Universidad Politécnica de Cartagena
Departamento de Tecnologías de la Información y las Comunicaciones

DESARROLLO DE SOFTWARE BASADO EN COMPONENTES Y DIRIGIDO POR MODELOS PARA ROBOTS DE SERVICIO

Bárbara Álvarez Torres
Diego Alonso Cáceres
Paco Ortiz Zaragoza
Juan Ángel Pastor Franco
Pedro Sánchez Palma

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Departamento de Tecnologías de la Información
y las Comunicaciones

DESARROLLO DE SOFTWARE BASADO EN COMPONENTES Y DIRIGIDO POR MODELOS PARA ROBOTS DE SERVICIO

Cartagena, Abril 2010

Índice General

1. Planteamiento de la investigación 1
1.1 Introducción 1
1.2 Objetivos de la actividad investigadora 4
1.3 Marco del trabajo investigador....6

2. Arquitecturas de referencia 9
2.1 Arquitectura de referencia para sistemas de teleoperacón.....9
2.2 Implementación de la arquitectura para el sistema ROSA....15
2.3 Implementación de la arquitectura para otros sistemas....16
2.4 Análisis de la arquitectura. Ventajas e Inconvenientes 17

3. Desarrollo de software basado en componentes 21
3.1 Marco Arquitectónico para unidades de control (ACRoSeT) 22
3.2 Instanciación de ACRoSeT en el proyecto EFTCoR....31

4. Desarrollo de software dirigido por modelos 35
4.1 Descripción de V^3CMM....39
4.2 Transformaciones entre modelos 42
4.3 Aplicación del enfoque a un caso de estudio 45

5. Descripción de actividades en el marco del proyecto EXPLORE....53
5.1 Generación de un *framework*:un caso de estudio....56
5.2 Solución del caso de estudio: un relato de patrones....68
5.2.1 Modelado de actividades y tareas 68
5.2.2 Modelado de las comunicaciones. Puertos, mensajes e int....72
5.2.3 Modelado de las máquinas de estados 77
5.2.4 Modelado de los componentes....84
5.2.5 Integración y prueba 85
5.3 Conclusiones 87

6. Conclusiones y trabajos futuros 89

7. Bibliografía. 93

Esta página está intencionalmente en blanco.

Índice de Tablas

Tabla 1. Escenarios de cambio 18
Tabla 2. Familia de robots construidos en el proyecto EFTCoR 22

Esta página está intencionalmente en blanco.

Índice de Figuras

Figura 1. ROSA. Herramientas para mantenimiento de cajas de aguas.....10
Figura 2. TRON, IRV y CRV. Mantenimiento de la vasija y el canal de t 11
Figura 3. Proceso de ingeniería de dominio 12
Figura 4. Descripción de la arquitectura para plataformas de teleoperación 14
Figura 5. Dispositivos EFTCoR 23
Figura 6. Subsistemas ACRoSeT 27
Figura 7. ACRoSeT: Vista conceptual de un SC 28
Figura 8. ACRoSeT: Vista conceptual de un MC.....29
Figura 9. ACRoSeT: Vista conceptual de un RC 30
Figura 10. Componentes del CCAS en la unidad de control de la Mesa XYZ ..31
Figura 11. Componentes CCAS del vehículo escalador.....32
Figura 12. MDA en el contexto de desarrollo de software para robótica.....36
Figura 13. Vista del proceso de desarrollo V^{3}CMM 38
Figura 14. Esquema de las vistas V^{3}CMM.....40
Figura 15. Vista estructural de V^{3}CMM 41
Figura 16. Proceso de desarrollo de una aplicación con V^{3}Studio.....45
Figura 17. Robot cartesiano desarrollado en el marco del proyecto EFTCoR ..46
Figura 18. Arquitectura software basada en ACRoSeT para robot cartesiano .47
Figura 19. Definición de modelos con V^{3}Studio para la mesa XYZ.....48
Figura 20. Máquina de estados para el componente SC_Axis.....49
Figura 21. Diagrama de clases y código Ada para el componente SC_Axis 50
Figura 22. Ejemplo del proceso de desarrollo V^{3}Studio para la mesa XYZ 51
Figura 23. Esquema de la propuesta de trabajo.....54
Figura 24. Vista estructural del caso de estudio 59
Figura 25. Reparto de actividades en tareas 60
Figura 26. Ejecución del caso de estudio 64
Figura 27. Controles de configuración 65
Figura 28. Interfaz de usuario del simulador.....67
Figura 29. Vistas de las máquinas de estado de los controladores 67
Figura 30. Hoja de ruta. Aplicación sucesiva de patrones.....69
Figura 31. La interfaz StateActivity.....70
Figura 32. La clase ActivityProcessor 71
Figura 33. Puertos de entrada y salida 73
Figura 34. Modelado de los datos intercambiados 74
Figura 35. Modelado de mensajes comando.....75
Figura 36. Modelado de datos del componente controlador.....80
Figura 37. Región ortogonal para gestión de puertos.....80
Figura 38. La clase State y sus subclases.....82

Figura 39. HierarchicalState y sus subclases 84
Figura 40. Diagrama de clases del framework..... 85
Figura 41. Sistemas robóticos desarrollados y paradigmas de la IS utilizados 91

1. Planteamiento de la investigación.

1.1. INTRODUCCIÓN.

La robótica engloba múltiples disciplinas que incluyen la mecánica, la electrónica, la ingeniería de control y la informática. La mayor parte de la algoritmia, que permite controlar los movimientos de un robot, y la "inteligencia", que le permite interaccionar con el entorno y actuar de forma autónoma, es software que se ejecuta bien, en procesadores embebidos en el propio robot o bien, en sistemas de tele-operación que lo gobiernan a distancia. Por esta razón, **los robots son "sistemas intensivos en software"**, es decir, sistemas en los que el software es una parte esencial de su diseño, construcción, desarrollo y evolución [Karlsson, 2000]. **Puesto que los robots son sistemas muy complejos, el software que los gobierna también lo es**.

Respecto a las aplicaciones de los robots, es bien conocida la utilización de robots industriales en tareas de soldadura, pintados, montaje automático, paletizado, etc. Sin embargo, hay otras muchas actividades en las que la robótica no está tan presente como cabría esperar, por ejemplo, en entornos más cercanos a los seres humanos (tareas del hogar, asistencia en hospitales, ocio, etc.) o bien en la realización de tareas más especializadas (trabajos de mecanizado y montaje, agricultura, construcción, etc.). Estas tareas se engloban en los objetivos de la robótica de servicio. De acuerdo con la definición de la Federación Internacional de Robótica **un robot de servicio es un robot que opera de forma autónoma o semi-autónoma para realizar servicios útiles para seres humanos, instalaciones o equipos, excluyendo operaciones de manufactura** [Karlsson, 2000]. Las diferencias fundamentales entre un robot industrial y un robot de servicio aparecen al considerar el tipo de tareas que realizan, las cuales condicionan su morfología y el modo en el que están programados. Mientras que los robots industriales están orientados a la realización de tareas muy repetitivas, normalmente en entornos estructurados, los robots de servicio realizan tareas muy concretas de servicio a los intereses

humanos en entornos cambiantes (asistencia personal, mantenimiento, reparación e inspección de instalaciones, etc.). Esto hace que la morfología de un robot de servicio suela estar orientada a la aplicación y que para adaptarse al entorno no puedan ser programados una sola vez como los robots industriales, sino que necesiten incorporar cierto grado de "inteligencia" que les permita adaptarse a un entorno cambiante, requisito que aumenta la complejidad de su software de control.

A finales de 2006 ya había instalados en el mundo alrededor de 40.000 robots de servicio [IFR, 2006], empleados principalmente en aplicaciones de defensa, rescate y seguridad. La robótica de servicio es un sector emergente que tendrá una influencia cada vez mayor en la sociedad y que tiene por delante un futuro muy prometedor; sin embargo, **hay dos factores que frenan su implantación: (1) la incapacidad de los robots para realizar tareas relativamente sencillas sin supervisión de una manera fiable y (2) la dificultad de los robots para interactuar con los seres humanos.**

Tradicionalmente, la investigación en robótica de servicio se ha centrado en el desarrollo de algoritmos para la navegación, la localización, la percepción sensorial, etc. Toda esta algoritmia es imprescindible para que los robots alcancen la autonomía deseada, pero no es suficiente, ya que como todo sistema complejo, un robot es mucho más que la suma de sus partes. Para conseguir una relación fluida con el entorno y con los seres humanos los robots deben ser fiables, seguros y eficientes. Cada uno de estos requisitos puede conseguirse por separado con relativa facilidad. Desafortunadamente, los sistemas reales deben satisfacerlos todos a un tiempo, lo cual supone en ciertos casos un auténtico reto tecnológico. Puesto que los robots son sistemas intensivos en software**, la robustez en su funcionamiento y su capacidad para interaccionar con el entorno dependen en gran medida de la manera en que se diseñe su software**, no sólo el que implementa los algoritmos, sino también el que gobierna su comportamiento global. A pesar de ello, es bastante habitual que dicho software sea desarrollado por ingenieros de sistemas, expertos en programación de algoritmos, pero poco familiarizados con los principios de la Ingeniería del Software [Bruyninckx, 2008]. Por esta razón, en la mayoría de los casos, no se utilizan sistemáticamente técnicas de análisis de requisitos ni de diseño del software, ni se aplican los principios y patrones de diseño más adecuados en función de las características de las aplicaciones. Los requisitos de eficiencia suelen considerarse únicamente para las tareas de control de más bajo nivel [Schlegel, 2008] y generalmente, el software que se obtiene, aunque *"funciona"*, es de baja calidad, ya que no proporciona la fiabilidad necesaria para que los robots realicen sus misiones sin supervisión y, lo que es peor, una vez que sus fallos se hacen patentes no ofrece mecanismos para solucionarlos, siendo imposible su mantenimiento y mejora.

El grupo DSIE (División de Sistemas e Ingeniería Electrónica) de la Universidad Politécnica de Cartagena tiene experiencia durante más de quince años en la utilización de las técnicas ofrecidas por la Ingeniería del Software en el proceso de desarrollo de los robots de servicio. A lo largo de este tiempo se han ido integrando en dicho desarrollo los paradigmas que dicha disciplina ha ido ofreciendo.

En la evolución de los trabajos desarrollados por el DSIE se observa un paralelismo entre cada nueva necesidad y el paradigma de desarrollo de software aplicado para solucionarla. Las primeras aplicaciones estaban dirigidas a sistemas tele-operados, cada uno de ellos especializado en un trabajo de mantenimiento para centrales nucleares muy específico en un entorno de operación completamente estructurado. El desarrollo de nuevos dispositivos robotizados implicaba el desarrollo de nuevo software de control. El reto estaba en reutilizar la mayor cantidad de código posible de una aplicación a otra. Para ello se desarrolló una arquitectura que definía los principales módulos de control y sus interacciones, y cuyo objetivo principal era reutilizar dichos módulos en distintas aplicaciones. Esta arquitectura se fundamentaba en el empleo de la programación basada en objetos y en el desarrollo de módulos de control genéricos. El lenguaje Ada permitía poner en práctica directamente estas ideas, por lo que se utilizó como lenguaje de implementación. Este modelo era válido mientras se mantenían las condiciones iniciales: tele-operación pura (los robots carecen de comportamiento autónomo), entornos estructurados, herramientas muy especializadas, etc., pero dejaba de serlo al cambiar alguna de ellas. Por ejemplo, cuando en una segunda fase hubo que desarrollar robots para limpieza de cascos de buques la arquitectura dejó de ser útil ya que surgieron entornos no completamente estructurados, tareas sólo parcialmente definidas, sistemas semi-automatizados, componentes industriales ya desarrollados, etc.

Además, a la hora de desarrollar las nuevas aplicaciones para los nuevos sistemas fue imposible definir una única arquitectura común para todas ellas. Puesto que la reutilización de componentes en diferentes sistemas se fundamentaba en que todos ellos compartían una misma arquitectura, sin dicha arquitectura común nos quedamos sin el marco que permitía la reutilización de los mismos. El reto fue entonces encontrar un enfoque para reutilizar código en aplicaciones con diferentes arquitecturas. La solución consistió en adoptar el paradigma de desarrollo software basado en componentes (*Component Based Development*, CBD) [Szyperski, 2002]. El CBD surge con el propósito de acelerar el proceso de desarrollo software al promulgar que éste debe realizarse mediante la unión de piezas independientes, al igual que sucede en el ámbito mecánico y electrónico. En esta línea, el Grupo DSIE desarrolló un marco abstracto, llamado ACRoSeT, en el cual es posible definir los componentes típicos de cualquier aplicación robótica, así como sus interfaces y

mecanismos de interacción. Esta definición es independiente de la arquitectura de la aplicación en la que se incluya el componente y de la tecnología final de implementación. Utilizando ACRoSeT se definieron las arquitecturas y los componentes de los diferentes sistemas que se desarrollaron para la limpieza de cascos de buques (vehículos trepadores, mesas XYZ para *blasting*, grúas robotizadas para grandes superficies, etc).

A pesar de los beneficios obtenidos, el empleo de ACRoSeT plantea nuevos retos que surgen como consecuencia del salto conceptual entre los conceptos de diseño basados en componentes y la tecnología de implementación dominante basada en objetos. A modo de ejemplo, cada uno de los componentes definidos en ACRoSeT tuvo que ser codificado manualmente en el lenguaje de programación elegido. Este tipo de codificación supone mucho esfuerzo y además es propensa a errores. Surge por tanto la necesidad de un enfoque que permita, entre otras cosas, trabajar al nivel de abstracción que dan los componentes (o incluso superior) y que venga acompañado de un conjunto de herramientas que faciliten la generación de aplicaciones de manera automática o semi-automática. Estas necesidades se solucionan mediante la adopción de un enfoque de desarrollo dirigido por modelos (*Model Driven Engineering*, MDE) [Stahl, 2006][Schmidt, 2006] . Dicho enfoque permite no sólo que el proceso de traducción pueda ser extendido a diversos lenguajes de programación sino incluso a componentes de *frameworks* robóticos ya existentes.

1.2. OBJETIVOS DE LA ACTIVIDAD INVESTIGADORA.

Dentro del ámbito de la robótica existe una cierta resistencia a la utilización de las técnicas de la Ingeniería del Software a pesar de que éstas poco a poco se van imponiendo. Así, las tecnologías más convencionales (basadas en los paradigmas de la programación estructurada y orientada a objetos) ya se utilizan habitualmente en el desarrollo del software para los sistemas robóticos. Sin embargo, estos sistemas evolucionan a gran velocidad al tiempo que se incrementa su complejidad encontrándonos en su desarrollo con las siguientes dificultades:

1. **Alta complejidad**, dado que deben integrar una algoritmia muy compleja, relacionada con diferentes disciplinas y habitualmente distribuida en sistemas embebidos repartidos en las diferentes partes del sistema robótico y en sistemas remotos de tele-operación.
2. **Requisitos de eficiencia y fiabilidad muy exigentes** derivados de la naturaleza de las misiones y de la interacción con los humanos y con el ambiente. Existencia de subsistemas críticos con requisitos temporales estrictos y de seguridad y tolerancia a fallos.

3. **Entornos de desarrollo ligados a lenguaje, plataforma, arquitectura y *middleware***. Dificultad de reutilización de componentes software desarrollados en o para diferentes *frameworks* de desarrollo.

El principal objetivo de la actividad investigadora que se presenta es contribuir en la medida de lo posible a la solución de los puntos arriba mencionados. Para ello, como ya se ha comentado, se han utilizado las técnicas ofrecidas por la Ingeniería del Software en las aplicaciones robóticas desarrolladas en los últimos quince años, integrando en el proceso de desarrollo de los robots de servicio los paradigmas que han ido surgiendo en este tiempo. Nuevos conceptos, como el desarrollo basado en componentes o dirigido por modelos, se van abriendo paso dentro del campo de la robótica para satisfacer aquellas expectativas que las tecnologías más convencionales no han conseguido satisfacer.

La propuesta de este trabajo de investigación es **proporcionar mejoras significativas en la productividad y calidad de los robots de servicio** desarrollados mediante la utilización de la ingeniería dirigida por modelos (MDE) y el desarrollo de software basado en componentes (CBD) combinado con el uso de patrones de diseño software.

MDE proporciona soporte para definir conceptos y notaciones que se ajusten a las necesidades de los diseñadores y transformar los modelos de diseño a otros modelos o a código. Según nuestra experiencia, el nivel de abstracción que mejor se adapta al dominio es el de los componentes considerados como unidades arquitectónicas. Por ello, **se adoptará un diseño basado en componentes (CBD)** en el que: las aplicaciones se construyen ensamblando componentes, se modela la estructura y el comportamiento de los componentes, los componentes se organizan en bibliotecas, se proporcionan mecanismos para integrar software ya disponible y las aplicaciones se generan, al menos parcialmente, de forma automática a partir del diseño de componentes.

La tecnología CBD presenta problemas que dificultan su aplicación: cada tecnología es incompatible con el resto, la reutilización queda confinada al modelo de componentes seleccionado, que está ligado a una plataforma de ejecución, y los modelos de componentes no suelen considerar ni requisitos temporales ni requisitos no funcionales. Para solucionar estos problemas **se aplicará MDE definiendo un meta-modelo de componentes independiente de la plataforma**, enfocado hacia los aspectos reutilizables de los componentes. Los detalles se añaden en los niveles inferiores incorporándolos en las transformaciones. **El objetivo es posponer la selección del modelo de componentes e integrar el diseño de componentes con herramientas de análisis y diseño**.

Todo ello requiere de la aplicación de los principios y patrones de diseño más adecuados. Respecto al empleo de dichos patrones, el reto es **organizarlos en lenguajes de patrones e integrarlos en un proceso de desarrollo basado en CBD y MDE**. La propuesta es considerar los siguientes objetivos: (1) selección de patrones relacionados con la eficiencia, la tolerancia a fallos, la distribución y la reutilización del software de los robots de servicio, (2) selección de patrones para transformar conceptos CBD en conceptos de orientación a objetos y (3) organización de los patrones en *"lenguajes de patrones"*.

Dada la dificultad del trabajo y la amplitud del dominio, **el enfoque será incremental y basado en un caso de estudio** como es el desarrollo de software para el control de un robot cartesiano. Así, se comenzará con un caso real que representa un problema abordable, pero relevante para poder aplicar MDE y CBD y experimentar con secuencias de patrones.

1.3. MARCO DEL TRABAJO INVESTIGADOR.

La experiencia de los miembros del grupo DSIE en la utilización de la Ingeniería del Software en el desarrollo de robots de servicio se enmarca en numerosos proyectos de investigación tanto nacionales como europeos. En todo este tiempo la Ingeniería del Software se ha utilizado con todas sus posibilidades para el desarrollo del software de las diferentes unidades de tele-operación y control, desde el empleo de los paradigmas de la programación estructurada y basada en objetos, en los primeros desarrollos, hasta la actual adopción de un enfoque dirigido por modelos.

En una primera etapa (1993-1998) se abordó el problema de definir arquitecturas software de referencia con componentes reutilizables para plataformas de tele-operación y control de robots que realizaban operaciones de mantenimiento en centrales nucleares. Dichos trabajos se iniciaron en el marco de los proyectos "*Aplicaciones automáticas para la reducción de dosis en operaciones de mantenimiento*" (ref. PIE 041049-AAA) y "*Software reconfigurable para robots de mantenimiento en centrales nucleares*" (ref. PATI-SARPA 753/95 y 53/96) en los que se trataba de desarrollar software original para la plataforma de tele-operación y control de un brazo robótico de Westinghouse. El software debía permitir la integración de nuevos controladores de las diferentes herramientas empleadas en las tareas de mantenimiento que se llevaban a cabo en la caja de aguas del generador de vapor. También en el marco de dichos proyectos se reutilizó la arquitectura en nuevos desarrollos como los vehículos IRV *(Inspection Retrieving Vehicle)* para la recogida de objetos en las toberas del primario y el CRV *(Canal Retrieving*

Vehicle) para la limpieza del canal de transferencia. Posteriormente, el proyecto "*Teleoperated and robotized system for maintenance in nuclear power plants*" (ref. EUREKA EU1565- MAINE TRON) permitió la reutilización de la arquitectura para un nuevo robot diseñado para el acceso a los internos inferiores de la vasija del reactor.
En una segunda etapa (1999-2007) se construyeron aplicaciones para robots de limpieza de cascos de buques. Inicialmente se evaluó la posibilidad de utilizar la misma arquitectura de referencia rediseñándola para este dominio de aplicaciones. Dichos trabajos se llevaron a cabo en el marco del proyecto "*Evaluación y rediseño de una arquitectura software de referencia para sistemas de teleoperación en base a un modelo de componentes utilizando métodos formales*" (ref. PB/5/FS/02) y se implementaron en el proyecto "*Robot escalador para limpieza de cascos de buques, respetuoso con el medio ambiente (GOYA)*" (ref. 1FD97-0823). Los resultados concluyeron que era necesario adoptar un nuevo enfoque para reutilizar código en aplicaciones con diferentes arquitecturas, por lo que el empleo de un paradigma como el desarrollo basado en componentes permitió definir un nuevo marco arquitectónico denominado ACRoSET. Dicho marco se desarrolló en el proyecto "*Arquitecturas Dinámicas para Sistemas de Teleoperación (ANCLA)*" (ref. TIC2003-07804-C05-02). Todos estos trabajos se aplicaron al desarrollo de software de los robots de limpieza de cascos de buques desarrollados en el proyecto europeo "*Environmental Friendly and Cost-Effective Technology for Coating Renoval (EFTCoR)*" (ref. G3RD-CT-2002-00794).

Recientemente, y siempre con el propósito de proporcionar mejoras significativas en la productividad y calidad de los robots de servicio desarrollados, era necesario disponer de herramientas que faciliten la generación de aplicaciones de manera automática o semi-automática a partir del nivel de abstracción que ofrece el uso de componentes o superior. Por ello, los trabajos se han enfocado a la aplicación de la ingeniería dirigida por modelos. En esta línea ya se ha trabajado en el marco del proyecto "*Modelo conceptual y tecnológico para el desarrollo de software de sistemas reactivos (MEDUSA)*" (ref. TIN2006-15175-C05-02) y se está trabajando actualmente en el proyecto "*Design Patterns and Models for the Development of Real-Time Systems (EXPLORE)*" (ref. TIN2009-08572). En estos últimos proyectos se ha extendido el nuevo enfoque de desarrollo de software dirigido por modelos a otros dominios de aplicación como las redes de sensores o la domótica.

En los siguientes capítulos se aborda con detalle cada una de las tecnologías empleadas recogiendo nuestra experiencia, desde el año 1993, en la utilización de los distintos paradigmas de la Ingeniería del Software.

Esta página está en blanco intencionalmente.

2. Arquitecturas de referencia.

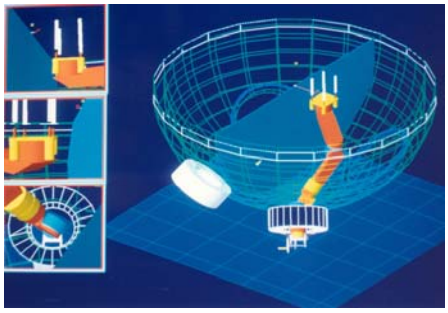
2.1. ARQUITECTURA DE REFERENCIA PARA SISTEMAS DE TELE-OPERACIÓN.

Las primeras aplicaciones para las que se utilizó de forma sistemática la Ingeniería del Software estaban relacionadas con trabajos de mantenimiento en centrales nucleares [Álvarez, 2001] [Iborra, 2003]. Estas aplicaciones se pueden clasificar en dos grandes grupos: por un lado las que estaban destinadas a proporcionar un nuevo software de control para el robot ROSA (*Remotely Operated Service Arm*) de Westinghouse, y por otro, las correspondientes a nuevos sistemas robotizados que fueron desarrollados.

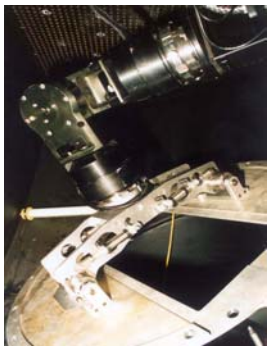
Dentro del primer grupo (véase figura 1) estaba la sustitución del software original de tele-operación del brazo ROSA por una aplicación completamente nueva que mejorara la interfaz gráfica y permitiera integrar nuevos controladores de herramientas. Se trata de un brazo articulado de seis ejes empleado para realizar distintas operaciones de mantenimiento en la caja de aguas de los generadores de vapor (inspección de la placa de tubos, instalación y retirada de las *nozzle-dams*, soldadura de tapones, eliminación de tapones, etc.).

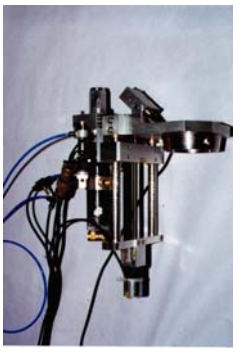
En el segundo grupo (veáse figura 2) se podrían incluir las otras aplicaciones de mantenimiento que no hacían uso del brazo ROSA, sino de nuevos robots desarrollados. Entre ellos: la pértiga TRON para acceder a los internos inferiores de la vasija del reactor, y los vehículos IRV, para recogida de objetos en las toberas del primario, y CRV, para la limpieza del canal de transferencia.

a) ROSA instalando nozzle-dam

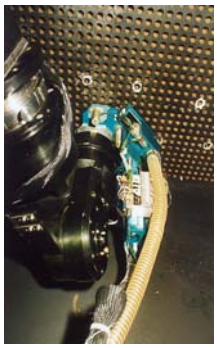


b) Representación gráfica con ROBCAD





c) Herramientas de colocación de nozzle-dams



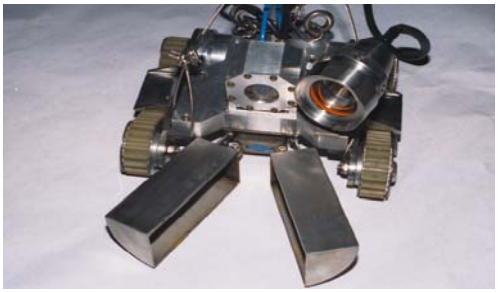
d) Herramienta de soldadura de tapones

Figura 1. ROSA. Herramientas para mantenimiento de caja de aguas.

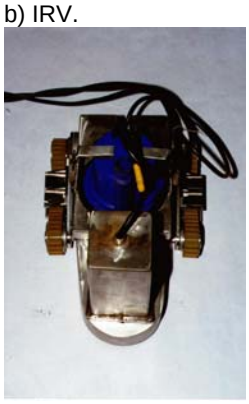
Las pautas empleadas en el desarrollo de todas estas aplicaciones se basaron en el proceso de ingeniería de dominio [Withey, 1994] para obtener una arquitectura de referencia común a todos estos sistemas. Dicho proceso distingue tres fases (véase figura 3):

- *Análisis de Dominio*. Cubre la identificación del dominio (sistemas de tele-operación), las características comunes a todas las aplicaciones del dominio y finalmente la identificación de los módulos comunes a dichas aplicaciones.
- *Diseño de Dominio*. Obtención de un diseño genérico basado en los resultados del análisis de dominio y en el estudio de patrones de diseño y arquitecturas software.
- *Implementación del diseño*. Obtención de módulos reutilizables en las distintas aplicaciones.

a) TRON.



b) IRV.



c) CRV.

Figura 2. TRON, IRV y CRV. Mantenimiento de la vasija y el canal de transferencia.

Para llevar a cabo el análisis de dominio se escogió el método FODA (*Featured-oriented domain analysis*) [Kang, 1990] ya que ofrecía técnicas para capturar y representar la información relativa a una familia de sistemas y la posibilidad de reutilización no sólo a nivel funcional sino también a nivel de arquitectura. En concreto, proponía las siguientes actividades: (1) realizar un análisis de contexto que permite definir el entorno del dominio y (2) realizar un análisis funcional para identificar similitudes y diferencias entre los sistemas existentes. Como resultado de la aplicación de este proceso obtuvimos una especificación genérica de requisitos [Alonso, 1997] en base a un modelo de características, un modelo de información (diagramas de relación entre entidades) y un modelo operacional (diagramas de actividades). Dicha especificación recogía tanto los requisitos funcionales incluyendo los requisitos temporales como los no funcionales.

Las principales características funcionales que se obtuvieron como resultado del análisis se pueden resumir en las siguientes: (1) los entornos de operación son perfectamente conocidos y estructurados, (2) el comportamiento del robot está dirigido siempre por el operador (no hay autonomía), (3) se necesita una interfaz gráfica de usuario en la que se represente en tiempo real y en tres dimensiones el estado del robot y cómo interactúa con su entorno (tal y como

se muestra en la figura 1), y (4) si se produce un fallo el sistema debe pasar a un estado conocido y seguro. Entre las características no funcionales destaca la adaptabilidad de las aplicaciones a nuevas herramientas, entornos de operación e interfaces de usuario, así como la portabilidad respecto del sistema operativo y los enlaces de comunicaciones.

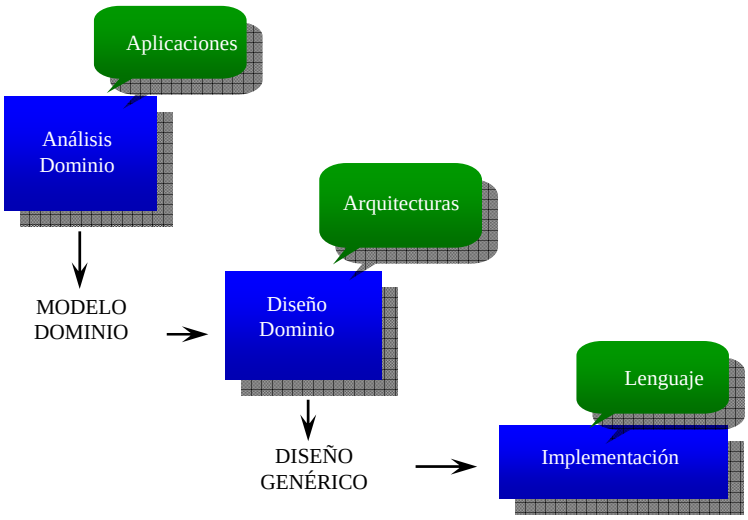


Figura 3. Proceso de Ingeniería de Dominio.

Considerando todos estos requisitos se utilizó el enfoque descrito en [Peterson, 1994] para trasladar el modelo a un diseño genérico. De acuerdo con dicho enfoque se agruparon en subsistemas los elementos que debían trabajar juntos para realizar una determinada tarea. La figura 4 muestra un diagrama de alto nivel de la arquitectura de referencia definida, en la cual se distinguen los siguientes subsistemas o componentes[1]:

- **Representación gráfica**. Este subsistema está encargado de representar en 3-D el robot y su entorno de tele-operación. Proporciona además procedimientos para seleccionar el entorno y configurar sus características. Por ejemplo, el modelo de robot, el tipo de caja de aguas, y la herramienta que porta el robot. Las actualizaciones gráficas se realizan en tiempo real de acuerdo con el estado recibido de los *resolvers* del robot.

- **Detección de colisiones & Simulación de movimientos**. Este subsistema realiza dos tipos de funciones. Por un lado chequea que un

[1] Para cubrir todos los aspectos de una descripción arquitectural se empleó el modelo 4+1 de Krutchen [Krutchen, 1995].

comando de movimiento no provocará colisiones del robot con su entorno. Por otro, una vez que el movimiento se está realizando comprueba que se ejecuta correctamente.

- **Interfaz de usuario**. Es el encargado de interaccionar con el operador, permitiéndole introducir los comandos y mostrándole el estado de ejecución del robot.

- **Comunicaciones**. Este subsistema encapsula el protocolo de comunicaciones con la unidad de control remota. Su función es enviar comandos al robot y recibir su estado.

- **Controlador del robot**. Constituye el núcleo del sistema, garantiza la viabilidad de los comandos del operador y monitoriza su ejecución. Interacciona con el resto de subsistemas para asegurar que el robot opera correctamente. En este tipo de sistemas es común tener varios controladores (por ejemplo, el caso de un brazo articulado cuya base está anclada a un vehículo y en cuyo extremo sujeta una herramienta para realizar una operación específica).

Era necesario especificar cómo los diferentes componentes iban a interactuar entre ellos. Así pues, se caracterizó la interacción entre componentes y se seleccionaron los patrones de diseño más adecuados para el manejo de flujo de datos y control [Garlan, 1996]. En concreto se utilizó el estilo arquitectónico cliente-servidor [Berson, 1992] para las interacciones entre los módulos de Representación Gráfica y Detección de Colisiones con el Controlador, siendo en ambos casos una interacción asíncrona impidiendo así el bloqueo de este último y utilizando como mecanismo de comunicación paso de mensajes [Bass, 1998] para poder implementarlos en distintas plataformas. Este mecanismo de comunicación se empleó también para las interacciones de los componentes Interfaz de Usuario y Comunicaciones con el Controlador, si bien en estos casos no era necesaria una relación causa-efecto (por ejemplo, el módulo de Comunicaciones envía periódicamente el estado del robot al Controlador y éste puede enviar comandos al robot en cualquier momento).

En cuanto al diseño interno de los distintos subsistemas, para el subsistema de Comunicaciones se empleó una arquitectura en niveles [McClain, 1991] que permitía la portabilidad a otras plataformas hardware o el empleo de protocolos alternativos. Para el resto se utilizaron tipos abstractos de datos y orientación a objetos, técnicas que promueven la reutilización y fácil modificación del sistema.

Dado que el Controlador no podía ser interrumpido por ninguna razón se introdujeron módulos con objeto de desacoplar la tarea de control del resto. Con ello, el Controlador podría seguir funcionando aunque se produjese un fallo en otro subsistema. Dichos módulos de desacoplo se ocupaban por tanto

de la comunicación con el resto de componentes y de la transferencia de datos: accedían a las interfaces mediante llamada a procedimiento y depositaban los mensajes para el Controlador en un buffer al cuál accedía el bucle de control. De manera resumida dicho bucle de control realiza las siguientes operaciones: (1) recibe comandos del operador, (2) comprueba si las operaciones son factibles, (3) simula los movimientos antes de ejecutarlos, (4) envía comandos a la unidad de control remota y (5) actualiza el estado del sistema. En caso de varios controladores es necesario tener en cuenta la sincronización entre ellos a la hora de llevar a cabo la segunda operación.

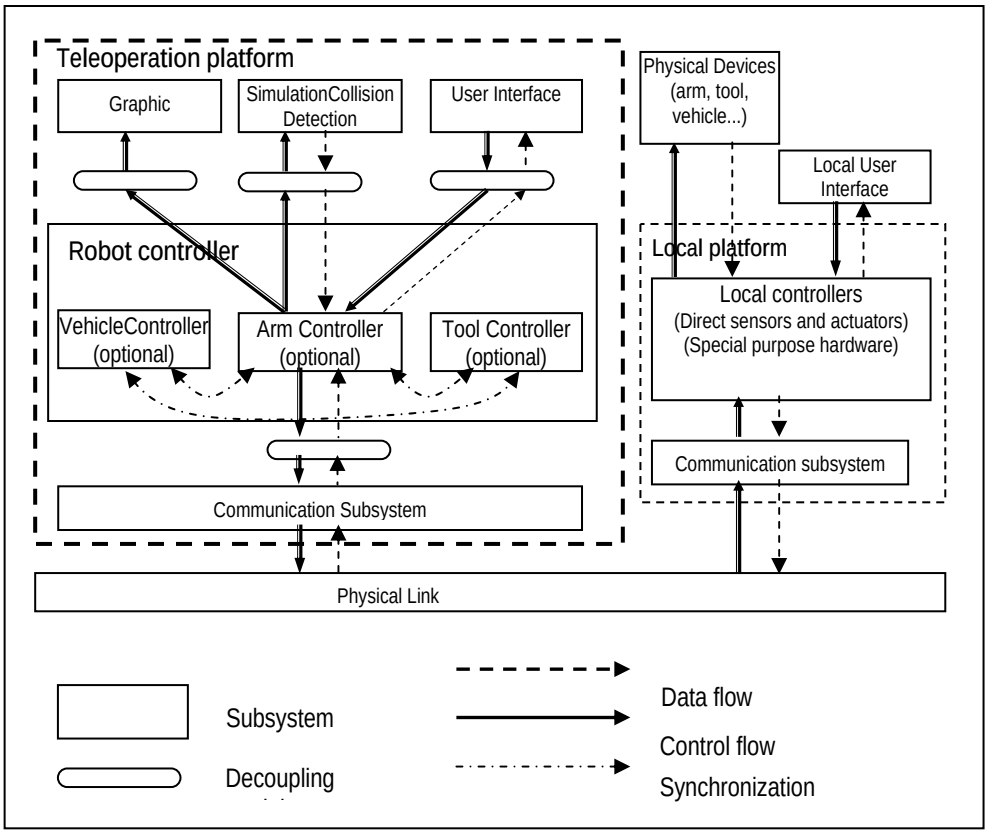


Figura 4. Descripción de la arquitectura para plataformas de tele-operación.

Dada la importancia del cumplimiento de los requisitos temporales en este tipo de sistemas, se definió un marco de trabajo [Álvarez, 1998] con objeto de ofrecer al diseñador la posibilidad de evaluar la capacidad de la arquitectura de cumplir los requisitos temporales en la implementación de una aplicación en particular haciendo uso de RMA (*Rate Monotonic Analysis*) [Klein, 1993]. El análisis proporciona un modelo para identificar las secuencias de eventos,

acciones y recursos compartidos en la arquitectura. De esta forma, el diseñador de la aplicación puede comprobar si los plazos de respuesta están garantizados, aportando la información temporal específica de la misma.

Por último, para garantizar la seguridad en este tipo de sistemas, todos los módulos debían incorporar un tratamiento de fallos a la hora de la implementación (aviso de errores, reintento de operaciones, etc.) siendo necesario el uso de código redundante o bloques de recuperación en el caso de fallo del Controlador.

2.2. IMPLEMENTACIÓN DE LA ARQUITECTURA PARA EL SISTEMA ROSA.

El primer sistema para el que se implementó la arquitectura fue para el control del brazo articulado de seis ejes ROSA en el marco de los proyectos *"Aplicaciones Automáticas para la reducción de dosis en centrales nucleares (ref. PIE-041049)"* y *"Software reconfigurable para robots de mantenimiento en centrales nucleares (ref. PATI_SARPA 753/95 y 53/96)"*. En concreto, la aplicación desarrollada provee una interfaz de usuario remota para el control del brazo y de las herramientas utilizadas en función del trabajo a realizar (máquina electro-desintegración, atornillador, soldadura de tapones, etc).

La plataforma de desarrollo para el sistema de tele-operación fue una estación de trabajo Hewlett Packard 9000 modelo 725. Una red de área local conecta dicha estación con la unidad de control que agrupa una CPU sobre un bus VME, un amplificador y una caja de entrada/salida para el control de herramientas. En concreto, se trata de una CPU Heurikon HK68/V30XE, que trabaja con el sistema operativo de tiempo real VxWork 5.1.

La implementación se llevó a cabo haciendo uso de diferentes herramientas. El módulo de Comunicaciones con la unidad de control se implementó mediante el lenguaje de programación Ada. En particular, haciendo uso de sockets TCP y UDP, a través de PARADISE (*Package of Asynchronous Real-Time Ada Drivers for Interconnected Systems Exchange*), que ofrece una interfaz a las rutinas de comunicación del sistema operativo Unix, y en concreto a las rutinas de comunicación mediante sockets. Los módulos genéricos para las interfaces entre subsistemas también fueron desarrollados haciendo uso de estas herramientas para ofrecer los servicios de comunicación entre procesos.

La Interfaz de Usuario se desarrolló en el lenguaje de programación C, siguiendo el estándar ANSI. Se emplearon los recursos ofrecidos por el sistema de ventanas X-Windows a través del lenguaje UIL (*User Interface Language)* y el gestor de recursos Motif sobre el entorno operativo Unix.

Para la implementación de los subsistemas de Representación Gráfica y Detección de Colisiones & Simulación de Movimientos se hizo uso de los servicios de ROBCAD quedando la aplicación en este caso desarrollada en el lenguaje de programación C, ya que las bibliotecas ofrecidas por la herramienta comercial estaban desarrolladas en este lenguaje.

Por último, la implementación del Controlador del robot también se llevó a cabo mediante el lenguaje de programación Ada. La cola que ordena los mensajes que el resto de los subsistemas le envían se implementó como un buffer genérico con prioridades. Para los módulos que desacoplan la tarea de control del resto de los subsistemas, se implementaron dos paquetes genéricos, uno con una tarea encargada del envío de una petición de servicio a los módulos servidores (Representación Gráfica y Detección de Colisiones) que almacena en el buffer el resultado para el Controlador, y otro que incluye dos tareas, una para el envío de mensajes, y otra que periódicamente comprueba la recepción de los mismos.

La tarea de control propiamente dicha es implementada como un bucle que chequea continuamente el buffer y toma decisiones de la siguiente acción a realizar en función del estado del sistema. Dicha tarea se ocultó en el cuerpo de un paquete genérico que tiene como parámetros formales los datos referentes a las condiciones de transición entre estados y las acciones a realizar en cada uno de ellos.

Todos los paquetes genéricos son instanciados con los datos propios del brazo articulado ROSA para definir el controlador del robot siendo reutilizados para el control de cada una de las herramientas empleadas en las tareas de mantenimiento de la placa de tubos del generador de vapor (máquina de electro-desintegración, atornillador, soldadura de tapones, etc). Cuando ambos controladores funcionan simultáneamente se habilitaban mecanismos de sincronización ya que durante la operación deben intercambiarse información de estado y señales de sincronismo.

2.3. IMPLEMENTACIÓN DE LA ARQUITECTURA PARA OTROS SISTEMAS.

Posteriormente, se reutilizó la arquitectura para otras aplicaciones de mantenimiento que no hacían uso del brazo ROSA, sino de nuevos robots desarrollados, entre ellos, los vehículos IRV para recogida de objetos en las toberas del primario y el CRV para la limpieza del canal de transferencia (véase figura 2).

El sistema IRV (*Inspection Retrieving Vehicle*) es un vehículo sumergible al cuál pueden acoplarse diversas herramientas, cámaras y sensores [Pastor, 1998]. El sistema está pensado para la recuperación de objetos que

accidentalmente hayan caído desde la caja de aguas a la tobera del primario durante las operaciones de mantenimiento. La plataforma hardware del sistema de tele-operación fue la misma que se empleó en el sistema ROSA. En este caso, era necesaria la implementación de tres controladores para vehículo, brazo y herramienta. Para ello, se instanciaron los paquetes genéricos ya desarrollados para dicho sistema. Únicamente el módulo de Representación Gráfica se implementó en otra plataforma hardware diferente al resto del sistema dado que se trataba de un sistema de visión artificial que superpone imágenes reales y virtuales. Cuando el sistema de visión reconoce una forma previamente determinada es capaz de determinar su posición respecto del extremo de la herramienta enviándosela al Controlador.

Por último, y en el marco del proyecto *"Teleoperated and robotized system for maintenance in nuclear power plants (ref. EUREKA EU1565-MAINE TRON)"* se reutilizó la arquitectura de referencia para el sistema TRON [Álvarez, 2000]. Se trata de un brazo articulado de tres ejes y un conjunto de herramientas para la recuperación de objetos en el fondo de la vasija del reactor. Como en el caso anterior, el módulo de Representación Gráfica estaba constituido por un sistema de navegación basado en visión artificial, el cuál superpone imágenes reales suministradas por las cámaras con una representación alámbrica virtual del entorno y fue implementado sobre un PC. Por otro lado, el módulo de Detección de Colisiones es muy sencillo, ya que al no estar previstos movimientos cartesianos no se contempló desarrollar la cinemática inversa. Por último, el resto de los subsistemas se implementaron sobre otro PC (dos Controladores, uno para el brazo y otro para las herramientas, así como la Interfaz de Usuario y módulos de Comunicación).

2.4. ANÁLISIS DE LA ARQUITECTURA. VENTAJAS E INCONVENIENTES.

Entre las ventajas de este enfoque cabe destacar la facilidad con que es posible reutilizar los subsistemas de una aplicación a otra, simplemente instanciando adecuadamente los parámetros genéricos. Para dar una idea más precisa de cómo se reutilizan o configuran los diferentes módulos, en la tabla 1 se muestran algunos escenarios relacionados con la modificación y extensión de las aplicaciones. Cuando se trata de añadir algo nuevo a una aplicación existente (por ejemplo, entorno, herramienta, modelo de robot) basta con definir los nuevos elementos e incorporarlos a las aplicaciones. Además, estas adiciones se realizan de forma sistemática siguiendo unas reglas definidas en la arquitectura. Si se trata de desarrollar una aplicación para un nuevo dispositivo (como muestra el último de los escenarios) hay que definir los parámetros de los módulos genéricos y programar algoritmos específicos, pero todo el código correspondiente a las interacciones entre subsistemas puede reutilizarse.

Escenario de Cambio	Cambios a realizar
Nueva operación de mantenimiento en la caja de aguas. Por ejemplo: Realización de una nueva aplicación para soldar tapones.	- Modelado gráfico de la herramienta (ROBCAD). - Instanciación del controlador genérico de la herramienta con características propias de la nueva herramienta. - Incluir el controlador de herramienta en la aplicación. - No es necesario modificar ningún subsistema software adicional.
Cambio del entorno de operación. Por ejemplo: Actualización de un nuevo modelo de caja de aguas.	- Modelado gráfico del nuevo entorno (ROBCAD).
Cambio del modelo de robot. Por ejemplo: Actualización de un nuevo modelo robot añadiendo un eje.	- Modelado gráfico del nuevo robot (ROBCAD). - Cambiar características correspondientes en el controlador del robot (módulos genéricos de Ada).
Definición de un nuevo robot para trabajar en un nuevo entorno. Por ejemplo: Un nuevo robot para recoger objetos en los internos inferiores de la vasija (TRON).	- Modelado gráfico del nuevo entorno (ROBCAD). - Modelado gráfico del nuevo robot y herramientas (ROBCAD). - Instanciación de los módulos controladores genéricos de robot y herramientas con las características de los nuevos dispositivos.

Tabla 1. Escenarios de cambio.

A pesar de las ventajas señaladas, la arquitectura no puede adaptarse a unos requisitos diferentes a los descritos. Por ejemplo, si el entorno no es estructurado deja de ser útil el subsistema de Representación Gráfica. Aunque se diseñara un nuevo sistema de Representación adaptado a ese entorno no estructurado las interacciones con el resto de subsistemas deberían ser necesariamente distintas, y por lo tanto las interfaces no serían reutilizables. De esta forma se pierde el principal beneficio de la arquitectura.

Estos inconvenientes se hicieron evidentes cuando, en el marco del proyecto *"Evaluación y rediseño de una arquitectura software de referencia para*

sistemas de teleoperación en base a un modelo de componentes utilizando métodos formales (ref. PB/5/FS/02)", se llevó a cabo un estudio con objeto de evaluar la posibilidad de reutilizar la arquitectura para el desarrollo de robots para mantenimiento de cascos de buques [Pastor, 2002]. En concreto, se utilizó el método ATAM *(Method for Architecture Evaluation)* [Kazman, 2000] para la evaluación y se emplearon herramientas como Rational Rose para la utilización del lenguaje UML (*Unified Modelling Language*) y una extensión de la misma UML-MAST [Drake, 2000] para tiempo real. Todo ello permitió hacer un análisis muy completo de la arquitectura y su aplicación al nuevo dominio.

De dicho estudio se llegó a la conclusión de que no era posible definir una única arquitectura de referencia para los sistemas de tele-operación ya que la mayor parte de los compromisos de diseño estaban relacionados con los patrones de interacción entre componentes. Era necesario un marco de desarrollo en el que poder definir diferentes arquitecturas en las que se reutilicen componentes comunes. La separación de los patrones de interacción entre componentes de la funcionalidad de dichos componentes exigía la consideración de los conectores como entidades de primera clase, al mismo nivel que los propios componentes. El enfoque de desarrollo basado en componentes tiene en cuenta estas consideraciones por lo que se adoptó con los resultados que se describen en la siguiente sección.

Esta página está en blanco intencionalmente.

3. Desarrollo de software basado en componentes.

El desarrollo de diversos robots de servicio para llevar a cabo tareas de mantenimiento en el sector naval fue abordado inicialmente por el grupo DSIE en el contexto del proyecto FEDER *"Robot escalador para la limpieza de cascos de buques respetuoso con el medio ambiente (GOYA) (ref. 1FD97-0823)"* y principalmente en el marco del proyecto europeo *"Enviromental Friendly and Cost-Effective Technology for Coating Removel (EFTCoR) del VPM de la Unión Europea (ref. Growth, G3RD-CT-2002-00794)"*. Los diferentes robots considerados debían realizar operaciones de limpieza en los cascos de los buques, en astilleros con instalaciones heterogéneas y para distintos tipos de barcos [Fernández, 2005]. Ante la dificultad de diseñar un único robot que cumpliera todos los requisitos, nos decidimos por diseñar una familia de robots (véase tabla 2) con las siguientes características:

- Debía de tratarse de robots cuyo coste económico fuera reducido, lo cual se conseguía mediante la especialización de los mismos.

- Trabajarían en entornos no estructurados, circunstancia que no debía imposibilitar que el grado de automatización de las tareas programadas fuera elevado.

- Se consideraría tanto el granallado del casco completo (*full blasting)* como el granallado de zonas muy concretas (*spotting)*.

Todos los robots de la familia EFTCoR constan de un sistema primario de posicionamiento que puede ser una torre vertical de hasta cinco grados de libertad, un vehículo escalador o una mesa elevadora. El cabezal de limpieza puede ser una turbina o varias boquillas de granallado con campana de confinamiento. Alguno de los sistemas también cuenta con un elemento secundario (mesa XYZ) que amplía el número de grados de libertad del elemento primario y mejora los tiempos de operación sobre todo en los trabajos

de *spotting*. La figura 5 muestra distintas imágenes de los sistemas construidos.

<table>
<tr><th rowspan="2">OPERACIÓN DE LIMPIEZA</th><th colspan="3">Área del casco considerada</th></tr>
<tr><th>Superficies verticales</th><th>Finos</th><th>Fondos</th></tr>
<tr><td rowspan="3">Full blasting
Grandes superficies</td><td rowspan="3">Sist. primario:
Torres verticales
Cabezal: Turbinas</td><td>Sist. primario:
Torres verticales
Cabezal: Boquilla</td><td>Sist. primario:
Mesa elevadora
Cabezal: Turbina</td></tr>
<tr><td>Sist. primario:
Vehículo escalador
Cabezal: Boquilla</td><td>Sist. primario:
Vehículo escalador
Cabezal: Boquilla</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td rowspan="3">Spotting
Pequeñas y múltiples superficies dispersas por toda la obra viva del casco.</td><td>Sist. primario:
Torres verticales
Sist. secundario:
Mesa XYZ
Cabezal: Boquilla</td><td>Sist. primario:
Torres verticales
Sist. secundario:
Mesa XYZ
Cabezal: Boquilla</td><td>Sist. primario:
Mesa elevadora
Sist. secundario:
Mesa XYZ
Cabezal: Boquilla</td></tr>
<tr><td>Sist. primario:
Vehículo escalador
Cabezal: Boquilla</td><td>Sist. primario:
Vehículo escalador
Cabezal: Boquilla</td><td>Sist. primario:
Vehículo escalador
Cabezal: Boquilla</td></tr>
<tr><td colspan="3"></td></tr>
</table>

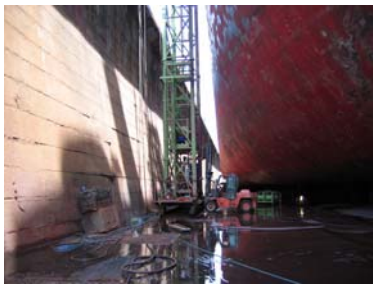
Tabla 2. Familia de robots construidos en el proyecto EFTCoR.

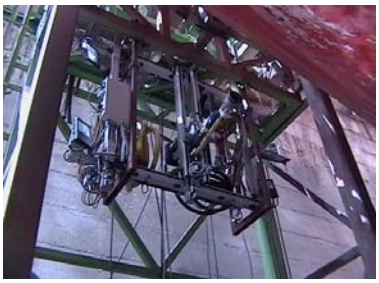
3.1. MARCO ARQUITECTÓNICO PARA UNIDADES DE CONTROL (ACRoSeT).

En el momento en que se abordó el diseñar la arquitectura software de las unidades de control para la familia de robots propuesta se detectaron los siguientes problemas:

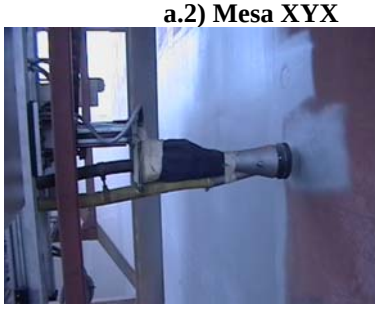
- No puede utilizarse la arquitectura de referencia usada para los robots del entorno nuclear, puesto que no se adapta a los requisitos de los nuevos sistemas.

- Los requisitos funcionales varían mucho de un sistema a otro por lo que tampoco es posible definir una nueva arquitectura de referencia común a los nuevos sistemas.





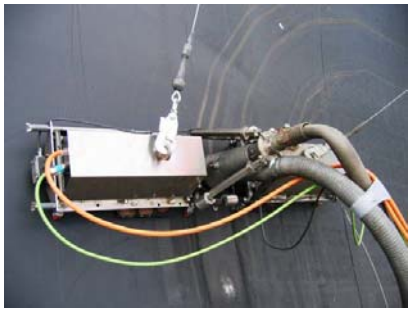
a.2) Mesa XYX



a.1) Torre vertical **a.3) Detalle del cabezal**

a) Torre vertical

b) Torre articulada



c) Robot escalador Lazaro II

Figura 5. Dispositivos EFTCoR.

- Se imponen lenguajes y plataformas de implementación distintas para cada dispositivo.

- Se impone el uso de componentes comerciales (controladores de motores, variadores de frecuencia, etc.) y lenguajes de programación de PLCs.

A la vista de estos problemas, se hizo patente la necesidad de definir un marco arquitectónico (proyecto ANCLA: "*Arquitecturas Dinámicas para Sistemas de Teleoperación*" ref. TIC2003-07804-C05-02) que:

- No impusiera ninguna arquitectura concreta, sino que permitiera definir distintas arquitecturas que se ajustaran a las restricciones particulares de cada aplicación y que fuera lo suficientemente flexible y extensible para facilitar el diseño y la evolución de un sistema robótico.

- Facilitara la reutilización de componentes para que el diseño de nuevos sistemas fuera más rápido y efectivo.

- Permitiera que la implementación final de los componentes fuera muy diversa, incluyendo tanto software como hardware, incluso componentes COTS (*Commercial Off The Shelf*).

Teniendo en cuenta estas directrices se evaluaron los *frameworks* de desarrollo de robots existentes (por ejemplo, Orocos [Bruyninckx, 2001], Carmen [Montemerlo, 2003] o Player/Stage [Gerkey, 2001]. Dichos *frameworks* proporcionan grandes tasas de reutilización y facilidad de uso; sin embargo, tienen poca flexibilidad en cuanto a la plataforma de ejecución: la mayoría de ellos están ligados a C/C++ y Linux, aunque algunos consiguen una mayor independencia gracias a la utilización de algún *middleware*. Los requisitos industriales del proyecto EFTCoR determinaban la utilización de dispositivos comerciales y lenguajes de programación de PLCs no contemplados en los *frameworks* disponibles. Además, éstos suelen tener implícita una arquitectura y ofrecen el bucle de control principal de la aplicación. Puesto que uno de los objetivos perseguidos es poder definir diferentes arquitecturas, el uso de un *framework* comercial suponía problemas adicionales.

Por todo ello, se desarrolló ACRoSeT (Arquitectura de Referencia para Robots de Servicio Teleoperados) [Ortiz, 2005] [Álvarez, 2006] como marco arquitectónico que sirve de guía en el diseño del software de control de robots de servicio teleoperados. ACRoSeT proporciona un *framework* de componentes *abstractos*, que pueden ser implementados de diversas formas (integrando distintas soluciones software/hardware e incluso componentes COTS). Definir un *framework* de componentes abstractos es hasta cierto punto paradójico, ya que lo que define a éste es precisamente la disponibilidad de

componentes concretos listos para ser utilizados de forma inmediata. ¿Por qué definir entonces componentes abstractos?
En los diferentes sistemas del EFTCoR existen componentes prácticamente idénticos en cuanto a interfaces y comportamiento, pero imposibles de integrar y por tanto de reutilizar en un sistema diferente al original, dadas las dependencias de plataforma. Necesitábamos, por tanto, una forma de definir tales interfaces y comportamiento a un nivel de abstracción más alto de forma que pudieran reutilizarse en sistemas con diferentes plataformas. Surge así la idea de los componentes *abstractos*, independientes de la plataforma de implementación, pero traducibles a un componente ejecutable, tanto software como hardware. Al optar por estos componentes abstractos se confió que las herramientas asociadas a UML para generar código evolucionaran favorablemente y esto permitiera generar código automáticamente a partir de los diagramas ACRoSeT. Se puede deducir fácilmente que estos problemas y soluciones se pueden extrapolar a otros sistemas robóticos.

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 sus principales características 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 residió en su habilidad para adaptarse a la variabilidad entre los sistemas del dominio para el que había sido definida.

Las metodologías de desarrollo orientadas a objetos y dirigidas por casos de uso como COMET [Gomaa, 2000] o ROPES [Douglas, 2000] podrían ser adecuadas; sin embargo, consideran el diseño de un sistema concreto y no de un dominio, por lo que se adopta una metodología orientada específicamente hacia el diseño de arquitecturas como el ABD (*Architecture Based Development Method)* [Bass, 1999]. ABD comienza con la acotación del dominio consistente en describir el propósito y características de los sistemas para los que se propone la arquitectura, teniendo también en cuenta los factores de negocio de la organización que la desarrolla. A partir de los requisitos (tanto funcionales como de calidad y negocio) pueden plantearse las directrices que condicionarán de forma decisiva el diseño. Dichas directrices junto con las opciones arquitectónicas para cumplirlas constituyen las entradas a dicho proceso de diseño. 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. 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. Las directrices permiten escoger un estilo arquitectónico gracias a la interacción de requisitos, funcionalidad y atributos de calidad.

Dicha metodología se completa con el modelo de 4 vistas propuesto por Hofmeister [Hofmeister, 1999] basado en UML y que permite tratar a componentes y conectores como entidades de primera clase. La vista conceptual de Hofmeister ofrece una notación que incorpora componentes, puertos y conectores, ideal para expresar el modelo de componentes abstracto o conceptual que ofrece la arquitectura propuesta. A nivel de implementación, ya para un sistema concreto, Hofmeister proponía la traducción de la vista conceptual a tres vistas: módulos, ejecución y código. En esta fase ya se pueden tener en cuenta muchos criterios de diseño aportados por los procesos de desarrollo orientados a objeto (por ejemplo, de COMET, sus criterios de estructuración y simplificación de tareas para la vista de ejecución, o los patrones propuestos en ROPES para garantizar la seguridad y los requisitos de tiempo-real).

La descripción más abstracta posible de ACRoSeT se muestra en la figura 6. Como se puede observar para cualquier robot y arquitectura se consideran cuatro grandes subsistemas:

- El subsistema de **Control, Coordinación y Abstracción de dispositivos** (CCAS), cuyas responsabilidades son la coordinación funcional y el control de los dispositivos que componen el sistema, la abstracción o representación del hardware presente en el sistema, la utilización de estrategias de coordinación y control intercambiables y configurables, y ofrecer acceso individual a los distintos mecanismos y dispositivos.

- El subsistema de **Inteligencia** (IS), tiene responsabilidad sobre: la supervisión y control de la ejecución de misiones pre-programadas, la realización de procesamientos autónomos sencillos y algunas tareas de planificación, y la generación de comportamientos reactivos que deban combinarse con las órdenes del operador (por ejemplo, evitación de obstáculos).

- El subsistema de **Interacción con los Usuarios** (UIS) proporciona al operador y a otros posibles usuarios del CCAS acceso a los servicios del sistema y muestra el estado del mismo. Interpreta los comandos que llegan de los usuarios y los redirecciona al subsistema o componente correspondiente. Además, comprueba sí los comandos son viables según el estado del sistema y gestiona su modo de control en el sistema. Por último, se encarga del arbitraje de las órdenes que lleguen a la vez de distintos usuarios.

- El subsistema de **Seguridad, Gestión y Configuración** (SMCS) monitoriza el estado y correcto funcionamiento del resto de subsistemas. Entre sus principales responsabilidades se encuentra el realizar diagnósticos en el arranque del sistema, implementar políticas

de tolerancia a fallos y monitorizar el sistema en su conjunto. Ofrece también una interfaz para la configuración de componentes, parámetros de funcionamiento y estrategias de control del sistema, así como una interfaz para la instalación y desinstalación de componentes.

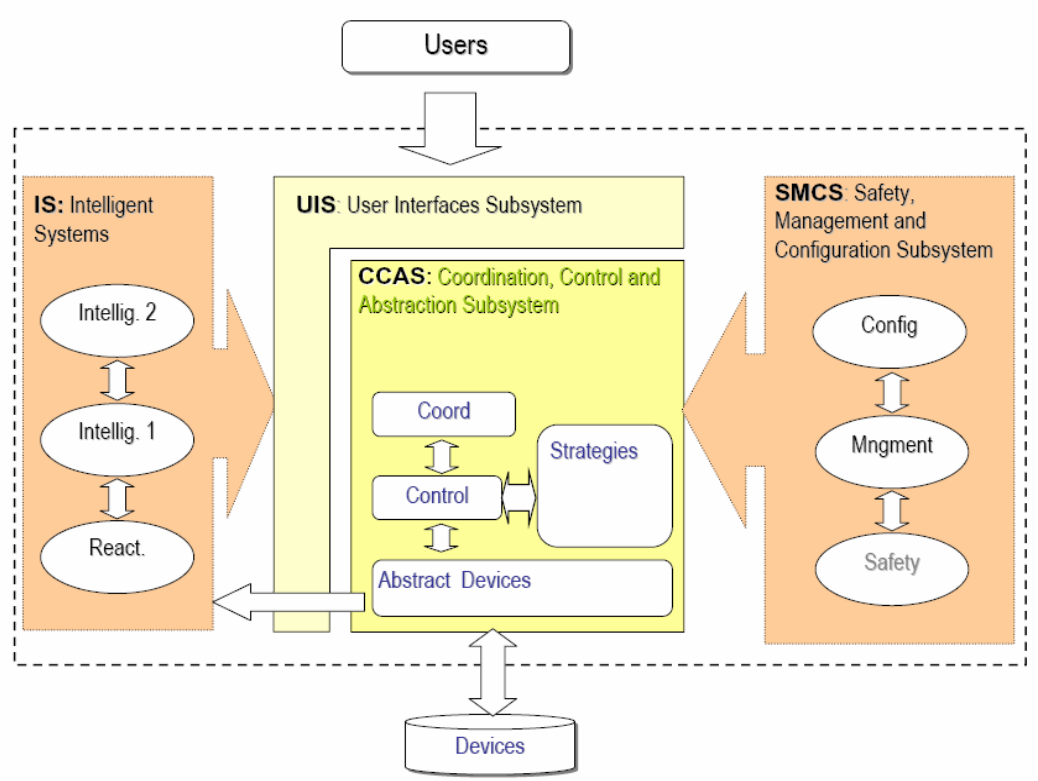


Figura 6. Subsistemas ACRoSeT.

El subsistema de **Control, Coordinación y Abstracción de dispositivos** (CCAS) se estructura según capas jerárquicas de granularidad más gruesa conforme se asciende desde los sensores y actuadores, hasta el controlador de un robot completo. La primera capa la constituyen los componentes "atómicos", normalmente sensores y actuadores. En la segunda capa aparecen los Controladores (SCs) para modelar el control de dispositivos simples (véase figura 7). Por una parte, contienen un gestor del diagrama de estado que decide si un comando debe ser ejecutado o no, o si el estado del componente debe cambiar en respuesta a una señal externa. Por otra parte, la responsabilidad principal del componente sigue el patrón *Estrategia,* de forma que el algoritmo de control puede ser modificado fácilmente, incluso en tiempo de ejecución (por ejemplo, cambiar de un control PID a un control *fuzzy*).

En un tercer nivel aparecería un coordinador que envía comandos a los SCs a los que está conectado y recibe información de éstos y de los sensores. Cada SC tiene 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. Todos estos componentes se encapsulan en un componente Controlador de Mecanismo (MC) que modela el control de un mecanismo completo (vehículo, manipulador, etc.) y su estrategia o algoritmo de coordinación (véase figura 8). Por ejemplo, para un manipulador dado, esta estrategia podría ser su cinemática inversa, que sería distinta si cambiara su configuración (número de grados de libertad, límites, etc.).

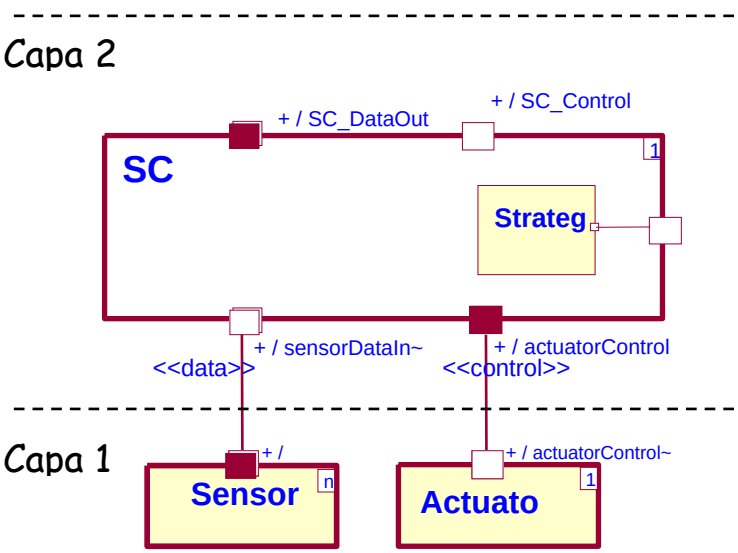


Figura 7. ACRoSeT: Vista conceptual de un SC.

Finalmente, en un cuarto nivel se define el Controlador de Robot RC (véase figura 9). Este componente modela el control sobre un robot completo, y estará compuesto de:

- MCs, habrá uno por cada mecanismo a controlar.
- SCs, que controlan elementos aislados más simples del robot, por ejemplo una herramienta sencilla.
- Un coordinador que genera los comandos para los MCs y SCs que coordina de acuerdo a órdenes que recibe, la información que recopila y la estrategia de coordinación que esté activa.

Cada componente define los servicios que ofrece y los que requiere a través de sus interfaces. Estas interfaces se corresponden con puertos, a través de los cuales se conectan los componentes usando conectores. Para que dos puertos

se puedan conectar, uno debe requerir los servicios que ofrezca el otro. Puesto que los servicios pueden requerirse de diferentes maneras, el conjunto *puerto-conector* puede encapsular un protocolo de comunicaciones. Este enfoque orientado a componentes permite diseñar el software de control a partir de componentes software que se ensamblan como si fueran componentes hardware.

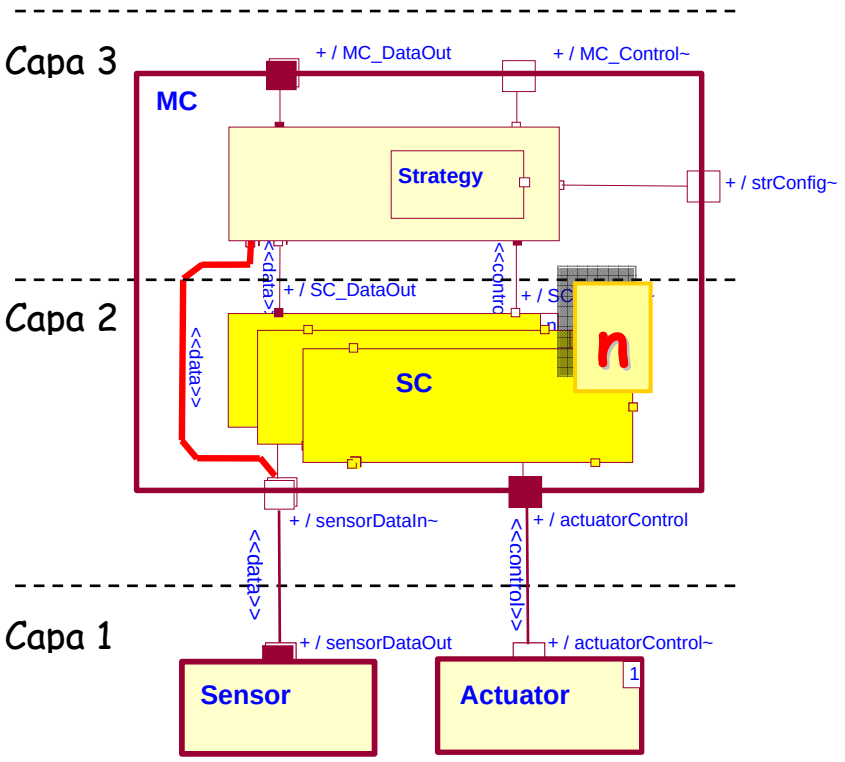


Figura 8. ACRoSeT: Vista conceptual de un MC.

La variabilidad entre los sistemas del dominio puede ser abordada por los componentes del CCAS variando los aspectos que se han definido: (1) la funcionalidad de cada componente, que mostrará sus servicios según unas interfaces, (2) las políticas de control, que están encapsuladas en los controladores y coordinadores y pueden ser intercambiadas, (3) los patrones de interacción, que son variables y están encapsulados en puertos y conectores, así como las relaciones entre los propios componentes y (4) el número y tipo de componentes, que será distinto según las características del sistema a controlar.

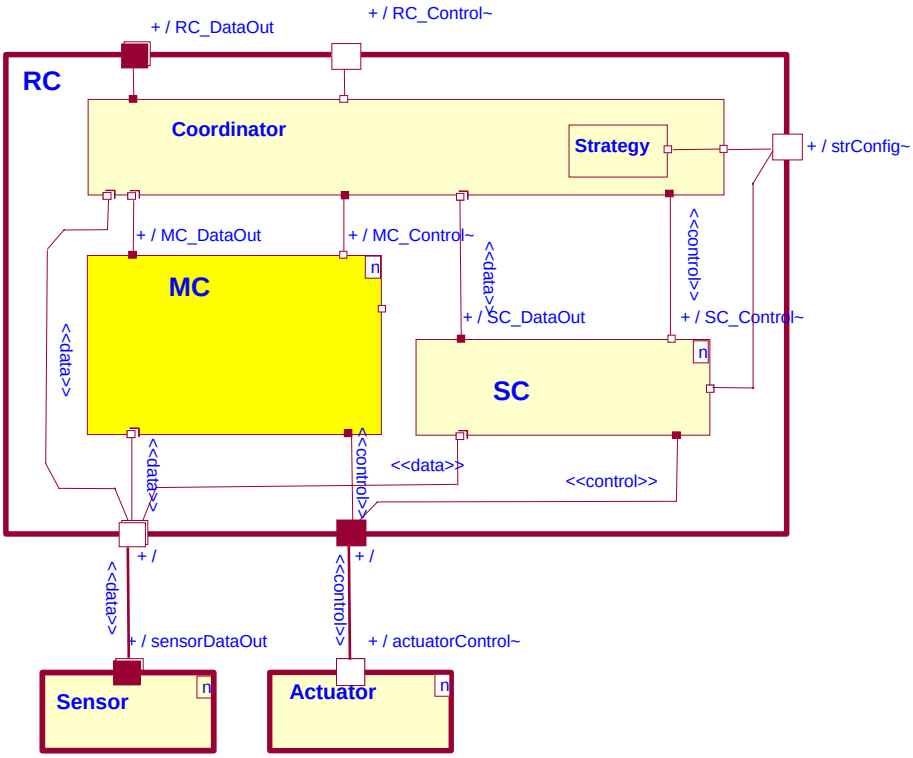


Figura 9. ACRoSeT: Vista conceptual de un RC.

Varios componentes pueden obtener información de otro componente sin ningún problema, pero cuando dos o más componentes quieren dar una orden a otro puede surgir un conflicto. Para solucionarlo, se propuso un componente que recogiese los comandos de origen y fuese interlocutor único con el destino: el componente **Árbitro**. Este componente, o conector complejo, 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 destino. Gracias al **Árbitro,** el componente que recoge el comando de control 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. En una vista jerárquica de las mismas capas que se han presentado del CCAS, se puede constatar que estos componentes de arbitraje no tienen por qué aparecer sólo en el nivel más alto, sino que pueden utilizarse siempre que varios componentes proporcionen comandos a otro componente.

Por otro lado, hay sistemas teleoperados donde el comportamiento autónomo es más complejo que la reactividad básica que incorporan los componentes del CCAS. El subsistema de **Inteligencia** (IS) introduce estos comportamientos autónomos a diferentes niveles, por ejemplo, un ejecutor de secuencias almacenadas, un componente de evitación de obstáculos, etc. El IS es un usuario más del subsistema de Coordinación, Control y Abstracción de dispositivos (CCAS), al igual que el operador u otros sistemas externos que incorporen inteligencia al sistema, por ejemplo un navegador, un sistema de visión capaz de determinar rutas libres de obstáculos, etc. El subsistema de **Interacción con los Usuarios** (UIS) hace de interfaz e intérprete entre el CCAS y sus usuarios, tanto externos, como el operador y otros sistemas (sistemas de navegación, visión artificial, etc.). Por último, en ACRoSeT se adopta la estrategia de separar la **Seguridad, Gestión y Configuración** de un sistema de su funcionalidad.

3.2. INSTANCIACIÓN DE ACROSET EN EL PROYECTO EFTCOR.

A modo de ejemplo se van a mostrar dos arquitecturas definidas con ACRoSeT, una correspondiente al controlador de la mesa XYZ y otra correspondiente al vehículo escalador. Ambas arquitecturas se corresponden con el subsistema de Control, Coordinación y Abstracción de dispositivos descrito en el apartado anterior (CCAS). El resto de subsistemas se definirían de una manera análoga. Los componentes fundamentales, que se observan en las figuras 10 y 11, se corresponden con controladores de dispositivos físicos (control de uno o varios ejes, de una herramienta, etc.).

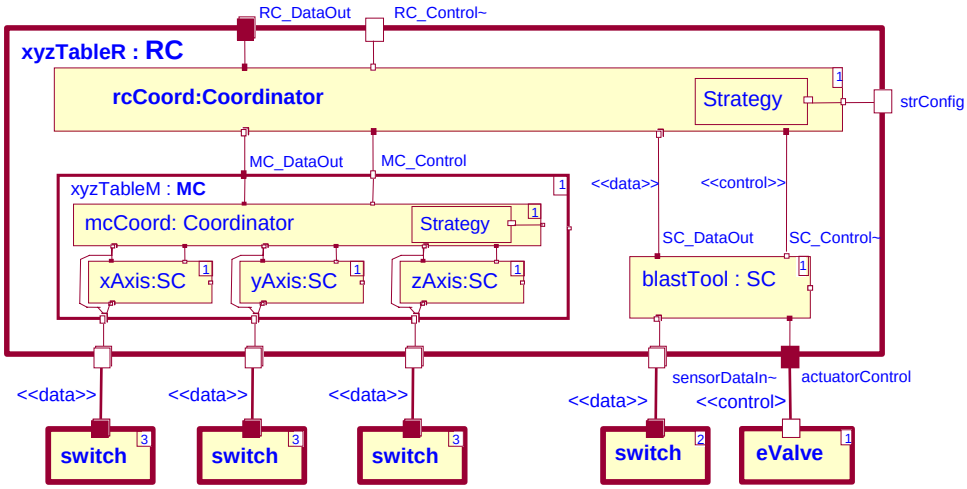


Figura 10. Componentes del CCAS en la unidad de control de la Mesa XYZ.

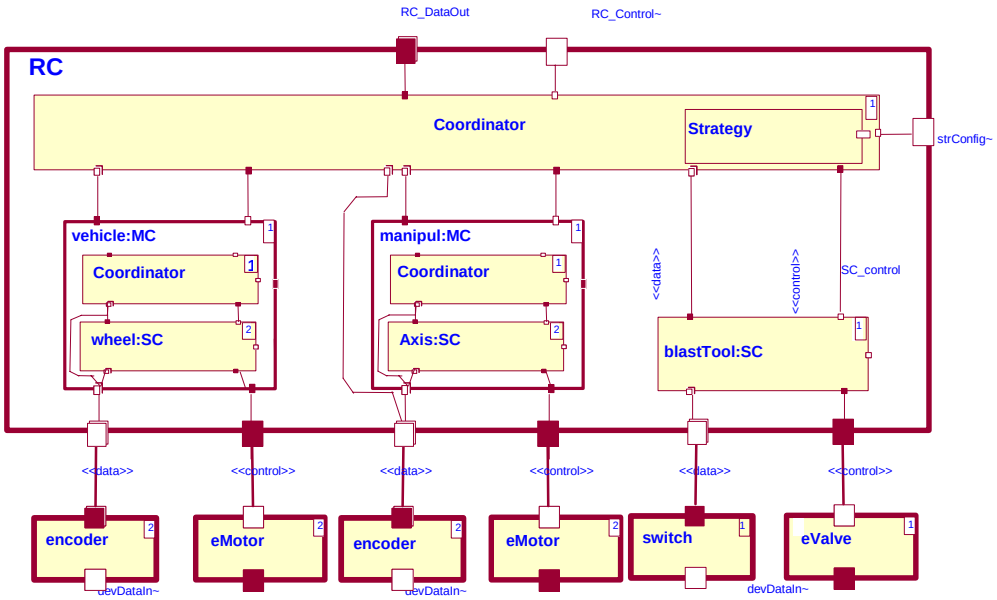


Figura 11. Componentes CCAS del vehículo escalador.

Este modelo abstracto que representa ACRoSeT se ha concretado sobre dos plataformas. Por un lado, el software de control de la mesa XYZ (figura 5a.2) se implementó sobre la infraestructura de desarrollo que proporciona SIEMENS usando un PLC 315-2DP y como entorno de desarrollo STEP7. En particular, los componentes ACRoSeT se han traducido a bloques de función (FBs) del lenguaje KOP y los diagramas de estado que definen su comportamiento se han implementado directamente en lenguaje AWL [SIMATIC, 2002] y las conexiones entre componentes se corresponden bien con llamadas a función, o bien con accesos a las tablas de entrada/salida del PLC.

Por otro lado, el vehículo escalador Lázaro (figura 5c) se ha implementado sobre un PC empotrado, haciendo uso del lenguaje Ada95. En este caso la implementación de los componentes ha sido mucho más sencilla debido al nivel de abstracción que proporciona dicho lenguaje. Los componentes se corresponden con clases y las conexiones se llevan a cabo empleando patrones de diseño propios de la orientación a objetos, como el patrón Observador [Gamma, 1995] y Objeto Activo [Schmidt, 2000].

El hecho de utilizar ACRoSeT nos ha permitido reducir significativamente el tiempo dedicado al análisis y diseño arquitectónico de los sistemas, dado que ha posibilitado:

- Diseñar los controladores y sus interacciones a un nivel de abstracción independiente de la plataforma.

- Reutilizar los componentes abstractos que define ACRoSeT en sistemas que utilizan diferentes plataformas de ejecución.

- Reutilizar los mismos componentes concretos en sistemas con diferentes arquitecturas que comparten la misma plataforma de ejecución. Por ejemplo, se han utilizado componentes comunes en las dos torres y en la mesa XYZ, si bien con diferente parametrización.

- Facilitar la extensión de los sistemas con funcionalidad adicional, ya que se simplifica en gran medida la adición y sustitución de componentes.

A pesar de todo ello y aunque la capacidad ofrecida por ACRoSeT para describir la arquitectura de distintos sistemas robóticos ha supuesto una mejora en los diseños realizados por el DSIE, la traducción manual de los componentes abstractos de ACRoSeT a componentes concretos específicos de la plataforma es una tarea difícil y propensa a errores. Por tanto, ACRoSeT sólo mostrará su potencial completo si es posible traducir automáticamente los componentes abstractos a componentes ejecutables sobre cualquier infraestructura de ejecución. En nuestra opinión, compartida por otros miembros de la comunidad científica [Bruyninckx, 2008], la solución puede venir dada por la adopción del enfoque de desarrollo dirigido por modelos.

Esta página está en blanco intencionalmente.

4. Desarrollo de software dirigido por modelos

Con los antecedentes que se han descrito, el grupo DSIE ha adoptado el enfoque de diseño *Model-Driven Engineering* (MDE) para solucionar las limitaciones detectadas en ACRoSeT. El Desarrollo Dirigido por Modelos es un enfoque que propone la utilización de los mismos como el artefacto principal para el desarrollo de software, entendiendo por modelo aquella representación simplificada de la realidad que muestra sólo aquellos aspectos que interesan y prescindiendo de los que no son de interés. En este enfoque, un modelo se define conforme a un meta-modelo, que a su vez define la sintaxis abstracta de un lenguaje de modelado y establece los conceptos y relaciones entre ellos, incluyendo además las reglas que determinan cuándo un modelo está bien formado. Existe una estrecha relación entre los diferentes modelos construidos en MDE: los modelos más abstractos son la base para la construcción de los modelos específicos. Esta relación es representada por transformaciones entre modelos, que constituyen una de las características fundamentales de este enfoque.

El hecho de que MDE sea una alternativa muy prometedora al proceso de desarrollo de software tradicional [Schmidt, 2006] se debe en gran parte a la iniciativa MDA (*Model Driven Approach*) del OMG (*Object Management Group*) [OMG, 2003]. La figura 12 muestra como los modelos en MDA pueden clasificarse en los siguientes niveles: Modelos Independientes de la Computación (CIM), Modelos Independientes de la Plataforma (PIM) y Modelos Específicos de la Plataforma (PSM). Los modelos a nivel CIM especifican los requisitos del sistema de alto nivel sin tener en cuenta decisiones de diseño o implementación. Los usuarios de estos modelos son expertos en el dominio (más que ingenieros de software o desarrolladores de aplicaciones) por lo que dichos modelos suelen especificarse haciendo uso de Lenguajes Específicos de Dominio (DSL) [Deursen, 2000]. Los modelos PIM se definen a un nivel más bajo que los CIM pero aportan una solución al diseño de alto nivel del sistema independiente de la plataforma. Por último, los modelos PSM, como su nombre indica, son específicos de la plataforma utilizada por lo que están ligados a la

implementación final. Hay que hacer notar que en cualquier caso, el concepto de plataforma está vagamente definido en MDA por lo que resulta difícil una separación clara entre los niveles PIM y PSM. Así por ejemplo, CORBA (*Common Object Request Broker Architecture*) es considerado por diferentes especificaciones de OMG tanto un PIM de bajo nivel como un PSM de alto nivel.

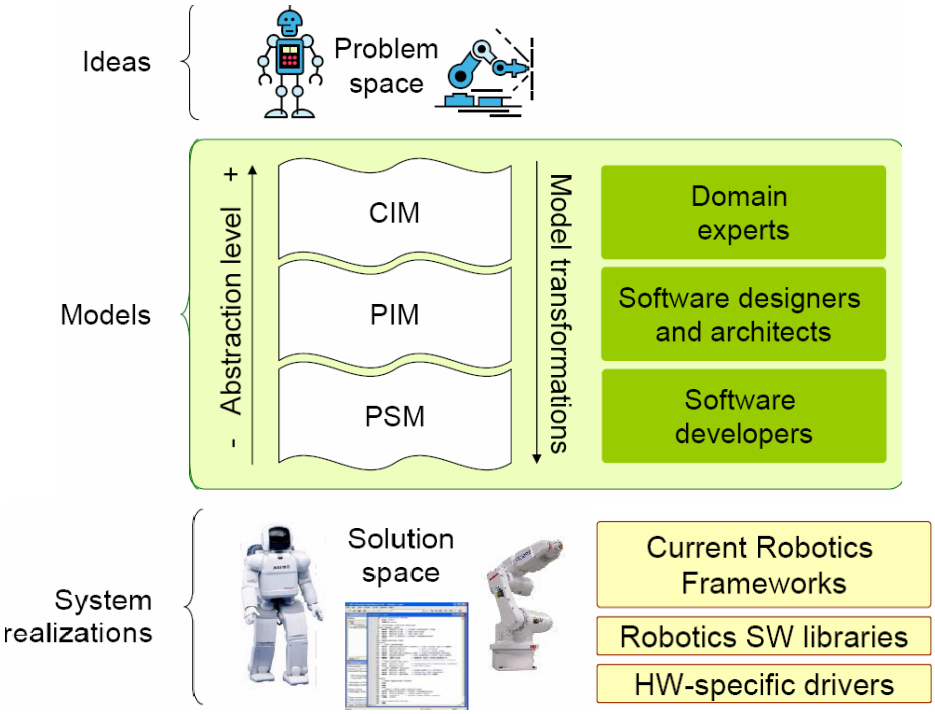


Figure 12. MDA en el contexto de desarrollo de software para robótica.

Como se ha comentado en puntos anteriores, el desarrollo de software para robótica ha estado fuertemente influenciado por paradigmas como la Orientación a Objetos o el Desarrollo Basado en Componentes. Más recientemente el paradigma MDE llama la atención de esta comunidad científica entre otras cosas debido a los resultados prometedores obtenidos en otros campos de aplicación [OMG, 2008]. Pero hay otro motivo, y es que la mayoría del software para robótica se codifica manualmente generándose la lógica necesaria para integrar la funcionalidad que proporcionan las distintas bibliotecas de componentes. Por otro lado, aunque los *frameworks* son excelentes ejemplos de reutilización de código y elevan el nivel de abstracción en relación con los lenguajes de programación dependen de una plataforma específica o *middleware* [Makarenko, 2007]. Esto lleva a que reutilizar el diseño

entre distintos *frameworks* sea casi imposible. Por ello, se propone el uso de MDE con objeto de superar las carencias que ofrecen los *frameworks* basados en componentes.

El uso del nuevo enfoque y de las tecnologías y herramientas que lo soportan, nos permite:

1. Definir un meta-modelo para ACRoSeT que permita definir formalmente tanto los conceptos que son importantes para el diseñador como las relaciones entre ellos. Además, las herramientas ligadas a MDE proporcionan la infraestructura de soporte necesaria para crear y manipular modelos de sistemas robóticos. Nótese que aún siendo conscientes de las ventajas que podría suponer para un desarrollador de software de robótica el disponer de un lenguaje a nivel de CIM, el primer paso es permitir la definición de los conceptos de ACRoSeT mediante un lenguaje de modelado independiente de la plataforma (PIM).

2. Semi-automatizar el proceso completo de desarrollo, desde la especificación de la arquitectura hasta la generación final de código, gracias a las distintas herramientas de transformación modelo-a-modelo (M2M) y modelo-a-texto (M2T). De esta forma se proporciona al usuario una herramienta que trabaja con el nivel de abstracción y con el conjunto mínimo de conceptos que requiere para diseñar cualquier aplicación basada en ACRoSeT. Posteriormente, este modelo de alto nivel de abstracción se puede concretar mediante la ejecución de distintas transformaciones de modelos, hasta obtener finalmente un modelo lo suficientemente cercano a la plataforma final de ejecución como para generar el código asociado.

La creación por parte del OMG de los estándares *Meta-Object Facility* (MOF) y *Model-Driven Architecture* (MDA) así como el desarrollo de las herramientas de soporte para dichos estándares en el contexto del entorno Eclipse [Shavor, 2003] han hecho viable un enfoque que, hasta ahora, resultaba impensable: el desarrollo de herramientas adaptadas a las necesidades de un dominio, que contienen los conceptos que necesita el diseñador con el nivel de abstracción deseado, y que evitan el trabajo directo con un lenguaje complejo como por ejemplo UML 2.0.

Así pues, y tras evaluar las ventajas e inconvenientes de utilizar UML como la notación principal para realizar el diseño de la arquitectura de un robot en base a ACRoSeT, se decidió finalmente: (1) definir un meta-modelo de componentes, denominado V^3CMM *(3-View Component Meta-Model)*, adaptado a las necesidades de modelado impuestas por ACRoSeT, en lugar de utilizar directamente UML, y (2) diseñar un proceso de desarrollo que permitiera semi-automatizar la generación del código asociado al modelo ACRoSeT. V^3CMM permite describir la arquitectura de una aplicación a partir

de los componentes que la forman, y también el comportamiento y los algoritmos que son ejecutados por dichos componentes. El lenguaje define el conjunto mínimo de elementos necesario para describir la arquitectura de las aplicaciones y prescinde de todos aquellos aspectos que, según la experiencia previa, no son necesarios. Este hecho facilita también su utilización por diseñadores que no tienen un amplio conocimiento de UML, ya que define un total de 51 conceptos, frente a los más de 200 que aparecen en UML 2.0.

La figura 13 muestra de forma esquemática el proceso de desarrollo de aplicaciones robóticas utilizando V^{3}CMM. Cada nivel de las pirámides representa un modelo que es conforme al meta-modelo ubicado en el nivel superior, cerrándose el ciclo en el meta-modelo MOF, que es conforme a sí mismo. Como se aprecia en la pirámide izquierda, V^{3}CMM es el meta-modelo en el que se definen los elementos constructivos de ACRoSeT. Una vez que se obtiene el modelo V^{3}CMM, éste podría traducirse directamente a código, o puede traducirse a un modelo orientado a objetos expresado con UML 2.0 (transformación M2M). Posteriormente, el modelo orientado a objetos se transforma automáticamente a código (transformación M2T de la pirámide derecha). Este último camino, que requiere un paso intermedio de transformación expresado en UML 2.0, tiene razón de ser porque es más fácil llegar al código final a través de dicha transformación intermedia que de manera directa. Es más sencillo realizar dos transformaciones que reducen paulatinamente el nivel de abstracción que una de mayor complejidad. Conviene resaltar que, en este esquema de desarrollo, UML 2.0 no se utiliza como lenguaje de diseño sino que la transformación M2M es automática y puede ocultarse al usuario final.

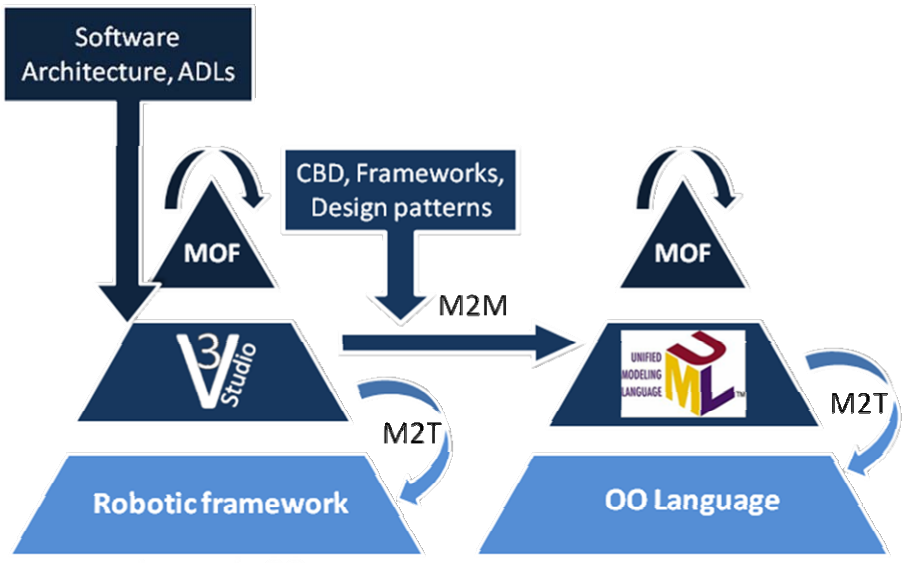


Figure 13. Vista del proceso de desarrollo V^{3}CMM.

Existen otras situaciones en los que la transformación directa M2T (pirámide izquierda) es la indicada. Éstas son aquellas en las que la aplicación a desarrollar permite utilizar alguno de los frameworks existentes (Orocos, Carmen, etc.) que son ya de por sí orientados a componentes. En estas aplicaciones la transformación intermedia sería innecesaria.

4.1. DESCRIPCIÓN DE V^{3}CMM.

El meta-modelo V^{3}CMM no sólo proporciona al diseñador un lenguaje de modelado basado en componentes independiente de la plataforma (PIM) para sistemas robóticos sino que además se trata de un lenguaje de propósito general. En este sentido, está siendo utilizado en el grupo DSIE en otros dominios como las aplicaciones basadas en redes de sensores y actuadores (WSAN) [Vicente, 2007] o el campo de la domótica [Jiménez, 2009]. El lenguaje proporciona no sólo los componentes necesarios sino un control absoluto sobre su semántica con objeto de facilitar la transformación entre modelos y garantizar las propiedades de la aplicación final. Dos ideas han sido determinantes para su diseño: la simplicidad de conceptos y la reutilización de componentes. La primera idea se plasma en el uso reducido de conceptos incluyendo únicamente el mínimo conjunto de elementos para modelar las aplicaciones. En este sentido, V^{3}CMM no “reinventa la rueda” sino que adopta y adapta algunos de los conceptos incluidos en UML, lo cual además facilita su uso en la comunidad del software. La segunda idea, como se muestra a continuación, se alcanza desacoplando los aspectos estructurales y de comportamiento de los componentes lo que favorece la reutilización con sólo actualizar las asociaciones entre ellos.

Con todos estos objetivos en mente, V^{3}CMM está estructurado en tres vistas relacionadas entre sí que permiten describir cada uno de los aspectos de la aplicación (véase figura 14): (1) **vista estructural** para describir la arquitectura de los componentes simples y complejos, (2) **vista de coordinación** para describir el comportamiento de cada componente, y (3) **vista algorítmica** para describir el algoritmo ejecutado por cada componente dependiendo de su estado actual. Además, el lenguaje permite la definición de interfaces y tipos de datos relacionados con las tres vistas pero que han sido definidos por separado. La vista de coordinación está basada en las máquinas de estado de UML, mientras que la vista algorítmica es una simplificación de sus diagramas de actividad. Así, y aunque para UML el comportamiento se puede modelar con uno u otro diagrama, se recogen todos los aspectos: con las primeras se modelan los aspectos concurrentes y dirigidos por eventos y con las segundas el flujo secuencial de la ejecución.

Un esquema de V^{3}CMM que muestra su vista estructural puede verse en la figura 15. Como puede apreciarse es similar a los diagramas de clases de UML. Los conceptos destacados en negro como `StateMachineDefinition`

y `StateMachine` pertenecen a la vista de coordinación pero se han incluido para mostrar el bajo acoplamiento entre ambas vistas. El lenguaje permite el modelado separado de definiciones e instancias de manera similar a como en programación OO se diferencia entre clases y objetos. De esta forma, las definiciones son artefactos completamente reutilizables que modelan toda la información relevante mientras que las instancias sólo contienen una referencia a la definición. Este mecanismo de reutilización es más eficiente que el que siguen la mayoría de los repositorios en los que reutilizar un componente implica crear una réplica completa cada vez que debe ser añadido a un diseño. Este hecho se muestra en la figura 15 con los conceptos de `ComponentDefinition` y `Component` respectivamente. Las definiciones de componente contienen `Ports` que exhiben las `Interfaces` provistas y requeridas por el mismo (las cuales son definidas globalmente). Los puertos definen los puntos de comunicación entre componentes mientras que las interfaces definen los mensajes que pueden intercambiarse. Por otro lado, los componentes (instancias) se definen de acuerdo a una definición de componente mediante la asociación denominada `type`.

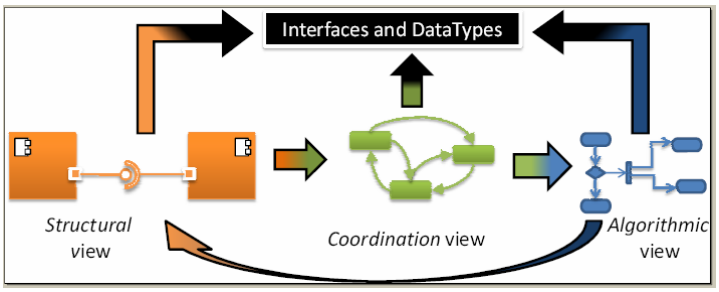


Figura 14. Esquema de las vistas de V^{3}CMM.

V^{3}CMM proporciona dos tipos de definiciones de componente: `SimpleComponentDefinitions` y `ComplexComponentDefinitions`. Los primeros son unidades atómicas con su propio comportamiento. Los segundos son unidades de composición que encapsulan instancias de otros componentes (simples o complejos). Con objeto de simplificar, se decidió que los componentes complejos no tuvieran comportamiento propio sino que vendría derivado de las instancias que contienen. De esta forma se evitan inconsistencias entre el comportamiento de un componente complejo y el resultante de combinar sus instancias internas. Si es necesario incluir en un componente complejo un comportamiento adicional, éste se modela incluyendo un componente simple (por ejemplo, un coordinador interno o un sincronizador).

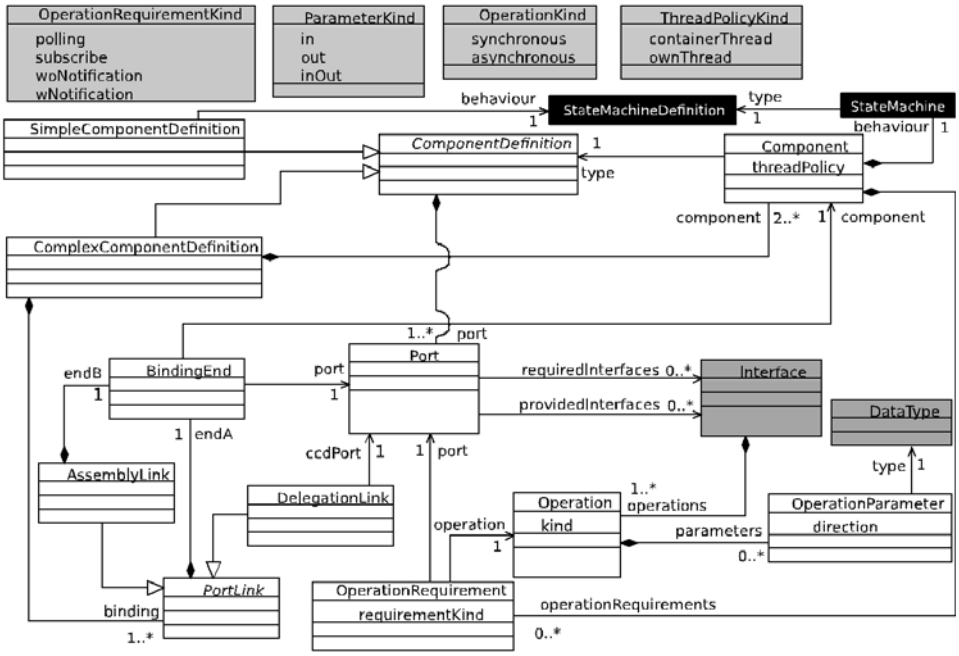


Figura 15. Vista estructural de V[3]CMM.

Como ya se ha indicado, V[3]CMM ha adoptado y adaptado los diagramas de máquinas de estados y de actividad de UML para sus vistas de coordinación y algorítmica respectivamente. La vista de coordinación incorpora muchos de los conceptos de las máquinas de estado de UML 2.0 (estado, transición, región, etc) manteniendo su significado. Sin embargo, como ocurre en la vista estructural, V[3]CMM distingue entre `StateMachineDefinitions`, que describen el comportamiento de `ComponentDefinition`, y `StateMachines` (instancias), que describen el comportamiento de `Components` (instancias). `StateMachineDefinitions` contienen los estados y transiciones que modelan el comportamiento dirigido por eventos de `ComponentDefinition`.

A diferencia de los diagramas de actividad de UML 2.0, la vista algorítmica permite a los diseñadores modelar únicamente la ejecución secuencial dado que el comportamiento concurrente queda modelado en la vista de coordinación por medio de regiones ortogonales. La vista algorítmica permite modelar tanto actividades simples (unidades atómicas) como complejas (incluyendo condicionadas o bucles) junto con el flujo de datos y de objetos que permiten su conexión. Las actividades simples que define V[3]CMM son las

siguientes: (1) `OperationCall`, para solicitar a través de un puerto operaciones que proporcionan otros componentes, (2) `ConstantActivity`, para definir una actividad constante y así bibliotecas de actividades constantes, (3) `LibraryCall`, para solicitar la ejecución de un algoritmo implementado en una biblioteca externa (engloba un mecanismo de *wrapping* para encapsular COTS), y (4) `UserDefinedActivity`, para especificar una actividad definida por el usuario. Si con las tres primeras se describen comportamientos que se van a reflejar completamente en la implementación final, cuando se incluye la última en un algoritmo, el diseñador necesitará especificar manualmente su implementación final. Este último tipo de actividad aunque proporciona un mecanismo adicional a los diseñadores puede causar una erosión en el modelo, es decir, los diseños podrían no estar representados fielmente en las implementaciones reales al haberlas alterado manualmente. Por otro lado, no hay forma de garantizar que el comportamiento codificado manualmente no viole los patrones de diseño descritos en los modelos V^3CMM.

La herramienta para dar soporte a V^3CMM ha sido integrada en Eclipse utilizando los siguientes *plug-ins*: **EMF** (*Eclipse Modeling Framework* [Steinberg, 2008], proporciona el soporte MOF), **EMF OCL** (*Object Constraint Language* [OMG, 2006], para definir limitaciones y reglas en los modelos), **ATL** (*Atlas Transformation Language* [Jouault, 2008] añade un lenguaje declarativo de transformación modelo a modelo) y **JET** (*Java Emitter Templates* [Steinberg, 2008] añade un lenguaje de transformación modelo a texto basado en plantillas).

4.2. TRANSFORMACIONES ENTRE MODELOS.

V^3CMM es un lenguaje de modelado para las partes del componente independientes de la plataforma subyacente. Las transformaciones proporcionan, por un lado, la traducción formal entre los conceptos incluidos en V^3CMM y las primitivas de la plataforma escogida (bien sea un lenguaje de programación o un *framework*), y por otro, añaden los detalles específicos de la aplicación completando el modelo y preparándolo para la generación de código. En el caso de los *frameworks* sólo se requiere un nivel de transformación ya que el resto es cubierto por la propia herramienta. Este trabajo se centra en el otro posible camino de transformación para los modelos V^3CMM (véase figura 13): se traducen a un modelo de objetos UML 2.0, mediante una transformación M2M y dicho modelo será traducido posteriormente a código Ada mediante una transformación M2T.

Como ya se ha comentado, la transformación intermedia a un modelo OO independiente de la plataforma reduce el nivel de abstracción y facilita la generación de posteriores transformaciones a diferentes lenguajes de programación (el modelo UML generado puede ser también entrada de

herramientas CASE capaces de generar código a partir del mismo). Esta transformación M2M define la correspondencia entre las tres vistas V^3CMM y sus implementaciones OO además de la infraestructura de ejecución para los componentes. De hecho, la parte que define dicha infraestructura representa un *framework* siendo fuertemente dependiente de las características de la plataforma. El diseño OO resultante de la transformación M2M propuesta mantiene la encapsulación del componente y su independencia del núcleo de ejecución. Mantener dicha encapsulación se consigue gracias al patrón de diseño Fachada [Gamma, 1995] y a las propiedades de visibilidad de UML, y la segunda utilizando el patrón Objeto Activo [Schmidt, 2000], que permite desacoplar la invocación de un método de su ejecución y facilita la concurrencia.

La transformación M2M comienza por crear un paquete base UML donde todos los artefactos serán almacenados. Dicho paquete contiene: (1) las clases abstractas *Port* y *Component*, que proporcionan la funcionalidad común necesaria para dar soporte a estos conceptos básicos, (2) un conjunto de paquetes que contienen los tipos de datos, las interfaces y sus operaciones tal como aparecen en los modelos V^3CMM, (3) un paquete auxiliar que contiene la infraestructura necesaria para asociar las llamadas a operaciones a las transiciones de la máquina de estados, y (4) un paquete para cada componente que contiene las clases que implementan su estructura y comportamiento. Mientras que los modelos estructurales se transforman a diagramas de clases UML, los modelos de comportamiento se transforman en máquinas de estado y diagramas de actividad UML haciendo directa esta parte de la transformación. Esto es particularmente interesante en el caso de las máquinas de estado ya que permite a los diseñadores retrasar la selección de un patrón de diseño a la transformación posterior a código. Por último, esta transformación genera elementos adicionales que no aparecen en los modelos V^3CMM como constructores para las clases, métodos para el envío y recepción de solicitudes de operaciones, etc.

La codificación a un lenguaje de programación OO se puede generar fácilmente a partir del modelo UML (este era el principal objetivo de la transformación M2M). Aunque el lenguaje podría ser cualquier otro, nosotros realizamos la transformación a Ada porque ofrece mecanismos robustos para el tipo de aplicaciones que nos atañen. Esta transformación M2T genera el esqueleto que los diseñadores deberán completar para las actividades `UserDefinedActivity`.

La herramienta V^3Studio permite el proceso completo de desarrollo de una aplicación. Como se resume en la figura 16, las etapas del desarrollo son las siguientes:

Paso 1. Diseño de la arquitectura de la aplicación utilizando la vista arquitectónica. En primer lugar, se definen las interfaces que van a exhibir los

componentes por sus puertos y los servicios que forman parte de estas interfaces. Posteriormente se definen los componentes que describen la aplicación y se les añaden los puertos por los que exponen su funcionalidad, en forma de interfaces ofrecidas y requeridas. Tras añadir todos los componentes, el usuario procede a enlazar los puertos compatibles. Los componentes pueden a su vez contener otros componentes.

Paso 2. Diseño de la máquina de estados de los componentes utilizando la vista de comportamiento. Para cada uno de los componentes que forman la aplicación, el diseñador tiene que crear la máquina de estados que describe el comportamiento interno del componente, así como su reacción frente a los mensajes que recibe del resto de componentes del sistema.

Paso 3. Diseño de algoritmos utilizando la vista algorítmica. Con esta vista se describen los algoritmos que serán ejecutados por el componente dependiendo del estado en que se encuentre.

Paso 4. Asociación de actividades con máquinas de estado. Tras haber definido por separado cada una de las tres vistas de la aplicación, tan sólo resta por recorrer este proceso en sentido inverso para completar el modelo. En primer lugar, se debe asociar a cada estado y transición de las máquinas de estados la correspondiente actividad que describe el algoritmo ejecutado cuando el componente se encuentre en dicho estado o dispare dicha transición.

Paso 5. Asociación de máquinas de estado a componentes. Posteriormente, se debe asociar a cada componente la máquina de estados que define su comportamiento, de entre las diseñadas en el paso 4.

Paso 6. Transformación del modelo V^3CMM a modelo UML 2.0. Se ejecuta la transformación necesaria para generar un modelo UML 2.0 equivalente. Este modelo contiene la traducción de los conceptos CBD en que se basa V^3CMM a los conceptos que manejan los lenguajes OO. Además, se generan un conjunto de clases y operaciones adicionales derivadas de los distintos patrones de diseño que se han utilizado para llevar a cabo esta traducción.

Paso 6b.Transformación del modelo V^3CMM a *Framework* robótico. Alternativamente, sería posible diseñar transformaciones que generaran código para un *framework* robótico a partir del modelo V^3CMM, ya que ambos utilizan conceptos similares (componente y conector, por ejemplo). Este paso no ha sido todavía implementado.

Paso 7. Transformación de modelo UML 2.0 a un lenguaje de programación. A partir del modelo UML 2.0 generado en el paso 6 se puede generar directamente código para cualquier lenguaje de programación OO, ya que

dicho modelo contiene únicamente los conceptos que utilizan este tipo de lenguajes.

V^{3}CMM ha sido diseñado con el requisito de facilitar y permitir la reutilización de los modelos diseñados en el mayor número de escenarios posibles. Los componentes se pueden reutilizar en diferentes arquitecturas. Las máquinas de estado se pueden reutilizar en diferentes componentes y finalmente las actividades pueden reutilizarse en diferentes máquinas de estado. Además, se ha parametrizado la forma en que un componente requiere los servicios de otros componentes. En este sentido, V^{3}CMM contempla dos tipos de servicios (síncronos y asíncronos) y dos patrones de comunicación para cada caso. En V^{3}CMM, también es posible especificar las políticas de concurrencia para los componentes indicando si el componente va a disponer de su propio hilo de ejecución, o si por el contrario, se va a ejecutar en el hilo de su contenedor.

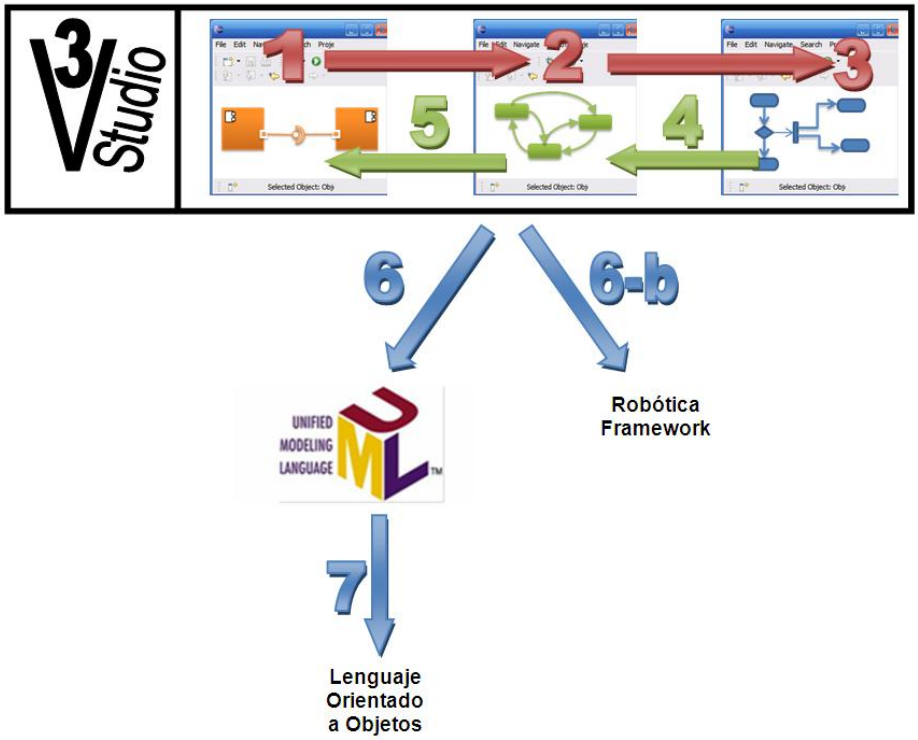


Figura 16. Proceso de desarrollo de una aplicación con V^{3}Studio

4.3. APLICACIÓN DEL ENFOQUE A UN CASO DE ESTUDIO.

A continuación se ilustra la aplicación del enfoque propuesto desde el punto de vista de un usuario de la herramienta. Se trata del desarrollo del software de

control de un robot cartesiano (mesa XYZ) para la limpieza de cascos de buques que se llevó a cabo en el marco del proyecto EFTCoR (véase figura 17). Aunque en dicho proyecto se desarrolló una familia de robots, se seleccionó como caso de estudio el control de la mesa XYZ porque anteriormente se había desarrollado el software utilizando un proceso tradicional, lo que nos permitía disponer de los algoritmos. Además, se trata de un mecanismo lo suficientemente sencillo para mostrar los beneficios del nuevo enfoque sin perdernos en otros detalles.

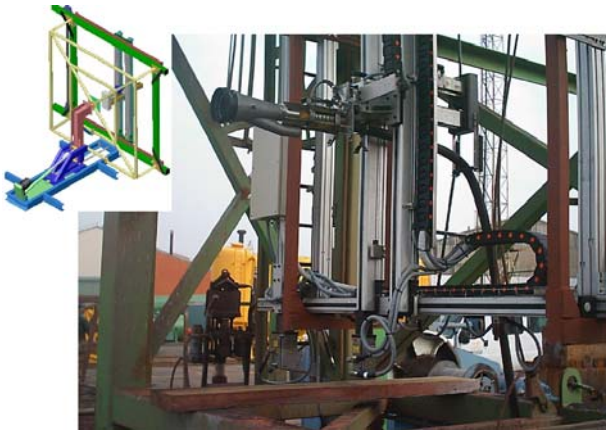


Figura 17. Robot cartesiano desarrollado en el marco del proyecto EFTCoR.

El robot cartesiano sujeta un cabezal de limpieza para la proyección de la granalla contra la superficie del barco y la recuperación de los residuos para su reciclado. La herramienta limpia un área previamente identificada por un sistema de visión artificial. Un servomotor es el encargado del movimiento de cada uno de los ejes, el cual está limitado por fines de carrera mecánicos. El robot es controlado por un PLC (*Programmable Logic Controller*) conectado a un PC sobre el que se ejecuta el software de control encargado de identificar los paños a limpiar, planificar el movimiento para posicionar el robot sobre ellos, y situar la herramienta de limpieza.

La aplicación del nuevo enfoque al caso de estudio comprende tres pasos principales: (1) modelar el sistema haciendo uso de las tres vistas V^3CMM, (2) ejecutar la transformación ATL (M2M) para generar una implementación OO (este paso puede ser transparente para el usuario final) y (3) ejecutar la transformación JET (M2T) para generar el código a partir del modelo UML obtenido.

La figura 18 muestra los componentes de la arquitectura basada en el uso de ACRoSeT para el robot: HAL (*Hardware Abstraction Layer*) para modelar la interfaz con el hardware de control, SC (*Simple Controller*) para modelar el controlador de sensores y actuadores, MC (*Mechanism Controller*) para

modelar un coordinador de varios SCs, y RC (*Robot Controller*) para modelar el coordinador de todo el robot. Tanto HALs como SCs son componentes simples, mientras que MCs y RCs son componentes complejos. Ya que éstos últimos no tienen comportamiento propio en V^{3}CMM, se requiere de un componente simple adicional (*Coordinator)* para la coordinación de sus componentes internos. La arquitectura del robot cartesiano consta también de los siguientes componentes complejos: una interfaz gráfica para mostrar los movimientos del robot al usuario (*Robot_Gui*), un planificador de movimientos (*Robot_Planner*), un sistema de visión (*Computer_Vision*) y una interfaz de usuario (*Robot_User_Interface*) que controla las órdenes que serán enviadas al componente RC.

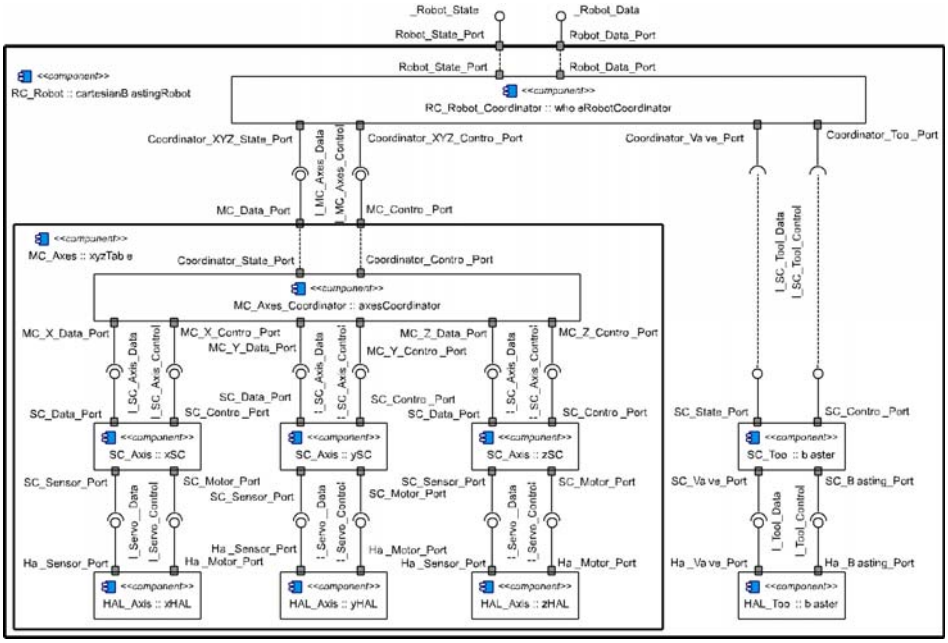


Figura 18. Arquitectura software basada en ACRoSeT para robot cartesiano.

La definición de la arquitectura por parte del diseñador haciendo uso de la herramienta V^{3}Studio consta de los siguientes pasos:

Paso 1. Definir los tipos de datos y las interfaces junto con sus operaciones (por ejemplo, en la figura 19 se muestra la definición para el componente *SC_Axis*).

Paso 2. Definir los componentes simples a partir de los tipos de datos e interfaces previamente definidas. Cada definición se realiza en su propio fichero.

Paso 3. Definir los componentes complejos a partir de las definiciones de tipos de datos, interfaces y componentes simples. Se definen en sus propios ficheros y contienen instancias de las definiciones de componentes (simples o complejos) así como enlaces entre los puertos compatibles de dichas instancias. Si los componentes están en el mismo componente complejo pueden ser conectados por medio de `AssemblyLink` (veáse figura 15) siempre que las interfaces requeridas por un puerto sean provistas por otro (y viceversa). Cuando conectamos un componente complejo a otro que lo contiene sus puertos son conectados por medio de `DelegationLink` siempre que las interfaces requeridas por un puerto sean provistas por otro (y viceversa). Como ejemplo de reutilización, nótese que el componente *MC_Axes* contiene tres instancias de *SC_Axis*, tres instancias de *HAL_Axis* y una instancia de *MC_Axes_Coordinator*.

Paso 4. Diseñar las máquinas de estado utilizando la vista de comportamiento (cada una en su propio fichero). Para cada componente simple el diseñador crea una máquina de estados describiendo su comportamiento interno y su reacción a los mensajes que recibe de otros componentes. La figura 20 describe la máquina de estados del componente simple *SC_Axis*.

Paso 5. Definir los algoritmos utilizando la vista algorítmica. Los algoritmos serán ejecutados por el componente en función de su estado actual.

Paso 6. Asociar actividades con máquinas de estado. Cada estado y transición de la máquina de estados debe ser asociada con la actividad correspondiente que describa el algoritmo que debe ser ejecutado cuando el componente está en un estado dado, entra o sale del estado o se produce una transición.

Paso 7. Asociar máquinas de estado a componentes. Cada definición de componente debe estar asociada a una de las máquinas de estado definidas en el paso 4. Dicha máquina de estados debe ser conforme con las interfaces provistas y requeridas por el componente. En nuestro caso de estudio, del mismo modo que las definiciones de componentes *HAL_Axis* y *SC_Axis* se han reutilizado tres veces, las definiciones de máquinas de estado asociadas a estos componentes son también reutilizadas tres veces dado que el comportamiento de los ejes es idéntico.

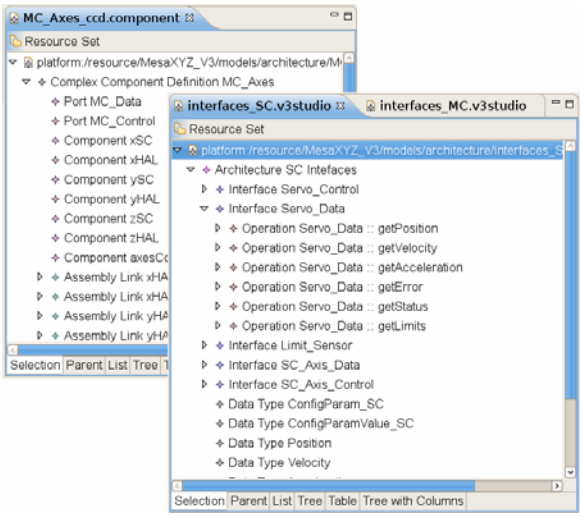


Figura 19. Definición de modelos con V[3]Studio para la mesa XYZ.

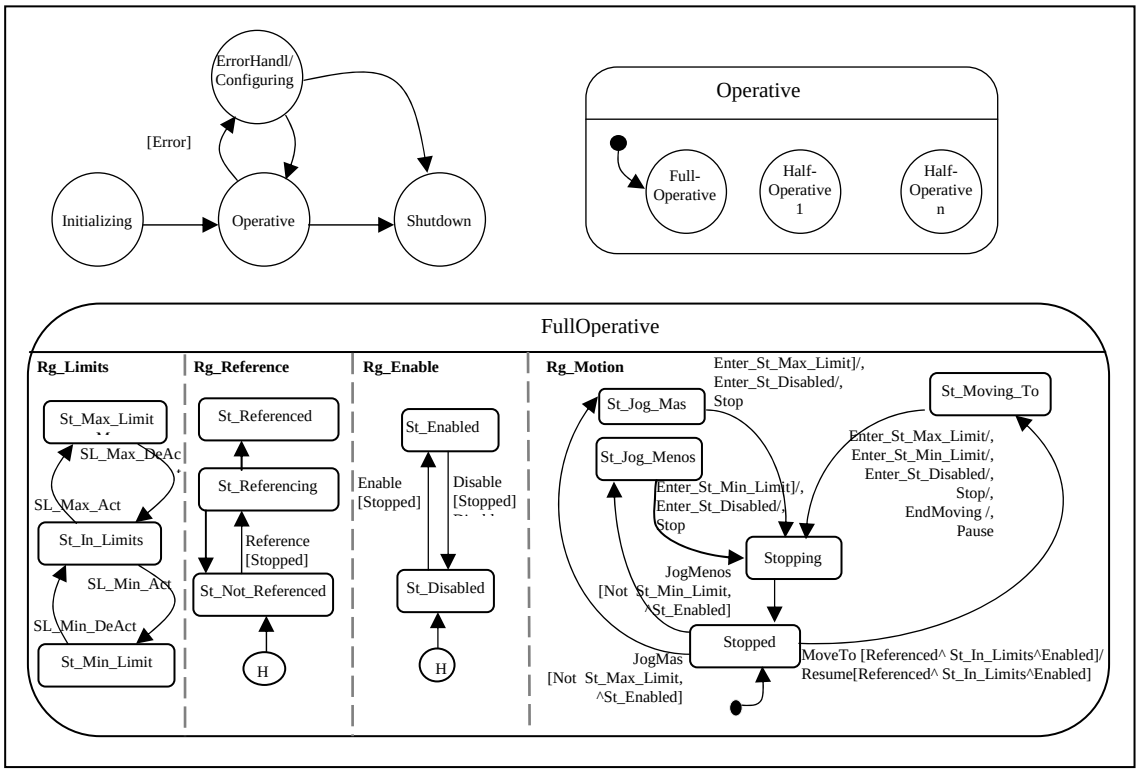


Figura 20. Máquina de estados para el componente SC_Axis.

Una vez que el usuario ha modelado la arquitectura debe configurar y ejecutar las transformaciones. Primero, debe ejecutarse la transformación M2M. La figura 21 describe una parte del diagrama de clases generado para el componente *SC_Axis*. La clase *Component_Facade_SC_Axis* proporciona la interfaz pública del componente que comprende un constructor y cuatro métodos para obtener una referencia a cada uno de sus puertos que serán utilizadas posteriormente para conectar los puertos a otros componentes. La funcionalidad del componente está implementada por la clase *SC_Axis_Core*. Cada puerto se traduce a una clase que requiere y provee interfaces. A partir del modelo generado se ejecutará la transformación M2T para la generación del código Ada (véase figura 21).

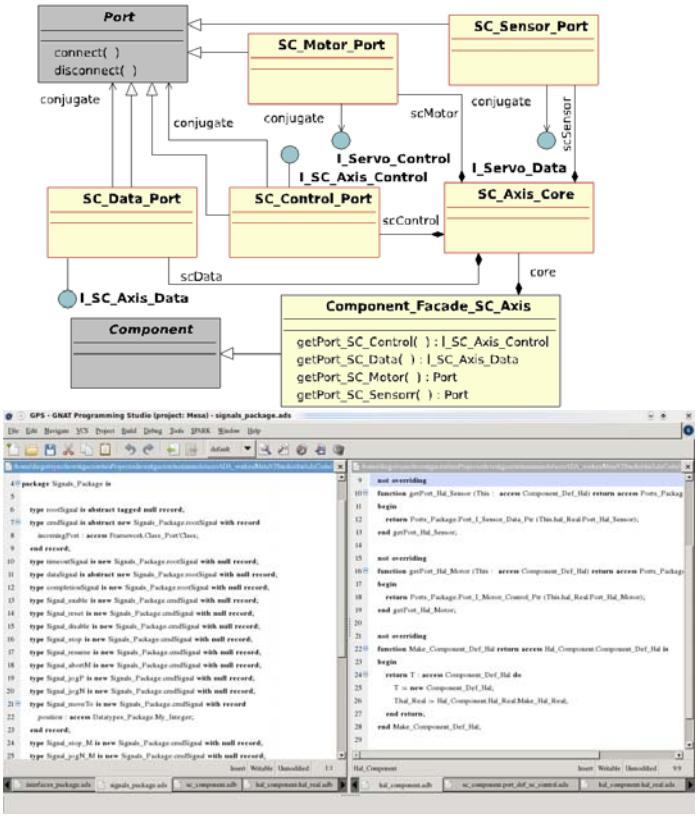


Figura 21. Diagrama de clases y código Ada para el componente SC_Axis.

La figura 22 muestra, de forma esquemática, los pasos detallados anteriormente y los distintos artefactos generados en el entorno de desarrollo V^{3}Studio hasta la obtención de código en Ada05.

El resultado de la transformación a código Ada fueron cincuenta y siete paquetes que cuentan con un total de sesenta y cinco clases. La aplicación entera contiene 4.800 líneas de código de las cuáles el 70% se han generado automáticamente. Aunque estas cifras puedan parecer elevadas se debe en parte a la aplicación de los patrones de diseño seleccionados. El tiempo de desarrollo hasta la obtención de la aplicación final fue de un mes: una semana para definir los modelos en V^{3}CMM y tres semanas para completar el código en Ada. Dicho tiempo se podría haber reducido si se hubiese contado con un conjunto de bibliotecas Ada cuya funcionalidad habría sido reutilizada (incluyendo actividades `LibraryCall` en lugar de emplear actividades `UserDefinedActivity`). Comparando con la implementación tradicional de la que ya se disponía, y que se está empleando en astilleros, decir que estaba desarrollada en C++ y contaba con 4.500 líneas de código agrupadas en 70 clases. El tiempo de desarrollo fueron 8 meses pero además, se emplea mucho tiempo en pruebas y mantenimiento de versiones sucesivas.

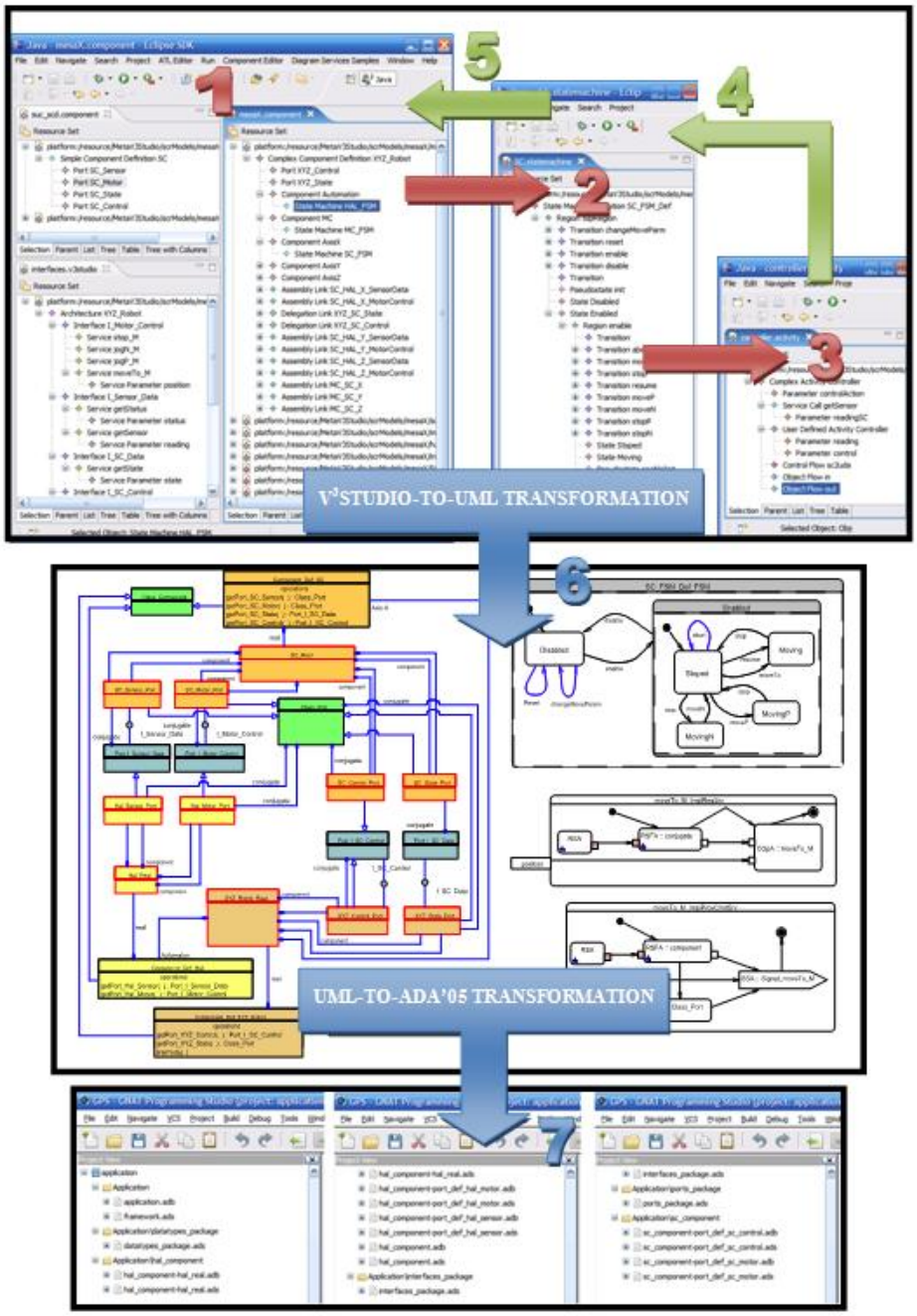


Figura 22. Ejemplo del proceso de desarrollo V^{3}Studio para la mesa XYZ.

En cuanto a la implementación de la herramienta, el desarrollo de transformaciones requiere de un profundo conocimiento del lenguaje destino, el uso de sus primitivas, y sus patrones de diseño. Todo ello puede llevar aún mucho más tiempo teniendo en cuenta el aprendizaje de las herramientas utilizadas. Sin embargo, las transformaciones proporcionan la flexibilidad requerida para poder pasar a otras plataformas y lenguajes de programación. Por ejemplo, el esfuerzo para realizar la transformación ATL de V^{3}CMM a UML fue cuatro veces mayor que el requerido para realizar la transformación JET de UML a Ada.

A pesar de las ventajas ofrecidas con la definición del entorno de desarrollo presentado faltan por cubrir diversos aspectos que no se habían considerado hasta el momento y que se irán incorporando a medida que se vayan completando las distintas fases del proyecto EXPLORE como se describe en la siguiente sección.

5. Descripción de actividades en el marco del proyecto EXPLORE[1].

El uso de MDE permite elevar el nivel de abstracción del diseño con componentes, su reutilización, así como retrasar la elección de las características de la plataforma final. La idea clave para que esto sea posible consiste en separar claramente lo que es independiente de la plataforma de ejecución de lo que no lo es. De esta forma, es posible añadir, posteriormente y por medio de transformaciones de modelos, la información específica de dicha plataforma sin contaminar el modelo original. Para ello es necesario separar el diseño de la solución para cada una de las partes. Dichas soluciones deberían ser reutilizables en todas las aplicaciones que comparten las mismas características.

Los *frameworks* son reutilizables y pueden completarse para la obtención de aplicaciones completas. Nuestro objetivo es ofrecer distintos *frameworks* de ejecución en función de las características de las aplicaciones. De esta forma, a la hora de realizar la transformación se obtendrá el código generado para los componentes y el usuario podrá seleccionar un *framework* de ejecución para dichos componentes. De esta forma, se da una infraestructura de ejecución para componentes independientes de la plataforma y se promueve el reuso de soluciones arquitectónicas a dos niveles: aplicación y *framework*.

La figura 23 resume los elementos fundamentales de nuestra propuesta de trabajo. Los desarrolladores de aplicaciones modelan la arquitectura de una aplicación utilizando V^{3}CMM y una serie de herramientas ligadas al proceso de desarrollo. Las vistas que define V^{3}CMM permiten modelar la funcionalidad de dicha aplicación, así como aquellos requisitos que dependen de la arquitectura de la misma. Para dar soporte de ejecución al modelo V^{3}CMM se aporta un *framework* que se diseña considerando los requisitos que debe cumplir dicha

[1] Proyecto EXPLORE (*DESIGN PATTERNS AND MODELS FOR THE DEVELOPMENT OF REAL-TIME SYSTEMS).*

aplicación y proporcionando los *hot-spots* necesarios para incluir la funcionalidad que aparece en el modelo V^{3}CMM. Hablamos de *framework* porque la misma estructura de ejecución puede reutilizarse en aplicaciones con distinta funcionalidad pero que comparten requisitos de ejecución similares.

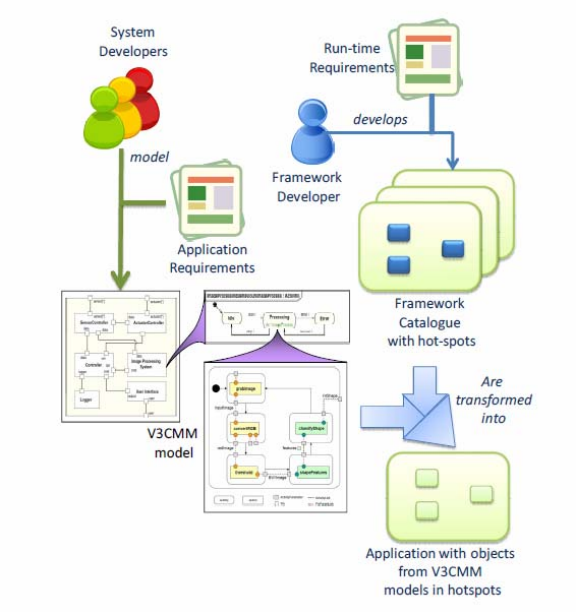


Figura 23. Esquema de la propuesta de trabajo.

En la aplicación final podrán distinguirse tres partes que no son completamente independientes:

CG1 Código correspondiente a la funcionalidad de la aplicación, obtenido tras la ejecución de una transformación de modelos a partir del modelo V^{3}CMM.

CG2 Código correspondiente a la interpretación OO de los conceptos V^{3}CMM. Por ejemplo, cómo los componentes se han traducido a objetos, cómo se implementado la máquina de estados, etc.

CG3 Código que proporciona un soporte de ejecución conforme a las características más relevantes del dominio de aplicación. Por ejemplo, si en el dominio es relevante el tiempo real estricto se debe dar soporte a características tales como *threads* y procesos, planificación, fiabilidad, etc; aún a costa de sacrificar otras características.

El código correspondiente a los items CG2 y CG3 constituye un *framework* donde debe integrarse para su ejecución la funcionalidad definida en CG1. El item CG2 define una estructura con *hot-spots* donde ubicar la funcionalidad de la aplicación. Constituye además la interfaz entre el código específico de la

aplicación (item CG1) y la infraestructura de ejecución (item CG3), facilitando la reutilización y evolución del código de ambos items por separado. Por tanto, sería deseable conseguir que el código del item CG2 fuera lo más general posible y desacoplado del código del item CG3.

Volviendo al proceso de la figura 23, una vez que el desarrollador ha terminado el modelo V^3CMM, selecciona el *framework* que mejor se adapta a las características de la aplicación, así como una transformación compatible con dicho *framework* que se encarga de: (i) generar el código correspondiente al item CG1 (funcionalidad de la aplicación), y (ii) ensamblar dicho código en los *hot-spots* del *framework* seleccionado. Nótese que las transformaciones que generan el código del item CG1 y los *framewoks* que alojan dicho código no pueden desarrollarse de forma completamente independiente, aunque esto no quiere decir que las transformaciones sólo puedan ser reutilizadas en un único *framework*. Todos estos elementos pueden gestionarse, como sucede en los *frameworks* tradicionales, mediante herramientas que proporcionen un *front-end* que permita configurar y extender la funcionalidad del *framework* en aquellos aspectos considerados en su diseño.

El desarrollo de diferentes *frameworks* de ejecución requiere la selección de un conjunto de patrones de diseño que identifiquen claramente estrategias relativas a planificación, concurrencia, tolerancia a fallos, etc, lo que a su vez puede permitir proporcionar un lenguaje de patrones para utilizarlos de forma sinérgica. Dichos lenguajes definen de manera precisa la forma de abordar el diseño de programas complejos mediante la aplicación sucesiva de patrones de diseño. De la misma manera que los patrones están focalizados hacia un cierto tipo de problemas, los lenguajes de patrones están focalizados hacia un cierto tipo de sistemas. Un lenguaje de patrones debe cumplir los siguientes objetivos: *cobertura suficiente* (que todos los problemas/requisitos queden cubiertos), *progresión sostenible* (que sea posible realizar un diseño incremental de la solución; esto es útil durante las fases de diseño y depuración y mejora), y por último, *integración fuerte/coherente* entre los patrones (depende de los roles que desempeñen las clases involucradas en cada patrón) [Buschmann, 2007].

En cualquier caso, disponer de un lenguaje de patrones implica un desarrollo incremental, por lo que hemos comenzado por un "*relato*", el cual narra la creación de un sistema software mediante la aplicación progresiva de patrones de diseño. A partir de un ejemplo específico podemos razonar de forma crítica sobre la aplicación de diversos patrones. Estos relatos no son siempre trasladables a otros sistemas pero en un paso posterior se puede generalizar la información contenida en los mismos para obtener "*secuencias de patrones*" en las que se describa como reutilizar tal conocimiento en otros sistemas. Dicha secuencia de patrones define una sucesión progresiva de decisiones de diseño y un conjunto de razonamientos que indican en qué situaciones y de qué

manera se aplica la secuencia. Finalmente, múltiples secuencias de patrones convenientemente integradas pueden formar un lenguaje de patrones.
En resumen, nuestro objetivo es utilizar MDE para generar *frameworks* especializados para cierto tipo de aplicaciones y plataformas utilizando secuencias de patrones de diseño para el desarrollo de aplicaciones basadas en componentes, así:

1. MDE proporciona la tecnología para el modelado de las aplicaciones basadas en componentes y para desarrollar las transformaciones hacia *frameworks* específicos.

2. V^3CMM proporciona el marco de componentes.

3. Los patrones de diseño, organizados en secuencias, proporcionan una guía para diseñar los *frameworks*.

5.1. GENERACIÓN DE UN *FRAMEWORK*: UN CASO DE ESTUDIO.

La materialización del enfoque es un problema muy complejo que admite diversas soluciones. Esta complejidad radica fundamentalmente en conseguir que el código generado a partir de los modelos V^3CMM pueda integrarse en cualquiera de los *frameworks* disponibles, y que además se pueda sistematizar todo el proceso. Las dependencias mutuas que existen entre el código a integrar en el *framework* y los mecanismos para integrarlo hacen que el diseño deba abordarse de forma iterativa. Por ello, el diseño de *frameworks* se va a abordar mediante la resolución de una serie de casos de estudio representativos que permitirán ir razonando sobre diferentes soluciones.

El *framework* de ejemplo que se describe en este trabajo se ha diseñado teniendo en cuenta las siguientes directrices de diseño:

- Se trata de aplicaciones reactivas con requisitos de tiempo real estrictos y con posibilidad de distribución de componentes.

- Es necesario proporcionar una implementación de los mecanismos de comunicación de acuerdo a las interfaces y puertos definidos en V^3CMM.

- Es necesario proporcionar una implementación de las máquinas de estados con *hot-spots* para incluir las actividades asociadas a cada estado en los modelos V^3CMM.

- Es necesario proporcionar un esquema de ejecución muy flexible que permita que las actividades asociadas a los estados del modelo V^3CMM puedan ser asignadas a diferentes tareas.

La última idea fue el primer objetivo: **dar una solución a cómo asignar las actividades de las máquinas de estados de los componentes de V^{3}CMM a un conjunto de tareas planificables** *(subtarea 2.2 del proyecto EXPLORE)*. Hasta el momento, V^{3}CMM no modela ninguna característica temporal. Por otro lado, el análisis de los sistemas de tiempo real asume un enfoque de diseño centrado en las tareas que realiza el sistema. Sin embargo, V^{3}CMM organiza el código en función del componente que lo ejecuta y de su estado. Esta característica introduce una dificultad para el análisis temporal: se diseña de una forma (máquinas de estados asociadas a componentes) pero las características de tiempo real se tienen que analizar de otra forma distinta (el sistema será planificable si es posible asignar el procesador a las tareas de forma que éstas cumplan sus requisitos temporales). Además, los componentes deben sincronizarse e intercambiar datos, lo cual complica aún más el problema puesto que la coordinación y comunicación entre componentes implica necesariamente la coordinación y comunicación entre tareas.

En este momento, V^{3}CMM se enfrenta a los siguientes problemas:

1. ¿Cómo añadir información para poder especificar el comportamiento temporal y analizarlo en fases posteriores?

   Al diseñar el sistema global se asigna a cada componente una funcionalidad que debe realizarse, de acuerdo con la especificación de requisitos, en un tiempo determinado. Así, dado un evento de entrada, el componente tendrá un plazo de tiempo para generar la respuesta correspondiente. Por otro lado, una vez que se ha implementado el componente hay que asegurar si esta implementación preserva la especificación temporal del mismo. Afortunadamente, es posible estimar los tiempos de cómputo de los componentes conociendo los tiempos de cómputo de las actividades que forman parte de su vista algorítmica.

2. ¿Cómo establecer correspondencias entre las actividades que realizan los componentes, que son función de su estado, y un conjunto de tareas planificables?

   El análisis de los sistemas de tiempo real requiere la caracterización del número de tareas, su tipo (periódica o esporádica), sus periodos y sus plazos. Como se ha comentado, V^{3}CMM está centrado en componentes y no en tareas, con lo que, incluso con la información temporal (tiempos de cómputo y periodicidad de actividades), hay que derivar el número y características de las tareas a partir de un modelo V^{3}CMM.

3. ¿Cómo resolver los bloqueos introducidos por los accesos a zonas de memoria protegida?

El número y la duración de las operaciones de acceso a recursos compartidos vienen dados por la estructura del código que se genera, y depende también del número de tareas que forman la aplicación. Como las interacciones entre componentes se realizan a través de puertos, cada puerto es potencialmente una zona de memoria protegida.

Para dar respuesta a las preguntas 2 y 3 se ha resuelto un caso de estudio mediante la aplicación de patrones. Se trata del robot cartesiano para la limpieza de paredes verticales de cascos de buques que se presentó en el capítulo anterior (véase figura 17). Sin embargo, el caso de estudio que se pretende resolver difiere de su precursor en algunos aspectos importantes:

- No se utiliza la mesa real, sino un simulador de una mesa con dos articulaciones, que se ha modelado como otro componente V^3CMM. Desde un punto de vista estructural se consideran los componentes de la figura 24: un simulador de una mesa de dos ejes (`XY_Simulator`), los controladores de los ejes de la mesa (`SC_Joint_X` y `SC_Joint_Y`) y una interfaz de usuario (`XY_Gui`).

- Sólo se considera el estado `FullOperative` de la máquina de estados del componente `SC_Axis` (figura 20) con cuatro regiones concurrentes. Considerando sólo este estado, y suponiendo que las máquinas de estados de `XY_Gui` y `XY_Simulator` tengan una sola región, disponemos ya de diez regiones potencialmente concurrentes, cantidad suficiente para un primer caso de estudio puesto que no es necesario complicar lo complicado con máquinas de estado más complejas.

- En un instante dado, cada región de la máquina de estados estará ejecutando la actividad correspondiente al estado en que se encuentra. Para simplificar el caso de estudio, vamos a considerar que las máquinas de estado están compuestas por regiones concurrentes compuestas a su vez por máquinas de estado no jerárquicas.

- Los puertos sólo soportan comunicaciones asíncronas sin respuesta. Si cliente y servidor necesitan otro tipo de semántica deben implementarla añadiendo puertos y definiendo el comportamiento deseado en sus máquinas de estado.

- Se considerará únicamente una planificación *Rate Monotonic Scheduling* (RMS) para las tareas.

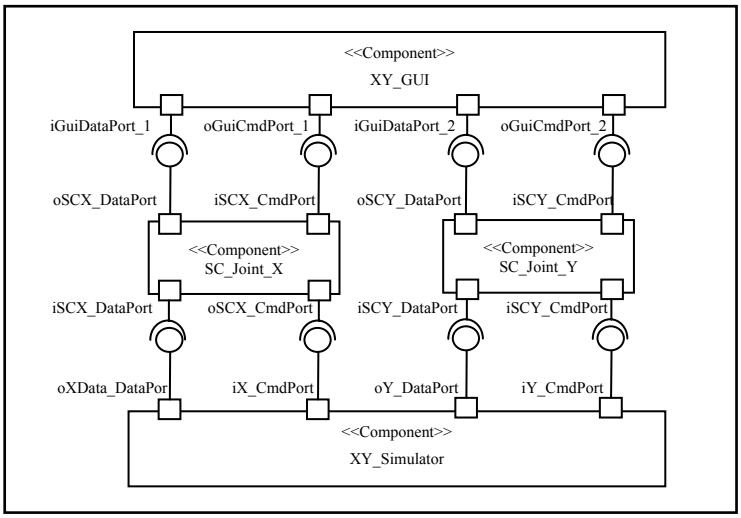


Figura 24. Vista estructural del caso de estudio.

Actualmente V^{3}CMM permite especificar la política de ejecución para cada componente, con dos valores posibles: o bien el componente se ejecuta en su propio hilo o bien en el hilo del componente que lo contiene. Esta política es un tanto rígida y por tanto de dudosa utilidad en un marco de tiempo real. Hay que llegar a una solución más flexible en la que el diseño arquitectónico se haga usando componentes y las tareas se generen haciendo uso de sus características temporales, pero sin que haya necesariamente una correspondencia directa entre componentes y tareas. Lo ideal sería coger las actividades asociadas a *todos* los estados de *todas* las máquinas de estado de *todos* los componentes, barajarlas, asignarlas arbitrariamente a un número también arbitrario de tareas, ejecutar las tareas y que todo funcione bien. En la práctica, la asignación de las actividades a las tareas no sería arbitraria, sino función de las imposiciones del método de planificación seleccionado y de los heurísticos de definición y agrupación de tareas que suelen emplearse para aumentar el rendimiento y facilitar la planificación. Sin embargo, estos requisitos, algoritmos, heurísticos y limitaciones difieren de un sistema a otro por lo que hay que dar una solución flexible, que permita: (1) diseñar las actividades desde una perspectiva independiente de las tareas y (2) asignar actividades a tareas sin imponer una relación directa entre tareas y componentes. La figura 25 muestra gráficamente el problema al que nos enfrentamos.

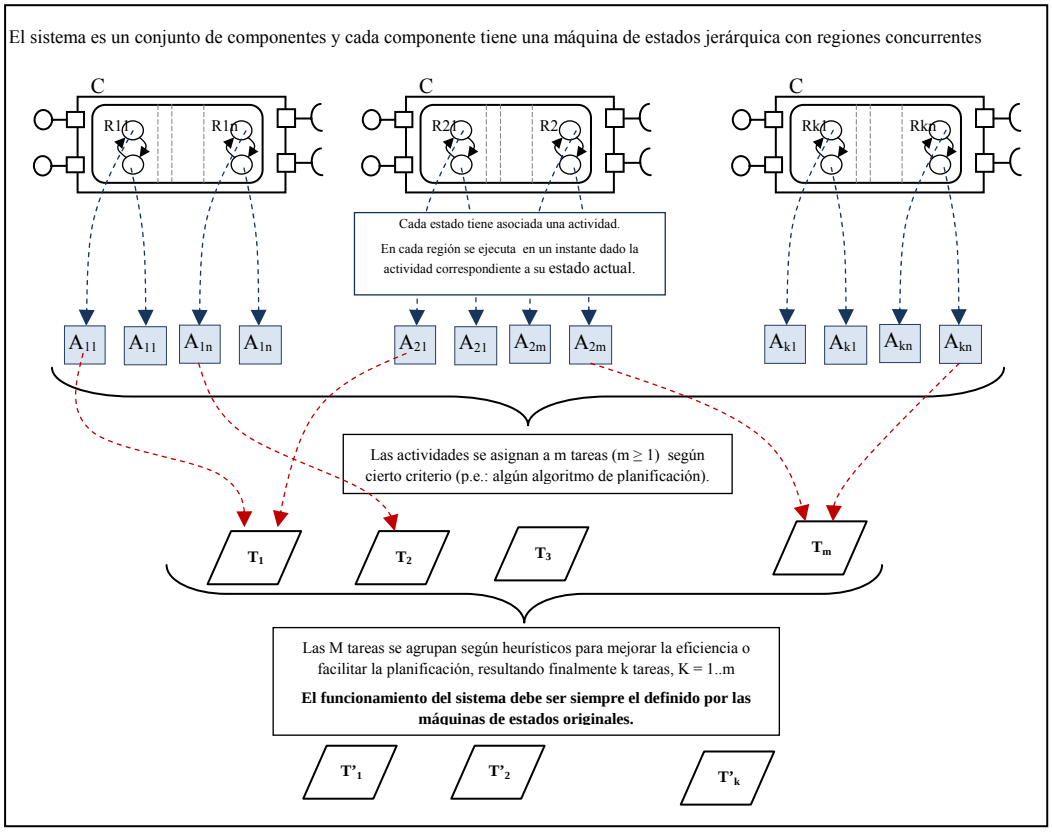


Figura 25. Reparto de actividades en tareas.

Para resolver el caso de estudio necesitamos concretar un poco más las reflexiones del párrafo anterior, de forma que se han definido los siguientes requisitos sobre actividades y tareas.

Requisitos sobre las actividades asociadas a los estados:

R1. Las ejecuciones de las actividades asociadas a los estados deben poder ser asignadas a diferentes tareas.

R2. Las actividades deben ser auto-contenidas. La tarea que las ejecuta no debe necesitar más información que la que pueda obtener de las propias actividades y esta información debe reducirse al mínimo. Se trata de minimizar el acoplamiento entre la tarea que ejecuta la actividad y la actividad misma.

R3. Se debe modelar el máximo intervalo de tiempo posible entre dos ejecuciones sucesivas de la actividad asociada a un estado.

R4. La actividad debe ser lo más corta posible. El tiempo de cómputo de las tareas no puede ser mayor que su periodo de ejecución. Puesto que el tiempo de cómputo de una tarea es la suma de los tiempos de cómputo de las actividades que se le han asignado conviene que dichas actividades se ejecuten en el mínimo tiempo posible.

Requisitos sobre las tareas:

R5. A las tareas debe poder asignárseles las actividades que deben ejecutar (**R1**).

R6. Las tareas deben poder planificar la ejecución de sus actividades.

R7. Las tareas deben poder ser a su vez planificadas. Muy especialmente, debe poder aplicarse una planificación RMS.

Como se ha comentado, es necesario proporcionar una implementación de los mecanismos de comunicación de acuerdo a las interfaces y puertos definidos en V^3CMM. La comunicación entre componentes en V^3CMM se hace a través de puertos. A los puertos se asocian interfaces compuestas por métodos definidos por sus signaturas y sus tipos de retorno. A diferencia de la orientación a objetos, la semántica de la llamada puede ser síncrona o asíncrona, siendo esta última la forma natural de comunicación entre componentes de V^3CMM. La llamada síncrona puede ser suficiente para objetos que colaboran dentro de un programa con un único hilo pero un diseño V^3CMM modela un sistema concurrente y potencialmente distribuido donde los componentes clientes no siempre pueden esperar a que el componente servidor devuelva una respuesta. En cualquier caso, parece razonable imponer en el caso de estudio los siguientes requisitos a la implementación de las comunicaciones entre componentes.

Requisitos sobre las comunicaciones entre componentes:

R8. Deben soportarse las interfaces y los tipos de comunicación definidos para los puertos de los componentes V^3CMM.

R9. El intercambio de datos en los puertos debe ser seguro garantizando la consistencia de los datos y la ausencia de interbloqueos. Si el funcionamiento del componente así lo exige deben evitarse pérdidas y duplicados de datos.

R10. El acceso a los puertos debe ser eficiente.

R11. La implementación debe complicar tan poco como sea posible el sincronismo entre tareas y/o componentes.

R12. Debe ser viable extender la solución para soportar la distribución de componentes.

V^3CMM considera máquinas de estado jerárquicas y concurrentes. Sin embargo, en el propio caso de estudio se plantea una versión simplificada con objeto de acotar la complejidad del problema. Con todo ello, se definen los siguientes requisitos para proporcionar una implementación de las máquinas de estados con *hot-spots* para incluir las actividades asociadas a cada estado en los modelos V^3CMM.

Requisitos sobre las máquinas de estados:

R13. Las actividades asociadas a sus estados deben cumplir los requisitos **R1** a **R4**.

R14. Deben considerarse explícitamente las acciones de entrada y salida de los estados.

R15. Deben implementarse regiones concurrentes compuestas por máquinas de estados no jerárquicas. En un instante determinado, cada región estará ejecutando la actividad correspondiente al estado en que se encuentra su máquina de estados no jerárquica. El caso peor de ejecución de la región se corresponde a aquel de sus estados cuya actividad tenga un tiempo de ejecución mayor.

R16. Debe considerarse que los cambios que se produzcan como consecuencia de las acciones realizadas en una región concurrente pueden disparar transiciones en otras regiones concurrentes.

R17. Respecto a las transiciones, debe proporcionarse un medio para animar la máquina de estados. En otras palabras, hay que enlazar el código de la máquina de estados con las fuentes de los eventos que provocan sus transiciones (cambios en el estado de los puertos, cambios en los valores de sus datos internos, errores detectados, lanzamiento de excepciones, etc.), de forma que cuando se detecte uno de estos eventos se dispare la transición correspondiente.

Una vez descritos los requisitos generales que conciernen a todos los componentes es preciso definir una serie de requisitos particulares de la aplicación que son específicos de cada componente o tipo de componente.

Requisitos particulares sobre la aplicación:

R18. Los componentes controladores de las articulaciones deben comportarse de acuerdo con las máquinas de estados del estado `FullOperative` definido en la figura 20.

R19. Las máquinas de estado de los controladores deben mostrar su estado en una vista independiente de la interfaz de usuario.

R20. La interfaz de usuario debe mostrar el estado del simulador en los plazos que se definan.

R21. El simulador debe ofrecer un comportamiento parecido al del dispositivo real, lo cual incluye al menos: habilitación y deshabilitación de ejes, referenciado de ejes y comandos de establecimiento de posición y de velocidad.

R22. El simulador debe mostrar su estado en una vista independiente de la interfaz de usuario. Este requisito es necesario para comprobar el correcto funcionamiento de los controladores y de la interfaz de usuario.

Una vez definidos los componentes hay que ensamblarlos y hacerlos funcionar en las condiciones más variadas posibles. Esto exige ciertas capacidades de parametrización que pueden concretarse en los siguientes requisitos:

R23. Deben poder fijarse desde una interfaz de usuario los periodos de las actividades.

R24. Las actividades deben poder asignarse a las tareas desde una interfaz de usuario.

R25. Deben registrarse las condiciones y los resultados de las pruebas.

Una vez establecidos todos los requisitos, mostrar los resultados obtenidos puede ayudar a comprender la solución del caso de estudio a través del relato de patrones que se describe en el siguiente apartado. La figura 26 muestra el aspecto de la aplicación. Pueden observarse dos ventanas:

- La ventana correspondiente a (1) la interfaz de usuario (**R20**, **R21**), (2) la vista de las máquinas de estado de los controladores (**R19**) y (3) la configuración de la ejecución del caso de estudio (**R23**, **R24**, **R25**).
- La ventana correspondiente a la vista del simulador que ofrece información sobre su posición y el estado de habilitación y referenciado de sus ejes (**R22**).

La ejecución se configura a través de la barra de menús y de la tabla que aparece en la parte inferior izquierda de la aplicación. Se puede configurar la política de asignación de actividades a tareas, así como los períodos y las sobrecargas de las actividades. La sobrecarga de una actividad indica el número de veces que se ejecuta un bucle interno cuyo único objetivo es aumentar el tiempo de cómputo de la actividad, con vistas a la simulación.

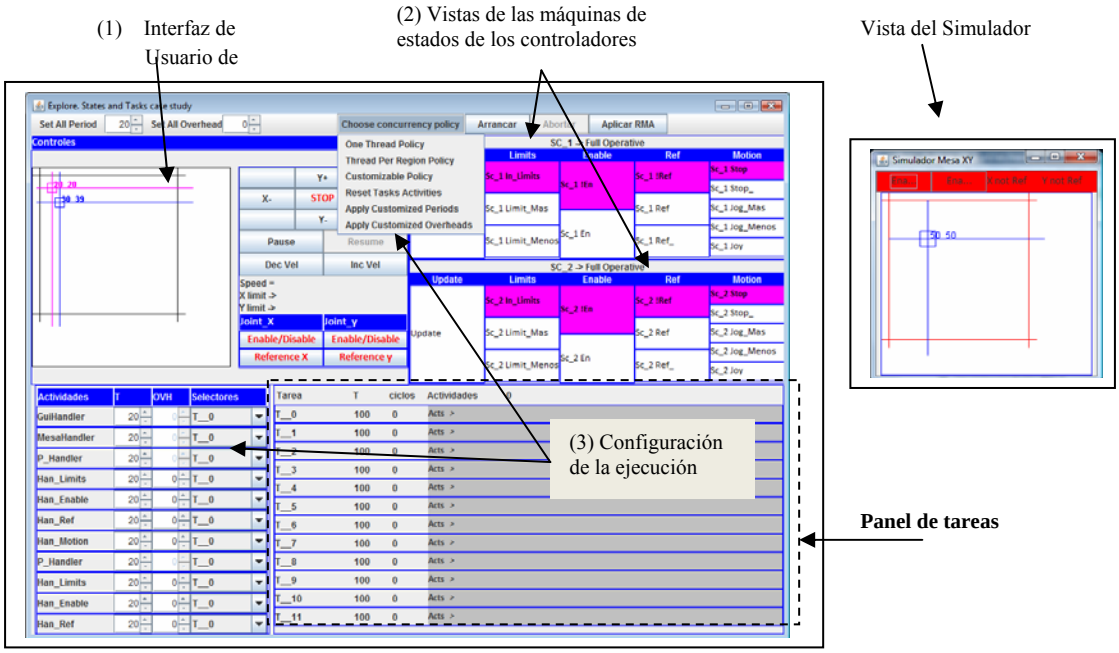


Figura 26. Ejecución del caso de estudio.

La **Tabla de Actividades** (figura 27) contiene una entrada por cada una de las actividades consideradas en el caso de estudio que, recuérdese, se corresponden con cada una de las regiones ortogonales consideradas. Puede observarse que hay doce entradas, correspondientes a:

- La máquina de estados asociada a la interfaz de usuario del simulador.
- La máquina de estados asociada al simulador.
- Las cuatro regiones del controlador del eje X y las cuatro regiones correspondientes al controlador del eje Y.
- Dos regiones adicionales añadidas por la implementación para animar las máquinas de estados de cada componente.

En cada una de las entradas se especifica el nombre de la actividad y se proporcionan controles para:

- Fijar su periodo (de 20 a 1000 ms en saltos de 20 ms).
- Fijar su sobrecarga (de 0 a 15 en saltos de 1). Como ya se ha comentado, la sobrecarga de una actividad indica el número de veces que se ejecuta un bucle interno. El *parámetro* que se especifica en la entrada no es la sobrecarga, pero sirve para calcularla:

Sobrecarga = 10^parámetro

- Fijar la tarea que debe ejecutarla. Al iniciarse la aplicación se crean tantas tareas como regiones ortogonales, de forma que pueda establecerse una política de una tarea por región.

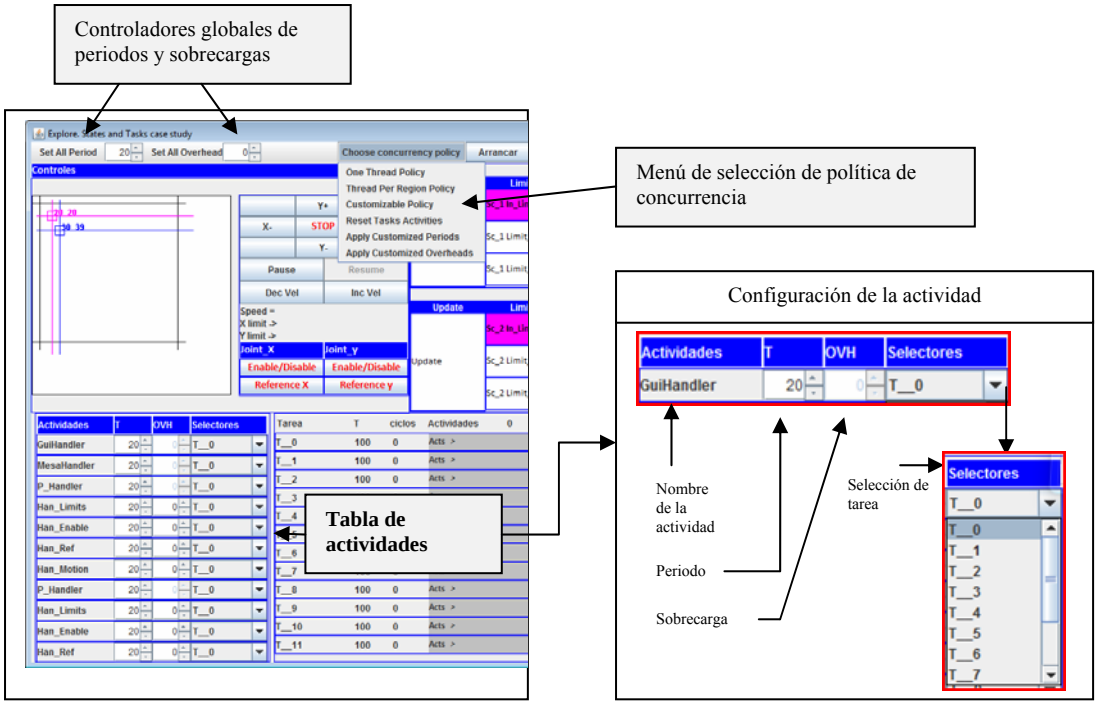


Figura 27. Controles de configuración.

Los **Controles Globales de Periodos y Sobrecargas** (figura 27) fijan el mismo periodo y la misma sobrecarga para todas las actividades. La política de actualización de periodos y sobrecargas es la misma que en la **Tabla de Actividades**, pero mientras en ésta se accede individualmente a cada actividad desde estos controles se accede a todas a un tiempo.

El **Menú de Selección de Política de Concurrencia** (figura 27) sirve para fijar el criterio general de asignación de actividades a tareas. Sus entradas son las siguientes:

- **One Thread Policy**. Todas las actividades se asignan a la tarea 0. Por defecto se toma el periodo y la sobrecarga definidos en los **Controles Globales de Periodos y Sobrecargas.**

- **Thread Per Region Policy**. Cada actividad se asigna a una tarea diferente. Por defecto se toma el periodo y la sobrecarga definidos en los **Controles Globales de Periodos y Sobrecargas.** *Esta es la política por defecto.*

- **Customizable Policy**. Las actividades se asignan a las tareas seleccionadas en la **Tabla de Actividades**.

- **Reset Taks Activities**. Deshace la asignación de actividades y establece **One Thread Policy** (es un control redundante y eliminable)**.**

- **Apply Customized Periods**. Fuerza a que se tomen los periodos de la **Tabla de Actividades** independientemente de la política de concurrencia seleccionada. Se ignora el **Controlador Global de Periodo.**

- **Apply Customized Overheads**. Fuerza a que se tomen las sobrecargas de la **Tabla de Actividades** independientemente de la política de concurrencia seleccionada. Se ignora el **Controlador Global de Sobrecarga.**

La interfaz de usuario del simulador.

En la figura 28 se nombran y muestran los controles de la interfaz de usuario que se explican a continuación.

- El **Panel de Posición** muestra la posición actual de la mesa y sirve también de control para introducir mediante el ratón comandos de movimiento absoluto a una posición (x,y). Estos comandos sólo se atienden si el eje correspondiente está habilitado y referenciado.
- En el **Panel de Control y Estado** se distinguen los siguientes elementos:
  - *Botones de movimiento en modo jog*: movimientos individuales en los ejes X e Y en sentidos positivo y negativo: Botones **X+**, **X-**, **Y+** e **Y-**. Estos comandos sólo se atienden si el eje correspondiente está habilitado.
  - *Botones de pausa y reanudación de comando de movimiento a posición (x,y)*: El botón **Pause** detiene el movimiento en curso y el botón **Resume** lo reanuda. Cuando uno está habilitado, el otro está deshabilitado.

- *Etiquetas con información sobre la velocidad comandada y el estado de los finales de carrera.*
- *Botones de habilitación/deshabilitación y referenciado de los ejes X e Y:* Botones **Enable/Disable** y **Reference**. Los botones indican mediante el color de su leyenda el estado correspondiente (rojo: no habilitado, no referenciado, negro: habilitado, referenciado).

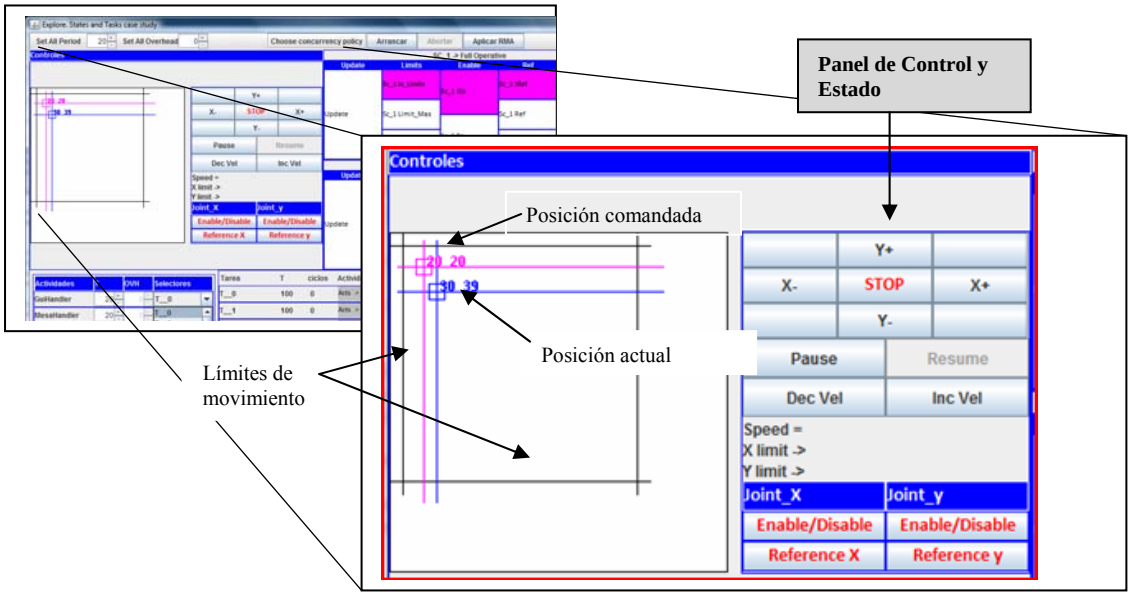


Figura 28. Interfaz de usuario del simulador.

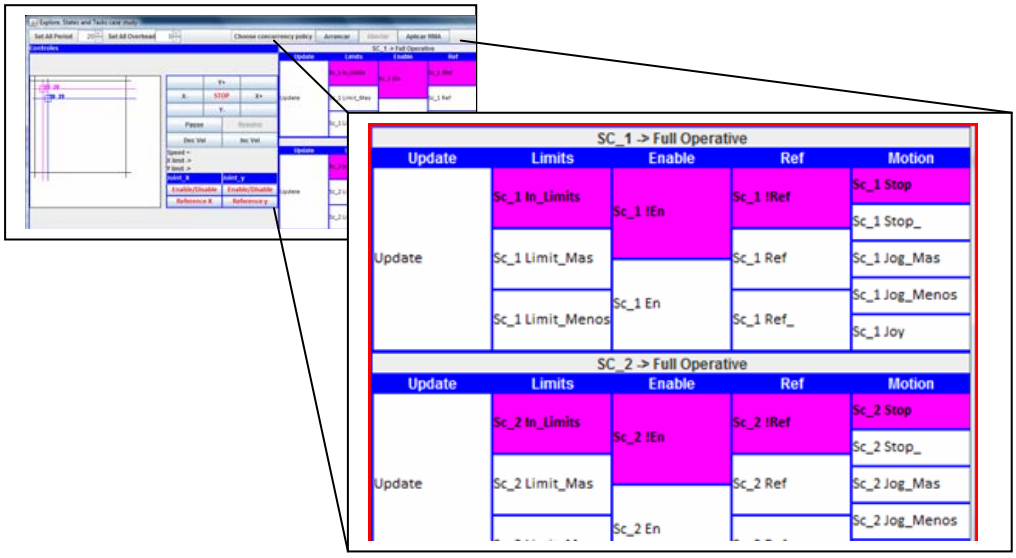


Figura 29. Vistas de las máquinas de estados de los controladores

Vistas de las Máquinas de Estado de los Controladores.

En la figura 29 se nombran y muestran los elementos de las vistas de los estados de los controladores. No hay mucho que comentar ya que se corresponden directamente con los estados definidos en la figura 20. El color magenta indica el estado en que se encuentra la región.

Arranque y parada del caso de estudio.

Para arrancar el caso de estudio simplemente hay que pulsar el botón Arrancar de la barra de menú y para pararla hay que pulsar el botón Abortar. El caso de estudio se ejecutará según la configuración elegida.

La aplicación todavía no es robusta. Se trata de un *trabajo en marcha* en el que va a ser necesario:

- Incluir manejadores de excepciones.
- Mejorar la estructura de las vistas y de los manejadores de eventos.
- Añadir nuevos controles y dar funcionalidad a los que no la tienen (botón aplicar RMA de la barra de herramientas).

Además, el nombre de las actividades no refleja el componente al que pertenecen, lo cual dificulta el seguimiento de la ejecución en el **Panel de Tareas**, y no hay registro de resultados, aunque se muestra el comportamiento de las tareas *on-line* en dicho panel.

5.2. SOLUCIÓN DEL CASO DE ESTUDIO: UN RELATO DE PATRONES.

El objetivo de la historia de patrones que se describe a continuación no es dar una solución definitiva a los problemas que se plantean, sino comenzar a razonar sobre los mismos en un estilo dirigido por patrones. Dicha historia se refiere a un sistema particular por lo que no constituye todavía una secuencia de patrones o un lenguaje de patrones tal y como se definen en [Buschmann, 2007], pero en cualquier caso constituye un paso previo necesario para su definición. Para dar una solución se ha seguido la hoja de ruta que se describe en la figura 30:

(1) El primer asunto que se abordó fue el reparto de las actividades en diferentes tareas ya que el principal problema es repartir las actividades en tareas según los requisitos **R1** a **R7**.

(2) En segundo lugar se modelaron los puertos y las comunicaciones entre componentes de acuerdo con las consideraciones y requisitos **R8** a **R12** del caso de estudio. Los puntos (1) y (2) son bastante independientes y se abordaron simultáneamente.

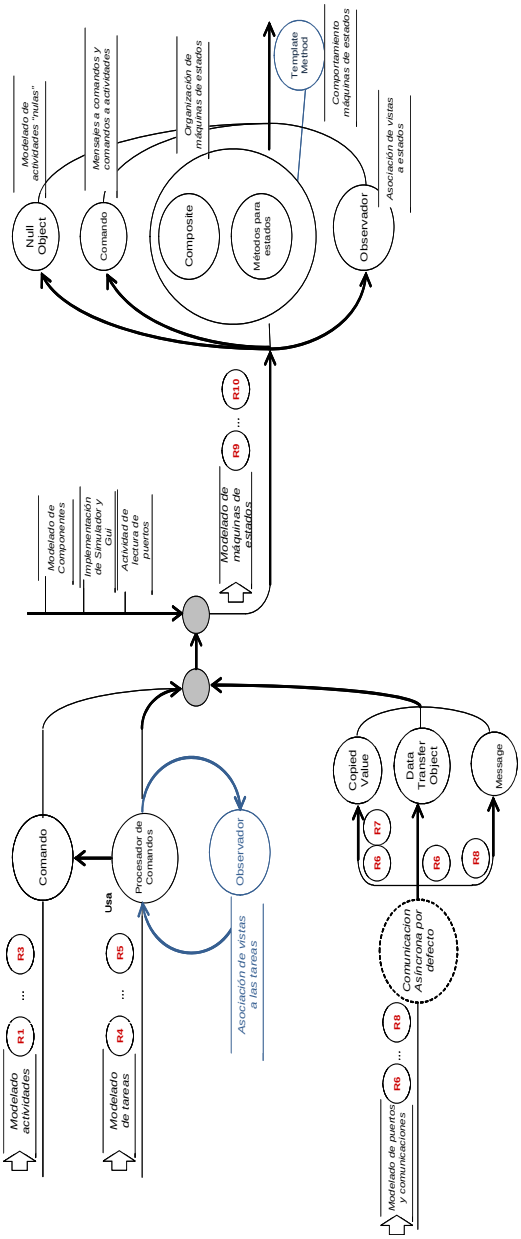


Figura 30. Hoja de ruta. Aplicación sucesiva de patrones.

(3) Seguidamente se definieron los componentes correspondientes al simulador y a la GUI y se les asignaron puertos, una máquina de estados para gestionarlos y vistas según los requisitos **R18** a **R23**.

(4) A continuación se implementaron las máquinas de estados de los componentes para proporcionar un "*andamiaje*" suficiente para implementar el caso de estudio. Este *andamiaje* debía estar de acuerdo por un lado con los requisitos **R13** a **R17** y por otro con las soluciones adoptadas respecto a actividades, tareas y puertos.

(5) Finalmente se ensamblaron todas las piezas para hacerlas funcionar en las condiciones más variadas posibles (requisitos **R24** a **R26**).

5.2.1. Modelado de actividades y tareas.

De acuerdo con los requisitos **R1** a **R7** las actividades asociadas a los estados se van a modelar como comandos y las tareas que las ejecutan se van a modelar como procesadores de comandos. Para ello se va a aplicar respectivamente los patrones **Comando** [Gamma, 1995] y **Procesador de Comandos** [Buschmann, 2007].

Modelado de las actividades. Patrón Comando.

Una forma de cumplir el requisito **R1** es modelar las actividades como objetos. Para ello se va a aplicar el patrón **Comando**. Dicho patrón se usa en los programas OO para tratar a los comandos como objetos, de forma que cliente y servidor puedan ejecutarse de forma asíncrona. El patrón **Comando** no proporciona por sí mismo dicha asincronía, pero es necesario como paso previo para usar los patrones que la proporcionan, en nuestro caso el **Procesador de Comandos**. El patrón **Comando** requiere a su vez el empleo previo del patrón **Interfaz Explícita** [Buschmann, 2007] ya que es necesario disponer de una interfaz uniforme para ejecutar y gestionar los comandos. Este último, no es más que la aplicación del principio de diseño OO de "definición de interfaces". La interfaz definida para modelar las actividades de los estados es `StateActivity` (figura 31).

```
public interface StateActivity {
    void executeTick(); //ejecuta el código asociado a la actividad
    boolean isDone();  //indica que la actividad ha terminado
    void setPeriod(int T); //permite cambiar el periodo para pruebas
    int getPeriod(); //devuelve el periodo de la actividad
    void setName(String name); // para depuración
    String getName(); //para depuración
}
```

Figura 31. La interfaz `StateActivity`.

Modelado de las tareas. Patrón Procesador de Comandos.

Las tareas se van a modelar de acuerdo con el patrón **Procesador de Comandos**. Este patrón separa la petición de un servicio de su ejecución. Para ello, el Procesador de Comandos gestiona las peticiones de servicio como objetos independientes (**R5**), planifica su ejecución (**R6**) y proporciona, si es necesario, servicios adicionales como *logging* o encolado de peticiones. De todas estas posibilidades sólo estamos interesados en las dos primeras: gestión de las peticiones de servicio como objetos independientes y planificación de su ejecución. Este patrón asume que las actividades se modelan como objetos, lo cual supone que previamente se han aplicado los patrones **Interfaz Explícita** y **Comando**. Las tareas se modelan mediante la clase `ActivityProcessor` que se muestra en la figura 32.

Vistas de las tareas. Patrón Observador.

Un problema importante de cara tanto a la prueba y depuración de las tareas como de cara a la visualización de "experimentos" es la visualización de las actividades asignadas a las tareas y su ejecución. Con objeto de desacoplar las tareas y sus vistas en la mayor medida posible se ha optado por establecer entre ambos una relación *sujeto-observador* con las tareas en el rol de sujeto y las vistas en el rol de observadores. Para ello se ha definido una interfaz `ActivityProcessorListener` que implementan las vistas y que define una serie de métodos que se invocan en el observador cuando se añade o se elimina una actividad de la tarea, en cada nueva ejecución de la tarea o cuando cambia su periodo.

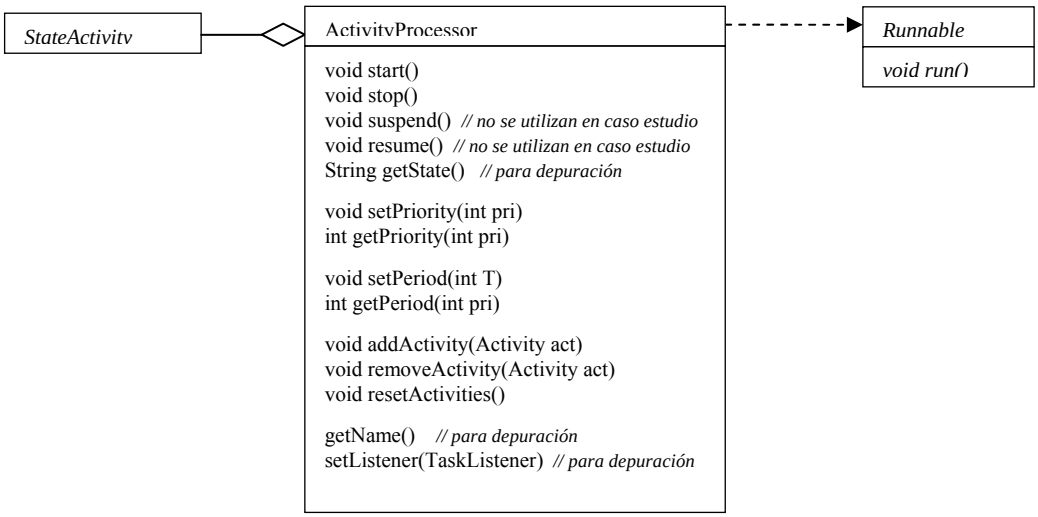


Figura 32. La clase `ActivityProcessor`.

Discusión.

1. Restricciones sobre el uso de las tareas como procesadores de comandos.

Las tareas como procesadores de comandos ofrecen una gran versatilidad. No imponen restricciones respecto a cuándo se pueden suscribir o eliminar las actividades, ni sobre su número ni duración. Sin embargo, la implementación debe mantener cierta disciplina para cumplir con los requisitos del caso de estudio:

- Sólo se suscriben o eliminan actividades durante la inicialización o configuración de la aplicación, pero nunca en tiempo de ejecución. Esto es necesario para poder aplicar RMS.
- Las actividades se deben implementar de forma que su ejecución sea lo más breve posible. Si hay que ejecutar un algoritmo con un número indeterminado de iteraciones o una actividad de control que no cesa hasta que se alcanza un objetivo, se divide la actividad en sub-actividades de duración acotable, por ejemplo un paso del algoritmo, una acción de control discreta, como calcular el error en el instante actual y generar la consigna para el instante siguiente, etc. Cada tarea es, vista aisladamente, una especie de ejecutivo cíclico en el que hay que encajar las actividades.

2. La planificación de las actividades en las tareas debe ser mejorada.

Actualmente la tarea calcula su periodo como el mínimo de los periodos de las actividades que tiene asignadas. Todas las actividades se ejecutan con este periodo. Hay que mejorar la implementación de forma que las actividades se agrupen de acuerdo con sus periodos y se ejecuten cuando les toque. Podría incluso extenderse la interfaz de la tarea de forma que se le pudieran asignar diferentes políticas de planificación de sus actividades.

5.2.2. Modelado de las comunicaciones. Puertos, mensajes e interfaces.

De acuerdo con los requisitos **R8** a **R12** los puertos se van a modelar con los siguientes criterios:

1. Los puertos se van a definir mediante clases. La clase del puerto se determina teniendo en cuenta si el puerto es de entrada o salida y el tipo de los datos que se leen o escriben en el puerto.

2. Las interfaces asociadas a los puertos se van a modelar como objetos que se intercambian a través de los puertos.

3. En los puertos de salida de datos el componente deja un objeto y en los puertos de entrada de datos el componente recoge un objeto.

4. En los puertos de entrada de datos un dato nuevo sobre-escribe al antiguo.

5. En los puertos de entrada de comandos éstos se encolan hasta que el componente los retire.

6. Cuando un puerto de salida está conectado a un puerto de entrada, cada vez que el componente deja un dato en el puerto de salida, éste actualiza inmediatamente al puerto de entrada conectado a él. El puerto de entrada es un *observador* del puerto de salida.

7. La semántica por defecto de los puertos va a ser comunicación asíncrona sin respuesta.

Con este enfoque pueden cumplirse directamente los requisitos **R9** y **R10**, se facilita el cumplimiento de **R11** y sirve de base, con extensiones del código, pero sin modificaciones, para cumplir con el requisito **R8**.

Modelado de los puertos. Patrones: Paso por Copia, Objeto de Transferencia de Datos y Mensaje.

Para la realización del caso de estudio se han definido cuatro tipos de puerto implementados en clases: entrada de comando, salida de comando, entrada de datos y salida de datos (véase figura 33).

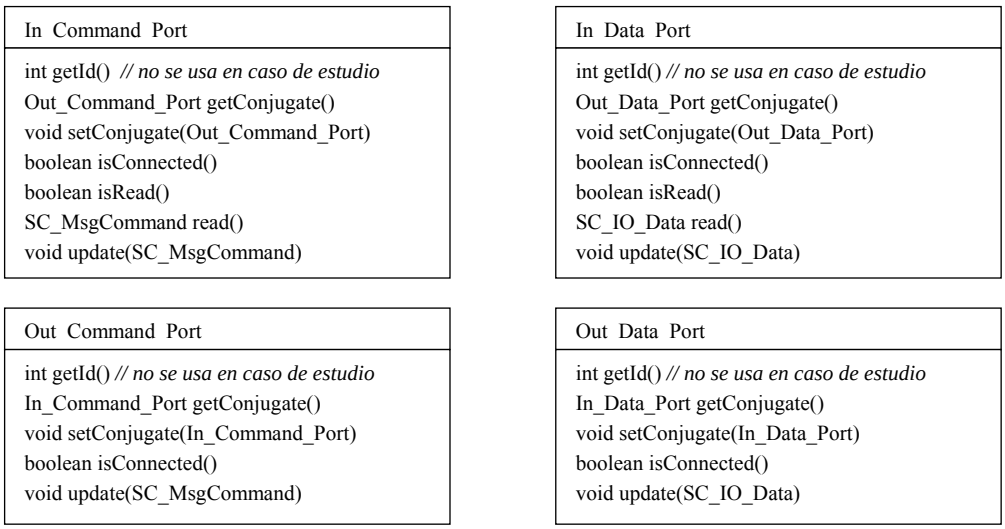


Figura 33. Puertos de entrada y salida.

Una de las decisiones de diseño más importantes es que la información entre componentes siempre se pasa por copia. Los objetos nunca comparten referencias a datos comunes. El acceso a datos comunes en un sistema

concurrente implica sincronizar los accesos mediante el empleo de estructuras tales como semáforos, objetos protegidos y monitores. La programación de tales recursos no siempre es fácil e introduce problemas para el análisis temporal. Además, suele afectar al rendimiento ya que los accesos sincronizados bloquean los recursos. El objetivo, es, por tanto, minimizar el uso de tales construcciones.

Cuando introducimos un dato en un puerto de salida el argumento que se pasa al método invocado por el componente es una referencia a un objeto. Esa referencia no puede ser transmitida sin más a otro componente, puesto que tendríamos a los componentes emisor y receptor accediendo a un mismo objeto del componente emisor. En lugar de eso, se crea una copia del objeto y se pasa al receptor una referencia de la copia. A partir de ahí, emisor y receptor trabajan con objetos diferentes y no es necesario establecer ningún sincronismo entre ambos. Esto es simplemente una aplicación del patrón Paso por Copia o ***Copied Valued***.

El patrón Objeto de Transferencia de Datos o ***Data Transfer Object*** reduce el número de llamadas de consulta o modificación de datos entre objetos empaquetando grupos de atributos en un solo objeto que se pasa o se retorna en una única llamada. En nuestro caso todos los datos que se intercambian en los puertos de datos se han encapsulado dentro de la clase SC_IO_Data (figura 34) que representa el estado del simulador.

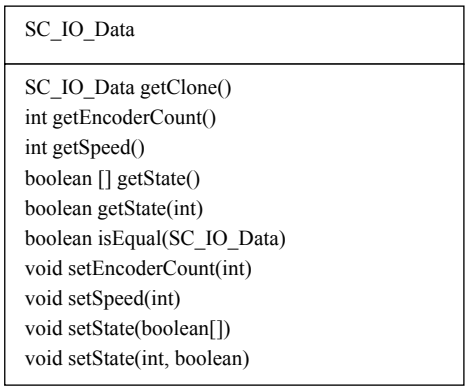


Figura 34. Modelado de los datos intercambiados.

Como en el caso de los datos, para los comandos también se ha optado por enviar un objeto que represente al comando en lugar de invocar un método. En este caso el propósito es, sin embargo, diferente y el patrón también. *No se trata de minimizar el número de mensajes trasegados entre componentes, sino de proporcionar semántica asíncrona y facilitar la futura distribución de los componentes*. El cliente pone en el puerto de salida un mensaje que representa al comando. A partir de ahí puede inspeccionar los datos que actualiza el servidor para ver si todo va bien o desentenderse por completo.

Puede incluso detenerse (sincronizarse) hasta obtener una respuesta. Pero es siempre una decisión del componente que envía el comando, no una imposición de la semántica de la llamada. Los mensajes se modelan mediante la clase `SC_MsgCommand` (figura 35). El uso del patrón Mensaje o ***Message*** exige que en el receptor el mensaje sea primero "decodificado" y después ejecutado, lo cual puede hacerse mediante una combinación de los patrones Comando y Procesador de Comando.

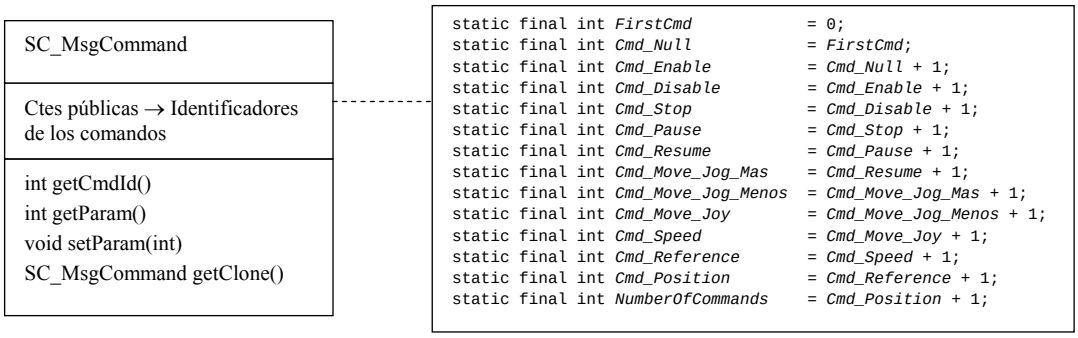


Figura 35. Modelado de mensajes comando.

Conexión de los puertos.

Con objeto de poder conectar y desconectar los puertos durante la configuración, se define una clase de utilidad con métodos estáticos `PortConnections` que mantiene estructuras de datos con todos los puertos de la aplicación y proporciona métodos para conectar los diferentes tipos de puertos.

Discusión.

1. Implementación de semántica petición/respuesta síncrona.

La semántica de los puertos es por defecto petición/respuesta asíncrona. Si a pesar de todo el cliente desea realizar llamadas síncronas, éstas pueden implementarse a partir de las asíncronas. Para ello podría definirse una región de la máquina de estados del componente cliente que esperara en un puerto de entrada el resultado de una petición previa. El bloqueo de un componente en una llamada síncrona no es en general aceptable, pero puede admitirse el bloqueo de una región de la máquina de estados del componente en un estado de espera.

2. Asociación de interfaces a los puertos al estilo V^3CMM.

Bastaría con *decorar* la implementación actual con una *fachada* que expusiera los métodos de las interfaces. El puerto decorador contendría un puerto de los que se han definido en el caso de estudio e implementaría la interfaz correspondiente.

3. Posibles pérdidas de comandos.

A pesar de que el requisito **R9** se establece que no pueden perderse datos, no se implementan colas en los puertos de entrada de comandos. Esto puede producir la sobre-escritura de un comando por otro, ya que entre dos lecturas del puerto puede haber más de una escritura de comando. Solucionar este problema requiere: (1) implementar colas de comandos en los puertos de entrada de comandos, (2) recoger todos los comandos en la operación de lectura y (3) decidir qué comandos se ejecutan y cuáles se ignoran.

4. Sincronización de acceso a los puertos.

En el acceso a los puertos hay exclusión mutua, pero no sincronismo. En principio no parece necesario incluir dicho sincronismo, pero si se hace podría parametrizarse la estrategia de sincronización (patrón ***Strategized Locking*** [Buschmann, 2007]).

5. Gestión global de los puertos.

No se realiza ninguna gestión global de los puertos de los componentes. En tiempo de ejecución cada componente gestiona sus puertos, pero en tiempo de inicialización y configuración puede ser necesario iterar sobre los puertos para comprobar las conexiones y hacerlas o rehacerlas. Puede ser necesario definir estructuras de datos e iteradores para manejar los puertos y sus conexiones.

6. Caducidad de los datos.

La utilización de copias introduce el problema del lapso temporal en el que la copia es válida. Se supone que los periodos de las actividades deben ajustarse para que los datos ofrecidos en los puertos sean válidos y reflejen el estado actual del componente.

5.2.3. Modelado de las máquinas de estados.

No se pretende dar una solución general a la implementación de las máquinas de estado de V^3CMM, sino proporcionar una solución que cumpla los requisitos **R13** a **R17**. Aun así, la implementación de las máquinas de estado del caso de estudio sigue siendo bastante complejo, por lo que en este caso merece la pena comenzar comentando las soluciones que nos ofrecen los patrones. En [Buschmann, 2007] se ofrecen patrones para el modelado de objetos cuyo comportamiento es función de su estado:

- **Patrón Objetos para Estados**: se encapsula el comportamiento dependiente del estado en una jerarquía de *clases estado*, cada una de las cuales modela un comportamiento diferente. Se usa una instancia de la clase apropiada en función del estado y cuando un cliente invoca un método, la llamada se delega en el *objeto estado*.

- **Patrón Métodos para Estados**: se implementa el comportamiento dependiente del estado en *métodos internos del objeto* y se definen *estructuras de datos internas para referenciar a dichos métodos* en función del estado. Cuando un cliente invoca un método se busca el método apropiado en el que delegar consultando la estructura de datos interna (puede ser una tabla) que representa al estado actual. Al crear al objeto se le asocia con la estructura de datos que representa a su estado inicial. Cuando cambia el estado, se cambian los valores referenciados en la estructura de datos para que el comportamiento sea el definido en el nuevo estado.

Dados los problemas de explosión de estados que se dan con una máquina de estados convencional, el estado **`FullOperative`** se modela mediante regiones ortogonales (figura 20). Esta forma de proceder escala bastante bien, ya que el estado de otras regiones se hace presente en las demás en las condiciones asociadas a sus transiciones. Si se añade una nueva región habrá que revisar cómo afectan sus estados a las transiciones de las regiones preexistentes, pero no se produce una explosión de estados. No obstante, al proceder así nos estamos alejando de los patrones para modelar el comportamiento modal de los componentes. Cada región de **`FullOperative`** representa un aspecto del comportamiento del componente, no un comportamiento diferente. Los patrones asumen que dependiendo del estado cambia la implementación de los métodos de la interfaz del objeto, pero en nuestro caso no es así: dentro del estado **`FullOperative`** los comandos siempre se ejecutan igual, lo que ocurre es que sólo se ejecutan en ciertas regiones.

En nuestro caso de estudio, cada región maneja sólo determinados comandos, y cada comando sólo se maneja en una única región (esto no es generalizable ni para otros casos de estudio, ni para otros tipos de eventos). Ocurre, además, que dentro de cada región el comportamiento es modal, es decir, los comandos gestionados en esa región se comportan de una forma u otra en función del

estado de la región. La diferencia de comportamiento es todo o nada: están permitidos y se ejecutan o no están permitidos y se ignoran. Aún así, *la estructura de las regiones nos ofrece una nueva oportunidad para aplicar los patrones de diseño de máquinas de estados, aplicándolos no al componente como un todo, sino al diseño de cada región*. En este caso, se ha utilizado el patrón Métodos para Estados ***(Methods for States).*** El empleo del patrón se ha visto condicionado por los siguientes factores:

- El objeto modal es la región ortogonal.
- El estado de cada región y los datos de los puertos de entrada deben ser visibles desde todas las regiones del diagrama de estados del componente.
- La naturaleza jerárquica (limitada) de las máquinas de estados consideradas.
- La necesidad de integrar las operaciones de lectura y escritura de los puertos dentro del comportamiento del componente y de establecer correspondencias entre los mensajes recibidos en los puertos, los comandos que deben ejecutarse y las actividades asociadas a los estados.

Datos de estado globales.

El requisito **R16** establece que los cambios que se produzcan como consecuencia de las acciones realizadas en una región concurrente pueden disparar transiciones en otras regiones concurrentes. Para que esto sea posible una región debe saber los cambios que se han producido en las demás. En el caso de estudio, las transiciones de las máquinas de estado de las regiones dependen de:

- El estado de otras regiones.
- Los datos de estado leídos de la articulación controlada.
- El comando recibido.

La forma más directa y sencilla de conseguir que estos datos sean accesibles a todas las regiones es mediante la definición de una estructura de datos global a la máquina de estados del componente, accesible para su actualización y lectura desde todas las regiones. Ese es el enfoque que se ha seguido en el caso de estudio, donde la clase `SC_Data` encapsula toda la información del componente (figura 36).

Mensajes, actividades y comandos.

Los mensajes de comandos se modelan mediante la clase `SC_MsgCommand` pero dicha clase sólo modela los datos del comando y no dice nada acerca de cómo debe ejecutarse. Acabamos de ver que cada región ortogonal maneja determinados comandos y que cada comando se maneja en una única región. Sabemos además que cada estado tiene asociada una actividad que determina

lo que hay que hacer en ese estado. Parece entonces inmediato asociar los mensajes de comandos con las actividades de los estados, sin embargo:

- Hay comandos cuya ejecución no está asociada a ningún estado y por tanto a ninguna actividad (por ejemplo, el comando para fijar la velocidad). En este caso, el componente controlador redirige el comando hacia el dispositivo sin realizar ninguna acción previa de control.
- Hay actividades que no tienen que ver con la ejecución de ningún comando, sino con la gestión de los estados, con la monitorización de alguna variable de estado (estados de `Rg_Limits`) o con la actualización de los puertos (estados de la región `Update`).
- La asociación actividad-comando es demasiado rígida.

La solución que se va a adoptar es modelar la actividad asociada a los comandos por separado y asignársela *después* a la actividad del estado correspondiente, el cual implementa su actividad delegando en el comando. Para ello vamos a volver a aplicar el patrón Comando definiendo una clase `SC_Command` que hereda de `SC_MsgCOmmand` y la extiende con dos nuevos métodos `executeTick` (contiene el código asociado a la ejecución del comando) y `isDone` (devuelve true si se ha terminado de ejecutar el comando). Cada instancia de la clase modela el comportamiento correspondiente a un comando determinado. Una de las instancias responde a la aplicación del patrón Objeto Nulo o ***Null Object***. Se trata de modelar la ausencia de comando o de actividad como un caso particular de comando que no hace nada. Esta forma de proceder nos permite definir un comportamiento por defecto y evita bastante código condicional.

Animando la máquina.

El requisito **R17** establece que hay que integrar las operaciones de lectura y escritura de los puertos dentro de la máquina de estados del componente. La solución adoptada va a ser añadir una región (`PortHandling`) más a la máquina de estados del componente (figura 37) cuya actividad consiste en:

- Leer los puertos de entrada: datos y comandos.
- Actualizar los datos globales si es necesario.
- Actualizar los puertos de salida de datos.
- Actualizar los puertos de salida de comandos en el caso de los comandos que simplemente son redirigidos hacia el dispositivo destino. En el resto de los casos (que son la mayoría) los puertos de comandos son escritos desde las actividades asociadas a los estados donde se controla su ejecución.

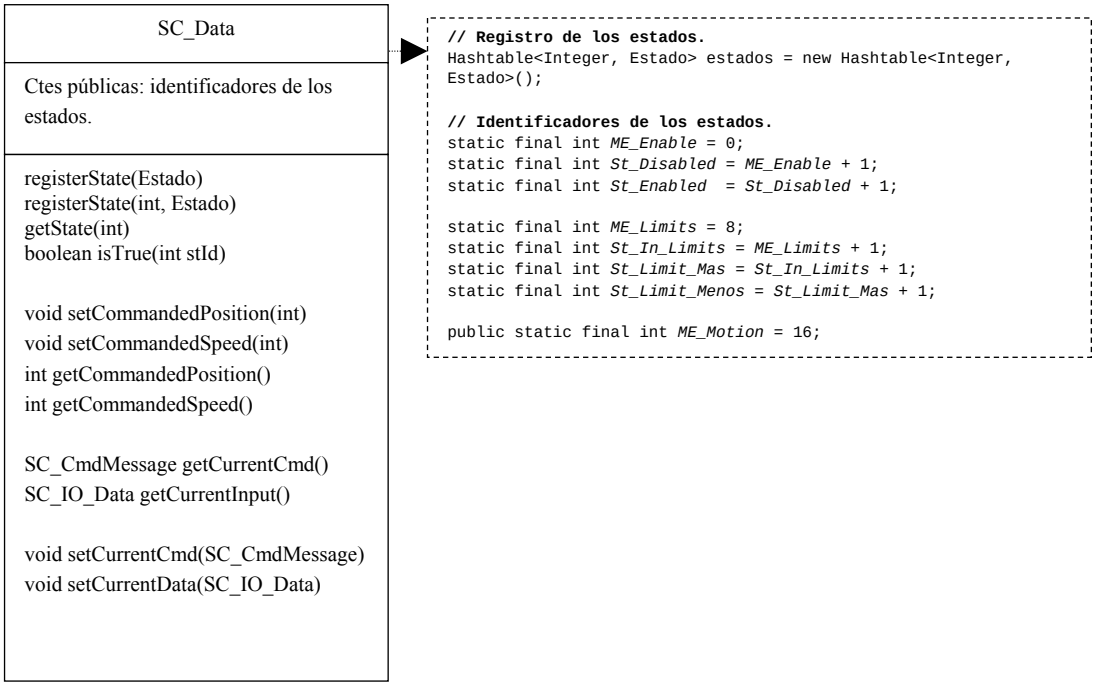


Figura 36. Modelado de datos del componente controlador. Clase `SC_Data`.

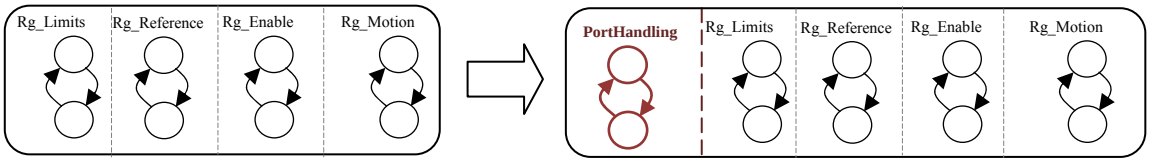


Figura 37. Región ortogonal para gestión de los puertos

Esta solución, que no se basa en la aplicación (consciente) de ningún patrón de diseño, ofrece las siguientes ventajas:

- La nueva región está encapsulada dentro del componente al que pertenece la máquina de estados.
- No supone ninguna "irregularidad" respecto de lo ya visto. La región añadida es una región más con una máquina de estados como las demás, aunque con una responsabilidad diferente. A la actividad asociada, como a todas las demás, se le puede fijar un periodo de ejecución y asignársela a una tarea para que la ejecute. Lo normal es que ésta sea la actividad más frecuente.
- Es un enfoque flexible:

- Permite la definición de diferentes políticas de gestión de las máquinas de estados, tan sencillas o complejas como se necesite.
- Pueden añadirse regiones específicas más rápidas asociadas a lecturas o actualizaciones con periodos muy cortos, permitiendo así planificaciones de granularidad más fina.
- Pueden añadirse regiones con propósitos específicos (p.e.: esperar la llegada de la respuesta a un servicio en un puerto determinado).

Modelado de los estados. Patrón *Composite*.

Las máquinas de estados de V^{3}CMM son máquinas de estados jerárquicas. Cada estado puede descomponerse a su vez en varias máquinas de estados. En nuestro caso vamos a considerar un estado (**`FullOperative`**) compuesto por cinco regiones ortogonales (las cuatro originales más una extra para gestionar los puertos, figura 37), cada una de las cuales contiene a su vez una máquina de estados no jerárquica. Cuando se trabaja con máquinas de estados jerárquicas es difícil diferenciar entre la máquina y los estados, ya que todo estado es en potencia una o varias máquina de estados. En estas circunstancias es muy conveniente poder manipular a las máquinas de estado y a los estados de la misma manera. Para ello contamos con el patrón ***Composite*** que ofrece muchas posibilidades para el modelado de las máquinas de estados jerárquicas. Dichas posibilidades apenas se explotan en el caso de estudio donde la utilidad del patrón se va a limitar casi exclusivamente a dos aspectos: (1) la extracción y gestión de las actividades asociadas a estados simples y regiones, y (2) la gestión de las vistas asociadas a estados y regiones. Además, el patrón se emplea de forma simplificada y restringida ya que los *estados jerárquicos* sólo pueden contener *estados hoja* que a su vez no pueden contener a otros estados (figura 38). Dadas las simplificaciones del caso de estudio, el uso del término *jerárquico* se vuelve incongruente: *dentro del ámbito del caso de estudio* las regiones son máquinas de estados *no* jerárquicas, sin embargo se ha preferido conservar el término ya que, *dentro de los objetivos del proyecto EXPLORE,* las regiones podrán ser máquinas de estados jerárquicas.

Los estados se modelan mediante la clase abstracta `State` (figura 38), de la cual derivan todas las clases que modelan estados o máquinas de estados. Dicha clase sólo define un método abstracto (`resolveNextState`) proporcionando una implementación para el resto. Sus métodos pueden clasificarse en tres grupos:

- Implementación de las actividades, estados y transiciones de la máquina de estados.
- Obtención de la actividad asociada a los estados.
- Representación del estado y asignación de vistas.

El método abstracto `resolveNextState` determina el siguiente estado en función de los eventos y condiciones que se han producido en el intervalo definido por dos ejecuciones sucesivas de la actividad. Si el siguiente estado es distinto del actual, la implementación de la máquina de estados se encarga de llevar a cabo la transición. Este método no produce ninguna transición, sino que indica la transición que debe producirse, siendo la implementación de este método responsabilidad de las subclases.

Los estados simples, aquellos que no se descomponen a su vez en una máquina de estados, se modelan mediante la clase abstracta `LeafState`. Se han definido tantas subclases concretas de `LeafState` como estados simples aparecen en las máquinas de estados de las regiones de **`FullOperative`**.

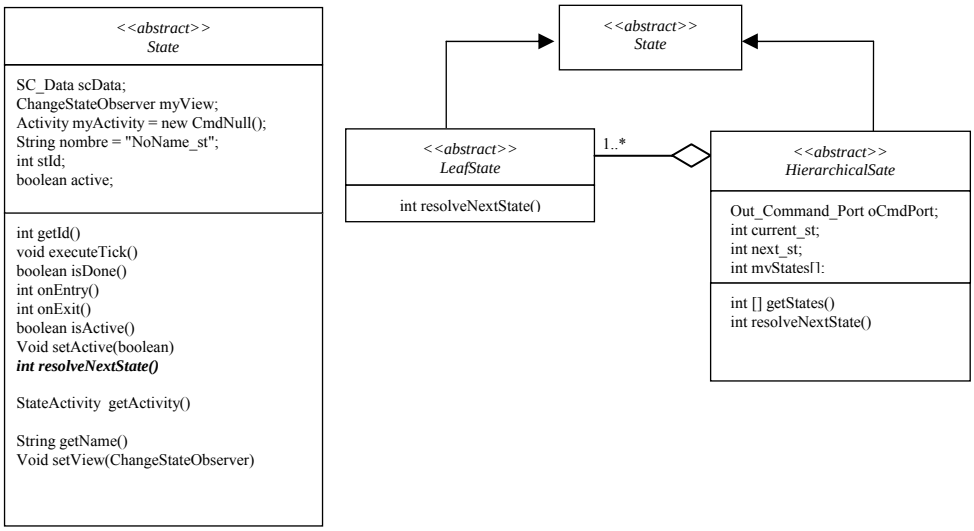


Figura 38. La clase `State` y sus subclases.

Los estados que a su vez representan máquinas de estados, se modelan mediante la clase abstracta `HierarchicalState` (figura 39). Dicha clase define nuevas variables de instancia, para gestionar sus estados hoja y pasarles información, y añade el método `getStates` para obtener los identificadores de los estados hoja contenidos en la máquina modelada por el estado jerárquico.

Se ha definido una subclase concreta de `HierarchicalState` por cada máquina de estados de `FullOPerative`. Ninguna de estas clases define nuevos métodos ni sobre-escribe los que hereda de sus superclases. Sin embargo, la clase `HierarchicalState` define su actividad para gestionar su máquina de estados en una clase interna `St_Machine_Handler`.

La clase `St_Machine_Handler` implementa una actividad común para todas las máquinas de estados. Esta actividad se encarga de: (1) ejecutar la actividad del estado en que se encuentra la región, (2) determinar si hay que realizar una transición y, (3) realizar, si es necesario, la transición e invocar los métodos `onExit` y `onEntry` de los estados saliente y entrante. El método `executeTick` de `St_Machine_Handler` se ha planteado según el patrón Método Plantilla ***(Template Method).*** Según este patrón se crea una clase (en este caso `St_Machine_Handler`) que proporciona un método plantilla (`executeTick`) que define un comportamiento común. Dentro de este método se delega la ejecución de las acciones en objetos que las implementan de diferentes formas.

Vistas para los estados.

Un problema importante de cara tanto a la prueba y depuración de las máquinas de estados y de los algoritmos de control que encapsulan es la visualización del "*estado de las máquinas de estado*" (**R19**). Con objeto de desacoplar los estados y sus vistas en la mayor medida posible se ha optado por establecer entre ambos una relación *sujeto-observador* con los estados en el rol de sujeto y las vistas en el rol de observadores. Para ello se han definido dos interfaces: (1) la primera la implementan las vistas y define un solo método ya que en el caso de estudio lo único que nos interesa de un estado es saber si la máquina de estados a la que pertenece se encuentra en él, y (2) la segunda, que implementan los estados define también un solo método que sirve para suscribir la vista a su estado.

Discusión.

1. Reducir la Sincronización.

Una opción que permitiría reducir la necesidad de sincronización hasta casi eliminarla y reducir las dependencias entre estados sería particionar adecuadamente la interfaz de la estructura de datos globales y ofrecer a cada objeto sólo la parte que necesita. Se puede estudiar el patrón ***Thread Specific Storage*** [Schmidt, 2000] por si pudiera ser de aplicación para estructurar los datos de los componentes en relación a los hilos.

2. Inicialiación de la estructura de datos global.

Un punto especialmente importante y sobre el que se pasa casi de largo en el caso de estudio es la inicialización de la estructura de datos `SC_Data` asociada al componente, en especial en lo que se refiere al registro de los estados de las regiones. Habría que definir un procedimiento general que pudiera encapsularse en un método plantilla.

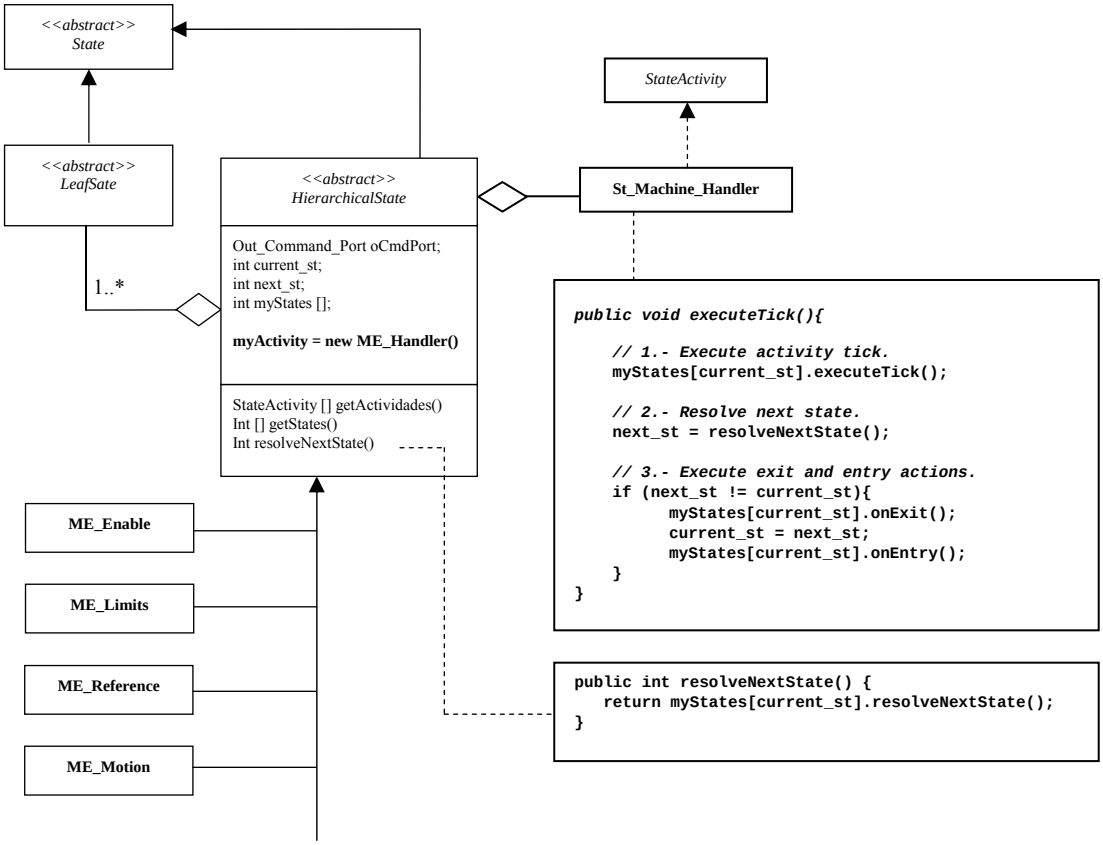


Figura 39. `HierarchicalState` y sus subclases.

5.2.4. Modelado de los componentes.

Hasta este punto el relato ha seguido un enfoque ascendente, definiendo los elementos del componente y sus relaciones. Así, se han definido las actividades, los estados y las máquinas de estados, los datos globales del componente, los puertos y sus conexiones. Ha llegado el momento de integrarlo. Para ello se ha definido la interfaz `V3Component` que no tiene más pretensiones que modelar los componentes con vistas al caso de estudio. La interfaz contiene dos métodos que devuelven el nombre del componente y las actividades asociadas a los estados hoja de su máquina de estados.

El caso de estudio ofrece tres implementaciones de dicha interfaz:

- Una clase que modela un controlador simple (lo que en terminología ACRoSeT se denomina SC, *simple controller*). El componente recibe

los puertos en su constructor y crea instancias correspondientes a su identificación, a sus máquinas de estados y a sus datos globales.
- Una clase que modela el componente simulador.
- Una clase que modela la interfaz de usuario.

5.2.5. Integración y prueba.

Todos los elementos explicados hasta ahora se integran dentro de la aplicación **`AppManager`** cuyo aspecto y funcionamiento han sido descritos en la sección 5.1. La aplicación se encarga de:

- Crear los puertos.
- Crear los componentes y asignarles sus puertos.
- Conectar los puertos.
- Extraer las actividades de los componentes y crear una colección con todas las actividades.
- Crear un conjunto de tareas igual al número de actividades.
- Asignar actividades a tareas según la política elegida (una única tarea, actividad por tarea o asignación personalizada).
- Arrancar y parar la ejecución del caso de estudio.

5.2.6. Diagrama de clases del *framework*.

La figura 40 muestra una versión simplificada del diagrama de clases del *framework*. Algunos de los patrones más relevantes son resaltados en las clases que implementan los roles definidos por tales patrones. Otros patrones que se han empleado no pueden ser claramente identificados. Cada elemento que aparece en la figura pertenece a uno de los items CG1, CG2 o CG3 que se han descrito.

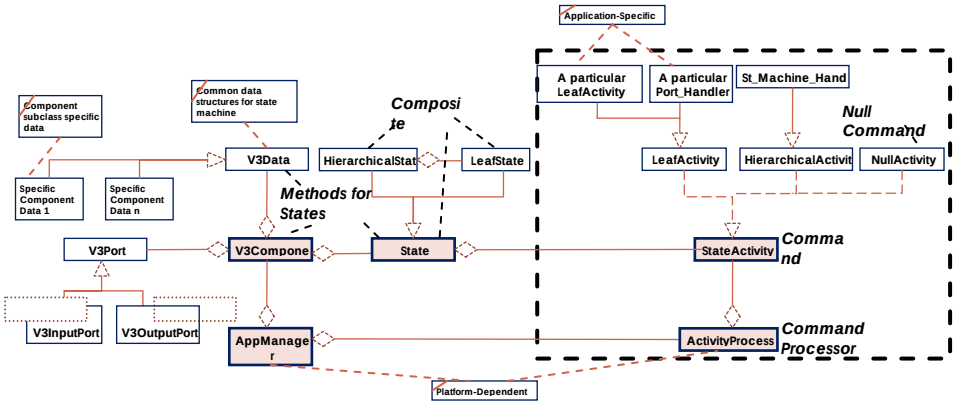


Figura 40. Diagrama de clases del *framework*.

Código específico de la plataforma (CG3). Este código integra las clases que dan soporte al entorno de ejecución. Las instancias de las clases `ActivityProcessor` y `StateActivity` constituyen el núcleo de este entorno. La clase `ActivityProcessor` es el resultado de aplicar el patrón procesador de comandos. Dicho patrón, como se ha descrito, separa las solicitudes de servicio de su ejecución gestionando dichas peticiones como objetos independientes. Así, cada tarea es modelada como un procesador de comandos y las actividades (instancias de las subclases de `StateActivity`) como comandos. Las tareas como procesadores de comandos proporcionan gran flexibilidad al diseño global ya que el patrón no impone restricciones acerca de las actividades suscritas, duración de la actividad, etc. El diseño del núcleo de ejecución proporciona un lugar donde ubicar la funcionalidad definida en V^{3}CMM, así como añadir aquellos elementos no definidos en el modelo V^{3}CMM pero que son necesarios para que la aplicación funcione (elementos referentes al manejo de puertos y gestión de la máquina de estados). Para ello se definen las clases `ParticularPortHandler` y `StMachineHandler`.

La especialización de `HieralchicalActivity` denominada `StMachineHandler`[2] gestiona las transiciones entre estados dentro de una región ortogonal. La transformación añade un objeto de esta clase por cada una de las regiones que aparecen en el modelo V^{3}CMM. La clase `ParticularPortHandler`[3] define la política de gestión de los puertos de un componente, cómo se leen y escriben los datos, la relación entre eventos y datos, etc. Como en el caso anterior, la transformación añade uno o varios objetos de esta clase por cada componente que aparece en el modelo V^{3}CMM, según la política de gestión de puertos seleccionada (comunicación síncrona, asíncrona, con respuesta, sin respuesta, etc). En caso de que todos los puertos tengan que gestionarse de forma diferente, se creará un objeto del tipo `ParticularPortHandler` por cada puerto. Si fuera posible gestionarlos todos por igual, podría crearse un solo objeto que los gestionara todos (como en el caso de estudio que se ha descrito).

Conceptos de V^{3}CMM (CG2). El código CG2 da soporte OO a los conceptos V^{3}CMM e integra las clases e interfaces denominadas `V3Component`, `V3Data`, `V3Port`, `V3InputPort`, `V3OutputPort`, `StateActivity`, `LeafState`, `State` y `HieralchicalState`. Las últimas tres clases, relacionadas por el patrón *Composite* permiten representar la estructura de los estados y su jerarquía. `V3Component`, `V3Port` y `V3Data` proporciona la implementación de los conceptos de componente, puerto y datos (internos al componente) de acuerdo a V^{3}CMM. Hay varias alternativas para modelar objetos con un

[2] Se ha añadido el hecho de ser una especialización de `HierarchicalActivity` para mantener un paralelismo con `HierarchicalState`.

[3] Asociada a la región `St_Update` y denominada `PortsHandler` en la implementación del caso de estudio.

comportamiento dependiente del estado. En este trabajo se ha adoptado el patrón Métodos para Estado, que implementa dicho comportamiento en subprogramas internos del objeto por medio de estructuras internas de datos definidas en `V3Data`[4]. Finalmente, `StateActivity` representa la interfaz de las actividades asociadas a los estados. Es el nexo de unión entre la implementación de la máquina de estados y las tareas que ejecutan su código. `V3InputPort` y `V3OutputPort` son tipos genéricos diseñados para ser especializados por la transformación con los tipos de datos enviados o recibidos entre puertos.

Código específico de la aplicación (CG1). Integra las subclases de `V3InputPort`, `V3OutputPort`, `LeafActivity`, `V3Data` y `V3Component` que tienen que ver con la funcionalidad específica de cada componente y que son generadas e integradas por la transformación. La funcionalidad asociada a cada estado del modelo V^{3}CMM de partida se implementa mediante las subclases de `LeafActivity` denominadas `ParticularLeafActivity`. El resto de subclases de `LeafActivity` se han descrito anteriormente y están relacionadas con el entorno de ejecución. Por su parte, los puertos de cada componente se obtienen especializando las clases `V3InputPort` y `V3OutputPort`. Igualmente, la clase `V3Data` es extendida por la transformación para incluir los datos específicos de los componentes definidos. Las especializaciones de la superclase `V3Component` son los componentes definidos por el usuario (por ejemplo, controladores de articulaciones, interfaces hombre-máquina, sensores, etc.). Como se ha mencionado en el item correspondiente al código CG3, la transformación añade un objeto de la clase `StMachineHandler` por cada una de las regiones que aparecen en el modelo V^{3}CMM y uno o varios objetos del tipo `ParticularPortHandler` de acuerdo con la política de gestión de puertos seleccionada.

Por último, la clase `V3Application` representa la aplicación final, en la que la transformación integra el código anterior.

5.3. CONCLUSIONES.

Para terminar, es importante recordar una vez más que el objetivo de este primer caso de estudio no era dar una solución definitiva a los problemas que se plantean, sino comenzar a razonar sobre los mismos en un estilo *dirigido por patrones*.

Los dos objetivos propuestos se han cumplido razonablemente:

[4] V3Data es una generalización de SC_Data en cuya estructura se separa la información de los estados que se genera al recorrer el modelo V^{3}CMM del resto de la información.

- Se ha proporcionado una solución para distribuir las actividades de las máquinas de estados en tareas.
- Se ha proporcionado un primer relato de patrones que trata de cumplir con las propiedades de los lenguajes de patrones.

Por otro lado:

- La aplicación principal está deficientemente estructurada y es poco robusta.

- Las soluciones responden a los patrones ya conocidos. No se han explorado otros patrones que podrían haber proporcionado mejores soluciones.

Las actividades que se han descrito se están desarrollando para dar cumplimiento a la Tarea 2 del proyecto EXPLORE: *Seleccionar un conjunto de patrones de diseño para STR que identifiquen claramente estrategias relativas a planificación, concurrencia y tolerancia a fallos.*

6. Conclusiones y trabajos futuros.

Uno de los problemas clave en el desarrollo de software para sistemas robóticos ha sido que ante nuevas aplicaciones, en numerosas ocasiones la reutilización de software está muy limitada. Esto hace que en la práctica se parta casi de cero para solucionar una y otra vez los mismos problemas. Entre las posibles causas que se pueden identificar para que se haya dado dicha limitación se pueden mencionar las siguientes:

- Normalmente los especialistas en robótica se centran más en el desarrollo de algoritmos y en el modo de resolver problemas concretos que en la organización del software.

- La falta de buenos estándares para el desarrollo de software robótico e implementaciones de los mismos.

- Tradicionalmente, los casos de estudio desarrollados para demostrar la viabilidad de las técnicas emergentes de la Ingeniería del Software se corresponden con sistemas de gestión de la información.

- Al mismo tiempo, la comunidad robótica no ha visto las técnicas de la Ingeniería del Software como una solución sino como un problema más, añadiendo complejidad a problemas de por sí complejos. Esto puede deberse a la falta de madurez en algunos conceptos de la Ingeniería del Software, y sobre todo a la falta de herramientas estandarizadas y robustas para ponerlos en práctica.

Este trabajo ha recogido nuestras experiencias de los últimos quince años en el desarrollo de aplicaciones robóticas para la industria, dentro del ámbito de la robótica de servicios, haciendo uso en cada momento de las técnicas propuestas por la Ingeniería del Software.

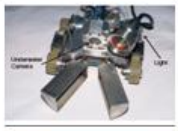
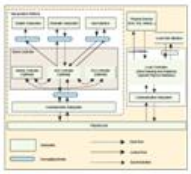
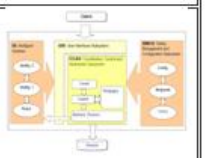
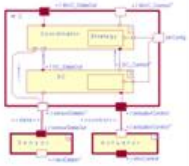
Proyecto	Dominio de Aplicación	Estrategia Ingeniería del Software / Propósitos	Dispositivos Robóticos	Características	Esquema de la propuesta
ROSA III EUREKA 1993-1999	Centrales Nucleares	**Estrategia Ingeniería del Software:** Arquitectura de Referencia **Propósitos:** • Definir una arquitectura común para todos los sistemas del dominio (teleoperación pura, entornos estructurados, herramientas especializadas). • Reutilización de código.		↑ Todos los sistemas comparten la misma estructura. ↑ Facilidad de reutilizar componentes genéricos en diferentes aplicaciones ↓ Imposibilidad de utilizar la arquitectura en otros dominios (sistemas autónomos o entornos no estructurados). ↓ Imposibilidad de reutilizar componentes en otras arquitecturas. ↓ Es difícil diseñar una arquitectura única para tratar tal heterogeneidad.	
GOYA FEDER 2000					
EFTCoR VFP 2004	Industria Reparación Astilleros	**Estrategia Ingeniería del Software:** Frameworks Arquitectónicos Desarrollo basado en componentes **Propósitos:** • Proporcionar un marco arquitectónico para definir componentes y arquitecturas. • Definir componentes abstractos y estrategias de integración • Reutilización de componentes independiente de la arquitectura.	 	↑ Incluye el enfoque de arquitectura de referencia, pero dentro de un marco más amplio. ↑ Posibilidad de definir y reutilizar arquitecturas específicas. ↑ Los componentes son independientes de la arquitectura y de la plataforma de ejecución. ↓ Los componentes no son fácilmente implementables con la tecnología orientada a objetos. ↓ Codificación manual muy trabajosa y propensa a errores.	 
EFTCoR v2 2008		**Estrategia Ingeniería del Software:** Desarrollo dirigido por modelos **Propósitos:** Arquitecturas como modelos. Modelos conformes a meta-modelos y transformaciones entre ellos.		↑ Incluye el enfoque de CBD, pero dentro de un marco más amplio. ↑ Traducción automática de componentes abstractos a componentes concretos ↑ Diseño de la arquitectura en alto nivel de abstracción independiente de plataforma y traducción semi-automática a código.	

Figura 41. Sistemas robóticos desarrollados y paradigmas de la Ingeniería del Software utilizados.

La figura 41 muestra un resumen de las ideas presentadas y algunas de las características principales de los sistemas robóticos desarrollados. El camino recorrido comienza con la ingeniería de dominio y el uso de arquitecturas de referencia, continúa con el desarrollo orientado a componentes y se centra hoy en día en el desarrollo dirigido por modelos.

Con todo ello se ha contribuido a demostrar la viabilidad de la utilización de las técnicas de dicha disciplina en aplicaciones industriales reales, si bien usando herramientas de carácter académico, tal vez difícilmente aceptables por la industria. En este sentido, hace falta que aparezcan herramientas comerciales que incorporen estas ideas, en particular CBD y MDE, dirigidas al desarrollo de aplicaciones robóticas. Desde nuestra experiencia, consideramos que la Ingeniería del Software ha contribuido decisivamente a mejorar la calidad de nuestras aplicaciones y a disminuir el esfuerzo de desarrollo.

Nuestra apuesta de futuro se centra en el uso del enfoque MDE y el desarrollo de herramientas que den soporte automatizado a la generación de código dando cumplimiento a los objetivos del proyecto EXPLORE.

Entre los retos que se nos plantean a corto plazo se pueden citar los siguientes:

- Incrementar la granularidad de la concurrencia (planificar estados hoja).
- Aplicar realmente RMA. Para ello, es necesario trasladar el caso de estudio a un sistema operativo de tiempo real y a un lenguaje de programación que proporcione más control sobre la ejecución de las tareas, bien por la vía del propio lenguaje, caso de Ada, o de las bibliotecas que lo acompañan, caso C++.
- Trasladar el caso de estudio a dispositivos reales, en especial al caso de estudio original y a los casos de estudio sobre robots móviles contemplados en el proyecto EXPLORE.
- Sería muy interesante adaptar el caso de estudio a las restricciones y recomendaciones definidas por el perfil Ada Ravenscar [Burns, 2004].
- Extender la solución para soportar la distribución de componentes.
- Revisar V^3CMM para incorporar requisitos temporales.
- Implementar las transformaciones e integrar herramientas (Eclipse, herramientas de desarrollo con V^3CMM, etc).
- Integración con herramientas de análisis (UML-MAST, MARTE, etc).

A largo plazo, el objetivo es:

- Comprobar que la solución aportada escala frente a diseños de decenas o centenas de componentes.

- Estudiar la integración de *middleware* de distribución.
- Realizar una implementación menos restrictiva de los *state-charts*. Llevar a cabo un estudio de autómatas temporizados.
- Crear catálogos de componentes y transformación de modelos e integrar V^3CMM en un enfoque de Línea de Productos Software [Northrop, 2002] con objeto de definir arquitecturas para diferentes familias de productos. Será posible obtener una arquitectura concreta para un producto seleccionando componentes (en forma de modelos V^3CMM) y transformaciones a partir de repositorios.

7. Bibliografía

[Alonso, 1997] A. Alonso, B. Álvarez, J.A. Pastor, J.A. de la Puente and A. Iborra. “Software architecture for a robot teleoperation system” *4th IFAC Workshop on Algorithms and Architectures for Real-Time Control*, Vilamoura, Portugal, 1997.

[Álvarez, 1998] B. Álvarez, A. Alonso, J.A. de la Puente. “Timing analysis of a generis robot teleoperation software architecture”. Control Engineering Practice, Vol.6, No.6, pp. 409-416, 1998.

[Álvarez, 2000] B. Álvarez, A. Iborra, P.J. Navarro, J.A. Pastor and J.M. Fdez-Meroño. Robotized system for retrieving fallen objects within the reactor vessel of a nuclear power plant (PWR). *IEEE International Symposium on Industrial Electronics*, Vol.2, pp. 529-534, Puebla, Mexico, 2000.

[Álvarez, 2001] B. Álvarez, A. Iborra, A. Alonso And J.A. de la Puente. “Reference architecture for robot teleoperacion: development details and practical use”. *Control Engineering Practice*, Vol. 9. No.4, pp. 395-402, April 2001.

[Álvarez, 2006] B. Álvarez, P. Sánchez, J.A. Pastor, and F. Ortiz, “An architectural framework for modelling teleoperated service robots”. *ROBOTICA International Journal of Information, Education and Research in Robotics and Artificial Intelligence*, Vol. 24, No. 4, pp. 411-418. Ed. Cambridge University Press. July 2006.

[Bass, 1998] L. Bass, P. Clements and R. Kazman. “Software Architecture in Practice” USA, Addison-Wesley, 1998.

[Bass, 1999] L. Bass, R. Kazman. “Architecture Based Development”, Technical Report CMU/SEI-99-TR-007, April 1999.

[Berson, 1992] A. Berson. Client/Server Architecture. New-York, USA, McGraw-Hill, 1992.

[Brugali, 2006] D. Brugali and P.Salvaneschi, “Stable aspects in robot software development”, *International Journal of Advanced Robotic Systems*, Vol. 3, no. 1, pp. 17-22, Mar 2006.

[Bruyninckx, 2001] H. Bruyninckx, “Open Robot Control Software: the OROCOS project”. *Proc. of the IEEE International Conference on Robotics and Automation*, Vol. 3, pp. 2523–2528. Ed. IEEE Computer Society. May 21-26, 2001. Seoul, Korea.

[Bruyninckx, 2008] H. Bruyninckx, “Robotics Software: The Future Should Be Open”. *IEEE Robotics & Automation Magazine,* Vol. 15, No. 1, pp. 9-11, March 2008.

[Burns, 2004] Alan Burns, Brian Dobbing and Tullio Vardanega. "Guide for the use of the Ada Ravenscar Profile in high integrity systems". ACM SIGAda Ada Letters XXIV (2): 1–74. doi:10.1145/997119.997120. June 2004 http://www.sigada.org/ada_letters/jun2004/ravenscar_article.pdf.

[Buschmann, 2007] F.R. Buschmann, D. Schmidt, M. Stal. “A pattern Language for Distributed Computing”, Wiley, 2007.

[Deursen, 2000] A. van Deursen, P. Klint and J. Visser, “Domain-specific languages: an annotated bibliography“, *SIGPLAN Not.*, Vol. 35, No. 6, pp. 26-36, June, 2000.

[Douglas, 2000] B.P. Douglas, “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.

[Drake, 2000] 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.

[Fernández, 2005] C. Fernández, A. Iborra, B. Álvarez, J. Pastor, P. Sánchez, J.M. Fernández-Meroño, N. Ortega “Ship Shape in Europe: Co-operative Robots in the Ship Repair Industry”, *IEEE Robotics and Automation Magazine*, Vol. 12, No. 3, pp. 65-77, September 2005.

[Gamma, 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design patterns: elements of reusable object oriented software”. *Ed. Addison-Wesley Professional*, January, 1995.

[Garlan, 1996] D. Garlan and M. Shaw. An introduction to software architecture. Technical Report, Software Engineering Institute, Carnegie Mellon, USA, 1996.

[Gerkey, 2001] B. Gerkey, R. Vaughan, K. Stoy, A. Howard, G. Sukhatme, and M. Mataric, “Most valuable player: a robot device server for distributed control”. *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vol. 3, pp. 1226–1231. Ed. IEEE Computer Society. 29 October - 3 November, 2001. Wailea, Hawaii.

[Gomaa, 2000] H. Gomaa, “Designing Concurrent Distributed and Real Time Applications with UML”, Addison-Wesley, ISBN 0-201-65793-7, May 2000.

[Hofmeister, 1999] C. Hofmeister, R. Nord, D. Soni, “Describing Software Architecture with UML”, Siemens Corporate Research, Princeton, New Jersey. *Proc. of the First Working IFIP Conference on Software Architecture*, Kluwer Academic Publisher, USA 1999.

[Iborra, 2003] A. Iborra, J.A. Pastor, B. Álvarez, C. Fernández and J.M. Fernández Meroño, “Robots in Radioactive Environments”, *IEEE Robotics and Automation Magazine*, Vol. 10, No. 4, pp. 12-22, December 2003.

[IFR, 2006] http://www.ifr.org/publications/World_Robotics.htm

[Jiménez, 2009] M. Jiménez, F. Rosique, P. Sánchez, B. Álvarez and A. Iborra. HAbitATION: a Domain Specific Language for Home Automation. *IEEE Software*, Vol.26, No.4, pp. 30-37, July/August 2009.

[Jouault, 2008] F. Jouault, F. Allilairea, J. Bézivin, and I. Kurtev, “ATL: A model transformation tool”, *Science of Computer Programming*, Vol. 72, No. 1-2, pp. 31-39, 2008.

[Kang, 1990] K.C. Kang and S. Cohen. “Feature-Oriented Domain Analysis (FODA) feasibility study”. Technical Report, CMU/SEI-90-TR-21, *Software Engineering Institute*, Carnegie-Mellon University, Pittsburgh, USA, November 1990.

[Karlsson, 2000] J. Karlsson, ‘‘UN world robotics statistics 1999,’’ *Ind. Robot*., vol. 27, no. 1, pp. 14–18, 2000.

[Kazman, 2000] R. Kazman, M. Klein, P. Clements. “ATAM: Method for Architecture Evaluation”, Technical Report CMU/SEI-2000-TR-004, 2000.

[Klein, 1993] M.H. Klein, T. Ralya, B. Pollack, R. Obenza and M. González-Harbour. “A Practitioner’s Handbook for Rate Monotonic Analysis”. USA, Kluwer Academics Publishers.

[Krutchen, 1995] P. Krutchen. Architectural blueprints – the “4+1” view of software architecture. *IEEE Software*, vol. 21, no. 2, april 1995.

[Macarenko, 2007] A. Makarenko, A. Brooks, and T. Kaupp. “On the benefits of making robotic software frameworks thin”, *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems* (IROS’07).

[McClain, 1991] G.R. McClain.Open systems Interconnection Handbook. Intertext Publications McGraw-Hill, 1991.

[Montemerlo, 2003] M. Montemerlo, N. Roy, and S. Thrun, “Perspectives on standardization in mobile robot programming: the Carnegie Mellon Navigation (CARMEN) Toolkit”. *Proc. of the Intelligent Robots and Systems*, Vol. 3, pp. 2436– 2441. Ed. IEEE Computer Society. October 27-31, 2003. Las Vegas, Nevada.

[Northrop, 2002] L. Northrop, “SEI’s Software Product Line Tenets”, *IEEE Software*, Vol. 19, No. 4, pp. 32-40, 2002.

[OMG, 2003] Model Driven Architecture Guide, version v1.0.1, omg/2003-06-01, June, 2003. [Online]. Available: http://www.omg.org/docs/omg/03-06-01.pdf.

[OMG, 2006] OMG, “Object Constraint Language (OCL) specification v2.0, formal/06-05-01”, May 2006. [Online]. Available: http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf.

[OMG, 2008] OMG, “MDA success stories”, June, 2008. [Online]. Available: http://www.omg.org/mda/products_success.htm.

[Ortiz, 2005] F.J. Ortiz Zaragoza, “*Arquitectura de referencia para unidades de control de robots de servicio teleoperados*”, Tesis Doctoral, Universidad Politécnica de Cartagena, 2005.

[Pastor, 1998] J.A. Pastor, B. Álvarez, A. Iborra and J.M Fernández-Meroño. An underwater teleoperated vehicle for inspection and retrieving. *1st International Symposium on Climbing and Walking Robots*, CLAWAR, Belgium,1998.

[Pastor, 2002] J.A. Pastor Franco, “*Evaluación y desarrollo incremental de una arquitectura software de referencia para sistemas de teleoperación utilizando métodos formales*”, Tesis Doctoral, Universidad Politécnica de Cartagena, 2002.

[Peterson, 1994] A.S. Peterson and J.L.Stanley. “Mapping a Domain Model and Architecture to a Generic Design”. Technical Report, CMU/SEI-94-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, 1994.

[Schlegel, 2008] C. Schlegel, “The Challenge of Real Time Robotics Behavior: An Applied Research Perspective”, *3rd Int’l Workshop on Software Development and Integration in Robotics (SDIR’08)*, Pasadena (CA), USA, 2008.

[Schmidt, 2000] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, “Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects”. Ed. Wiley, 2000.

[Schmidt, 2006] D. Schmidt, “Model-Driven Engineering”. *IEEE Computer*, Vol. 39, No. 2, pp. 25-31, February 2006.

[Shavor, 2003] S. Shavor, J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy, “The Java Developer's Guide to Eclipse”. Ed. Addison-Wesley Professional, 2003.

[SIMATIC, 2002] SIMATIC, “Working with STEP 7 5.2” ref. 6ES7810-4CA06-8BA0, www.siemens.com, 2002.

[Stahl, 2006] T. Stahl and M. Vöelter, “Model-Driven Software Development: Technology, Engineering, Management”, 1st ed. Wiley, 2006.

[Steinberg, 2008] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. “EMF: Eclipse Modeling Framework”, *2nd Ed.,* Addison-Wesley Professional, 2008.

[Szyperski, 2002] C. Szyperski, “Component Software: Beyond Object-Oriented Programming”, 2nd ed. Addison-Wesley Professional, 2002.

[Vicente, 2007] C. Vicente, F. Losilla, B. Álvarez, A. Iborra and P. Sánchez. “Applying MDE to the Development of Flexible and Reusable Wireless Sensor Networks”, *International Journal of Cooperative Information Systems*, Vol. 16, No. 3-4, pp. 393-412, December 2007.

[Withey, 1994] J.V. Withey, “Implementing model based software engineering in your organization: An approach to domain engineering” *Technical Report, CMU/SEI-94-TR-21. Software Engineering Institute*, Carnegie-Mellon University, Pittsburgh, USA. November 1994.