Revista Iberoamericana de Automática e Informática Industrial
CEA IFAC comité español de automática
RIAI
ISSN: 1697-7912. Vol. 3, Núm. 2, Abril 2006, pp. 79-89
http://riai.isa.upv.es

# ARQUITECTURA PARA CONTROL DE ROBOTS DE SERVICIO TELEOPERADOS

**Bárbara Álvarez, Francisco Ortiz, Juan A Pastor, Pedro Sánchez, Fernando Losilla, Noelia Ortega***

*División de Sistemas e Ingeniería Electrónica (DSIE)*
*Universidad Politécnica de Cartagena*
*e-mail: balvarez@upct.es*

**Centro Tecnológico Naval y del Mar (Cartagena)*
*e-mail: nortega@ctnaval.com*

Resumen: La teleoperación de robots es utilizada para realizar tareas que un operador humano no puede llevar a cabo, bien por la complejidad de las mismas o bien por la necesidad de realizarlas en entornos hostiles. Actualmente, se pueden encontrar en la literatura muchas arquitecturas de control para trabajar con este tipo de sistemas; sin embargo, debido a la gran variabilidad en el comportamiento de los mismos, ninguna llega a cubrir todos los objetivos que se plantea inicialmente. El propósito de este artículo es presentar un nuevo marco arquitectónico (ACROSET) que tiene en cuenta las últimas tendencias en arquitecturas para el control de robots adoptando una aproximación orientada a componentes. Tal aproximación provee un marco común para el desarrollo de sistemas con comportamientos muy diferentes y permite la integración de componentes inteligentes. La arquitectura está siendo implementada en el proyecto EFTCoR para el desarrollo de una familia de robots (grúas y vehículos) que realizan labores de mantenimiento en la superficie de cascos de buques. 

Palabras Clave: Arquitectura software, teleoperación, sistemas de tiempo real.

## 1. INTRODUCCIÓN

El propósito de este artículo es presentar una arquitectura de referencia para aplicaciones de control de robots de servicio (ACROSET). Dichas aplicaciones se emplean para teleoperar manipuladores, vehículos y herramientas que realizan operaciones de inspección y mantenimiento en entornos hostiles. En general, tales actividades son complejas y no es posible trabajar con sistemas completamente autónomos. Por ello, un operador se encarga de monitorizar y operar los diferentes mecanismos. El sistema recibe órdenes del mismo y lleva a cabo las acciones correspondientes para ejecutarlas.

Los sistemas de teleoperación cubren un amplio rango de mecanismos y misiones con características y requisitos muy específicos, pero a la vez comparten muchas características comunes que permiten describir un dominio de aplicaciones y su correspondiente arquitectura de referencia. Ello ha permitido al grupo de investigación DSIE de la Universidad Politécnica de Cartagena reutilizar durante los últimos años una arquitectura de referencia para el desarrollo de dispositivos teleoperados para la industria nuclear (Álvarez, *et al*., 2001). Estos dispositivos, a pesar de sus diferencias, compartían características comunes desde el punto de vista de su control, y por tanto fue relativamente fácil reutilizar la misma arquitectura para su desarrollo. Así:

- el entorno de trabajo era estructurado (perfectamente conocido).
- el comportamiento era deliberativo (dirigido por el operador), estando el

[1] Este trabajo está parcialmente financiado por la CICYT (TIC2003-07804-C05-02) y la Fundación Séneca (PB/5/FS/02).

comportamiento reactivo (autónomo) limitado a determinadas acciones de seguridad, y
- cada aplicación controlaba un único sistema.

Sin embargo, ninguna de las características mencionadas se puede aplicar a los sistemas del proyecto EFTCoR (Growth, 2001) que comprenden una familia de robots cuya misión es básicamente la limpieza de cascos de buques respetando el medio ambiente. En este caso:

- los entornos de trabajo no son estructurados debido a la gran variabilidad entre las superficies de los cascos de buques y entre las diferentes áreas a tratar.
- estos sistemas deben tener un elevado grado de autonomía, y
- es posible que diferentes sistemas tengan que trabajar de manera cooperativa.

Debido a estas nuevas características, la arquitectura original no puede reutilizarse para los robots EFTCoR. Sin embargo, como el uso de una arquitectura común para todos los desarrollos es extremadamente útil, ya que reduce costes y tiempo de desarrollo, el grupo de investigación DSIE se puso a trabajar inmediatamente en una nueva arquitectura (ACROSET) que se presenta en este trabajo. A continuación, se hace una breve descripción del sistema EFTCoR, se justifica la aproximación metodológica que se ha empleado, se describen los límites del dominio considerado y las principales directrices para la definición de la arquitectura, y se hace una descripción de la misma. Por último, se recogen las conclusiones a día de hoy y los trabajos futuros.

## 2. EL SISTEMA EFTCoR

El sistema EFTCoR es una plataforma de teleoperación para la realización de operaciones de mantenimiento de cascos de buques respetuosas con el medio ambiente. Para alcanzar la funcionalidad requerida, se han identificado los siguientes subsistemas que trabajan de forma cooperativa:

- Sistema de tratamiento y reciclado de residuos.
- Familia de sistemas robotizados especializados y de bajo coste que soportan y posicionan los cabezales de limpieza.
- Sistema de control para operar y coordinar los diferentes subsistemas.
- Sistema de visión para detectar y clasificar defectos y supervisar la calidad del acabado superficial.
- Sistema de monitorización para la planificación y seguimiento global de los trabajos de mantenimiento.

La Universidad Politécnica de Cartagena (UPCT) es responsable del desarrollo o adaptación de dispositivos robóticos teleoperados para el posicionamiento de los cabezales de limpieza a lo largo del casco del buque, en concreto del:

- Desarrollo de los dispositivos robotizados para el posicionamiento de los cabezales de limpieza.
- Desarrollo de la unidad o unidades de control de los dispositivos robotizados.
- La integración de dicha unidad de control con el resto de subsistemas del EFTCoR.

Centrándonos en el segundo punto, es de gran interés la obtención de un marco arquitectónico que permita la reutilización y facilite el desarrollo de las aplicaciones.

## 3. ENFOQUE METODOLÓGICO DE LA ARQUITECTURA

se han descrito muchas arquitecturas para control de robots (Coste-Manière and Simmons, 2000), es difícil encontrar ejemplos de los procesos de desarrollo que se han seguido para definirlas. Por otro lado, los métodos de desarrollo basados en casos de uso, principalmente RUP[2] (Jacobson, *et al*., 1999) y otros similares aunque son apropiados para definir la arquitectura de un determinado sistema, no son adecuados para definir arquitecturas de referencia. Los casos de uso definen una funcionalidad concreta y para diseñar una arquitectura de referencia hay que buscar la generalidad, puesto que el éxito o fracaso de dicha arquitectura depende de su capacidad para englobar la variabilidad existente entre los sistemas del dominio considerado. En este sentido, se puede decir que los casos de uso que son relevantes para un sistema dado pueden no serlo para otros. Más aún, en el nivel de abstracción requerido para manejar la variabilidad entre sistemas, los casos de uso concretos no pueden ser definidos de manera apropiada.

Por todo ello, en este trabajo se ha seguido la metodología ABD *(Architecture Based Design)* (Bachmann, et al., 2001) propuesta por el SEI *(Software Engineering Institute)* de la Universidad Carnegie Mellon, completándola con las 4 vistas de

[2] *Rational Unified Process*

Hofmeister (Hofmeister, et al., 2000) y su notación basada en UML (Booch, *et al*., 1997) para componentes. El propósito de dicha metodología es diseñar arquitecturas software para un dominio de aplicación o familia de productos y se basa en:

- Una descomposición funcional de acuerdo con los principios de alta cohesión y bajo acoplamiento y el conocimiento del dominio de aplicación.
- La realización de los requisitos funcionales y de calidad haciendo uso de patrones adecuados.
- La utilización de plantillas software para definir elementos y responsabilidades comunes a un grupo de componentes así como sus interacciones con la infraestructura.

La metodología ABD propone la descomposición del sistema en subsistemas recursivamente, de forma que las mismas reglas que se emplean para descomponer el sistema en subsistemas se pueden emplear para descomponer dichos subsistemas en otros más simples.

Como modelo final se obtiene una vista conceptual de la arquitectura en la que quedan identificados los principales subsistemas y las relaciones entre ellos, que son descritas utilizando patrones de diseño.

Hofmeister (Hofmeister, et al., 2000) propone otro método de desarrollo orientado a arquitecturas que puede superponerse a ABD en las primeras etapas del desarrollo. Dicha aproximación es interesante ya que propone el uso de puertos y conectores entre componentes usando una notación inspirada en ROOM (Selic, et al., 1994). Para ello, se ha extendido UML con clases estereotipadas y símbolos especiales para expresar tales componentes, puertos y conectores. Se puede destacar que el método de Hofmeister facilita la conexión entre los componentes conceptuales y sus implementaciones.

## 4. CARACTERIZACIÓN DEL DOMINIO: ROBOTS DE SERVICIO TELEOPERADOS

Los robots de servicio son sistemas mecatrónicos generalmente diseñados para una aplicación concreta. Sin embargo, a pesar de las diferencias en su estructura física, comparten componentes comunes tanto lógicos como físicos. La caracterización del dominio de aplicación es el punto de partida para definir requisitos funcionales y de calidad.

A partir de nuestra experiencia las principales características que deben ser consideradas son las siguientes:

- El alto grado de especialización y por lo tanto la gran variabilidad en cuanto a funcionalidad y estructura física.
- Las diferentes combinaciones de vehículos, manipuladores y herramientas.
- La gran variedad de infraestructuras de ejecución incluyendo procesadores, enlaces de comunicación e interfaces hombre-máquina.
- La gran variedad de sensores y actuadores.
- Los diferentes tipos de algoritmos de control: desde algoritmos para realizar actuaciones muy simples a otros extremadamente complejos que hacen uso de estrategias de navegación.
- Los diferentes grados de autonomía, desde sistemas dirigidos completamente por el operador a robots semi-autónomos.
- La presencia de requisitos de tiempo real.
- Las implementaciones pueden ser intensivas en hardware o en software con infinidad de posibilidades intermedias.
- Por último, la seguridad de estos sistemas es un tema que no se puede obviar.

Un análisis más preciso de las diferencias entre sistemas (Pastor, 2002) revela que la mayoría de estas diferencias afectan no tanto a los componentes del sistema sino a las interacciones entre ellos. De acuerdo con los puntos anteriores, podríamos considerar que entre las **directrices arquitectónicas**, aquellas que tratan de la variabilidad o modificabilidad, serán las más influyentes en el diseño de la arquitectura:

1. Diferentes instanciaciones de la arquitectura deben poder compartir y reutilizar los mismos componentes.
2. Se deben adoptar políticas que permitan una clara separación entre dichos componentes y sus patrones de interacción.
3. La implementación de los componentes de la arquitectura podrá ser hardware o software, pudiéndose tratar de componentes COTS[3]. La implementación podrá realizarse sobre diversas plataformas, incluso distribuidas.
4. Debe ser posible que los sistemas realicen tanto acciones autónomas (principalmente reactivas) como teleoperadas (dictadas por el operador) o misiones pre-programadas.

## 5. UN RECORRIDO POR LA ARQUITECTURA

Como indica la primera directriz arquitectónica, debería ser posible que diferentes sistemas compartan los mismos componentes cumpliendo así el importante requisito de reutilización. Por ello, además de utilizar componentes, puertos y conectores, se deberán definir las reglas e infraestructura común que permita que dichos

[3] Componentes comerciales

componentes sean ensamblados o conectados de diferentes formas, cambiando si es preciso sus esquemas de interacción, como define la segunda directriz. Un puerto proporciona una manera regular de intercambiar datos y control entre componentes, independientemente de su funcionalidad y granularidad, proporcionan una puerta a los servicios que los componentes ofrecen y requieren.

La conexión entre puertos (y en 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 dichos patrones 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. Los puertos tienen una relación de composición con los componentes. Dicha relación descrita por Gamma[4] (1995) permite tratar los componentes complejos de la misma forma que un componente simple, ocultando su complejidad interna.

Una vez que se ha definido cómo son los componentes tienen que ser ensamblados, el siguiente paso es definir qué componentes tienen que aparecer. La tercera directriz arquitectónica identificada en el apartado anterior establece que los componentes pueden ser implementados en hardware o software, siendo preferible que tales componentes sean COTS. Para alcanzar este objetivo, será necesario identificar los componentes típicos de este tipo de sistemas, los cuales pueden definirse a distintos niveles de granularidad. Al nivel más bajo, un robot tendrá como componentes hardware fundamentales: sensores y actuadores. El siguiente nivel lo marcarían los controladores de dichos actuadores que tienen asociados a su vez algunos sensores (por ejemplo, un servosistema, como un motor controlado, necesita al menos la realimentación de un sensor). En el siguiente nivel se definen los controladores para grupos de actuadores (por ejemplo, una tarjeta capaz de controlar el movimiento de las articulaciones de un mecanismo), y así sucesivamente.

Muchos de estos componentes pueden ser adquiridos en el mercado: dispositivos hardware, tarjetas de control, paquetes software para una determinada plataforma, etc. Para facilitar el uso de componentes COTS, la arquitectura define sus interfaces típicas. El enlace entre la interfaz definida en la arquitectura y el componente COTS se lleva a cabo utilizando *Bridge pattern* (Gamma, *et al*., 1995). A la hora de definir componentes, la arquitectura identifica cuatro niveles de granularidad y adopta la idea de nivel de abstracción del hardware descrita en el marco OROCOS (Bruyninckx, *et al*., 2002). Estos niveles modelan las características de los componentes físicos del sistema, definiendo sensores, actuadores, controladores de movimiento, etc, haciendo posible la definición de bibliotecas de componentes y el intercambio de implementaciones hardware/ software con un impacto mínimo.

Finalmente, la última directriz arquitectónica era la posibilidad de obtener arquitecturas concretas para sistemas con predominio de comportamiento tanto deliberativo como reactivo. Para ello, es necesario separar los dos comportamientos como se muestra en la figura 1. Este esquema coincide con el empleado por la arquitectura CLARAty (*Coupled Layered Architecture for Robotic Autonomy*) (Nesnas, et al., 2003). CLARAty distingue entre un Nivel Funcional y un Nivel de Decisión que encapsula los subsistemas responsables de la planificación y ejecución de las misiones. Sin embargo, la arquitectura que se propone en este trabajo separa estas ideas de forma diferente. Como en el caso de CLARAty, los niveles más altos de inteligencia pueden acceder a los componentes del nivel más bajo, siendo la inteligencia un cliente de la funcionalidad. Pero, a diferencia de CLARAty donde los comportamientos autónomos pueden ser añadidos al nivel funcional, en nuestro modelo la inteligencia del sistema está completamente separada del resto de la funcionalidad.

### *5.1 Niveles y componentes.*

La arquitectura propuesta (ACROSET) identifica cuatro niveles de diferente granularidad en los que definir los componentes que forman parte de lo que se denomina subsistema *CCAS* (Coordinación, control y abstracción de dispositivos):

- Nivel 1: Características abstractas de componentes elementales, tales como sensores y actuadores.
- Nivel 2: Controlador de Unidad de Dispositivo Simple, SUCs *(Simple Unit Controllers)*
- Nivel 3: Controlador de Unidad de Mecanismo, MUCs *(Mechanism Controllers)*
- Nivel 4: Controlador de Unidad de Robot, RUCs *(Robot controllers)*

Como ya se ha mencionado, estos niveles se denominan de abstracción del hardware, ya que sus componentes podrían ser implementados tanto en hardware como en software. Los componentes más simples modelados por la arquitectura son los sensores y actuadores, que son definidos en el nivel arquitectónico más bajo. Un sensor es un componente que provee la información requerida para controlar un elemento activo, como por ejemplo un *encoder* o un *fin de carrera* asociado a una articulación. Los actuadores modelan los elementos activos más simples, por ejemplo un motor.

[4] *Composite Pattern*.

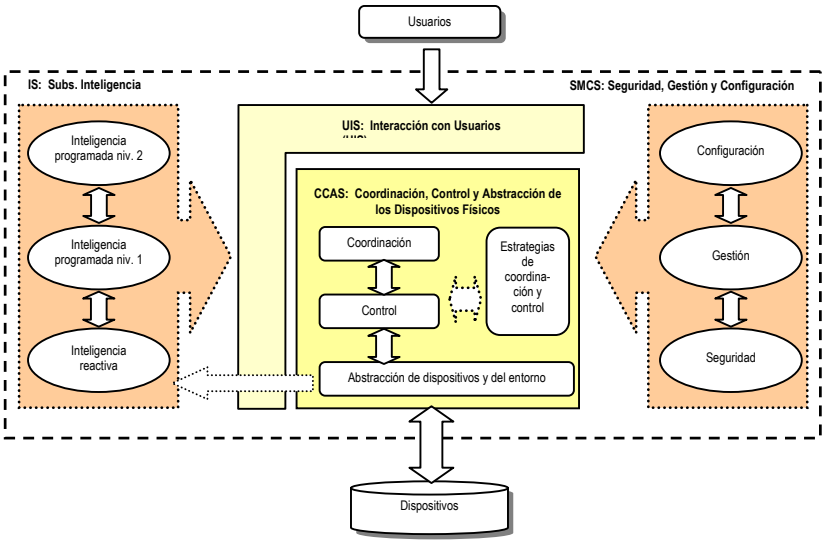


Figura 1. Descripción de alto nivel de la arquitectura

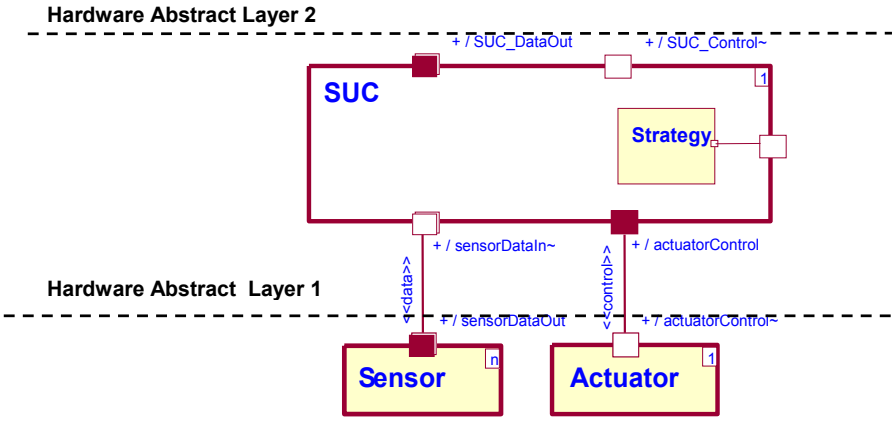


Figura 2. SUC (Simple Unit Controller)

Los SUCs *(Simple Unit Controllers)* son los componentes definidos en el segundo nivel de la arquitectura, y modelan el control sobre los actuadores a partir de la información sensorial. Pueden definirse por ejemplo para controlar las articulaciones de un mecanismo dado. Como puede verse en la figura 2, un SUC genera las órdenes para un actuador a partir de la orden que recibe de otro componente (a través del puerto ***controllerControl***), la información sensorial que describe el estado del actuador, y su propia política de control, la cual puede ser modificada. Así, la estrategia de control de una determinada articulación podría ser un tradicional PID y ser cambiada, por ejemplo, por una estrategia de lógica borrosa. Un SUC necesita satisfacer requisitos de tiempo críticos, por lo que generalmente estará implementado en hardware. Cuando se implementan en software imponen severos requisitos de tiempo real que repercuten sobre el sistema operativo y la plataforma de ejecución.

En el tercer nivel de granularidad están los componentes MUC *(Mechanism Unit Controller)*, que modelan el control sobre un mecanismo global (vehículo, manipulador o herramienta). Como muestra la figura 3, un MUC es una entidad lógica responsable de coordinar los SUCs, de acuerdo a la información recibida y a su propia estrategia de coordinación. Al igual que en el caso anterior, esta estrategia puede ser modificada, por ejemplo, para un manipulador puede ser una solución particular de su cinemática inversa, para un vehículo ser una estrategia de navegación, etc. Aunque la arquitectura define los MUCs como agregados relacionales, pueden llegar a ser componentes inclusivos cuando la arquitectura se instancia para desarrollar un sistema concreto. En otras palabras, la arquitectura define componentes y subsistemas lógicos, con una relación débil entre las partes y el todo. Las instanciaciones concretas pueden imponer que el todo y las partes no puedan existir por separado. De la misma manera, es una decisión de la instaciación concreta de la arquitectura si las interfaces de los SUCs internos son accesibles o no. De hecho, aunque los MUCs podrían ser implementados en hardware o software, es muy usual que sean tarjetas de control de movimiento comerciales, lo que limita la posibilidad de comandos a los componentes internos. Se puede

decir que los COTS limitan la flexibilidad del enfoque, en el sentido de que no siempre proveen acceso directo a sus sub-componentes ni a su estado interno.

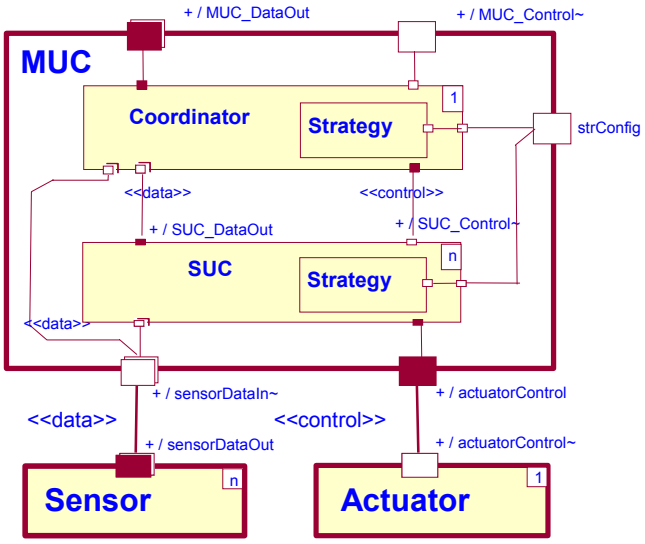


Figura 3. MUC (*Mechanism Unit Controller*)

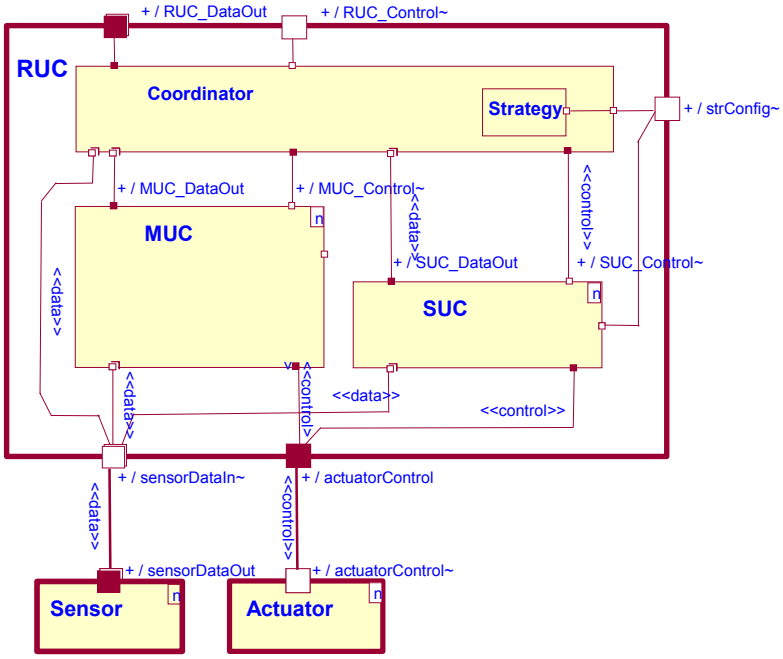


Figura 4. *RUC (Robot Unit Controller)*

Finalmente, la arquitectura define el componente RUC (*Robot Unit Controller*) en el cuarto nivel, para modelar el control de un robot completo, por ejemplo, un robot compuesto por un vehículo, un brazo y varias herramientas intercambiables. Como se muestra en la figura 4, un RUC es una agregación de MUCs con un coordinador global que genera las órdenes para éstos y coordina sus acciones a partir de la información recibida y de su propia estrategia de coordinación. Dicha estrategia, como en los niveles anteriores es intercambiable. Por ejemplo, si el robot comprende un vehículo con un manipulador podría ser una solución cinemática generalizada para tener en cuenta el movimiento del vehículo. Al igual que los MUCs, los RUCs son componentes lógicos que podrían ser componentes físicos dependiendo de la instancia para un sistema concreto. En general, un RUC es un componente complejo que comprende componentes hardware y software y puede ofrecer una gran variedad de interfaces, dependiendo de la complejidad del sistema controlado.

Habiendo definido los componentes SUCs, MUCs y RUCs, parece lógico definir un controlador de grupo GUC (*Group Unit Controller*) capaz de coordinar un grupo de robots que cooperen para realizar un determinada tarea. Sin embargo, dicho componente no queda reflejado en la arquitectura, por la siguiente razón: la inteligencia requerida

para el control de una articulación o un mecanismo suele ser bien conocida y puede estar encapsulada en componentes reutilizables, mientras que la requerida para trabajar cooperativamente necesita de una aproximación más flexible. Un componente capaz de ofrecer inteligencia o tomar sus propias decisiones tiene definida su propia arquitectura (por ejemplo, un sistema de visión capaz de determinar caminos libres de obstáculos). Por tanto, se trata de no imponer una estructura a los componentes *inteligentes*, sino de proporcionar una forma de integrarlos.

*5.2 Añadiendo el comportamiento autónomo.*

La composición de SUCs y MUCs permite obtener una arquitectura jerárquica en la que las decisiones fluyen de arriba hacia abajo y la información de abajo hacia arriba. Esta estructura encaja bien con los sistemas dirigidos por el operador, donde el comportamiento autónomo no existe o se limita a acciones de seguridad hardware. También encaja bien con sistemas donde el comportamiento reactivo o autónomo responde a reglas simples que pueden ser añadidas a los controladores o coordinadores que, siguiendo estas reglas, pueden tomar decisiones y notificárselas al controlador o coordinador de nivel superior. Sin embargo, hay sistemas en los que el comportamiento autónomo no es tan simple. En estos casos, el componente inteligente necesita integrar más información y acceder a más funcionalidad que la embebida en un componente dado.

La aproximación en ese caso (ver figura 5) es superponer el comportamiento autónomo inteligente y el comportamiento dirigido por el operador, proveyendo los medios necesarios para la integración de ambos y la resolución de conflictos. Esta aproximación no implica ningún cambio en los componentes definidos hasta ahora sino nuevas fuentes de comandos para ellos. Dichas fuentes están constituidas por nuevos componentes que tienen acceso al sistema global de información y son capaces de decidir qué hacer de acuerdo a algunas reglas programadas, algoritmos o heurísticos.

Cada componente de un nivel dado puede acceder a la información y a los puertos de control de componentes de niveles más bajos. Desde este punto de vista, cada componente de un nivel dado es un componente inteligente para el nivel más bajo. Por ejemplo, desde el punto de vista de un MUC, no hay distinción si los comandos vienen del coordinador del RUC que lo engloba (ver figura 5), desde el operador o desde algún componente inteligente definido sobre los RUCs. Dado que un componente puede recibir órdenes de más de una fuente, es necesario decidir qué orden ejecutar. La lógica para esta decisión está fuera del componente. La figura 5 muestra un nuevo tipo de componente: el árbitro. Los árbitros encapsulan las reglas que determinan qué orden debería ser enviada a un componente dado. El árbitro se define separadamente, ya que las reglas que encapsula o incluso el propio árbitro podrían cambiar de un sistema a otro o durante la vida de uno en diferentes etapas de su funcionamiento. El concepto de árbitro deriva de la noción de *composition filter* (Bergmans, 1994) y está fuertemente ligada a la necesidad de separar funcionalmente los patrones de interacción entre componentes.

Esta aproximación es altamente flexible y hace posible integrar inteligencia que no esté directamente relacionada con las misiones de dispositivos robóticas, sino con la gestión de la aplicación como las políticas de tolerancia de fallos, o un meta-nivel para reconfigurar la aplicación.

*5.3 Interacción con los usuarios.*

El subsistema de Interacción con los Usuarios (*UIS*) (figura 1) hace de interfaz entre el *CCAS* (Coordinación, control y representación de los dispositivos físicos) 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 también se contempla como un usuario 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 subsistema de Interacción con los Usuarios (*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)*, como se muestra en la figura 6, 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. La estrategia de arbitraje dependerá del modo de control que esté activo en cada momento.

*5.4 Seguridad, gestión y configuración.*

Entre los requisitos del dominio se define la seguridad como un factor importante. En la visión

general de la arquitectura mostrada en la figura 1 se plantea un subsistema que engloba tres componentes con funcionalidades relativas a seguridad, configuración y gestión respectivamente. El primero será un componente que monitoriza y diagnostica el correcto funcionamiento del sistema. Esta opción arquitectónica parece acercar la adopción del mecanismo "*Ejecutivo de seguridad*", pero opcionalmente en los subsistemas *IS* y *CCAS* se podrán añadir *watchdogs*, cuyas señales sean recogidas por el ejecutivo de seguridad.

El segundo componente será el encargado de gestionar la configuración del resto de los componentes del sistema, y por último un gestor almacenará el diagrama de estado de la aplicación completa y gestionará la viabilidad para realizar ciertas operaciones. Además, será el encargado de gestionar el arranque del sistema, creando el resto de componentes según el orden adecuado.

*5.5 Comportamiento temporal.*

Aunque las directrices arquitecturales puedan señalar como requisito prioritario la variabilidad o modificabilidad, las necesidades de tiempo real, sobre todo las críticas, deben quedar satisfechas. Sin embargo, ACROSET no se puede analizar temporalmente como arquitectura de referencia porque no sugiere un modelo de tareas fijo, sino que ofrece un modelo de componentes que puede instanciarse de muy diversas formas y según distintas combinaciones hardware/software. La elección del 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 instancia 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. De hecho ACROSET se ha diseñado para favorecer dicha posibilidad de variación.

Se trata de una arquitectura lo suficientemente flexible para implementar distintas estrategias y algoritmos de planificación. El particionado de la arquitectura en componentes distintos según sean algoritmos o estrategias, controladores, coordinadores, etc., es fundamental para alcanzar dicha flexibilidad, y debe hacerse teniendo en cuenta qué componentes participarán de un procesamiento en *tiempo real*, y cuáles no lo harán.

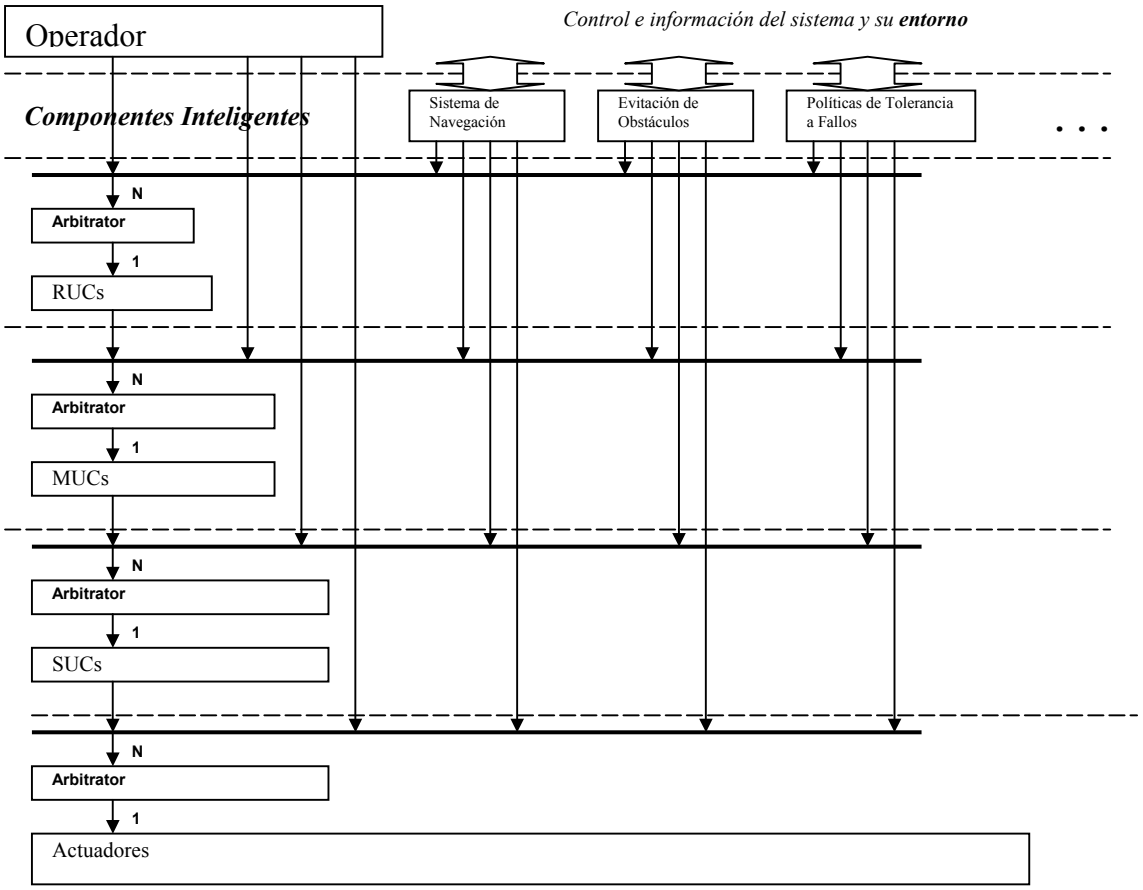


Figura 5. Superposición de comportamientos autónomos sobre los dirigidos por el operador

Normalmente, en el subsistema *CCAS* (Coordinación, Control y Abstracción de Dispositivos) habrá componentes con un comportamiento de tiempo real asociado, puesto que dicho comportamiento estará relacionado con lazos de control de movimiento. Sin embargo, los procesamientos relacionados con la planificación de misiones, generalmente no tendrán necesidades de tiempo real estrictas salvo en el caso de comportamientos reactivos que afecten a la seguridad, de cuya reacción *a tiempo* pueda depender la integridad de sistemas y personas. En todo caso, tal como se ha dividido la arquitectura, se puede diferenciar entre componentes con necesidades de tiempo real críticos frente a los acríticos de manera flexible. Incluso si fuera necesario se tomaría la estrategia de subdividir un componente en parte crítica y acrítica.

A modo de ejemplo, en (Ortiz, *et al*., 2003) se pueden consultar varios ejemplos de análisis temporal basado en *Rate Monotonic Analysis* RMA (Klein, *et al*., 1994) utilizando la extensión para Racional Rose, UML-MAST (Drake, *et al*., 2000).

Dicho análisis se realiza sobre un modelo de tareas concreto cambiando los patrones de interacción entre las mismas para la implementación que se hizo de ACROSET en el sistema GOYA (1998) robot para la limpieza de cascos de buques (figura 7).

## 6. RESUMEN Y TRABAJOS FUTUROS

La arquitectura descrita en este artículo recoge los avances más prometedores en el dominio de la teleoperación combinados con la aproximación orientada a componentes. Esta aproximación está enfocada a la definición de un marco de componentes comunes que permite la definición de componentes que pueden ser reutilizados en diferentes sistemas e integrados en sistemas inteligentes capaces de conducir el comportamiento del robot. Dicho trabajo se ha basado fundamentalmente en algunas ideas obtenidas de OROCOS (Bruyninckx, *et al*., 2002), CLARAty (Nesnas, *et al*., 2003) (arquitecturas robóticas) y la aproximación PRISMA (Pérez, *et al*., 2003) (aproximación orientada a aspectos y componentes).

La arquitectura está siendo utilizada actualmente para el control de una familia de robots cuya misión es fundamentalmente la limpieza de cascos de buques. Las figuras 7, 8, 9 y 10 muestran algunos de estos dispositivos robóticos desarrollados en la Universidad Politécnica de Cartagena en colaboración con Izar Carenas. En concreto, la figura 7 muestra un primer prototipo (sistema GOYA) desarrollado para la limpieza de superficies verticales, consistente en una plataforma elevadora, que constituye el sistema de posicionamiento primario, y un sistema de posicionamiento secundario encargado de aproximar el cabezal de limpieza al área a tratar.
La figura 8 muestra un segundo prototipo para la limpieza de dichas superficies. En este caso se utiliza como sistema de posicionamiento primario una grúa *cherry-picker*, la cuál posiciona una mesa *x-y* encargada de situar el cabezal de limpieza en el punto de trabajo. Dicho 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.

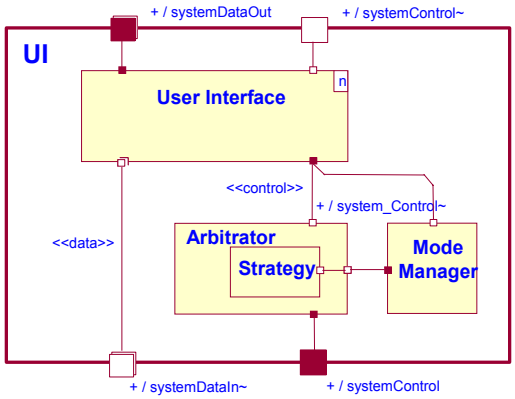


Figura 6. Subsistema de Interacción con Usuarios.

Las soluciones mostradas en las figuras 9 y 10 son más flexibles y permiten el acceso a los finos y fondos del casco del buque. Se muestra un primer y segundo prototipo respectivamente. Ambos vehículos transportan sobre la superficie del barco un cabezal de chorreado. Como en el caso anterior 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.

Esta familia de robots al presentar una amplia variedad de comportamientos y grados de complejidad es un excelente banco de pruebas para la arquitectura. Nuestra experiencia en el uso de la misma ha sido satisfactoria, si bien requiere de mejor soporte para expresar las abstracciones de los componentes, así como de técnicas reconocidas para cubrir la variabilidad de componentes entre instancias. Estos desafíos los cubre el enfoque y lenguaje PRISMA (Pérez, *et al.,* 2003) que va a ser usado para definir los componentes, y los cambios entre dichos componentes y la propia evolución de los componentes a partir de sus niveles meta y base.

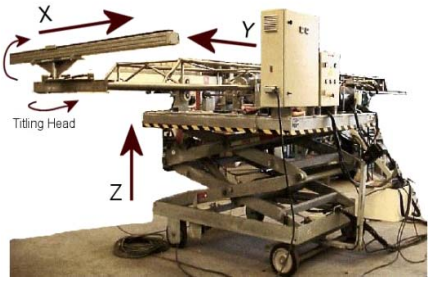


Figura 7. Plataforma elevadora

Figura 8. Cherry-picker

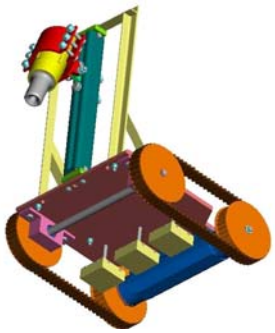

Figura 9. Vehículo Lázaro 1.

Figura 10. Vehículo Lázaro 2

REFERENCIAS

Álvarez, B., A. Iborra, A. Alonso, J.A. De la Puente (2001). Reference Architecture for Robot Teleoperation: Development Details and Practical Use. *Control Engineering Practice*, (9), 4, 395-402.

Bachmann F., L. Bass et al. The Architecture Based Design Method. *Technical Report Carnegie Mellon University*, (USA) CMU/SEI-200-TR-001.

Bergmans L. (1994). Composing Concurrent Objects, Ph.D. thesis, University of Twente, The Netherlands.

Booch, G., J. Rumbaugh, I. Jacobson (1997). Unified Modeling Language User Guide. *Addison-Wesley*, 1997.

Bruyninckx H., B. Koninckx, P. Soetens (2002). A Software Framework for Adavanced Motion Control. Dpto. of Mechanical Engineering, K.U. Leuven *OROCOS project inside EURON*, Bélgica.

Coste-Manière, E. and R. Simmons (2000). Architecture, the Backbone of Robotic System. *Proc. Of the 2000 IEEE International Conference on Robotics & Automation*, San Francisco.

Drake J.M., M. González Harbour, J.S. Medina (2000). 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.

Hofmeister C., R. Nord, D. Soni (2000). Applied Software Architecture. *Addison-Wesley*, ISBN 0-201-3257169-2.

Jacobson I., G. Booch, J. Rumbaugh (1999), The Unified Software Development Process. *Addison-Wesley*, ISBN 0-201-57169-2.

Gamma E., R. Helm, R. Johnson, J. Vlissides (1995). Design Patterns: Elements of Reusable Object Oriented Software. *Addison-Wesley, Reading Mass.*

GOYA (1998). Robot escalador para la limpieza de cascos de buques, respetuoso con el medio ambiente. Proyecto Feder (UPCT, Izar Carenas). FEDER IFD97-0823 (TAP).

Growth (2001). Environmentally Friendly and Cost-Effective techmology for Coating Renoval (EFTCoR). *Fifth Framework Programme, Growth* (GRD2-2001-50004).

Klein M.H., *et al*. (1994). A Practitioner´s Handbook for Real-Time Analysis Guide to Rate Monotonic Analysis for Real-Time Systems. Kluwer Academic Publishers.

Nesnas I.A., A. Wright, M. Bajracharya, R. Simmons, T. Estlin,. W.Soo Kim (2003). CLARAty: An Architecture for Reusable Robotic Software. *SPIE Aerosense Conference*, Orlando, Florida.

Ortiz F., B. Álvarez, J.A. Pastor, P. Sánchez (2003). A case study in performance evaluation of real-time teleoperation software archuitectures using UML-MAST. 8th Ada-Europe International Conference on Reliable Software Technologies. Lecture Notes on Computer Science – LNCS2361 Ed. Springer Verlag. ISBN 3-540-43784-3.

Pastor J.A. (2002). Evaluación y desarrollo incremental de una arquitectura software de referencia para sistemas de teleoperación utilizando métodos formales. *PhD Thesis, Universidad Politécnica de Cartagena*.

Pérez J., I. Ramos, J. Jaen, P. Letelier, E. Navarro (2003). PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures. *3rd IEEE International Conference on Quality Software,* Dallas, Texas.

Selic B., G. Gullekson, P.T. Ward (1994). Real-Time Object-Oriented Modeling (ROOM). *John Wiley and Son.*