen en el proceso de decisión comentadas en los puntos anteriores, se puede definir un diagrama de flujo del proceso de limpieza, desde que el barco entra en el astillero hasta que se finalizan los trabajos, que se muestra en la **Fig. 3.2**.

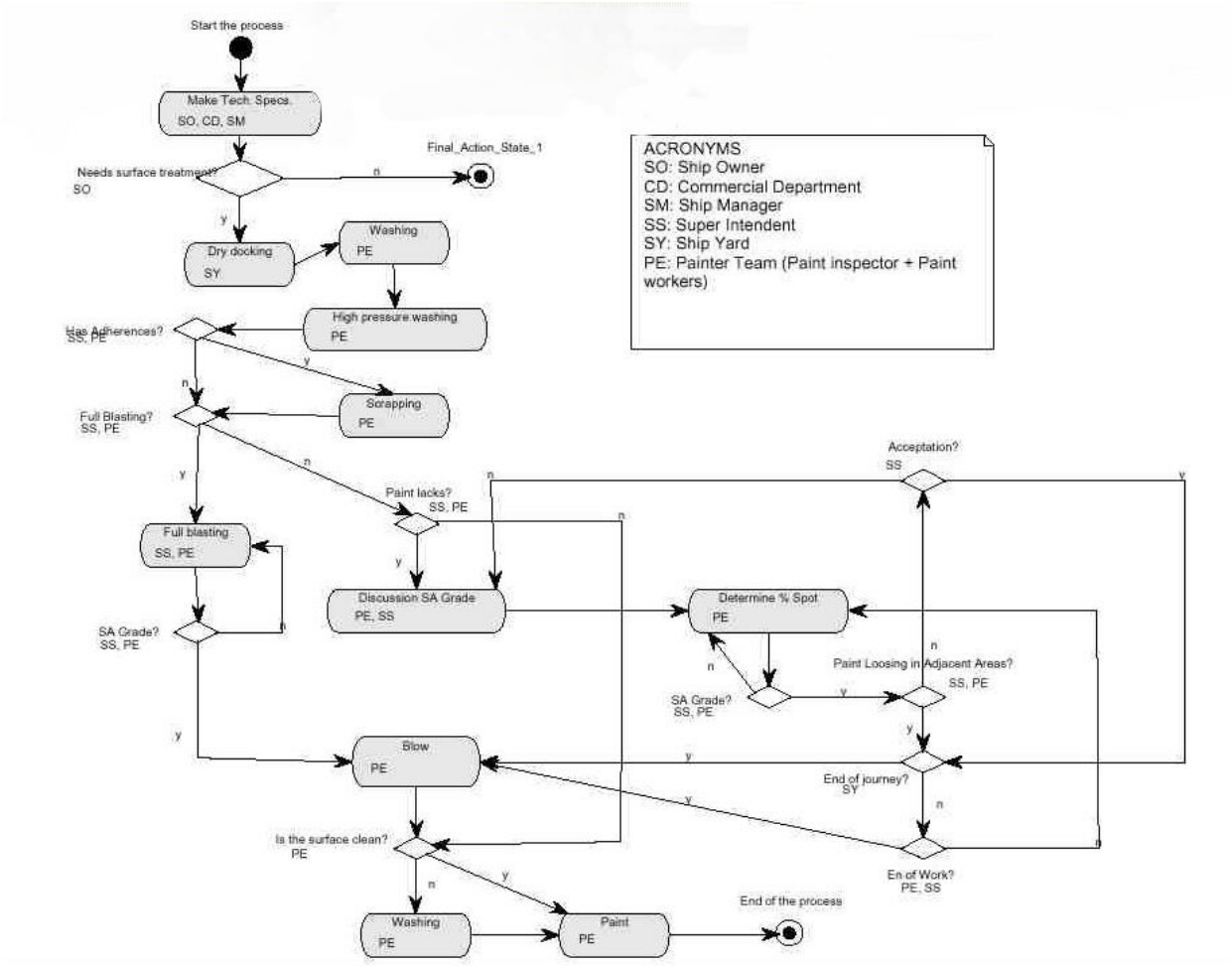


Fig. 3.2.- Diagrama de flujo del proceso de limpieza completo

3.3. Especificación de los Casos de Uso

En el Capítulo 7 ya se indicó que los casos de uso de un usuario local que accede al sistema a través de una botonera electromecánica son significativos del funcionamiento básico de un sistema típico en el dominio de la tesis y sus objetivos primarios. A continuación se procederá a la descripción de los mismos.

Aunque estos casos de uso pueden presentarse también en otros sistemas del dominio de aplicación de la arquitectura, su descripción precisa sólo será posible para un sistema concreto. A modo de ejemplo, en la siguiente página se detallan dos de estos casos de uso para el proyecto EFTCoR, para comprobar de qué manera se aborda su descripción en un sistema concreto. El resto de ellos, así como los que más adelante se presenten se pueden consultar en [GOYA98b], [EFTCoR03-d2] y [EFTCoR04-d2].

Nombre del Caso de Uso: Start Up.

Actor: Operador

Resumen: El operador pone en marcha el sistema parado para empezar granallado.

Precondición: El sistema está parado.

Secuencia Normal:

1. El operador pulsa Power.

2. El sistema muestra vídeo.

3. Se establece enlace con sistema de monitorización.

4. Se chequea el estado ok del sistema.

5. Se invoca el caso de uso "Consultar Información Tarea".

Postcondición: El sistema está arrancado.

Alternativas:

1. El sistema no está ok. Se le muestra la causa al operador y se le sugiere acción correctora para continuar.

2. No hay enlace de comunicaciones con el sistema de monitorización. Se muestra mensaje no necesitando acción correctora. Se puede funcionar sin este enlace.

3. No funciona sistema de visión. Se le muestra la causa al operador y se le sugiere acción correctora para continuar.

Comentarios: Obsérvese que se permite trabajar aunque no haya enlace con el sistema de monitorización. En el caso del sistema de visión, el funcionamiento es imprescindible.

Nombre del Caso de Uso: Shutdown

Actor: Operador

Resumen: El operador da por finalizada la tarea hasta ese momento.

Precondición: El sistema está parado.

Secuencia Normal:

1. El operador pulsa el botón de shutdown.

3. Se mandan logs de la actidad realizada al sistema de monitorización.

4. Se muestra el menú de notificación para que el usuario elija la notificación pertinente.

5. Se cierra el enlace con el sistema de monitorización.

6. Se muestra mensaje recordatorio de apagar el sistema de alimentación.

Postcondición: El sistema está off.

Alternativas: Ninguna.

Comentarios: Este caso de uso se ejecutará cuando el usuario no vaya a continuar la tarea de limpieza en los instantes posteriores (fin de jornada, etc.)

De la jerarquía de paquetes de casos de uso presentada en el Capítulo 7, se hará aquí una descripción detallada de los contenidos de cada paquete. Al igual que se ha dicho antes, aunque esta descripción ha sido hecha para el sistema EFTCoR, puede ser fácilmente extrapolable al resto de sistemas del dominio de aplicación de la tesis. Con ello se obtiene una clasificación y caracterización de casos de uso que servirá como guía y punto de partida para otras aplicaciones.

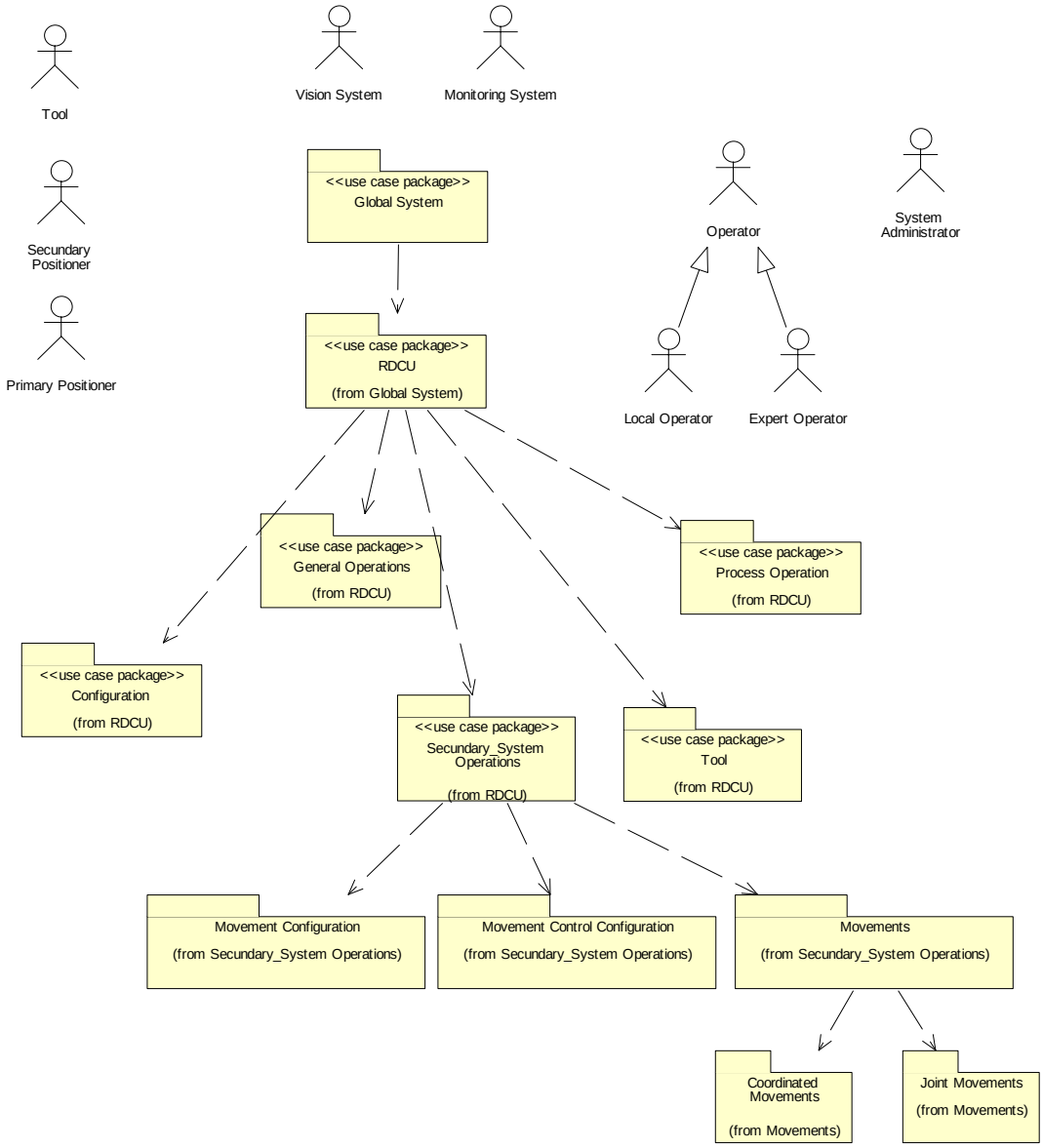


Fig. 3.3.- Jerarquía de casos de uso en el sistema GOYA

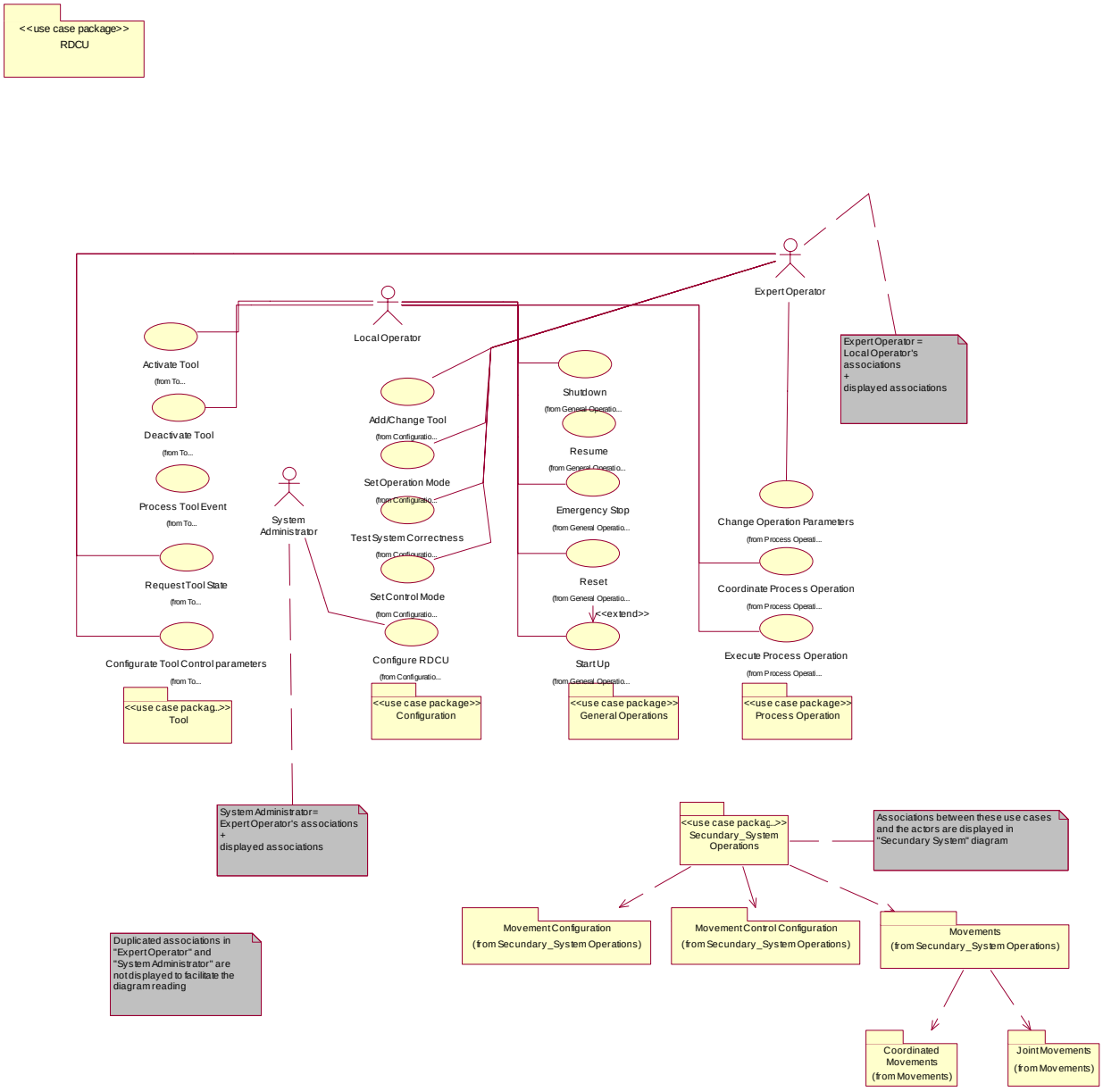


Fig. 3. 4.- Casos de uso pertenecientes al paquete general RDCU (*Robotic Devices Control Unit*)

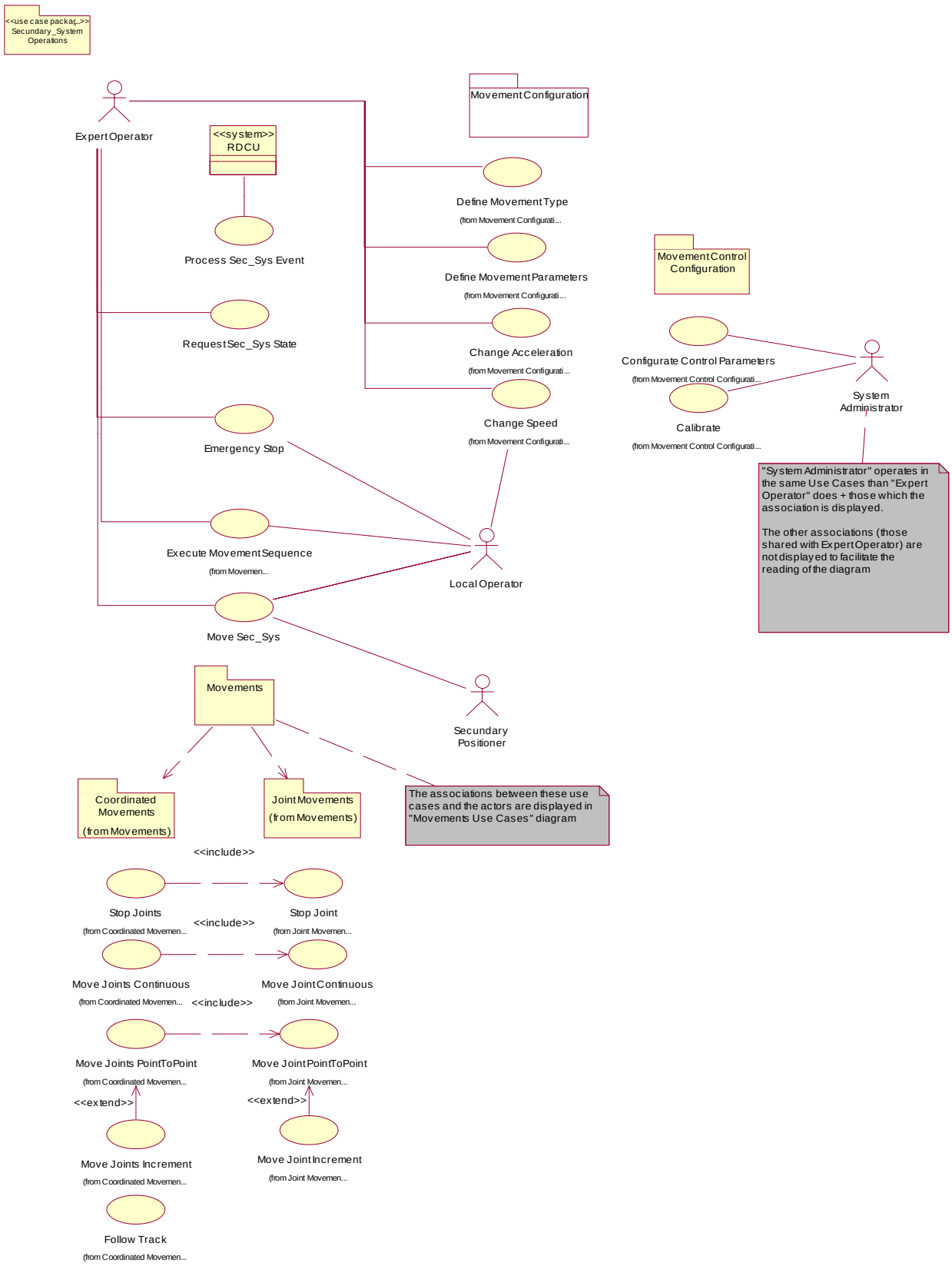


Fig. 3.5.- Casos de uso pertenecientes al paquete *Secundary System Operations*

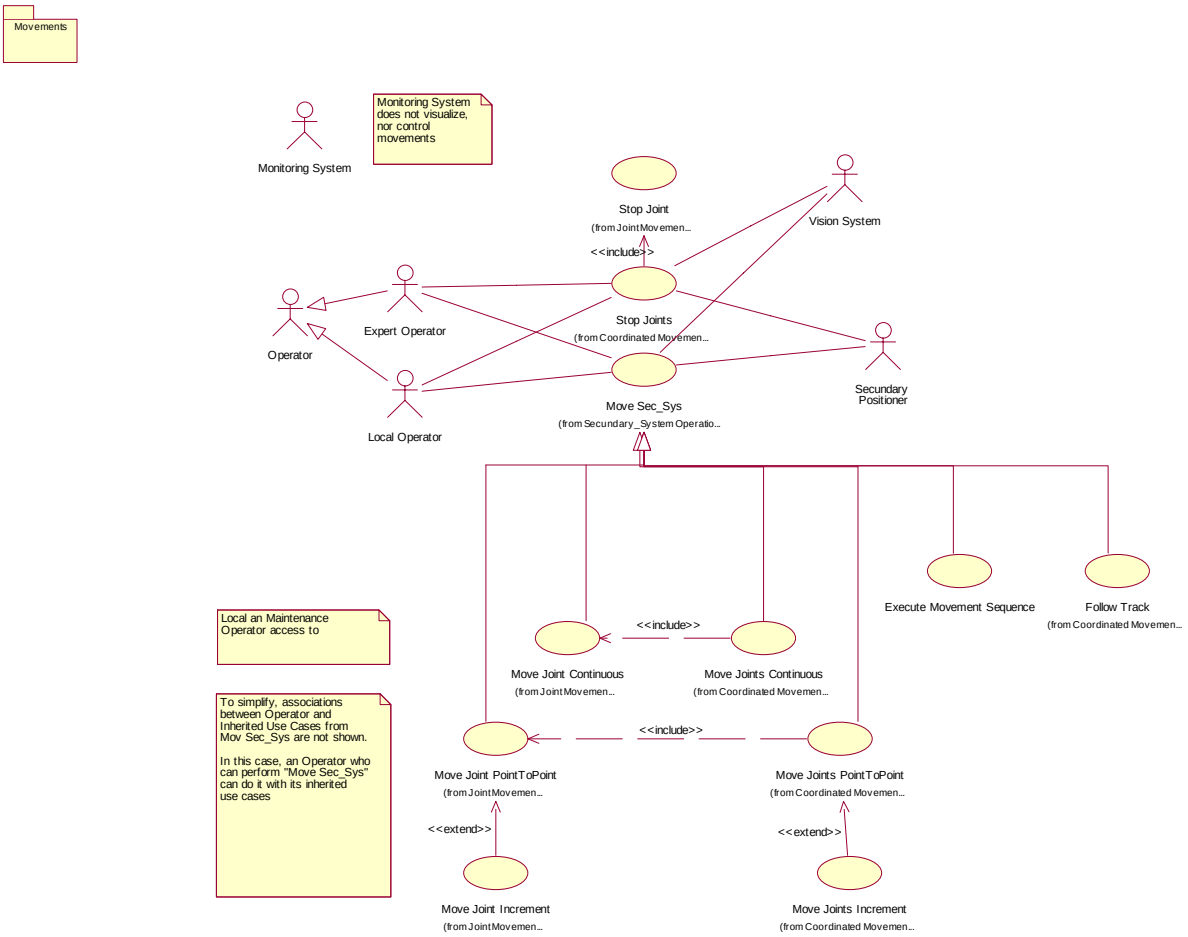


Fig. 3.6.- Casos de uso pertenecientes al paquete *Movements*

Bibliografía

[AAA95] Proyecto de Aplicaciones Automáticas para la Reducción de Dosis en Operaciones de Mantenimiento, Programa de Investigación Electrotécnico del MINER (PIE-041049). 1995

[Abderrahim99] M. Abderrahim, C. Balaguer, A. Giménez, J.M. Pastor, V.M. Padrón, "ROMA: A Climbing Robot for Inspection Operations", Proceedings of the 1999 IEEE International Conference on Robotics & Automation, Detroit, Michigan, USA. Mayo 1999.

[Ada95] Ada 95 Reference manual: Language and Standard Libraries. International Standard ANSI/ISO/IEC-8652:1995. Springer-Verlag, LNCS no. 1246.

[Ada-Answ04] http://www.gnat.com/aa_about.php Ada Answers. Ada Core. GNAT. 2004.

[Adept91] ADEPT TECHNOLOGY INC., San Jose, CA, USA. AIM Users Guide, 1991.

[Alami00] R. Alami et al, "Around the lab in 40 labs", IEEE International Conference on Robotics and Automation, ICRA'00, San Francisco, USA. 2000.

[Albus87] J. S. Albus, R. Lumia, H. McClain, "NASREM: Standard Reference Model for Telerobot control.". Technical report, NASA Conferences on Space Applications of AI and Robotics, 1987.

[Albus91] J. S. Albus, "Outline for a theory of intelligence", IEEE Transactions on Systems, Man and Cybernetics, 21(3): 473-509. 1991

[Álvarez97-td] Álvarez B., "Arquitectura Software de Referencia para Sistemas de Teleoperación", Tesis Doctoral, ETSIT, Universidad Politécnica de Madrid, 1997.

[Álvarez98] Álvarez B., Alonso A., de la Puente J.A. " Timing Analysis of a Generic Robot Teleoperation Software Architecture", Control Engineering Practice, vol.6, no.6, Junio 1998, pp.409-416. ISSN 0967-0661.

[Álvarez00a] Álvarez B., Iborra A, Alonso, A and de la Puente J.A. " Reference architecture for robot teleoperation: Development details and practice use", Proceedings of the 6th IFAC Workshop on Algorithms and Architectures for Real-Time Control. Palma de Mallorca (Spain), Mayo 2000. ISBN 0-08-042930-0

[Álvarez00b] Álvarez B., Iborra A, Alonso A , de la Puente J.A, Pastor J.A., "Developing multi-application remote systems", Nuclear Engineering International, ISSN-0029-5507,vol.45, no.548, Marzo 2000.

[Álvarez01a] Álvarez B, Iborra A, Alonso A, and de la Puente J.A. "Reference architecture for robot teleoperation: Development details and practical use". March 2001, Control Engineering Practice, ISSN 0967-0661

[Álvarez01b] Álvarez B, Iborra A, Sánchez P, Ortiz F, Pastor J.A. "Experiences on the Product Synthesis of Mechatronic Systems using UML in a Software Architecture Framework", 1st International Conference on Information Technology in Mechatronics ITM'01, Estambul, Turquía, 1-6 Octubre 2001

[Álvarez02] Álvarez B, Ortiz FJ, Martínez A , Sánchez, P, Pastor JA, Iborra A., "Towards a Generic Software Architecture for a Service Robot Controller", 15th IFAC World Congress, Barcelona. Julio 2002.

[ANCLA03] Subproyecto "Arquitecturas dinámicas para sistemas de teleoperación". Subproyecto del proyecto DYNAMICA [DYNAMICA03]. DSIE – UPCT. TIC2003-07804-C05-02. 2003

[Anderson95] R. Anderson, "SMART: A Modular Architecture for Robotics and Teleoperation", 4th International Symposium on Robotics and Manufacturing, Santa Fe, México. Noviembre 1995.

[Andrade99] L.F. Andrade, J.L. Fiadeiro, "Interconnecting Objects via Contracts", UML'99 - Beyond the Standard, R. France and B. Rumpe (eds), LNCS1723, Springer Verlag 1999, 566-583

[ANSI99] "American National Starndard for Industrial Robots and Robot Systems – Safety Requirements", ANSI/RIA R15.06-1999. Robotic Industries Association, Ann Arbor MI 4 8106.

[Aracil02] R. Aracil, E. Pinto, M. Ferre, "Robots for Live-Power Lines: Maintenance and Inspection Tasks", DISCA, UPV, 15th Triennial World Congress IFAC, Barcelona, España. 2002.

[Arcara02-td] P. Arcara, "Control of Haptic and Robotic Telemanipulation Systems", Ph.D. Thesis, Universidad de Bolonia, Italia. 2001.

[Arkin89] R.C. Arkin, "Motor schema-based mobile robot navigation", International Journal of Robotics Research, 8, pp. 92-112. 1989.

[Axelsson00] J. Axelsson, "Real-World Modeling in UML", Volvo Technological Development Corporation, Göteborg, Suecia. 2000.

[Backes90] P.G. Backes, K.S. Tso, "UMI: An interactive supervisory and shared control system for telerobotics", Proc. IEEE Int. Conf. on Robotics and Automation, Cincinnati, OH, USA. Mayo 1990.

[Backes93] P.G. Backes, M. Long, R. Steele, "The modular telerobot task execution system for space telerobotics", Proc. IEEE Int. Conf. on Robotics and Automation, Atlanta, GA, USA. Mayo 1993.

[Bachmann00a] F. Bachmann, L. Bass et al, "The Architecture Based Design Method", Technical Report CMU/SEI-200-TR-001, Carnegie Mellon University, USA. Enero 2000.

[Bachmann00b] F. Bachmann, L. Bass et al, "An application of the Architecture Based Design Method to the Electronic House", Special Report CMU/SEI-200-SR-009, Carnegie Mellon University, USA. Enero 2000.

[Bahill03] T. Bahill, J. Daniels, "Using Objected-Oriented and UML Tools for Hardware Design: A Case Study", Systems Engineering, Vol. 6, No. 1, Wiley Periodicals, Inc. 2003

[Barabanov97] M. Barabanov. "A Linux-based Real-Time Operating System". Master Thesis, New Mexico Institute of Mining and Technology, Socorro, New Mexico, Junio 1997.

[Barreca02] A. Barreca, G. Cannata, L. Dentone, F. Giorgi, "Embedded Control Architecture for a Redundant Robotic Arm", 15th IFAC Triennal World Congress, Barcelona, España. 2002.

[Bass98] L. Bass, P. Clements, K. Kazman. "Software architecture in practice", 1st Ed. Addison-Wesley, 1998. ISBN 0-201-19930-0.

[Bass99] L. Bass, R. Kazman, "Arquitecture-Based Development", Technical Report CMU/SEI-99-TR-007, Software Engineering Institute, Carnegie Mellon University, USA. Abril 1999.

[Bass00] L. Bass, M. Klein, F. Bachmann, "Quality Attribute Design Primitives", Technical Report, CMU/SEI-2000-TR-017. Software Engineering Institute, Carnegie Mellon University, USA. 2000.

[Bates98] J. Bates, "The State of the Art in Distributed and Dependable Computing", Laboratory for communications Engineering, University of Cambridge, UK, 1998.

[Beccari97] G. Beccari et al, "A Real-Time Library for the Design of Hybrid Control Architectures", Dipartimento di Ingegneria dell'Informazione, Universitá di Parama, Italia. 1997

[Bellini02] C. Bellini, F. Panepinto, S. Panzieri, G. Ulivi, "Exploiting a Real-Time Linux Platform in Controlling Robotic Manipulators", 15th IFAC Triennal World Congress, Barcelona, España. 2002.

[Benali01] A. Benali, V. Idasiak, J.G. Fontaine, "Remote robot teleoperation via Internet. A first approach", IEEE International Workshop on Robot and Human Interactive Communication, 2001.

[Björkander03] M. Björkander, C. Kobryn, "Architecting Systems with UML 2.0", IEEE Software, IEEE Computer Society. 2003.

[Boehm88] B. W. Boehm, "A Spiral Model of Software Development and Enhancement", IEEE Computer 21, nº 5. Mayo 1988.

[Boehm96] B. W. Boehm, Hoh in, "Identifying Quality Requirements Conflicts", IEEE Software. Marzo 1996

[Bonasso97] R.P. Bonasso, R.J.Firby et al "A proven three-tiered architecture for programming autonomous robots", Journal of Experimental and Theoretical Artificial Intelligence, 9(2), 1997.

[Booch94] G. Booch, "Object Oriented Analysis and Design with Applications", Redwood City, CA, Benjamin/Cummings, 1994

[Booch97] G. Booch, J. Rumbaugh, I. Jacobson. "Unified Modeling Language User Guide", Addison Wesley, 1997.

[Borrelly98] J.J. Borrelly et al, "The ORCCAD architecture", International Journal of Robotics Research, pp. 338-359. 1998.

[Bosch91] R. Bosch, "CAN Specification 2.0A", Rober Bosch GmbH.

[Bosch00] J. Bosch, "Product Line Architectures", ObjectiveView, Object Component Architecture Series, (4):13-18, http://www.ratio.co.uk

[Bosch01] J. Bosch, J. Van Gurp, M. Svahnberg, "On the Notion of Variability in Software Product Lines", Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), IEEE Computer Society, 2001.

[Brad00] Brad Appleton, http://www.enteract.com/~bradapp/docs/patterns-intro.html

[Bridge00] Project Technology Bridge, htttp://www.projtech.com

[Brooks86] R.A. Brooks, "A robust layered control system for mobile robot", IEEE Journal of Robotics and Automation", 2, pp 14-23, marzo 1986.

[Brown98] Alan W. Brown and Kurt C. Wallnau. "The current state of CBSE". IEEE Software, Vol. 15, nº 5, September/ October 1998.

[Bruyninckx02] H. Bruyninckx, B. Koninckx, P. Soetens, "A Software Framework for Advanced Motion Control", Dpt. of Mechanical Engineering, K.U. Leuven. OROCOS project inside EURON. Bélgica. Febrero 2002.

[Buiu02] C. Buiu, I. Dumitrache, F. Mihai, "Complex Behaviors of Teleoperated Robots", IFAC 15th Triennial World Congress, Barcelona, España. Julio 2002

[Burns01] A. Burns, A. Wellings, "Real-Time Systems and Programming Languages", 3rd Edition. Ed. Addison-Wesley, ISBN 0-201-72988-1. UK, 2001.

[Buschmann96] F. Buschmann, R. Meunier et al "Pattern Oriented Software Architecture Vol1: A system of patterns: Pattern languages of program design", John Wiley & Son Ltd, 1996.

[Clawar01-t13] EC Brite Euram Thematic Network on Climbing and Walking Robots Including the Support Technologies for Mobile Robotic Machines "Technical Tasks Year 3 Final Reports. Technical Task 13: Computing requirements", enero 2001.

[Clements96] P. Clements, "A Survey of Architecture Description Languages", 8th International Workshop on Software Specification and Design, Alemania. Marzo 1996.

[Colon99] E. Colon, "Virtual and augmented reality aided vehicle control", ISMCR'99.

[Connell90] J.H. Connell, P. Viola, "Cooperative Control of a Semi-Autonomous Mobile Robot", Proceedings fo the 1990 IEEE Conference on Robotics and Automation (ICRA-90). 1990.

[Connell92a] J.H. Connell, "SSS: A Hybrid Architecture Applied to Robot Navigation", Proceedings fo the 1992 IEEE Conference on Robotics and Automation (ICRA-92), pp. 2719-2724. 1992.

[Connell92b] J.H. Connell, "Designing Behavior-based Robots", www.johuco.com.

[Coplien95] J.O. Coplien, D.C. Smith, "Pattern Languages of Program Design", Addison Wesley 1995.

[Coste00] E. Coste-Manière, R.Simmons, "Architecture, the Backbone of Robotic System", Proc. of the 2000 IEEE international conference on robotics & Automation, San Francisco, abril 2000.

[Cox98] P. Cox, T.J. Smedley, "Visual Programming for Robot Control", Proceedings of IEEE Visual Languages Symposium. Halifax. 1998.

[Dario96] P. Dario et al, "Robotics for medical Applications", IEEE Robotics and Automation Magazine, Vol. 3, No. 3, septiembre 1996.

[Decotignie02] J.D. Decotignie, "Wireless Fieldbusses – A survey of issues and Solutions", IFAC 15th Triennial World Congress, Barcelona, España. Julio 2002

[Douglass00] B. P. Douglass, "Doing Hard Time. Developing Real-Time Systems with UML, Objects, Frameworks and Patterns", Object Technology Series, ISBN 0-201-49837-5. Ed. Addison-Wesley, Canada. 2000.

[Drake00] J. M. Drake, M. González Harbour, J.S. Medina, "Mast Real Time View: Graphic UML Tool for Modeling Object Oriented Real Time Systems", Grupo de Computación y Sistemas de Tiempo Real de la Universidad de Cantabria. Internal Report 2000

[D'Souza99] D. F. D'Souza et al, "Objects, Components and Frameworks with UML. The Catalysis Approach", Object Technology Series, Addison Wesley.

[DYNAMICA03] Proyecto Europeo DYNAMICA, "Dynamic and Aspect-Oriented Modeling for Integrated Component-based Architectures". U. Politécnica de Cartagena, U. Carlos III, U. Politécnica de Valencia, U. de Murcia, U. de Castilla-La Mancha, TIC2003-07804-C05-02. 2003

[EFTCoR02] Proyecto GROWTH – V Programa Marco, Unión Europea. (UPCT, IZAR Carenas, UPM, HEMPEL, INDASA, DOSH, BYG, IAPETOS, LISNAVE), Environmental Friendly and Cost-Effective Technology for Coating Removal. 2002-2005.

[EFTCoR03]. "Environmental Friendly and Cost-Effective Technology For Coating Removal". Convocatoria de acciones especiales del Ministerio de Ciencia y Tecnología. Ref. DPI2002-11583-E. Septiembre 2003.

[EFTCoR03-d1] "General Requirements" EFTCoR_WP1_D1-A_Izar. Documentación del proyecto EFTCoR. 2003

[EFTCoR03-d2] "Robot Control Unit Requirements", EFTCoR_WP_7_D7-a_UPCT_v0. Documentación del proyecto EFTCoR.2003

[EFTCoR04-d1] "Robotic Devices. Detailed Design Document", EFTCoR_WP_5_D5-c_v0. Documentación del proyecto EFTCoR. 2004

[EFTCoR04-d2] Detailed Design of the Remote Control Unit , EFTCoR_WP_7_D7-c_UPCT_v4.0. Documentación del proyecto EFTCoR. 2004.

[Egyed99] A. Egyed, N. Medvidovic, "Extending Architectural representation in UML with View Integration", Proceediigs of the 2nd International Conference on the Unified Modeling Languag (UML), Fort Colins, CO, October 1999.

[EURON00] European Robotics Research Network. www.euron.org.

[EURON01] European Robotics Forum (IFR ERF), European Robotics Research Network (EURON), "European Robotics: A white paper on the status and opportunities of the European Robotics Industry". www.euron.org. Septiembre 2001.

[EURON04] Euron Research Network, "EURON Research Roadmaps". www.euron.org. Marzo 2004

[Fayad97] M. Fayad, D.C. Schmidt, "Object-Oriented Application Frameworks", Communications of the ACM, Vol. 40, No. 10, octubre 1997.

[Fernández04] Fernández, J.L, "Architecting Real Time Systems. PPOOA, an Architectural Style, PAP, an architecting Process, PPOOA-Visio, a CASE tool". ETS Ingenieros Industriales, Universidad Politécnica de Madrid. 2004.

[Ferre97-td] Ferre M, "Diseño de interfaces avanzadas para robots teleoperados. Desarrollo de un entorno de teleoperación", Tesis Doctoral, DISAM, Universidad Politécnica de Madrid, 1997.

[Fink97] B. Fink et al. "Cartesian Controlled heavy Machines Supported by Advanced Human-Machine-Interfaces", 14th International Symposium on Automation and Robotics in Construction, Pittsburgh, USA, 1997.

[Fiorini97] P. Fiorini, R. Oboe, "Issues on Internet-Based Teleoperation", Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA. 1997.

[Firby95] R. James Firby and Marc Slack. "Task execution: Interfacing to reactive skill networks" Working notes of the 1995 AAAI Spring Symposium on Lessons Learned from Implemented Architectures for Physical Agents, 1995.

[Fong01] T. Fong, C. Thorpe, C. Baur, "A Safeguarded Teleoperation Controller", The Robotics Institute, Carnegie Mellon University. IEEE International Conferenceon Advanced Robotics, Budapest, Hungría. Agosto 2001

[Fong02] T. Fong, C. Thorpe, "Robot as Partner: Vehicle Teleoperation with Collaborative Control", The Robotics Institute, Carnegie Mellon University. USA 2002.

[Gamma95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object Oriented Software", Addison Wesley, Reading Mass. 1995.

[Gannod00] G.C. Gannod, R.R. Lutz, "An approach to Architectural Analysis of Product Lines", Proceed. of the IEEE 2000 International Conference on Software Engineering, 548-557

[García02] F.J. García, J.A. Barras, M.A. Laguna, J.M. Marqués, "Líneas de Productos, Componentes, Frameworks y Mecanos", Informe Técnico DPTOIA-IT-2004-004. Dpto. Informática y Automática, Universidad de Salamanca. Marzo 2002.

[García03] E. Garcia, J. Estremera, P. Gonzalez de Santos, "A control architecture for humanitarian-demining legged robots".6th Int. Conf. . Climbing and Walking Robots, Catania, Italia. Septiembre 2003.

[Garlan94] D. Garlan, M. Shaw. "An Introduction to Software Architecture". CMU Software Engineering Institute Technical Report. CMU/SEI-94-TR-21. Pittsburg, USA. Enero 1994.

[Garlan95] D. Garlan and D. Perry. "Introduction to a special issue on software architecture". IEEE Transactions Software Engineering, 21(4), Abril 1995.

[Garlan00] D. Garlan, A.J. Kompanek. "Reconciling the Needs of Architectural Description with Object-Modeling Notations". Carnegie Mellon University. Proceedings of the 3rd International Conference on the UML. York, UK. Octubre 2000.

[Gasperoni01] F. Gasperoni, "Developing Software that Matters", ACT Europe. http://libre.act-europe.fr/Software_Matters. 2001

[Georgeff89] M. P. Georgeff and F. F. Ingrand, "Decision-making in an embedded reasoning system" In Proc. of the International Joint Conference on Artificial Intelligence, pp. 972-978, Tokyo, Japan, 1989

[Glass02] R. L. Glass, I. Vessey, V. Ramesh, "Research in software engineering: an analysis of the literature", Information and Software Technology 44, pp.491-506. Elsevier. 2002.

[Gomaa00] Hasan Gomaa, "Designing Concurrent, Distributed and Real Time Applications with UML", Addison-Wesley, ISBN 0-201-65793-7. Mayo 2000.

[González01] M. González-Harbour, M. Aldea, "MaRTE OS: An Ada Kernel for Real-Time Embedded Applications", Int. Conf. On Reliable Software Technologies, Ada-Europe'01. Leuven, Mayo 2001.

[GonzálezDS00] P. González de Santos, M.A. Armada, M.A. Jiménez, "Ship Building with ROWER", IEEE Robotics & Automation Magazine, pp. 35-43. Dic. 2003.

[GOYA98a] Proyecto FEDER (UPCT, Izar Carenas) Nº 1FEDER IFD97 - 0823 (TAP). "Robot Escalador para la limpieza de cascos de buques, respetuoso con el medio ambiente".

[GOYA98b] GOYA-WP7-001. Documento de especificación de requisitos detallado del proyecto GOYA. 1998.

[GOYA99] GOYA-WP7-003. Documento de arquitectura y diseño detallado del proyecto GOYA. 1999.

[Gowdy00] Jay Gowdy, "A Qualitative Comparison of Interprocess Communications Toolkits for Robotics", Carnegie Mellon University Technical Report, CMU-RI-TR-00-16. USA. Junio 2000.

[Grange00] S. Grange, T. Fong, C. Baur, "Effective Vehicle Teleoperation on the World Wide Web", IEEE International Conference on Robotics and Automation (ICRA 2000), San Francisco, USA. Abril 2000.

[Granot01] R. Granot, "Telerobotics – a New Paradigm", International Symposium on Mechatronics, Budapest. Noviembre 2001.

[Graves99] A. Graves, C. Czameck, "A Generic Control Architecture for Telerobotics", SMART. Proceedings of Towards Intelligent Mobile Robots. Bristol, UK. Marzo 1999.

[Graves01] A. Graves, C. Czarnecki, "Design Patterns for Behavior-based Robotics", IEEE Transactions on Systems Man and Cybernetics Part A: Systems and Humans. Enero 2000.

[Hans02] M. Hans, B. Graf, R.D. Schraft, "Robotic Home Assistant Care-O-bot: Past – Present – Future", Fraunhofer institute for manufacturing Engineering and automation, Stuttgart, Alemania. 2002.

[Harel90] D. Harel, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", IEEE Transactions on Software Engineering, 16(4), April 90

[Hasemann95] J.M. Hasemann, "Robot Control Architectures: Application, Requirements, Approaches, and Technologies", SPEI Intelligent Robots and Manufacturing Systems, Philadelphia, PA, 1995.

[Held92] R.M. Hels, N.I. Durlach, "Telepresence", Presnece, Vol.1, no.1, enero 1992.

[Hirzinger93] G. Hirzinger et al, "Sensor-based space robotics: ROTEX and its telerobotic features", IEEE Transactions on Robotics and Automation, vol.9, no.5. Octubre 1993.

[Hofmeister99] C. Hofmeister, R. Nord, D. Soni, "Describing Software Architecture with UML", Siemens Corporate Research, Princeton, New Jersey. Proceedings of the First Working IFIP Conference on Software Architecture, Kluwer Academic Publisher. USA 1999

[Hofmeister00] C. Hofmeister, R. Nord, D. Soni, "Applied Software Architecture", Addison-Wesley. ISBN 0-201-32571-3. USA. Enero 2000.

[Huang91] H. M. Huang, R. Quintero, J.S. Albus, "A Reference Model, Design Approach, and Development Illustration toward Hierarchical Real- Time System Control for Coal Mining Operations", Advances in Control & Dynamic Systems, Academic Press. 1991

[Iborra00] Iborra A, Alvarez B., Pastor J.A, Fdez Meroño J.M. "Robotized system for retrieving fallen objects within the reactor vessel of a nuclear power plant (PWR)" IEEE International Symposium of Industrial Electronics (ISIE´2000), Puebla (México). Diciembre, 2000.

[Iborra01] Iborra A, Álvarez B, Ortiz F, Marín F, Fernández JM., "Service Robot for Hull-Blasting", 27th Annual Conference of the IEEE Industrial Electronics Society. IECON01. Denver, USA. Noviembre 2001.

[Iborra03] Iborra A, Pastor J.A., Álvarez B., Fernández C. and Fernández-Meroño J. M., "Robots in Radioactive Environments", IEEE Robotics and Automation Magazine, vol. 10, no. 4, pp. 12-22. Diciembre 2003.

[IEEE-610.12] IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology. New York, NY, IEEE-1990

[IEEE Std 1471-2000] The IEEE 1471-2000 Standard, "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems". The Institute of Electrical and Electronics Engineers, Inc. New York, USA. Septiembre 2000.

[Ilogix00] I-Logix, http://www.ilogix.com

[Islam96] N. Islam, M. Deravakova, "An Essential Design Pattern for Fault Tolerant Distributed State Sharing", Communications of the ACM, Octubre 1996.

[IZAR-CT] IZAR – Carenas Cartagena. www.izar.es/carenas

[Jacobson92] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, "Object OrientedSoftware Engineering: A Use-Case Driven Approach", Reading, MA, Addison-Wesley, 1992

[Jacobson00] I. Jacobson, G. Booch, J. Rumbaugh, "El Proceso Unificado de Desarrollo de Software", Addison Wesley. ISBN 84-7829-036-2. Madrid, 2000.

[Kande00] M.M. Kandé, A. Strohmeier, "Towards a UML Profile for Software Architecture Descriptions", The UML 3rd International Conference, York, UK. Octubre 2000

[Kang90] K.C. Kang, S. Cohen, "Feature Oriented Domain Analysis (FODA) Feasibility Study", Technical Report CMU/SEI, 1990

[Kazman94] R. Kazman, L. Bass, G. Abowd, M. Webb, "SAAM: A Method for Analyzing the Properties of Software Architectures", Proceedings of 16th International Conference on Software Engineering, IEEE-1994.

[Kazman00] R. Kazman, M. Klein, P. Clemens, "ATAM SM: Method for Architecture Evaluation", Technical Report, CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, USA ,2000.

[Kiczales97] G. Kickzales et al, "Aspect Oriented Programming", Proceeedings of the European Conference on Object Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997

[Klein99] M. Klein, R. Kazman, Len Bass, J. Carriere, M Barbacci, H. Lipson, "Attribute-Based Architectural Styles", Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, Feb 99.

[Konolige96] K. Konolige, K. Myers, "The Saphira Architecture for Autonomous Mobile Robots", Artificial Intelligence Center SRI International, CA – USA. Noviembre 1996.

[Kobryn03] C. Kobryn, E. Samuelsson, "Driving Architectures with UML 2.0", Telelogic White Paper. 2003.

[Kopetz97] H. Kopetz. "Real-Time Systems. Design Principles for Distributed Embedded Applications". Kluwer Academic Publishers, ISBN-0-7923-9894-7. Massachusett, USA, 1997.

[Kopetz00] H. Kopetz, "Software Engineering for Real-Time: A Roadmap", The Future of Software Engineering, ACM Press, Jun. 2000.

[Kopetz02] H. Kopetz, G. Bauer, "The Time-Triggered Architecture", Proceedings of the IEEE Special Issue on Modeling and Design of Embedded Software, Oct 2002.

[Kozaczynski98] W. Kozaczynski and G. Booch. "Component-Based Software Engineering". IEEE Software, Vol. 15, nº 5. September/October, 1998.

[Kramer93] T.R. Kramer, M.K. Senehi, "Feasibility Study: Reference Architecture for Machine Control System Integration". Tesis Doctoral. Catholic University. National Institute of Standards and Technology. USA, 1993.

[Kratz98] J. Kratz, "Unified Modeling Language for Real-Time Systems Development" Dpt. of Computing Science, University of Nijmegen, Holanda. 1997.

[Kruchten95] P.B. Kruchten, "The 4+1 View Model of Architecture". IEEE Software, pp. 42-50. Noviembre 1995.

[Larcombe84] Larcombe and J.R. Halsall. "Robotic in nuclear engineering". C.E.C., EUR 9312 EN, 1994.

[Liddle00] D. Liddle, "Recent Developments in Underwater Remotely Operated Vehicles", EEZ Technology. 2000.

[Lin95] I. Lin et al, "An Advanced Telerobotic Control system for a Mobile Robot with Multisensor Feedback", IAS-4, IOS Press, 1995.

[Lorenc97] S.J. Lorenc, B.E. Handlon, L.E. Bernold, "Development of a Robotic Bridge maintenance System", 14th International Symposium on Automation and Robotics in Construction, Pittsburgh, USA. 1997.

[Macchelli00] A. Macchelli, C. Melchiorri, D. Pescoller, "An Experimental Set-up for Robotics and Control Systems Research using Real-Time Linux and Comau SMART 3-S Robot", LAR – Laboratorio di Automazione e Robotica, University of Bologna, Italia. 2000.

[Macchelli02] A. Macchelli, C. Melchiorri, "A Real-Time Control system for Industrial Robots and Control Applications Based on Real-Time Linux", 15th IFAC Triennal World Congress, Barcelona, España. 2002.

[Mallet02a] A. Mallet, S. Fleury, "Tentative Specification of Component's Interface", working document for OROCOS project. LAAS – CNRS, Toulouse, Francia. 2002.

[Mallet02b] A. Mallet, S. Fleury, H. Bruyninckx, "A Specification of Generic Robotics Software Components: Future Evolutions of GenoM in the Orocos Context", LAAS – CNRS, Toulouse, Francia. 2002.

[Masmano03] M. Masmano, J. Real, I. Ripoll and A. Crespo. "Running Ada on Real-Time Linux". Ada Europe 2003, LNCS 2655, pp. 322-333, 2003.

[MAST01] J.M. Drake, M. González-Harbour, J.L. Medina. "Vista UML de Tiempo Real de Sistemas Diseñados bajo una Metodología Orientada a Objetos", Grupo de Computadores y Tiempo Real, Universidad de Cantabria. Santander, España. 2001.

[MCA2] "Modular Controller Architecture", K.U. Scholl, Universidad de Karlsruhe, Alemania. Disponible en http://mca2.sourceforge.net 2001

[Medvidovic99] N. Medvidovic, R. Taylor, "A Classification and Comparison Framework for ADLS", Computer Science Department, University of Southern California. USA, 1999.

[Medvidovic00] N. Medvidovic, D.S. Resenblum, J.E. Robbins, D.F. Redmiles, "Modeling Software Architectures in the Unified Modeling Language", Computer Science Department, University of Southern California. USA, 2000.

[Medvidovic01] N. Medvidovic, A. Egyed, D.S. Rosenblum, "Round-Trip Software Engineering Using UML: From Architecture to Design and Back", Computer Science Department, Uuniversity of Southern California. USA, 2001.

[Metha00] N. Metha et al, "Towards a Taxonomy of Software Connectors", Proceedings of the ICSE 2000, ACM Press. Limerick, Irlanda. 2000.

[Mobility02] Mobility. Mobility Software. http://www.irobot.com/rwi/p10.asp.

[Molina01] JP. Molina, JA. Carrión, F. Ortiz, B. Alvarez, A. Iborra, J.M. Fernández. "GOYA: A Teleoperated System for Blasting Applied to Ships Maintenance", Consolidation of Technical Advances in the Protective & Marine Coatings Industry PCE 2001. Antwerp, Bélgica, Marzo 2001.

[Musliner93] D. Musliner, E. Durfee, K. Shin, "CIRCA: A cooperative intelligent real-time control architecture", IEEE Transactions on Systems, Man, and Cybernetics, 23 (6), 1993

[Nesnas01] I. Nesnas et al, "Toward Developing Reusable Software Components for Robotic Applications", Jet Propulsion Laboratory, California Institute of Technology, 2001.

[Nesnas03] I. Nesnas et al, "CLARAty: An Architecture for Reusable Robotic Software", Jet Propulsion Laboratory, NASA, Carnegie Mellon University, March 2003

[ObjectTime00] http://www.objectime.com

[OCERA03] Open Components for Embedded Real-Time Applications. Disponible en www.ocera.org. 2003

[OCL97] Rational Software Corporation and IBM, OCL specification. OMG document ad/97-8-08. Disponible en www.omg.org/docs/ad. 1997.

[OMAC00] www.omac.org

[OMG99] Object Management Group: Analysis and Design Platform Task Force. "White Paper on the Profile mechanism". Version 1.0. OMG Document ad/99-04-07. Disponible en www.omg.org. 1999.

[OMG00] Object Management Group: Unified Modeling Language Specification. Version 1.4 beta R1, Nov 2000, http://cgi.omg.org

[OMG-CORBA] OMG. CORBA/IIOP 2.2 specification. Disponible en www.omg.org/corba

[ONU04] "World Robotics 2004 - Statistics, Market Analysis, Forecast, Case Studies and Profitability", United Nations Economic Commission for Europe (UN/ECE). Sales and Marketing Section United Nations. Sales N° GV.E.04.0.20 o ISBN No. 92-1-101084-5. Ginebra, Suiza. 2004.

[OROCOS] Open Robot Control Software. Disponible en www.orocos.org

[Ortiz00] Ortiz F, Iborra A, Marín F, Álvarez B, Fdez Meroño J.M. "GOYA: A teleoperated system for blasting applied to ships maintenance". 3rd International

Conference on Climbing and Walking Robots. CLAWAR´2000. Octubre, 2000. ISBN 1-86058-268-0

[Ortiz02a] Ortiz F, Martínez A, Álvarez B, Navarro P, Iborra A, Fernández-Meroño JM., "Modelling a Software Architecture for Robots Control Using UML and COMET Architectural Design Method", 5ª Jornadas de Tiempo Real. Cartagena, España. Febrero 2002

[Ortiz02b] Ortiz F, Martínez AS, Álvarez B, Iborra A, Fernández JM, "Development of a Control System for Teleoperated Robots Using UML and Ada95", 7th Ada-Europe International Conference on Reliable Software Technologies, Viena, Austria. Junio 2002. ISBN 3-540-43784-3

[Ortiz03] Ortiz FJ, Álvarez B, Pastor JA, Sánchez P. "A Case Study in Performance Evaluation of Real-Time Teleoperation Software Architectures using UML-MAST", 8th Ada-Europe International Conference on Reliable Software Technologies. Lecture Notes on Computer Science – LNCS2361 Ed. Springer, ISBN 3-540-43784-3

[OSACA01] Open System Architecture for Controls within Automation Systems, http: //osaca.isbe.ch/osaca.

[Pautet98] Pautet, L., Tardieu, S. GLADE user´s guide. Technical report version 3.14a. ACT. 1998

[Pastor98] J.A. Pastor, b. Alvarez, A. Iborra, Meroño, " An underwater teleoperated vehicle for inspection and retrieving", In VRMech'98, 1st International Symposium-CLAWAR'98, Brussels, Nov-98

[Pastor02-td] Pastor J.A., "Evaluación y Desarrollo Incremental de una Arquitectura Software de Referencia para Sistemas de Teleoperación utilizando Métodos Formales", Tesis Doctoral, ETSIT, Universidad Politécnica de Cartagena, 2002.

[Pastor04] Pator JA, Álvarez B, Sánchez P, Ortiz F, "A Layered Architectural Component Model for Service Teleoperated Robots", Jornadas de Ingeniería del Software y Bases de Datos. Málaga, España. Noviembre 2004

[PC104] http://www.pc104.org

[Pérez02] P. Pérez, J.L. Posadas, J.E. Simó, G. Benet, F. Blanes, "A Software Framework for Mobile Robot Sensor Fusion and Teleoperation", DISCA, UPV, 15th Triennial World Congress IFAC, Barcelona, España. 2002.

[Pérez03] Pérez J., I. Ramos, J. Jaen, P. Letelier and E. Navarro. "PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures". 3rd IEEE International Conference on Quality Software, Dallas, Texas, USA, Nov. 2003.

[PFC00] A. Martínez, "Análisis y Desarrollo de un Protocolo de Comunicaciones para un Robot Teleoperado", Proyecto Fin de Carrera dirigido por B. Álvarez. ETSII, UPCT. 2000.

[PFC01a] R. J. Martínez, "Diseño del Control Manual y Automático del Sistema de Limpieza de Cascos de Buques GOYA", Proyecto Fin de Carrera dirigido por F.J. Ortiz. ETSII, UPCT. 2001.

[PFC01b] J. Alonso, "Desarrollo de la Interfaz de Usuario del robot teleoperado GOYA", Proyecto Fin de Carrera dirigido por F.J. Ortiz. ETSII, UPCT. 2001.

[PFC01c] J. Tobal, "Desarrollo de Interfaz de Usuario para teleoperación del robot GOYA sobre PDA", Proyecto Fin de Carrera dirigido por F.J. Ortiz. ETSII, UPCT. 2001.

[PFC02] F. Santoyo, "Implementación de una Arquitectura Software de Teleoperación para el Sistema Goya en Ada95", Proyecto Fin de Carrera dirigido por F.J. Ortiz. ETSII, UPCT. 2002.

[PFC04a] P. Magaña, "Desarrollo de controlador para robot implementado con FPGAs sobre plataforma PCI", Proyecto Fin de Carrera dirigido por F.J. Ortiz. ETSIT, UPCT. 2004.

[PFC04b] P.J. Jara, "Diseño de Sistema de Control para Plataforma Teleoperada de Limpieza de Barcos", Proyecto Fin de Carrera dirigido por F.J. Ortiz. ETSIT, UPCT. 2004.

[PFC04c] F. García, " Evaluación del rendimiento de arquitecturas software para sistemas de teleoperación utilizando UML-MAST", Proyecto Fin de Carrera dirigido por J.A. Pastor. ETSIT, UPCT. 2004.

[Popkin97] Popkin Software & Systems. "The User Guide for the System Architect Family of Tools". 1997.

[Posadas97] J.L. Posadas, J. Simó, F.Blanes, "Un modelo para el desarrollo de aplicaciones distribuidas. El Servidor de comunicaciones", actas de la Jornadas españolas de Automática, JA'97, Gerona 1997.

[Posadas02] J.L. Posadas, P. Pérez, J. Simó, G. Benet, F.Blanes, "Communications structure for sensory data in mobile robots", Engineering Applications of Artificial Intelligence 15, pp.341-350, Science Direct, 2002.

[Rational01] Rational Rose, "Using Ada for Forward and Reverse Engineering", Version 2001A.04.00.

[Rational02] Rational Software Corporation, www.rational.com

[RationalRose00] Rational ROSE'2000, "Manual of Rational Rose 2000".

[Redell98] O. Redell, "Modelling and Implementation Analysis of the Distributed Real-Time Control System for a Four Legged Vehicle", Mechatronics Lab, Department of Machine Design. Royal Institute of Technology. Stockholm, Suecia. Abril 1998.

[Requena03] Requena F, Ortiz FJ, Suardíaz J, Iborra A, "Adaptation of Real-Time Software Robot Control Units to Generic Hardware Architectures". Dedicated Systems e-Magazine (pub. Internet) (http://www.dedicated-systems.com/Magazine/emagazine/magazineform.aspx?year=2003&quarter=2). Sept. 2003. Bélgica.

[Roberts96] D. Roberts et al, "Evolving Frameworks. A pattern Language for Developing Object-Oriented Frameworks", Proceedings of the PloP-3, 1996

[ROVA01] Proyecto Fundación Séneca, Comunidad Autónoma de la Región de Murcia (UPCT). Arquitectura de referencia para unidades de control de sistemas teleoperados. 2002-2004.

[Rumbaugh91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, "Object-Oriented Modeling and Design", Englewood Cliffs, NJ: Prentice Hall, 1991

[Salzmann02] C. Salzmann, D. Gillet, "Real-Time Iteraction over the Internet", IFAC 15th Triennial World Congress, Barcelona, España. Julio 2002.

[Schilling00] T. Schilling, "Telerobotic Applications", professional Engineering Publishing, 2000.

[Schlegel99] C. Schlegel, R. Wörz "Interfacing Different Layers of a Multilayer Architecture for Sensorimotor Systems using the Object-Oriented Framework

SMARTSOFT", Research Institute for Applied Knowledge Processing – FAW. Ulm, Alemania. 1999.

[Schlegel01] C. Schlegel, "Software Engineering in Robotics", Research Institute for Applied Knowledge Processing – FAW. Ulm, Alemania. Junio 2001.

[Schlegel02] C. Schlegel, "Communication Patterns for OROCOS. Hints, Remarks, Specification", Research Institute for Applied Knowledge Processing – FAW. Ulm, Alemania. Febrero 2002.

[Schmidt-ACE] D. Schmidt. ACE – Adaptive Communication Environment. www.cs.wustl.edu/schmidt/ACE.html

[Schmuller00] J. Schmuller, "Aprendiendo UML en 24 horas", Pearson Education, México, 2000.

[Schneider94] S.A. Schneider, V.W. Chen, G. Pardo-Castellote, "Control Shell: A real-Time Software Framework", AIAA Conference on Intelligent Robots in Field, Factory, Service and Space, marzo 1994.

[Scholl01] K.U. Scholl, J. Albiez, B. Gassmann, "MCA – An Expandable Modular Controller Architecture", Facultad de Informática, Universidad de Karlsruhe, Alemania, 2001.

[SEI03] "How do you define Software Architecture?", http://www.sei.cmu.edu/architecture/definitions.html , Software Engineering Institute, Carnegie Mellon University, Pittsburgh. 2003.

[SEI03-TN6] "DoD Architecture Framework and Software Architecture Workshop Report", Technical Note CMU/SEI-2003-TN-006. Software Engineering Institute, Carnegie Mellon University, Pittsburgh. Marzo 2003.

[Selic94] B. Selic, G. Gullekson, P.T. Ward, "Real-Time Object-Oriented Modeling" (ROOM). John Wiley and Sons, New York. 1994.

[Selic96] B. Selic, G. Gullekson, "Design Patterns for Real-Time Software", ObjectTime Lmited. Embedded Systems Conference West'96.

[Selic98] B. Selic, J. Rumbaugh, "Using UML for Modeling Complex Real-Time Systems", Rational Whitepaper, 1998.

[Selic03] B. Selic, "Modeling Real-Time System Architectures with UML 2.0", Rational Software. 2003.

[Séneca02] "Evaluación y rediseño de una arquitectura software de referencia para sistemas de teleoperación en base a un modelo de componentes utilizando métodos formales". PB/5/FS/02. Fundación Séneca, 2002

[Shaw96] M. Shaw, "Some Patterns for Software Architecture", en Jj. Vlissides, ete al, Pattern Languages of Program Design, 2, pp. 255-269. Reading, MA. Addison-Wesley, 1996

[Shaw01] M. Shaw, "The Coming-of-Age of Software Architecture Research", Institute for Software Research, Carnegie Mellon University, Pittsburgh, USA. 2001

[Sheridan92] T.B. Sheridan. "Telerobotics, Automation and Human Supervisory Control". MIT Press. Massachusetts, 1992.

[Shlaer88] S.Shlaer, J. Mellor, J. Stephen, "Object Oriented Analysis-Modelling, the Word in Date", Englewood Cliffs, Prentice-Hall, 1988

[SIMATIC02] SIMATIC - Working with STEP 7 5.2. ref. 6ES7810-4CA06-8BA0. www.siemens.com. 2002

[Simon95] D. Simon, B. Espiau, E. Castillo, K. Kapellos, "Computer-Aided Design of a Generic Robot Controller Handling Reactivity and Real-Time Control Issues", INRIA – ISIA, Francia. Agosto 1995.

[Simmons94] R. Simmons, "Structured control for autonomous robots". IEEE Transactions on Robotics and Automation, 10, feb 1994

[Smolander01] K. Smolander et al, "Required and Optional Viewpoints. What is included in Software Architecture", Telecom Business Research Center, Lappeeranta University of Technology, http://www.lut.fi/TBRC, 2001.

[Sowa92] J.F. Sowa and J.A. Zachman, "Extending and formalizing the framework for information systems architecture". IBM Systems Journal, 31(3):590--616, 1992

[Staunstrup97] J. Staunstrup, W. Wolf, "Hardware/Software Co-Design: Principles and Practice", Kluwer Academic Publishers, ISBN 0-7923-8013-4. Holanda 1997.

[Stewart97] D. Stewart, R. Volpe, P. Khosla, "Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects", IEEE Transactions on Software Engineering, Vol.23, No.12. Diciembre 1997.

[Thomesse02] J.P. Thomesse, "Open Issues in Fieldbus based Systems", IFAC 15th Triennial World Congress, Barcelona, España. Julio 2002.

[Trigaux03] J.C. Trigaux, P. Heymans, "Modelling variability requirements in Software Product Lines: A comparative survey", Technical report PLENTY project, Institut d'Informatique FUNDP, Namur, Bélgica. Noviembre 2003.

[TRON96] PROYECTO EUREKA-TRON (EU-1565), Expediente MINER ATYCA 163/96.

[UML99a] OMG, "Unified Modeling Language Specification", Version 1.3., Junio 1999.

[UML99b] Booch, Rumbaugh, Jacobson, "The UML Modeling Language User Guide", Addison-Wesley, 1999.

[Vertut85] J. Vertut, P. Coiffet, "Teleoperation and Robotics. Evolution and Development", Kogan Page, Londres, 1985.

[VHDL93] IEEE Standard VHDL Language Reference Manual. Std 1076-1993, 1993.

[Vidal02] J. Vidal, P. Mendoza, J. Vila, A. Crespo, S. Sáez, "A Minimal RT-Linux Embedded System for Control Applications", 15th IFAC Triennal World Congress, Barcelona, España. 2002.

[Warmer98] J.B. Warmer, A.G. Kleppe. "The Object Constraint Language: Precise Modeling with UML". Addison-Wesley, 1998.

[Yourdon 79] Yourdon, Ed and Larry L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. (1979), Prentice Hall

[Zamorano02] J. Zamorano, J.F. Ruíz, "GNAT/ORK: An Open Cross-Development Environment for Embedded Ravenscar – Ada software", 15th IFAC Triennal World Congress, Barcelona, España. 2002.